

Handbook of Magma Functions

Part XVI: Arithmetic 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

130 Rational Curves and Conics **1**
130.1 Introduction . . . 1
130.2 Rational Curves and Conics . . . 1
130.3 Conics . . . 6
130.4 Local-Global Correspondence . . . 7
130.5 Rational Points on Conics . . . 9
130.6 Isomorphisms . . . 12
130.7 Automorphisms . . . 15
130.8 Bibliography . . . 17

131 Elliptic Curves **19**
131.1 Introduction . . . 19
131.2 Creation Functions . . . 20
131.3 Operations on Curves . . . 27
131.4 Polynomials . . . 30
131.5 Subgroup Schemes . . . 32
131.6 The Formal Group . . . 34
131.7 Operations on Point Sets . . . 34
131.8 Morphisms . . . 36
131.9 Operations on Points . . . 41
131.10 Bibliography . . . 48

132 Elliptic Curves over Finite Fields **49**
132.1 Supersingular Curves . . . 49
132.2 The Order of the Group of Points . . . 50
132.3 Enumeration of Points . . . 55
132.4 Abelian Group Structure . . . 55
132.5 Pairings on Elliptic Curves . . . 56
132.6 Weil Descent in Characteristic Two . . . 61
132.7 Discrete Logarithms . . . 63
132.8 Bibliography . . . 64

133 Elliptic Curves over $\mathbb{Q}$ and Number Fields **65**
133.1 Introduction . . . 65
133.2 Curves over the Rationals . . . 65
133.3 Curves over Number Fields . . . 89
133.4 Curves over p-adic Fields . . . 97
133.5 Mordell–Weil Groups and Descent Methods . . . 97
133.6 Bibliography . . . 126

134 Elliptic Curves over Function Fields **128**
134.1 An Overview of Relevant Theory . . . 128
134.2 Local Computations . . . 129
134.3 Elliptic Curves of Given Conductor . . . 130
134.4 Heights . . . 130
134.5 The Torsion Subgroup . . . 131
134.6 The Mordell–Weil Group . . . 131

134.7 Two Descent . . . 133
134.8 The L-function and Counting Points . . . 134
134.9 Action of Frobenius . . . 136
134.10 Extended Examples . . . 136
134.11 Bibliography . . . 139

135 Models of Genus One Curves **140**
135.1 Introduction . . . 140
135.2 Creation of Genus One Models . . . 141
135.3 Attributes of Genus One Models . . . 143
135.4 Transformations between Genus One Models . . . 144
135.5 Equivalence of Genus One Models . . . 145
135.6 Minimisation and Reduction . . . 145
135.7 Local Solubility . . . 147
135.8 Genus One Models as Coverings . . . 147
135.9 Families of Elliptic Curves with Prescribed n-Torsion . . . 148
135.10 Invariants for Genus One Models . . . 148
135.11 Covariants and Contravariants for Genus One Models . . . 149
135.12 Examples . . . 149
135.13 Bibliography . . . 151

136 Plane Quartics **152**
136.1 Invariants . . . 152
136.2 Creation from Invariants . . . 153
136.3 Isomorphisms . . . 154
136.4 Twists . . . 156
136.5 Bibliography . . . 157

137 Hyperelliptic Curves **158**
137.1 Introduction . . . 158
137.2 Creation Functions . . . 158
137.3 Operations on Curves . . . 164
137.4 Function Field . . . 167
137.5 Points . . . 167
137.6 Jacobians . . . 170
137.7 Richelot Isogenies . . . 171
137.8 Points on the Jacobian . . . 177
137.9 Rational Points and Group Structure over Finite Fields . . . 183
137.10 Jacobians over Number Fields or $\mathbb{Q}$ . . . 188
137.11 Two-Selmer Set of a Curve . . . 205
137.12 Chabauty's Method . . . 207
137.13 Cyclic Covers of $\mathbb{P}^1$ . . . 211
137.14 Kummer Surfaces . . . 218
137.15 Points on the Kummer Surface . . . 219
137.16 Analytic Jacobians of Hyperelliptic Curves . . . 221
137.17 Invariants . . . 231
137.18 Isomorphisms and Transformations . . . 239
137.19 Reduced Automorphism Group and Reduced Isomorphism Testing . . . 248
137.20 Bibliography . . . 250

138 Hypergeometric Motives **253**
138.1 Introduction . . . 253
138.2 Functionality . . . 254
138.3 Examples . . . 260
138.4 Jacobi Motives . . . 271
138.5 Jacobi Motive Functionality . . . 272
138.6 Jacobi Motive Examples . . . 273
138.7 Bibliography . . . 277

139 L-Functions **279**
139.1 Overview . . . 279
139.2 Built-in L-series . . . 279
139.3 Computing L-values . . . 291
139.4 General L-series . . . 293
139.5 Accessing the Invariants . . . 299
139.6 Modifying the L-function . . . 301
139.7 Precision . . . 302
139.8 Verbose Printing . . . 304
139.9 Arithmetic with L-series . . . 304
139.10 Advanced Examples . . . 317
139.11 Bibliography . . . 324

References **326**

Index **333**

Chapter 130

Rational Curves and Conics

130.1 Introduction

This chapter describes the specialised categories of nonsingular plane curves of genus zero: The rational plane curves and conics. Rational curves and conics in Magma are nonsingular plane curves of degree 1 and 2, respectively. The central functionality for conics concerns the existence of points over the rationals. If a point is known to exist then a conic can be parametrised by a projective line or a rational curve. In addition, several algorithms are implemented to convert conics to standard Legendre, or diagonal, models and, for curves over $\mathbb{Q}$, to a reduced Legendre model or a minimal model. A rational curve in Magma is a linearly embedded image of the projective line to which the full machinery of algebraic plane curves can be applied. The special category of conics is called `CrvCon` and that of rational curves is `CrvRat`.

The central algorithms of this chapter deal with the classification and reduction of genus zero curves to one of the standard models to which efficient algorithms can be applied. These special types serve to classify all curves up to birational isomorphism. Since the canonical divisor K_C of a genus zero curve is of degree -2, a basis for the Riemann–Roch space of the effective divisor $-K_C$ has dimension 3 and gives an anti-canonical embedding in the projective plane. The homogeneous quadratic relations between the functions define a conic model for any genus zero curve. If the curve has a rational point then a similar construction with the divisor of this point gives a birational isomorphism with the projective line. For conics over the rationals, efficient algorithms of Simon [Sim05b] allow one to first find a point, if one exists, and then to reduce to simpler models. The existence of a point is easily determined by local conditions, and this local data is carried by the data of the ramified or bad primes of reduction; if such a point exists then the existence can be certified by a certificate. Simon's algorithm in fact parametrises the curve (by the projective line) which gives a birational isomorphism with the curve.

Not every curve of genus zero can be "trivialised" by reduction to a rational curve in this way; the obstruction to having a rational point, and therefore to being parametrised by a projective line, is measured by the primes of bad reduction and also by the automorphism group, both of which are closely associated to an isomorphism class of quaternion algebras. The final algorithms of this chapter make use of this connection to compute the automorphism group of a curve and to find isomorphisms between conics.

130.2 Rational Curves and Conics

The general tools for constructing and analysing curves are described in Chapter Algebraic Curves. We do not repeat them here, but rather give some examples in Section *Rational Curve and Conic Examples* to demonstrate those basics that the user will need. In this section we describe the main parametrisation function for rational curves and functions which enable type change from a curve of genus zero to a rational curve.

130.2.1 Rational Curve and Conic Creation

Rational curves and conics are the specialised types for nonsingular plane curves of genus zero defined by polynomials of degree 1 and 2, respectively. The condition of nonsingularity is equivalent to that of absolute irreducibility for conics, and imposes no condition on a linear equation in the plane.

`Conic(coeffs): [RngElt] -> CrvCon`

`Ambient: Sch`	`Default:`

This creates a conic curve with the given sequence of coefficients (which should have length 3 or 6). A sequence $[a, b, c]$ designates the conic $aX^2+bY^2+cZ^2$, while a sequence $[a, b, c, d, e, f]$ designates the conic $aX^2 + bY^2 + cZ^2 + dXY + eYZ + fXZ$.

The optional parameter `Ambient` may be used to give the specific ambient projective space in which to create the conic; otherwise a new ambient will be created for it to lie in.

`Conic(M): Mtrx -> CrvCon`

`Ambient: Sch`	`Default:`

This creates a conic curve associated to M, which must be a symmetric 3×3 matrix. Explicitly, the equation of the conic is $[X, Y, Z]M[X, Y, Z]^{tr}$.

The optional parameter `Ambient` may be used to give the specific ambient projective space in which to create the conic; otherwise a new ambient will be created for it to lie in.

`Conic(X, f): Prj, RngMPolElt -> CrvCon`

Returns the conic defined by the polynomial f in the projective plane X.

`IsConic(S): Sch -> BoolElt, CrvCon`

Returns `true` if and only if the scheme S is a nonsingular plane curve of degree 2, in which case it also returns a curve (of type `CrvCon`) with the same defining polynomial as S.

`RationalCurve(X, f): Prj, RngMPolElt -> CrvRat`

Returns the rational curve defined by the linear polynomial f in the projective plane X.

`IsRationalCurve(S): Sch -> BoolElt, CrvRat`

Returns `true` if and only if the scheme S is defined by a linear polynomial in some projective plane $\mathbb{P}^2$; if so, it also returns a curve of type `CrvRat` with the same defining polynomial as S in $\mathbb{P}^2$.

Example: Creating a Conic from a Curve `(ex-e93efc)`

In the following example we create a degree two curve over the rational field, then create a new curve of conic type using `IsConic`.

```
> P2<x,y,z> := ProjectivePlane(Rationals());
> C0 := Curve(P2, x^2 + 3*x*y + 2*y^2 - z^2);
> C0;
Curve over Rational Field defined by
x^2 + 3*x*y + 2*x^2 - z^2
> bool, C1 := IsConic(C0);
```

This creates a new curve C_1 in the same ambient space P_2, explicitly as a conic curve.

```
> C1;
Conic over Rational Field defined by
x^2 + 3*x*y + 2*y^2 - z^2
> AmbientSpace(C0) eq AmbientSpace(C1);
true
> DefiningIdeal(C0) eq DefiningIdeal(C1);
true
```

(continues on next page)

(continued from previous page)

```
> Type(C0), Type(C1);
CrvPln CrvCon
```

130.2.2 Access Functions

The basic access functions for rational curves and conics are inherited from the general machinery for plane curves and hypersurface schemes.

DefiningPolynomial(C): Crv -> RngMPolElt

Returns the defining polynomial of the conic or rational curve C.

DefiningIdeal(C): Crv -> RngMPol

Returns the defining ideal of the conic or rational curve C.

BaseRing(C): Crv -> Rng

BaseField(C): Crv -> Rng

Returns the base ring of the curve C.

Category(C): Crv -> Cat

Type(C): Crv -> Cat

Returns the category of rational curves `CrvRat` or of conics `CrvCon`; these are special subtypes of planes curves (type `CrvPln`), which are themselves subtypes of general curves (type `Crv`).

130.2.3 Rational Curve and Conic Examples

These examples illustrate how to obtain standard models of a curve of genus zero, either as a conic or as a parametrisation by the projective line.

Example: Singular Curve of Genus Zero (ex-626aa7)

We begin with an example of a singular curve of geometric genus zero.

```
> P2<x,y,z> := ProjectivePlane(FiniteField(71));
> C := Curve(P2, (x^3 + y^2*z)^2 - x^5*z);
> C;
Curve over GF(71) defined by
x^6 + 70*x^5*z + 2*x^3*y^2*z + y^4*z^2
> ArithmeticGenus(C);
10
> Genus(C);
0
> #RationalPoints(C);
73
> Z := SingularSubscheme(C);
> Degree(Z);
18
```

We see that C is highly singular and that its desingularisation has genus zero. At most 18 of 73 points are singular; note that a nonsingular curve of genus zero would have 72 points. We now investigate the source of the extra points.

```
> cmps := IrreducibleComponents(Z);
> [ Degree(X) : X in cmps ];
[ 11, 7 ]
> [ Degree(ReducedSubscheme(X)) : X in cmps ];
[ 1, 1 ]
> [ RationalPoints(X) : X in cmps ];
[
```

(continues on next page)

(continued from previous page)

```
{@ (0 : 0 : 1) @},
{@ (0 : 1 : 0) @}
]
```

Since the only singular rational points on C are $(0 : 0 : 1)$ and $(0 : 1 : 0)$, the "obvious" point $(1 : 0 : 1)$ must be nonsingular and we can use it to obtain a rational parametrisation of the curve as explained in Section *Isomorphisms*.

```
> P1<u,v> := ProjectiveSpace(FiniteField(71), 1);
> p := C![1, 0, 1];
> m := Parametrization(C, Place(p), Curve(P1));
> S1 := {@ m(q) : q in RationalPoints(P1) @};
> #S1;
72
> [ q : q in RationalPoints(C) | q notin S1 ];
[ (0 : 1 : 0) ]
```

We conclude that the extra point comes from a singularity whose resolution does not have any degree one places over it (see Section Divisors of Chapter Algebraic Curves for background on places of curves). We can verify this explicitly.

```
> [ Degree(p) : p in Places(C![0, 1, 0]) ];
[ 2 ]
```

Example: Testing a Curve for Rationality (ex-5e7c60)

In this example we start by defining a projective curve and we check that it is rational; that is, that it has genus zero.

```
> P2<x,y,z> := ProjectiveSpace(Rationals(), 2);
> C0 := Curve(P2, x^2 - 54321*x*y + y^2 - 97531*z^2);
> IsNonsingular(C0);
true
```

The curve C_0 is defined as a degree 2 curve over the rationals. By making a preliminary type change to the type of conics, `CrvCon`, we can test whether there exists a rational point over $\mathbb{Q}$ and use efficient algorithms of Section *Finding Points* for finding rational points on curves in conic form. The existence of a point (defined over the base field) is equivalent to the existence of a parametrisation (defined over the base field) of the curve by the projective line.

```
> bool, C1 := IsConic(C0);
> bool;
true
> C1;
Conic over Rational Field defined by
x^2 - 54321*x*y + y^2 - 97531*z^2
> HasRationalPoint(C1);
true (398469/162001 : -118246/162001 : 1)
> RationalPoint(C1);
(398469/162001 : -118246/162001 : 1)
```

The parametrisation intrinsic requires a one-dimensional ambient space as one of the arguments. This space will be used as the domain of the parametrisation map.

```
> P1<u,v> := ProjectiveSpace(Rationals(), 1);
> phi := Parametrization(C1, Curve(P1));
> phi;
```

(continues on next page)

(continued from previous page)

```
Mapping from: Prj: P1 to CrvCon: C1
with equations :
398469*u^2 + 944072*u*v + 559185*v^2
-118246*u^2 - 200850*u*v - 85289*v^2
162001*u^2 + 329499*u*v + 162991*v^2
and inverse
-4634102139*x + 30375658963*y + 31793350442*z
5456130497*x - 25641668406*y - 32136366969*z
and alternative inverse equations :
5456130497*x - 25641668406*y - 32136366969*z
-6424037904*x + 21645471041*y + 31600239062*z
```

The defining functions for the parametrisation may look large, but they are defined simply by a linear change of variables from the 2-uple embedding of the projective line in the projective plane. We now do a naive search for rational points on the curve.

```
> time RationalPoints(C1 : Bound := 100000);
{@ @}
Time: 2.420
```

Although there were no points with small coefficients, the parametrisation provides us with any number of rational points:

```
> phi(P1![0, 1]);
(559185/162991 : -85289/162991 : 1)
> phi(P1![1, 1]);
(1901726/654491 : -404385/654491 : 1)
> phi(P1![1, 0]);
(398469/162001 : -118246/162001 : 1)
```

Example: Diagonal Equations for Conics (ex-8b9072)

The first part of this example illustrates how to obtain diagonal equations for conics.

```
> P2<x,y,z> := ProjectiveSpace(RationalField(), 2);
> f := 1134*x^2 - 28523*x*y - 541003*x*z - 953*y^2 - 3347*y*z - 245*z^2;
> C := Conic(P2, f);
> LegendrePolynomial(C);
1134*x^2 - 927480838158*y^2 - 18604260593650520388494 1*z^2
> ReducedLegendrePolynomial(C);
817884337*x^2 - y^2 - 353839285266278*z^2
```

Now we demonstrate how to extend the base field of the conic; that is, to create the conic with the same coefficients but in a larger field. This can be done by calling `BaseExtend`; in this instance, we move from the rationals to a particular number field.

(Note the assignment of names to C in order to make the printing nicer.)

```
> P<t> := PolynomialRing(RationalField());
> K := NumberField(t^2 - t + 723);
> C<u,v,w> := BaseExtend(C, K);
> C;
Conic over K defined by
1134*u^2 - 28523*u*v - 953*v^2 - 541003*u*w - 3347*v*w - 245*w^2
```

130.3 Conics

In this section we discuss the creation and basic attributes of conics, particularly the standard models for them. In subsequent sections we treat the local-global theory and existence of points on conics, the efficient algorithms for finding rational points, parametrisations, and isomorphisms of genus zero curves with standard models, and finally the automorphism group of conics.

130.3.1 Elementary Invariants

`Discriminant(C): CrvCon -> FldElt`

Given a conic C, returns the discriminant of C. The discriminant of a conic with defining equation

$$a_{11}x^2 + a_{12}xy + a_{13}xz + a_{22}y^2 + a_{23}yz + a_{33}z^2 = 0$$

is defined to be the value of the degree 3 form

$$4a_{11}a_{22}a_{33} - a_{11}a_{23}^2 - a_{12}^2a_{33} + a_{12}a_{13}a_{23} - a_{13}^2a_{22}.$$

Over any ring in which 2 is invertible this is just 1/2 times the determinant of the matrix

$$\begin{pmatrix} 2a_{11} & a_{12} & a_{13} \\ a_{12} & 2a_{22} & a_{23} \\ a_{13} & a_{23} & 2a_{33} \end{pmatrix}.$$

130.3.2 Alternative Defining Polynomials

The functions described here provide access to basic information stored for a conic C. In addition to the defining polynomial, curves over the rationals compute and store a diagonalised *Legendre* model for the curve, whose defining polynomial can be accessed.

`LegendrePolynomial(C): CrvCon -> RngMPolElt, ModMatRngElt`

Returns the Legendre polynomial of the conic C, a diagonalised defining polynomial of the form $ax^2 + by^2 + cz^2$. Once computed, this polynomial is stored as an attribute. The transformation matrix defining the isomorphism from C to the Legendre model is returned as the second value.

`ReducedLegendrePolynomial(C): CrvCon -> RngMPolElt, ModMatRngElt`

Returns the reduced Legendre polynomial of the conic C, which must be defined over $\mathbb{Q}$ or $\mathbb{Z}$; that is, a diagonalised integral polynomial whose coefficients are pairwise coprime and square-free. The transformation matrix defining the isomorphism from C to this reduced Legendre model is returned as the second value.

130.3.3 Alternative Models

`LegendreModel(C): CrvCon -> CrvCon, MapIsoSch`

Returns the Legendre model of the conic C — an isomorphic curve of the form

$$ax^2 + by^2 + cz^2 = 0,$$

together with an isomorphism to this model.

`ReducedLegendreModel(C): CrvCon -> CrvCon, MapIsoSch`

Returns the reduced Legendre model of the conic C, which must be defined over $\mathbb{Q}$ or $\mathbb{Z}$; that is, a curve in the diagonal form $ax^2 + by^2 + cz^2 = 0$ whose coefficients are pairwise coprime and square-free. The isomorphism from C to this model is returned as a second value.

130.3.4 Other Functions on Conics

`MinimalModel(C): CrvCon -> CrvCon, Map`

Returns a conic, the matrix of whose defining polynomial has smaller discriminant than that of the conic C (where possible). The algorithm used is the minimisation stage of Simon's algorithm [Sim05b], as used in `HasRationalPoint`. A map from the conic to C is also returned.

Example: Minimal Model of a Conic (ex-4f68ee)

In the following example we are able to reduce the conic at 13.

```
> P2<x,y,z> := ProjectiveSpace(RationalField(), 2);
> f := 123*x^2 + 974*x*y - 417*x*z + 654*y^2 + 113*y*z - 65*z^2;
> C := Conic(P2, f);
> BadPrimes(C);
[ 491, 18869 ]
> [ x[1] : x in Factorization(Integers()!Discriminant(C)) ];
[ 13, 491, 18869 ]
> MinimalModel(C);
Conic over Rational Field defined by
-9*x^2 + 4*x*y + 6*x*z + 564*y^2 + 178*y*z + 1837*z^2
Mapping from: Conic over Rational Field defined by
-9*x^2 + 4*x*y + 6*x*z + 564*y^2 + 178*y*z + 1837*z^2 to CrvCon: C
with equations :
x + 6*y - 10*z
-x - 8*y + 4*z
-8*x - 50*y + 61*z
and inverse
-144/13*x + 67/13*y - 28/13*z
29/26*x - 19/26*y + 3/13*z
-7/13*x + 1/13*y - 1/13*z
```

130.4 Local-Global Correspondence

The Hasse–Minkowski principle for quadratic forms implies that a conic has a point over a number field if and only if it has a point over its completion at every finite and infinite prime. This provides an effective set of conditions for determining whether a conic has a point: Only the finite number of primes dividing the discriminant of the curve need to be checked, and by Hensel's lemma it is only necessary to check this condition to finite precision. The theory holds over any global field (a number field or the function field of a curve over a finite field) but the algorithms implemented at present only treat the field $\mathbb{Q}$.

130.4.1 Local Conditions for Conics

We say that p is a ramified or bad prime for a conic C if there exists no p-integral model for C with nonsingular reduction. Every such prime is a divisor of the coefficients of the Legendre polynomial. However, in general it is not possible to have a Legendre model whose coefficients are divisible only by the ramified primes. We use the term ramified or bad prime for a conic to refer to the local properties of C which are independent of the models.

`BadPrimes(C): CrvCon -> SeqEnum`

Given a conic C over the rationals, returns the sequence of the finite ramified primes of C: Those at which C has intrinsic locally singular reduction. This uses quaternion algebras.

N.B. Although the infinite prime is not included, the data of whether or not $C/\mathbb{Q}$ is ramified at infinity is carried by the length of this sequence. The number of bad primes, including infinity, must be even so the parity of the sequence length specifies the ramification information at infinity.

130.4.2 Local Solubility

By computing the Hilbert symbol at all bad primes, it is possible to determine whether a conic is *globally* soluble. This is an easier problem than finding a global solution.

`IsLocallySolvable(C): CrvCon -> BoolElt`

This returns `true` if and only if the conic has points over all completions of its coefficient field. It is implemented for conics over $\mathbb{Q}$, number fields, and function fields.

130.4.3 Norm Residue Symbol

Hilbert's norm residue symbol gives a precise condition for a quadratic form to represent zero over a local field (equivalently, for a conic to have points over that field).

An explicit treatment of the properties and theory of the norm residue symbol can be found in Cassels [Cas78]; the Hilbert symbol is treated by Lam [Lam73].

`NormResidueSymbol(a, b, p): FldRatElt, FldRatElt, RngIntElt -> RngIntElt`

`NormResidueSymbol(a, b, p): RngIntElt, RngIntElt, RngIntElt -> RngIntElt`

Given two rational numbers or integers a and b, and a prime p, returns 1 if the quadratic form $ax^2 + by^2 - z^2$ represents zero over $\mathbb{Q}_p$ and returns -1 otherwise.

`HilbertSymbol(a, b, p : parameters): FldRatElt, FldRatElt, RngIntElt -> RngIntElt`

`HilbertSymbol(a, b, p : parameters): RngIntElt, RngIntElt, RngIntElt -> RngIntElt`

`HilbertSymbol(a, b, p): FldAlgElt, FldAlgElt, RngOrdIdl -> RngIntElt`

`Al: MonStgElt` Default: `"NormResidueSymbol"`

Computes the Hilbert symbol $(a, b)_p$, where a and b are elements of a number field and p is either a prime number (if $a, b \in \mathbb{Q}$) or a prime ideal. For $a, b \in \mathbb{Q}$, by default Magma uses `NormResidueSymbol` to compute the Hilbert symbol; one may insist on instead using the same algorithm as for number fields by setting the optional argument `Al` to `"Evaluate"`.

Example: Bad Primes via Norm Residue Symbols (ex-d2227e)

In the following example we show how the norm residue symbols can be used to determine the bad primes of a conic.

```
> P2<x,y,z> := ProjectiveSpace(RationalField(), 2);
> a := 234;
> b := -33709;
> c := 127;
> C := Conic(P2, a*x^2 + b*y^2 + c*z^2);
> HasRationalPoint(C);
false
> fac := Factorization(Integers()!Discriminant(C));
> fac;
[ <2, 3>, <3, 2>, <13, 2>, <127, 1>, <2593, 1> ]
```

So we only need to test the primes 2, 3, 13, 127, and 2593. By scaling the defining polynomial of the curve $ax^2 + by^2 + cz^2 = 0$ by $-1/c$, we obtain the quadratic form $-a/cx^2 - b/cy^2 - z^2$; this means that we want to check the Hilbert symbols for the coefficient pair $(-a/c, -b/c)$.

```
> [ NormResidueSymbol(-a/c, -b/c, p[1]) : p in fac ];
[ -1, 1, 1, -1, 1 ];
```

The norm residue symbol indicates that only 2 and 127 have no local p-adic solutions, which confirms the bad primes as reported by Magma:

```
> BadPrimes(C);
[ 2, 127 ]
```

130.5 Rational Points on Conics

This section contains functions for deciding solubility and finding points on conics over the following base fields: the rationals, finite fields, number fields, and rational function fields in odd characteristic.

A point on a conic C is given as a point in the pointset $C(K)$ where K is the base ring of the conic, unless that base ring is the integers in which case the returned point belongs to the pointset $C(\mathbb{Q})$.

Over the rationals (or integers), the algorithm used to find rational points is due to D. Simon [Sim05b]. Simon's algorithm works with the symmetric matrix of the defining polynomial of the conic. It computes transformations reducing the determinant of the matrix by each of the primes which divide the discriminant of the conic until it has absolute value 1. After this it does an indefinite LLL which reduces the matrix to a unimodular integral diagonal matrix, which is equivalent to the conic $x^2 + y^2 - z^2 = 0$, and then the Pythagorean parametrisation of this can be pulled back to the original conic.

The existence of a point on a conic $C/\mathbb{Q}$ is determined entirely by local solubility conditions, which are encapsulated in a solubility certificate. The algorithm of Simon works prime-by-prime, determining local solubility as it goes through its calculation of square roots modulo primes. In theory the existence of a point can be determined using Legendre symbols instead of computing these square roots. This is not implemented because the running time is dominated by the time needed to factorise the discriminant.

Over number fields the algorithm for finding points is as follows. One first reduces to diagonal form, which is done with care to ensure that one can factor the coefficients (assuming that one can factor the discriminant of the original conic). The algorithm for diagonal conics is a variant of Lagrange's method (which was once the standard method for solving diagonal conics over $\mathbb{Q}$). It involves a series of reduction steps in each of which the conic is replaced by a simpler conic. Reduction is achieved by finding a short vector in a suitable lattice sitting inside two copies of the base field. (This lattice is defined by congruence conditions arising from local solutions of the conic.)

After several reduction steps one is often able to find a solution via an easy search. In other cases, one is unable to reduce further and must call `NormEquation` to solve the reduced conic. (This is still vastly superior to calling `NormEquation` on the original conic.) The basic reduction loop is enhanced with several tricks; in particular, when it is not too large the class group of the base field may be used to reduce much further than would otherwise be feasible.

Over rational function fields the algorithm for solving conics is due to Cremona and van Hoeij [CR06]. The implementation was contributed by John Cremona and David Roberts.

Over finite fields the algorithm used for finding a point $(x : y : 1)$ on a conic is to solve for y at random values of x.

130.5.1 Finding Points

The main function is `HasRationalPoint`, which also returns a point when one exists. This (and also `RationalPoint`) works over various kinds of fields; the other functions work only over the rationals (or integers) and finite fields.

When a point is found it is also cached for later use.

`HasRationalPoint(C): CrvCon -> BoolElt, Pt`

The conic C should be defined over the integers, rationals, a finite field, a number field, or a rational function field over a finite field of odd characteristic. The function returns `true` if and only if there exists a point on the conic C, and if so also returns one such point.

`RationalPoint(C): CrvCon -> Pt`

The conic C should be defined over the integers, rationals, a finite field, a number field, or a rational function field over a finite field of odd characteristic. If there exists a rational point on C over its base ring then a representative such point is returned; otherwise an error results.

`Random(C : parameters): CrvCon -> Pt`

```
Bound : RngIntElt                    Default: 10^9
Reduce: BoolElt                      Default: false
```

Returns a randomly selected rational point of the conic C. Such a solution is obtained by choosing random integers up to the bound specified by the parameter `Bound`, followed by evaluation at a parametrisation of C, then by point reduction if the parameter `Reduce` is set to `true`. (See Section *Point Reduction* for more information about point reduction.)

`Points(C : parameters): CrvCon -> SetIndx`
`RationalPoints(C : parameters): CrvCon -> SetIndx`

Bound: RngIntElt	Default:

Given a conic over the rationals or a finite field, returns an indexed set of the rational points of the conic. If the curve is defined over the rationals then a positive value for the parameter `Bound` *must* be given; this function then returns those points whose integral coordinates, on the reduced Legendre model, are bounded by the value of `Bound`.

Example: Points on Three Legendre Conics (ex-979e60)

We define three conics in Legendre form, having the same prime divisors in their discriminants, which differ only by a sign change (twisting by $\sqrt{-1}$) of one of the coefficients.

```
> P2<x,y,z> := ProjectiveSpace(Rationals(), 2);
> C1 := Conic(P2, 23*x^2 + 19*y^2 - 71*z^2);
> RationalPoints(C1 : Bound := 32);
{@ @}
> C2 := Conic(P2, 23*x^2 - 19*y^2 + 71*z^2);
> RationalPoints(C2 : Bound := 32);
{@ @}
> C3 := Conic(P2, 23*x^2 - 17*y^2 - 71*z^2);
> RationalPoints(C3 : Bound := 32);
{@ (-31 : 36 : 1), (-31 : -36 : 1), (31 : 36 : 1), (31 : -36 : 1),
(-8/3 : 7/3 : 1), (-8/3 : -7/3 : 1), (8/3 : 7/3 : 1), (8/3 : -7/3 : 1),
(-28/15 : 11/15 : 1), (-28/15 : -11/15 : 1), (28/15 : 11/15 : 1),
(28/15 : -11/15 : 1) @}
```

This naive search yields no points on the first two curves and numerous points on the third one. A guess that there are no points on either of the first two curves is proved by a call to `BadPrimes`, which finds that 19 and 23 are ramified primes for the first curve and 23 and 71 are ramified primes for the second.

```
> BadPrimes(C1);
[ 19, 23 ]
> BadPrimes(C2);
[ 23, 71 ]
> BadPrimes(C3);
[]
```

The fact that there are no ramified primes for the third curve is equivalent to a `true` return value from the function `HasRationalPoint`. As we will see in Section *Isomorphisms*, an alternative approach which is guaranteed to find points on the third curve would be to construct a rational parametrisation.

130.5.2 Point Reduction

If a conic $C/\mathbb{Q}$ in Legendre form $ax^2 + by^2 + cz^2 = 0$ has a point, then Holzer's theorem tells us that there exists a point $(x : y : z)$ satisfying

$$|x| \leq \sqrt{|bc|}, \quad |y| \leq \sqrt{|ac|}, \quad |z| \leq \sqrt{|ab|},$$

with x, y, and z in $\mathbb{Z}$, or equivalently $\max(|ax^2|, |by^2|, |cz^2|) \leq |abc|$. Such a point is said to be Holzer-reduced. There exist constructive algorithms to find a Holzer-reduced point from a given one; the current Magma implementation uses a variant of Mordell's reduction due to Cremona [CR03].

`IsReduced(p): Pt -> BoolElt`

Returns `true` if and only if the projective point p on a conic in reduced Legendre form satisfies Holzer's bounds. If the curve is not a reduced Legendre model then the test is done after passing to this model.

`Reduction(p): Pt -> Pt`

Returns a Holzer-reduced point derived from p.

Example: Reduced Points on a Conic (ex-3fc80b)

We provide a tiny example to illustrate reduction and reduction testing on conics.

```
> P2<x,y,z> := ProjectiveSpace(Rationals(), 2);
> C1 := Conic(P2, x^2 + 3*x*y + 2*y^2 - 2*z^2);
> p := C1![0, 1, 1];
> IsReduced(p);
false
```

The fact that this point is not reduced is due to the size of the coefficients on the reduced Legendre model.

```
> C0, m := ReducedLegendreModel(C1);
> C0;
Conic over Rational Field defined by
X^2 - Y^2 - 2*Z^2
> m(p);
(3/2 : 1/2 : 1)
> IsReduced(m(p));
false
> Reduction(m(p));
(-1 : 1 : 0)
> Reduction(m(p)) @@ m;
(-2 : 1 : 0)
> IsReduced($1);
true
```

Example: Finding Points on a Conic (ex-b34839)

In this example we illustrate the intrinsics which apply to finding points on conics.

```
> P2<x,y,z> := ProjectiveSpace(RationalField(), 2);
> f := 9220*x^2 + 97821*x*y + 498122*y^2 + 8007887*y*z - 3773857*z^2;
> C := Conic(P2, f);
```

We will now see that C has a reduced solution; indeed, we find two different reduced solutions. For comparison we also find a non-reduced solution.

```
> HasRationalPoint(C);
true (157010/741 : -19213/741 : 1)
> p := RationalPoint(C);
> p;
(157010/741 : -19213/741 : 1)
> IsReduced(p);
true
> q := Random(C : Reduce := true);
> q;
(-221060094911776/6522127367379 : -49731359955013/6522127367379 : 1)
> IsReduced(q);
true
```

(continues on next page)

(continued from previous page)

```
> q := Random(C : Bound := 10^5);
> q;
(4457174194952129/84200926607090 : -1038901067062816/42100463303545 : 1)
> IsReduced(q);
false
> Reduction(q);
(-221060094911776/6522127367379 : -49731359955013/6522127367379 : 1)
> IsReduced($1);
true
```

To make a parametrisation of C one can use the intrinsic `Parametrization` to create a map of schemes from a line to C.

```
> phi := Parametrization(C);
> P1<u,v> := Domain(phi);
> q0 := P1![0, 1];  q1 := P1![1, 1];  q2 := P1![1, 0];
> phi(q0);
(7946776/559407 : -21332771/1118814 : 1)
> phi(q1);
(26443817/1154900 : -5909829/288725 : 1)
> phi(q2);
(157010/741 : -19213/741 : 1)
> C1, psi := ReducedLegendreModel(C);
> psi(phi(q0));
(-1793715893111/4475256 : -22556441171843029/4475256 : 1)
> p0 := Reduction($1);
> p0;
(1015829527/2964 : -59688728328467/2964 : 1)
> IsReduced(p0);
true
> p0 @@ psi;
(157010/741 : -19213/741 : 1)
```

130.6 Isomorphisms

130.6.1 Isomorphisms with Standard Models

In this section we discuss isomorphisms between heterogeneous types — isomorphisms between combinations of curves of type `Crv`, `CrvCon`, and `CrvRat`, and their parametrisations by a projective line.

The first function which we treat here is `Conic`, which is as much a constructor as an isomorphism. It takes an arbitrary genus zero curve C and uses the anti-canonical divisor $-K_C$ of degree 2 to construct the Riemann–Roch space. For a genus zero curve this is a dimension 3 space of degree 2 functions and gives a projective embedding of C in $\mathbb{P}^2$ as a conic. This provides the starting point to make any genus zero curve amenable to the powerful machinery for point finding and isomorphism classification of conics.

On the other hand, if C is an arbitrary genus 0 curve with a known non-singular point or place p, then a parametrization of C can be constructed directly from the degree 1 Riemann–Roch space. There are intrinsics to do this.

An isomorphism — provided that one exists — of the projective line with a conic can be described as follows. The 2-uple embedding $\phi : \mathbb{P}^1 \to \mathbb{P}^2$ defined by $(u : v) \mapsto (u^2 : uv : v^2)$ gives an isomorphism of $\mathbb{P}^1$ with the conic C_0 with defining equation $y^2 = xz$. The inverse isomorphism $C_0 \to \mathbb{P}^1$ is defined by the maps

$$(x : y : z) \mapsto (x : y) \text{ on } x \neq 0,$$
$$(x : y : z) \mapsto (y : z) \text{ on } z \neq 0,$$

respectively. Since these open sets cover the conic C_0 this defines an isomorphism and not just a birational map. In order to describe an isomorphism of a conic C_1 with $\mathbb{P}^1$ it is then necessary and sufficient to give a change of variables which maps $C_0 = \phi(\mathbb{P}^1)$ onto the conic C_1. This matrix is called the *parametrisation matrix* and is stored with C_1 once a rational point is found.

Given a parametrization over the rationals of an arbitrary genus 0 curve in ordinary projective space, a simpler parametrization can often be found using the intrinsic `ImproveParametrization(p): MapSch → MapSch` from the `Scheme` chapter of the handbook.

`Conic(C): Crv -> MapSch`

Given a curve of genus zero, returns a conic determined by the anti-canonical embedding of C.

Example: Conic Model of a Rational Curve (ex-564e2f)

We demonstrate the function `Conic` on the curve of Example *Singular Curve of Genus Zero* to find a conic model, even though we know that it admits a rational parametrisation.

```
> P2<x,y,z> := ProjectivePlane(FiniteField(71));
> C0 := Curve(P2, (x^3 + y^2*z)^2 - x^5*z);
> C1, m := Conic(C0);
> C1;
Conic over GF(71) defined by
x^2 + 70*x*z + y^2
> m : Minimal;
(x : y : z) -> (x^3*y*z : 70*x^5 + x^4*z + 70*x^2*y^2*z : x^3*y*z + y^3*z^2)
```

`ParametrizationMatrix(C): CrvCon -> ModMatRngElt`

This function is an optimised routine for parametrising a conic C defined over $\mathbb{Z}$ or $\mathbb{Q}$. It returns a 3×3 matrix M which defines a parametrisation of C as a projective change of variables from the 2-uple embedding of a projective line in the projective plane; i.e., for a point $(x_0 : y_0 : z_0)$ on C, the point

$$(x_1 : y_1 : z_1) = (x_0 : y_0 : z_0)M$$

satisfies the equation $y_1^2 = x_1 z_1$. Note that as usual in Magma the action of M is on the right and, consistently, the action of scheme maps is also on the right.

Example: Parametrisation Matrix of a Conic (ex-c3a025)

In this example we demonstrate that the parametrisation matrix determines the precise change of variables to transform the conic equation into the equation $y^2 = xz$. We begin with a singular plane curve C_0 of genus zero and construct a nonsingular conic model in the plane.

```
> P2<x,y,z> := ProjectiveSpace(Rationals(), 2);
> C0 := Curve(P2, (x^3 + y^2*z)^2 - x^5*z);
> C1, m := Conic(C0);
> C1;
Conic over Rational Field defined by
x^2 - x*z + y^2
```

The curve C_1 has obvious points, such as $(1 : 0 : 1)$, which Magma internally verifies without requiring an explicit user call to `HasRationalPoint`.

```
> ParametrizationMatrix(C1);
[1 0 1]
[0 1 0]
[0 0 1]
> Evaluate(DefiningPolynomial(C1), [x, y, x+z]);
-x*z + y^2
```

We note (as is standard in Magma) that the action of matrices, as with maps of schemes, is a right action on coordinates $(x : y : z)$.

`Parametrization(C): CrvCon -> MapSch`

`Parametrization(C, P): CrvCon, Crv -> MapSch`

`Parametrization(C, p): Crv, Pt -> MapSch`

`Parametrization(C, p): Crv, PlcCrvElt -> MapSch`

`Parametrization(C, p, P): CrvCon, Pt, Crv -> MapSch`

`Parametrization(C, p, P): CrvRat, Pt, Crv -> MapSch`

`Parametrization(C, p, P): CrvCon, PlcCrvElt, Crv -> MapSch`

`Parametrization(C, p, P): CrvRat, PlcCrvElt, Crv -> MapSch`

Given a conic curve C over a general field, these functions return a parametrisation as an isomorphism of schemes $P \to C$. Here P is a copy of a projective line; it may be specified as one of the arguments or a new projective line will be created. Note that it is now required that P is given (or created) as a curve rather than as an ambient space (as used to be permitted). This allows the immediate use of pullback/push-forward functionality for the parametrisation map.

When a rational point or place is not specified as one of the arguments then the base field of C must be one of the kinds allowed in `HasRationalPoint`. If the conic has no rational points then an error results.

`ParametrizeOrdinaryCurve(C): Crv -> MapSch`

`ParametrizeOrdinaryCurve(C, p): Crv, Pt -> MapSch`

`ParametrizeOrdinaryCurve(C, p): Crv, PlcCrvElt -> MapSch`

`ParametrizeOrdinaryCurve(C, p, I): Crv, Pt, RngMPol -> MapSch`

`ParametrizeOrdinaryCurve(C, p, I): Crv, PlcCrvElt, RngMPol -> MapSch`

`ParametrizeRationalNormalCurve(C): Crv -> MapSch`

These functions are as above (see `Parametrization`), but use different algorithms.

When C is a plane curve with only ordinary singularities (see subsection Ordinary Plane Curves) then a slightly different procedure is followed that relies less on the general function field machinery and tends to be faster and can produce nicer parametrisations. The variants `ParametrizeOrdinaryCurve` allow direct calls to these more specialised procedures. The I argument is the adjoint ideal of C (*loc. cit.*), which may be passed in if already computed.

The final function listed is slightly different; it applies only to rational normal curves. i.e., nonsingular rational curves of degree d in ordinary d-dimensional projective space for $d \geq 1$. For the sake of speed the irreducibility of C is not checked. The function uses adjoint maps to find either a line or conic parametrisation of C: If d is odd then an isomorphism from the projective line to C is returned, and if d is even then an isomorphism from a plane conic is returned. The method uses no function field machinery and can be much faster than the general function.

Example: Parametrising a Rational Curve (ex-270a43)

In this example we show how to parametrise a projective rational curve with a map from the one-dimensional projective space. First we construct a singular plane curve and verify that it has geometric genus zero.

```
> k := FiniteField(101);
> P2<x,y,z> := ProjectiveSpace(k, 2);
> f := x^7 + 3*x^3*y^2*z^2 + 5*y^4*z^3;
> C := Curve(P2, f);
> Genus(C);
0
```

In order to parametrise the curve C we need to find a nonsingular point on it, or at least a point of C over which there exists a unique degree one place; geometrically, such a point is one at which C has a cusp. To find such a point we invoke the intrinsic `RationalPoints` on C; since C is defined over a finite field this call returns an indexed set of all the points of C that are rational over its base field.

Having done the previous in the background, we demonstrate that the particular point $(2 : 33 : 1)$ is such a nonsingular rational point.

```
> p := C![2,33,1];
> p;
(2 : 33 : 1)
> IsNonsingular(C, p);
true
```

The parametrisation function takes a projective line as the third argument; this will be used as the domain of the parametrisation map.

```
> P1<u,v> := ProjectiveSpace(k, 1);
> phi := Parametrization(C, Place(p), Curve(P1));
> phi;
Mapping from: Prj: P1 to Prj: P2
with equations :
2*u^7 + 5*u^6*v + 81*u^5*v^2 + 80*u^4*v^3 + 13*u^3*v^4
33*u^7 + 88*u^6*v + 90*u^5*v^2 + 73*u^4*v^3 + 25*u^3*v^4 +
    83*u^2*v^5 + 72*u*v^6 + 24*v^7
u^7
```

Finally we confirm that the map really does parametrise the curve C. Note that the map is normalised so that the point at infinity on the projective line P_1 maps to the prescribed point p.

```
> Image(phi);
Scheme over GF(101) defined by
x^7 + 3*x^3*y^2*z^2 + 5*y^4*z^3
> DefiningIdeal(Image(phi)) eq DefiningIdeal(C);
true
> phi(P1![1, 0]);
(2 : 33 : 1)
```

130.7 Automorphisms

130.7.1 Automorphisms of Rational Curves

Automorphisms of the projective line $\mathbb{P}^1$ are well-known to be 3-transitive — for any three distinct points p_0, p_1, and p_∞, there exists an automorphism taking $0 = (0 : 1)$, $1 = (1 : 1)$, and $\infty = (1 : 0)$ to p_0, p_1, and p_∞. Conversely, the images of 0, 1, and ∞ uniquely characterise an automorphism. We use this characterisation of isomorphisms to define automorphisms of rational curves as bijections of three element sequences of points over the base ring.

`Automorphism(C, S, T): CrvRat, SetIndx, SetIndx -> MapIsoSch`

Given a rational curve C and two indexed sets $S = \{p_0, p_1, p_\infty\}$ and $T = \{q_0, q_1, q_\infty\}$, each of distinct points over the base ring of C, returns the unique automorphism of C taking p_0, p_1, p_∞ to q_0, q_1, q_∞.

130.7.2 Automorphisms of Conics

The automorphism group of a conic, and indeed the conics themselves, have a special relationship with quaternion algebras (see Lam [Lam73] or Vignéras [Vig80]). For the sake of exposition we focus on conics C/K defined by a Legendre equation

$$ax^2 + by^2 + cz^2 = 0$$

where $abc \neq 0$. We define a quaternion algebra A over K with anticommuting generators i, j, and $k = c^{-1}ij$ satisfying relations

$$i^2 = -bc, \quad j^2 = -ac, \quad k^2 = -ab,$$

and setting $I = ai$, $J = bj$, and $K = ck$. Then for any extension L/K we may identify the ambient projective pointset $\mathbb{P}^2(L)$ with the projectivisation of the trace zero part

$$A_L^0 = \{\alpha \in A_L \mid \mathrm{Tr}(\alpha) = 0\} = LI + LJ + LK$$

of the quaternion algebra $A_L = A \otimes_K L$ via $(x : y : z) \mapsto xI + yJ + zK$. Under this map, the pointsets $C(L)$ are identified with the norm zero elements

$$\mathrm{N}(xI + yJ + zK) = abc(ax^2 + by^2 + cz^2) = 0$$

in $\mathbb{P}^2(L)$. This allows us to identify the automorphism group $\mathrm{Aut}_K(C)$ with the quotient unit group A^*/K^* acting on each A_L^0 by conjugation.

In the Magma implementation of this correspondence, the quaternion algebra A is computed and stored as an attribute of the curve C. Automorphisms of C can be created from any invertible element of the quaternion algebra. We note that the isomorphism of two conics is equivalent to the isomorphism of their quaternion algebras, and that a rational parametrisation of a conic is equivalent to an isomorphism $A \cong < Meta > -_2(K)$.

We note that while the advanced features of finding rational points (and hence finding parametrisations) and determining isomorphism only exist over $\mathbb{Q}$, it is possible to represent the automorphism group and find automorphisms of the curve independently of these other problems. The current implementation works only in characteristic different from 2, as otherwise a Legendre model does not exist.

`QuaternionAlgebra(C): CrvCon -> AlgQuat`

Returns the quaternion algebra in which automorphisms of the conic C can be represented.

`Automorphism(C, a): CrvCon, AlgQuatElt -> MapIsoSch`

Given a conic C and a unit a of the quaternion algebra associated to C, returns the automorphism of C corresponding to a.

Example: Automorphisms of a Conic Without Points `(ex-85304b)`

In this example we demonstrate how to use the quaternion algebra of a conic to construct nontrivial automorphisms of the curve when no rational points exist over the base ring. (We use the printing level `Minimal` to produce human readable output for scheme maps.)

```
> P2<x,y,z> := ProjectiveSpace(Rationals(), 2);
> C0 := Conic(P2, 2*x^2 + x*y + 5*y^2 - 37*z^2);
> HasRationalPoint(C0);
false
> BadPrimes(C0);
[ 3, 37 ]
> A := QuaternionAlgebra(C0);
> RamifiedPrimes(A);
[ 3, 37 ]
> B := Basis(MaximalOrder(A));
> B;
[ 1, 1/52*i + 1/4*j + 2/481*k, j, 1/2 + 1/148*k ]
> m1 := Automorphism(C0, A!B[2]);
> m1 : Minimal;
(x : y : z) -> (-5/8*x + 259/48*y - 37/3*z : 37/12*x + 69/8*y - 74/3*z :
    x + 7/2*y - 61/6*z)
> m2 := Automorphism(C0, A!B[3]);
> m2 : Minimal;
(x : y : z) -> (x + 1/2*y : -y : z)
> m3 := Automorphism(C0, A!B[4]);
> m3 : Minimal;
(x : y : z) -> (-x - 1/2*y : 1/5*x - 9/10*y : z)
> [ Trace(B[2]), Trace(B[3]), Trace(B[4]) ];
```

(continues on next page)

(continued from previous page)

```
[ 0, 0, 1 ]
> m1 * m1 : Minimal;
(x : y : z) -> (x : y : z)
> m2 * m2 : Minimal;
(x : y : z) -> (x : y : z)
> m3 * m3 : Minimal;
(x : y : z) -> (9/10*x + 19/20*y : -19/50*x + 71/100*y : z)
```

The last example points out that *pure* quaternions τ in A_0 give rise to involutions — a reflection of the projective plane about the point defined by τ in $\mathbb{P}^2(K)$.

One of the strengths of the scheme model in Magma is the ability to work with points of a curve over any extension. Provided that we have a rational point over some extension field L/K, we are able to apply automorphisms of the curve to generate an unbounded number of new points over that extension by means of the action of the automorphism group. We demonstrate this by looking at quadratic twists of the curve to find a point over $\mathbb{Q}$, which identifies a point on the original curve over a quadratic extension.

```
> C1 := Conic(P2, 2*x^2 + x*y + 5*y^2 - z^2);
> HasRationalPoint(C1);
false
> C2 := Conic(P2, 2*x^2 + x*y + 5*y^2 - 2*z^2);
> HasRationalPoint(C2);
true
> RationalPoint(C2);
(-1 : 0 : 1)
```

This gives rise to the obvious points $(\pm 1 : 0 : \sqrt{74}/37)$ on the original curve.

```
> P<t> := PolynomialRing(RationalField());
> L<a> := NumberField(t^2 - 74);
> p := C0(L)![1, 0, a/37];
> m1(p);
(1/1880*(207*a + 3034) : 1/940*(-37*a + 2146) : 1)
> m2(p);
(1/2*a : 0 : 1)
> m3(p);
(-1/2*a : 1/10*a : 1)
```

We could alternatively have formed the base extension of the curve to the new field L and used the known point over L to find a parametrisation of the curve by the projective line. This approach demonstrates that it is possible to work with points and curve automorphisms without passing to a new curve.

130.8 Bibliography

- [Cas78] J. W. S. Cassels. *Rational Quadratic Forms.* Academic Press, London–New York–San Francisco, 1978.
- [CR03] J. E. Cremona and D. Rusin. Efficient solution of rational conics. *Mathematics of Computation*, 72(243):1417–1441, 2003.
- [CR06] J. E. Cremona and D. Rusin. Solving conics over function fields. *Journal de Theorie des Nombres de Bordeaux*, 18:595–606, 2006.
- [Lam73] T. Y. Lam. *The Algebraic Theory of Quadratic Forms.* W. A. Benjamin, Inc., Reading, MA, 1973.

- [Sim05b] Denis Simon. Solving quadratic equations using reduced unimodular quadratic forms. *Math. Comp.*, 74(251):1531–1543 (electronic), 2005.
- [Vig80] M.-F. Vignéras. *Arithmétique des Algèbres de Quaternions*. Volume 800 of Lecture Notes in Mathematics. Springer-Verlag, Berlin, 1980.

Chapter 131

Elliptic Curves

131.1 Introduction

This chapter describes features for working with elliptic curves in Magma. It contains basic functionality that is applicable to curves over fairly general fields. There are separate chapters describing features that are specific to

- curves over finite fields (Chapter *Elliptic Curves over Finite Fields*),
- curves over the rationals or number fields (Chapter *Elliptic Curves over {\mathbb{Q}} and Number Fields*),
- curves over univariate function fields (Chapter *Elliptic Curves over Function Fields*).

An elliptic curve E is the projective closure of the curve given by the generalised Weierstrass equation

$$y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6.$$

The curve is specified by the sequence of coefficients $[a_1, a_2, a_3, a_4, a_6]$; the two element sequence $[a_4, a_6]$ may be used instead when $a_1 = a_2 = a_3 = 0$.

Elliptic curve functionality covers both elementary invariants of curves and arithmetic in the group of rational points, as well as higher level features for computing local invariants, heights, and Mordell–Weil groups for curves over $\mathbb{Q}$, and for point counting and the determination of the group structure over $\mathbb{F}_q$. The base ring of elliptic curves is currently restricted to fields. Curves over the rationals have special features to allow the construction of integral and minimal models, and for base change to finite fields, in acknowledgement of the integral structure over $\mathbb{Z}$ or $\mathbb{Z}_p$.

For curves over the rationals or over number fields there are routines for determining minimal models and an implementation of Tate's algorithm for determining Kodaira symbols and various local invariants. Algorithms for the computation of the Mordell–Weil group are heavily based on publications of John Cremona; see [Cre97] for details. There are also separate implementations of 2-descent for curves over number fields, and 3- and 4-descent for curves over the rationals. Additionally, several aspects of the analytic theory (including modular parametrisations and Heegner points) are implemented for curves over the rationals.

Elliptic curves are specialised forms of the more general curve and scheme types, and as such all functions which apply to these general types work on elliptic curves (although a few of them behave differently for elliptic curves). Some of these functions are described here, but not all of them — refer to chapters Schemes (Schemes) and Algebraic Curves (Curves) for descriptions of these functions, as well as an explanation of the relationships between points, point sets, and schemes. In particular, note that the parent of a point is a point set, and *not* the curve.

The name of the category of elliptic curves is `CrvEll`, with points of type `PtEll` lying in point sets of type `SetPtEll`. There is also the category `SchGrpEll` for subgroup schemes of elliptic curves, and a special category `SymKod` exists for the datatype of Kodaira symbols, which classify the local structure of the special fibre at p of the Néron model of an elliptic curve $E/\mathbb{Q}$.

This chapter, the first of four on elliptic curves, contains a treatment of the basics for curves over general fields: their construction, their arithmetic, and their basic properties. Specialised machinery provided for elliptic curves over finite fields is described in Chapter *Elliptic Curves over Finite Fields*; Chapter *Elliptic Curves over {\mathbb{Q}} and Number Fields* presents the wide range of techniques available for determining information about the group of rational points for curves over $\mathbb{Q}$ and over number fields, while elliptic curves over function fields are discussed in Chapter *Elliptic Curves over Function Fields*.

131.2 Creation Functions

131.2.1 Creation of an Elliptic Curve

An elliptic curve E may be created by specifying Weierstrass coordinates for the curve over a field K, where integer coordinates are interpreted as elements of $\mathbb{Q}$. Note that the coordinate ring K defines the base point set of E, and points defined over an extension field of K must be created in the appropriate point set.

`EllipticCurve([a, b]): [ RngElt ] -> CrvEll`

`EllipticCurve([a1, a2, a3, a4, a6]): [ RngElt ] -> CrvEll`

Given a sequence of elements of a ring K, this function creates the elliptic curve E over K defined by taking the elements of the sequence as Weierstrass coefficients. The length of the sequence must either be two, that is $s = [a, b]$, in which case E is defined by $y^2 = x^3 + ax + b$, or five, that is $s = [a_1, a_2, a_3, a_4, a_6]$, in which case E is given by $y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$. Currently K must be field; if integers are given then they will be coerced into $\mathbb{Q}$. The given coefficients must define a nonsingular curve; that is, the discriminant of the curve must be nonzero.

`EllipticCurve(f): RngUPolElt -> CrvEll`

`EllipticCurve(f, h): RngUPolElt, RngUPolElt -> CrvEll`

Given univariate polynomials f and h, this function creates the elliptic curve defined by $y^2 + h(x)y = f(x)$, or by $y^2 = f(x)$ if h is not given. The polynomial f must be monic of degree 3 and h must have degree at most 1.

`EllipticCurveFromjInvariant(j): RngElt -> CrvEll`

`EllipticCurveWithjInvariant(j): RngElt -> CrvEll`

Given a ring element j, this function creates an elliptic curve E over K with j-invariant equal to j defined as follows: If $j = 0$ and K does not have characteristic 3 then E is defined by $y^2 + y = x^3$; if $j = 1728$ then E is defined by $y^2 = x^3 + x$ (which covers the case where $j = 0$ and K has characteristic 3); otherwise $j - 1728$ is invertible in K and E is defined by $y^2 + xy = x^3 - (36/(j - 1728))x - 1/(j - 1728)$.

Example: Three Constructions of One Curve (ex-bf37da)

We create one particular elliptic curve over $\mathbb{Q}$ in three different ways:

```
> EllipticCurve([1, 0]);
Elliptic Curve defined by y^2 = x^3 + x over Rational Field
> Qx<x> := PolynomialRing(Rationals());
> EllipticCurve(x^3 + x);
Elliptic Curve defined by y^2 = x^3 + x over Rational Field
> EllipticCurveWithjInvariant(1728);
Elliptic Curve defined by y^2 = x^3 + x over Rational Field
```

`EllipticCurve(C): Sch -> CrvEll, MapSch`

```
EllModel: Verbose                    Default: 3
```

Given a scheme C describing a curve of genus 1 with an easily recognised rational point, this function returns an elliptic curve E together with a birational map from C to E. If there is no "obvious" rational point then this routine will fail. C must belong to one of the following classes:

(i) Hyperelliptic curves of genus 1 of the form

$$C : y^2 + h(x)y = f(x)$$

with f of degree 3 or 4 and h of degree at most 1. If the function x on C has a rational branch point then that point is sent to the origin on E. Otherwise, if C has a rational point at $x = \infty$ then that point is used.

(ii) Nonsingular plane curves of degree 3. If the curve is already in general Weierstrass form up to a permutation of the variables then this is recognised and used as a model for the elliptic curve. Otherwise the base field of the curve must have characteristic different from 2 and 3; in this case, the curve is tested for having a rational flex. If it has then a linear transformation suffices to get the curve into general Weierstrass form, and this is used.

(iii) Singular plane curves of degree 4 over a base field of characteristic different from 2 with a unique cusp, with the tangent cone meeting the curve only at that point. Up to linear transformation, these are curves of type

$$y^2 = f(x),$$

with f of degree 4. Such curves are brought into the standard form above. If either a rational point exists with $x = 0$ or the curve intersects the line at infinity in a rational point then that point is used to put the curve in general Weierstrass form.

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

```
EllModel: Verbose                    Default: 3
```

Given a scheme C describing a curve of genus 1 with a nonsingular rational point P, this function returns an elliptic curve E together with a birational map sending the supplied point to the origin on E.

If C is a plane curve of degree 3 over a base field having characteristic different from 2 and 3 then particular tricks are tried. If the supplied point is a flex then a linear transformation is used. Otherwise, Nagell's algorithm [Nag28], also described in [Cas91], is used.

If C is a plane curve of degree 4 over a base field of characteristic different from 2 with a unique cusp and a tangent cone that meets the curve only in that cusp, then a construction in [Cas91] is used.

In all other cases, C has to be a plane curve and a Riemann–Roch computation is used. This is potentially very expensive.

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

```
EllModel: Verbose                    Default: 3
```

Given a plane curve C of genus 1 and a place pl of degree 1, this function returns an elliptic curve E together with a birational map from C to E.

This routine uses a (potentially expensive) Riemann–Roch computation. This is different to the routine that takes a point instead of a place, which uses special methods when the curve has degree 3 or 4.

`SupersingularEllipticCurve(K): FldFin -> CrvEll`

Given a finite field K, this function returns a representative supersingular elliptic curve over K.

Example: Nagell's Algorithm for a Plane Cubic `(ex-86c457)`

In the following example Nagell's algorithm is applied.

```
> P2<X, Y, Z> := ProjectiveSpace(Rationals(), 2);
> C := Curve(P2, X^3 + Y^2*Z - X*Y*Z - Z^3);
> pt := C![0, 1, 1];
> time E1, phi1 := EllipticCurve(C, pt);
Time: 0.070
> E1;
Elliptic Curve defined by y^2 = x^3 - 5/16*x^2 + 1/32*x + 15/1024 over Rational
Field
```

Whereas this next call (given a place) is forced to do the Riemann–Roch computation.

```
> time E2, phi2 := EllipticCurve(C, Place(pt));
Time: 0.120
> E2;
Elliptic Curve defined by y^2 + 1024*y = x^3 + 16*x^2 over Rational Field
> phi1;
Mapping from: CrvPln: C to CrvEll: E1
with equations :
9/8*X^2 - 1/4*X*Y
-1/8*X^2 + 1/4*X*Y + 13/16*X*Z - 1/8*Y*Z - 1/8*Z^2
X^2 - 2*X*Y + 2*X*Z
and inverse
$.1^2 - 3/16*$.1*$.3 + 1/128*$.3^2
1/2*$.1^2 + $.1*$.2 - 15/32*$.1*$.3 + 9/256*$.3^2
$.1*$.2 + 1/8*$.1*$.3 - 1/8*$.2*$.3 - 1/64*$.3^2
> phi2;
Mapping from: CrvPln: C to CrvEll: E2
with equations :
-64*X^2*Y + 64*X^2*Z - 128*X*Y^2 + 256*X*Y*Z - 128*X*Z^2 + 64*Y^2*Z - 64*Y*Z^2
256*X^2*Y + 256*X^2*Z + 1024*X*Y*Z - 1024*X*Z^2 + 1792*Y^2*Z - 4352*Y*Z^2 +
    2048*Z^3
Y^3 - 3*Y^2*Z + 3*Y*Z^2 - Z^3
```

Example: Elliptic Model via Riemann-Roch (ex-d9c6ad)

In this example the starting curve is less clearly elliptic. Since this curve does not match any of the special forms described the Riemann–Roch computation will be used.

```
> 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);
> E;
Elliptic Curve defined by y^2 = x^3 + 3*x^2 + 3*x over Rational Field
> toE;
Mapping from: CrvPln: C to CrvEll: E
```

131.2.2 Creation Predicates

`IsEllipticCurve([a, b]): [ RngElt ] -> BoolElt, CrvEll`

`IsEllipticCurve([a1, a2, a3, a4, a6]): [ RngElt ] -> BoolElt, CrvEll`

The function returns `true` if the given sequence of ring elements defines an elliptic curve (in other words, if the discriminant is nonzero). When true, the elliptic curve is also returned.

`IsEllipticCurve(C): CrvHyp -> BoolElt, CrvEll, MapIsoSch, MapIsoSch`

Given a hyperelliptic curve, the function returns `true` if C has degree 3. When true, the function

also returns the elliptic curve, and isomorphisms to and from it. This function is deprecated and will be removed in a later release. Instead, use `Degree(C) eq 3` and `EllipticCurve(C)`.

Example: Testing Coefficients for Elliptic Curves (ex-9904dd)

We check a few small primes to see which ones make certain coefficients define an elliptic curve.

```
> S := [ p : p in [1..20] | IsPrime(p) ];
> for p in S do
>     ok, E := IsEllipticCurve([GF(p) | 1, 1, 0, -3, -17 ]);
>     if ok then print E; end if;
> end for;
Elliptic Curve defined by y^2 + x*y = x^3 + x^2 + 1 over GF(3)
Elliptic Curve defined by y^2 + x*y = x^3 + x^2 + 10*x + 9 over GF(13)
Elliptic Curve defined by y^2 + x*y = x^3 + x^2 + 14*x over GF(17)
```

131.2.3 Changing the Base Ring

The following general scheme functions provide a convenient way to generate a new elliptic curve similar to an old one but defined over a different field. Some care must be taken, however: If the result is not a valid elliptic curve then the functions will still succeed but the object returned will be a general curve rather than an elliptic curve.

`BaseChange(E, K): CrvEll, Rng -> CrvEll`

`BaseExtend(E, K): CrvEll, Rng -> CrvEll`

Given an elliptic curve E defined over a field k, and a field K which is an extension of k, returns an elliptic curve E' over K using the natural inclusion of k in K to map the coefficients of E into K.

`ChangeRing(E, K): CrvEll, Rng -> CrvEll`

Given an elliptic curve E defined over a field k, and a field K, returns an elliptic curve E' over K by using the standard coercion from k to K to map the coefficients of E into K. This is useful when there is no appropriate ring homomorphism between k and K (e.g., when $k = \mathbb{Q}$ and K is a finite field).

`BaseChange(E, h): CrvEll, Map -> CrvEll`

`BaseExtend(E, h): CrvEll, Map -> CrvEll`

Given an elliptic curve E over a field k and a ring map $h : k \to K$, returns an elliptic curve E' over K by applying h to the coefficients of E.

`BaseChange(E, n): CrvEll, RngIntElt -> CrvEll`

`BaseExtend(E, n): CrvEll, RngIntElt -> CrvEll`

Given an elliptic curve E defined over a finite field K, returns the base extension of E to the degree n extension of K.

Example: Extending the Base Field (ex-9a8eb0)

```
> K1 := GF(23);
> K2 := GF(23, 2);
> f := hom<K1 -> K2 | >;
> E1 := EllipticCurve([K1 | 1, 1]);
> E2 := EllipticCurve([K2 | 1, 1]);
> assert E2 eq BaseExtend(E1, K2);
> assert E2 eq BaseExtend(E1, f);
> assert E2 eq BaseExtend(E1, 2);
> // this is illegal, since K1 is not an extension of K2
> BaseExtend(E2, K1);

>> BaseExtend(E2, K1);
```

(continues on next page)

(continued from previous page)

```
                ^
Runtime error in 'BaseExtend': Coercion of equations is not possible
> assert E1 eq ChangeRing(E2, K1);  // but this is OK
```

131.2.4 Alternative Models

Given an elliptic curve E, there are standard alternative models for E that may be of interest. Each of the functions in this section returns an isomorphic curve E' that is the desired model, together with the isomorphisms $E \to E'$ and $E' \to E$.

Note: The second isomorphism is now completely redundant as it is the inverse of the first; in a later release only the first will be returned.

`WeierstrassModel(E): CrvEll -> CrvEll, Map, Map`

Given an elliptic curve E, this function returns an isomorphic elliptic curve E' in simplified Weierstrass form $y^2 = x^3 + ax + b$. It does not apply when the base ring of E has characteristic 2 or 3 (in which case such a simplified form may not exist).

`IntegralModel(E): CrvEll -> CrvEll, Map, Map`

Given an elliptic curve E defined over a number field K (which may be $\mathbb{Q}$), this function returns an isomorphic elliptic curve E' defined over K with integral coefficients.

`SimplifiedModel(E): CrvEll -> CrvEll, Map, Map`

A simplified model of the elliptic curve E is returned. If E is defined over $\mathbb{Q}$ this has the same effect as `MinimalModel` below. Otherwise, if the characteristic of the base ring is different from 2 and 3, the simplified model agrees with the `WeierstrassModel`. For characteristics 2 and 3 the situation is more complicated; see chapter 4 of [Con99] for the definition for curves defined over finite fields.

`MinimalModel(E): CrvEll -> CrvEll, Map, Map`

Given an elliptic curve E defined over $\mathbb{Q}$ or a number field K with class number one, returns a global minimal model E' for E. By definition, the global minimal model E' is an integral model isomorphic to E over K such that the discriminant of E' has minimal valuation at all non-zero prime ideals π of K. (For $\mathbb{Q}$, this means that it has minimal p-adic valuation at all primes p.)

Such a global minimal model is only guaranteed to exist when the class number of K is 1 (and so one always exists when $K = \mathbb{Q}$); depending on the curve, however, such a model may exist even when this is not the case. If no such minimal model exists then a runtime error will occur.

`MinimalModel(E, p): CrvEll, RngIntElt -> CrvEll, Map, Map`

`MinimalModel(E, P : parameters): CrvEll, RngOrdIdl -> CrvEll, Map`

UseGeneratorAsUniformiser : BoolElt : `false`

Given an elliptic curve E defined over a number field K, and a prime ideal P, returns a curve isomorphic to E which is minimal at P. If K is $\mathbb{Q}$ then the ideal may be specified by the appropriate (integer) prime p.

For curves over number fields, when the parameter `UseGeneratorAsUniformiser` is set to `true` then Magma will check whether the ideal is principal; if so, an ideal generator will be used as the uniformising element. This means that at other primes the returned model will remain integral, or minimal, if the given model was.

131.2.5 Predicates on Curve Models

`IsWeierstrassModel(E): CrvEll -> BoolElt`

Returns `true` if and only if the elliptic curve E is in simplified Weierstrass form.

`IsIntegralModel(E): CrvEll -> BoolElt`

Returns `true` if and only if the elliptic curve E is an integral model.

`IsSimplifiedModel(E): CrvEll -> BoolElt`

Returns `true` if and only if the elliptic curve E is a simplified model.

`IsMinimalModel(E): CrvEll -> BoolElt`

Returns `true` if and only if the elliptic curve E is a minimal model.

`IsIntegralModel(E, P): CrvEll, RngOrdIdl -> BoolElt`

Returns `true` if the defining coefficients of the elliptic curve E, a curve over a number field, have non-negative valuation at the prime ideal P, which must be an ideal of an order of the coefficient ring of E.

Example: Integral and Minimal Models (ex-42e4da)

We define an elliptic curve over the rationals and then find an integral model, a minimal model, and an integral model for the short Weierstrass form.

```
> E := EllipticCurve([1/2, 1/2, 1, 1/3, 4]);
> E;
Elliptic Curve defined by y^2 + 1/2*x*y + y = x^3 + 1/2*x^2 + 1/3*x + 4
over Rational Field
> IE := IntegralModel(E);
> IE;
Elliptic Curve defined by y^2 + 3*x*y + 216*y = x^3 + 18*x^2 + 432*x +
186624 over Rational Field
> ME := MinimalModel(IE);
> ME;
Elliptic Curve defined by y^2 + x*y + y = x^3 - x^2 + 619*x + 193645
over Rational Field
> WE := WeierstrassModel(E);
> WE;
Elliptic Curve defined by y^2 = x^3 + 9909/16*x + 6201603/32 over
Rational Field
> IWE := IntegralModel(WE);
> IWE;
Elliptic Curve defined by y^2 = x^3 + 649396224*x + 208091266154496
over Rational Field
> IsIsomorphic(IWE, ME);
true
```

131.2.6 Twists of Elliptic Curves

In the following, all twists will be returned in the form of simplified models.

`QuadraticTwist(E, d): CrvEll, RngElt -> CrvEll`

Given an elliptic curve E and an element d of the base ring, returns the quadratic twist by d. This is isomorphic to E if and only if either the characteristic is 2 and the trace of d is 0, or the characteristic is not 2 and d is a square. The routine does not always work in characteristic 2.

`QuadraticTwist(E): CrvEll -> CrvEll`

Given an elliptic curve E over a finite field, returns a quadratic twist; that is, a nonisomorphic curve whose trace is the negation of `Trace(E)`.

`QuadraticTwists(E): CrvEll -> SeqEnum`

Given an elliptic curve E over a finite field, returns the sequence of nonisomorphic quadratic twists. The first of these curves is isomorphic to E.

`Twists(E): CrvEll -> SeqEnum`

Given an elliptic curve over a finite field K, returns the sequence of all nonisomorphic elliptic curves over K which are isomorphic over an extension field. The first of these curves is isomorphic to E.

Example: Quadratic Twists over a Finite Field (ex-eb8251)

In this example we compute the quadratic twists of an elliptic curve over the finite field $\mathbb{F}_{13}$ and verify that they fall in two isomorphism classes over the base field.

```
> E1 := EllipticCurve([GF(13) | 3, 1]);
> E5 := QuadraticTwist(E1, 5);
> E5;
Elliptic Curve defined by y^2 = x^3 + 10*x + 8 over GF(13)
> S := QuadraticTwists(E1);
> S;
[
    Elliptic Curve defined by y^2 = x^3 + 3*x + 1 over GF(13),
    Elliptic Curve defined by y^2 = x^3 + 12*x + 5 over GF(13)
]
> [ IsIsomorphic(E1, E) : E in S ];
[ true, false ]
> [ IsIsomorphic(E5, E) : E in S ];
[ false, true ]
```

`IsTwist(E, F): CrvEll, CrvEll -> BoolElt`

Returns `true` if and only if the elliptic curves E and F are isomorphic over an extension field — this function only tests the j-invariants of the curves.

`IsQuadraticTwist(E, F): CrvEll, CrvEll -> BoolElt, RngElt`

Returns `true` if and only if the elliptic curves E and F are isomorphic over a quadratic extension field; if so, returns an element d of the base field such that F is isomorphic to `QuadraticTwist(E, d)`.

Example: Nonquadratic Twists over a Finite Field (ex-3728b4)

In this example we take curves of j-invariant 0 and $12^3 = 12$ over the finite field $\mathbb{F}_{13}$, and compute all twists. Since the automorphism groups are nontrivial the number of twists is larger than the number of quadratic twists. By computing the numbers of points we see that the curves are indeed pairwise nonisomorphic over the base field.

```
> E3 := EllipticCurve([GF(13) | 0, 1]);
> jInvariant(E3);
0
> E4 := EllipticCurve([GF(13) | 1, 0]);
> jInvariant(E4);
12
> T3 := Twists(E3);
> T3;
[
    Elliptic Curve defined by y^2 = x^3 + 1 over GF(13),
    Elliptic Curve defined by y^2 = x^3 + 2 over GF(13),
    Elliptic Curve defined by y^2 = x^3 + 4 over GF(13),
    Elliptic Curve defined by y^2 = x^3 + 8 over GF(13),
    Elliptic Curve defined by y^2 = x^3 + 3 over GF(13),
    Elliptic Curve defined by y^2 = x^3 + 6 over GF(13)
]
> [ #E : E in T3 ];
[ 12, 19, 21, 16, 9, 7 ]
> T4 := Twists(E4);
> T4;
[
    Elliptic Curve defined by y^2 = x^3 + x over GF(13),
    Elliptic Curve defined by y^2 = x^3 + 2*x over GF(13),
```

(continues on next page)

(continued from previous page)

```
    Elliptic Curve defined by y^2 = x^3 + 4*x over GF(13),
    Elliptic Curve defined by y^2 = x^3 + 8*x over GF(13)
]
> [ #E : E in T4 ];
[ 20, 10, 8, 18 ]
```

Observe that exactly two of the twists are quadratic twists (as must always be the case).

```
> [ IsQuadraticTwist(E3, E) : E in T3 ];
[ true, false, false, true, false, false ]
> [ IsQuadraticTwist(E4, E) : E in T4 ];
[ true, false, true, false ]
```

MinimalQuadraticTwist(E): CrvEll -> CrvEll, RngIntElt

Determine the minimal twist of the rational elliptic curve E. This is defined locally prime-by-prime, and the algorithm for finding it is based on this observation. For each odd prime p of bad reduction the algorithm iteratively replaces the curve by its twist by p if the latter has smaller discriminant.

For $p = 2$ there is no accepted definition of minimal, and the prime at infinity also plays a role. After having handled all the odd primes there are only twists by -1, 2, and -2 left to consider. The algorithm implemented in Magma first chooses a twist with minimal 2-valuation of the conductor; this is unique precisely when this 2-valuation is less than 5. [Note that others have chosen instead to minimise the 2-valuation of Δ.]

The prime 2 can be eliminated at this point as was done in the case of the odd primes, leaving only the consideration of twisting the curve by -1. Here the algorithm arbitrarily chooses the curve that has $(-1)^{v_2(c_6)}\text{odd}(c_6)$ congruent to 3 mod 4. The function returns as its second argument the integer d by which the curve was twisted to obtain the minimal twist.

Example: Minimal Quadratic Twist (ex-a9631b)

```
> E:=EllipticCurve([0, 1, 0, 135641131, 1568699095683]);
> M, tw := MinimalQuadraticTwist(E);
> M, tw;
Elliptic Curve defined by y^2 + y = x^3 + x^2 + 3*x + 5 over Rational Field
7132
> MinimalModel(QuadraticTwist(M, tw));
Elliptic Curve defined by y^2 = x^3 + x^2 + 135641131*x + 1568699095683 over
Rational Field
```

131.3 Operations on Curves

131.3.1 Elementary Invariants

aInvariants(E): CrvEll -> [RngElt]

Coefficients(E): CrvEll -> [RngElt]

ElementToSequence(E): CrvEll -> [RngElt]

Eltseq(E): CrvEll -> [RngElt]

Given an elliptic curve E, this function returns a sequence consisting of the Weierstrass coefficients of E; this is the sequence $[a_1, a_2, a_3, a_4, a_6]$ such that E is defined by $y^2z + a_1xyz + a_3yz^2 = x^3 + a_2x^2z + a_4xz^2 + a_6z^3$. Note that this function returns the five coefficients even if E was defined by a sequence $[a, b]$ of length two (the first three coefficients are zero in such a case).

bInvariants(E): CrvEll -> [RngElt]

This function returns a sequence of length 4 containing the b-invariants of the elliptic curve E,

namely $[b_2, b_4, b_6, b_8]$. In terms of the coefficients of E these are defined by

$$\begin{aligned} b_2 &= a_1^2 + 4a_2 \\ b_4 &= a_1a_3 + 2a_4 \\ b_6 &= a_3^2 + 4a_6 \\ b_8 &= a_1^2a_6 + 4a_2a_6 - a_1a_3a_4 + a_2a_3^2 - a_4^2. \end{aligned}$$

`cInvariants(E): CrvEll -> [ RngElt ]`

This function returns a sequence of length 2 containing the c-invariants of the elliptic curve E, namely $[c_4, c_6]$. In terms of the b-invariants of E these are defined by

$$\begin{aligned} c_4 &= b_2^2 - 24b_4 \\ c_6 &= -b_2^3 + 36b_2b_4 - 216b_6. \end{aligned}$$

`Discriminant(E): CrvEll -> RngElt`

This function returns the discriminant Δ of the elliptic curve E. In terms of the b-invariants of E it is defined by

$$\Delta = -b_2^2b_8 - 8b_4^3 - 27b_6^2 + 9b_2b_4b_6$$

and there is also the relationship $1728\Delta = c_4^3 - c_6^2$.

`jInvariant(E): CrvEll -> RngElt`

Return the j-invariant of the elliptic curve E. In terms of the c-invariants and the discriminant of E it is defined by $j = c_4^3/\Delta$. Two elliptic curves defined over the same base field are isomorphic over some extension field exactly when their j-invariants are equal.

`HyperellipticPolynomials(E): CrvEll -> RngUPolElt, RngUPolElt`

Returns polynomials $x^3 + a_2x^2 + a_4x + a_6$ and $a_1x + a_3$, formed from the invariants of the elliptic curve E.

Example: Invariants of an Elliptic Curve (ex-aa8afb)

Here are a few simple uses of the above functions.

```
> E := EllipticCurve([0, -1, 1, 1, 0]);
> E;
Elliptic Curve defined by y^2 + y = x^3 - x^2 + x over Rational Field
> aInvariants(E);
[ 0, -1, 1, 1, 0 ]
> Discriminant(E);
-131
> c4, c6 := Explode(cInvariants(E));
> jInvariant(E) eq c4^3 / Discriminant(E);
true
```

Example: Invariants of a Generic Curve (ex-6fe5ca)

By constructing a generic elliptic curve we can see that the relationships described above hold.

```
> F<a1, a2, a3, a4, a6> := FunctionField(Rationals(), 5);
> E := EllipticCurve([a1, a2, a3, a4, a6]);
> E;
Elliptic Curve defined by y^2 + a1*x*y + a3*y = x^3 + a2*x^2 + a4*x + a6
over F
> aInvariants(E);
[
    a1,
```

(continues on next page)

(continued from previous page)

```
    a2,
    a3,
    a4,
    a6
]
> bInvariants(E);
[
    a1^2 + 4*a2,
    a1*a3 + 2*a4,
    a3^2 + 4*a6,
    a1^2*a6 - a1*a3*a4 + a2*a3^2 + 4*a2*a6 - a4^2
]
> b2,b4,b6,b8 := Explode(bInvariants(E));
> cInvariants(E);
[
    a1^4 + 8*a1^2*a2 - 24*a1*a3 + 16*a2^2 - 48*a4,
    -a1^6 - 12*a1^4*a2 + 36*a1^3*a3 - 48*a1^2*a2^2 + 72*a1^2*a4 +
        144*a1*a2*a3 - 64*a2^3 + 288*a2*a4 - 216*a3^2 - 864*a6
]
> c4,c6 := Explode(cInvariants(E));
> c4 eq b2^2 - 24*b4;
true
> c6 eq -b2^3 + 36*b2*b4 - 216*b6;
true
> d := Discriminant(E);
> d;
-a1^6*a6 + a1^5*a3*a4 - a1^4*a2*a3^2 - 12*a1^4*a2*a6 + a1^4*a4^2 +
    8*a1^3*a2*a3*a4 + a1^3*a3^3 + 36*a1^3*a3*a6 - 8*a1^2*a2^2*a3^2 -
    48*a1^2*a2^2*a6 + 8*a1^2*a2*a4^2 - 30*a1^2*a3^2*a4 + 72*a1^2*a4*a6 +
    16*a1*a2^2*a3*a4 + 36*a1*a2*a3^3 + 144*a1*a2*a3*a6 - 96*a1*a3*a4^2 -
    16*a2^3*a3^2 - 64*a2^3*a6 + 16*a2^2*a4^2 + 72*a2*a3^2*a4 +
    288*a2*a4*a6 - 27*a3^4 - 216*a3^2*a6 - 64*a4^3 - 432*a6^2
> d eq -b2^2*b8 - 8*b4^3 - 27*b6^2 + 9*b2*b4*b6;
true
> 1728*d eq c4^3 - c6^2;
true
```

131.3.2 Associated Structures

Category(E): CrvEll -> Cat

Type(E): CrvEll -> Cat

Returns the category of elliptic curves, CrvEll.

BaseRing(E): CrvEll -> Rng

CoefficientRing(E): CrvEll -> Rng

The base ring of the elliptic curve E; that is, the parent of its coefficients and the coefficient ring of the default point set of E.

131.3.3 Predicates on Elliptic Curves

E **eq** F: CrvEll, CrvEll -> BoolElt

Returns **true** if and only if the elliptic curves E and F are defined over the same ring and have the same coefficients.

E **ne** F: CrvEll, CrvEll -> BoolElt

The logical negation of **eq**.

IsIsomorphic(E, F): CrvEll, CrvEll -> BoolElt, Map

Given two elliptic curves E and F this function returns `true` if there exists an isomorphism between E and F over the base field, and `false` otherwise. If E and F are isomorphic then the isomorphism is returned as a second value. This function requires being able to take roots in the base field.

IsIsogenous(E, F): CrvEll[FldRat], CrvEll[FldRat] -> BoolElt, Map

IsIsogenous(E, F): CrvEll[FldFin], CrvEll[FldFin] -> BoolElt

Given two elliptic curves E and F defined over the rationals or a finite field, this function returns `true` if the curves E and F are isogenous over this field and `false` otherwise. In the rational case, if the curves are isogenous then the isogeny will be returned as the second value. For finite fields the isogeny computation operates via point counting and thus no isogeny is returned.

Example: Twists Isomorphic over an Extension (ex-4c9255)

We return to the curves in the earlier quadratic twist example. By definition, these curves are not isomorphic over their base field, but are isomorphic over a quadratic extension.

```
> K := GF(13);
> E := EllipticCurve([K | 3, 1]);
> E5 := QuadraticTwist(E, 5);
> IsIsomorphic(E, E5);
false
> IsIsomorphic(BaseExtend(E, 2), BaseExtend(E5, 2));
true
```

Since they are isomorphic over an extension, their j-invariants must be the same.

```
> jInvariant(E) eq jInvariant(E5);
true
```

131.4 Polynomials

The *torsion* or *division* polynomials are the polynomials defining the subschemes of n-torsion points on the elliptic curve.

DefiningPolynomial(E): CrvEll -> RngMPolElt

Returns the homogeneous defining polynomial for the elliptic curve E.

DivisionPolynomial(E, n): CrvEll, RngIntElt -> RngUPolElt, RngUPolElt, RngUPolElt

DivisionPolynomial(E, n, g): CrvEll, RngIntElt, RngUPolElt -> RngUPolElt, RngUPolElt, RngUPolElt

Given an elliptic curve E and an integer n, returns the n-th division polynomial as a univariate polynomial over the base ring of E; its roots are the x-coordinates of the nonzero n-torsion points, each occurring once. The second return value is this polynomial divided by the univariate 2-torsion polynomial if n is even. The third argument is the cofactor, equal to the univariate 2-torsion polynomial if n is even, and 1 otherwise.

If a polynomial is passed as a third argument then the division polynomial is computed efficiently modulo that polynomial.

TwoTorsionPolynomial(E): CrvEll -> RngMPolElt

Returns the multivariate 2-torsion polynomial $2y + a_1x + a_3$ of the elliptic curve E, as a bivariate polynomial.

Example: Division Polynomials and Torsion Points (ex-d2495b)

Let E be an elliptic curve over a finite field K. The following code fragment illustrates the relationship between the roots in K of the n-th division polynomial for E, and the x-coordinates of the points of n-torsion on E.

```
> K := GF(101);
> E := EllipticCurve([ K | 1, 1]);
> Roots(DivisionPolynomial(E, 5));
[ <86, 1>, <46, 1> ]
> [ P : P in RationalPoints(E) | 5*P eq E!0 ];
[ (86 : 34 : 1), (0 : 1 : 0), (46 : 25 : 1), (86 : 67 : 1), (46 : 76 : 1) ]
```

It is worth noting that even if the roots of the division polynomial lie in the field, the corresponding points may not, lying instead in a quadratic extension.

```
> Roots(DivisionPolynomial(E, 9));
[ <4, 1>, <17, 1>, <28, 1>, <34, 1>, <77, 1> ]
> Points(E, 4);
[]
> K2<w> := ext<K | 2>;
> Points(E(K2), 4);
[ (4 : 82*w + 57 : 1), (4 : 19*w + 44 : 1) ]
> Order($1[1]);
9
```

131.4.1 Complex Multiplication Division Polynomials

Let E be an elliptic curve with complex multiplication by an order O in an imaginary quadratic field K, and let $\alpha \in O$. The endomorphism $[\alpha] : E \to E$ acts on x-coordinates by a rational function $x([\alpha]P) = \phi_\alpha(x)/\psi_\alpha(x)^e$, generalising the classical division polynomials; here $e = 2$ when $N(\alpha)$ is odd and $e = 1$ when the kernel is entirely 2-torsion (a prime above 2), matching the convention of `MultiplicationByMMap`; for other even $N(\alpha)$, where the kernel mixes 2-torsion with higher-order points, no single e works and `CMEndomorphism` should be used for the map instead. The endomorphism is normalised so that $[\alpha]^*\omega = \alpha\omega$ on the invariant differential ω; equivalently ϕ_α has leading coefficient $1/\alpha^2$. The element α may be prime, a prime power, or composite: a composite α is handled by composing the prime endomorphisms, which requires K to have class number 1. Currently E must be defined over $\mathbb{Q}$ or over a field containing the CM field K (when K has class number greater than 1 the CM endomorphisms are defined over the ring class field, so E must be given there), and positive characteristic (finite fields) is not supported.

`DivisionPolynomial(E, alpha): CrvEll, RngOrdElt -> RngUPolElt`

Given an elliptic curve E with complex multiplication by an order containing α, returns the complex multiplication division polynomial ψ_α: the monic squarefree polynomial over the CM field K whose roots are the x-coordinates of the nonzero points in the kernel of the endomorphism $[\alpha]$ (each occurring once). For a prime α lying above an odd rational prime p its degree is $(p^2 - 1)/2$ when p is inert and $(p - 1)/2$ when p splits or ramifies. For a prime above 2 the degree is 3 when 2 is inert (the kernel is all of $E[2]$) and 1 when 2 splits or ramifies. For a split prime the two prime ideals above p give distinct kernels, and the polynomial returned is the one belonging to the specific generator α, its conjugate giving $\psi_{\bar{\alpha}}$.

`DivisionPolynomial(E, a): CrvEll, RngOrdIdl -> RngUPolElt`

Given an elliptic curve E with complex multiplication and an ideal a of its CM order, returns the a-division polynomial: the monic polynomial whose roots are the x-coordinates of the nonzero points of the a-torsion $E[a]$ (the points killed by every element of a). For a principal ideal $a = (\alpha)$ this equals `DivisionPolynomial(E, alpha)` and depends only on a, not on the choice of generator. Non-principal ideals (which occur only when the CM field has class number greater than 1) are not currently supported.

`MultiplicationPolynomial(E, alpha): CrvEll, RngOrdElt -> RngUPolElt`

Given an elliptic curve E with complex multiplication by an order containing α, returns the numerator ϕ_α of the multiplication-by-α map, so that $x([\alpha]P) = \phi_\alpha(x)/\psi_\alpha(x)^e$ where ψ_α is `DivisionPolynomial(E, alpha)` and $e = 2$ for $N(\alpha)$ odd, $e = 1$ when the kernel is entirely 2-torsion (a prime above 2). For other even $N(\alpha)$, where the kernel mixes 2-torsion with higher-

order points, no single e reproduces $x([\alpha]P)$ from ψ_α; use `CMEndomorphism` for the map in that case. This is the complex multiplication analogue of the multiplication map `MultiplicationByMMap(E, n)` for a rational integer n. The polynomial ϕ_α has degree $N(\alpha)$ and leading coefficient $1/\alpha^2$.

`CMEndomorphism(E, alpha): CrvEll, RngOrdElt -> Map`

Given an elliptic curve E with complex multiplication by an order containing α, returns the CM endomorphism $[\alpha] : E \to E$ as a map, normalised so that $[\alpha]^*\omega = \alpha\omega$ on the invariant differential. On x-coordinates it acts by the rational function of `MultiplicationPolynomial` and `DivisionPolynomial` above. This is the complex multiplication analogue of `MultiplicationByMMap(E, n)` for a rational integer n.

Example: Division Polynomials for Complex Multiplication (ex-58df2e)

We take the $j = 0$ curve $y^2 = x^3 - 1$, which has complex multiplication by the Eisenstein integers, and compute the division and multiplication polynomials of the endomorphism $[\alpha]$ for $\alpha = 2 + a$, where $a = (1 + \sqrt{-3})/2$; here $N(\alpha) = 7$.

```
> K<w> := QuadraticField(-3);
> O<a> := MaximalOrder(K);
> E := EllipticCurve([K | 0, 0, 0, 0, -1]);
> _, D := HasComplexMultiplication(E);
> D;
-3
> alpha := 2 + a;
> Norm(alpha);
7
> psi := DivisionPolynomial(E, alpha);
> psi;
x^3 + 1/7*(-6*w + 2)
> ChangeRing(DivisionPolynomial(E, 7), K) mod psi eq 0;
true
> phi := MultiplicationPolynomial(E, alpha);
> phi;
1/98*(-5*w + 11)*x^7 + 1/7*(26*w + 10)*x^4 + 1/7*(-16*w - 32)*x
```

131.5 Subgroup Schemes

A subgroup scheme G of an elliptic curve E is a subscheme of E defined by a univariate polynomial ψ and closed under the group law on E. The points of G are those points of E whose x-coordinate is a root of ψ. All elliptic curves are considered to be subgroup schemes with defining polynomial $\psi = 0$.

131.5.1 Creation of Subgroup Schemes

`SubgroupScheme(G, f): SchGrpEll, RngUPolElt -> SchGrpEll`

Creates the subgroup scheme of the subgroup scheme G defined by the univariate polynomial f. No checking is done to ensure that the rational points of the result actually do form a group under the addition law. Note that G can be an elliptic curve.

`TorsionSubgroupScheme(G, n): SchGrpEll, RngIntElt -> SchGrpEll`

Returns the subgroup scheme of n-torsion points of the subgroup scheme G. Note that G can be an elliptic curve.

131.5.2 Associated Structures

`Category(G): SchGrpEll -> Cat`

`Type(G): SchGrpEll -> Cat`

Returns the category of elliptic curve subgroup schemes, `SchGrpEll`.

Curve(G): SchGrpEll -> CrvEll

Generic(G): SchGrpEll -> CrvEll

Returns the elliptic curve E of which G is a subgroup scheme.

BaseRing(G): SchGrpEll -> Rng

CoefficientRing(G): SchGrpEll -> Rng

Returns the base ring of the subgroup scheme G; this is the same as the base ring of its curve.

DefiningSubschemePolynomial(G): SchGrpEll -> RngUPolElt

Returns the univariate polynomial that defines G as a subscheme of its curve.

131.5.3 Predicates on Subgroup Schemes

G1 **eq** G2: SchGrpEll, SchGrpEll -> BoolElt

Returns **true** if and only if $G1$ and $G2$ are subgroup schemes of the same elliptic curve and are defined by equal polynomials.

G1 **ne** G2: SchGrpEll, SchGrpEll -> BoolElt

The logical negation of **eq**.

131.5.4 Points of Subgroup Schemes

G: SchGrpEll -> RngIntElt

Order(G): SchGrpEll -> RngIntElt

The order of the group of rational points on the subgroup scheme G.

FactoredOrder(G): SchGrpEll -> RngIntElt

The factorisation of the order of the group of rational points on the subgroup scheme G.

Points(G): SchGrpEll -> SetIndx

RationalPoints(G): SchGrpEll -> SetIndx

The indexed set of rational points of the subgroup scheme G over its base ring.

Example: Subgroup Schemes of a Curve (ex-125594)

We construct a curve over $\mathbb{F}_{49}$ and form several subgroup schemes from it.

```
> K<w> := GF(7, 2);
> P<t> := PolynomialRing(K);
> E := EllipticCurve([K | 1, 3]);
> G := SubgroupScheme(E, (t-4)*(t-5)*(t-6));
> G;
Subgroup scheme of E defined by x^3 + 6*x^2 + 4*x + 6
> Points(G);
{@ (0 : 1 : 0), (6 : 1 : 1), (6 : 6 : 1), (4 : 1 : 1), (4 : 6 : 1),
(5 : 0 : 1) @}
```

The points of order 3 form a further subgroup.

```
> [ Order(P) : P in $1 ];
[ 1, 3, 3, 6, 6, 2 ]
> G2 := SubgroupScheme(G, t - 6);
> G2;
Subgroup scheme of E defined by x + 1
> Points(G2);
{@ (0 : 1 : 0), (6 : 1 : 1), (6 : 6 : 1) @}
```

We can find this subgroup another way, as the intersection of the 15-torsion points of E and G.

```
> G3 := TorsionSubgroupScheme(E, 15);
> #G3;
15
> G4 := SubgroupScheme(G3, DefiningSubschemePolynomial(G));
> G4;
Subgroup scheme of E defined by x + 1
> G2 eq G4;
true
```

131.6 The Formal Group

These functions are for elliptic curves over any (exact) field.

FormalGroupLaw(E, prec): CrvEll, RngIntElt -> RngMPolElt

This function returns a polynomial in two variables, $T_1 + T_2 + \ldots$, which expresses the formal group law associated to addition on E, up to precision `prec`. (More precisely, it contains the terms which have total degree less than or equal to `prec`.)

The formal variables T_1 and T_2 may be identified with the function $-x/y$ on E, where x and y are the standard affine coordinates on E. (Note that this function is a local parameter in a neighbourhood of O_E.)

FormalGroupHomomorphism(phi, prec): MapSch, RngIntElt -> RngSerPowElt

This function returns the homomorphism of formal groups associated to the isogeny `phi`, represented as a power series in one variable up to precision `prec`. As in `FormalGroupLaw`, this is in terms of the parameter $-x/y$ on each curve.

FormalLog(E): CrvEll -> RngSerPowElt, PtEll

```
Precision: RngIntElt                    Default: 10
```

This function returns the formal logarithm for the elliptic curve E as a power series $f(T)$, where the parameter T is the function $-x/y$ on E. (This is the same parameter used in `FormalGroupLaw`).

The function also returns a point $P(T)$ on E with coordinates in a Laurent series ring with generator T, which again corresponds to $-x/y$. Thus $P(T)$ is a formal parametrisation of E in a neighbourhood of O_E.

131.7 Operations on Point Sets

Each elliptic curve E has associated with it a family of *point sets* of E indexed by coefficient rings. These point sets, not E, are the objects in which points lie. If K is the base ring of E and L is some extension of K then the elements of the point set $E(L)$ comprise all points lying on E whose coordinates are in L.

There is a distinguished point set $E(K)$ of E which is called the *base point set* of E. Many intrinsics (such as `#`, or `TorsionSubgroup`), strictly speaking, only make sense when applied to point sets; as a convenience, when E is passed to these functions the behaviour is the same as if the base point set $E(K)$ were passed instead. It is important to remember, however, that E and $E(K)$ are different objects and will not always behave in the same manner.

The above statements are equally valid if the elliptic curve E is replaced by some subgroup scheme G. Moreover, the types of the point sets of G and E are the same, and similarly for points. (They may be distinguished by checking the type of the scheme of which they are point sets.)

131.7.1 Creation of Point Sets

`E(L): CrvEll, Rng -> SetPtEll`

`PointSet(E, L): CrvEll, Rng -> SetPtEll`

Given an elliptic curve E (or a subgroup scheme thereof) and an extension L of its base ring, this function returns a point set whose elements are points on E with coefficients in L.

`E(m): CrvEll, Map -> SetPtEll`

`PointSet(E, m): CrvEll, Map -> SetPtEll`

Given an elliptic curve E (or a subgroup scheme thereof) and a map m from the base ring of E to a field L, this function returns a point set whose elements are points on E with coefficients in L. The map is retained to permit coercions between point sets.

131.7.2 Associated Structures

`Category(H): SetPtEll -> Cat`

`Type(H): SetPtEll -> Cat`

Given a point set H of an elliptic curve, returns the category `SetPtEll` of point sets of elliptic curves.

`Scheme(H): SetPtEll -> CrvEll`

Returns the associated scheme (either an elliptic curve or a subgroup scheme) of which H is a point set.

`Curve(H): SetPtEll -> CrvEll`

Returns the associated elliptic curve that contains `Scheme(H)`.

`Ring(H): SetPtEll -> Rng`

Returns the ring that contains the coordinates of points in H.

131.7.3 Predicates on Point Sets

`H1 eq H2: SetPtEll, SetPtEll -> BoolElt`

Returns whether the two point sets are equal. That is, whether the point sets have equal coefficient rings and elliptic curves (subgroup schemes).

`H1 ne H2: SetPtEll, SetPtEll -> BoolElt`

The logical negation of `eq`.

Example: Point Sets over a Finite Field (ex-127ba4)

We create an elliptic curve E over GF(5) and then construct two associated point sets:

```
> K := GF(5);
> E := EllipticCurve([K | 1, 0]);
> H := E(K);
> H;
Set of points of E with coordinates in GF(5)
> H2 := E(GF(5, 2));
> H2;
Set of points of E with coordinates in GF(5^2)
```

We note that although these are point sets of the same curve, they are not equal because the rings are not equal.

```
> Scheme(H) eq Scheme(H2);
true
> Ring(H) eq Ring(H2);
false
```

(continues on next page)

(continued from previous page)

```
> H eq H2;
false
```

Similarly, we see that a point set of a subgroup scheme is not the same object as the point set of the curve because the schemes are different.

```
> P<t> := PolynomialRing(K);
> G := SubgroupScheme(E, t - 2);
> HG := G(K);
> Scheme(HG) eq Scheme(H);
false
> Ring(HG) eq Ring(H);
true
> HG eq H;
false
```

Also note that the scheme and the parent curve of point sets of G are different:

```
> Scheme(HG);
Subgroup scheme of E defined by x + 3
> Curve(HG);
Elliptic Curve defined by y^2 = x^3 + x over GF(5)
```

131.8 Morphisms

Four types of maps between elliptic curves may be constructed: isogenies, isomorphisms, translations, and rational maps. Isogenies and isomorphisms are by far the most important and have the most functions associated to them. Isogenies are always surjective as scheme maps, even though the Magma parlance of a map from $E(\mathbb{Q}) \to F(\mathbb{Q})$ may seem to indicate that (for instance) the multiplication-by-two map is not surjective. There is an internal limit that the degrees of the polynomials defining an isogeny cannot be more than 10^7.

131.8.1 Creation Functions

Example: Parametrised Family of 2-Isogenies `(ex-4a6549)`

The following example gives a construction of a 2-isogeny between elliptic curves. This example follows Example III 4.5 of Silverman [Sil86], and demonstrates a parametrised family of 2-isogenies of elliptic curves together with its dual.

```
> FF := FiniteField(167);
> a := FF!2; b := FF!3; r := a^2 - 4*b;
> E1 := EllipticCurve([0, a, 0, b, 0]);
> E2 := EllipticCurve([0, -2*a, 0, r, 0]);
> _<x> := PolynomialRing(BaseRing(E1));
> Ff, f := IsogenyFromKernel(E1, x);
> Fg, g := IsogenyFromKernel(E2, x);
> b, m1 := IsIsomorphic(Ff, E2); assert b;
> b, m2 := IsIsomorphic(Fg, E1); assert b;
> // Verify that f and g are dual isogenies of degree 2
> &and[ m2(g(m1(f(P)))) eq 2*P : P in RationalPoints(E1) ];
true
> &and[ m1(f(m2(g(Q)))) eq 2*Q : Q in RationalPoints(E2) ];
true
```

`Isomorphism(E, F, [r, s, t, u])`: `CrvEll, CrvEll, SeqEnum -> Map`

Given elliptic curves E and F defined over the same field K, and four elements r, s, t, u of K

with $u \neq 0$, this function returns the isomorphism $E \to F$ mapping $O_E \mapsto O_F$ and mapping $(x, y) \mapsto (u^2x + r, u^3y + su^2x + t)$. This function returns an error if the values passed do not define such an isomorphism.

`Isomorphism(E, F): CrvEll, CrvEll -> Map`

Given elliptic curves E and F defined over the same field K, this function computes and returns an isomorphism from E to F where such an isomorphism exists. The map returned will be the same as the second return value of `IsIsomorphic`.

`Automorphism(E, [r, s, t, u]): CrvEll, SeqEnum -> Map`

Given an elliptic curve E defined over a field K, and four elements r, s, t, u of K with $u \neq 0$, this function returns the automorphism $E \to E$ mapping $O_E \mapsto O_E$ and mapping $(x, y) \mapsto (u^2x + r, u^3y + su^2x + t)$. This function returns an error if the values passed do not define such an automorphism.

`IsomorphismData(I): Map -> [ RngElt ]`

The sequence $[r, s, t, u]$ of elements defining the isomorphism I.

Example: Testing Curves for Isomorphism (ex-3ea7a9)

We illustrate the isomorphism routines with some simple examples.

```
> K := GF(73);
> E1 := EllipticCurve([K | 3, 4, 2, 5, 1]);
> E2 := EllipticCurve([K | 8, 2, 29, 45, 28]);
> IsIsomorphic(E1, E2);
true
> m := Isomorphism(E1, E2, [3, 2, 1, 4]);
> m;
Elliptic curve isomorphism from: CrvEll: E1 to CrvEll: E2
Taking (x : y : 1) to (16*x + 3 : 64*y + 32*x + 1 : 1)
> P1 := Random(E1);
> P2 := Random(E1);
> m(P1 + P2) eq m(P1) + m(P2);
true
```

From the isomorphism data we can apply the map by hand if desired:

```
> r, s, t, u := Explode(IsomorphismData(Inverse(m)));
> P3 := E2![ 69, 64 ];
> x, y := Explode(Eltseq(P3));
> E1 ! [ u^2*x + r, u^3*y + s*u^2*x + t ];
(68 : 32 : 1)
> m($1) eq P3;
true
```

`IsIsomorphism(I): Map -> BoolElt, Map`

This function returns `true` if and only if the isogeny I has the same action as some isomorphism; if so, the isomorphism is also returned.

`IsomorphismToIsogeny(I): Map -> Map`

This function takes a map I of type isomorphism and returns an equivalent map with type isogeny.

Example: Converting Isogenies and Isomorphisms (ex-f4901b)

An example of how to use the previous two functions to transform isogenies to isomorphisms and vice versa.

```
> FF := FiniteField(23);
> E0 := EllipticCurve([FF | 1, 1]);
```

(continues on next page)

(continued from previous page)

```
> E1 := EllipticCurve([FF | 3, 2]);
> b, iso := IsIsomorphic(E0, E1);
> b;
true
> iso;
Elliptic curve isomorphism from: CrvEll: E0 to CrvEll: E1
Taking (x : y : 1) to (16*x : 18*y : 1)
> isog := IsomorphismToIsogeny(iso);
> isog;
Elliptic curve isogeny from: CrvEll: E0 to CrvEll: E1
taking (x : y : 1) to (16*x : 18*y : 1)
> b, new_iso := IsIsomorphism(isog);
> b;
true
> inv := Inverse(new_iso);
> P := Random(E0);
> inv(isog(P)) eq P;
true
```

TranslationMap(E, P): CrvEll, PtEll -> Map

Given a rational point P on the elliptic curve E, this function returns the morphism $t_P : E \to E$ defined by $t_P(Q) = P + Q$ for all rational points Q of E.

RationalMap(i, t): Map, Map -> Map

Let i be an isogeny and t be a translation map $t_P : E \to E$ where $t_P(Q) = P + Q$ for some rational point $P \in E$. This function returns the rational map $\phi : E \to F$ obtained by composing i and t (applying t first). Any rational map $E \to F$ can be represented in this form.

TwoIsogeny(P): PtEll -> Map

Given a 2-torsion point P of the elliptic curve E, this function returns a 2-isogeny on E with P as its kernel.

Example: Generic Maps Between Curves (ex-fccac1)

One may, of course, also define maps between elliptic curves using the generic map constructors, as the following examples show.

```
> E1 := EllipticCurve([ GF(23) | 1, 1 ]);
> E2 := EllipticCurve([ GF(23, 2) | 1, 1 ]);
```

The doubling map on E_1 lifted to E_2:

```
> f := map<E1 -> E2 | P :-> 2*P>;
```

Two slightly different ways to define the negation map on E_1 lifted to E_2:

```
> f := map<E1 -> E2 | P :-> E2![ P[1], -P[2], P[3] ]>;
> f := map<E1 -> E2 | P :-> (P eq E1!0) select E2!0
>                           else E2![ P[1], -P[2], 1 ]>;
```

IsogenyFromKernel(G): SchGrpEll -> CrvEll, Map

Let G be a subgroup scheme of an elliptic curve E. There is a separable isogeny $f : E \to E_f$ to some other elliptic curve E_f, which has kernel G. This function returns the curve E_f and the map f using Velu's formulae.

IsogenyFromKernelFactored(G): SchGrpEll -> CrvEll, Map

Returns a sequence of isogenies whose product is the isogeny returned by the invocation IsogenyFromKernel(G). These isogenies have either degree 2 or odd degree. This function was

introduced because composing isogenies can be computationally expensive. The generic `Expand` function on a composition of maps can then be used if desired.

`IsogenyFromKernel(E, psi): CrvEll, RngUPolElt -> CrvEll, Map`

Given an elliptic curve E and a univariate polynomial *psi* which defines a subgroup scheme of E, compute an isogeny using Velu's formulae as above.

`IsogenyFromKernelFactored(E, psi): SchGrpEll, RngUPolElt -> CrvEll, Map`

Returns a sequence of isogenies whose product is the isogeny returned by the invocation `IsogenyFromKernel(E, psi)`. These isogenies have either degree 2 or odd degree. This function was introduced because composing isogenies can be computationally expensive. The generic `Expand` function on a composition of maps can then be used if desired.

`PushThroughIsogeny(I, v): Map, RngUPolElt -> RngUPolElt`

`PushThroughIsogeny(I, G): Map, SchGrpEll -> SchGrpEll`

Given an isogeny I and a subgroup G which contains the kernel of I, find the image of G under the action of I. The subgroup G may be replaced by its defining polynomial v.

`DualIsogeny(phi): Map -> Map`

Given an isogeny $\phi : E_1 \to E_2$, the function returns another isogeny $\phi^* : E_2 \to E_1$ such that $\phi^* \circ \phi$ is multiplication by the degree of ϕ. The result is remembered and `DualIsogeny(DualIsogeny(phi))` returns `phi`.

Example: Dual of an Isogeny (ex-b64f03)

We exhibit one way of calculating the dual of an isogeny. First we create a curve and an isogeny f with kernel equal to the full 5-torsion polynomial.

```
> E := EllipticCurve([GF(97) | 2, 3]);
> E1, f := IsogenyFromKernel(E, DivisionPolynomial(E, 5));
```

The image curve E_1 is isomorphic, but not equal, to the original curve E. We proceed to find the dual of f.

```
> deg := Degree(f);
> psi := DivisionPolynomial(E, deg);
> f1 := PushThroughIsogeny(f, psi);
> E2, g := IsogenyFromKernel(E1, f1);
> // Velu's formulae give an isomorphic curve, not the curve itself.
> IsIsomorphic(E2, E);
true
> h := Isomorphism(E2, E);
> f_dual := g*IsomorphismToIsogeny(h);
```

The latter isogeny is the dual of f, as we verify:

```
> &and [ f_dual(f(P)) eq deg*P : P in RationalPoints(E) ];
true
```

A simpler way to verify this is just to check equality (note that both constructions of the dual involved an arbitrary choice of isomorphism):

```
> f_dual eq DualIsogeny(f);
true
> f_dual eq DualIsogeny(f) * NegationMap(E);
false;
```

131.8.2 Predicates on Isogenies

`IsZero(I): Map -> BoolElt`

`IsConstant(I): Map -> BoolElt`

Returns `true` if and only if the image of the isogeny I is the zero element of its codomain.

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

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

131.8.3 Structure Operations

`IsogenyMapPsi(I): Map -> RngUPolElt`

Returns the univariate polynomial ψ used in defining the isogeny I. The roots of ψ determine the kernel of I.

`IsogenyMapPsiMulti(I): Map -> RngUPolElt`

Returns the polynomial ψ used in defining the isogeny I as a multivariate polynomial.

`IsogenyMapPsiSquared(I): Map -> RngUPolElt`

Returns the denominator of the fraction giving the image of the x-coordinate of a point under the isogeny I (the numerator is returned by `IsogenyMapPhi(I)`).

`IsogenyMapPhi(I): Map -> RngUPolElt`

Returns the univariate polynomial ϕ used in defining the isogeny I.

`IsogenyMapPhiMulti(I): Map -> RngUPolElt`

Returns the polynomial ϕ used in defining the isogeny I as a multivariate polynomial.

`IsogenyMapOmega(I): Map -> RngMPolElt`

Returns the multivariate polynomial ω used in defining the isogeny I.

`Kernel(I): Map -> SchGrpEll`

Returns the subgroup of the domain consisting of the elements that map to zero under the isogeny I.

`Degree(I): Map -> RngIntElt`

Returns the degree of the morphism I.

131.8.4 Endomorphisms

Let E be an elliptic curve defined over the field K. The set of endomorphisms $\mathrm{End}(E)$ of E to itself forms a ring under composition and addition of maps.

The endomorphism ring of E always contains a subring isomorphic to $\mathbb{Z}$. A curve E whose endomorphism ring contains additional isogenies is said to have *complex multiplication*. If E is defined over a finite field then E always has complex multiplication. The endomorphism ring of E is isomorphic to an order in either a quadratic number field or a quaternion algebra.

`MultiplicationByMMap(E, m): CrvEll, RngIntElt -> Map`

Returns the multiplication-by-m endomorphism $[m] : E \to E$ of the elliptic curve E, such that $[m](P) = m * P$.

`IdentityIsogeny(E): CrvEll -> Map`

Returns the identity map $E \to E$ of the elliptic curve E as an isogeny.

`IdentityMap(E): CrvEll -> Map`

Returns the identity map $E \to E$ of the elliptic curve E as an isomorphism. The defining coefficients are $[r, s, t, u] = [0, 0, 0, 1]$.

`NegationMap(E): CrvEll -> Map`

Returns the isomorphism of the elliptic curve E which maps P to $-P$, for all $P \in E$.

`FrobeniusMap(E, i): CrvEll, RngIntElt -> Map`

Returns the Frobenius isogeny $(x : y : 1) \mapsto (x^{p^i} : y^{p^i} : 1)$ of an elliptic curve E defined over a finite field of characteristic p.

`FrobeniusMap(E): CrvEll -> Map`

Equivalent to `FrobeniusMap(E, 1)`.

Example: Action of the Frobenius Map (ex-e6e186)

We check the action of the `FrobeniusMap` function.

```
> p := 23;
> FF1 := FiniteField(p);
> FF2 := FiniteField(p, 2);
> E1 := EllipticCurve([FF1 | 1, 3]);
> E2 := BaseExtend(E1, FF2);
> frob := FrobeniusMap(E2, 1);
> #{ E1!P : P in RationalPoints(E2) | P eq frob(P) } eq #E1;
true
```

131.8.5 Automorphisms

The functions in this section deal with automorphisms in the category of elliptic curves. For an elliptic curve E these are the automorphisms of the underlying curve that also preserve the group structure (equivalently, that send the identity O_E to itself).

Warning: The behaviour of these functions depends on the type of the input curve. For a general curve in Magma (an object that is a `Crv` but not a `CrvEll`) over a finite field, `AutomorphismGroup` and `Automorphisms` give automorphisms in the category of curves.

`AutomorphismGroup(E): CrvEll -> Grp, Map`

For an elliptic curve E over a general field K, this function determines the group of automorphisms of E that are defined over K. It returns an abstract group A (an abelian group or a polycyclic group), together with a map which sends elements of A to actual automorphisms of E (maps of schemes with domain and codomain E).

`Automorphisms(E): CrvEll -> SeqEnum`

For an elliptic curve E over a general field K, this function returns a sequence containing the elements of the `AutomorphismGroup` of E.

131.9 Operations on Points

Points on an elliptic curve over a field are given in terms of projective coordinates: A point (a, b, c) is equivalent to (x, y, z) if and only if there exists an element u (in the field of definition) such that $ua = x$, $ub = y$, and $uc = z$. The equivalence class of (x, y, z) is denoted by the projective point $(x : y : z)$. At least one of the projective coordinates must be nonzero. We call the coefficients normalised if either $z = 1$, or $z = 0$ and $y = 1$.

131.9.1 Creation of Points

In this section the descriptions will refer to a point set H, which is either the H in the signature or the base point set of the elliptic curve E in the signature.

`H ! [x, y, z]: SetPtEll, [ RngElt ] -> PtEll`

`elt< H | x, y, z >: SetPtEll, RngElt, RngElt, RngElt -> PtEll`

`E ! [x, y, z]: CrvEll, [ RngElt ] -> PtEll`

`elt< E | x, y, z >: CrvEll, RngElt, RngElt, RngElt -> PtEll`

Given a point set $H = E(R)$ and coefficients x, y, z in R satisfying the equation for E, return the normalised point $P = (x : y : z)$ in H. If z is not specified it is assumed to be 1.

`H ! 0: SetPtEll, RngIntElt -> PtEll`

`Id(H): SetPtEll -> PtEll`

`Identity(H): SetPtEll -> PtEll`

`E ! 0: CrvEll, RngIntElt -> PtEll`

`Id(E): CrvEll -> PtEll`

`Identity(E): CrvEll -> PtEll`

Returns the normalised identity point $(0 : 1 : 0)$ of the point set H on the elliptic curve E.

`Points(H, x): SetPtEll, RngElt -> [ PtEll ]`

`Points(E, x): CrvEll, RngElt -> [ PtEll ]`

Returns the sequence of points in the pointset H on the elliptic curve E whose x-coordinate is x.

`Points(H): SetPtEll -> {@ PtEll @}`

`RationalPoints(H): SetPtEll -> {@ PtEll @}`

`Points(E): CrvEll -> {@ PtEll @}`

`RationalPoints(E): CrvEll -> {@ PtEll @}`

```
Bound           : RngIntElt                 Default: 0
DenominatorBound: RngIntElt                 Default: Bound
Denominators    : Setq                      Default:
Max             : RngIntElt                 Default:
NPrimes         : RngIntElt                 Default: 30
```

Given an elliptic curve E (or associated point set) over the rationals, a number field, or a finite field, this function returns an indexed set of points on the curve (or in the point set). When over a finite field this will contain all the rational points.

When over $\mathbb{Q}$ or a number field, a positive value for the parameter `Bound` must be given. Over $\mathbb{Q}$ this refers to the height of the x-coordinate of the points. Over number fields the algorithm searches x-coordinates in some chosen box and with some chosen denominators depending on the `Bound` (so here, too, there is loosely a linear relationship between the `Bound` and the heights of the x-coordinates searched).

The other optional parameters are only for the number field case. The denominators of x-coordinates to be searched can be specified as `Denominators` (a set of integral elements in the field) or by setting `DenominatorBound` (an integer). If an integer `Max` is specified then the routine returns as soon as this many points are found. The parameter `NPrimes` is an internal parameter that controls the number of primes used in the sieving.

The algorithm uses a sieve method; the number field case is described in Appendix A of [Bru02]. In both cases the implementation of the sieving is reasonably fast.

`PointsAtInfinity(H): SetPtEll -> {@ PtEll @}`

`PointsAtInfinity(E): CrvEll -> {@ PtEll @}`

Returns the indexed set containing the identity point of the pointset H or on the elliptic curve E.

131.9.2 Creation Predicates

`IsPoint(H, S): SetPtEll, [ RngElt ] -> BoolElt, PtEll`

`IsPoint(E, S): CrvEll, [ RngElt ] -> BoolElt, PtEll`

Returns `true` if the sequence of values in S are the coordinates of a point in the pointset H or on the elliptic curve E, `false` otherwise. If this is `true` then the corresponding point is returned as the second value.

`IsPoint(H, x): SetPtEll, RngElt -> BoolElt, PtEll`

`IsPoint(E, x): CrvEll, RngElt -> BoolElt, PtEll`

Returns `true` if x is the x-coordinate of a point in the pointset H or on the elliptic curve E, `false` otherwise. If this is `true` then a corresponding point is returned as the second value. Note that the point at infinity of H will never be returned.

131.9.3 Access Operations

`P[i]: PtEll, RngIntElt -> RngElt`

Returns the i-th coefficient for an elliptic curve point P, for $1 \leq i \leq 3$.

`ElementToSequence(P): PtEll -> [ RngElt ]`

`Eltseq(P): PtEll -> [ RngElt ]`

Given a point P on an elliptic curve, this function returns a sequence of length 3 consisting of its coefficients (normalised).

131.9.4 Associated Structures

`Category(P): PtEll -> Cat`

`Type(P): PtEll -> Cat`

Given a point P on an elliptic curve, this function returns `PtEll`, the category of elliptic curve points.

`Parent(P): PtEll -> SetPtEll`

Given a point P on an elliptic curve, this function returns the parent point set for P.

`Scheme(P): SetPtEll -> CrvEll`

`Curve(P): SetPtEll -> CrvEll`

Given a point P on an elliptic curve, this function returns the corresponding scheme or elliptic curve for the parent point set of P.

131.9.5 Arithmetic

The points on an elliptic curve over a field form an abelian group, for which we use the additive notation. The identity element is the point $O = (0 : 1 : 0)$.

`- P: PtEll -> PtEll`

Returns the additive inverse of the point P on an elliptic curve E.

`P + Q: PtEll, PtEll -> PtEll`

Returns the sum $P + Q$ of two points P and Q on the same elliptic curve.

`P +:= Q: PtEll, PtEll`

Given two points P and Q on the same elliptic curve, sets P equal to their sum.

`P - Q: PtEll, PtEll -> PtEll`

Returns the difference $P - Q$ of two points P and Q on the same elliptic curve.

`P -:= Q: PtEll, PtEll`

Given two points P and Q on the same elliptic curve, sets P equal to their difference.

`n * P: RngIntElt, PtEll -> PtEll`

`P * n: PtEll, RngIntElt -> PtEll`

Returns the n-th multiple of the point P on an elliptic curve.

`P *:= n: PtEll, RngIntElt`

Sets the point P equal to the n-th multiple of itself.

131.9.6 Division Points

`P / n: PtEll, RngIntElt -> PtEll`

Given a point P on an elliptic curve E and an integer n, this function returns a point Q on E such that $P = nQ$, if such a point exists. If no such point exists then a runtime error results.

`P /:= n: PtEll, RngIntElt`

Given a point P on an elliptic curve E and an integer n, this function sets P equal to a point Q on E such that $P = nQ$, if such a point exists. If no such point exists then a runtime error results.

`DivisionPoints(P, n): PtEll, RngIntElt -> [ PtEll ]`

Given a point P on an elliptic curve E and an integer n, this function returns the sequence of all points Q on E such that $P = nQ$ holds. If there are no such points then an empty sequence is returned.

`IsDivisibleBy(P, n): PtEll, RngIntElt -> BoolElt, PtEll`

Given a point P on an elliptic curve E and an integer n, this function returns `true` if P is n-divisible, and if so, an n-division point. Otherwise `false` is returned.

Example: Three-Division Points (ex-8bfcbb)

```
> E := EllipticCurve([1, 0, 0, 12948, 421776]);
> P := E![ -65498304*1567, -872115836268, 1567^3 ];
> DivisionPoints(P, 3);
[ (312 : -6060 : 1), (-30 : -66 : 1), (216 : 3540 : 1) ]
```

Note that P has three three-division points — this tells us that there are three 3-torsion points in E. In fact, there are 9 points in the torsion subgroup.

```
> DivisionPoints(E!0, 9);
[ (0 : 1 : 0), (24 : 852 : 1), (24 : -876 : 1), (-24 : -300 : 1), (-24 : 324 :␣
↪1),
(600 : 14676 : 1), (600 : -15276 : 1), (132 : 2040 : 1), (132 : -2172 : 1) ]
```

Example: Reducing a Set of Generators (ex-c4b7a1)

We construct some points in a certain elliptic curve over $\mathbb{Q}$ and try by hand to find a "smaller" set of points that generate the same group.

```
> E := EllipticCurve([0, 0, 1, -7, 6]);
> P1 := E![ 175912024457 * 278846, -41450244419357361, 278846^3 ];
> P2 := E![ -151 * 8, -1845, 8^3 ];
> P3 := E![ 36773 * 41, -7036512, 41^3 ];
> P1; P2; P3;
(175912024457/77755091716 : -41450244419357361/21681696304639736 : 1)
(-151/64 : -1845/512 : 1)
(36773/1681 : -7036512/68921 : 1)
```

Now we try small linear combinations in the hope of finding nicer looking points. We shall omit the bad guesses and just show the good ones.

```
> P1 + P2;
(777/3364 : 322977/195112 : 1)
```

Success! We replace P1 with this new point and keep going.

```
> P1 +:= P2;
> P2 + P3;
(-3 : 0 : 1)
> P2 +:= P3;
```

(continues on next page)

(continued from previous page)

```
> P3 - P1;
(-1 : -4 : 1)
> P3 -:= P1;
> P1 - 2*P2;
(0 : 2 : 1)
> P1 -:= 2*P2;
> [ P1, P2, P3 ];
[ (0 : 2 : 1), (-3 : 0 : 1), (-1 : -4 : 1) ]
```

The pairwise reductions no longer help, but there is a smaller point with x-coordinate 1:

```
> IsPoint(E, 1);
true (1 : 0 : 1)
```

After a small search we find:

```
> P1 - P2 - P3;
(1 : 0 : 1)
> P2 := P1 - P2 - P3;
> [ P1, P2, P3 ];
[ (0 : 2 : 1), (1 : 0 : 1), (-1 : -4 : 1) ]
```

Using a naive definition of "small" these are the smallest possible points. (Note that there are points of smaller canonical height.) These points are in fact the generators of the Mordell–Weil group for this particular elliptic curve. Since none of the transformations changed the size of the space spanned by the points it follows that the original set of points are also generators of E. However, the reduced points form a much more convenient basis.

Example: Generic Point on an Elliptic Surface `(ex-10e418)`

We construct an elliptic curve over a function field (hence an elliptic surface) and form a "generic" point on it. First, we construct the function field.

```
> E := EllipticCurve([GF(97) | 1, 2]);
> K<x, y> := FunctionField(E);
```

Now we lift the curve to be over its own function field and form a generic point on E.

```
> EK := BaseChange(E, K);
> P := EK![x, y, 1];
> P;
(x : y : 1)
> 2*P;
((73*x^4 + 48*x^2 + 93*x + 73)/(x^3 + x + 2) : (85*x^6 + 37*x^4 + 5*x^3
    + 60*x^2 + 96*x + 8)/(x^6 + 2*x^4 + 4*x^3 + x^2 + 4*x + 4)*y : 1)
```

Finally, we verify that addition of the generic point defines the addition law on the curve.

```
> m2 := MultiplicationByMMap(E, 2);
> P := E![ 32, 93, 1 ];
> m2(P);
(95 : 63 : 1)
> 2*P;
(95 : 63 : 1)
```

131.9.7 Point Order

`Order(P): PtEll -> RngIntElt`

Given a point on an elliptic curve defined over $\mathbb{Q}$ or a finite field, this function computes the order of P; that is, the smallest positive integer n such that $n \cdot P = O$ on the curve. If no such positive n exists then 0 is returned to indicate infinite order. If the curve is defined over a finite field then the order of the curve will first be computed.

`FactoredOrder(P): PtEll -> RngIntElt`

Given a point on an elliptic curve defined over $\mathbb{Q}$ or over a finite field, this function returns the factorisation of the order of P. If the curve is over a finite field then on repeated applications this is generally much faster than factorising `Order(P)` because the factorisation of the order of the curve will be computed and stored. An error ensues if the curve is defined over $\mathbb{Q}$ and P has infinite order.

Example: Points over a Large Finite Field (ex-712399)

We show a few simple operations with points on an elliptic curve over a large finite field.

```
> E := EllipticCurve([GF(NextPrime(10^12)) | 1, 1]);
> Order(E);
1000001795702
> FactoredOrder(E);
[ <2, 1>, <7, 1>, <13, 1>, <19, 1>, <31, 1>, <43, 1>, <59, 1>, <3677, 1> ]
> P := E ! [652834414164, 320964687531, 1];
> P;
(652834414164 : 320964687531 : 1)
> IsOrder(P, Order(E));
true
> FactoredOrder(P);
[ <2, 1>, <7, 1>, <13, 1>, <19, 1>, <31, 1>, <43, 1>, <59, 1>, <3677, 1> ]
> FactoredOrder(3677 * 59 * P);
[ <2, 1>, <7, 1>, <13, 1>, <19, 1>, <31, 1>, <43, 1> ]
```

131.9.8 Predicates on Points

`IsId(P): PtEll -> BoolElt`

`IsIdentity(P): PtEll -> BoolElt`

`IsZero(P): PtEll -> BoolElt`

Returns `true` if and only if the point P is the identity point of its point set, `false` otherwise.

`P eq Q: PtEll, PtEll -> BoolElt`

Returns `true` if and only if P and Q are points on the same elliptic curve and have the same normalised coordinates.

`P ne Q: PtEll, PtEll -> BoolElt`

The logical negation of `eq`.

`P in H: PtEll, SetPtEll -> BoolElt`

Given a point P, return `true` if and only if P is in the point set H. That is, it satisfies the equation of E and its coordinates lie in R, where $H = E(R)$.

`P in E: PtEll, CrvEll -> BoolElt`

Given a point P, return `true` if and only if P is on the elliptic curve E (i.e., satisfies its defining equation). Note that this is an exception to the general rule, in that P does not have to lie in the base point set of E for this to be true.

`IsOrder(P, m): PtEll, RngIntElt -> BoolElt`

Returns `true` if and only if the point P has order m. If you believe that you know the order of the point then this intrinsic is likely to be much faster than just calling `Order`.

`IsIntegral(P): PtEll -> BoolElt`

Given a point P on an elliptic curve defined over $\mathbb{Q}$, this function returns `true` if and only if the coordinates of the (normalisation of) P are integers.

`IsSIntegral(P, S): PtEll, SeqEnum -> BoolElt`

Given a point P on an elliptic curve defined over $\mathbb{Q}$ and a sequence S of primes, this function returns `true` if and only if the coordinates of the (normalisation of) P are S-integers. That is, the denominators of $x(P)$ and $y(P)$ are only supported by primes of S.

Example: Integral and S-Integral Points (ex-e572c3)

We find some integral and S-integral points on a well-known elliptic curve.

```
> E := EllipticCurve([0, 17]);
> P1 := E![ -2, 3 ];
> P2 := E![ -1, 4 ];
> S := [ a*P1 + b*P2 : a,b in [-3..3] ];
> #S;
49
> [ P : P in S | IsIntegral(P) ];
[ (43 : -282 : 1), (5234 : -378661 : 1), (2 : -5 : 1), (8 : 23 : 1),
(4 : 9 : 1), (-2 : -3 : 1), (52 : -375 : 1), (-1 : -4 : 1), (-1 : 4 : 1),
(52 : 375 : 1), (-2 : 3 : 1), (4 : -9 : 1), (8 : -23 : 1), (2 : 5 : 1),
(5234 : 378661 : 1), (43 : 282 : 1) ]
> [ P : P in S | IsSIntegral(P, [2, 3]) and not IsIntegral(P) ];
[ (1/4 : 33/8 : 1), (-8/9 : 109/27 : 1), (-206/81 : -541/729 : 1),
(137/64 : 2651/512 : 1), (137/64 : -2651/512 : 1), (-206/81 : 541/729 : 1),
(-8/9 : -109/27 : 1), (1/4 : -33/8 : 1) ]
```

131.9.9 Weil Pairing

Magma contains an optimised implementation of the Weil pairing on an elliptic curve. This function is used in the computation of the group structure of elliptic curves over finite fields, making the determination of the group structure efficient.

`WeilPairing(P, Q, n): PtEll, PtEll, RngIntElt -> RngElt`

Given n-torsion points P and Q of an elliptic curve E, this function computes the Weil pairing of P and Q.

`IsLinearlyIndependent(S, n): [ PtEll ], RngIntElt -> BoolElt`

Given a sequence S of points of an elliptic curve E such that each point has order dividing n, this function returns `true` if and only if the points in S are linearly independent over $\mathbb{Z}/n\mathbb{Z}$.

`IsLinearlyIndependent(P, Q, n): PtEll, PtEll, RngIntElt -> BoolElt`

Given points P and Q of an elliptic curve E, this function returns `true` if and only if P and Q form a basis of the n-torsion points of E.

Example: Weil Pairing of 3-Torsion Points (ex-c9a88c)

We compute the Weil pairing of two 3-torsion points on the curve $y^2 = x^3 - 3$ over $\mathbb{Q}$.

```
> E := EllipticCurve([0, -3]);
> E;
> P1, P2, P3 := Explode(ThreeTorsionPoints(E));
> P1;
(0 : 2*zeta_3 + 1 : 1)
> Parent(P1);
Set of points of E with coordinates in Cyclotomic Field
  of order 3 and degree 2
> Parent(P2);
```

(continues on next page)

(continued from previous page)

```
Set of points of E with coordinates in Number Field with
  defining polynomial X^3 - 18*X - 30 over the Rational Field
```

In order to take the Weil pairing of two points we need to coerce them into the same point set. This turns out to be a point set over a number field K of degree 6.

```
> Qmu3 := Ring(Parent(P1));
> K<w> := CompositeFields(Ring(Parent(P1)), Ring(Parent(P2)))[1];
> wp := WeilPairing( E(K)!P1, E(K)!P2, 3 );
> wp;
1/975*(14*w^5 + 5*w^4 - 410*w^3 - 620*w^2 + 3964*w + 8744)
> Qmu3!wp;
zeta_3
```

131.10 Bibliography

- [Bru02] N. R. Bruin. *Chabauty methods and covering techniques applied to generalized Fermat equations*. Volume 133 of CWI Tract. Stichting Mathematisch Centrum Centrum voor Wiskunde en Informatica, Amsterdam, 2002. ISBN 90-6196-508-X. Dissertation, University of Leiden, Leiden, 1999.
- [Cas91] J. W. S. Cassels. *Lectures on elliptic curves*. Volume 24 of London Mathematical Society Student Texts. Cambridge University Press, Cambridge, 1991. ISBN 0-521-41517-9; 0-521-42530-1.
- [Con99] I. Connell. *Elliptic Curve Handbook*. McGill University, 1999.
- [Cre97] J. E. Cremona. *Algorithms for modular elliptic curves*. Cambridge University Press, Cambridge, second edition, 1997. ISBN 0-521-59820-6.
- [Nag28] Trygve Nagell. Sur les propriétés arithmétiques des cubiques planes du premier genre. *Acta Math.*, 52:93–126, 1928.
- [Sil86] J. Silverman. *The arithmetic of elliptic curves*. Volume 106 of Graduate Texts in Mathematics. Springer-Verlag, New York, 1986.

Chapter 132

Elliptic Curves over Finite Fields

This chapter describes the specialised facilities for elliptic curves defined over finite fields. Details concerning their construction, arithmetic, and basic properties may be found in Chapter *Elliptic Curves*. Most of the machinery has been constructed with Elliptic Curve Cryptography in mind.

The first major group of intrinsics relate to the determination of the order of the group of rational points of an elliptic curve over a large finite field. A variety of canonical lift algorithms are provided for characteristic 2 fields while the SEA algorithm is used for fields having characteristic greater than 2.

A function for computing the Weil pairing forms the basis of the MOV reduction of the discrete logarithm problem (DLP) for a supersingular elliptic curve to a DLP in a finite field. A second type of attack on the DLP is based on the use of Weil descent. Tools implementing a generalisation of the GHS attack for ordinary curves in characteristic 2 are provided.

Finally, for a direct attack on the DLP for elliptic curves, a parallel collision search version of the Pollard rho algorithm is available.

132.1 Supersingular Curves

`IsSupersingular(E : parameters): CrvEll -> BoolElt`

`Proof: BoolElt`	`Default: true`

Given an elliptic curve E over a finite field, this function returns `false` if E is ordinary, otherwise proves that E is supersingular and returns `true`. If the parameter `Proof` is set to `false` then the effect of the function is the same as that of `IsProbablySupersingular`.

`SupersingularPolynomial(p): RngIntElt -> RngUPolElt`

Given a prime p, returns the separable monic polynomial over $\mathbb{F}_p$ whose roots are precisely the j invariants of supersingular elliptic curves in characteristic p. The polynomial is computed by a formula; ignoring factors corresponding to $j = 0$ or 1728, it is the partial power-series expansion of a certain hypergeometric function reduced mod p. For p of moderate size this is a very fast method.

`IsOrdinary(E): CrvEll -> BoolElt`

Given an elliptic curve E over a finite field, this function returns `true` if the elliptic curve E is ordinary, otherwise `false` (i.e., if it is supersingular). Thus, this function is the logical negation of `IsSupersingular`.

`IsProbablySupersingular(E): CrvEll -> BoolElt`

Given an elliptic curve E over a finite field, this function returns `false` if the elliptic curve E is proved to be ordinary, otherwise `true`. The algorithm is nondeterministic and repeated tests are independent.

132.2 The Order of the Group of Points

132.2.1 Point Counting

Magma contains an efficient implementation of the Schoof–Elkies–Atkin (SEA) algorithm, with Lercier's extension to base fields of characteristic 2, for computing the number of points on an elliptic curve over a finite field. There are also implementations of the faster p-adic canonical lift methods and the p-adic deformation method in small characteristic. Calculations are performed in the smallest field over which the curve is defined, and the result is lifted to the original field.

Like all group functions on elliptic curves, these intrinsics really apply to a particular point set; the curve is identified with its base point set for the purposes of these functions. To aid exposition only the versions that take the curves are shown but an appropriate point set (over a finite field) may be used instead.

`# H: SetPtEll -> RngIntElt`

`# E: CrvEll -> RngIntElt`

`Order(H): SetPtEll -> RngIntElt`

`Order(E): CrvEll -> RngIntElt`

The order of the group of K-rational points of E, where E is an elliptic curve defined over the finite field K.

`FactoredOrder(H): SetPtEll -> RngIntElt`

`FactoredOrder(E): CrvEll -> RngIntElt`

This function returns the factorisation of the order of the group of K-rational points of E, where E is an elliptic curve defined over the finite field K. This factorisation is stored on E and is reused when computing the order of points on E.

`SEA(H : parameters): SetPtEll -> RngIntElt`

`SEA(E : parameters): CrvEll -> RngIntElt`

```
Al         : MonStgElt                    Default: "Default"
MaxSmooth  : RngIntElt                    Default: Infinity()
AbortLevel: RngIntElt                     Default: 0
UseSEA     : BoolElt                      Default: false
```

This is the internal algorithm used by the point counting routines, but it can be invoked directly on an elliptic curve E or a point set H if desired. The most obvious reason to do this is because of the parameters `AbortLevel` and `MaxSmooth`, which allow the computation to be stopped early if it can be shown not to satisfy certain conditions (such as the order of the group being prime).

Warning: Unlike the external functions which call it (such as `Order` and `FactoredOrder`), `SEA` does not store the computed group order back on the curve itself and so functions which need the group order will have to recompute it.

The parameter `Al` can be set to one of `"Enumerate"`, `"BSGS"`, or `"Default"`. Setting `Al` to `"Enumerate"` causes the group order to be found by enumeration of all points on the curve and hence is only practical for small orders. Setting `Al` to `"BSGS"` is currently equivalent to setting `Al` to `"Default"`, which uses point enumeration for suitably small finite fields, the full Schoof–Elkies–Atkin algorithm if the characteristic is not small, and canonical lift methods or deformation if it is. Currently, small characteristic means $p < 1000$. For $p < 37$ or $p \in \{41, 47, 59, 71\}$, canonical lift is used; deformation is the chosen method for other small p.

The canonical lift methods use a range of techniques to lift a modular parameter of the curve to an unramified p-adic ring. If a Gaussian Normal Basis of type ≤ 2 exists for that ring (see `HasGNB(R, n, t): RngPad, RngIntElt, RngIntElt → BoolElt`) then the method of Lercier and Lubicz from [LL03] is used with some local adaptations by Michael Harrison to increase the speed for fields of moderate size. Otherwise, the MSST algorithm as described in [Gau02] is used for fields up to a certain size with Harley's recursive version ([Hara]) taking over for larger fields.

For $p < 10$ and $p = 13$ the modular parameter used is a generator of the function field of a genus 0 modular curve $X_0(p^r)$. The parameter gives a particularly nice modular polynomial Φ for use in the above-mentioned iterative lifting processes (see [Koh03]). In these cases Φ has a relatively simple factored form and evaluations are hand-coded for extra efficiency.

For p where $X_0(p)$ is hyperelliptic (excepting the anomalous case $p = 37$) we use a new method, developed by Michael Harrison, which lifts a *pair* of modular parameters corresponding to the x and y coordinates in a simple Weierstrass model of $X_0(p)$.

The deformation method uses code designed and implemented by Hendrik Hubrechts. It is based on ideas of Dwork, first used in the design of explicit computational algorithms by Alan Lauder, and works by computing the deformation over a family of curves of the Frobenius action on rigid cohomology by solving a certain differential equation p-adically. Details can be found in [Hub07]. The method is used for E over $\mathbb{F}_{p^n}$ with $p < 1000$ for which the canonical lift method is not used and for n greater than a smallish bound that increases slowly with p. For such p and smaller n, either enumeration or small characteristic SEA is used.

To use Schoof–Elkies–Atkin in small characteristic (with Lercier's improvements) instead of canonical lift or deformation, set `UseSEA` to `true`.

The parameter `MaxSmooth` can be used to specify a limit on the number of small primes that divide the group order. That is, we will consider the group order to be "smooth" if it is divisible by the product of small primes and this product is larger than the value of `MaxSmooth`. Then the possible values of `AbortLevel` have the following meanings:

- 0 : Abort if the curve has smooth order.
- 1 : Abort if both the curve and its twist have smooth order.
- 2 : Abort if either the curve or its twist have smooth order.

If the early abort is triggered then the returned group order is 0.

One common use of these parameters is when searching for a curve with prime order. Setting `MaxSmooth := 1` will cause the early termination of the algorithm if a small factor of the order is found. If, however, the algorithm did not abort then the group order is not necessarily prime — this just means that no small prime dividing the order was encountered during the computation.

Note that the canonical lift methods give no intermediate data on the order of E, so `MaxSmooth` and `AbortLevel` have no effect when these methods are used.

Example: SEA Point Counting in Characteristic 2 (ex-05153b)

The first examples in characteristic 2 illustrate the increased speed when we work with fields that have GNBs available:

```
> //example with no Gaussian Normal Basis (GNB)
> K<w> := FiniteField(2, 160); // finite field of size 2^160
> f<x> := MinimalPolynomial(w); f;
x^160 + x^5 + x^3 + x^2 + 1
> E := EllipticCurve([K| 1, 0, 0, 0, w]);
> time #E;
1461501637330902918203686141511652467686942715904
Time: 0.050
> //example with a GNB of Type 1
> HasGNB(pAdicRing(2, 1), 162, 1);
true
> K<w> := FiniteField(2, 162); // finite field of size 2^162
> f<x> := MinimalPolynomial(w); f;
x^162 + x^7 + x^6 + x^5 + x^2 + x + 1
> E := EllipticCurve([K| 1, 0, 0, 0, w]);
> time #E;
```

(continues on next page)

(continued from previous page)

```
584600654932361167281474262668517478408555834163 2
Time: 0.010
> //example with a GNB of Type 2
> K<w> := FiniteField(2, 173); // finite field of size 2^173
> f<x> := MinimalPolynomial(w); f;
x^173 + x^8 + x^5 + x^2 + 1
> E := EllipticCurve([K| 1, 0, 0, 0, w]);
> time #E;
119726214130147567059245860576499719239117422020567 04
Time: 0.030
```

and here is an example in characteristic 3 (with type 1 GNB)

```
> F := FiniteField(3, 100);
> j := Random(F);
> E := EllipticCurveWithjInvariant(j);
> time #E;
515377520732011331036461505619702343613256090042
Time: 0.010
```

Finally (for small characteristic), a case with $p = 37$ where the deformation method is used.

```
> F := FiniteField(37, 30);
> j := Random(F);
> E := EllipticCurveWithjInvariant(j);
> time #E;
111186413610993811950186296693286649955782417767
Time: 0.700
```

Here we can see the early abort feature in action:

```
> p := NextPrime(10^9);
> p;
1000000007
> K := GF(p);
> E := EllipticCurve([K | -1, 1]);
> time SEA(E : MaxSmooth := 1);
0
Time: 0.010
> time #E;
1000009476
Time: 0.070
```

For curves this small the amount of time saved by an early abort is fairly small, but for curves over large fields the savings can be quite significant. As mentioned above, even when SEA does not abort there is no guarantee that the curve has prime order:

```
> E := EllipticCurve([K | 0, 1]);
> time SEA(E : MaxSmooth := 1);
1000000008
> IsSupersingular(E);
true
> E := EllipticCurve([K | -30, 1]);
> time SEA(E : MaxSmooth := 1);
1000035283
> Factorization($1);
[ <13, 1>, <709, 1>, <108499, 1> ]
```

In the first case the curve was supersingular and so the order was easily computed directly; thus no early abort was attempted. The latter case shows that small primes may not be looked at even when the curve is ordinary.

Next we find a curve with prime order:

```
> for k in [1..p] do
>   E := EllipticCurve([K | k, 1]);
>   n := SEA(E : MaxSmooth := 1);
>   if IsPrime(n) then
>     printf "Found curve of prime order %o for k = %o\\n", n, k;
>     break;
>   end if;
> end for;
Found curve of prime order 999998501 for k = 29
> E;
Elliptic Curve defined by y^2 = x^3 + 29*x + 1 over GF(1000000007)
> #E;
999998501
```

SetVerbose("SEA", v): MonStgElt, RngIntElt

This procedure changes the verbose printing level for the SEA algorithm which counts the number of points on an elliptic curve. Currently the legal values for v are `false`, `true`, or an integer between 0 and 5 (inclusive), where `false` is equivalent to 0 and `true` is equivalent to 1. A nonzero value gives information during the course of the algorithm, increasing in verbosity for higher values of v.

N.B. The previous name for this verbose flag, `"ECPointCount"` is now deprecated and will be removed in a future release. It should continue to function in this release, but its verbose levels are different to those of `"SEA"`.

Order(H, r): SetPtEll, RngIntElt -> RngIntElt

Order(E, r): CrvEll, RngIntElt -> RngIntElt

Returns the order of the elliptic curve E over the extension field of K of degree r, where K is the base field of E. The order is found by calculating the order of E over K and lifting.

Trace(H): SetPtEll -> RngIntElt

Trace(E): CrvEll -> RngIntElt

TraceOfFrobenius(E): CrvEll -> RngIntElt

Returns the trace of the Frobenius endomorphism on the elliptic curve E, equal to $q+1-n$ where q is the cardinality of the coefficient field and n is the order of the group of rational points of E.

Trace(H, r): SetPtEll, RngIntElt -> RngIntElt

Trace(E, r): CrvEll, RngIntElt -> RngIntElt

TraceOfFrobenius(H, r): SetPtEll, RngIntElt -> RngIntElt

TraceOfFrobenius(E, r): CrvEll, RngIntElt -> RngIntElt

Returns the trace of the r-th power Frobenius endomorphism, where E is an elliptic curve over a finite field.

Example: Group Order and Trace of Frobenius (ex-e97387)

The computation of the order of a curve over a finite field invokes the SEA algorithm. This data determines the number of points over all finite extensions, and is equivalent to the data for the trace of Frobenius. The values of the trace of Frobenius and group order over all extensions are therefore a trivial computation.

```
> K<w> := GF(2^133);
> E := EllipticCurve([K| 1, 0, 0, 0, w]);
> time #E;
```

(continues on next page)

(continued from previous page)

```
1088903574147003083086942872086986821888O
Time: 8.830
> FactoredOrder(E);
[ <2, 9>, <3, 4>, <5, 1>, <29, 1>, <1103, 1>, <3793, 1>, <16855669, 1>,
  <25678053050475964927, 1> ]
> time TraceOfFrobenius(E);
-41441283053285452287
Time: 0.000
> time TraceOfFrobenius(E, 3);
1282596408769540337607822605365889523499049843333311465783809
Time: 0.000
```

Example: Twists and Trace of Frobenius (ex-94b05b)

The following code demonstrates the computation of twists of an elliptic curve over a finite field, and the relationship with the trace of Frobenius.

```
> E := EllipticCurveFromjInvariant(GF(101^2)!0);
> Twists(E);
[
   Elliptic Curve defined by y^2 = x^3 + 1 over GF(101^2),
   Elliptic Curve defined by y^2 = x^3 + (61*w + 91) over GF(101^2),
   Elliptic Curve defined by y^2 = x^3 + (98*w + 16) over GF(101^2),
   Elliptic Curve defined by y^2 = x^3 + (9*w + 66) over GF(101^2),
   Elliptic Curve defined by y^2 = x^3 + (59*w + 76) over GF(101^2),
   Elliptic Curve defined by y^2 = x^3 + (87*w + 81) over GF(101^2)
]
> IsSupersingular(E);
true
```

Since E is supersingular, so are the twists of E. Hence the traces are divisible by the prime:

```
> [ TraceOfFrobenius(F) : F in Twists(E) ];
[ -202, -101, 101, 202, 101, -101 ]
> [ TraceOfFrobenius(F) : F in QuadraticTwists(E) ];
[ -202, 202 ]
```

We see that the traces come in pairs; the trace of a curve is the negative of the trace of its quadratic twist. This is not just true of the supersingular curves:

```
> E := EllipticCurveFromjInvariant(GF(101^2)!12^3);
> Twists(E);
[
   Elliptic Curve defined by y^2 = x^3 + x over GF(101^2),
   Elliptic Curve defined by y^2 = x^3 + (40*w + 53)*x over GF(101^2),
   Elliptic Curve defined by y^2 = x^3 + (3*w + 99)*x over GF(101^2),
   Elliptic Curve defined by y^2 = x^3 + (92*w + 19)*x over GF(101^2)
]
> IsSupersingular(E);
false
> [ TraceOfFrobenius(F) : F in Twists(E) ];
[ -198, 40, 198, -40 ]
> [ TraceOfFrobenius(F) : F in QuadraticTwists(E) ];
[ -198, 198 ]
```

132.2.2 Zeta Functions

ZetaFunction(E): CrvEll -> FldFunRatUElt

Given an elliptic curve E over a finite field, the function returns the zeta function of E as a rational function in one variable.

Note that each of the function calls **Order(E)**, **Trace(E)**, and **ZetaFunction(E)** invoke the same call to the SEA algorithm and determine equivalent data.

Example: Zeta Function of an Elliptic Curve (ex-7a0b38)

The zeta function $\zeta_E(t)$ of an elliptic curve E over a finite field $\mathbb{F}_q$ is a rational function whose logarithmic derivative has the power series expansion:

$$\frac{d \log \zeta_E(t)}{d \log t} = \sum_{n=1}^{\infty} |E(F_{q^n})|\, t^n$$

The following example illustrates this property of $\zeta_E(t)$.

```
> p := 11;
> E := EllipticCurve([GF(p) | 1, 2]);
> Z := ZetaFunction(E);
> Q<t> := LaurentSeriesRing(Rationals());
> t*Derivative(Log(Evaluate(Z, t))) + O(t^7);
16*t + 128*t^2 + 1264*t^3 + 14848*t^4 + 160976*t^5 + 1769600*t^6 + O(t^7)
> [ Order(E, n) : n in [1..6] ];
[ 16, 128, 1264, 14848, 160976, 1769600 ]
```

132.3 Enumeration of Points

In this section the descriptions will refer to a point set H, which is either the H in the signature or the base point set of the elliptic curve E in the signature.

Points(E): CrvEll -> {@ PtEll @}

Points(H): SetPtEll -> {@ PtEll @}

RationalPoints(E): CrvEll -> {@ PtEll @}

RationalPoints(H): SetPtEll -> {@ PtEll @}

Returns the set of rational points of the point set H or of the base point set of the elliptic curve E, including the point at infinity.

Random(E): CrvEll -> PtEll

Random(H): SetPtEll -> PtEll

Returns a random rational point of the point set H or of the base point set of the elliptic curve E. Every rational point has a roughly equal chance of being selected, including the point at infinity.

132.4 Abelian Group Structure

Like all group functions on elliptic curves, these intrinsics really apply to a particular point set; the curve is identified with its base point set for the purposes of these functions. To aid exposition only the versions that take the curves are shown but an appropriate point set over a finite field may be used instead.

AbelianGroup(H): SetPtEll -> GrpAb, Map

AbelianGroup(E): CrvEll -> GrpAb, Map

TorsionSubgroup(H): SetPtEll -> GrpAb, Map

`TorsionSubgroup(E): CrvEll -> GrpAb, Map`

Computes the abelian group isomorphic to the group of rational points on the elliptic curve E over a finite field. The function returns two values: an abelian group A and a map m from A to E. The map m provides an isomorphism between the abstract group A and the group of rational points on the curve.

`Generators(H): SetPtEll -> [ PtEll ]`

`Generators(E): CrvEll -> [ PtEll ]`

Given an elliptic curve E defined over a finite field or a pointset H of E, this function returns generators for the group of points of E. The i-th element of the sequence corresponds to the i-th generator of the group as returned by the function `AbelianGroup`.

`NumberOfGenerators(H): SetPtEll -> RngIntElt`

`NumberOfGenerators(E): CrvEll -> RngIntElt`

`Ngens(H): SetPtEll -> RngIntElt`

`Ngens(E): CrvEll -> RngIntElt`

The number of generators of the group of rational points of (the point set H of) the elliptic curve E; this is simply the length of the sequence returned by `Generators(E)`.

Example: Abelian Group of Points (ex-04fd99)

A simple example of some of the above calls in action:

```
> FF<w> := GF(1048583, 2);
> E := EllipticCurve([ 1016345*w + 272405, 660960*w + 830962 ]);
> A, m := AbelianGroup(E);
> A;
Abelian Group isomorphic to Z/3 + Z/366508289334
Defined on 2 generators
Relations:
  3*A.1 = 0
  366508289334*A.2 = 0
> S := Generators(E);
> S;
[ (191389*w + 49138 : 878749*w + 1008891 : 1), (852793*w + 24192 :
    376202*w + 48552 : 1) ]
> S eq [ m(A.1), m(A.2) ];
true
```

132.5 Pairings on Elliptic Curves

Pairings on elliptic curves over finite fields have many uses, ranging from checking independence of torsion points to more practical applications in cryptography. Both destructive applications (such as the MOV attack) and constructive applications (such as pairing-based cryptography) are worth mentioning. Magma now contains an implementation of the most popular pairings on elliptic curves over finite fields, including the Weil pairing, the Tate pairing, and various versions of the Eta and Ate pairings.

All pairings evaluate what is now called a Miller function, denoted by $f_{n,P}$ and defined as any function on E with divisor $n(P) - ([n]P) - (n-1)\infty$, where ∞ is the point at infinity of the curve.

132.5.1 Weil Pairing

The Weil pairing is a non-degenerate bilinear map from $E[n] \times E[n]$ to μ_n, the n-th roots of unity. The Weil pairing $w_n(P, Q)$ is computed as $(-1)^n f_{n,P}(Q)/f_{n,Q}(P)$.

`WeilPairing(P, Q, n): PtEll, PtEll, RngIntElt -> RngElt`

Given n-torsion points P and Q on an elliptic curve E, this function computes the Weil pairing of P and Q. Both points need to be in the same point set.

132.5.2 Tate Pairing

The Tate pairing (sometimes called the Tate–Lichtenbaum pairing) is a non-degenerate bilinear map from $E[n] \times E/nE$ into $K^*/(K^*)^n$, where K is a finite field containing the n-th roots of unity. In practice one often works with the reduced version of the pairing by mapping into μ_n using the final exponentiation; i.e., powering by $\#K^*/n$. The Tate pairing $t_n(P, Q)$ is computed as $f_{n,P}(Q)$.

`TatePairing(P, Q, n): PtEll, PtEll, RngIntElt -> RngElt`

Given an n-torsion point P and a point Q on an elliptic curve, this function computes the Tate pairing $t_n(P, Q)$ by returning a random representative of the coset. Both points need to be in the same point set; the field K is the field of definition of this point set.

`ReducedTatePairing(P, Q, n): PtEll, PtEll, RngIntElt -> RngElt`

Given an n-torsion point P and a point Q on an elliptic curve, this function computes the reduced Tate pairing $e_n(P, Q) = t_n(P, Q)^{(\#K-1)/n}$. Both points need to be in the same point set; the field K is the field of definition of this point set and n should divide $\#K - 1$.

132.5.3 Eta Pairing

The Eta pairing is only defined on supersingular curves and can be considered as an optimised version of the Tate pairing. It works as follows.

Let E be a supersingular elliptic curve defined over $\mathbb{F}_q$ and $n|\#E(\mathbb{F}_q)$. The security multiplier k is the smallest positive integer such that $q^k \equiv 1 \bmod n$. Here, in the supersingular case, k divides 6. Now, $E[n] \subseteq E(\mathbb{F}_{q^k})$ is a free $\mathbb{Z}/n\mathbb{Z}$-module of rank 2 which splits into the direct sum of 2 rank one eigenspaces under the action of q-Frobenius (denoted by F in the following), with eigenvalues 1 and q (as long as $(n, k) = 1$ and $k > 1$).

The Eta pairing is defined on the subgroup of $E[n] \times E[n]$ given by the product of the two F-eigenspaces $G_1 \times G_2$, where $P \in G_1$ iff $F(P) = P$ and $Q \in G_2$ iff $F(Q) = [q]Q$. G_1 is just $E(\mathbb{F}_q)[n]$. If R is an arbitrary n-torsion point in $E(\mathbb{F}_{q^k})$, k times its G_1-component is just the F-trace of R. This can be used to find non-trivial points in G_2 (see the example below).

There are three versions of the Eta pairing: one unreduced and two reduced versions. The Eta pairing $e_T(P, Q)$ is defined as $f_{T,P}(Q)$ where $T = t - 1$ and $\#E(\mathbb{F}_q) = q + 1 - t$ or $T = q$.

As for the Tate pairing, the unreduced version takes values in $K^*/(K^*)^n$ and the reduced versions in $\mu_n(K)$ where K is $\mathbb{F}_{q^k}$.

`EtaTPairing(P, Q, n, q): PtEll, PtEll, RngIntElt, RngIntElt -> RngElt`

```
CheckCurve : BoolElt                    Default: false
CheckPoints: BoolElt                    Default: false
```

Given a supersingular elliptic curve $E/\mathbb{F}_q$ and two points P, Q in $E[n]$ such that $P = F(P)$ and $F(Q) = [q]Q$ with F the q-power Frobenius, this function computes the Eta pairing with $T = t - 1$ where $\#E(\mathbb{F}_q) = q + 1 - t$. The result is non-reduced and thus a random representative of a whole coset. Both points need to be in the same point set $E(\mathbb{F}_{q^k})$ and q should be the size of the base field.

`ReducedEtaTPairing(P, Q, n, q): PtEll, PtEll, RngIntElt, RngIntElt -> RngElt`

```
CheckCurve : BoolElt                    Default: false
CheckPoints: BoolElt                    Default: false
```

The reduced version of the `EtaTPairing` function.

`EtaqPairing(P, Q, n, q): PtEll, PtEll, RngIntElt, RngIntElt -> RngElt`

```
CheckCurve : BoolElt                    Default: false
CheckPoints: BoolElt                    Default: false
```

Given a supersingular elliptic curve $E/\mathbb{F}_q$, two points P, Q in $E[n]$ such that $P = F(P)$ and $F(Q) = [q]Q$ with F the q-power Frobenius, this function computes the Eta pairing with $T = q$. This pairing is automatically reduced: It maps directly into n-th roots of unity. Both points need to be in the same point set $E(\mathbb{F}_{q^k})$ and q should be the size of the base field.

In the previous intrinsics one can explicitly check that the curve is supersingular by setting the parameter `CheckCurve` to `true`, and that the input points are indeed in the eigenspaces of Frobenius by setting the parameter `CheckPoints` to `true`. By default Magma does not perform these checks for efficiency reasons.

132.5.4 Ate Pairing

The Ate pairing generalises the Eta pairing to all elliptic curves, but is defined on $G_2 \times G_1$. i.e., the arguments are swapped with respect to the Eta pairing. Like the Eta pairing there are three versions, one unreduced and two reduced. The Ate pairing $a_T(Q, P)$ with $Q \in G_2$ and $P \in G_1$ is defined as $f_{T,Q}(P)$, where $T = t - 1$ and $\#E(\mathbb{F}_q) = q + 1 - t$ or $T = q$.

`AteTPairing(Q, P, n, q): PtEll, PtEll, RngIntElt, RngIntElt -> RngElt`

```
CheckPoints: BoolElt                    Default: false
```

Given two points Q, P in $E[n]$ such that $F(Q) = [q]Q$ and $P = F(P)$ with F the q-power Frobenius, this function computes the Ate pairing with $T = t - 1$ where $\#E(\mathbb{F}_q) = q + 1 - t$. The result is non-reduced and thus a random representative of a whole coset. Both points need to be in the same point set $E(\mathbb{F}_{q^k})$ and q should be the size of the base field.

`ReducedAteTPairing(Q, P, n, q): PtEll, PtEll, RngIntElt, RngIntElt -> RngElt`

```
CheckPoints: BoolElt                    Default: false
```

The reduced version of the `AteTPairing` function.

`AteqPairing(P, Q, m, q): PtEll, PtEll, RngIntElt, RngIntElt -> RngElt`

```
CheckPoints: BoolElt                    Default: false
```

Given two points Q, P in $E[n]$ such that $F(Q) = [q]Q$ and $P = F(P)$ with F the q-power Frobenius, this function computes the Ate pairing with $T = q$. This pairing is automatically reduced: It maps directly into n-th roots of unity. Both points need to be in the same point set $E(\mathbb{F}_{q^k})$ and q should be the size of the base field.

In the previous intrinsics one can explicitly check that the input points are indeed in the eigenspaces of Frobenius by setting the parameter `CheckPoints` to `true`. By default Magma does not perform these checks for efficiency reasons.

Example: Pairings on a BN-Curve `(ex-cdc875)`

In this example, we first create a so-called BN-curve, which is an elliptic curve of the form $y^2 = x^3 + b$ defined over $\mathbb{F}_p$ of prime order n and embedding degree $k = 12$. These curves can be found easily by testing for which s both $p(s) = 36s^4 + 36s^3 + 24s^2 + 6s + 1$ and $n(s) = 36s^4 + 36s^3 + 18s^2 + 6s + 1$ are prime. Finding the parameter b is equally easy by testing a few random values, since there are only 6 isogeny classes. The point $G = (1, y)$ can be used as base point.

```
> Zs<s> := PolynomialRing(Integers());
> ps := 36*s^4 + 36*s^3 + 24*s^2 + 6*s + 1;
> ns := 36*s^4 + 36*s^3 + 18*s^2 + 6*s + 1;
> z := 513235038556;  // arbitrary initial value
> repeat
>   z := z+1;
>   p := Evaluate(ps, z);
>   n := Evaluate(ns, z);
```

(continues on next page)

(continued from previous page)

```
> until IsProbablePrime(p) and IsProbablePrime(n);
> p;
249785771109578071340305660639915127509902072 4723
> n;
249785771109578071340305502593791761863447 3494829
> Fp := FiniteField(p);
> b := Fp!0;
> repeat
>   repeat b := b + 1; until IsSquare(b + 1);
>   y := SquareRoot(b + 1);
>   E := EllipticCurve([Fp!0, b]);
>   G := E![1, y];
> until IsZero(n*G);
> E;
Elliptic Curve defined by y^2 = x^3 + 18 over
  GF(249785771109578071340305660639915127509902072 4723)
> #E eq n;  // just to verify
true
> t := p + 1 - n;
> t;
158046123365646454722 9895
> k := 12; // security multiplier
> (p^k - 1) mod n; // check on p, n and k
0
> Fpk := GF(p, k);
> N := Resultant(s^k - 1, s^2 - t*s + p);  // number of points over big field
> Cofac := N div n^2;
> P := E(Fpk)!G;
> Q := Cofac*Random(E(Fpk));  // Q has order n now
```

Up to this point we have constructed a BN-curve and two points of order n. As such we can test, for instance, that P and $2P$ are linearly dependent:

```
> WeilPairing(P, 2*P, n) eq 1;
true
```

and also that the Weil pairing can be obtained from 2 Tate pairing computations:

```
> WeilPairing(P, Q, n) eq (-1)^n*TatePairing(P, Q, n)/TatePairing(Q, P, n);
true
```

or test the bilinearity of the reduced Tate pairing

```
> ReducedTatePairing(P, 2*Q, n) eq ReducedTatePairing(P, Q, n)^2;
true
```

and that the corresponding test for the Tate pairing would fail since the result is a random representative of the coset:

```
> TatePairing(P, 2*Q, n) eq TatePairing(P, Q, n)^2;
false
```

Since the curve is ordinary we cannot use the Eta pairing on this curve. We can use the Ate pairing, but this is defined on $G_2 \times G_1$ and up to this point we only have a generator of G_1. To find a generator of G_2 we need to remove the component in Q that lies in G_1. This can be done easily by using the trace of the point Q: $\mathrm{Tr}(Q) = \sum_{i=0}^{k-1} F(Q)$ with F the p-power Frobenius. The trace equals k times the component of Q in G_1.

```
> TrQ := &+[ E(Fpk) ! [Frobenius(Q[1], Fp, i), Frobenius(Q[2], Fp, i)]  :
>                      i in [0..k-1]];
> Q := k*Q - TrQ;
```

At this point Q is in G_2 and we can compute the Ate pairing of Q and P. For instance, we can test compatibility of the reduced Ate pairing with the reduced Tate pairing:

```
> T := t - 1;
> L := (T^k - 1) div n;
> c := &+[ T^(k - 1 - i)*p^i : i in [0..k-1] ] mod n;
> ReducedAteTPairing(Q, P, n, p)^c eq ReducedTatePairing(Q, P, n)^L;
true
> Frobenius(AteqPairing(Q, P, n, p)^k, Fp, k - 1) eq ReducedTatePairing(Q, P, n);
true
```

To test the Eta pairing computations we will use one of the supersingular elliptic curves $y^2 + y = x^3 + x + b$ with $b = 0$ or 1 over a finite field $\mathbb{F}_{2^m}$ and m odd. These curves have security parameter $k = 4$.

```
> F2m := GF(2, 163);
> q := #F2m;
> E := EllipticCurve([F2m!0, 0, 1, 1, 1]);
> #E;
11692013098647223345629483497433542615764159168513
> IsPrime($1);
true
> n := #E;
> t := TraceOfFrobenius(E);
> P := Random(E);
> k := 4;
> F2m4 := ExtensionField<F2m, X | X^4 + X + 1>;
> N := Resultant(s^k - 1, s^2 - t*s + q);  // number of points over big field
> Cofac := N div n^2;
> P := E(F2m4) ! P;
> Q := Cofac*Random(E(F2m4));  // Q has order n now
> TrQ := &+[ E(F2m4) ! [Frobenius(Q[1], F2m, i), Frobenius(Q[2], F2m, i)]  :
>                       i in [0..k-1]];
> Q := k*Q - TrQ;
```

After this setup we can now compute the Eta pairing and test compatibility with the Tate pairing.

```
> d := GCD(k, q^k - 1);
> Frobenius(EtaqPairing(P, Q, n, q)^(k div d), F2m, k - 1) eq
>                                        ReducedTatePairing(P, Q, n);
true
```

Example: MOV Reduction with the Weil Pairing (ex-a16e5c)

The following example demonstrates the Menezes, Okamoto, and Vanstone (MOV) reduction of the discrete logarithm on a supersingular elliptic curve to a discrete logarithm in a finite field using the Weil Pairing. The group structure of a supersingular curve E over a finite prime field $\mathbb{F}_p$ for $p > 3$ can be $\mathbb{Z}/n\mathbb{Z}$ or $\mathbb{Z}/2\mathbb{Z} \times \mathbb{Z}/(n/2)\mathbb{Z}$, where $n = p + 1$, and the group structure over a degree 2 extension is $\mathbb{Z}/n\mathbb{Z} \times \mathbb{Z}/n\mathbb{Z}$. The nontrivial Weil pairing on this is the basis for this reduction.

```
> p := NextPrime(2^131);
> n := p + 1;
> n;
2722258935367507707706996859454145691688
```

(continues on next page)

(continued from previous page)

```
> Factorization(n);
[ <2, 3>, <3, 2>, <37809151880104273718152734159085356829, 1> ]
> E0 := SupersingularEllipticCurve(GF(p));
> G<x>, f := AbelianGroup(E0);
> G;
Abelian Group isomorphic to Z/272225893536750770770699685945414569168 8
Defined on 1 generator
Relations:
    272225893536750770770699685945414569168 8*x = 0
> n eq #G;
true
> P0 := f(x);
> E1 := BaseExtend(E0, GF(p^2));
> P1 := E1!P0;
> repeat
>    Q1 := Random(E1);
>    z1 := WeilPairing(P1, Q1, n);
> until Order(z1) eq n;
> IsOrder(Q1, n);
true
> r := 1234567;
> z2 := WeilPairing(r*P1, Q1, n);
> z1^r eq z2;
true
> WeilPairing(P1, r*P1, n);
1
```

132.6 Weil Descent in Characteristic Two

One approach to attacking the discrete logarithm problem for elliptic curves over finite fields is through Weil descent. Let E have base field K and let k be a subfield of K. The basic idea is to find a higher genus curve C/k along with a non-trivial homomorphism from $E(K)$ to $\mathrm{Jac}(C)(k)$ [the Jacobian of C]. This transfers the problem to one over a smaller field and Index Calculus methods over C can then be applied.

Magma now contains an implementation, due to Florian Heß, of the GHS Weil Descent for ordinary (i.e., $j(E) \neq 0$) elliptic curves in characteristic 2 (see [PGS00] or the chapter by F. Heß in [Bla05]), which constructs such a C along with the divisor map from E to C.

`WeilDescent(E, k, c): FldFun, FldFin, FldFinElt -> CrvPln, Map`

```
HyperellipticImage: BoolElt                    Default: true
```

The curve E/K is an ordinary elliptic curve over a finite field K of characteristic 2. The argument k is a subfield of K and c is a non-zero element of K.

The function uses GHS descent to construct a plane curve C/k and a map from $E(K)$ to k-divisors of C. There are two possibilities:

i) The curve C is hyperelliptic and Magma implements the group law on its Jacobian, $\mathrm{Jac}(C)$. In this case, C will be returned as a hyperelliptic curve (type `CrvHyp`) and the map returned will actually be the divisor class homomorphism from $E(K)$ to $\mathrm{Jac}(C)(k)$ with points of $E(K)$ being identified with divisor classes in the usual way: $P \leftrightarrow (P) - (0)$.

ii) Otherwise, C is returned as a general plane curve and the map is from points of $E(K)$, considered as divisors of degree 1, to the corresponding k-divisor of C. In this case, where specialised Jacobian arithmetic is not available, it may be more convenient to work with the effective divisors rather

than degree 0 divisors. The user may readily convert to the degree 0 case by getting $h(0)$ initially and then using $h((P) - (0)) = h(P) - h(0)$, where h is the divisor map.

In case ii), no reduction of the image divisor is performed by the divisor map and if $[K : k]$ is large then this divisor will have high degree and the computation may be quite slow. h is defined by divisor pullback corresponding to the function field inclusion $K(E) \hookrightarrow K(C)$ followed by the trace from K down to k.

If the user prefers to have C and h returned as in case ii), even when C is hyperelliptic, the parameter `HyperellipticImage` should be set to `false`.

The third argument c is a parameter to specify the precise data for the descent. If $r = \sqrt{(1/j(E))}$ and $b = r/c$, the function field $K(E)$ is isomorphic to the degree 2 extension of the rational function field $K(x)$ with equation

$$y^2 + y = c/x + a_2(E) + bx$$

and it is this Artin–Schreier extension that is used for the GHS descent as explained in the above references.

It is possible to work directly with function fields, if preferred, using the function `WeilDescent(E, k): FldFun, FldFin → FldFunG, Map`. However this function produces a divisor map between function fields rather than curves and does not give the divisor class map in case i) above.

The genus of C and its degree are very dependent on the choice of c. For a description of how to choose a good c for given E and k, see sections VIII.2.4 and VIII.2.5 in the chapter by Heß cited in the introduction to this section.

We merely note here that $c \in k$ or $b \in k$ leads to a hyperelliptic C, though this may have very large genus and other choices, giving non-hyperelliptic curves, may be better.

`WeilDescentGenus(E, k, c): FldFun, FldFin, FldFinElt -> RngIntElt`

Returns the genus of the Weil descent curve C produced by *`WeilDescent`* for input E, k, c.

`WeilDescentDegree(E, k, c): FldFun, FldFin, FldFinElt -> RngIntElt`

Returns the degree in the second variable of the plane Weil descent curve C produced by *`WeilDescent`* for input E, k, c.

Example: Weil Descent to a Genus 31 Curve `(ex-352154)`

The following example is similar to one from the article of F. Heß referred to above. E is defined over $\mathbb{F}_{2^{155}}$ and C is a hyperelliptic curve of genus 31 over $\mathbb{F}_{2^5}$.

```
> K<w> := FiniteField(2, 155);
> k := FiniteField(2, 5);
> Embed(k, K);
> b :=  w^154 + w^152 + w^150 + w^146 + w^143 + w^142 + w^141 +
>    w^139 + w^138 + w^137 + w^136 + w^134 + w^133 + w^132 + w^131 +
>    w^127 + w^125 + w^123 + w^121 + w^117 + w^116 + w^115 + w^112 +
>    w^111 + w^109 + w^108 + w^107 + w^105 + w^104 + w^102 + w^101 +
>    w^100 + w^99 + w^98 + w^97 + w^95 + w^90 + w^89 + w^88 + w^85 +
>    w^83 + w^81 + w^80 + w^79 + w^78 + w^76 + w^75 + w^73 + w^72 +
>    w^69 + w^68 + w^62 + w^61 + w^59 + w^54 + w^52 + w^47 + w^45 +
>    w^44 + w^43 + w^40 + w^39 + w^37 + w^36 + w^34 + w^32 + w^31 +
>    w^25 + w^15 + w^13 + w^10 + w^8 + w^7 + w^6;
> E := EllipticCurve([K| 1, 0, 0, b, 0]);
> C,div_map := WeilDescent(E, k, K!1);
> C;
Hyperelliptic Curve defined by y^2 + (k.1^16*x^32 + k.1^27*x^8 + k.1^2*x^4 +
    k.1^29*x^2 + k.1^30*x + k.1^20)*y = k.1^30*x^32 + k.1^10*x^8 + k.1^16*x^4 +
    k.1^12*x^2 + k.1^13*x + k.1^3 over GF(2^5)
> ptE := Points(E, w^2)[1];
```

(continues on next page)

(continued from previous page)

```
> ord := Order(ptE);
> ord;
142724769270595988105829091515089019330391026
> ptJ := div_map(ptE); // point on Jacobian(C)
> // check that order ptJ on J = order ptE on E
> J := Jacobian(C);
> ord*ptJ eq J!0;
true
> [ (ord div p[1])*ptJ eq J!0 : p in Factorization(ord) ];
[ false, false, false, false, false, false, false, false ]
```

132.7 Discrete Logarithms

Computing discrete logarithms on elliptic curves over finite fields is considered to be a very difficult problem. The best algorithms for general elliptic curves take exponential time, and do not take much advantage of properties of the curve. Solving such problems can thus be computationally infeasible for large curves. Indeed, elliptic curves are becoming increasingly appealing for applications in cryptography.

Although hard instances of the elliptic curve discrete logarithm may be impossible to solve, Magma is able to efficiently solve reasonable sized instances, or instances where the large prime factor of the order of the base point is not too big. Magma does this as follows: The first step is to compute the factorisation of the order of the base point, which may require calling SEA if the order of the curve is not already known to Magma. Then it checks that the order of the base point is a multiple of the order of the other point. If this is not true then there is no solution. It next breaks the problem down to solving the discrete logarithm modulo the prime power factors of the order using the Pohlig–Hellman algorithm.

For very small primes it will try to solve this by exhaustive search. If a solution does not exist then Magma might determine this during the exhaustive search and abort the remaining computations. For the larger prime power factors, Magma uses the parallel collision search version of Pollard's rho algorithm. The implementation includes Edlyn Teske's idea of r-adding walks and Michael Wiener's and Robert Zuccherato's idea of treating the point P the same as $-P$ (for curves of the form $y^2 = x^3 + ax + b$) so as to reduce the search space by a factor of $1/\sqrt{2}$ (this idea was independently discovered by other researchers). It should be noted that Magma is not always able to determine the nonexistence of a solution and therefore may run forever if given very bad parameters. For this reason, the user has the option of setting a time limit on the calculation.

`Log(Q, P): PtEll, PtEll -> RngIntElt`

The discrete log of P to the base Q (an integer n satisfying $n * Q = P$ where $0 \leq n <$ `Order`(Q)). The arguments Q and P must be points on the same elliptic curve, which must be defined over a finite field. The function returns -1 if it is determined that no solution exists.

`Log(Q, P, t): PtEll, PtEll, RngIntElt -> RngIntElt`

The discrete log of P to the base Q (an integer n satisfying $n * Q = P$ where $0 \leq n <$ `Order`(Q)). The arguments Q and P must be points on the same elliptic curve, which must be defined over a finite field. The argument t is a time limit in seconds on the calculation; if this limit is exceeded then the calculation is aborted. The time limit t must be a small positive integer. This function returns -1 if it is determined that no solution exists; if the calculation is aborted due to the time limit being exceeded then -2 is returned.

Example: Elliptic Curve Discrete Logarithm `(ex-fd64db)`

In the example below we create a random elliptic curve over a randomly chosen 40-bit prime field. Our base point Q is chosen randomly on the curve, and the other point P is selected as a random multiple of Q. Using the `Log` function, we recover that multiplier (m) and finally we verify that the solution is correct.

```
> K := GF(RandomPrime(40));
> E := EllipticCurve([Random(K), Random(K)]);
> E;
Elliptic Curve defined by y^2 = x^3 + 456271502613*x + 504334195864
over GF(742033232201)
> Q := Random(E);
> Q;
(174050269867 : 191768822966 : 1)
> FactoredOrder(Q);
[ <31, 1>, <7789, 1>, <3073121, 1> ]
> P := Random(Order(Q))*Q;
> P;
(495359429535 : 455525174166 : 1)
> m := Log(Q, P);
> m*Q - P;
(0 : 1 : 0)
```

132.8 Bibliography

- [Bla05] I. Blake, G. Serrousi and N. Smart, editor. *Advances in Elliptic Curve Cryptography*. Volume 317 of LMS LNS. Cambridge University Press, Cambridge, 2005.
- [Gau02] P. Gaudry. A Comparison and a Combination of SST and AGM Algorithms for Counting Points on Elliptic Curves in Characteristic 2. In Y. Zheng, editor, *Advances in Cryptology—AsiaCrypt 2002*, volume 2501 of LNCS, 311–327. Springer-Verlag, 2002. Proc. 8th International Conference on the Theory and Applications of Cryptology and Information Security, Dec. 1–5 2002, Queenstown, New Zealand.
- [Hara] R. Harley. Web posting. Under November 2002 entry at http://listserv.nodak.edu/archives/nmbrthry.html.
- [Hub07] Hendrik Hubrechts. Quasi-quadratic elliptic curve point counting using rigid cohomology. *to appear in the Journal of Symbolic Computation*, 2007. http://wis.kuleuven.be/algebra/hubrechts/.
- [Koh03] David R. Kohel. The AGM-$X_0(N)$ Heegner Point Lifting Algorithm and Elliptic Curve Point Counting. In *Advances in Cryptology—AsiaCrypt 2003*, number 2894 in LNCS. Berlin, 2003. Springer.
- [LL03] R. Lercier and D. Lubicz. Counting Points on Elliptic Curves over Finite Fields of Small Characteristic in Quasi Quadratic Time. In *Advances in Cryptology—EuroCrypt 2003*, volume 2656 of LNCS, pages 360–373. Springer, 2003.
- [PGS00] F. Heß P. Gaudry and N. P. Smart. Constructive and destructive facets of Weil descent on elliptic curves. *J. Cryptology*, 15(1):19–46, 2000.

Chapter 133

Elliptic Curves over $\mathbb{Q}$ and Number Fields

133.1 Introduction

This chapter deals with functionality that is specific to elliptic curves defined over the rationals or over number fields. As a general goal, these functions are developed in parallel. Naturally, machinery for curves over $\mathbb{Q}$ has been developed much further than for general number fields, although the latter has been a major focus of recent releases.

The material on computing the Mordell–Weil group (of rational points) and descent methods is presented in the final part of the chapter, and deals with all base fields at the same time. The other topics are presented separately for $\mathbb{Q}$ and for number fields, although many of the functions are the same for both.

Note: functions declared for number fields can also be used over $\mathbb{Q}$. To do so, it may be necessary to construct $\mathbb{Q}$ as `RationalsAsNumberField()` rather than `Rationals()`.

133.2 Curves over the Rationals

133.2.1 Local Invariants

`Conductor(E): CrvEll -> RngIntElt`

The conductor of the elliptic curve E defined over $\mathbb{Q}$.

`BadPrimes(E): CrvEll -> [ RngIntElt ]`

Given an elliptic curve E defined over $\mathbb{Q}$, return the sequence of primes dividing the minimal discriminant of E. These are the primes at which the minimal model for E has bad reduction; note that there may be other primes dividing the discriminant of the given model of E.

`TamagawaNumber(E, p): CrvEll, RngIntElt -> RngIntElt`

Given an elliptic curve E defined over $\mathbb{Q}$ and a prime number p, this function returns the local Tamagawa number of E at p, which is the index in $E(\mathbb{Q}_p)$ of the subgroup $E^0(\mathbb{Q}_p)$ of points with nonsingular reduction modulo p. For any prime p that is of good reduction for E, this function returns 1.

`TamagawaNumbers(E): CrvEll -> [ RngIntElt ]`

Given an elliptic curve E defined over $\mathbb{Q}$, this function returns the sequence of Tamagawa numbers at each of the bad primes of E, as defined above.

`LocalInformation(E, p): CrvEll, RngIntElt -> <RngIntElt, RngIntElt, RngIntElt, RngIntElt, SymKod, BoolElt>, CrvEll`

Given an elliptic curve E defined over $\mathbb{Q}$ and a prime number p, this function returns the local information at the prime p as a tuple of the form $< P, vpd, fp, c_p, K, split >$, consisting of p, its

multiplicity in the discriminant, its multiplicity in the conductor, the Tamagawa number at p, the Kodaira symbol, and finally a boolean which is false iff the curve has nonsplit multiplicative reduction. The second object returned is a local minimal model for E at p.

`LocalInformation(E): CrvEll -> [ Tup ]`

Given an elliptic curve E this function returns a sequence of tuples of the kind described above, for the primes dividing the discriminant of E.

`ReductionType(E, p): CrvEll, RngIntElt -> MonStgElt`

Returns a string describing the reduction type of E at p; the possibilities are "Good", "Additive", "Split multiplicative", or "Nonsplit multiplicative". These correspond to the type of singularity (if any) on the reduced curve. This function is necessary as the Kodaira symbols (see below) do not distinguish between split and unsplit multiplicative reduction.

`TraceOfFrobeniusDirect(E, p): CrvEll, RngIntElt -> RngIntElt`

This function provides an efficient way to obtain the trace of Frobenius of a rational elliptic curve E at a prime p. The argument p is **not** checked to be prime.

`TracesOfFrobenius(E, B): CrvEll, RngIntElt -> SeqEnum`

`TracesOfFrobenius(E, B0, B1): CrvEll, RngIntElt, RngIntElt -> SeqEnum`

`TracesOfFrobenius(E, Ps): CrvEll, [ RngIntElt ] -> SeqEnum`

This function provides an efficient way to obtain the traces of Frobenius $a_p(E)$ of the reduction mod p of E. The first form returns these traces for all primes p up to B. The second form returns them for all primes p in the interval from B_0 to B_1, including the endpoints, and agrees with the first form on overlapping ranges. The elements of the sequence are in increasing order of prime, one for each prime of the interval; for instance, over a range where `PrimesInInterval(B0, B1)` can be called, the i-th element corresponds to its i-th entry. The third form returns the traces at the primes in the given sequence `Ps`, with the i-th element of the result corresponding to `Ps[i]`; the entries may be given in any order, may repeat, and each entry is checked to be prime. All forms use the number of threads set by `SetNthreads`. The first two forms sieve the primes of the range, so an upper endpoint beyond the reach of that sieve is rejected with an error naming the largest endpoint the sieve can address; that reach depends on the binary variant, and on a standard build it is about 1.15×10^{18}. A request whose result would hold more traces than a sequence can contain is also rejected, before any traces are computed; this limit depends on the binary variant too, and on a standard build it already rejects an upper bound of about 2.5×10^{10}. The third form likewise rejects a sequence `Ps` with more entries than a result sequence can contain.

Example: Traces of Frobenius (ex-eca3aa)

```
> E := EllipticCurve([ 0, 1, 1, 3, 5 ]);
> T := TracesOfFrobenius(E, 100); T;
[ 0, 1, -3, -2, -5, 1, -3, -1, 5, 6, 8, -1, -6, 7, -2, 1, 14, -1, -12,
16, -16, -7, 6, 12, 2 ]
> time T := TracesOfFrobenius(E, 10^6);
Time: 5.600
```

133.2.2 Kodaira Symbols

Kodaira symbols have their own type `SymKod`. Apart from the two functions that determine symbols for elliptic curves, there is a special creation function and a comparison operator to test Kodaira symbols.

`KodairaSymbol(E, p): CrvEll, RngIntElt -> SymKod`

Given an elliptic curve E defined over $\mathbb{Q}$ and a prime number p, this function returns the reduction type of E modulo p in the form of a Kodaira symbol.

`KodairaSymbols(E): CrvEll -> [ SymKod ]`

Given an elliptic curve E defined over $\mathbb{Q}$, this function returns the reduction types of E modulo the bad primes in the form of a sequence of Kodaira symbols.

`KodairaSymbol(s): MonStgElt -> SymKod`

Given a string s, return the Kodaira symbol it represents. The values of s that are allowed are: `"I0"`, `"I1"`, `"I2"`, ..., `"In"`, `"II"`, `"III"`, `"IV"`, and `"I0*"`, `"I1*"`, `"I2*"`, ..., `"In*"`, `"II*"`, `"III*"`, `"IV*"`. The dots stand for `"Ik"` with k a positive integer. The 'generic' type `"In"` allows the matching of types `"In"` for any integer $n > 0$ (and similarly for `"In*"`).

`h eq k: SymKod, SymKod -> BoolElt`

Given two Kodaira symbols h and k, this function returns `true` if and only if either both are identical, or one is generic (of the form `"In"`, or `"In*"`) and the other is specific of the same type: `"In"` will compare equal with any of `"I1"`, `"I2"`, `"I3"`, etc., and `"In*"` will compare equal with any of `"I1*"`, `"I2*"`, `"I3*"`, etc. Note however that `"In"` and `"I3"` are different from the point of view of set creation.

`h ne k: SymKod, SymKod -> BoolElt`

The logical negation of `eq`.

Example: Curves with Kodaira Symbol I0* (ex-c22679)

We search for curves with a particular reduction type '`I0*`' in a family of curves.

```
> S := [ ];
> for n := 2 to 100 do
>    E := EllipticCurve([n, 0]);
>    for p in BadPrimes(E) do
>       if KodairaSymbol(E, p) eq KodairaSymbol("I0*") then
>          Append(~S, <p, n>);
>       end if;
>    end for;
> end for;
> S;
[ <3, 9>, <3, 18>, <5, 25>, <3, 36>, <3, 45>, <7, 49>, <5, 50>,
<3, 63>, <3, 72>, <5, 75>, <3, 90>, <7, 98>, <3, 99>, <5, 100> ]
```

133.2.3 Complex Multiplication

`HasComplexMultiplication(E): CrvEll -> BoolElt, RngIntElt`

Given an elliptic curve E over the rationals (or a number field), the function determines whether the curve has complex multiplication or not and, if so, also returns the discriminant of the CM quadratic order. The algorithm uses fairly straightforward analytic methods, which are not suited to very high degree j-invariants with CM by orders with discriminants more than a few thousand.

133.2.4 Isogenous Curves

`IsogenousCurves(E): CrvEll[FldRat] -> SeqEnum, RngIntElt`

The set of curves $\mathbb{Q}$-isogenous to a rational elliptic curve E. The method used is to proceed prime-by-prime. Mazur's Theorem restricts the possibilities to a short list, and j-invariant considerations leave only p-isogenies for $p = 2, 3, 5, 7, 13$ to be considered. For $p = 2$ or 3, the method proceeds by finding the rational roots of the p-th division polynomial for the curve E. From these roots, one can then recover the isogenous curves. The question as to whether or not there is a p-isogeny is entirely a function of the j-invariant of the curve, and this idea is used for $p = 5, 7, 13$. In these cases the algorithm first takes a minimal twist of the elliptic curve, and then finds rational roots of the polynomial of degree $(p + 1)$ that comes from a fibre of $X_0(p)$. The isogenous curves of the minimal twist corresponding to these roots are then computed, and then these curves are twisted back to get the isogenous curves of E. In all cases, if the conductor is squarefree, some small values of the Frobenius traces are checked mod p to ensure the feasibility of a p-isogeny. Other similar ideas involving checking congruences are also used to try to eliminate the possibility of a p-isogeny without finding roots. Tree-based methods are used to extend the isogeny tree from (say) 2-isogenies to 4-isogenies to 8-isogenies, etc. The integer returned as a second argument corresponds to the largest degree of a cyclic isogeny between two curves in the isogeny class. The

ordering of the list of curves returned is well-defined; the first curve is always the curve of minimal Faltings height, and the heights generically increase upon going down the list. However, when there are isogenies of two different prime degrees, a different ordering is used. In the case where there is a 4-isogeny (or a certain type of 9-isogeny), there is an arbitrary choice made on the bottom tree leaves in order to make the ordering consistent.

FaltingsHeight(E): CrvEll[FldRat] -> FldReElt

Precision: RngIntElt	Default:

Computes the unstable Faltings height of an elliptic curve over the rationals. This is defined as $-\log\sqrt{\Omega}$ where Ω is the fundamental volume. Note that **FaltingsHeight** for an elliptic curve over a number fields uses a different normalization.

StableFaltingsHeight(E): CrvEll[FldRat] -> FldReElt

Precision: RngIntElt	Default:

Computes the stable Faltings height of an elliptic curve over the rationals. This is defined as

$$\frac{\log \operatorname{denom}(j_E)}{12} - \frac{\log|\Delta_E|}{12} - \log\sqrt{\Omega}$$

where j_E is the j-invariant, Δ_E is the (minimal) discriminant, and Ω is the fundamental volume.

Example: Isogenous Curves via Division Polynomials (ex-17c07d)

First we compute isogenous curves using **DivisionPolynomial**; in this special case we obtain the entire isogeny class by considering only 3-isogenies directly from E.

```
> E:=EllipticCurve([1,-1,1,1,-1]);
> F:=Factorization(DivisionPolynomial(E,3));
> I:=IsogenyFromKernel(E,F[1][1]); I;
Elliptic Curve defined by y^2 + x*y + y = x^3 - x^2 - 29*x - 53 over Rational
Field
> I:=IsogenyFromKernel(E,F[2][1]); I;
Elliptic Curve defined by y^2 + x*y + y = x^3 - x^2 - 14*x + 29 over Rational
Field
```

Alternatively, we can get the **IsogenousCurves** directly. Note that **IsogenyFromKernel** also returns the map between the isogenous curves as a second argument.

```
> IsogenousCurves(E);
[
    Elliptic Curve defined by y^2 + x*y + y = x^3 - x^2 + x - 1 over Rational
    Field,
    Elliptic Curve defined by y^2 + x*y + y = x^3 - x^2 - 29*x - 53 over
    Rational Field,
    Elliptic Curve defined by y^2 + x*y + y = x^3 - x^2 - 14*x + 29 over
    Rational Field
]
9
```

133.2.5 Heights and Height Pairing

These functions require that the corresponding elliptic curve has integral coefficients.

NaiveHeight(P): PtEll -> FldPrElt

WeilHeight(P): PtEll -> FldPrElt

Given a point $P = (a/b, c/d, 1)$ on an elliptic curve E defined over $\mathbb{Q}$ with integral coefficients, this function returns the naive (or Weil) height $h(P)$ whose definition is $h(P) = \log\max\{|a|, |b|\}$.

Height(P: parameters): PtEll -> NFldComElt

CanonicalHeight(P: parameters): PtEll -> NFldComElt

Precision: RngIntElt	Default:

Given a point P on an elliptic curve E defined over $\mathbb{Q}$ with integral coefficients, this function returns the canonical height $\hat{h}(P)$.

One definition of $\hat{h}(P)$ is $\hat{h}(P) = \lim_{n\to\infty} 4^{-n}h(2^nP)$, although this is of limited computational use. A more useful computational definition is as the sum of local heights: $\hat{h}(P) = \sum_{p\leq\infty} \hat{h}_p(P)$, where the sum ranges over each prime p and the so-called 'infinite prime'. Each of these local heights can be evaluated fairly simply, and in fact most of them are 0. The function *always* uses a minimal model for the elliptic curve internally, as otherwise the local height computation could fail. The computation at the infinite prime uses a slight improvement over the σ-function methods given in Cohen, with the implementation being based on an AGM-trick due to Mestre.

LocalHeight(P, p): PtEll, RngIntElt -> FldComElt

Precision : RngIntElt	Default:
Check : BoolElt	Default: false
Renormalization: BoolElt	Default: false

Given a point P on an elliptic curve E defined over $\mathbb{Q}$ with integral coefficients, this function returns $\hat{h}_p(P)$, the local height of P at p as described in **Height** above. The integer p must be either a prime number or 0; in the latter case the height at the infinite prime is returned. The **Check** parameter determines whether to check if the second argument is prime. The **Renormalization** parameter changes what definition of heights is used. Without this flag, a factor of $\frac{1}{6}\log\Delta_v$ is added at every place.

HeightPairing(P, Q: parameters): PtEll, PtEll -> FldComElt

Precision: RngIntElt	Default:

Given two points P, Q on the same elliptic curve defined over $\mathbb{Q}$ with integral coefficients, this function returns the height pairing of P and Q, which is defined by $\hat{h}(P,Q) = (\hat{h}(P+Q) - \hat{h}(P) - \hat{h}(Q))/2$.

HeightPairingMatrix(S: parameters): [PtEll] -> AlgMat

HeightPairingMatrix(E: parameters): CrvEll -> AlgMat

Precision: RngIntElt	Default:

Given a sequence of points on an elliptic curve defined over $\mathbb{Q}$ with integral coefficients, this function returns the height pairing matrix. If an elliptic curve is passed to it, the corresponding matrix for the Mordell–Weil generators is returned.

Regulator(S): [PtEll] -> FldComElt

Precision: RngIntElt	Default:

Given a sequence of points S on an elliptic curve E over $\mathbb{Q}$, this function returns the determinant of the Néron-Tate height pairing matrix of the sequence.

Regulator(E): CrvEll -> FldComElt

Precision: RngIntElt	Default:

Given an elliptic curve E this function returns the regulator of E; i.e., the determinant of the Néron-Tate height pairing matrix of a basis of the free quotient of the Mordell–Weil group.

Example: Canonical Heights of Generators (ex-d6c540)

```
> E := EllipticCurve([0,0,1,-7,6]);
> P1, P2, P3 := Explode(Generators(E));
> Height(P1);
0.66820516565192793503314205 09
> IsZero(Abs(Height(2*P1) - 4*Height(P1)));
true
> BadPrimes(E);
[ 5077 ]
```

The local height of a point P at a prime is 0 except possibly at the bad primes of E, the 'infinite' prime, and those primes dividing the denominator of the x-coordinate of P. Since these generators have denominator 1 we see that only two local heights need to be computed to find the canonical heights of these points.

```
> P2;
(2 : 0 : 1)
> LocalHeight(P2, 0);
-0.65503594715968618249727806 9814
> LocalHeight(P2, 5077); // 0 + Log(5077)/6
1.42207930249123238829272871637
> LocalHeight(P2, 0 : Renormalization);
0.76704335533154620579545064 6552
> LocalHeight(P2, 5077 : Renormalization);
0.00000000000000000000000000000
> Height(P2);
0.76704335533154620579545064 66
```

The above shows that the local height at a bad prime may still be zero, at least in the renormalised value.

SilvermanBound(H): SetPtEll -> FldPrElt

SilvermanBound(E): CrvEll -> FldPrElt

Given an elliptic curve E over $\mathbb{Q}$ with integral coefficients, this function returns the Silverman bound B of E. For any point P on E we will have $h(P) - \hat{h}(P) \leq B$.

SiksekBound(H: parameters): SetPtEll -> FldPrElt

SiksekBound(E: parameters): CrvEll -> FldPrElt

```
Torsion: BoolElt                    Default: false
```

Given an elliptic curve E over $\mathbb{Q}$ which is a minimal model, this function returns a real number B such that for any point P on E $h(P) - \hat{h}(P) \leq B$. In many cases, the Siksek bound is much better than the Silverman bound.

If the parameter `Torsion` is `true` then a potentially better bound B_{Tor} is computed, such that for any point P on E there exists a torsion point T so that $h(P+T) - \hat{h}(P) \leq B_{Tor}$. Note that $\hat{h}(P+T) = \hat{h}(P)$.

Example: Comparing Silverman and Siksek Bounds (ex-0bdc03)

We demonstrate the improvement of the Siksek bound over the Silverman bound for the three example curves from Siksek's paper [Sik95].

```
> E := EllipticCurve([0, 0, 0, -73705, -7526231]);
> SilvermanBound(E);
13.00022113685530200655193
> SiksekBound(E);
```

(continues on next page)

(continued from previous page)

```
0.82150471924479497959096194911
> E := EllipticCurve([0, 0, 1, -6349808647, 193146346911036]);
> SilvermanBound(E);
21.75416864448061105008
> SiksekBound(E);
0.617777290687848386342334921728509577480
> E := EllipticCurve([1, 0, 0, -5818216808130, 5401285759982786436]);
> SilvermanBound(E);
27.56255914401769757660
> SiksekBound(E);
15.708189654302901611424812945 45
```

This last curve has a torsion point, so we can further improve the bound:

```
> T := E![ 1402932, -701466 ];
> Order(T);
2
> SiksekBound(E : Torsion := true);
11.030987623117983983182951265 2688
```

Here is a point which demonstrates the applicability of the modified bound.

```
> P := E![ 14267166114 * 109, -495898392903126, 109^3 ];
> NaiveHeight(P) - Height(P);
12.193000709615680011116868901084
> NaiveHeight(P + T) - Height(P);
2.60218831527724007036
```

IsLinearlyIndependent(P, Q): PtEll, PtEll -> BoolElt, ModTupElt

Given points P and Q on an elliptic curve E, this function returns `true` if and only if P and Q are independent free elements of the group of rational points of an elliptic curve (modulo torsion points). If `false`, the function returns a vector $v = (r, s)$ as a second value such that $rP + sQ$ is a torsion point.

IsLinearlyIndependent(S): [PtEll] -> BoolElt, ModTupElt

Given a sequence of points S belonging to an elliptic curve E, this function returns `true` if and only if the points in S are linearly independent (modulo torsion). If `false`, the function returns a vector v in the kernel of the height pairing matrix, i.e. giving a torsion point as a linear combination of the points in S.

ReducedBasis(S): [PtEll] -> [PtEll]

Given a sequence of points S belonging to an elliptic curve E over the rationals, the function returns a sequence of points that are independent modulo the torsion subgroup (equivalently, they have non-degenerate height pairing), and which generate the same subgroup of $E(\mathbb{Q})/E_{tors}(\mathbb{Q})$ as points of S.

Example: Linear Independence of Points (ex-ce19b5)

We demonstrate the linear independence test on an example curve with an 8-torsion point and four independent points. The torsion point is easily recognized and we establish the independence of the remaining points using the height function.

```
> E := EllipticCurve([0,1,0,-95549172512866864, 1169099874279855380833 4900]);
> P0 := E![ 39860582, 2818809365988 ];
> P1 := E![ 144658748, -946639447182 ];
> P2 := E![ 180065822, 569437198932 ];
> P3 := E![ -339374593, 2242867099638 ];
> P4 := E![ -3492442669/25, 590454479818404/125 ];
```

(continues on next page)

(continued from previous page)

```
> S0 := [P0, P1, P2, P3, P4];
> IsLinearlyIndependent(S0);
false (1 0 0 0 0)
> Order(P0);
8
> S1 := [P1, P2, P3, P4];
> IsLinearlyIndependent(S1);
true
```

We now demonstrate the process of solving for a nontrivial linear dependence among points on an elliptic curve by constructing a singular matrix, forming the corresponding linear combination of points, and establishing that the dependence is in the matrix kernel.

```
> M := Matrix(4, 4, [-3,-1,2,-1,3,3,3,-2,3,0,-1,-1,0,2,1,1]);
> Determinant(M);
0
> S2 := [ &+[ M[i, j]*S1[j] : j in [1..4] ] : i in [1..4] ];
> IsLinearlyIndependent(S2);
false ( 5 -3  8  7)
> Kernel(M);
RSpace of degree 4, dimension 1 over Integer Ring
Echelonized basis:
( 5 -3  8  7)
```

Despite the moderate size of the numbers which appear in the matrix M, we note that height is a quadratic function in the matrix coefficients. Since height is a logarithmic function of the coefficient size of the points, we see that the sizes of the points in this example are very large:

```
> [ RealField(16) | Height(P) : P in S2 ];
[ 137.376951049198, 446.51954933694, 52.724183282292, 59.091649046171 ]
> Q := S2[2];
> Log(Abs(Numerator(Q[1])));
467.65985870406594118081172 53
> Log(Abs(Denominator(Q[1])));
449.25545877278405839494427 65
```

pAdicHeight(P, p): PtEll, RngIntElt -> FldPadElt

```
Precision: RngIntElt                    Default: 0
E2       : FldPadElt                    Default: 0
```

Given a point P on an elliptic curve defined on the rationals and a prime $p \geq 5$ of good ordinary reduction, this function computes the p-adic height of P to the desired precision. The value of the `EisensteinTwo` function for the curve can be passed as a parameter. The algorithm dates back to [MT91] with improvements due to [MST06] and [Har08]. The normalization is that of the last-named paper and is $2p$ (or $-2p$ in some cases) as large as in other papers.

pAdicRegulator(S, p): [PtEll], RngIntElt -> FldPadElt

```
Precision: RngIntElt                    Default: 0
E2       : FldPadElt                    Default: 0
```

Given a set of points S on an elliptic curve defined over the rationals and a prime $p \geq 5$ of good ordinary reduction, this function computes the p-adic height regulator of S to the desired precision. Here the normalization divides out a power of p from the individual heights.

EisensteinTwo(E, p): CrvEll, RngIntElt -> FldPadElt

Precision: RngIntElt	Default: 0

Given an elliptic curve over the rationals and a prime $p \geq 5$ of good ordinary reduction, this function computes the value of the Eisenstein series E_2 using the Monsky-Washnitzer techniques via Kedlaya's algorithm. See `pAdicHeight` above for references.

`FrobeniusMatrix(E, p): CrvEll, RngIntElt -> Mtrx`

Precision: RngIntElt	Default: 10

Given an elliptic curve over the rationals and a prime $p \geq 5$ of good reduction, this function computes the matrix corresponding to the action of Frobenius, to the given precision. The basis used is $\{dx/y, x(dx/y)\}$.

Example: p-Adic Heights of Points (ex-1e7aba)

We give examples for computing p-adic heights.

```
> E := EllipticCurve("5077a");
> P1 := E ! [2, 0];
> P2 := E ! [1, 0];
> P3 := E ! [-3, 0];
> assert P1+P2+P3 eq E!0; // the three points sum to zero
> for p in PrimesInInterval(5,30) do pAdicHeight(P1, p); end for;
4834874647277 + O(5^20)
5495832406058373*7 + O(7^20)
-12403985313704524674*11 + O(11^20)
6167277315947533603 89*13 + O(13^20)
5314497586743410875 4867*17 + O(17^20)
6514787540337334207 44924*19 + O(19^20)
1382894029404692415656094*23^2 + O(23^20)
-26155034412147476888009935 18*29 + O(29^20)
> pAdicHeight(P1, 37); // 37 is not ordinary for this curve
>> pAdicHeight(P1, 37); // 37 is not ordinary for this curve
Runtime error in 'pAdicHeight': p cannot be supersingular
> for p in PrimesInInterval(5,30) do pAdicRegulator([P1, P2], p); end for;
-263182161797834*5^-2 + O(5^20)
-35022392779944725 + O(7^20)
-331747503271490921584 + O(11^20)
246446095809126124462 + O(13^20)
-15285279157972275150672 70 + O(17^20)
33388427569565384672997 0*19 + O(19^20)
41297839835611557028076 0602 + O(23^20)
-45973362301048046561945193743 + O(29^20)
> pAdicRegulator([P1, P2, P3], 23); // dependent points
O(23^20)
> eisen_two := EisensteinTwo(E, 13 : Precision:=40); eisen_two;
25360291252705414983472710631099471724343012 + O(13^40)
> pAdicRegulator([P1, P2], 13 : Precision:=40, E2:=eisen_two);
85894629213918025547455609490608727790695215 + O(13^40)
```

133.2.6 Heegner Points

For an elliptic curve of rank 1, it is possible to compute a generator by an analytic process; the elliptic logarithm of some multiple nP of the generator is the sum of the values of the modular parameterization at a series of points in the upper half-plane corresponding to a full set of class representatives for an appropriately-chosen quadratic field. There is no such thing as a free lunch, sadly; the calculation requires computing $O(hN)$ terms to a precision of $O(h)$ digits (where N is the conductor), so in practice it works best for curves contrived to have small conductors.

The calculation proceeds in three stages: choosing the quadratic field $\mathbb{Q}(\sqrt{-d})$, evaluating the modular parameterization, and recovering the generator from the elliptic logarithm value. Essentially, this is an implementation of the method of Gross and Zagier [GZ86]; Elkies performed some substantial computations using this method in 1994, and much of the work required to produce this implementation was done by Cremona and Womack. An array of tricks have been added, and are described to some extent in some notes of Watkins.

HeegnerPoint(E : parameters): CrvEll -> BoolElt, PtEll

```
NaiveSearch    : RngIntElt                    Default: 1000
Discriminant   : RngIntElt                    Default:
Cover          : Crv                          Default:
DescentPossible: BoolElt                      Default: true
IsogenyPossible: BoolElt                      Default: true
Traces         : SeqEnum                      Default: []
Heegner        : Verbose                      Default: 1
```

Attempts to find a point on a rank 1 rational elliptic curve E using the method of Heegner points, returning `true` and the point if it finds one, otherwise `false`. An error is triggered if the (analytic) rank is not 1.

The parameter `NaiveSearch` indicates to what height the algorithm does a preliminary search for points on E (using `Points`). The `Discriminant` parameter allows the user to specify an auxiliary discriminant; this must satisfy the Heegner hypothesis, and the corresponding quadratic twist must have rank 0. The `Cover` option allows the user to specify a 2-cover or 4-cover (perhaps obtained from `TwoDescent` or `FourDescent`) to speed the calculation. This should be either a hyperelliptic curve or a quadric intersection, and there must be a map from the cover to the given elliptic curve. If the `DescentPossible` option is true, then the algorithm might perform such a descent in any case. If the `IsogenyPossible` option is true, then the algorithm will first try to guess the best isogenous curve on which to do the calculation — if a `Cover` is passed to the algorithm, it will be ignored if E is not the best isogenous curve. The `Traces` option takes an array of integers which correspond to the first however-many traces of Frobenius for the elliptic curve, thus saving having to recompute them.

The algorithm assumes that the Manin constant of the given elliptic curve is 1 (which is conjectured in this case), and does not return a generator for the Mordell–Weil group, but a point whose height is that given by a conjectural extension of the Gross–Zagier formula. This should be $\sqrt{Sha}$ times a generator.

HeegnerPoint(C : parameters): CrvHyp -> BoolElt, PtHyp

HeegnerPoint(f : parameters): RngUPolElt -> BoolElt, PtHyp

HeegnerPoint(C : parameters): Crv -> BoolElt, Pt

```
NaiveSearch : RngIntElt                    Default: 10000
Discriminant: RngIntElt                    Default:
Traces      : SeqEnum                      Default: []
Heegner     : Verbose                      Default: 3
```

These are utility functions for the above. The input is either a hyperelliptic curve given by a polynomial of degree 4, or this quartic polynomial — in both cases the quartic should have no rational roots — or a nonsingular intersection of two quadrics in $\mathbb{P}^3$. These functions call `HeegnerPoint` on the underlying elliptic curve (which must have rank 1). They then aim to map the computed point back to the given covering curve, though this will not be possible when the Tate–Shafarevich group of the elliptic curve has even order. In such cases an error is triggered, so too when the cover corresponds to a torsion point.

The rational reconstruction step of the `HeegnerPoint` algorithm can be quite time-consuming for some coverings, especially if the index is large.

ModularParametrization(E, z, B : parameters): CrvEll[FldRat], FldComElt, RngIntElt -> FldComElt

ModularParametrisation(E, z, B : parameters): CrvEll[FldRat], FldComElt, RngIntElt -> FldComElt

ModularParametrization(E, z : parameters): CrvEll[FldRat], FldComElt -> FldComElt

ModularParametrisation(E, z : parameters): CrvEll[FldRat], FldComElt -> FldComElt

ModularParametrization(E, Z, B : parameters): CrvEll[FldRat], [FldComElt], RngIntElt -> [FldComElt]

ModularParametrisation(E, Z, B : parameters): CrvEll[FldRat], [FldComElt], RngIntElt -> [FldComElt]

ModularParametrization(E, Z : parameters): CrvEll[FldRat], [FldComElt] -> [FldComElt]

ModularParametrisation(E, Z : parameters): CrvEll[FldRat], [FldComElt] -> [FldComElt]

Traces: SeqEnum	Default: []

ModularParametrization(E, f, B : parameters): CrvEll[FldRat], QuadBinElt, RngIntElt -> FldComElt

ModularParametrisation(E, f, B : parameters): CrvEll[FldRat], QuadBinElt, RngIntElt -> FldComElt

ModularParametrization(E, f : parameters): CrvEll[FldRat], QuadBinElt -> FldComElt

ModularParametrisation(E, f : parameters): CrvEll[FldRat], QuadBinElt -> FldComElt

ModularParametrization(E, F, B : parameters): CrvEll[FldRat], [QuadBinElt], RngIntElt -> [FldComElt]

ModularParametrisation(E, F, B : parameters): CrvEll[FldRat], [QuadBinElt], RngIntElt -> [FldComElt]

ModularParametrization(E, F : parameters): CrvEll[FldRat], [QuadBinElt] -> [FldComElt]

ModularParametrisation(E, F : parameters): CrvEll[FldRat], [QuadBinElt] -> [FldComElt]

Traces : SeqEnum	Default: []
Precision: RngIntElt	Default:

Given a rational elliptic curve E and a point z in the upper-half-plane, compute the modular parametrization $\int_z^\infty f_E(\tau)\,d\tau$ where f_E is the modular form associated to E. The version with a bound B uses the first B terms of the q-expansion, while the other version determines how many terms are needed. The optional parameter `Traces` allows the user to pass the first however-many traces of Frobenius. There are also versions which take an array Z of complex points, and versions which take positive definite binary quadratic forms f (or an array F) rather than points in the upper-half-plane (a `Precision` can be specified with the latter).

HeegnerDiscriminants(E, lo, hi): CrvEll[FldRat], RngIntElt, RngIntElt -> SeqEnum

Fundamental: BoolElt	Default: false
Strong : BoolElt	Default: false

Given a rational elliptic curve and a range from lo to hi, compute the negative fundamental discriminant in this range that satisfies the Heegner hypothesis for the curve. The `Fundamental` option restricts to fundamental discriminants. The `Strong` option restricts to discriminants that satisfy a stronger Heegner hypothesis, namely that $a_p = -1$ for all primes p that divide $\gcd(D, N)$.

HeegnerForms(E, D : parameters): CrvEll[FldRat], RngIntElt -> SeqEnum

UsePairing : BoolElt	Default: false
UseAtkinLehner: BoolElt	Default: true
Use_wQ : BoolElt	Default: true
IgnoreTorsion : BoolElt	Default: false

Given a rational elliptic curve of conductor N and a negative discriminant that meets the Heegner hypothesis for N, the function computes representatives for the complex multiplication points on $X_0(N)$.

In general the return value is a sequence of 3-tuples (Q, m, T) where Q is a quadratic form, m is a multiplicity, and T is a torsion point on E. Letting ϕ be the modular parametrization map, the sum on E of the values $m(\phi(Q)+T)$ is a Heegner point in $E(\mathbb{Q})$. When the number of these values equals the class number $h = h(D)$ (and the multiplicities are all 1 or -1) then they are actually the images on E of the CM points on $X_0(N)$. (They all correspond to a single choice of square root of D modulo $4N$.)

If `UsePairing` is added, then CM points that give conjugate values under the modular parametrisation map are combined. If `UseAtkinLehner` is added, then more than one square root of D modulo $4N$ can be used. If `Use_wQ` is added, then forms can appear with extra multiplicity, due to primes that divide the GCD of the conductor and the fundamental discriminant. If `IgnoreTorsion` is added, then the fact that Atkin-Lehner can change the result by a torsion point will be ignored. The `UsePairing` option requires that `IgnoreTorsion` be true.

This function requires (so as not to confuse the user) that the `ManinConstant` of the curve be equal to 1.

`HeegnerForms(N, D : parameters): RngIntElt, RngIntElt -> SeqEnum`

```
AtkinLehner: [RngIntElt]                    Default: []
```

Given a level N and a discriminant that meets the Heegner hypothesis for the level, the function returns a sequence of binary quadratic forms which correspond to the Heegner points. The Atkin-Lehner option takes a sequence of q with $\gcd(q, N/q) = 1$ and has the effect of allowing more than one square root of D modulo $4N$ to be used.

`ManinConstant(E): CrvEll[FldRat] -> RngIntElt`

Compute the Manin constant of a rational elliptic curve. This is in most cases simply a conjectural value (and most often just 1).

`HeegnerTorsionElement(E, Q): CrvEll[FldRat], RngIntElt -> PtEll`

Given a rational elliptic curve and an integer Q corresponding to an Atkin-Lehner involution (so that $\gcd(Q, N/Q) = 1$), this function computes the torsion point on the curve corresponding to the period given by the integral from $i\infty$ to $w_Q(i\infty)$.

`HeegnerPoints(E, D : parameters): CrvEll[FldRat], RngIntElt -> Tup, PtEll`

```
ReturnPoint: BoolElt                        Default: false
Precision  : RngIntElt                      Default: 100
Heegner    : Verbose                        Default: 1
```

Given an elliptic curve E over $\mathbb{Q}$, and a suitable discriminant D (of a quadratic field) this function computes the images on E under the modular parametrization of the CM points on $X_0(N)$ associated to D. It returns a tuple $\langle p_D, m\rangle$, where p_D is an irreducible polynomial whose roots are the x-coordinates of these images, and where m is the multiplicity of each of these images (the number of CM points on $X_0(N)$ mapping to a given point on E). These x-coordinates lie in the ring class field of $\mathbb{Q}(\sqrt{D})$, and the class number $h(D)$ must equal either $m \deg(p_D)$ or $2m \deg(p_D)$.

The conductor of the order $\mathbb{Q}(\sqrt{D})$ is required to be coprime to the conductor of E.

The second object returned by the function is one of the conjugate points, as an element of the point set $E(H)$ where H is the field defined by p_D, or a quadratic extension of it. This step can be time consuming; if `ReturnPoint` is set to `false`, the point is not computed.

Warning: The computation is not rigorous, as it involves computing over the complex numbers, and then recognising the coefficients of the polynomial as rationals. However, the program performs a heuristic check: it checks that the polynomial has the correct splitting at some small primes (sufficiently many to be sure that wrong answers will not occur in practice).

Example: Finding Heegner Points (ex-f2bb35)

```
> time HeegnerPoint(EllipticCurve([1,0,0,312,-3008])); //uses search
true (12 : -56 : 1)
Time: 0.240
> time HeegnerPoint(EllipticCurve([0,0,1,-22787553,-41873464535]));
true (1100363782947843220359298466100412 9/1048524607168660222036584535396 :
-1004181871409718654255966342764883958203316448270339/1073662885578314794 6
9939327005831098688999805 6 : 1)
Time: 1.380
```

Example: Heegner Points Using an Isogeny (ex-350bc4)

Here are some more complicated examples. In the first one, the curve has a 163-isogenous curve, which the algorithm uses to speed the computation. This occurs automatically, with most of the time taken being in computing the isogeny map.

```
> E := EllipticCurve([0,0,1,-57772164980,-5344733777551611]);
> b, pt:= HeegnerPoint(E);
> Height(pt);
373.47866156914237988407748 0412
```

Example: Heegner Points via Two-Descent (ex-24a537)

In this next example, we first use descent to get a covering on which the desired point will have smaller height.

```
> E := EllipticCurve([0,-1,0,-71582788120,-7371563751267600]);
> T := TwoDescent(E : RemoveTorsion)[1];
> T;
Hyperelliptic Curve defined by y^2 = -2896*x^4 - 57928*x^3 - 202741*x^2
 + 868870*x - 651725 over Rational Field
> S := FourDescent(T : RemoveTorsion)[1];
> b, pt := HeegnerPoint(S);
> pt;
(-34940281640330765977951793/72963317481453011430052232 :
46087465795237503244048957/72963317481453011430052232 :
82501230298438806677528297/72963317481453011430052232 : 1)
```

We obtain a point on S, not on the original curve. We map it to E using the descent machinery.

```
> _, m := AssociatedEllipticCurve(S);
> PT := m(pt);
> PT;
(269356432396748248246118697936017740034773039452236777412444550587 53
924460587724041316046901475913814274843012625394997597714766923750555
669810857471061235926971161094484197799515212721830555528087646969545
65/837619099331786545303500955405701693904657590793269053332825491870
645799230089011267382251975387071464373622714525213870033427638236014
228801250428843907758749650143692004410924389857019093192586024416 4 :
410189886613094359515065087530226951251222017120181558101276037601794
678635473792334597914052557787798899390943937170051840037860568689664
871351793404398352109768736545284569745952273382778021947446766526118
621612003004627401344216069791103330332004546699363266079516476593864
985383039985673798699741432591743 95/766601839981278884919411893932635
388671474045058772153268571977045787439290021029065740244648077391646
579430077900352635000328805236464794479797855430820809227462762666048
009219642700664370632930446228302292313415544188481623273194456758440
446505454248062292834244615276350323795396202280072306278575288 : 1)
```

(continues on next page)

(continued from previous page)

```
> Height(PT);
476.81118281872033694972478178O
> ConjecturalRegulator(E : Precision := 5);
476.81 1
```

Example: Heegner Points over a Class Field (ex-6e096b)

Finally, we do some Heegner point calculation with the curve 43A and the discriminant -327. Note that the obtained trace down to the rationals is 3-divisible, but the point over the Hilbert class field is not.

```
> E := EllipticCurve([0,1,1,0,0]);
> HeegnerDiscriminants(E,-350,-300);
[ -347, -344, -340, -335, -331, -328, -327, -323, -319, -308, -303 ]
> HF := HeegnerForms(E,-327); HF;
[ <<43,19,4>, 1, (0 : 1 : 0)>, <<43,67,28>, 1, (0 : 1 : 0)>,
<<86,105,33>, 1, (0 : 1 : 0)>, <<86,153,69>, 1, (0 : 1 : 0)>,
<<129,105,22>, 1, (0 : 1 : 0)>, <<129,153,46>, 1, (0 : 1 : 0)>,
<<172,67,7>, 1, (0 : 1 : 0)>, <<258,363,128>, 1, (0 : 1 : 0)>,
<<258,411,164>, 1, (0 : 1 : 0)>, <<301,67,4>, 1, (0 : 1 : 0)>,
<<473,621,204>, 1, (0 : 1 : 0)>, <<473,841,374>, 1, (0 : 1 : 0)> ]
> H := [x[1] : x in HF]; mul := [x[2] : x in HF];
> params := ModularParametrization(E,H : Precision := 10);
> wparams := [ mul[i]*params[i] : i in [1..#H]];
> hgpt := EllipticExponential(E,&+wparams);
> hgpt;
[ 1.000000002 - 1.098466121E-9*I, 1.000000003 - 1.830776870E-9*I ]
> HeegnerPt := E![1,1];
> DivisionPoints(HeegnerPt, 3);
[ (0 : -1 : 1) ]
```

So the Heegner point is 3 times $(0,-1)$ in $E(\mathbb{Q})$. We now find algebraically one of the points (of which we took the trace to get $HeegnerPt$). This point will be defined over a subfield of the class field of $\mathbb{Q}(\sqrt{-327})$, and will have degree dividing the class number. The *poly* below is the minimal polynomial of the x-coordinate.

```
> ClassNumber(QuadraticField(-327));
12
> poly, pt := HeegnerPoints(E, -327 : ReturnPoint);
> poly;
<t^12 + 10*t^11 + 76*t^10 - 1150*t^9 + 475*t^8 - 4823*t^7 +
 997*t^6 - 5049*t^5 - 2418*t^4 - 468*t^3 - 3006*t^2 + 405*t - 675, 1>
> pt;
(u : 1/16976844562625*(58475062076*u^11 + 568781661961*u^10 +
 4238812569862*u^9 - 68932294336288*u^8 + 42534102728187*u^7 -
 238141610215111*u^6 + 134503848441911*u^5 - 122884262733563*u^4 -
 58148031982456*u^3 + 129014145075851*u^2 -
 68190988248855*u + 40320643901175) : 1)
> DivisionPoints(pt,3);
[]
```

Here is another example of Heegner points over class fields.

Example: Heegner Points for a Given Discriminant (ex-339f1a)

```
> E := EllipticCurve([0,-1,0,-116,-520]);
> Conductor(E);
```

(continues on next page)

(continued from previous page)

```
1460
> ConjecturalRegulator(E);
4.488874747706661735767268062641
> HeegnerDiscriminants(E,-200,-100);
[ -119, -111 ]
> P := HeegnerPoints(E,-119);
> P;
<122760944933399668 9*t^10 - 106261506377143984603*t^9 +
    24598266676939452036 84*t^8 - 12539974356047058417320*t^7 -
    29852470882365441140834 3*t^6 + 44408766844345979261611 75*t^5 +
    75735490991206189798332 41*t^4 - 39393804886040638610811 3130*t^3 -
    21531810713569162829866 8863*t^2 + 13958289016298162706904 004974*t
    + 38624559371249968900024945369, 1>
```

The function automatically performs a heuristic check that the polynomial has the right properties, using reduction mod small primes. A more expensive check is the following.

```
> G := GaloisGroup( P[1] );
> IsIsomorphic(G,DihedralGroup(10));
true
```

The extension of $\mathbb{Q}(-119)$ defined by the polynomial is the class field. We now obtain a nice representation of it, and use this to compute the height of the point (which is a bit quicker than computing the height directly).

```
> K<u> := NumberField(P[1]);
> L<v>, m := OptimizedRepresentation(K);
> _<y> := PolynomialRing(Rationals());  // use 'y' for printing
> DefiningPolynomial(L);
y^10 + 2*y^9 - y^8 - 7*y^7 - 7*y^6 + 10*y^5 + 13*y^4 - 9*y^3 - 5*y^2 +
    5*y - 1
> PT := Points(ChangeRing(E,L),m(u))[1];  // y-coord is defined over L
> Height(PT);
6.97911761109714376876370533
```

133.2.7 Analytic Information

Periods(E: parameters): CrvEll -> [FldComElt]

Precision: RngIntElt	Default:

Returns the sequence of periods of the Weierstrass $\wp$-function associated to the (rational) elliptic curve E, to Precision digits. The first element of the sequence is the real period. The function accepts a non-minimal model, and returns the periods corresponding to that model. As with many algorithms involving analytic information on elliptic curves, the implementation exploits an AGM-trick due to Mestre. There is some functionality for elliptic curves over number fields (package level code), though the exact normalisation is not always given.

Periods(E, k): CrvEll, RngIntElt -> [FldComElt]

Precision: RngIntElt	Default:

Given an elliptic curve over a number field an integer denoting the kth real embedding, calculate the periods. This uses the C implementation after turning the coefficients of the defining equation into real numbers. Again the first period in the returned array should be real.

EllipticCurveFromPeriods(om: parameters): [FldComElt] -> CrvEll

```
Epsilon: FldReElt                    Default: 0.001
```

Given two complex numbers ω_1, ω_2 such that ω_2/ω_1 is in the upper half-plane that correspond to an integral model of an elliptic curve over $\mathbf{Q}$, return such a minimal model of such a curve. This uses the classical `Eisenstein` series, with the parameter `Epsilon` indicating how close to integers the computed $-27c_4$ and $-54c_6$ need to be.

`RealPeriod(E: parameters): CrvEll -> FldReElt`

```
Precision: RngIntElt                 Default:
```

Returns the real period of the Weierstrass $\wp$-function associated to the elliptic curve E to `Precision` digits.

`EllipticExponential(E, z): CrvEll, FldComElt -> [ FldComElt ]`

Given a rational elliptic curve E and a complex number z, the function computes the pair $[\wp(z), \wp'(z)]$ where $\wp(s)$ is the Weierstrass $\wp$-function. A nonminimal model may be given, though it should be integral. The algorithm used is taken from [Coh93] for small precision, and Newton iteration on `EllipticLogarithm` is used for high precision. The function returns a sequence rather than a point on the curve.

`EllipticExponential(E, k, z): CrvEll, RngIntElt, FldComElt -> [ FldComElt ]`

Given an elliptic curve E over a number field, an integer denoting the kth real embedding, and a complex number z, the function computes the pair $[\wp(z), \wp'(z)]$ where $\wp(s)$ is the Weierstrass $\wp$-function. As with `Periods`, the function proceeds by using the C implementation over the reals. There is no restriction that the curve equation be integral. The function returns a sequence rather than a point on the curve.

`EllipticExponential(E, S): CrvEll, [ FldRat ] -> [ FldComElt ]`

Given a rational elliptic curve E and a sequence $S = [p, q]$, where p and q are rational numbers, this function computes the elliptic exponential of p times the `RealPeriod` and q times the imaginary period.

`EllipticLogarithm(P): PtEll[FldRat] -> FldComElt`

```
Precision: RngIntElt                 Default:
```

Denote by ω_1, ω_2 the periods of the Weierstrass $\wp$-function related to E. This function returns the elliptic logarithm $\phi(P)$ of the point P, such that $-\omega_1/2 \leq \mathrm{Re}(\phi(P)) < \omega_1/2$ and $-\omega_2/2 \leq \mathrm{Im}(\phi(P)) < \omega_2/2$. The value is returned to `Precision` digits. As with `Periods`, a non-minimal model can be given (though the model must be integral), and the `EllipticLogarithm` will be computed with respect to it. The algorithm is again an AGM-trick due to Mestre.

`EllipticLogarithm(P, k): PtEll[FldNum], RngIntElt -> FldComElt`

```
Precision: RngIntElt                 Default:
```

Given an elliptic curve over a number field and a point on it, return the elliptic logarithm associated to the kth real embedding. There is no restriction that the model be integral.

`EllipticLogarithm(E, S): CrvEll, [ FldComElt ] -> FldComElt`

```
Precision: RngIntElt                 Default:
Check    : BoolElt                   Default: true
```

Given an elliptic curve E and a sequence $S = [z_1, z_2]$, where z_1 and z_2 are complex numbers approximating a point P on E, this function returns the elliptic logarithm $\phi(P)$. For details see the previous intrinsic. When `Check` is false, z_1 and z_2 need not have any relation to a point on the curve, in which case only the x-coordinate matters up to (essentially) a choice of sign in the resulting logarithm.

pAdicEllipticLogarithm(P, p: parameters): PtEll, RngIntElt -> FldLocElt

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

For a point P on an elliptic curve E which is a minimal model and a prime p, returns the p-adic elliptic logarithm of P to `Precision` digits. The order of P must not be a power of p.

Example: Elliptic Exponential and Logarithm (ex-ca5786)

Verify that `EllipticExponential` and `EllipticLogarithm` are inverses.

```
> C<I>:=ComplexField(96);
> E:=EllipticCurve([0,1,1,0,0]);
> P:=EllipticExponential(E,0.571+0.221*I); P;
[ 1.65694760521018623886829817 5 + -1.78544018006768141861869594 7*I,
-2.41707728470031533038520025 7 + 3.89415379266120836015385353 0*I ]
> EllipticLogarithm(E,P);
0.57100000000000000000000000 + 0.22100000000000000000000000*I
```

Add points on a curve via Napier's method.

```
> E:=EllipticCurve([0,0,1,-7,6]);
> P1:=E![0,2]; P2:=E![2,0]; P3:=E![1,0];
> L:=EllipticLogarithm(E!P1)+EllipticLogarithm(E!P2)+EllipticLogarithm(E!P3);
> L;
0.52174821975779076985312019 7937 + 1.48054826824141499330733111064*i
> P:=EllipticExponential(E,L);
> [Round(Real(x)) : x in P];
[ 4, -7 ]
> P1+P2+P3;
(4 : -7 : 1)
```

Example: Periods with Real Embeddings (ex-bd92df)

Verify that the use of real embeddings works with `Periods`.

```
> Q := RationalsAsNumberField();
> E := EllipticCurve([0,1,1,0,0]);
> Periods (E : Precision:=15);
[ 5.46868952996758, 2.73434476498379 + 1.36318241817043*$.1 ]
> EQ := BaseChange (E, Q);
> Periods (EQ,1 : Precision:=15);
[ 5.46868952996758, 2.73434476498379 + 1.36318241817043*$.1 ]
> _<s> := QuadraticField(5);
> E := EllipticCurve([s+1,1,-1,-s,2]);
> Periods (E,1 : Precision:=15);
[ 3.87725004777908, 1.93862502388954 + 0.697984279657861*$.1 ]
> Periods (E,2 : Precision:=15);
[ 3.20379220349709, 1.60189610174854 + 1.31468730940820*$.1 ]
```

And go back-and-forth with exponentials and logarithms.

```
> K<s> := QuadraticField(5);
> E := EllipticCurve([s+1,1,-1,-s,2]);
> P := Points(E,1)[1];
> Q := Points(E,1/2*(-9*s+11))[1];
> R := Points(E,1/2*(25*s + 55))[1];
> lP1 := EllipticLogarithm(P,1); lP1;
0.78650175450078897133315483 0602
```

(continues on next page)

(continued from previous page)

```
> P1 := EllipticExponential(E,1,lP1); P1;
 0.99999999999999999999999999998 - 8.0983345870190157557217617618 7E-40*i,
-2.8541019662496845446137605030 9 - 5.8039855123066225145773292483 5E-40*i ]
> Abs(Conjugates(P[2])[1]-P1[2]); // y-coord diff is close to 0
3.1554436208840472217002922514 8E-30
> lQ2 := EllipticLogarithm(Q,2);
> lR2 := EllipticLogarithm(R,2);
> QR2 := EllipticExponential(E,2,lQ2+lR2);
[ 0.2360679774997896964091736687 32 - 9.2879463010315367072062005167 9E-40*i,
-1.0901699437494742410229341718 3 - 2.9516848688945524106880642679 0E-39*i ]
> Abs(Conjugates((Q+R)[1])[2]-QR2[1]);  // compare x-coords
1.1832913578315177084821097774 2E-30
```

`RootNumber(E): CrvEll -> RngIntElt`

Calculates the global root number of an elliptic curve E defined over $\mathbb{Q}$. This is equal to the sign in the functional equation of the L-series associated to the curve. The method used is to take a product of local root numbers over the primes of bad reduction.

`RootNumber(E, p): CrvEll, RngIntElt -> RngIntElt`

Calculates the local root number at a prime p of an elliptic curve E. The method used is due to Halberstadt; for $p > 3$ it is a straightforward calculation involving the valuation of the discriminant, while for $p = 2, 3$ it involves a more careful analysis of the reduction type.

`AnalyticRank(E): CrvEll -> RngIntElt, FldReElt`

```
AnalyticRank: Verbose                    Default: 1
Precision   : RngIntElt                  Default: 5
```

Determine the analytic rank of the rational elliptic curve E. The algorithm used is heuristic, computing derivatives of the L-function $L(E, s)$ at $s = 1$ until one appears to be nonzero. Local power series methods are used to compute the special functions used for higher derivatives. The function returns the first nonzero derivative $L^{(r)}(1)/r!$ as a second argument. The precision is optional, and is taken to be 17 bits if omitted; the time taken can increase dramatically if this is increased much beyond 50 digits as the time to compute the special functions then starts to dominate the running time.

`ConjecturalRegulator(E): CrvEll -> FldReElt, RngIntElt`

```
Precision: RngIntElt                  Default:
```

Using the `AnalyticRank` function, this function calculates an approximation, assuming that the Birch–Swinnerton-Dyer conjecture holds, to the product of the regulator of the elliptic curve E and the order of the Tate–Shafarevich group. The (assumed) analytic rank is returned as a second value.

`ConjecturalRegulator(E, v): CrvEll, FldReElt -> FldReElt`

Given an elliptic curve E with L-function $L(E, s)$ and the value v of the first nonzero derivative $L^{(r)}(1)/r!$, this function calculates an approximation, assuming that the Birch–Swinnerton-Dyer conjecture holds, to the product of the regulator of the elliptic curve E and the order of its Tate–Shafarevich group.

Example: Analytic Rank of a Rank 6 Curve (ex-293403)

```
> E:=EllipticCurve([1,1,0,-2582,48720]);
> time r, Lvalue := AnalyticRank(E : Precision:=9); r,Lvalue;
Time: 17.930
6 320.781164
> ConjecturalRegulator(E,Lvalue);
```

(continues on next page)

(continued from previous page)

```
68.2713770
> time G, map := MordellWeilGroup(E : HeightBound := 8); G;
Time: 21.540
Abelian Group isomorphic to Z (6 copies)
Defined on 6 generators (free)
> Regulator([map(g): g in OrderedGenerators(G)]);
68.2713769
```

Example: Conjectural Regulator of a Curve (ex-6c18ce)

```
> ConjecturalRegulator(EllipticCurve([0,-1,1,0,0]) : Precision := 8);
1.0000000 0
> ConjecturalRegulator(EllipticCurve([0,-1,1,0,0]) : Precision := 21);
1.000000000000000000 0
> ConjecturalRegulator(EllipticCurve([0,-1,1,0,0]) : Precision := 35);
1.00000000000000000000000000000000 0
> ConjecturalRegulator(EllipticCurve([0,-1,1,0,0]) : Precision := 50);
1.00000000000000000000000000000000000000000000000000 0
```

`ModularDegree(E): CrvEll -> RngIntElt`

```
ModularDegree: Verbose                    Default: 1
```

Determine the modular degree of a rational elliptic curve E. The algorithm used is described in [Wat02]. One computes the special value $L(Sym^2 E, 2)$ of the motivic symmetric-square L-function of the elliptic curve, and uses the formula

$$deg(\phi) = L(Sym^2 E, 2)/(2 * \Pi * \Omega) * (Nc^2) * E_p(2)$$

where Ω is the area of the fundamental parallelogram of the curve, N is the conductor, c is the Manin constant, and $E_p(2)$ is a product over primes whose square divides the conductor. The Manin constant is assumed to be 1 except in the cases described in [SW], where it is conjectured that the optimal curves for parameterizations from $X_1(N)$ and $X_0(N)$ are different. A warning is given in these cases. The optimal curve for parameterizations from $X_1(N)$ is assumed to be the curve in the isogeny class of E that has minimal Faltings height (maximal Ω). The algorithm is based upon a sequence of real-number approximations converging to an integer — the use of verbose printing for `ModularDegree` allows the user to see the sequence of approximations.

Example: Modular Degree of an Elliptic Curve (ex-d0adbe)

First we define a space of ModularForms.

```
> M:=ModularForms(Gamma0(389),2);
```

The first of the newforms associated to M corresponds to an elliptic curve.

```
> f := Newform(M,1); f;
q - 2*q^2 - 2*q^3 + 2*q^4 - 3*q^5 + 4*q^6 - 5*q^7 + q^9 + 6*q^10 - 4*q^11 +
O(q^12)
> E := EllipticCurve(f); E;
Elliptic Curve defined by y^2 + y = x^3 + x^2 - 2*x over Rational Field
```

We can compute the modular degree of this using modular symbols.

```
> time ModularDegree(ModularSymbols(f));
40
Time: 0.200
```

Or via the algorithm based on elliptic curves.

```
> time ModularDegree(E);
40
Time: 0.000
```

The elliptic curve algorithm is capable of handling examples of high level, particularly when bad primes have multiplicative reduction.

```
> E := EllipticCurve([0,0,0,0,-(10^4+9)]);
> Conductor(E);
14425931664
> time ModularDegree(E);
6035544576
Time: 3.100
```

133.2.8 Integral and S-integral Points

Let E be an elliptic curve defined over the rational numbers $\mathbb{Q}$ and denote by $S = \{p_1, \ldots, p_{s-1}, \infty\}$ a finite set of primes containing the prime at infinity. There are only finitely many S-integral points on E. (That is, points where the denominators of the coordinates are only supported by primes in S.) Note that the point at infinity on E is never returned, since it is supported at every prime.

The following algorithms use the technique of linear forms in complex and p-adic elliptic logarithms. They were initially implemented by Emmanuel Hermann in the late 90s, and then Steve Donnelly made various improvements. The main theoretical reference is Stroeker and Tzanakis [ST94], and Tzanakis [ST96] for the case of a quartic equation.

The main routine here is `(S)IntegralPoints` for an elliptic curves. The functions listed afterwards, for determining integral points on various other kinds of genus one curves, are applications of the main routine.

`IntegralPoints(E): CrvEll[FldRat] -> [ PtEll ]`

```
FBasis      : [ PtEll ]                    Default:
SafetyFactor: RngIntElt                    Default: 1
```

Given an elliptic curve E over $\mathbb{Q}$, having integral coefficients, this returns a sequence containing all the integral points on E, up to sign (only one of P and $-P$ is listed).

The algorithm involves first computing generators of the Mordell–Weil group, using the tools available in Magma for this. Alternatively, the user may precompute generators and pass them to `IntegralPoints` as the optional parameter `FBasis`. This should be a sequence of points on E that are independent modulo torsion (for instance, as returned by `ReducedBasis`). `IntegralPoints` will then find all integral points on E that are in the group generated by `FBasis` and torsion.

If the optional argument `SafetyFactor` is specified, the search phase at the final step is extended as a safety check. (The number of points checked is increased by approximately this factor, which must be at least 1.)

`SIntegralPoints(E, S): CrvEll, SeqEnum -> [ PtEll ]`

```
FBasis      : [ PtEll ]                    Default:
SafetyFactor: RngIntElt                    Default: 1
```

Given an elliptic curve E over the rationals and a set S of finite primes, returns the sequence of all S-integral points, up to sign (only one of P and $-P$ is listed).

The optional parameters `FBasis` and `SafetyFactor` have the same meaning as in `IntegralPoints`.

Example: All Integral Points on a Curve `(ex-5cfac6)`

We find all integral points on a certain elliptic curve:

```
> E := EllipticCurve([0, 17]);
> Q, reps := IntegralPoints(E);
> Q;
[ (-2 : -3 : 1), (8 : 23 : 1), (43 : -282 : 1), (4 : 9 : 1),
(2 : -5 : 1), (-1 : 4 : 1), (52 : -375 : 1), (5234 : 378661 : 1) ]
> reps;
[
    [ <(-2 : -3 : 1), 1> ],
    [ <(-2 : -3 : 1), 2> ],
    [ <(-2 : -3 : 1), 1>, <(4 : -9 : 1), -2> ],
    [ <(4 : -9 : 1), -1> ],
    [ <(-2 : -3 : 1), 1>, <(4 : -9 : 1), -1> ],
    [ <(-2 : -3 : 1), 1>, <(4 : -9 : 1), 1> ],
    [ <(-2 : -3 : 1), 2>, <(4 : -9 : 1), 1> ],
    [ <(-2 : -3 : 1), 1>, <(4 : -9 : 1), 3> ]
]
```

We see here that the chosen basis consists of the points $(-2 : -3 : 1)$ and $(4 : -9 : 1)$, and that coefficients which are zero are omitted.

Example: S-Integral Points on a Rank 2 Curve (ex-406509)

We find all S-integral points on an elliptic curve of rank 2, for the set of primes $S = \{2, 3, 5, 7\}$.

```
> E := EllipticCurve([-228, 848]);
> Q := SIntegralPoints(E, [2, 3, 5, 7]);
> for P in Q do P; end for;    // Print one per line
(4 : 0 : 1)
(-11 : 45 : 1)
(16 : 36 : 1)
(97/4 : -783/8 : 1)
(-44/9 : -1160/27 : 1)
(857/4 : -25027/8 : 1)
(6361/400 : -282141/8000 : 1)
(534256 : -390502764 : 1)
(946/49 : -20700/343 : 1)
(-194/25 : -5796/125 : 1)
(34/9 : 172/27 : 1)
(814 : 23220 : 1)
(13 : 9 : 1)
(-16 : 20 : 1)
(1/4 : -225/8 : 1)
(52 : -360 : 1)
(53 : 371 : 1)
(16/49 : 9540/343 : 1)
(-16439/1024 : -631035/32768 : 1)
(34 : 180 : 1)
(-2 : 36 : 1)
(-14 : -36 : 1)
(14 : -20 : 1)
(754 : -20700 : 1)
(94/25 : -828/125 : 1)
(2 : 20 : 1)
(94 : 900 : 1)
(-818/49 : 468/343 : 1)
(49/16 : -855/64 : 1)
(196 : -2736 : 1)
(629/25 : 13133/125 : 1)
```

(continues on next page)

(continued from previous page)

```
(1534/81 : -42020/729 : 1)
(8516/117649 : -1163623840/40353607 : 1)
```

IntegralQuarticPoints(Q): [RngIntElt] -> [SeqEnum]

If Q is a sequence of five integers $[a, b, c, d, e]$ where e is a square, this function returns all integral points (modulo negation) on the curve $y^2 = ax^4 + bx^3 + cx^2 + dx + e$.

IntegralQuarticPoints(Q, P): [RngIntElt], [RngIntElt] -> [SeqEnum]

If Q is a list of five integers $[a, b, c, d, e]$ defining the hyperelliptic quartic $y^2 = ax^4+bx^3+cx^2+dx+e$ and P is a sequence representing a rational point $[x, y]$, this function returns all integral points on Q.

SIntegralQuarticPoints(Q, S): [RngIntElt], [RngIntElt] -> [SeqEnum]

Given a sequence of integers $Q = [a, b, c, d, e]$ where a is a square, this function returns all S-integral points on the quartic curve $y^2 = ax^4 + bx^3 + cx^2 + dx + e$ for the set S of finite primes.

Example: Integral Points on a Quartic (ex-8d65c5)

We find all integral points (modulo negation) on the curve $y^2 = x^4 - 8x^2 + 8x + 1$. Since the constant term is a square we can use the first form and do not have to provide a point as well.

```
> IntegralQuarticPoints([1, 0, -8, 8, 1]);
[
    [ 2, -1 ],
    [ -6, 31 ],
    [ 0, 1 ]
]
```

SIntegralLjunggrenPoints(Q, S): [RngIntElt], [RngIntElt] -> [SeqEnum]

Given a sequence of integers $Q = [a, b, c, d]$, this function returns all S-integral points on the curve $C : ay^2 = bx^4 + cx^2 + d$ for the set S of finite primes, provided that C is nonsingular.

SIntegralDesbovesPoints(Q, S): [RngIntElt], [RngIntElt] -> [SeqEnum]

Given a sequence of integers $Q = [a, b, c, d]$, this function returns all S-integral points on $C : ay^3 + bx^3 + cxy + d = 0$ for the set S of finite primes, provided that C is nonsingular.

Example: S-Integral Points on a Desboves Curve (ex-1bb564)

We find the points supported by $[2, 3, 5, 7]$ on the curve $9y^3 + 2x^3 + xy + 8 = 0$.

```
> S := [2, 3, 5, 7];
> SIntegralDesbovesPoints([9, 2, 1, 8], S);
[
    [ 1, -1 ],
    [ -94/7, 172/21 ],
    [ -11/7, 2/7 ],
    [ 5, -3 ],
    [ 2, -4/3 ],
    [ 2/7, -20/21 ],
    [ -8/5, -2/15 ]
]
```

133.2.9 Elliptic Curve Database

Magma includes John Cremona's database of all elliptic curves over $\mathbb{Q}$ of small conductor (up to 499 999 as of September 2019). This section defines the interface to that database.

For each conductor in the range stored in the database the curves with that conductor are stored in a number of isogeny classes. Each curve in an isogeny class is isogenous (but not isomorphic) to the other

curves in that class. Isogeny classes are referred to by number rather than the alphabetic form given in the Cremona tables.

All of the stored curves are global minimal models.

EllipticCurveDatabase(: parameters) -> DB

CremonaDatabase(: parameters) -> DB

BufferSize: RngIntElt	Default: 10000

This function returns a database object which contains information about the elliptic curve database and is used in the functions which access it. The optional parameter `BufferSize` controls the size of an internal buffer — see the description of `SetBufferSize` for more information.

SetBufferSize(D, n): DB, RngIntElt

The elliptic curve database D uses an internal buffer to cache disk reads; if the buffer is large enough then the entire file can be cached and will not need to be read from the disk more than once. On the other hand, if only a few curves will be accessed then a large buffer is not especially useful. `SetBufferSize` can be used to set the size n (in bytes) of this buffer. There are well defined useful minimum and maximum sizes for this buffer, and values outside this range will be treated as the nearest useful value.

LargestConductor(D): DB -> RngIntElt

Returns the largest conductor of any elliptic curve stored in the database. It is an error to attempt to refer to larger conductors in the database.

ConductorRange(D): DB -> RngIntElt, RngIntElt

Returns the smallest and largest conductors stored in the database. It is an error to attempt to refer to conductors outside of this range.

D: DB -> RngIntElt

NumberOfCurves(D): DB -> RngIntElt

Returns the number of elliptic curves stored in the database.

NumberOfCurves(D, N): DB, RngIntElt -> RngIntElt

Returns the number of elliptic curves stored in the database for conductor N.

NumberOfCurves(D, N, i): DB, RngIntElt, RngIntElt -> RngIntElt

Returns the number of elliptic curves stored in the database in the i-th isogeny class for conductor N.

NumberOfIsogenyClasses(D, N): DB, RngIntElt -> RngIntElt

Returns the number of isogeny classes stored in the database for conductor N.

EllipticCurve(D, N, I, J): DB, RngIntElt, RngIntElt, RngIntElt -> CrvEll

EllipticCurve(D, N, S, J): DB, RngIntElt, MonStgElt, RngIntElt -> CrvEll

Returns the J-th elliptic curve of the I-th isogeny class of conductor N from the database. I may be specified either as an integer (first form) or as a label like `"A"` (second form).

EllipticCurve(D, S): DB, MonStgElt -> CrvEll

EllipticCurve(S): MonStgElt -> CrvEll

Returns a representative elliptic curve with label S (e.g., `"101a"` or `"101a1"`) from the specified database (or if not specified, from the Cremona database).

Random(D): DB -> CrvEll

Returns a random curve from the database.

CremonaReference(D, E): DB, CrvEll -> MonStgElt

`CremonaReference(E): CrvEll -> MonStgElt`

Returns the database reference to the minimal model for E (e.g., `"101a1"`). E must be defined over $\mathbb{Q}$ and its conductor must lie within the range of the database. The second form of this function must open the database for each call, so if it is being used many times the database should be created once and the first form used instead.

Example: Conductors in the Cremona Database (ex-d41275)

```
> D := CremonaDatabase();
> #D;
3064705
> minC, maxC := ConductorRange(D);
> minC, maxC;
1 499999
> &+[ NumberOfCurves(D, C) : C in [ minC .. maxC ] ];
3064705
```

The conductor in that range with the most curves is 369600.

```
> S := [ NumberOfCurves(D, C) : C in [ minC .. maxC ] ];
> cond := maxval + minC - 1 where _,maxval := Max(S);
> cond;
369600
> NumberOfCurves(D, cond);
1386
> NumberOfIsogenyClasses(D, cond);
612
```

The unique curve of conductor 5077 has rank 3.

```
> NumberOfCurves(D, 5077);
1
> E := EllipticCurve(D, 5077, 1, 1);
> E;
Elliptic Curve defined by y^2 + y = x^3 - 7*x + 6 over Rational Field
> CremonaReference(D, E);
5077a1
> Rank(E);
3
```

`EllipticCurves(D, N, I): DB, RngIntElt, RngIntElt -> [ CrvEll ]`

`EllipticCurves(D, N, S): DB, RngIntElt, MonStgElt -> [ CrvEll ]`

Returns the sequence of elliptic curves in the I-th isogeny class for conductor N from the database. I may be specified either as an integer (first form) or as a label like `"A"` (second form).

`EllipticCurves(D, N): DB, RngIntElt -> [ CrvEll ]`

The sequence of elliptic curves for conductor N from the database.

`EllipticCurves(D, S): DB, MonStgElt -> [ CrvEll ]`

The sequence of elliptic curves with label S from the database. S may specify just a conductor (like `"101"`), or both a conductor and an isogeny class (like `"101A"`).

`EllipticCurves(D): DB -> [ CrvEll ]`

The sequence of elliptic curves stored in the database. Note: this function is extremely slow due to the number of curves involved. Where possible it would be much better to iterate through the database instead (see example).

Example: Iterating over the Cremona Database (ex-850f6b)

Here are two ways to iterate through the database:

```
> D := CremonaDatabase();
> &+[ Discriminant(E) : E in D ];
-7948594696775435281517286995609618816722789395 36
> sum := 0;
> for E in D do sum +:= Discriminant(E); end for;
> sum;
-7948594696775435281517286995609618816722789395 36
```

Now we choose a random curve from the database, and look up the other curves in the same isogeny class. (Note: when this input is entered, the random curve will probably be different to the example shown here!)

```
> E := Random(D);
> E;
Elliptic Curve defined by y^2 = x^3 - 225*x over Rational Field
> r := CremonaReference(E);
> r;
7200bg1
// remove the numeric characters from the end of the string r
> while r[#r] in "0123456789" do Prune(~r); end while;
> r;
7200bg
> EllipticCurves(D, r);
[
    Elliptic Curve defined by y^2 = x^3 - 225*x over Rational Field,
    Elliptic Curve defined by y^2 = x^3 - 2475*x - 47250 over Rational Field,
    Elliptic Curve defined by y^2 = x^3 - 2475*x + 47250 over Rational Field,
    Elliptic Curve defined by y^2 = x^3 + 900*x over Rational Field
]
> E in $1;
true
```

133.3 Curves over Number Fields

The functions in this section are for elliptic curves defined over number fields. For the most part, the functionality is a subset of that available for curves over $\mathbb{Q}$, with functions having similar names and arguments.

Note that functions related to Mordell–Weil groups and descent are in Section *Mordell–Weil Groups and Descent Methods* rather than this section.

133.3.1 Local Invariants

The routines listed here, when nontrivial, are based on an implementation of Tate's algorithm.

`Conductor(E): CrvEll -> RngOrdIdl`

The conductor is part of the data computed by `LocalInformation` (described below).

`BadPlaces(E): CrvEll -> SeqEnum`

Given an elliptic curve E defined over a number field K, returns the places of K of bad reduction for E. i.e., the places dividing the discriminant of E.

`BadPlaces(E, L): CrvEll, FldNum -> SeqEnum`

Given an elliptic curve E defined over a number field K and a number field L, such that K is a subfield of L, returns the places of L of bad reduction for E.

`LocalInformation(E, P): CrvEll, RngOrdIdl -> Tup, CrvEll`

```
UseGeneratorAsUniformiser: BoolElt                    Default: false
```

Implements Tate's algorithm for the elliptic curve E over a number field. This intrinsic computes local reduction data at the prime ideal P, and a local minimal model. The model is not required to be integral on input. Output is $\langle P, v_p(d), f_p, c_p, K, s\rangle$ and E_{min} where P is the prime ideal, $v_p(d)$ is the valuation of the local minimal discriminant, f_p is the valuation of the conductor, c_p is the Tamagawa number, K is the Kodaira Symbol, and s is `false` if the curve has non-split multiplicative reduction at P and `true` otherwise. E_{min} is a model of E (integral and) minimal at P.

When the optional parameter `UseGeneratorAsUniformiser` is set `true`, the computation checks whether P is principal, and if so, uses generator of P as the uniformiser. This means that at primes other that P, the returned model will still be integral or minimal if the given curve was.

`LocalInformation(E): CrvEll -> [ Tup ]`

Return a sequence of the tuples returned by `LocalInformation(E, P)` for all prime ideals P in the decomposition of the discriminant of E in the maximal order of the number field the elliptic curve E is over.

`Reduction(E, p): CrvEll, RngOrdIdl -> CrvEll, Map`

Given an elliptic curve E over a number field given by a model which is integral at p and has good reduction at p, returns an elliptic curve over the residue field of p which represents the reduction of E at p. The reduction map is also returned.

133.3.2 Complex Multiplication

`HasComplexMultiplication(E): CrvEll -> BoolElt, RngIntElt`

Given an elliptic curve E over a number field, the function determines whether the curve has complex multiplication and, if so, also returns the discriminant of the CM quadratic order. The algorithm uses fairly straightforward analytic methods, which are not suited to very high degree j-invariants with CM by orders with discriminants more than a few thousand.

133.3.3 Heights

`NaiveHeight(P): PtEll -> FldPrElt`

Given a point P on an elliptic curve over a number field K, the function returns the naive height of P; i.e., the absolute logarithmic height of the x-coordinate.

`Height(P : parameters): PtEll -> FldPrElt`

```
Precision: RngIntElt                  Default: 27
Extra    : RngIntElt                  Default: 8
```

Given a point P on an elliptic curve defined over a number field K, returns the Néron-Tate height $\hat{h}(P)$. (This is based on the absolute logarithmic height of the x-coordinate.) The parameter `Precision` may be used to specify the desired precision of the output. The parameter `Extra` is used in the cases when one of the points $2^n P$ gets too close to the point at infinity, in which case there is a huge loss in precision in the archimedean height and increasing `Extra` remedies that problem.

`HeightPairingMatrix(P : parameters): [PtEll] -> AlgMatElt`

Compute the height pairing matrix for an array of points. Same parameters as above.

`LocalHeight(P, Pl : parameters): PtEll, PlcNumElt -> FldPrElt`

```
Precision: RngIntElt                  Default: 0
Extra    : RngIntElt                  Default: 8
```

Given a point P on an elliptic curve defined over a number field K, and a place Pl (finite or infinite) of K, returns the local height $\lambda_{Pl}(P)$ of P at Pl. The parameter `Precision` sets the precision of the output. A value of 0 takes the default precision.

When Pl is an infinite place, the parameter `Extra` can be used to remedy the huge loss in precision when one of the points $2^n P$ gets too close to the point at infinity.

`HeightDifferenceBounds(E): CrvEll -> FldReElt, FldReElt`

`HeightDifferenceLowerBound(E): CrvEll -> FldReElt`

`HeightDifferenceUpperBound(E): CrvEll -> FldReElt`

These functions return bounds for the difference between the naive and canonical heights on the elliptic curve E. Here all heights are absolute logarithmic heights with respect to the x-coordinate on E. The functions return real numbers l and u such that $l \leq$ `NaiveHeight(P) - Height(P)` $\leq u$ for all points $P \in E(K)$, where K is the base field of E. Note that the lower bound l is negative.

The bounds are the strongest available, choosing between the Cremona–Prickett–Siksek bounds and the Silverman bounds.

`CPSHeightBounds(E): CrvEll -> FldReElt, FldReElt`

The Cremona–Prickett–Siksek bounds for the height difference.

`SilvermanHeightBounds(E): CrvEll -> FldReElt, FldReElt`

The Silverman bounds for the height difference.

133.3.4 Integral Points

`IntegralPoints(E): CrvEll[FldNum] -> [ PtEll ]`

```
FBasis      : [ PtEll ]                    Default:
SafetyFactor: RngIntElt                    Default: 2
```

Given an elliptic curve E over a number field which is a direct extension of the rational field, this intrinsic returns a sequence containing all integral points on the given model of E, up to sign (only one of P and $-P$ is listed).

The parameters `FBasis` and `SafetyFactor` have exactly the same meaning as in the function *IntegralPoints* for curves over $\mathbb{Q}$.

Currently, `FBasis` *must* be specified: it must be a (nonempty) sequence of points in the Mordell–Weil group of E. The routine finds all integral points in the group generated by the given points.

The base field of E must also current be totally real and E must have nontrivial torsion.

133.3.5 Elliptic Curve Chabauty

This refers to a method for finding the rational points on a curve, if the curve admits a suitable map to an elliptic curve over some extension field. The method was developed by Nils Bruin (see [Bru03] or [Bru]). The first intrinsic follows a Mordell–Weil sieve based strategy, similar to that used in `Chabauty` for genus 2 curves (see [BS10]).

`Chabauty(MWmap, Ecov): Map, MapSch -> SetEnum, RngIntElt`

```
InertiaDegreeBound: RngIntElt              Default: 20
SmoothBound       : RngIntElt              Default: 50
PrimeBound        : RngIntElt              Default: 30
IndexBound        : RngIntElt              Default:
InitialPrimes     : RngIntElt              Default: 50
EllChab           : Verbose                Default: 3
```

Let E be an elliptic curve defined over a number field K. This function attempts to determine the subset of $E(K)$ consisting of those points that have a $\mathbb{Q}$-rational image under a given map $E \to \mathbf{P}^1$.

The arguments are:

- a map `MWmap`: $A \to E$ from an abstract abelian group into $E(K)$,
- a map `Ecov` : $E \to \mathbf{P}^1$ defined over K.

The returned values are a finite subset V of A and an integer R. The set V consists of all the points in the image of A in $E(K)$ that have a $\mathbb{Q}$-rational image under the given map `Ecov`. If the index of A in $E(K)$ is finite and coprime to R then V consists of all points in $E(K)$ with $\mathbb{Q}$-rational image. Changing the various optional parameters can change the performance of this routine considerably. In general, it is not proven that this routine will return at all.

The parameter `InertiaDegreeBound` determines the maximum inertia degree of primes at which local information is obtained.

Only places for which the group order of the rational points on the reduction of E is smooth relative to `SmoothBound` are used.

The Mordell–Weil sieving process only uses information from $E(K)/BE(K)$, where B is smooth relative to `PrimeBound`.

If `IndexBound` is specified, then the returned value R will only contain prime divisors from `IndexBound`. Setting this parameter may make it impossible for the routine to terminate.

`Chabauty(MWmap, Ecov, p): Map, MapSch, RngIntElt -> RngIntElt, SetEnum, RngIntElt, Tup`

```
Cosets   : Tup                          Default:
Aux      : SetEnum                      Default:
Precision: RngIntElt                    Default:
Bound    : RngIntElt                    Default:
EllChab  : Verbose                      Default: 3
```

Let E be an elliptic curve defined over a number field K. This function bounds the set of points in $E(K)$ whose images under a given map $E \to \mathbf{P}^1$ are $\mathbb{Q}$-rational.

The arguments are as follows:

- a map `MWmap` : $A \to E$ from an abstract group into $E(K)$ (as returned by `MordellWeilGroup`),
- a map of varieties `Ecov` : $E \to \mathbf{P}^1$ defined over K, and
- a rational prime p, such that E and the map `Ecov` have good reduction at primes above p.

The returned values are N, V, R, and L as follows. Let G denote the image of A in $E(K)$.

- N is an upper bound for the number of points $P \in G$ such that `Ecov`$(P) \in \mathbf{P}^1(\mathbb{Q})$ (note that N can be ∞).
- V is a set of elements of A found by the program that have images in $\mathbf{P}^1(\mathbb{Q})$,
- R is a number with the following property: if $[E(K) : G]$ is finite and coprime to R, then N is also an upper bound for the number of points $P \in E(K)$ such that `Ecov`$(P) \in \mathbf{P}^1(\mathbb{Q})$, and
- L is a collection of cosets in A such that $(\bigcup L) \cup V$ contains all elements of A that have images in $\mathbf{P}^1(\mathbb{Q})$.

If `Cosets` $(< C_j >)$ are supplied (a coset collection of A), then the bounds and results are computed for $A \cap (\bigcup C_j)$.

If `Aux` is supplied (a set of rational primes), then the information at p is combined with the information at the other primes supplied.

If `Precision` is supplied, this is the p-adic precision used in the computations. If not, a generally safe default is used.

If `Bound` is supplied, this determines a search bound for finding V.

The algorithm used is based on [Bru03] and [Bru02].

Further examples and discussion can be found in [Bru].

Example: Elliptic Curve Chabauty over a Number Field (ex-73eaae)

This example is motivated and explained in detail in [Bru] (Section 7). Let E be the elliptic curve

$$y^2 = x^3 + (-3\zeta^3 - \zeta + 1)dx^2 + (-\zeta^2 - \zeta - 1)d^2x, \quad d = 2\zeta^3 - 2\zeta^2 - 2$$

over $K := \mathbb{Q}(\zeta)$ where ζ is a primitive 10th root of unity.

```
> _<z> := PolynomialRing(Rationals());
> K<zeta> := NumberField(z^4-z^3+z^2-z+1);
> OK := IntegerRing(K);
> d := 2*zeta^3-2*zeta^2-2;
> E<X,Y,Z> := EllipticCurve(
>         [ 0, (-3*zeta^3-zeta+1)*d, 0, (-zeta^2-zeta-1)*d^2, 0 ]);
```

Next we determine as much as possible of $E(K)$, allowing Magma to choose the method.

```
> success, G, GtoEK := PseudoMordellWeilGroup(E);
> G;
Abelian Group isomorphic to Z/2 + Z/2 + Z + Z
Defined on 4 generators
Relations:
    2*G.1 = 0
    2*G.2 = 0
> success;
true
```

Here G is an abstract group and `GtoEK` injects it into $E(K)$. Since the flag is `true`, this is a subgroup of finite, odd index in $E(K)$. Next, we determine the points $(X : Y : Z)$ in this subgroup for which the function

$$u : E \to \mathbf{P}^1 : (X : Y : Z) \to d(-X + (\zeta^3 - 1)Z : X + d(-\zeta^3 - \zeta)Z)$$

takes values in $\mathbf{P}^1(\mathbb{Q})$.

```
> P1 := ProjectiveSpace(Rationals(),1);
> u := map< E->P1 | [-X + (zeta^3 - 1)*d*Z, X+(-zeta^3-zeta)*d*Z] >;
> V, R := Chabauty( GtoEK, u);
> V;
{
    0,
    G.3 - G.4,
    -G.3 + G.4,
    G.3 + G.4,
    -G.3 - G.4
}
> R;
320
```

We see that the routine assumed that the image of `GtoEK` is 2- and 5-saturated in $E(K)$. We can ask it to not assume anything outside 2.

```
> V2, R := Chabauty( GtoEK, u: IndexBound:= 2);
> V eq V2;
true
> R;
16
```

This means that we have found all points in $E(K)$ that have a rational image under u. If one wants to find the images of these points then it is necessary in this example to first extend u (by finding alternative defining equations for it) so that it is defined on all the points.

```
> u := Extend(u);
> [ u( GtoEK(P) ) : P in V ];
[ (-1 : 1), (-1/3 : 1), (-1/3 : 1), (-3 : 1), (-3 : 1) ]
```

Alternatively, we can apply Chabauty's method without Mordell–Weil sieving.

```
> N, V, R, C := Chabauty( GtoEK, u, 3);
> N;
5
> V;
{
    0,
    G.3 - G.4,
    -G.3 + G.4,
    G.3 + G.4,
    -G.3 - G.4
}
> R;
4
```

The `Chabauty` calculations prove that there are at most N elements in G whose image under u is $\mathbb{Q}$-rational. Also, V is a set of elements with this property. Since here $N = 5 = \#V$, these are the only such elements. Moreover, the calculations prove that *if* $[E(K) : G]$ is coprime to R, then N is actually an upper bound on the number of elements in $E(K)$ whose image under u is $\mathbb{Q}$-rational. We know from the `PseudoMordellWeilGroup` computation that $[E(K) : G]$ is odd. Therefore we have solved our problem for $E(K)$, not just for G.

133.3.6 Auxiliary Functions for Etale Algebras

This section contains machinery for number fields (and more generally "etale algebras", i.e. algebras of the form $\mathbb{Q}[x]/p(x)$), intended to perform the number field calculations that are involved in computing Selmer groups of geometric objects.

It has become conventional to refer to "the p-Selmer group" of a number field K (or, more generally, of an etale algebra) relative to a finite set S of K-primes. It means the finite subgroup $K(S, p)$ of $K^*/(K^*)^p$, consisting of those elements whose valuation at every prime outside S is a multiple of p.

`AbsoluteAlgebra(A): RngUPolRes -> SetCart, Map`

```
Fields  : SetEnum                    Default: { }
EtaleAlg: Verbose                    Default: 1
```

Given a separable commutative algebra over $\mathbb{Q}$, an absolute number field, or a finite field, the function returns the isomorphic direct sum of absolute fields as a cartesian product and the isomorphisms to and from this product. The optional parameter `Fields` enables the user to suggest representations of the absolute fields. If this function finds it needs a field isomorphic to one occurring in the supplied list, then it will use the given field. Otherwise it will construct a field itself. The isomorphism is returned as a second argument. If called twice on the same algebra, it will recompute if the Fields argument is given. Otherwise it will return the result computed the first time.

`pSelmerGroup(A, p, S): RngUPolRes, RngIntElt, SetEnum[RngOrdIdl] -> GrpAb, Map`

```
EtaleAlg: Verbose                    Default: 1
```

Returns the p-Selmer group of a semi-simple algebra A. The set S of ideals should be prime ideals of the underlying number field. The group returned is the direct sum of the p-Selmer groups of the irreducible summands of A and the map is the obvious embedding in the multiplicative group of A. An implied restriction is that `BaseRing(A)` be of type `FldNum`. See also `pSelmerGroup(p, S):`

RngIntElt, { RngOrdIdl } → GrpAb, Map. If an algebra over the rationals is required, create a degree 1 extension by

```
RationalsAsNumberField();
```

LocalTwoSelmerMap(P): RngOrdIdl -> Map

Let K be the number field associated with the prime ideal P. The map returned is $K^* \to K_P^*/K_P^{*2}$, where K_P is the completion of K at P. The codomain is represented as a finite abelian group.

LocalTwoSelmerMap(A, P): RngUPolRes, RngOrdIdl -> Map, SeqEnum

Let K be the base field of the commutative algebra A and let P be a prime ideal in K. Then this function returns a map $A^* \to A^*/A^{*2} \otimes K_P$, where K_P is the completion of K at P. The codomain is represented as a finite abelian group.

This map is computed by using LocalTwoSelmerMap(Q) for the various extensions Q of P to the fields making up AbsoluteAlgebra(A). The map returned is essentially the direct sum of all these maps. For technical purposes, one may also wish to use the components of the map coming from each number field; these are given by the second return value, which is a sequence of records (ordered in the same way as the results are concatenated in the returned map). Each record contains items i, p, map and vmap. Here i is an index indicating which number field in AbsoluteAlgebra(A) the record corresponds to, p is an extension of P to a prime in that number field, map is LocalTwoSelmerMap(p), and vmap is the valuation map at p on that number field.

Example: Absolute Algebra of an Etale Algebra (ex-bd3f0b)

```
> P<x> := PolynomialRing(Rationals());
> A := quo<P | x^3 - 1>;
> AA := AbsoluteAlgebra(A);
> AA;
Cartesian Product
<Number Field with defining polynomial x - 1 over the Rational Field,
 Number Field with defining polynomial x^2 + x + 1 over the Rational Field>
```

133.3.7 Analytic Information

RootNumber(E, P): CrvEll, RngOrdIdl -> RngIntElt

The local root number of the elliptic curve E (defined over a number field) at the prime ideal P. The formulae are due to Rohrlich, Halberstadt, Kobayashi, and the Dokchitser brothers.

RootNumber(E): CrvEll -> RngIntElt

Calculates the global root number of an elliptic curve E defined over a number field K. This is the product of local root numbers over all places of K (-1 from each infinite place), and is the (conjectural) sign in the functional equation relating $L(E/K, s)$ to $L(E/K, 2-s)$.

AnalyticRank(E): CrvEll -> RngIntElt, FldReElt

```
Precision: RngIntElt                    Default: 6
```

Determine the analytic rank of the elliptic curve E, which is defined over a number field K. The algorithm used is heuristic, computing derivatives of the L-function $L(E/K, s)$ at $s = 1$ until one appears to be nonzero; it also assumes the analytic continuation and the functional equation for the L-function. The function returns the first nonzero derivative $L^{(r)}(1)/r!$ as a second argument. The precision is optional, and is taken to be 6 digits if omitted.

ConjecturalRegulator(E): CrvEll -> FldReElt, RngIntElt

```
Precision: RngIntElt                    Default: 10
```

Using the AnalyticRank function, this function calculates an approximation, assuming that the Birch–Swinnerton-Dyer conjecture holds, to the product of the regulator of the elliptic curve E

and the order of the Tate–Shafarevich group. The (assumed) analytic rank is returned as a second value.

ConjecturalSha(E, Pts): CrvEll, SeqEnum[PtEll] -> FldReElt

```
Precision: RngIntElt                    Default: 6
```

For an elliptic curve E defined over a number field K and a sequence of points in $E(K)$ which purportedly form its basis modulo torsion, computes the conjectural order of the Tate–Shafarevich group Ш(E/K). This function computes the product of the regulator and Ш from the Birch–Swinnerton-Dyer conjecture (using `ConjecturalRegulator`) and divides by the determinant of the height pairing matrix for the points supplied in `Pts`. It returns 0 if the points are linearly dependent or they generate a group of rank less than the analytic rank. If the points generated a subgroup of index $n > 1$, it returns $n^2 \cdot |Ш|$.

133.3.8 Elliptic Curves of Given Conductor

This section describes search routines for finding elliptic curves with given conductor, or with good reduction outside a given set of primes. The aim is not to provably find all such curves; in most cases, this would be a very difficult task using current algorithms. Rather the aim is to efficiently search for these curves, using a variety of techniques, taking advantage of all available information, and taking advantage of all the tools available in Magma which can be applied to the problem.

The routine is very much more effective when some traces of Frobenius are known for the desired curve. The principal application of this is for finding elliptic curves that match known modular forms.

These functions are effective for finding elliptic curves over $\mathbb{Q}$, as well as over number fields. For this, one must create $\mathbb{Q}$ using `RationalsAsNumberField()`.

EllipticCurveSearch(N, Effort): RngOrdIdl, RngIntElt -> SeqEnum

EllipticCurveWithGoodReductionSearch(S, Effort): Set, RngIntElt -> SeqEnum

EllipticCurveWithGoodReductionSearch(N, Effort): RngOrdIdl, RngIntElt -> SeqEnum

```
Full    : BoolElt                       Default: false
Max     : RngIntElt                     Default:
Primes  : SeqEnum                       Default:
Traces  : SeqEnum                       Default:
ECSearch: Verbose                       Default: 2
```

These functions perform a search for elliptic curves with specified conductor(s). The first function finds curves with conductor equal to the given ideal N. The "good reduction" functions find curves with conductors divisible only by primes that belong to the given set S or that divide the given ideal N. The functions aim to find as many distinct isogeny classes as possible (or `Max` classes, if `Max` is specified). A sequence is returned containing all curves found that have the specified conductor(s). It will not return more than one curve in the same isomorphism class. It may return several curves in the same isogeny class, although no attempt is made to find all isogenous curves.

The second argument, `Effort`, is an integer which controls how much effort is used before returning; the running time depends on this variable in a roughly linear way, except with significant step-ups at values where an additional technique begins to be used. Currently, `Effort := 400` is the smallest value where all available techniques are used. (This may be changed in the future.)

There are two ways to specify an early stopping condition:

(i) If the optional argument `Max` is set to some positive integer, the routine terminates as soon as `Max` non-isogenous curves are found.

(ii) The optional arguments `Primes` and `Traces` may be used together to specify some traces of Frobenius of the desired curve(s). Here `Primes` is a sequence of prime ideals coprime to the conductor(s), and `Traces` is a sequence of integers of the form $[\ a_P : P \in \text{Primes}\ ]$. Alternatively, `Traces` may be a sequence of sequences of that form. The routine aims to find

curves whose traces match the specified sequences, and terminates as soon as one curve of each kind has been found. The algorithm uses the trace information to cut down the search by a large factor.

Warning: ALL curves found during the search are returned (possibly including curves whose traces do not match the specified data).

If an early stopping condition has been specified, the routine initially tries with lower effort, so that it will succeed quickly for easier input. The effort is incremented exponentially up to the specified `Effort`. (This is a breadth-first search strategy. The algorithm involves doing a large number of independent searches, and each elliptic curve can be found in only one of these searches, so it is necessary to alternate between them. There is no way to reliably predict how much effort will be needed, but the search usually takes far longer when the effort is larger than necessary.) To override this behaviour and begin with the full specified `Effort`, the option `Full` should be used. (This is intended for the case where the same search has already been done with a lower `Effort`).

If `SetVerbose("ECSearch", 1)` is used, information about the search is printed during computation. In particular, curves with the desired conductor are printed when they are found, so that they can be obtained without waiting for the routine to finish.

133.4 Curves over p-adic Fields

The functions in this section are for elliptic curves defined over p-adic fields. They provide an interface to the same code for Tate's algorithm that is used for curves over number fields.

133.4.1 Local Invariants

`Conductor(E): CrvEll -> FldPadElt`

The conductor of the elliptic curve E defined over a p-adic field.

`LocalInformation(E): CrvEll -> Tup, CrvEll`

Implements Tate's algorithm for the elliptic curve E over a p-adic field. This intrinsic computes local reduction data and a local minimal model. The model is not required to be integral on input. Output is $\langle P, v_p(d), f_p, c_p, K, s\rangle$ and E_{min} where P is the uniformizer of the ground field, $v_p(d)$ is the valuation of the local minimal discriminant, f_p is the valuation of the conductor, c_p is the Tamagawa number, K is the Kodaira Symbol, and s is `false` if the curve has non-split multiplicative reduction and `true` otherwise. E_{min} is an integral minimal model of E.

`RootNumber(E): CrvEll -> RngIntElt`

The local root number of the elliptic curve E (defined over a p-adic field).

133.5 Mordell–Weil Groups and Descent Methods

The remainder of the chapter describes the functions and methods for determining the Mordell–Weil group (the group of rational points) for a curve defined over $\mathbb{Q}$ or a number field.

The Mordell–Weil theorem states that for an elliptic curve E defined over a global field F (such as $\mathbb{Q}$ or a number field), the set $E(F)$ of points on E with F-rational coordinates forms a finitely generated abelian group. This is called the *Mordell–Weil group*, and the $\mathbb{Z}$-rank of the free part is called the *Mordell–Weil rank* of E.

It is an open problem to give an explicit algorithm which can determine the Mordell–Weil group of any elliptic curve over $\mathbb{Q}$ (or any number field). The general approach is to use the method of n-descent, for one or more suitable integers n. There are two main problems with this. Firstly, n-descent cannot determine the rank of E if the Tate–Shafarevich group $Ш(E)$ has elements of order n. It is an open conjecture that, for fixed E, this occurs for only finitely many values of n. The second problem is more practical: one cannot implement n-descent for general n.

Magma has implementations of descent over $\mathbb{Q}$ for $n = 2, 4, 8, 3, 9, 5$ as well as for isogenies of various degrees. It is the only software that has a *complete* implementation of 2-descent over number fields (in

the sense that 2-coverings are reduced and therefore can be used to search for points). In addition, the Cassels-Tate pairing on the 2-Selmer group is implemented over all global fields, and also on the 4-Selmer group over $\mathbb{Q}$. All these techniques (and some others) are combined in the main functions to determine ranks and generators. Each of the main functions returns a result together with the status of the result, i.e. whether the rank has been proven, or the group returned is known to be the full Mordell–Weil group.

Prior to version 2.21, the commands `MordellWeilGroup`, `Rank`, `Generators`, etc, did not use all available techniques. Furthermore they did not return the status of their results, but instead a warning was printed in the event of an unproven result; no such warnings are printed now.

Magma provides several ways to ask for information about ranks and Mordell–Weil groups.

(i) The main commands are `Rank`, `RankBounds`, `MordellWeilGroup`, `Generators`. These apply the available techniques in a suitable order.

(ii) At a slightly lower level, the routine `MordellWeilShaInformation` follows the same procedure, and also prints a summary of what is known about the rank, generators, and sha after applying each technique. The information about sha is also returned at the end.

(iii) The techniques may be called individually. This is often necessary to get the best results in hard examples – the automated procedure used by the higher level routines cannot always make the best choices. All the descent techniques are described later in this chapter; the other main techniques are `AnalyticRank` and `HeegnerPoint`.

133.5.1 Torsion

`TorsionSubgroup(E): CrvEll -> GrpAb, Map`

`TorsionSubgroup(H): SetPtEll -> GrpAb, Map`

Given an elliptic curve E defined over $\mathbb{Q}$ or a number field, this returns an abelian group A isomorphic to the torsion subgroup of the Mordell–Weil group, and a map from this abstract group A to the elliptic curve providing the isomorphism.

For a curve over $\mathbb{Q}$, by a theorem of Mazur, A is either C_k (for k in $\{1..10\}$ or 12) or $C_2 \times C_{2k}$ (for k in $\{1..4\}$). When there are two generators, they are chosen in such a way that the first generator has order 2.

The algorithm (for curves over number fields) is based on the `TorsionBound` described in this section. (For curves over $\mathbb{Q}$ a similar method is used.)

`TwoTorsionSubgroup(E): CrvEll -> GrpAb, Map`

This returns the group of 2-torsion points on the elliptic curve E (returned in the same way as the `TorsionSubgroup`). Note that this is a very cheap computation.

`TorsionBound(E, n): CrvEll, RngIntElt -> RngIntElt`

Given an elliptic curve E defined over a number field, returns a bound on the size of the torsion subgroup of E. This is done by considering at least n primes of good reduction (with early exit in some cases, such as when the torsion is shown to be only two-torsion).

`pPowerTorsion(E, p): CrvEll, RngIntElt -> GrpAb, Map`

`Bound: RngIntElt`	`Default:`

Given an elliptic curve E defined over $\mathbb{Q}$ or a number field, this returns the p-power torsion subgroup of E over its base field (returned in the same way as the `TorsionSubgroup`). A bound on the size of the p-power torsion subgroup may be given, as the parameter `Bound`, to cut short the computation in hard cases.

133.5.2 Mordell–Weil Group and Rank

`RankBounds(H: parameters): SetPtEll -> RngIntElt, RngIntElt`

`RankBounds(E: parameters): CrvEll -> RngIntElt, RngIntElt`

`MordellWeilRankBounds(H: parameters): SetPtEll -> RngIntElt, RngIntElt`

MordellWeilRankBounds(E: parameters): CrvEll -> RngIntElt, RngIntElt

Effort: RngIntElt	Default: 1

Given an elliptic curve E defined over $\mathbb{Q}$ or a number field, this returns lower and upper bounds on the rank of the Mordell–Weil group of E.

The parameter **Effort** must be an integer at least 1 (the default). Its usage is explained under *MordellWeilShaInformation*.

Rank(H: parameters): SetPtEll -> RngIntElt, BoolElt

Rank(E: parameters): CrvEll -> RngIntElt, BoolElt

RankBound(E): CrvEll -> RngIntElt, BoolElt

MordellWeilRank(H: parameters): SetPtEll -> RngIntElt, BoolElt

MordellWeilRank(E: parameters): CrvEll -> RngIntElt, BoolElt

Effort: RngIntElt	Default: 1

Given an elliptic curve E defined over $\mathbb{Q}$ or a number field, this returns a lower bound r for the Mordell–Weil group of E. The second value returned is a boolean, which is **true** iff it is known that the rank is exactly r.

The parameter **Effort** must be an integer at least 1 (the default). Its usage is explained under *MordellWeilShaInformation*.

MordellWeilGroup(H: parameters): SetPtEll -> GrpAb, Map, BoolElt, BoolElt

MordellWeilGroup(E: parameters): CrvEll -> GrpAb, Map, BoolElt, BoolElt

AbelianGroup(H: parameters): SetPtEll -> GrpAb, Map, BoolElt, BoolElt

AbelianGroup(E: parameters): CrvEll -> GrpAb, Map, BoolElt, BoolElt

Effort : RngIntElt	Default: 1
HeightBound: RngIntElt	Default: 15

The Mordell–Weil group of an elliptic curve E defined over $\mathbb{Q}$. The first two values returned are an abelian group A and a map m from A to E. The map m provides an isomorphism between the abstract group A and the Mordell–Weil group.

Additionally the function returns two booleans. The first boolean is **true** iff the rank of the group returned is known to be the rank of the curve. The second boolean is **true** iff the group returned is known to be the full Mordell–Weil group.

The parameter **Effort** must be an integer at least 1 (the default). Its usage is explained under *MordellWeilShaInformation*.

The parameter **HeightBound** limits the search for points directly on E to the given naive height. (Note: this affects only saturation of the group, not determination of the rank.)

Generators(H): SetPtEll -> [PtEll]

Generators(E): CrvEll -> [PtEll]

This function returns the generators of the **MordellWeilGroup** of the elliptic curve E (listed in the same order).

NumberOfGenerators(H): SetPtEll -> RngIntElt

NumberOfGenerators(E): CrvEll -> RngIntElt

Ngens(H): SetPtEll -> RngIntElt

Ngens(E): CrvEll -> RngIntElt

This is equivalent to **NumberOfGenerators**(**MordellWeilGroup**(E)).

Saturation(points, n): [PtEll], RngIntElt -> [PtEll]

TorsionFree:	BoolElt	Default: false
OmitPrimes :	[RngIntElt]	Default: []
Check :	BoolElt	Default: true

Given a sequence of points on an elliptic curve E over the rationals or a number field, and an integer n, this function returns a sequence of points generating a subgroup of $E(\mathbb{Q})$ which contains the given points and which is *p-saturated* for all primes p up to n. (A subgroup S is p-saturated in a group G if there is no intermediate subgroup H for which the index $[H : S]$ is finite and divisible by p.)

If **OmitPrimes** is set to be a sequence of primes, then the group is not checked to be p-saturated for those primes. If **TorsionFree** is set to **true**, torsion points are omitted from the result (that is, the result contains independent generators modulo torsion). If **Check** is set to **false**, the input sequence of points are assumed to be independent modulo torsion.

Saturation(points): [PtEll] -> [PtEll]

TorsionFree:	BoolElt	Default: false
Check :	BoolElt	Default: true

Given a sequence of points on an elliptic curve E over the rationals or a number field, returns generators of the saturation of the given points in the Mordell–Weil group of E.

Example: Local Data and Mordell-Weil Group (ex-0fa71f)

```
> E := EllipticCurve([73, 0]);
> E;
Elliptic Curve defined by y^2 = x^3 + 73*x over Rational Field
> Factorization(Integers() ! Discriminant(E));
[ <2, 6>, <73, 3> ]
> BadPrimes(E);
[ 2, 73 ]
> LocalInformation(E);
[ <2, 6, 6, 1, II, true>, <73, 3, 2, 2, III, true> ]
> G, m := MordellWeilGroup(E);
> G;
Abelian Group isomorphic to Z/2 + Z + Z
Defined on 3 generators
Relations:
    2*G.1 = 0
> Generators(E);
[ (0 : 0 : 1), (36 : -222 : 1), (4/9 : 154/27 : 1) ]
> 2*m(G.1);
(0 : 1 : 0)
```

Example: Curve of Moderately Large Rank (ex-ee2e20)

Here is a curve with moderately large rank; we do not attempt to compute the full group since that would be quite time-consuming.

```
> E := EllipticCurve([0, 0, 0, -9217, 300985]);
> T, h := TorsionSubgroup(E);
> T;
Abelian Group of order 1
> time RankBounds(E);
7 7
Time: 0.070
```

This curve was well-behaved in that the computed lower and upper bounds on the rank are the same, and so we know that we have computed the rank exactly. Here is a curve where that is not the case:

```
> E := EllipticCurve([0, -1, 0, -140, -587]);
> time G, h := MordellWeilGroup(E);
Warning: rank computed (2) is only a lower bound
(It may still be correct, though)
Time: 0.250
> RankBounds(E);
2 3
```

The difficulty here is that the Tate–Shafarevich group of E is not trivial, and this blocks the 2-descent process used to compute the rank. We can compute the `AnalyticRank` of the curve to be 2, but this is only conjecturally equal to the rank. In cases like this, higher level descents may provide a more definitive answer; the machinery for two- and four-descents described in the next two sections can be used to confirm that the rank is indeed 2. In any case, we can certainly get the group with rank equal to the lower bound.

```
> G;
Abelian Group isomorphic to Z + Z
Defined on 2 generators (free)
> S := Generators(E);
> S;
[ (-6 : -1 : 1), (-7 : 1 : 1) ]
> [ Order(P) : P in S ];
[ 0, 0 ]
> h(G.1) eq S[1];
true
> h(G.2) eq S[2];
true
> h(2*G.1 + 3*G.2);
(-359741403/57729604 : 940675899883/438629531192 : 1)
> 2*S[1] + 3*S[2];
(-359741403/57729604 : 940675899883/438629531192 : 1)
```

As mentioned above, the rank of this curve is actually 2, so we have computed generators for the full group of E.

MordellWeilShaInformation(E: parameters): CrvEll -> [RngIntElt], [PtEll], [Tup]

DescentInformation(E: parameters): CrvEll -> [RngIntElt], [PtEll], [Tup]

```
RankOnly: BoolElt                    Default: false
ShaInfo : BoolElt                    Default: false
Silent  : BoolElt                    Default: false
Effort  : RngIntElt                  Default: 1
```

This is a special function which uses all relevant Magma machinery to obtain as much information as possible about the Mordell–Weil group and the Tate–Shafarevich group of the given elliptic curve E over $\mathbb{Q}$ or a number field. The tools used include 2-descent, 4-descent, Cassels-Tate pairings, 8-descent, 3-descent. When the conductor is not too large, analytic methods (analytic rank and Heegner points) are also used. The information is progressively refined as the tools are applied (in order of their estimated cost), and a summary is printed at each stage. At the end, all information obtained is returned in three sequences.

The first sequence returned contains lower and upper bounds on the Mordell–Weil rank of $E(\mathbb{Q})$, and the second is the sequence of independent generators of the Mordell–Weil group (modulo the torsion subgroup) that have been found. The third sequence returned contains the information obtained about the Tate–Shafarevich group $Sha(E)$: letting r_n denote the largest integer such

that $\mathbb{Z}/n\mathbb{Z}$ is contained in $Sha(E)$, the tuple `<n, [l,u]>` would indicate that the computations prove $l \leq r_n \leq u$.

By default, the routine attempts to determine the rank and generators, and returns whenever it succeeds in determining them. When `RankOnly` is set to true, it returns as soon as the Mordell–Weil rank has been determined. When `ShaInfo` is set to true, it additionally probes the structure of $Sha(E)$ using all available tools.

The parameter `Effort` must be an integer at least 1 (the default). Currently it has the following effect. For curves over $\mathbb{Q}$: when the effort is 1, the algorithm terminates before attempting the hardest methods (`FourDescent` is used but not `EightDescent` or `ThreeDescent`), otherwise all methods are used. For curves over number fields: all available methods are always used, and the `Effort` is a linear multiplier that controls the time spent in searching for points in some of the methods.

Example: Rank 2 Curve of Conductor 389 (ex-fb2edb)

Conductor 389 is the smallest conductor of a curve with rank 2:

```
> E := EllipticCurve("389a1");
> time rank, gens, sha := MordellWeilShaInformation(E);

Torsion Subgroup is trivial
Analytic rank = 2
The 2-Selmer group has rank 2
Found a point of infinite order.
Found 2 independent points.
After 2-descent:
    2 <= Rank(E) <= 2
    Sha(E)[2] is trivial
(Searched up to height 100 on the 2-coverings.)

Time: 0.280
```

We now take a curve of conductor 571, which has rank 0 and nontrivial Tate–Shafarevich group:

```
> E := EllipticCurve("571a1");
> time rank, gens, sha :=MordellWeilShaInformation(E);

Torsion Subgroup is trivial
Analytic rank = 0
     ==> Rank(E) = 0

Time: 0.010
```

We must specify that we want information about the Tate–Shafarevich group:

```
> time rank, gens, sha :=MordellWeilShaInformation(E : ShaInfo);

Torsion Subgroup is trivial
Analytic rank = 0
     ==> Rank(E) = 0
The 2-Selmer group has rank 2
After 2-descent:
    0 <= Rank(E) <= 0
    (Z/2)^2 <= Sha(E)[2] <= (Z/2)^2
(Searched up to height 10000 on the 2-coverings.)
The Cassels-Tate pairing on Sel(2,E)/E[2] is
[0 1]
[1 0]
```

(continues on next page)

(continued from previous page)

```
After using Cassels-Tate:
    0 <= Rank(E) <= 0
    (Z/2)^2 <= Sha(E)[4] <= (Z/2)^2
The 3-Selmer group has rank 0
After 3-descent:
    0 <= Rank(E) <= 0
    (Z/2)^2 <= Sha(E)[12] <= (Z/2)^2

Time: 0.840
```

133.5.3 Two-Descent

This section describes the main implementations of two-descent (by the "algebraic" algorithm) in Magma, for curves over $\mathbb{Q}$ and for curves over number fields.

The two-descent process determines the locally soluble 2-coverings of a given elliptic curve, returning them as hyperelliptic curves $C : y^2 = f(x)$ of degree four. The process breaks into two main parts: first to compute the 2-Selmer group, then to construct 2-covering curves corresponding to its elements. To obtain nice models of the coverings, minimisation and reduction techniques are used. This is done over (general) number fields using algorithms developed by Donnelly and Fisher.

A separate implementation is available for the case where E admits a 2-isogeny: this involves first computing the 2-coverings for the isogeny and its dual (in this case the covering maps have degree 2 instead of degree 4), and then lifting these to the level of a "full 2-descent".

This section also describes functions for dealing with 2-covering curves as binary quartic forms: invariant theory, and minimisation and reduction. This functionality overlaps with the package for genus one models (see *Models of Genus One Curves*). There is also a straightforward interface to the 2-Selmer group machinery.

TwoDescent(E: parameters): CrvEll -> [CrvHyp], [Map], Map

```
RemoveTorsion: BoolElt                  Default: false
RemoveGens   : { PtEll}                 Default: { }
WithMaps     : BoolElt                  Default: true
MinRed       : BoolElt                  Default:
TwoDescent   : Verbose                  Default: 1
```

Given an elliptic curve E over the rationals or a number field, this performs 2-descent on E producing 2-covering curves that represent the elements of the 2-Selmer group of E. The function returns these as a sequence of hyperelliptic curves, and also returns a corresponding list of maps (unless the parameter `WithMaps` is set to `false`).

If `RemoveTorsion` is `true`, the generators of the torsion subgroup are factored out from the group. If `RemoveGens` is given, the group spanned by the specified points is factored out.

For curves over number fields: a third item is returned which specifies the group structure on the set of 2-coverings. This is a map (with inverse) from an abstract group to the sequence of 2-coverings; the abstract group is either `TwoSelmerGroup(E)`, or the appropriate quotient in the case where `RemoveTorsion` or `RemoveGens` were specified.

Currently, for curves over number fields: the parameter `MinRed` controls whether the coverings are to be minimised and reduced. (This may be expensive for various reasons, especially when the field discriminant is not small: in particular, a large integer may need to be factored.)

AssociatedEllipticCurve(f): RngUPolElt -> CrvEll, Map

AssociatedEllipticCurve(C): CrvHyp -> CrvEll, Map

```
E: CrvEll                       Default:
```

Gives the minimal model of the elliptic curve associated with a two-covering given as a polynomial f or a hyperelliptic curve C, along with a map from points $[x, y]$ on the two-covering to the curve.

If an elliptic curve is given as E, this must be isomorphic to the Jacobian of C, and then the map returned will be a map to the given E.

`TwoCover(e): FldNumElt -> CrvHyp, Map`

`TwoCover(e): RngUPolResElt -> CrvHyp, Map`

The purpose of this function is to calculate the 2-covers that are returned by `TwoDescent` individually rather than all together.

The argument e is an element of a cubic extension A/F, where F is a number field, and where A may be either a number field over F or an affine algebra over F. The element e determines a 2-cover of some elliptic curve over F (by the construction given in the description of `DescentMaps`).

Example: Two-Descent on an Elliptic Curve (ex-17ccb5)

```
> SetSeed(1); // results may depend slightly on the seed
> E := EllipticCurve([0, 1, 0, -7, 6]);
> S := TwoDescent(E);
> S;
[
    Hyperelliptic Curve defined by y^2 = x^4 + 4*x^3 - 2*x^2 - 20*x + 9 over
    Rational Field,
    Hyperelliptic Curve defined by y^2 = x^4 - x^3 - 2*x^2 + 2*x + 1 over
    Rational Field,
    Hyperelliptic Curve defined by y^2 = 2*x^4 - 4*x^3 - 8*x^2 + 4*x + 10 over
    Rational Field
]
```

The curve E has three non-trivial two-descendants, hence its rank is at most 2. The first two curves yield obvious rational points, so we can find two independent points on E (and it has exact rank 2).

```
> pt_on_S1 := Points(S[1] : Bound:=10 )[1];
> pt_on_S1;
// We obtain the map from S[1] to E by
> _, phi := AssociatedEllipticCurve(S[1] : E:=E );
> phi( pt_on_S1 );
(1 : -1 : 1)
// Now do the same for the second curve.
> pts := Points(S[2] : Bound:=10 );
> if #pts eq 0 then // depends on random state
>     S[2], S[3] := S[3], S[2];
>     pts := Points(S[2] : Bound:=10 );
> end if;
> pt_on_S2 := pts[1];
> _, phi := AssociatedEllipticCurve(S[2] : E:=E );
> phi( pt_on_S2 );
(-3 : 3 : 1)
```

Two Descent Using Isogenies

`TwoIsogenyDescent(E : parameters): CrvEll -> SeqEnum[CrvHyp], List, SeqEnum[CrvHyp], List, MapSch, MapSch`

```
Isogeny        : MapSch                    Default:
TwoTorsionPoint: PtEll                     Default:
```

Given an elliptic curve E over $\mathbb{Q}$ admitting a 2-isogeny $\phi : E' \to E$, this function computes 2-coverings representing the nontrivial elements of the Selmer groups of ϕ and of the dual isogeny $\phi' : E \to E'$. These coverings are given as hyperelliptic curves $C : y^2 = quartic(x)$. Six objects are returned: (i) the sequence of coverings C of E for ϕ; (ii) the corresponding list of maps $C \to E$; (iii) and (iv) the coverings C' and maps $C' \to E'$ for ϕ'; (v) and (vi) the isogenies ϕ and ϕ' that were used.

`LiftDescendant(C): CrvHyp -> SeqEnum[ CrvHyp ], List, MapSch`

`TwoDescendantsOverTwoIsogenyDescendant(C): CrvHyp -> SeqEnum[ CrvHyp ], List, MapSch`

This routine performs a higher descent on curves arising in `TwoIsogenyDescent(E)` on an elliptic curve E over $\mathbb{Q}$. The curves obtained are 2-coverings of E in the sense of ordinary (full) `TwoDescent`; more precisely, they are exactly the set of 2-coverings D for which the covering map $D \to E$ factors through C. Up to isomorphism, they are a subset of the 2-coverings returned by `TwoDescent(E)`. The advantage of this approach is that it works entirely over the base field of E, whereas `TwoDescent` will in general compute a class group over a quadratic extension of the base field. The example below explains how to recover all coverings produced by `TwoDescent(E)` using the 2-isogeny approach.

This function accepts any curve C in the first sequence of curves returned by `TwoIsogenyDescent(E)` (these are the 2-isogeny-coverings of E). More generally it accepts any hyperelliptic curve of the form $y^2 = d_1x^4 + cx^2 + d_2$. A model for the associated elliptic curve E is then $y^2 = x(x^2 + cx + d_1d_2)$.

The function returns three objects: a sequence containing the covering curves D, a list containing the corresponding maps $D \to C$, and lastly the covering map $C \to E$ from the given curve to some model of its associated elliptic curve.

Invariants

`QuarticIInvariant(q): RngUPolElt -> RngIntElt`

`QuarticJInvariant(q): RngUPolElt -> RngIntElt`

`QuarticG4Covariant(q): RngUPolElt -> RngUPolElt`

`QuarticG6Covariant(q): RngUPolElt -> RngUPolElt`

`QuarticHSeminvariant(q): RngUPolElt -> RngIntElt`

`QuarticPSeminvariant(q): RngUPolElt -> RngIntElt`

`QuarticQSeminvariant(q): RngUPolElt -> RngIntElt`

`QuarticRSeminvariant(q): RngUPolElt -> RngIntElt`

Compute invariants, semivariants, and covariants, as in paper [Cre01], of a given quartic polynomial q. The G4 and G6 covariants are polynomials of degrees four and six respectively; the I and J invariants are integers that generate the ring of integer invariants of the polynomial and satisfy $J^2 - 4I^3 = 27\Delta(f)$. The names H and P have been used for essentially the same seminvariant in different papers; they are related by $H = -P$.

`QuarticNumberOfRealRoots(q): RngUPolElt -> RngUPolElt`

Using invariant theory, compute the number of real roots of a real quartic polynomial q.

`QuarticMinimise(q): RngUPolElt -> RngUPolElt, AlgMatElt`

This computes a minimal model of the quartic polynomial q over the rationals or a univariate rational function field.

Three objects are returned: the minimised quartic, the transformation matrix, and the scaling factor.

For further explanation see Chapter *Models of Genus One Curves*. The algorithm can be found in [CFS10].

`QuarticReduce(q): RngUPolElt -> RngUPolElt, AlgMatElt`

Given a quartic q, the algorithm of [Cre99] is applied to find the reduced quartic and the matrix that reduced it.

`IsEquivalent(f, g): RngUPolElt, RngUPolElt -> BoolElt`

Determines if the quartics f and g are equivalent.

133.5.4 Selmer Groups

First we give a short overview of the theory of Selmer groups. This enables us to fix the notation that is used in the naming of the Magma functions. For a more complete account, see [Sil86]. The actual algorithms to compute the Selmer groups are closer to the description in [Cas66].

Let E', E be elliptic curves over a number field K and let $\phi : E' \to E$ be an isogeny. The only cases currently implemented are where ϕ is a 2-isogeny, i.e., an isogeny of degree 2, and where $E' = E$ and ϕ is multiplication by 2.

Let $E'[\phi]$ be the kernel subscheme of ϕ. By taking Galois cohomology of the short exact sequence of schemes over K,

$$0 \to E'[\phi] \to E' \to E \to 0,$$

we obtain

$$E'(K) \to E(K) \to H^1(K, E'[\phi]) \to H^1(K, E').$$

For the middle map, we write $\mu : E(K) \to H^1(K, E'[\phi])$. Thus, $E(K)/\phi(E'(K))$ injects in $H^1(K, E'[\phi])$ and the image consists exactly of the cocycles that vanish in $H^1(K, E')$.

As an approximation to this image, we define the ϕ-Selmer group of E over K to consist of those cocycles $H^1(K, E'[\phi])$ that vanish in all restrictions $H^1(K_p, E')$, where p runs though all places of K. It fits in the exact sequence

$$0 \to S^{(\phi)}(E/K) \to H^1(K, E'[\phi]) \to \prod_p H^1(K_p, E')$$

The main application of Selmer groups is that they provide the bound: $\#E(K)/\phi E'(K) \leq \#S^{(\phi)}(E/K)$. If $\phi^* : E \to E'$ is the isogeny dual to ϕ, then $\phi \circ \phi^* : E \to E$ is multiplication by $d = \mathrm{Deg}(\phi)$. Thus, one can use the ϕ and ϕ^* Selmer groups to provide a bound on $\#E(K)/dE(K)$ and thus on the Mordell–Weil rank of $E(K)$.

Representation of $H^1(K, E'[\phi])$:

If ϕ is a 2-isogeny, then $H^1(K, E'[\phi]) \sim K^*/K^{*2}$. Thus, we can represent elements by $\delta \in K^*$. In Magma, the map μ corresponds to a representation of the map $E(K) \to K^*$. The map μ also accepts just x-coordinates.

If ϕ is multiplication-by-2 and $E : y^2 = f(x)$, we write $A = K[x]/(f(x))$ then

$$H^1(K, E[2]) \sim \{\delta \in A^*/A^{*2} | N_{A/K}(\delta) \in K^{*2}\}.$$

In fact, the full set A^*/A^{*2} corresponds to $H^1(K, E[2] \times \{\pm 1\})$.

The set $H^1(K, E'[\phi])$ also corresponds to the set of covers $\tau_\delta : T_\delta \to E$ over K, modulo isomorphy over K, that are isomorphic to $\phi : E' \to E$ over the algebraic closure of K (see [Sil86], Theorem X.2.2). In this section, we write `tor` for the map $\delta \mapsto \tau_\delta$.

`DescentMaps(phi): Map -> Map, Map`

`CasselsMap(phi): Map -> Map, Map`

`Fields: SetEnum` `Default: { }`

Given an isogeny $\phi : E \to E_1$ of elliptic curves over a number field K (or $\mathbb{Q}$), the function returns the connecting homomorphism

$$\mu : E_1(K) \to H^1(K, E[\phi]),$$

and a map τ sending an element of $H^1(K, E[\phi])$ to the corresponding homogeneous space. Here elements of $H^1(K, E[\phi])$ are represented as elements of $A^*/(A^*)^2$ (as described above). The maps are actually given as maps to, and from, A (rather than $A^*/(A^*)^2$).

The isogeny ϕ must be either a 2-isogeny or multiplication–by–2.

When ϕ is multiplication–by–two, then the computation of μ involves a call to `AbsoluteAlgebra`. The optional parameter `Fields` is passed on to that. If the fields mentioned in this set are found to be of any use, then these will be used and whatever class group and unit data stored on the fields will be used in subsequent computations.

When ϕ is multiplication–by–two, the second map τ accepts all elements $\delta \in A^*$. An element $\delta \in A^*$ represents an element of $H^1(K, E[2])$ if and only if δ must have square norm. In general, $\delta \in A^*$ represents an element of $H^1(K, E[2] \times \{\pm 1\})$, in which case $\tau(\delta)$ is the corresponding covering $T_\delta \to \mathbf{P}^1$. (This covering is a twist of the covering $E \to E \to \mathbf{P}^1$ given by $P \mapsto 2P \mapsto x(2P)$.)

`SelmerGroup(phi): Map -> GrpAb, Map, Map, SeqEnum, SetEnum`

```
Hints : SetEnum                         Default: { }
Raw   : BoolElt                         Default: false
Bound : RngIntElt                       Default: -1
Selmer: Verbose                         Default: 4
```

Given an isogeny $\phi : E \to E_1$ defined over a number field K, computes the associated Selmer group $Sel(\phi) := Sel^\phi(E/K)$.

The Selmer group is returned as a finite abelian group S, together with a map $AtoS : A \to S$, where A is as in the introduction. This is a map only in the Magma sense; it is defined only on a finite subset of A. Its "inverse" $S \to A$ provides the mathematically meaningful injection $S \hookrightarrow A^*/(A^*)^2$. The standard map

$$E_1(K) \to E_1(K)/\phi E(K) \to Sel(\phi)$$

is given by the composition of μ with $AtoS$, where $\mu : E_1(K) \to H^1(K, E[\phi])$ is the first map returned by `DescentMaps`.

If the optional parameter `Hints` is given, it is used as a list of x-coordinates to try first when determining local images. Supplying `Hints` does not change the outcome, but may speed up the computation.

The calculation of the Selmer group involves possibly expensive class group and unit group computations. If no such data has been precomputed, `SelmerGroup` will attempt to obtain this information unconditionally, unless `Bound` is positive. This bound is then passed on to any called `ClassGroup`. However, if such data is already stored, it will be used and subsequent results will be conditional on whatever assumptions were made while computing this information. If conditional results are desired (for instance, assuming GRH), one should precompute class group information on the codomain of `CasselsMap(phi)` prior to calling `SelmerGroup(phi)`.

If `Raw` is `true`, then three technical items are also returned. The first two of these, $toVec$ and FB, enable one to represent elements of the Selmer group in terms of a "factor base" FB consisting of elements of A that generate a relevant subgroup of $A^*/(A^*)^2$. The map $toVec$ sends an element of S to an exponent vector relative to FB. The map from S to A obtained by multiplying out the results is inverse to $AtoS$.

The final returned value (when `Raw` is `true`) is a set of `Hints` (just as in the optional parameter).

`TwoSelmerGroup(E): CrvEll -> GrpAb, Map, SetEnum, Map, SeqEnum`

```
Hints     : SetEnum                     Default: { }
Raw       : BoolElt                     Default: false
Bound     : RngIntElt                   Default: -1
TwoDescent: Verbose                     Default: 2
```

The 2-Selmer group of an elliptic curve defined over $\mathbb{Q}$ or a number field. The function simply calls `SelmerGroup` for the multiplication–by–two isogeny. The given model for E should be integral. The options and return values are the same as for `SelmerGroup`.

Example: Rank via the 2-Selmer Group (ex-dcb28c)

In this example, we determine the rank of $y^2 = x^3 + 9x^2 - 10x + 1$ by computing the 2-Selmer group.

```
> E := EllipticCurve([0,9,0,-10,1]);
> two := MultiplicationByMMap(E,2);
> mu, tor := DescentMaps(two);
```

The hard work: computing the Selmer group.

```
> S, AtoS := SelmerGroup(two);
> #S;
8
```

So the Selmer rank is 3. We deduce the following upper bound on the rank of $E(\mathbb{Q})$, taking into account any 2-torsion.

```
> RankBound(E : Isogeny := two);
3
```

In fact, there are 3 points: $(0,1), (1,1)$ and $(2,5)$.

```
> g1 := E![ 0, 1 ];
> g2 := E![ 1, 1 ];
> g3 := E![ 2, 5 ];
```

We now test these points for linear independence. It will follow that the points generate $E(K)/2E(K)$, which means they are independent nontorsion points in $E(K)$ (since we know there is no 2-torsion).

```
> IsLinearlyIndependent ([g1, g2, g3]);
true
```

Next we compute the homogeneous space associated to the point $g1 + g2$. It must have a rational point mapping to $g1 + g2$. Note that @@ always denotes "preimage" in Magma. The algebra for `AtoS` is not the same as the original cubic, so we must translate before applying the requisite map.

```
> K := NumberField(Modulus(Domain(AtoS)));
> L := NumberField(Polynomial([1,-10,9,1]));
> b, m := IsIsomorphic (K, L); assert b;
> theta := (Rationals()! (L.1 - m(K.1))) + Domain(AtoS).1;
> H, mp := TwoCover((g1+g2)[1] - theta : E:=E); H;
Hyperelliptic Curve defined by y^2 = 9*x^4 - 4*x^3 - 18*x^2 + 4*x + 13
```

The next command finds all $\mathbb{Q}$-rational points in the preimage of the point $g1 + g2$ on E. (In Magma the preimage is constructed as a scheme.)

```
> RationalPoints( (g1+g2) @@ mp);
{@ (-1 : -2 : 1) @}
```

Example: 2-Isogeny Descents with Full 2-Torsion (ex-b6ce90)

We consider the elliptic curve $E : y^2 = dx(x+1)(x+3)$ where d is the product of the primes less than 50. Since E has full 2-torsion, we can carry out 2-isogeny descent in three non-equivalent ways, resulting in three different rank bounds.

```
> P<x> := PolynomialRing(Integers());
> d := &*[ p : p in [1..50] | IsPrime(p) ];
> E := EllipticCurve(HyperellipticCurve(d*x*(x + 1)*(x + 3)));
```

The nontrivial 2-torsion points in $E(\mathbb{Q})$:

```
> A, mp := TorsionSubgroup(E);
> T := [ t : a in A | t ne E!0 where t := mp(a) ];
```

The corresponding 2-isogenies:

```
> phis := [ TwoIsogeny(t) : t in T ];
```

The rank bounds obtained from these isogenies:

```
> [ RankBound(E : Isogeny := phi) : phi in phis ];
[ 9, 5, 7 ]
```

We find [9, 5, 7]! Each descent gives a different rank bound. However, a full 2-descent gives:

```
> two := MultiplicationByMMap(E,2);
> RankBound(E : Isogeny := two);
1
```

Now doing full 2-descent on the three isogenous curves, we see where the obstacle for sharp rank bounds comes from.

```
> OtherTwos := [ MultiplicationByMMap(Codomain(phi), 2) : phi in phis ];
> [ RankBound(Domain(two) : Isogeny := two) : two in OtherTwos ];
[ 9, 5, 7 ]
```

We find [9, 5, 7] again.

Example: A Classic Curve from Kramer `(ex-fc7e9e)`

The next example is a classic one from [Kra81].

```
> E := EllipticCurve([0, 977, 0, 976, 0]);
> #TwoTorsionSubgroup(E);
4
> RankBound(E);
> ptsE := [E| [-4, 108], [4, 140]];
> IsLinearlyIndependent(ptsE); // they are non-torsion points
true
> // So E is really of rank 2
> d := 109;
> Ed := QuadraticTwist(E, d);
> Points(Ed, -976);
[ (-976 : 298656 : 1), (-976 : -298656 : 1) ]
```

Since Ed has a nontorsion point, its rank is at least 1. We will show that its rank is exactly 1, by comparing 2-descent on E and on its base change.

```
> _<x> := PolynomialRing(Rationals());
> K := NumberField(x^2 - d);
> EK := BaseChange(E, K);
> Ngens(TwoSelmerGroup(EK));
5
```

Note that the Selmer group contains the 2-torsion subgroup, so this tells us that the rank of $E(K)$ is at most 3, hence it is exactly 3 because we know 3 independent points. The rank of $E_d(\mathbb{Q})$ must equal the rank of $E(K)$ minus the rank of $E(Q)$, therefore $E_d(\mathbb{Q})$ has rank 1. This is smaller than the bound of 3 we get from a 2-descent on E_d alone:

```
> Ngens(TwoSelmerGroup(Ed));
5
```

We may confirm the result as follows (this performs higher descents).

```
> RankBound(Ed);
1 true
```

Example: Two-Selmer Groups and Two-Covers (ex-63feb8)

Here we give some examples of using `TwoDescent`, `TwoSelmerGroup`, and `TwoCover`.

```
> E := EllipticCurve( [ 0, 0, 1, -7, 6] ); // rank 3 curve
> T := TwoDescent(E);
> T[6];
Hyperelliptic Curve defined by y^2 = 3*x^4 - 10*x^3 + 10*x + 1 over
Rational Field
> G, m := TwoSelmerGroup(E);
> G.1 @@ m;
theta^2 - 12*theta + 33
> Parent($1); // Modulus has y^2 = modulus isomorphic to E
Univariate Quotient Polynomial Algebra in theta over Rational Field
with modulus theta^3 - 112*theta + 400
> TwoCover( (G.1 + G.2) @@ m);
Hyperelliptic Curve defined by y^2 = 3*x^4 - 10*x^3 + 10*x + 1 over
Rational Field
> TwoCover( Domain(m) ! 1 );
Hyperelliptic Curve defined by y^2 = 2*x^3 - 12*x^2 - 32*x + 196 over
Rational Field
```

133.5.5 The Cassels-Tate Pairing

The Tate–Shafarevich group of any elliptic curve E admits an alternating bilinear form on $Ш(E)$ with values in $\mathbb{Q}/\mathbb{Z}$, known as the Cassels-Tate pairing. The key property is that if $Ш(E)$ is finite (as conjectured), the Cassels-Tate pairing is non-degenerate. When restricted to the 2-torsion subgroup, one obtains a non-degenerate alternating bilinear form on $Ш(E)[2]/2Ш(E)[4]$, or equivalently on $Sel^2(E)$ modulo the image of $Sel^4(E)$, with values in $\mathbb{Z}/2\mathbb{Z}$.

This means that if C and D are 2-coverings of E and the pairing (C, D) has value 1, then both C and D represent elements of order 2 in $Ш(E)$, and moreover there are no locally solvable 4-coverings of E lying above them (in other words, `FourDescent(C)` and `FourDescent(D)` would both return an empty sequence). In this sense the Cassels-Tate pairing provides the same information as 4-descent, but is much easier to compute.

Similarly, for an element in $Ш(E)[4]/2Ш(E)[8]$, the values of the pairing between this element and all elements in $Ш(E)[2]/2Ш(E)[4]$ provides the same information as performing an 8-descent on C. These elements may be represented by a 4-covering $C \to E$ and a 2-covering $D \to E$ respectively.

In Magma the pairing between 2-coverings is implemented over $\mathbb{Q}$ number fields, and rational function fields $F(t)$ for F finite of odd characteristic. The pairing between a 2-covering and a 4-covering is implemented over $\mathbb{Q}$. A new, very efficient implementation of pairing on 2-coverings over $\mathbb{Q}$ was released in Magma V2.15.

The algorithms are due to Steve Donnelly and will be described in a forthcoming paper, a draft of which is available on request. For the pairing between 2-coverings, the only nontrivial computation is to solve a conic over the base field of E, so over $\mathbb{Q}$ the pairing is easy to compute. For the pairing between 2-

and 4-coverings, the key step is to solve a conic defined over a degree 4 field; this is also the case for performing 8-descent on the 4-covering, however the advantage here is that there is considerable freedom to choose the field to have small discriminant. Consequently it is more efficient to use the pairing than to apply `EightDescent`.

To have information about the computation printed while it is running, one may use `SetVerbose("CasselsTate",n);` with $n = 1$ (for fairly concise information) or $n = 2$.

`CasselsTatePairing(C, D): CrvHyp, CrvHyp -> RngIntElt`

`CasselsTate: Verbose`	`Default: 2`

This evaluates the Cassels-Tate pairing on 2-coverings of an elliptic curve over $\mathbb{Q}$, a number field, or a function field $F(t)$ where F is a finite field of odd characteristic. The given curves C and D must be hyperelliptic curves of the form $y^2 = q(x)$ where $q(x)$ has degree 4, and they must admit 2-covering maps to the same elliptic curve. In addition, they must both be locally solvable over all completions of their base field (otherwise the pairing is not defined).

Typically the input curves C and D would be obtained using `TwoDescent(E)`.

The pairing takes values in $\mathbb{Z}/2\mathbb{Z}$ (returned as elements of $\mathbb{Z}$).

`CasselsTatePairing(C, D): Crv, CrvHyp -> RngIntElt`

`CasselsTate: Verbose`	`Default: 2`

This evaluates the Cassels-Tate pairing between a 4-covering C and a 2-covering D of the same elliptic curve over $\mathbb{Q}$. The arguments must be curves over $\mathbb{Q}$, with C an intersection of two quadrics in $\mathbb{P}^3$ (for instance, a curve obtained from `FourDescent`), and D a hyperelliptic curve of the form $y^2 = q(x)$. In addition, they must both be locally solvable over all completions of $\mathbb{Q}$ (otherwise the pairing is not defined).

The pairing takes values in $\mathbb{Z}/2\mathbb{Z}$ (returned as elements of $\mathbb{Z}$).

Example: Cassels-Tate Pairing with Nontrivial Sha `(ex-a17545)`

We consider the first elliptic curve with trivial 2-torsion and nontrivial Tate–Shafarevich group.

```
> E := EllipticCurve("571a1"); E;
Elliptic Curve defined by y^2 + y = x^3 - x^2 - 929*x - 10595 over Rational Field
> #TorsionSubgroup(E);
1
> time covers := TwoDescent(E); covers;
Time: 0.270
[
    Hyperelliptic Curve defined by y^2 = -11*x^4 - 68*x^3 - 52*x^2 + 164*x - 64
    over Rational Field,
    Hyperelliptic Curve defined by y^2 = -19*x^4 + 112*x^3 - 142*x^2 - 68*x - 7
    over Rational Field,
    Hyperelliptic Curve defined by y^2 = -4*x^4 + 60*x^3 - 232*x^2 + 52*x - 3
    over Rational Field
]
> time CasselsTatePairing(covers[1], covers[2]);
1
Time: 0.130
```

This proves that these two coverings both represent nontrivial elements in the Tate–Shafarevich group; in fact our computations show that the 2-primary part of $Ш(E)$ is precisely $\mathbb{Z}/2 \times \mathbb{Z}/2$. We could have reached the same conclusion using 4-descent:

```
> time FourDescent(covers[1]);
[]
Time: 0.460
```

133.5.6 Four-Descent

This section describes an implementation of the algorithm for higher descent given in a 1996 paper [MSS96] by Merriman, Siksek, and Smart, for elliptic curves over $\mathbb{Q}$. Four-descent is performed as a higher descent on a given two-cover, that is a hyperelliptic curve defined by a polynomial of degree four, as returned by `TwoDescent`. (The trivial case where the quartic has a rational root is excluded: such a quartic can be transformed to an elliptic curve by a linear transformation, and four-descent then becomes standard two-descent.)

Note: for curves over number fields, the same algorithm is partly implemented by the function `TwoCoverDescent`. However in that case we lack techniques for reducing the coverings, so this can not be used to find points but only to bound the rank.

A *four-covering* F is a pair of symmetric 4×4 matrices, defining an intersection of two quadrics in P^3. Associated to F is an elliptic curve E; there is a rational map from F to E of degree 16. The four-descent process takes a two-covering curve C (something of the shape $y^2 = f(x)$ with f quartic, and possessing points over $\mathbb{Q}_p$ for all p), and returns a set of four-coverings that arise from C.

In particular, if C represents an element of order two in the Tate–Shafarevich group of E, then the four-descent process will return the empty set. If C represents an element of the Mordell–Weil group, at least one of the four-coverings arising from C will have a rational point — all of them will do so if the Tate–Shafarevich group of E is trivial — and once found this point can be lifted to a point on E.

`FourDescent(C : parameters): CrvHyp -> [Crv]`

`FourDescent(f : parameters): RngUPolElt -> [Crv]`

`FourDescent(S : parameters): SeqEnum -> [Crv]`

`FourDescent(C : parameters): ModelG1 -> [Crv]`

```
RemoveTorsion       : BoolElt                    Default: false
IgnoreRealSolubility: BoolElt                    Default: false
RemoveGensEC        : { PtEll}                   Default: { }
RemoveGensHC        : { PtHyp}                   Default: { }
FourDescent         : Verbose                    Default: 3
LocalQuartic        : Verbose                    Default: 2
MinimiseFD          : Verbose                    Default: 2
QISearch            : Verbose                    Default: 1
ReduceFD            : Verbose                    Default: 2
QuotientFD          : Verbose                    Default: 2
```

Performs a four-descent on the curve $y^2 = f(x)$, where f is a quartic. Returns a set of four-coverings of size 2^{s-1}, where s is the Selmer 2-rank of the curve. If the verbose level of the main function is set to 3, then the auxiliary verbose levels are all set to at least 1. If `RemoveTorsion` is `true`, the generators of the torsion subgroup are factored out from the set. The optional argument `RemoveGensHC` performs a quotient by the images of given points that are on the input quartic to `FourDescent`. The optional argument `RemoveGensEC` forms a quotient by the images of given points; these can be on any elliptic curve that is isomorphic to the `AssociatedEllipticCurve` of the quartic (though all given points must be on the same elliptic curve). These options can be used together. It can be non-trivial to remove torsion and generators, as points on the elliptic curve need not pull-back to the given $y^2 = f(x)$ curve. The algorithm used exploits various primes of good reduction, and attempts to determine whether the images under the μ_p are the same. This in turn can be tricky, due to the Cassels kernel, and even more so when there are extra automorphisms (that is, when $j = 0, 1728$ over $\mathbf{F}_p$).

Example: Four-Descent Proving Rank 0 (ex-acf772)

This example shows that a well-known curve has rank 0 and that the 2-torsion subgroup of its Tate–Shafarevich group is isomorphic to $(\mathbb{Z}/2\mathbb{Z})^2$.

```
> D := CremonaDatabase();
> E := EllipticCurve(D, 571, 1, 1);
> time td := TwoDescent(E);
Time: 2.500
> #td;
3
```

There are three 2-covers, so the 2-Selmer group has order four (since `TwoDescent` elides the trivial element).

```
> time [ FourDescent(t) : t in td ];
[
    [],
    [],
    []
]
Time: 3.290
```

So none of the two-covers have four-covers lying over them; hence they all represent elements of IIIwidth9.5pt depth0pt height0.3pt, and the Mordell–Weil rank must be zero.

AssociatedEllipticCurve(qi): Crv -> CrvEll, Map

AssociatedHyperellipticCurve(qi): Crv -> CrvHyp, Map

```
E: CrvEll                    Default:
```

Given an intersection of quadrics qi, return the associated elliptic and hyperelliptic curves, respectively, together with maps to them.

If an elliptic curve is given as E, this must be isomorphic to the Jacobian of the curve qi, and then the map returned will be a map to the given E.

QuadricIntersection(F): [AlgMatElt] -> Crv

QuadricIntersection(P, F): Prj, [AlgMatElt] -> Crv

Given a pair of symmetric 4×4 matrices F, this function returns the associated quadric intersection in $P = P^3$.

QuadricIntersection(E): CrvEll -> Crv, MapIsoSch

QuadricIntersection(C): CrvHyp -> Crv, MapIsoSch

Given an elliptic curve E or a hyperelliptic curve C, write it as an intersection of quadrics. The inverse map for the hyperelliptic curve has problems due to difficulties with weighted projective space.

IsQuadricIntersection(C): Crv -> BoolElt, [AlgMatElt]

Given a curve C, determines if C is in P^3 and has two defining equations, both of which involve only quadrics. In the case where C is a quadric intersection, the associated pair of matrices are also returned.

PointsQI(C, B : parameters): Crv, RngIntElt -> [Pt]

```
OnlyOne    : BoolElt                 Default: false
ExactBound: BoolElt                  Default: false
QISearch   : Verbose                 Default: 1
```

Given a quadric intersection C, this function searches, by a reasonably efficient method due to Elkies [Elk], for a point on C of na"10ve height up to B; the asymptotic running time is $O(B^{2/3})$.

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 coordinates 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.

`TwoCoverPullback(H, pt): CrvHyp[FldRat], PtEll[FldRat] -> [PtHyp]`

`TwoCoverPullback(f, pt): RngUPolElt[FldRat], PtEll[FldRat] -> [PtHyp]`

Given a two-covering of a rational elliptic curve (as either a hyperelliptic curve or a quartic) and a point on the elliptic curve, compute the pre-images on the two-covering. This is faster than using the generic machinery.

`FourCoverPullback(C, pt): Crv[FldRat], PtEll[FldRat] -> [Pt]`

`FourCoverPullback(C, pt): Crv[FldRat], PtHyp[FldRat] -> [Pt]`

Given a four-covering of a rational elliptic curve as an intersection of quadrics and a point either on the associated elliptic curve or the associated hyperelliptic curve, this function computes the pre-images on the covering. This is faster than using the generic machinery.

Example: Four-Descent and Mapping Points (ex-532591)

This example exhibits a four-descent computation, and manipulation of points once they have been found, by mapping from the curve to its two- and four-covers.

```
> P<x> := PolynomialRing(Integers());
> E := EllipticCurve([0, -1, 0, 203, -93]);
> f := P!Reverse([-7, 12, 20, -120, 172]);
> f;
-7*x^4 + 12*x^3 + 20*x^2 - 120*x + 172
```

The quartic given was obtained with `mwrank`; the two-descent routine could have been used instead, although it provides a different (but equivalent) quartic.

```
> time S := FourDescent(f);
Time: 4.280
> #S;
1
```

The single cover indicates that the curve E has Selmer rank 1, though this was already known from the calculation that constructed f.

```
> _,m := AssociatedEllipticCurve(S[1] : E:=E );
> pts := PointsQI(S[1], 10^4);
> pts;
[ (-5/3 : 13/3 : -34/3 : 1) ]
```

We now map this point back to E.

```
> m(pts[1]);
(2346223045599488598/1033322524668523441 :
20326609223460937753264735467/10503978993582666056926724 89 : 1)
> Height($1);
44.196795967396224772619833 70
```

133.5.7 Eight-Descent

One may perform 8-descent (ie a further 2-descent) on curves of the kind produced by a 4-descent on an elliptic curve E over $\mathbb{Q}$. These are nonsingular intersections of two quadrics in $\mathbb{P}^3$ that are locally soluble. The 8-descent determines whether such a curve has any 2-coverings (in the sense of 2-descent) that are locally soluble everywhere.

The routine can therefore be used to prove, in many cases, that a given 4-covering of E is in fact an element of order 4 in the Tate–Shafarevich group of E. It can also be used to find 8-coverings of E, and to verify these are elements of the 8-Selmer group.

The 8-coverings are given as genus one normal curves of degree 8 in $\mathbb{P}^8$. They are minimised and reduced, so are useful in searching for points on E.

The algorithm and implementation (from Magma 2.17) are due to Tom Fisher; this implementation partly incorporates and partly replaces an earlier one by Sebastian Stamminger.

EightDescent(C : parameters): Crv -> [Crv], [MapSch]

```
BadPrimesHypothesis       : BoolElt                          Default: false
DontTestLocalSolvabilityAt: { RngIntElt}                     Default: { }
StopWhenFoundPoint        : BoolElt                          Default: false
EightDescent              : Verbose                          Default: 4
```

For a curve C obtained from `FourDescent` on some elliptic curve E over $\mathbb{Q}$, this performs a further 2-descent on C. It returns a sequence of curves D, together with a sequence containing maps $D \to C$ from each of these curves to C. The curves returned are precisely the 8-descendants of E that lie above C and are locally soluble at all places.

When the optional argument `StopWhenFoundPoint` is set to `true`, the computation will stop if it happens to find a rational point on C, and immediately return the point instead of continuing to computing the 8-coverings.

In some cases local solubility testing (at large primes which may arise due to choices made in the algorithm) can be time-consuming. Testing at specified primes may be skipped by setting the optional argument `DontTestLocalSolvabilityAt` to the desired set of prime integers, or by setting `BadPrimesHypothesis` to be true. In that case, certain primes are omitted from the set of bad primes (namely those primes at which the intersection of C with an auxiliary third quadric has bad reduction, and which are not bad primes for any other reason).

The algorithm involves class group and unit computations in a number field of degree 4 (and sometimes also 8). It is recommended to set the bounds to be used in all such computations before calling EightDescent, via `SetClassGroupBounds`.

Verbose output: For a readable summary of the computation, set the verbose level to 1 by entering `SetVerbose("EightDescent",1)`. For full information about all the time-consuming steps in the process, set the verbose level to 3. For additional information about solving the conic, set the verbose level for `"Conic"` to 1.

133.5.8 Three-Descent and Five-Descent

Three descent is implemented for elliptic curves over the rationals. This involves computing the 3-Selmer group of the curve, and then representing the elements as plane cubics. The functions related to 5-descent take a very similar form to those for 3-descent, although they are not listed in the handbook.

There is also a separate implementation of "descent by three isogenies", for elliptic curves over the rationals which admit a $\mathbb{Q}$-rational isogeny of degree 3.

The main application of full three-descent is in studying elements of order 3 in Tate–Shafarevich groups. For the problem of determining the Mordell–Weil group, three-descent has no advantage over four descent except in special cases: the cost is greater (three-descent requires computing the class group and S-units in a degree 8 number field, compared with degree 4 for four-descent), and the reward is smaller (a point on a 3-coverings has height 1/6 as large as its image on the elliptic curve, while with four-descent the ratio is 1/8). However three-descent can be useful when there are elements of order 4 in the Tate–Shafarevich group, which four-descent cannot deal with.

On the other hand, for a curve with a $\mathbb{Q}$-rational isogeny of degree 3, descent by 3-isogenies is likely to be the most efficient way to bound the Mordell–Weil rank, because it only requires class group and S-unit computations in quadratic fields.

There are two steps to the 3-descent process: firstly, computing the 3-Selmer group as a subgroup of $H^1(\mathbb{Q}, E[3])$ (explicitly, as a subgroup of $A^\times/A^{\times 3}$ for a suitable algebra A), and secondly, expressing the elements as genus one curves with covering maps to E. The elements of the 3-Selmer group are given as plane cubics, and the process of obtaining these cubics is far from trivial. (Note that in general, an element of $H^1(\mathbb{Q}, E[3])$ can only be given as a curve of degree 9 rather than degree 3). The main commands are `ThreeSelmerGroup(E)`, which performs the first step, and `ThreeDescent(E)`, which performs both steps together, while `ThreeDescentCubic` performs the second step for a given element of 3-Selmer group.

The algorithm for the first step is presented in [SS04], while the theory and algorithms for the second step are developed in the forthcoming series of papers [CFO+08], [CFO+09], [CFO+]. The bulk of the code was written by Michael Stoll and Tom Fisher (however, responsibility for the final version rests with Steve Donnelly).

The following verbose flags provide information about various stages of the three descent process: `Selmer`, `ThreeDescent`, `CSAMaximalOrder`, `Minimise` and `Reduce`. For instance, to see what `ThreeSelmerGroup` is doing while it is running, first enter `SetVerbose("Selmer",2);`. The verbose levels range from 0 to 3.

`ThreeDescent(E : parameters): CrvEll -> [ Crv ], List`

```
Method      : MonStgElt                    Default: "HessePencil"
Selmer      : Verbose                      Default: 3
ThreeDescent: Verbose                      Default: 3
```

Given an elliptic curve over the rationals, this function returns the elements of the 3-Selmer group as projective plane cubic curves C, together with covering maps $C \to E$. Two objects are returned: a sequence containing one curve for each inverse pair of nontrivial elements in the 3-Selmer group, and a corresponding list of maps from these curves to E. (Note that a pair of inverse elements in the 3-Selmer group both correspond to the same cubic curve C, and their covering maps $C \to E$ differ by composition with the negation map $E \to E$.)

The function is equivalent to calling `ThreeSelmerGroup(E)`, and then calling `ThreeDescentCubic` on each of the Selmer elements.

For more information see the description of `ThreeDescentCubic` below.

Example: Selmer's Famous Cubic Curve (ex-e7c713)

Here is Selmer's famous example $3x^3 + 4y^3 + 5z^3 = 0$, which is an element of order 3 in the Tate–Shafarevich group of its Jacobian, the elliptic curve $x^3 + y^3 + 60z^3 = 0$.

```
> Pr2<x,y,z> := ProjectiveSpace(Rationals(),2);
> J := x^3 + y^3 + 60*z^3;
> E := MinimalModel(EllipticCurve(Curve(Pr2,J)));
> cubics, mapstoE := ThreeDescent(E);
> cubics;
[
    Curve over Rational Field defined by 2*x^3 + 30*y^3 - z^3,
    Curve over Rational Field defined by x^3 + 5*y^3 - 12*z^3,
    Curve over Rational Field defined by 6*x^3 + 5*y^3 - 2*z^3,
    Curve over Rational Field defined by 3*x^3 + 5*y^3 - 4*z^3
]
```

The 3-Selmer group of E is isomorphic to $\mathbb{Z}/3\mathbb{Z} \oplus \mathbb{Z}/3\mathbb{Z}$, and one element for each (nontrivial) inverse pair is returned. (Note: $3x^3 + 4y^3 + 5z^3$ will not necessarily appear; due to random choices in the program, an equivalent model may appear instead.)

The covering maps from these curves to E have degree 9, and are given by forms of degree 9 in x, y, z. For example, the map from the first curve $2x^3 - 3y^3 + 10z^3$ to E is given by:

```
> DefiningEquations(mapstoE[1]);
[
    -1377495072000*x*y^7*z + 4591650240000*x*y^4*z^4 - 15305500800000*x*y*z^7,
    24794911296000*y^9 - 123974556480000*y^6*z^3 - 413248521600000*y^3*z^6 +
    918330048000000*z^9,
    -4251528000*x^3*y^3*z^3
]
```

Since E has trivial Mordell–Weil group, we should not find any rational points on these curves! (In fact they are all nontrivial in the Tate–Shafarevich group.) Here we search for rational points on the first cubic, up to height roughly 10^4:

```
> time PointSearch( cubics[1], 10^4);
[]
Time: 0.490
```

An extended example, concerning "visible" 3-torsion in a Tate–Shafarevich group, can be found at the end of Chapter *Models of Genus One Curves*.

ThreeSelmerGroup(E : parameters): CrvEll -> GrpAb, Map

ThreeTorsPts:	Tup	Default:
Selmer :	Verbose	Default: 3

Given an elliptic curve E over the rationals, this function returns the 3-Selmer group as an abelian group, together with a map to the natural affine algebra.

When the optional parameter `ThreeTorsPts` is given, this determines the number fields used in the computation, and also the affine algebra.

Typically much of the computation time is spent on class group and unit group computations. In many instances it will not be feasible to do these calculations rigorously – in this situation the user must request a heuristic class group computation. The recommended way to do this is (in advance): `SetClassGroupBounds("GRH")`. See Section Setting the Class Group Bounds for details. An alternative way is to precompute class groups (and possibly units) for the fields of the `ThreeTorsPts`.

The algorithm is described in [SS04].

ThreeDescentCubic(E, α : parameters): CrvEll, Tup -> Crv, MapSch

ThreeTorsPts:	Tup	Default:
Method :	MonStgElt	Default: "HessePencil"
ThreeDescent:	Verbose	Default: 3

Given an elliptic curve E over the rationals, and an element α in the 3-Selmer group of E, the function returns a projective plane cubic curve C, together with a map of schemes $C \to E$. The cubic is a principal homogeneous space for E, and the covering map $C \to E$ represents the same Selmer element as either α or $1/\alpha$ (and α can be recovered, up to inverse, by calling `ThreeSelmerElement` of the cubic).

The 3-Selmer element α is given as an element of the algebra associated to the 3-Selmer group (the algebra is the codomain of the map returned by `ThreeSelmerGroup`, as in `S3, S3toA := ThreeSelmerGroup(E)`). In this situation, where α is `S3toA(s)` for some `s`, there is no need to specify the optional parameter `ThreeTorsPts`.

The algorithm comes from the series of papers cited in the introduction. There are three alternative ways to perform the final step of computing a ternary cubic. All three ways are implemented, and the optional parameter `Method` may be "HessePencil" (default), "FlexAlgebra", or "SegreEmbedding". However, the choice of `Method` is not expected to make a big difference to the running time.

The optional parameter `ThreeTorsPts` is a tuple containing one representative from each Galois orbit of $E[3] \setminus O$. Its purpose is to fix an embedding of the 3-Selmer group in $A^\times/A^{\times 3}$ (otherwise there is ambiguity when the fields involved have nontrivial automorphisms, or occur more than once). When `ThreeTorsPts` is not specified, `ThreeTorsionPoints(E)` is called (its value is stored internally and used throughout the Magma session).

`ThreeIsogenyDescent(E : parameters): CrvEll -> [ Crv ], List, [ Crv ], List, MapSch`

```
Isog        : MapSch                    Default:
Selmer      : Verbose                   Default: 3
ThreeDescent: Verbose                   Default: 3
```

Given an elliptic curve E over $\mathbb{Q}$ that admits a $\mathbb{Q}$-rational isogeny $E \to E_1$ of degree 3, the function performs "descent by 3-isogenies" on E. This involves computing the Selmer groups attached to the isogeny and its dual, and representing the elements of both Selmer groups as plane cubics, with covering maps of degree 3 to E_1 or E respectively. One cubic is given for each nontrivial pair of inverse Selmer elements, and one covering map is given for each cubic (the other covering map, which can be obtained by composing with the negation map on the elliptic curve, would correspond to the inverse Selmer element.)

There are five returned values, in the following order: a list of curves for $Sel(E \to E_1)$, a corresponding list of covering maps to E_1, a list of curves for the dual isogeny Selmer group $Sel(E_1 \to E)$, a corresponding list of covering maps to E, and finally the isogeny $E \to E_1$.

This function works simply by calling `ThreeIsogenySelmerGroups(E)`, and then calling `ThreeIsogenyDescentCubic` for each Selmer element.

The isogeny $E \to E_1$ may be passed in as `Isog`. If `Isog` is not specified, and E admits more than one such isogeny, then this is chosen at random.

`ThreeIsogenySelmerGroups(E : parameters): CrvEll -> GrpAb, Map, GrpAb, Map, MapSch`

```
Isog  : MapSch                      Default:
Selmer: Verbose                     Default: 3
```

Given an elliptic curve E over the rationals that admits a $\mathbb{Q}$-rational isogeny $E \to E_1$ of degree 3, the function computes the Selmer groups associated to the isogeny, and to its dual isogeny $E_1 \to E$. The Selmer groups are returned as abstract groups, together with maps to the relevant algebra. There are five returned values, in the following order: the group, and the map, for $E \to E_1$, the group, and the map, for $E_1 \to E$, and finally the isogeny $E \to E_1$.

A bound for the rank of $E(\mathbb{Q})$ can be deduced, by taking the sum of the ranks of the Selmer groups for the two isogenies, and subtracting 1 if the kernel of one of the isogenies consists of rational points.

The isogeny $E \to E_1$ may be passed in as `Isog`. If `Isog` is not specified, and E admits more than one such isogeny, then this is chosen at random.

The algebra in which the Selmer group of a particular isogeny is exhibited is the etale algebra corresponding to the nontrivial points in the kernel of the dual isogeny; it is either a quadratic field, or a Cartesian product of two copies of $\mathbb{Q}$.

`ThreeIsogenyDescentCubic(`φ`,` α`): MapSch, Any -> Crv, MapSch`

```
ThreeDescent: Verbose                   Default: 3
```

Given an isogeny ϕ of degree 3 between elliptic curves over $\mathbb{Q}$, and any element α of $H^1(\mathbb{Q}, E[\phi])$, this function returns a plane cubic curve C representing α, together with a covering map $C \to E$ of degree 3.

The element α is given as an element in the algebra A associated to the Selmer group of ϕ; the algebra can be obtained from `ThreeIsogenySelmerGroups`. $H^1(\mathbb{Q}, E[\phi])$ is represented as the subgroup of $A^\times/(A^\times)^3$ consisting of elements whose norm is a cube.

ThreeDescentByIsogeny(E): CrvEll -> [Crv], [Map]

Selmer: Verbose	Default: 3

This performs a full 3-descent, returning models for the nontrivial inverse pairs in the 3-Selmer group and maps to E making them into 3-coverings. The only difference with **ThreeDescent** is that the computation is by first and second 3-isogeny descents. This restricts the number fields used to cubic extensions, rather than the sextic fields which may be needed for the generic **ThreeDescent** routine. The advantage is noticeable for larger examples.

Example: Three-Descent by Isogeny (ex-a782ba)

```
> E := EllipticCurve([0,0,0,0,131241]);
> G,m := TorsionSubgroup(E);
> #G;
1
> Rank(E);
2
> S1,_,S2 := ThreeIsogenyDescent(E);
> #S1,#S2;
0 4
```

This shows that there is nontrivial 3-torsion in Ш on some isogenous curve. Without computing the full 3-Selmer group of E it is impossible to tell if there is any nontrivial 3-torsion in $Ш(E/\mathbb{Q})$.

```
> time Sel3 := ThreeDescentByIsogeny(E);
Time: 5.280
```

This shows that $Ш(E/\mathbb{Q})[3] = 0$ (since we know the rank is 2). Alternatively one could get the same by calling **ThreeDescent**. This requires computing the class group and unit group information in the sextic field shown below.

```
> E3reps := ThreeTorsionPoints(E);
> Parent(E3reps[2,1]);
Number Field with defining polynomial x^6 + 14174028 over the Rational Field
> SetClassGroupBounds("GRH");
> time Sel3 := ThreeDescent(E);
Time: 14.700
```

Jacobian(C): RngMPolElt -> CrvEll

Given the equation of a nonsingular projective plane cubic curve C over the rationals, this function returns the Jacobian of C as an elliptic curve.

ThreeSelmerElement(E, C): CrvEll, RngMPolElt -> Tup

ThreeSelmerElement(E, C): CrvEll, Crv -> Tup

ThreeSelmerElement(C): RngMPolElt -> Tup

ThreeSelmerElement(C): Crv -> Tup

Given an elliptic curve E over the rationals, and a plane cubic C with the same invariants as E (for instance, with E equal to **Jacobian(C)**) the function returns an element α in the algebra A associated to the 3-Selmer group of E. This α represents the same element of $H^1(\mathbb{Q}, E[3])$ that is represented by a covering $C \to E$ that takes the flex points of C to O. Note that α is only determined up to inverse in $H^1(\mathbb{Q}, E[3])$.

In particular, if we have computed **S3, S3toA := ThreeSelmerGroup(E)**, and if C is an everywhere locally soluble covering corresponding to the element **s** in **S3**, then α equals either **S3toA(s)** or **S3toA(-s)** in $A^\times/(A^\times)^3$.

AddCubics(cubic1, cubic2 : parameters): RngMPolElt, RngMPolElt -> RngMPolElt

```
E          : CrvEll                    Default:
ReturnBoth: BoolElt                    Default: false
```

Given equations of two plane cubics over the rationals with the same invariants (in other words, they are homogeneous spaces for the same elliptic curve E, which is their Jacobian, over the rationals), the function computes the sum of the corresponding elements of $H^1(\mathbb{Q}, E[3])$. The sum is returned as another plane cubic, if possible (and otherwise an error results).

An element of $H^1(\mathbb{Q}, E[3])$ may be expressed as a plane cubic when it has index 3, equivalently when the so-called "obstruction" is trivial. This is always the case for elements that are everywhere locally soluble. In particular, the function will always succeed when the two given cubics are everywhere locally soluble (in other words, when they belong to the 3-Selmer group). The computation is done by converting the cubics to elements of $H^1(\mathbb{Q}, E[3])$ as given by `ThreeSelmerElement`, adding the cocycles, and then converting the result to a cubic.

Note that a cubic only determines an element of $H^1(\mathbb{Q}, E[3])$ up to taking inverse, so the sum is not well defined; if the function is called with `ReturnBoth := true`, it returns both of the possible cubics.

`ThreeTorsionType(E): CrvEll -> MonStgElt`

For an elliptic curve E over the rationals, this function classifies the Galois action on $E[3]$. The possibilities are "Generic", "2Sylow", "Dihedral", "Generic3Isogeny", "Z/3Z-nonsplit", "mu3-nonsplit", "Diagonal", and "mu3+Z/3Z".

`ThreeTorsionPoints(E : parameters): CrvEll -> Tup`

```
OptimisedRep: BoolElt                  Default: true
```

For an elliptic curve E over the rationals, this function returns a tuple containing one representative point from each set of Galois conjugates in $E[3] \setminus O$.

Each point belongs to a point set $E(L)$, where L is the field generated by the coordinates of that point.

If `OptimisedRep` is set to `false`, then optimised representations of the fields will not be computed, in general.

`ThreeTorsionMatrices(E, C): CrvEll, RngMPolElt -> Tup`

Given an elliptic curve E over the rationals, and a plane cubic with the same invariants as E (in other words, a principal homogeneous space for E of index 3), the function returns a tuple of matrices. The matrices M_i correspond to the points T_i in `ThreeTorsionPoints(E)`, and have the corresponding base fields. Each matrix describes the action-by-translation on C of the corresponding point: the action of P_i on C is the restriction to C of the automorphism of the ambient projective space $\mathbb{P}^2$ given by the image of M_i in PGL_3.

Note that only the images in PGL_3 of the matrices are well-determined.

133.5.9 Six and Twelve Descent

If a 3-descent has been performed, the results can be used in conjunction with a 2-descent or a 4-descent to obtain coverings of degree 6 or 12 respectively. These "combined" coverings can be useful for finding Mordell–Weil generators of large height on the underlying elliptic curve.

The coverings are given as genus one normal curves (of degree 6 in $\mathbb{P}^5$, and of degree 12 in $\mathbb{P}^1 1$, respectively).

The algorithms are described in [Fis08b].

`SixDescent(C2, C3): CrvHyp, Crv -> Crv, MapSch`

`SixDescent(model2, model3): ModelG1, ModelG1 -> Crv, MapSch`

Given a 2-covering and a 3-covering of an elliptic curve E (either as curves or as genus one models), this returns the 6-covering that represents their sum in the 6-Selmer group. The covering map $C6 \to C3$ is also returned.

`TwelveDescent(C3, C4): Crv, Crv -> SeqEnum, MapSch`

`TwelveDescent(model3, model4): ModelG1, ModelG1 -> SeqEnum, MapSch`

Given a 3-covering and a 4-covering of an elliptic curve E (either as curves or as genus one models), this returns the two 12-coverings that represent their sum and difference in the 12-Selmer group. The covering maps to $C12 \to C4$ are also returned.

133.5.10 Nine-Descent

A nine-descent is performed on an everywhere locally solvable plane cubic curve. The cubic represents a class in the 3-Selmer group of its Jacobian (up to sign). A nine-descent computes the fibre above this class under the map from the 9-Selmer group to the 3-Selmer group induced by multiplication by 3. This fibre is the set of everywhere locally solvable 3-coverings of the cubic. These coverings are given as intersections of 27 quadrics in $\mathbb{P}^8$. As with four-descents the terminology "nine-descent" is slightly incorrect, as Magma performs a second 3-descent on a specific 3-covering and does not try to combine such information from all 3-coverings to form the 9-Selmer group.

The algorithm was developed in the PhD thesis of Brendan Creutz [Cre10]. Typically most of the computation time is spent on class group and unit group computations in the constituent fields of the degree 9 etale algebra associated to the flex points on the cubic. The primary use is to obtain information on the 3-primary part of the Shafarevich–Tate group. It is only in very rare circumstances that a nine descent is required (in addition to the other descent machinery) to determine the Mordell–Weil rank (e.g. if there were elements of order 24 in Ш).

`NineDescent(C : parameters): Crv -> SeqEnum, List`

```
ExtraReduction: RngIntElt                    Default: 10
NineDescent   : Verbose                      Default: 3
```

Given a plane cubic curve C over $\mathbb{Q}$ this returns a sequence of curves in $\mathbb{P}^8$ defined by 27 quadrics and a list of degree 9 maps making these curves into 3-coverings of C. This is the 3-Selmer set of C (i.e. the set of everywhere locally solvable 3-coverings of C). If there are no coverings, then $C(\mathbb{Q}) = \emptyset$ and the class of C in the Shafarevich-Tate group of its Jacobian is not divisible by 3. Otherwise the coverings returned are lifts of C to the 9-Selmer group.

The coverings returned are minimised and reduced in an ad hoc fashion. If `ExtraReduction` is set to a larger value, more time will be spent reducing possibly resulting in smaller models. The current implementation requires that Galois act transitively on the flex points. If this is not the case, then one can use `pIsogenyDescent` instead.

`NineSelmerSet(C): Crv -> RngIntElt`

```
NineDescent: Verbose                         Default: 3
```

Computes the 3-rank of the 3-Selmer set of plane cubic curve C defined over $\mathbb{Q}$. The value -1 is returned when the 3-Selmer set is empty. In this case $C(\mathbb{Q}) = \emptyset$ and the class of C in the Shafarevich-Tate group of its Jacobian is not divisible by 3. If the 3-Selmer set is nonempty, its 3-rank is the same as that of the 3-Selmer group of the Jacobian. In this case the class of C in the Shafarevich-Tate group of the Jacobian is divisible by 3.

133.5.11 Higher 2-power Isogeny Descents

Given an elliptic curve $E/\mathbb{Q}$ with a rational 2-torsion point $T \in E(\mathbb{Q})[2]$, there is an degree-2 isogeny $\phi : E \to E'$ with kernel $\{0, T\}$. Computing the Selmer groups attached to ϕ and the dual isogeny $\widehat{\phi} : E' \to E$ gives an upper bound for the rank of $E(\mathbb{Q})$. In many cases this upper bound may be improved by performing higher descents. These higher descents can also help find generators of $E(\mathbb{Q})$ of large height. See for example the paper of Bremner and Cassels [BC84] on the elliptic curves $Y^2 = X(X^2 + p)$ where p is a prime with $p \equiv 5 \pmod 8$.

The function below applies these methods, together with use of the Cassels-Tate pairing and other refinements. The algorithm is fully described in [Fisb]. No class group and unit calculations are required; instead the global parts of the calculation involve solving quadratic forms of ranks 3 and 4 over $\mathbb{Q}$.

The same information is obtained by calling this for (E, T) and for the isogenous (E', T'). In fact, the output and the verbose printing will be the same, but in the opposite order, for the two computations.

```
TwoPowerIsogenyDescentRankBound(E, T : parameters): CrvEll[FldRat], PtEll[FldRat] ->
                                RngIntElt, SeqEnum, SeqEnum
TwoPowerIsogenyDescentRankBound(E : parameters): CrvEll[FldRat] -> RngIntElt, SeqEnum,
                                SeqEnum
```

```
Cutoff             : RngIntElt                    Default: 2
MaxSteps           : RngIntElt                    Default: 5
ReturnFourCoverings: BoolElt                      Default: false
cbrank             : Verbose                      Default: 3
```

This calculates an upper bound on the Mordell–Weil rank of the elliptic curve $E/\mathbb{Q}$. The second argument T must be a $\mathbb{Q}$-rational 2-torsion point on E, specifying which 2-isogeny is to be used. When there is only one 2-torsion point, the second argument can be omitted.

Let $\phi : E \to E'$ be the isogeny with kernel generated by T, and $\widehat{\phi} : E' \to E$ the dual isogeny, with kernel generated by T'. Let ϕ_m and $\widehat{\phi}_m$ be the isogenies of degree 2^m obtained by composing ϕ and $\widehat{\phi}$ a total of m times. Then the rank bound after m steps is that obtained by computing the images of the Selmer groups attached to ϕ_m and $\widehat{\phi}_m$ in the Selmer groups attached to ϕ and $\widehat{\phi}$. The dimensions of these images are also returned. At each step, the rank bound improves by an even integer (or stays the same).

A summary table is printed after each step, if the verbose flag `cbrank` is set to 1, which displays the rank bounds obtained so far.

If the current rank bound is strictly less than the `Cutoff`, then the program returns immediately without performing any further descents. By default the `Cutoff` is 2, so the program ends if a bound of 0 or 1 is obtained (since no stronger bound can then be obtained).

The maximum number of descent steps carried out can be controlled by specifying `MaxSteps` (5 by default). For example, if `MaxSteps` is 1, then the program carries out a descent by 2-isogeny. If `MaxSteps` is 2, then the upper bound on the rank is at least as good as that obtained by carrying out a 2-descent on either of the curves E or E'. In cases where E or E' has full rational 2-torsion, a 6th step, corresponding to 8-descent, is also possible. Note that this is not done by default, but only when `MaxSteps` is set to 6.

If `ReturnFourCoverings` is `true`, then the function also returns a list of quadric intersections (i.e. genus one models of degree 4). These may be used in the same way as those returned by `FourDescent`, to search for rational points of large height on E. A total of 2^d quadric intersections are returned, where d is the dimension of the image of the 4-Selmer group of E in the 2-Selmer group of E. To obtain from these a complete list of quadric intersections (i.e. one for each element of the 4-Selmer group) the functions `TwoDescent` and `AddCovers` should be used. If `ReturnFourCoverings` is `true`, the descent steps done are the minimum necessary to get this set of 4-coverings (that is, 4 steps for E and 3 steps for E'), and the values of `Cutoff` and `MaxSteps` are ignored.

133.5.12 p-Isogeny Descent

Given an isogeny $\phi : E_1 \to E_2$ of elliptic curves, one may define the ϕ-Selmer group of E_2 to be the set of everywhere locally solvable ϕ-coverings of E_2. Given a genus one curve C with Jacobian E_2, one may define its ϕ-Selmer set to be the set of everywhere locally solvable ϕ-coverings of C. A ϕ-isogeny descent computes the ϕ-Selmer group (or ϕ-Selmer set).

We denote the dual isogeny by $\phi^\vee$. In the case of elliptic curves, performing ϕ- and $\phi^\vee$-descents can be used to bound the Mordell–Weil rank and get information on Ш. For general genus one curves descent by isogeny can be used to rule out divisibility in Ш and consequently prove that there are no rational points.

In practice it is often easier to compute a 'fake' Selmer set. This is a set parameterising everywhere locally solvable unions of ϕ-coverings, with the property that every ϕ-covering lies in exactly one such union. Any locally soluble ϕ-covering gives rise to a locally soluble union, hence there is a map from the

Selmer set to the fake Selmer set. In general this map may be neither surjective nor injective. However, if the fake Selmer set is empty, then this is also true of the Selmer set.

Current functionality allows one to compute ϕ-Selmer groups for an isogeny ϕ of prime degree in a number of situations. For isogenies of degree 2 or 3, they may be computed using `TwoIsogenyDescent` and `ThreeIsogenyDescent`. When ϕ is the quotient by a subgroup generated by a $\mathbb{Q}$-rational point of order $p \in \{5, 7\}$ the dimensions of the ϕ- and $\phi^\vee$-Selmer groups can be computed using an algorithm of Fisher [Fis00] and [Fis01]. This method also produces models for the everywhere locally solvable $\phi^\vee$-coverings of E.

Given a genus one normal curve C of prime degree p produced by a ϕ-isogeny descent on an elliptic curve, the $\phi^\vee$-Selmer set of C can be computed using an algorithm of Creutz described in [Cre10] and [CM12]. Computing the $\phi^\vee$-Selmer sets of all elements in the ϕ-Selmer group yields information that is equivalent to that given by the $\phi^\vee \circ \phi$-Selmer group.

Current functionality allows one to perform these second isogeny descents when ϕ has degree $p = 3$ or when $p \in \{5, 7\}$ and the flex points of C lie on the coordinate hyperplanes in $\mathbb{P}^{p-1}$. If the kernel of the isogeny is generated by a $\mathbb{Q}$-rational point of order p, then such a model always exists. Moreover, the models returned by the algorithm of Fisher have this property. For $p = 7$, computation of the coverings is not practical and only a 'fake'-Selmer set can be computed (see [CM12]).

pIsogenyDescent(E, P): CrvEll, PtEll -> RngIntElt, RngIntElt, SeqEnum, CrvEll

pIsogenyDescent(E, p): CrvEll, RngIntElt -> RngIntElt, RngIntElt, SeqEnum, CrvEll

pIsogenyDescent(lambda, p): FldRatElt, RngIntElt -> RngIntElt, RngIntElt, SeqEnum, CrvEll, CrvEll

This performs a ϕ-descent on an elliptic curve over $\mathbb{Q}$ using the algorithm of Fisher. The descent requires an isogeny ϕ whose kernel is generated by a $\mathbb{Q}$-rational point of order $p \in \{5, 7\}$.

The input can be specified in one of three ways: (1) by giving an elliptic curve and a point of order p on the curve; (2) by giving an elliptic curve containing a point of order p and specifying p; or (3) by specifying p and a $\mathbb{Q}$-rational point on the modular curve $X_1(p)$. In the final case the choice for the coordinate λ on $X_1(p)(\mathbb{Q})$ is as described in [Fis00].

The return values are the p-ranks of the ϕ- and $\phi^\vee$-Selmer groups, a sequence of genus one normal curves of degree p representing the inverse pairs of nontrivial elements of the $\phi^\vee$-Selmer group modulo the image of the subgroup generated by the p-torsion point, and the isogenous elliptic curve. If the input is a coordinate on $X_1(p)$, the curve E is also returned.

pIsogenyDescent(C, phi): Crv, MapSch -> SeqEnum, List

pIsogenyDescent(C, E1, E2): Crv, CrvEll, CrvEll -> SeqEnum, List

pIsogenyDescent(C, P): Crv, PtEll -> SeqEnum, List

```
Selmer: Verbose                    Default: 3
```

This performs an isogeny descent on a genus one normal curve C over $\mathbb{Q}$ of degree $p \in \{3, 5\}$, as described in [CM12]. The curve C must be a projective plane cubic (in case $p = 3$) or an intersection of 5 quadrics in $\mathbb{P}^4$, with the additional requirement that the flex points of C lie on the coordinate hyperplanes. For the descent one must specify: (1) the isogeny ϕ; (2) the domain $E1$ and the codomain $E2$ of the isogeny; or (3) a $\mathbb{Q}$-rational point P of order p on an elliptic curve E which generates the kernel of the isogeny.

The return values are a sequence consisting of genus one normal curves of degree p representing the elements of the ϕ-Selmer set and a list of maps making these into ϕ-coverings of C.

FakeIsogenySelmerSet(C, phi): Crv, MapSch -> RngIntElt

FakeIsogenySelmerSet(C, E1, E2): Crv, CrvEll, CrvEll -> RngIntElt

FakeIsogenySelmerSet(C, P): Crv, PtEll -> RngIntElt

```
Selmer: Verbose                    Default: 3
```

This determines the $\mathbb{F}_p$-dimension of the 'fake'-ϕ-Selmer set of the genus one normal curve C of degree $p \in \{3, 5, 7\}$ (see [CM12] for the definition). By convention the empty set has dimension -1. The input is exactly as for `pIsogenyDescent`, however coverings are not produced. This makes the computations feasible as well for $p = 7$.

Example: Rank 0 via 5-Isogeny Descent (ex-a5a6be)

We use a 5-isogeny descent to show that the elliptic curve with Cremona reference 570l3 has rank 0 and that there are nontrivial elements of order 5 in its Shafarevich-Tate group.

```
> lambda := 48/5;
> r_phiSel,r_phidualSel,Sha,E1,E2 := pIsogenyDescent(lambda,5);
> CremonaReference(E1);
57011
> CremonaReference(E2);
57013
> TorsionSubgroup(E1);
Abelian Group isomorphic to Z/10
Defined on 1 generator
Relations:
    10*$.1 = 0
> r_phiSel;
0
> r_phidualSel;
3
```

This shows that $E_2(\mathbb{Q})/\phi(E_1(\mathbb{Q})) \simeq 0$, while the $\phi^\vee$-Selmer group gives an exact sequence: $0 \to E_2(\mathbb{Q})/\phi^\vee(E_1(\mathbb{Q})) \subset (\mathbb{Z}/5\mathbb{Z})^3 \to Ш(E_2/\mathbb{Q})[\phi^\vee] \to 0$. From this we conclude that the rank (for both curves) is 0, and so $Ш(E_2/\mathbb{Q})[\phi^\vee]$ has $\mathbb{F}_5$-dimension 2. Representatives for the inverse pairs of nontrivial elements are given by the third return value.

```
> #Sha*2 + 1;
25
> C := Sha[1];
> CremonaReference(Jacobian(GenusOneModel(C)));
57013
```

The above computation does not conclusively show that $Ш(E_1/\mathbb{Q})$ has no nontrivial 5-torsion. For this we need a second isogeny descent. Namely we show that the C is not divisible by ϕ in Ш.

```
> time DimSelC := FakeIsogenySelmerSet(C,E1,E2);
Time: 2.950
> DimSelC;
-1
```

Example: Large Generator via 5-Descent (ex-900bfb)

Now we use a full 5-descent to find a large generator.

```
> r1,r2,SelModTors,E,EE := pIsogenyDescent(326/467,5);
> r1,r2;
0 2
> E5,m := TorsionSubgroup(E);
> E5;
Abelian Group isomorphic to Z/5
Defined on 1 generator
Relations:
    5*E5.1 = 0
> time ConjecturalRegulator(E);
```

(continues on next page)

(continued from previous page)

```
242.01381346041570800013826895 8 1
Time: 70.180
> Conductor(E);
271976526950
> ThreeTorsionType(E);
Generic
```

This shows that $E(\mathbb{Q}) \simeq \mathbb{Z} \times \mathbb{Z}/5\mathbb{Z}$ and that (assuming $Ш(E/\mathbb{Q})$ is trivial) that the regulator is 242.01... This means one probably needs more than a 4-descent to find the generator. The conductor is too large for Heegner point techniques and the 3-torsion on E is generic, meaning 6- or 12-descent will be very slow (even if they are performed nonrigorously). We can do a further isogeny descent to obtain a full 5-covering.

```
> P := m(E5.1);
> C := Curve(Minimize(GenusOneModel(SelModTors[2])));
> time Ds,Pis := pIsogenyDescent(C,P);
7.880
> D := Ds[1];
> pi := Pis[1];
> time rp := PointSearch(D,10^10 :
>     Dimension := 1, OnlyOne := true, Nonsingular := true);
Time: 39.410
> Q := rp[1];
> Q;
(-11610740223/2573365369 : 1350220049/2573365369 :
-110993809/2573365369 : -3970329088/2573365369 : 1)
> piQ := pi(Q); // gives the point on C
> Dnew,Enew,FiveCovering := nCovering(GenusOneModel(D));
> Qnew := Dnew(Rationals())!Eltseq(Q);
> Ecan,EnewtoEcan := MinimalModel(Enew);
> P2 := EnewtoEcan(FiveCovering(Qnew));
> P2;
(187420468938753943863107148788058371189457516723324644302197836255922353361079466416310634589812396922966
1/99146671672911590582413148538700094541200749718044181289334064405545427071003081064161999700884312811 :
71245662543288432230853647434377610311616709098508077082874898143715069493690489458124709392518651254352942841744887961253603235705564184725054480767436761578/98722742040021206194215985208166099089985806405143789926209543140373470096170812446567353837866650186446834616709610109677611997365704706933027761807
1 : 1)
> CanonicalHeight(P2);
242.01381346041570800013826895
```

Example: Genuine and Fake Selmer Sets (ex-0153da)

Finally we give an example which shows that the map from the genuine Selmer set to the fake Selmer set need not be surjective.

```
> E1 := EllipticCurve("254a1");
> E2 := EllipticCurve("254a2");
> bool, phi := IsIsogenous(E1,E2);
> phicoveringsofE2,_,phidualcoveringsofE1,_,phi := ThreeIsogenyDescent(E1);
> C := phicoveringsofE2[1];
> C;
x^3 - x^2*y - x^2*z - 2*x*y^2 + 3*x*y*z - 2*x*z^2 + 2*y^2*z
> phidual := DualIsogeny(phi);
> MinimalModel(Codomain(phidual)) eq MinimalModel(Jacobian(C));
```

(continues on next page)

(continued from previous page)

```
true
> time FakeIsogenySelmerSet(C,phidual);
2
Time: 0.620
> time SelC := pIsogenyDescent(C,phidual);
Time: 1.160
> Ilog(3,#SelC);
1
```

133.6 Bibliography

- [BC84] A. Bremner and J. W. S. Cassels. On the equation $Y^2 = X(X^2 + p)$. *Math. Comp.*, 42(165):257–264, 1984.
- [Bru02] N. R. Bruin. *Chabauty methods and covering techniques applied to generalized Fermat equations*. Volume 133 of CWI Tract. Stichting Mathematisch Centrum Centrum voor Wiskunde en Informatica, Amsterdam, 2002. ISBN 90-6196-508-X. Dissertation, University of Leiden, Leiden, 1999.
- [Bru] N. Bruin. "Some ternary Diophantine equations of signature $(n, n, 2)$."
- [Bru03] Nils Bruin. Chabauty methods using elliptic curves. *J. reine angew. Math.*, 562:27–49, 2003.
- [BS10] Nils Bruin and Michael Stoll. The Mordell-Weil sieve: Proving non-existence of rational points on curves. *LMS J. Comput. Math.*, 13:272–306, 2010.
- [Cas66] J. W. S. Cassels. Diophantine equations with special reference to elliptic curves. *J. London Math. Soc.*, 41:150–158, 1966.
- [Coh93] Henri Cohen. *A Course in Computational Algebraic Number Theory*. Volume 138 of Graduate Texts in Mathematics. Springer, Berlin–Heidelberg–New York, 1993.
- [CFO+08] J. E. Cremona, T. A. Fisher, C. O'Neil, D. Simon, and M. Stoll. Explicit n-descent on elliptic curves. I. Algebra. *J. Reine Angew. Math.*, 615:121–155, 2008.
- [CFO+] J.E. Cremona, T.A Fisher, C. O'Neil, D. Simon, and M Stoll. Explicit n-Descent On Elliptic Curves, III. Algorithms. ArXiv preprint. http://arxiv.org/abs/1107.3516.
- [CFO+09] J.E. Cremona, T.A Fisher, C. O'Neil, D. Simon, and M Stoll. Explicit n-Descent On Elliptic Curves, II. Geometry. *J. reine angew. Math.*, 632:63–84, 2009.
- [CFS10] J.E. Cremona, T.A Fisher, and M Stoll. Minimisation and reduction of 2-, 3- and 4-coverings of elliptic curves. *Algebra & Number Theory*, 4(6):763–820, 2010.
- [Cre99] John Cremona. Reduction of binary cubic and quartic forms. *LMS JCM*, 2:62–92, 1999.
- [Cre01] John Cremona. Classical invariants and 2-descent on elliptic curves. *J. Symbolic Comp.*, 31:71–87, 2001.
- [CM12] B. Creutz and R.L. Miller. Second isogeny descents and the Birch and Swinnerton-Dyer conjectural formula. *J.Algebra*, 372:673–701, 2012.
- [Cre10] Brendan Creutz. *Explicit second p-descents on elliptic curves*. PhD Thesis, Jacobs University Bremen, 2010.
- [Elk] N. Elkies. "Rational Points Near Curves and Small Nonzero $|x^3 - y^2|$ via Lattice Reduction." 33–63.
- [Fisb] (no details available)
- [Fis00] Tom Fisher. *On 5 and 7 descents for elliptic curves*. PhD thesis, University of Cambridge, 2000.

- [Fis01] Tom Fisher. Some examples of 5 and 7 descent for elliptic curves over Q. *J. Eur. Math. Soc.*, 3(Issue 2):169–201, 2001.
- [Fis08b] Tom Fisher. Finding rational points on elliptic curves using 6-descent and 12-descent. *J. Algebra*, 320(2):853–884, 2008.
- [GZ86] Gross and Zagier. Heegner Points and Derivatives of L-series. *Invent. Math.*, 84:225–320, 1986.
- [Har08] D. Harvey. Efficient computation of p-adic heights. *LMS J. Comput. Math.*, 11:40–59, 2008.
- [Kra81] K. Kramer. Arithmetic of elliptic curves upon quadratic extension. *Trans. Amer. Math. Soc.*, 264(1):121–135, 1981.
- [MST06] B. Mazur, W. Stein, and J. Tate. Computation of p-adic heights and log convergence. *Documenta Mathematica*, Extra:577–614, 2006.
- [MT91] B. Mazur and J. Tate. The p-adic sigma function. *Duke Math Journal*, 62(3):663–688, 1991.
- [MSS96] J. R. Merriman, S. Siksek, and N. P. Smart. Explicit 4-descents on an elliptic curve. *Acta Arith.*, 77(4):385–404, 1996.
- [SS04] Edward F. Schaefer and Michael Stoll. How to do a p-descent on an elliptic curve. *Trans. Amer. Math. Soc.*, 356(3):1209–1231 (electronic), 2004.
- [Sik95] Samir Siksek. Infinite descent on elliptic curves. *Rocky Mountain J. Math.*, 25(4):1501–1538, 1995.
- [Sil86] J. Silverman. *The arithmetic of elliptic curves.* Volume 106 of Graduate Texts in Mathematics. Springer-Verlag, New York, 1986.
- [SW] (no details available)
- [ST96] R. J. Stroeker and Tzanakis T. Solving elliptic diophantine equations by estimating linear forms in elliptic logarithms, the quartic case. *Acta Arith.*, 75:165–190, 1996.
- [ST94] R. J. Stroeker and N. Tzanakis. Solving elliptic diophantine equations by estimating linear forms in elliptic logarithms. *Acta Arith.*, 67:177–196, 1994.
- [Wat02] M. Watkins. Computing the modular degree of an elliptic curve. *Experimental Mathematics*, 11(4):487–502, 2002.

Chapter 134

Elliptic Curves over Function Fields

This section involves elliptic curves with coefficients in a function field $k(C)$ where C is a regular projective curve over some field k (usually a number field or a finite field). The commands are largely parallel to those for elliptic curves over the rationals; one can compute local information (Tate's algorithm and so forth), a minimal model, the L-function, the 2-Selmer group, and the Mordell–Weil group. This goes in order of decreasing generality: Local information is available for curves over univariate function fields over any exact base field, while at the other extreme Mordell–Weil groups are available only for curves over rational function fields over finite fields for which the associated surface is a rational surface. The generality of many of the commands will be expanded in future releases.

134.1 An Overview of Relevant Theory

An elliptic curve over $K = k(C)$ may be regarded as a surface $\mathcal{E}$ over k with a map $\pi : \mathcal{E} \to C$ (in other words, an *elliptic surface*); the generic fibre of $\mathcal{E}$ is E. Under this interpretation, elements of the Mordell–Weil group $E(K)$ are in one-to-one correspondence with sections of π. (A section is a morphism $s : C \to \mathcal{E}$ such that $\pi \circ s = \mathrm{Id}_C$.) This means that one may study the Mordell–Weil group by studying the geometry of the surface.

Given E, there is a unique $\mathcal{E}$ up to isomorphism that is projective, regular, and relatively minimal. This is called the Kodaira–Néron model, and we will always assume that we are working with this model of the surface.

Let $\bar{k}$ denote the separable closure of k, and $\mathcal{E}_{\bar{k}}$ the elliptic surface considered over $\bar{k}$. The Néron–Severi group $\mathrm{NS}(\mathcal{E}_{\bar{k}})$ of $\mathcal{E}_{\bar{k}}$ is the group of divisors of $\mathcal{E}_{\bar{k}}$ modulo algebraic equivalence. It is a finitely generated group and is closely connected with the Mordell–Weil group $E(K)$.

Let N be the subgroup of $\mathrm{NS}(\mathcal{E}_{\bar{k}})$ that is generated by all components of all the fibres of π together with the section corresponding to the zero point of $E(\bar{k}(C))$; this is known as the *trivial lattice* of $\mathrm{NS}(\mathcal{E}_{\bar{k}})$. It can easily be determined since the number of components in reducible fibres can be computed by Tate's algorithm. The following divisor classes together form a basis of N

(i) the image of the section corresponding to the zero point;

(ii) one complete fibre; and

(iii) the components of all the reducible fibres, with one component from each fibre omitted.

It is known that the quotient $\mathrm{NS}(\mathcal{E}_{\bar{k}})/N$ is generated by images of sections of π, and that $\mathrm{NS}(\mathcal{E}_{\bar{k}})/N \cong E(\bar{k}(C))$ (via the identification of sections with points). In particular, this implies the Shioda–Tate formula $\mathrm{rank}(E(\bar{k}(C))) + 2 + \sum_{v \in C(\bar{k})}(m_v - 1) = \mathrm{rank}(\mathrm{NS}(\mathcal{E}_{\bar{k}}))$ where m_v denotes the number of components of the fibre $\pi^{-1}(v)$.

The Galois group $G = G_{\bar{k}/k}$ acts on $\mathrm{NS}(\mathcal{E}_{\bar{k}})$, and it maps N to itself. Moreover, after extending scalars to $\mathbb{Q}$ one can split the Galois representation. That is, there exists $M \subset \mathrm{NS}(\mathcal{E}_{\bar{k}}) \otimes \mathbb{Q}$ such that $\mathrm{NS}(\mathcal{E}_{\bar{k}}) \otimes \mathbb{Q} \cong M \oplus (N \otimes \mathbb{Q})$ and $M \cong E(\bar{k}(C)) \otimes \mathbb{Q}$ as G-modules. In particular, $M^G \cong E(K) \otimes \mathbb{Q}$.

In the case that k is a finite field, the Frobenius action on N can be determined with the functions `FrobeniusActionOnReducibleFiber` and `FrobeniusActionOnTrivialLattice`.

In order to study $\mathrm{NS}(\mathcal{E}_{\bar{k}})$ as a G-module, one can embed it in the ℓ-adic cohomology group $H^2(\mathcal{E}_{\bar{k}}, \mathbb{Q}_\ell)$. To get a G-equivariant map one must slightly change the G-action on $H^2(\mathcal{E}_{\bar{k}}, \mathbb{Q}_\ell)$. Let $H^2(\mathcal{E}_{\bar{k}}, \mathbb{Q}_\ell)(1)$ denote the (1)-Tate twist of $H^2(\mathcal{E}_{\bar{k}}, \mathbb{Q}_\ell)$. The main property that we need to know about this twist is that it transforms the q-eigensubspace in $H^2(\mathcal{E}_{\bar{k}}, \mathbb{Q}_\ell)$ of some q-Frobenius element to the 1-eigenspace of this Frobenius in $H^2(\mathcal{E}_{\bar{k}}, \mathbb{Q}_\ell)(1)$. The cycle class map then yields a G-equivariant embedding

$$\mathrm{NS}(\mathcal{E}_{\bar{k}}) \otimes \mathbb{Q}_\ell \hookrightarrow H^2(\mathcal{E}_{\bar{k}}, \mathbb{Q}_\ell)(1).$$

It is conjectured by Tate that the image of $(\mathrm{NS}(\mathcal{E}_{\bar{k}}) \otimes \mathbb{Q}_\ell)^G$ under this map exactly equals $H^2(\mathcal{E}_{\bar{k}}, \mathbb{Q}_\ell)(1)^G$.

In the case that k is a finite field one can in principal determine $H^2(\mathcal{E}_{\bar{k}}, \mathbb{Q}_\ell)(1)^G$ via the Lefschetz trace formula. Suppose that $k \subset \mathbb{F}_q$ and let F_q denote the q-th power Frobenius map. Then

$$\#\mathcal{E}(\mathbb{F}_q) = 1 + q^2 - (1+q)\,\mathrm{Trace}(F_q | H^1(C_{\bar{k}}, \mathbb{Q}_\ell)) + \mathrm{Trace}(F_q | H^2(\mathcal{E}_{\bar{k}}, \mathbb{Q}_\ell)).$$

The trace on $H^1(C_{\bar{k}}, \mathbb{Q}_\ell)$ is zero if C is a rational curve, and it can be determined by counting $\mathbb{F}_q$-rational points on C in the general case. Hence one can determine $\mathrm{Trace}(F_q | H^2(\mathcal{E}_{\bar{k}}, \mathbb{Q}_\ell))$ by counting $\mathbb{F}_q$-rational points on $\mathcal{E}$. By doing this for various powers of q one can determine the characteristic polynomial of F_q acting on $H^2(\mathcal{E}_{\bar{k}}, \mathbb{Q}_\ell)$, and hence the conjectural ranks of $E(\bar{k}(C))$ and $E(K)$ by using Tate's conjectures. The conjectural ranks obtained in this way give unconditional upper bounds on the true ranks.

As the Galois action on N can be determined, the difficult part is to compute

$$\det(1 - T \cdot F_q | H^2(\mathcal{E}_{\bar{k}}, \mathbb{Q}_\ell) / \mathrm{im}(N \otimes \mathbb{Q}_\ell)),$$

where $\mathrm{im}(N \otimes \mathbb{Q}_\ell)$ stands for the image of $N \otimes \mathbb{Q}_\ell$ under the cycle class map. It can be shown that this polynomial is equal to the L-function of E over K; it follows that this L-function is a polynomial. This shows that Tate's conjectures are linked to a geometric version of the Birch and Swinnerton-Dyer conjecture. Just as in the number field case, this conjecture expresses the rank and the product of the order of the Tate–Shafarevich group and the regulator of E in terms of its L-function. The L-function can be computed with the function `LFunction` and the conjectural information on the rank, Tate–Shafarevich group, and regulator can be obtained with the function `AnalyticInformation`.

If E can be defined by a Weierstrass equation in which the coefficients a_i are polynomials of degree at most i, then $\mathcal{E}_{\bar{k}}$ is a rational surface and $\mathrm{rank}(\mathrm{NS}(\mathcal{E}_{\bar{k}})) = 10$. In this case $\mathrm{rank}(E(\bar{k}(C))) = 10 - \mathrm{rank}(N)$ can be easily determined. In the case that k is a finite field then $E(\bar{k}(C))$ and $E(K)$ can be computed using functions in this section.

134.2 Local Computations

`BadPlaces(E): CrvEll -> [ PlcFunElt ]`

A sequence containing the places where the given model of E has bad reduction, for an elliptic curve E defined over a function field.

`Conductor(E): CrvEll -> DivFunElt`

The conductor of an elliptic curve E defined over a function field F. In general this is returned as a divisor of F. When F is a rational function field it is returned as a sequence of tuples $\langle f, e \rangle$ of places (specified by field elements f) and multiplicities e.

`LocalInformation(E, Pl): CrvEll[FldFun], PlcFunElt -> Tup, CrvEll`
`LocalInformation(E, f): CrvEll[FldFunRat], FldFunRatUElt -> Tup, CrvEll`

This function performs Tate's algorithm for an elliptic curve E over a function field to determine the reduction type and a minimal model at the given place Pl. When E is defined over a rational function field $F(t)$ the place is simply given as a field element f (which must either be $1/t$ or an irreducible polynomial in t.)

The model is not required to be integral on input. The output is of the form $\langle Pl, v_p(d), f_p, c_p, K, split \rangle$ and E_{min} where Pl is the place, $v_p(d)$ is the valuation of the local minimal discriminant, f_p is the valuation of the conductor, c_p is the Tamagawa number, K is the

Kodaira Symbol, *split* is a boolean that is false if reduction is of nonsplit multiplicative type and true otherwise, and E_{min} is a model of E (integral and) minimal at Pl.

`LocalInformation(E): CrvEll -> [ < Tup > ]`

Returns a sequence of tuples as described above for all places of bad reduction of the elliptic curve E.

`KodairaSymbols(E): CrvEll -> [ <SymKod, RngIntElt> ]`

A sequence of tuples $\langle K, n \rangle$, corresponding to the places of bad reduction of the elliptic curve E. Here K is the Kodaira symbol and n is the degree of the corresponding place.

`NumberOfComponents(K): SymKod -> RngIntElt`

The number of components of a fibre with the Kodaira symbol K.

`MinimalModel(E): CrvEll[FldFunG] -> CrvEll, MapIsoSch`

A model of the elliptic curve E (defined over a function field, which must have genus 0) that is minimal at all finite places, together with a map from E to this minimal model.

`MinimalDegreeModel(E): CrvEll[FldFunRat] -> CrvEll, Map, Map`

A model of the elliptic curve E (defined over a rational function field) which minimises the quantity `Max`([`Degree`$(a_i)/i$]), where a_1, a_2, a_3, a_4, a_6 are the Weierstrass coefficients.

`IsConstantCurve(E): CrvEll[FldFunRat] -> BoolElt, CrvEll`

For an elliptic curve E defined over a rational function field $F(t)$, the function returns `true` if and only if E is isomorphic over $F(t)$ to an elliptic curve with coefficients in F (and also returns such a curve in that case).

`TraceOfFrobenius(E, p): CrvEll[FldFunRat], RngElt -> BoolElt, CrvEll`

The trace of Frobenius a_p for the reduction of E at the place p, specified as an element of the base field.

134.3 Elliptic Curves of Given Conductor

`EllipticCurveSearch(N, Effort): [], RngIntElt -> SeqEnum`

```
Full    : BoolElt                    Default: false
Max     : RngIntElt                  Default:
Primes  : SeqEnum                    Default:
Traces  : SeqEnum                    Default:
ECSearch: Verbose                    Default: 2
```

This routine searches for elliptic curves over a field $F_q(t)$ whose conductor is equal to the specified conductor N. Alternatively, the first argument may be a set or sequence of possible conductors. Here a conductor is specified in the same format as returned by `Conductor`.

For explanation of the optional parameters and other details, see the description of `EllipticCurveSearch` for elliptic curves over number fields.

134.4 Heights

`NaiveHeight(P): PtEll -> FldPrElt`

The naive x-coordinate height of a point P on an elliptic curve over a function field K; in other words, the degree of the point $(x(P) : 1)$ on the projective line.

`Height(P): PtEll -> FldRatElt`

The Néron–Tate height $\hat{h}(P)$ of the given point P on an elliptic curve defined over a function field.

`LocalHeight(P, Pl): PtEll, PlcFunElt -> FldPrElt`

Given a point P on an elliptic curve defined over a function field F and a place Pl of the function field F, returns the local height $\lambda_{Pl}(P)$ at Pl of P.

`HeightPairing(P, Q): PtEll[FldFunG], PtEll[FldFunG] -> FldRatElt`

Returns the height pairing of the points P and Q, defined as $\langle P, Q\rangle = (\hat{h}(P+Q) - \hat{h}(P) - \hat{h}(Q))/2$ (where as usual $\hat{h}$ denotes the Néron–Tate height).

`HeightPairingMatrix(S): SeqEnum[PtEll[FldFunG]] -> AlgMatElt`

Given a sequence S of points P_i on an elliptic curve defined over a function field, this function returns the matrix $(\langle P_i, P_j\rangle)$, where $\langle , \rangle$ is the height pairing.

`HeightPairingLattice(S): [PtEll[FldFunG]] -> AlgMatElt, Map`

The height pairing lattice of a sequence of independent points on an elliptic curve defined over a function field.

`Basis(S): [ PtEll ] -> [ PtEll ], ModMatAlgElt`

Given a sequence S of points on an elliptic curve, returns a sequence of points that form a basis for the free part of the subgroup generated by the points in S. The second returned value is a Gram matrix for this basis with respect to the Néron–Tate pairing.

`Basis(S, r, disc): SeqEnum, RngIntElt, RngIntElt -> SeqEnum`

Given a sequence S of points on an elliptic curve, returns a sequence of independent points in the free part of the subgroup generated by S such that these points generate a lattice of rank r and discriminant *disc*. The answer is returned as soon as such a lattice has been found, ignoring any additional points in the given sequence.

`IsLinearlyDependent(points): [PtEll] -> BoolElt, ModTupRngElt`

`IsLinearlyIndependent(points): [PtEll] -> BoolElt, ModTupRngElt`

`IndependentGenerators(points): [PtEll] -> [PtEll]`

These functions are available for elliptic curves over function fields, and behave the same way as for elliptic curves over the rationals.

134.5 The Torsion Subgroup

`TorsionSubgroup(E): CrvEll[FldFunG] -> GrpAb, Map`

Given an elliptic curve E defined over a function field F, this function returns an abelian group A isomorphic to the torsion subgroup of $E(F)$, together with a map from A to $E(F)$.

`TorsionBound(E, n): CrvEll[FldFunG], RngIntElt -> RngIntElt`

`TorsionBound(E, n, B): CrvEll[FldFunG], RngIntElt, RngIntElt -> RngIntElt`

Given an elliptic curve over a function field F and an integer n, this function computes a bound on the size of the torsion subgroup of $E(F)$ by considering the torsion subgroups of the fibres of E at n different places of F.

When an integer B is given as a third argument then the subgroup of elements of order dividing B is bounded, rather than the whole torsion subgroup.

`GeometricTorsionBound(E): CrvEll[FldFunG] -> RngIntElt`

Given an elliptic curve E defined over a function field F, this function computes a bound for the geometric torsion subgroup of E. That is, the torsion group of $E(K)$ where K/F is the smallest extension with algebraically closed constant field. In cases where a bound cannot be computed then 0 is returned.

134.6 The Mordell–Weil Group

The machinery in this section and the next section is for curves defined over a rational function field $k(t)$ whose field of constants k is finite.

The Mordell–Weil group can be computed (and generators found) for a curve $y^2 + a_1xy + a_3y = x^3 + a_2x^2 + a_4x + a_6$ where each a_i is a *polynomial* in $k[t] \subset k(t)$ of degree at most i.

When this hypothesis is not satisfied, it may still be possible to bound the Mordell–Weil rank and find Mordell–Weil generators using the 2-descent routines described in the next section.

RankBounds(E): CrvEll[FldFunG] -> RngIntElt, RngIntElt

RankBound(E): CrvEll[FldFunG] -> RngIntElt

These functions return lower and upper bounds (or just an upper bound) on the rank of the Mordell–Weil group $E(F)$ for an elliptic curve E defined over a function field F with finite constant field. The bound is obtained by applying all the available tools (those described in this section together with the `AnalyticInformation` obtained from the L-function).

MordellWeilGroup(E : parameters): CrvEll[FldFunRat] -> GrpAb, Map

Al: MonStgElt	Default: "Geometric"

This function computes the Mordell–Weil group of an elliptic curve E that satisfies the hypotheses stated in the introduction immediately above. The function returns two values: an abelian group A and a map m from A to E. The map m provides an isomorphism between the abstract group A and the Mordell–Weil group.

The algorithm used by default is the geometric method described above. However, when `Al` is set to `"Descent"` it instead uses the 2-descent tools described in the next section; if the curve admits 2-isogenies then it uses a separate implementation of descent by 2-isogenies (described in [Rob07]). These descent methods do not always determine the full Mordell–Weil group (in which case a warning is printed); their advantages are that they do not require the degrees of the coefficients to be bounded and in many cases are very efficient.

MordellWeilLattice(E): CrvEll[FldFunRat] -> Lat, Map

This function computes the Mordell–Weil lattice of an elliptic curve E that satisfies the hypotheses stated in the introduction immediately above. This is the free part of the Mordell–Weil group with an inner product given by the Néron–Tate height pairing. The function returns two values: the lattice L and a map m from L to E.

GeometricMordellWeilLattice(E): CrvEll[FldFunRat] -> Lat, Map

This function computes the geometric Mordell–Weil lattice of an elliptic curve E that satisfies the hypotheses stated in the introduction immediately above. This consists of the free part of the group of points on E that are defined over the function field with a possibly extended constant field, together with the Néron–Tate pairing. The function returns two values: a lattice L and a map m from L to E', where E' is a base change of E over the larger field.

Generators(E): CrvEll[FldFunRat] -> SeqEnum

Given an elliptic curve E over a rational function field F that satisfies the hypotheses stated in the introduction immediately above, this function returns a sequence of points in $E(F)$ which are generators of the Mordell–Weil group.

Example: Rank 2 Curve over a Function Field (ex-ee4c8d)

We find that the curve $y^2 = x^3 + (t^4 + 2t^2)x + t^3 + 2t$ over $F_3(t)$ has rank 2 and has no 2-torsion in its Tate–Shafarevich group.

```
> F<t> := FunctionField(GF(3));
> E := EllipticCurve([ t^4 + 2*t^2, t^3 + 2*t ]);
> S2E := TwoSelmerGroup(E);
> S2E;
Abelian Group isomorphic to Z/2 + Z/2
Defined on 2 generators
Relations:
    2*S2E.1 = 0
    2*S2E.2 = 0
```

(continues on next page)

(continued from previous page)

```
> MordellWeilGroup(E);
Abelian Group isomorphic to Z + Z
Defined on 2 generators (free)
Mapping from: Abelian Group isomorphic to Z + Z
Defined on 2 generators (free) to CrvEll: E given by a rule [no inverse]
```

Furthermore, we may compute the regulator of E as follows.

```
> Determinant(HeightPairingMatrix(Generators(E)));
12
```

134.7 Two Descent

In odd characteristic the 2-Selmer group can be computed and its elements can be represented as minimised 2-coverings. A search for points on these 2-coverings can be made.

In characteristic 2 descent by 2-isogeny can be performed for an ordinary curve E using the isogenies $E \to E^{frob} \to E$. The Selmer groups for both isogenies are computed; if these have ranks s_1 and s_2 then the rank of E is at most $s_1 + s_2 - 1$.

TwoSelmerGroup(E): CrvEll[FldFunG] -> GrpAb, MapSch

This function computes the 2-Selmer group of an elliptic curve E defined over a rational function field $\mathbb{F}_q(t)$ of odd characteristic. This is returned as an abstract group together with a map from the group to the relevant algebra.

The algorithm is standard (similar to the one used for elliptic curves over number fields).

TwoDescent(E): CrvEll[FldFunG] -> SeqEnum[CrvHyp], List[MapSch]

This represents the nontrivial elements of the 2-Selmer group of E as hyperelliptic curves $C : y^2 = f(x)$ of degree 4, and returns a sequence containing the curves together with a list containing the corresponding covering maps $C \to E$. The elliptic curve should be defined over a rational function field $\mathbb{F}_q(t)$ of odd characteristic.

The curves returned have polynomial coefficients and are minimised at all finite places. The conic parametrisation step uses the algorithm by Cremona and van Hoeij.

QuarticMinimize(f): RngMPolElt[FldFunRat] -> RngMPolElt[FldFunRat]

This is the routine used by `TwoDescent` to minimise two-coverings. The function takes a homogeneous quartic in two variables with coefficients in a rational univariate function field and returns a minimal quartic equivalent to f together with the appropriate transformation matrix.

Points(C : parameters): CrvHyp -> [Pt]

```
Bound: RngIntElt                    Default:
```

This finds all rational points with height up to the given `Bound` on the hyperelliptic curve C, which must be defined over a rational function field $\mathbb{F}_q(t)$. The parameter `Bound` is *not optional*, and it refers to the x-coordinate (logarithmic) height of the points: In other words, the routine finds all projective points $(X : Y : Z)$ where X and Z are polynomials in t of degree less than or equal to `Bound`.

PointsQI(C, H): Crv, RngIntElt -> [Pt]

```
OnlyOne   : BoolElt                 Default: false
ExactBound: BoolElt                 Default: false
```

This is an optimised routine for finding rational points on a curve given as an intersection of two quadrics defined over a rational function field $\mathbb{F}_q(t)$. It searches for projective points whose

coordinates are polynomials in t of degree up to H. To guarantee finding *all* such points the parameter `ExactBound` must be set to true.

The algorithm uses a lattice reduction method described in [Rob07].

`TwoIsogenySelmerGroups(E): CrvEll[FldFunG] -> GrpAb, GrpAb, MapSch, MapSch`

This performs descent by 2-isogenies for a non-supersingular elliptic curve E defined over a rational function field $k(t)$ where k is finite *of characteristic 2*. The isogenies used are the Frobenius map $E \to E^{frob} : (x, y) \mapsto (x^2, y^2)$ and the dual isogeny $E^{frob} \to E$ (which is separable). The function returns four objects: the Selmer group S^{sep} of the separable isogeny (as an abstract group), followed by the Selmer group S^{Frob} of the Frobenius isogeny, followed by maps $S^{sep} \to k(t)/\Phi(k(t))$ and $S^{Frob} \to k(t)^*/(k(t)^*)^2$. Here Φ denotes the Artin–Schreier map $a \mapsto a^2 + a$.

Notes describing the algorithm will be made available (on request). The theoretical background is presented in [Kra77].

134.8 The L-function and Counting Points

To compute the L-function one can use the intrinsic `LFunction` or alternatively one can do it "by hand", using the tools in this section and the next (as illustrated in Example *Partial L-Function by Hand*).

`LFunction(E): CrvEll[FldFunRat] -> RngUPolElt`

`LFunction(E, S): CrvEll[FldFunRat], <SeqEnum> -> RngUPolElt`

`Check: BoolElt` `Default: true`

The L-function of the elliptic curve E, which should be defined over a rational function field over a finite field.

To speed up the computation, the points in S are used to reduce the unknown part of H^2; S should be a sequence of points on E or on some base change of E by an extension of the constant field. If the parameter `Check` is set to `false` then the points in S must form a basis for a subgroup of the geometric Mordell–Weil group modulo torsion that is mapped to itself by the Galois group of the extension of the fields of constants.

`LFunction(E, e): CrvEll[FldFunRat], RngIntElt -> RngUPolElt`

The L-function for the base change of the elliptic curve E by a constant field extension of degree e.

`AnalyticRank(E): CrvEll[FldFunG] -> RngIntElt`

The Mordell–Weil rank of E as predicted by Tate's version of the Birch–Swinnerton-Dyer conjecture (or equivalently, the Artin–Tate conjecture). This is an unconditional upper bound and is proved to be the actual rank for several classes of curves, such as those arising from rational elliptic surfaces and K3 elliptic surfaces. It is required that E is defined over a function field over a finite field.

`AnalyticInformation(E): CrvEll[FldFunG] -> Tup`

`AnalyticInformation(E, L): CrvEll[FldFunG], RngUPolElt -> Tup`

`AnalyticInformation(E, e): CrvEll[FldFunG], RngIntElt -> Tup`

This function computes what is predicted by the Birch–Swinnerton-Dyer (or Artin–Tate) conjecture about E, which should be an elliptic curve defined over a function field over a finite field. The data is given as a tuple consisting of the rank, geometric rank, and the product of the order of Tate–Shafarevich group and the regulator.

When a polynomial L is given as a second argument then function will assume that L is the L-function of E.

When an integer e is given as a second argument then the function will return analytic data for the base change of E by the constant field extension of degree e. This is more efficient than simply calling `AnalyticInformation` on the base change.

Example: Tate-Shafarevich Group via the L-Function (ex-b7cb32)

The curve $y^2 = x^3 + 2t^2x + (t^3 + 2t^2 + 2t)$ over $\mathbb{F}_3(t)$ has rank 0 and its Tate–Shafarevich group contains $\mathbb{Z}/3 \oplus \mathbb{Z}/3$. We may conclude this (unconditionally) from the analytic information, since the Tate conjecture is proved for rational elliptic surfaces.

```
> F<t> := FunctionField(GF(3));
> E := EllipticCurve([ 2*t^2, t^3 + 2*t^2 + 2*t ]);
> AnalyticInformation(E);
<0, 4, 9>
```

Furthermore, the rank of $E(\overline{F_3}(t))$ is 4.

Example: L-Function of a Rank 2 Curve (ex-f412e7)

We consider again the curve $y^2 = x^3 + (t^4 + 2t^2)x + t^3 + 2t$ over $\mathbb{F}_3(t)$, which was found to have rank 2 and regulator 12 in Example *Rank 2 Curve over a Function Field*.

```
> F<t> := FunctionField(GF(3));
> E := EllipticCurve([ t^4 + 2*t^2, t^3 + 2*t ]);
> AnalyticInformation(E);
<2, 8, 12>
```

This confirms that the rank is 2 and tells us that the regulator multiplied by the order of Sha is 12; hence the Tate–Shafarevich group is trivial. Again this is unconditional.

`NumberOfPointsOnSurface(E, e): CrvEll, RngIntElt -> RngIntElt`

Returns the number of points on the Kodaira–Néron model of the elliptic curve E over an extension of the constant field of degree e.

`NumbersOfPointsOnSurface(E, e): CrvEll, RngIntElt -> [ RngIntElt ], [ RngIntElt ]`

If an elliptic curve E is defined over a function field over a finite field of q elements then this intrinsic computes a sequence containing the number of points on the corresponding Kodaira–Néron model of E over extensions of $\mathbb{F}_q$ of degree up to e, together with the number of points on the base curve over these extensions.

`BettiNumber(E, i): CrvEll, RngIntElt -> RngIntElt`

Returns the i-th Betti number of the Kodaira–Néron model of the elliptic surface corresponding to the elliptic curve E.

`CharacteristicPolynomialFromTraces(traces): [ Fld ] -> RngUPolElt`

Given a sequence $traces = [t_1, t_2, \ldots, t_d]$, this function computes a polynomial of degree d with constant coefficient equal to 1 such that the j-th powersum of the reciprocal zeroes of this polynomial is equal to t_j. This is typically used when one wants to compute the characteristic polynomial of Frobenius acting on some piece of cohomology from traces of various powers of Frobenius.

`CharacteristicPolynomialFromTraces(traces, d, q, i): [ Fld ], RngIntElt, RngIntElt, RngIntElt -> RngUPolElt, RngUPolElt`

Given a sequence of field elements $[t_1, t_2, \ldots, t_k]$ and integers d, q, and i, this function computes a polynomial of degree d with constant coefficient 1 such that the reciprocals of the zeroes of this polynomial have absolute value $q^{i/2}$ and the j-th powersum of the reciprocals of these zeroes equals t_j. For this function one must have k at least equal to Floor$(d/2)$. A priori the leading coefficient of the polynomials is not determined from the list of traces; in this case two polynomials are returned, one for each choice of sign. One usually needs to provide Ceiling$(d/2)$ traces in order to determine the sign, but in certain cases more traces are required. This function is typically used when one wants to compute the characteristic polynomial of Frobenius acting on a piece of the i-th cohomology group from traces of various powers of Frobenius.

134.9 Action of Frobenius

Frobenius(P, q): PtEll[FldFunRat], RngIntElt -> PtEll

The q-th power Frobenius map on the point P of an elliptic curve that can be defined over a function field with constant field $\mathbb{F}_q$).

FrobeniusActionOnPoints(S, q : parameters): [PtEll], RngIntElt -> AlgMatElt

gram: AlgMatElt	Default:

A matrix representing the q-power Frobenius map on the subgroup of the geometric Mordell–Weil group (modulo torsion) with the given basis S. (This subgroup is assumed to be invariant under the q-power Frobenius.)

The optional parameter gram should be the Gram matrix with respect to the height pairing of the points in S.

FrobeniusActionOnReducibleFiber(L): < Tup > -> AlgMatElt

Given reduction data L for an elliptic curve E, such as given by the command LocalInformation, this function returns a matrix representing the Frobenius action on the non-identity components of corresponding fibres.

FrobeniusActionOnTrivialLattice(E): CrvEll -> AlgMatElt

Given an elliptic curve E defined over a rational function field over a finite field, returns a matrix representing the Frobenius action on fibre components and the zero section of the corresponding elliptic surface.

134.10 Extended Examples

Example: Elliptic Curve from a Pencil of Cubics (ex-4e11bd)

In this example we construct an elliptic curve by using a pencil of cubic curves passing through 8 given points. The 8 points are defined over $GF(1831^8)$ and form a Galois orbit over $GF(1831)$.

```
> p := 1831;
> F := GF(p);
> Fe<u> := ext<F | 8>;
> K<t> := FunctionField(F);
> P2<X,Y,Z> := ProjectivePlane(K);
> // define the 8 points:
> points := [ [ u^(p^i), (u^3+u+1)^(p^i) ] : i in [0..7] ];
> M := [ [ p[1]^i*p[2]^j : i in [0..3-j], j in [0..3] ] : p in points ];
> // find the coefficients of 2 cubics that pass through the points:
> B := Basis(Kernel(Transpose(Matrix(M))));
> R<x,y> := PolynomialRing(F, 2);
> mono_aff := [ x^i*y^j : i in [0..3-j], j in [0..3] ];
> // f1 and f2 are the cubics:
> f1 := &+[ (F!B[1][i])*mono_aff[i] : i in [1..10] ];
> f2 := &+[ (F!B[2][i])*mono_aff[i] : i in [1..10] ];
> // Find the 9th intersection point, which we use as zero, to put
> // it to a nice Weierstrass model :
> P9 := Points(Scheme(Spec(R), [f1, f2]))[1];
> F1 := Homogenization(Evaluate(f1, [X, Y]), Z);
> F2 := Homogenization(Evaluate(f2, [X, Y]), Z);
> C := Curve(P2, F1 + t*F2);
> E := MinimalDegreeModel(EllipticCurve(C, C![P9[1], P9[2]]));
```

We could transfer the 8 points to this Weierstrass model, and use them to determine the Mordell–Weil group. Instead, we will see what Magma is able to compute just from the Weierstrass model.

```
> KodairaSymbols(E);
[ <I1, 1>, <I1, 7>, <II, 1>, <I1, 2> ]
```

All fibres are irreducible. According to the theory of rational elliptic surfaces, the geometric Mordell–Weil lattice should then be isomorphic to the root lattice E_8. We can check this. We also compute the Mordell–Weil lattice over the ground field.

```
> Lgeom := GeometricMordellWeilLattice(E);
> IsIsomorphic(Lgeom, Lattice("E", 8));
true
[-1  0 -2  1  1 -1  0 -1]
[ 0  0  0  1  0  0 -1  1]
[ 2  0  2 -2 -1  1  1  0]
[-2  0 -1  0  1  0  0 -1]
[ 1  0  0  1 -1 -1 -1  1]
[-1  0  0  0  1  0  1 -1]
[ 2 -1  2 -3 -1  1  1  1]
[ 1 -1  1 -1  0  0  0  0]
> L, f := MordellWeilLattice(E);
> L;
Standard Lattice of rank 1 and degree 1
Inner Product Matrix:
[8]
> f(L.1);
((1057*t^8 + 384*t^7 + 351*t^6 + 728*t^5 + 872*t^4 + 948*t^3 + 1473*t^2 + 257*t
    + 1333)/(t^6 + 100*t^5 + 1565*t^4 + 1145*t^3 + 927*t^2 + 1302*t + 1197) :
    (1202*t^12 + 1506*t^11 + 1718*t^10 + 1365*t^9 + 656*t^8 + 325*t^7
    + 1173*t^6    + 902*t^5 + 1555*t^4 + 978*t^3 + 616*t^2 + 779*t +
    964)/(t^9 + 150*t^8 + 1520*t^7 + 747*t^6 + 1065*t^5 + 340*t^4 +
    1618*t^3 + 1669*t^2 + 1150*t + 1768) : 1)
```

That the rank equals 1 is not surprising; this point corresponds to the $\mathbb{F}_{1831}$-rational degree 8 divisor consisting of the sum of the 8 points we started with.

To determine the L-function of E one would normally have to count points over various extension fields of $\mathbb{F}_{1831}$, in this case up to $\mathbb{F}_{1831^4}$. This would be costly using current techniques. But since Magma is able to determine the geometric Mordell–Weil lattice, it can compute the L-function by simply considering the Galois action on points. Magma automatically uses this when asked for the L-function:

```
> LFunction(E);
-126330075128099803866555841*T^8 + 1
```

Example: Complex Rank by Reduction Modulo p (ex-198933)

In this example we determine the $\mathbb{C}(t)$-rank of an elliptic curve, following [Klo07].

```
> K<t> := FunctionField(Rationals());
> E := EllipticCurve([-(2*t-1)^3*(4*t-1)^2, t*(2*t-1)^3*(4*t-1)^3]);
```

To determine where the surface has bad reduction we determine the primes at which singular fibres collapse.

```
> &*BadPlaces(E);
(t^5 - 99/32*t^4 + 337/128*t^3 - 251/256*t^2 + 3/16*t - 1/64)/t
> Discriminant(Numerator($1));
-87887055375/4503599627370496
> Factorisation(Numerator($1));
```

(continues on next page)

(continued from previous page)

```
[ <3, 15>, <5, 3>, <7, 2> ]
> Factorisation(Denominator($2));
[ <2, 52> ]
```

So 11 and 13 are the smallest primes of good reduction. We remark that in [Klo07] the primes 17 and 19 were used.

```
> K11<t11> := FunctionField(GF(11));
> E11 := ChangeRing(E,K11);  // Reduce E mod 11
> LFunction(E11);
161051*T^5 - 7986*T^4 - 363*T^3 - 33*T^2 - 6*T + 1
> AnalyticInformation(E11);
<0, 1, 1>
> AnalyticInformation(E11, 2);
<1, 1, 35/2>
```

So modulo 11 the rank is 0, but over $\mathbb{F}_{11^2}(t)$ the rank equals the geometric rank of 1, and the height of a generator is congruent to $\frac{35}{2}$ modulo $\mathbb{Q}^2$. From this it can be concluded that the geometric rank in characteristic 0 is at most 1. As the L-function has odd degree 5 for any p of good reduction, it will always have a zero at $1/p$ or $-1/p$ and consequently the geometric rank modulo p will always be at least 1. To determine what the geometric rank is in characteristic zero, one can combine information at 2 different primes.

```
> K13<t13> := FunctionField(GF(13));
> E13 := ChangeRing(E, K13);  // Redude E mod 13
> AnalyticInformation(E13);
<0, 1, 1>
> AnalyticInformation(E13,2);
<1, 1, 121/2>
```

So over $\mathbb{F}_{13^2}(t)$ the rank equals the geometric rank of 1. The height of a generator is congruent to $\frac{121}{2}$ modulo $\mathbb{Q}^2$. As the quotient of the heights of generators in characteristics 11 and 13 is not a square in $\mathbb{Q}$, there cannot exist a Mordell–Weil group in characteristic zero that both modulo 11 and 13 reduces to a finite index subgroup of the Mordell–Weil group modulo p. Hence one can conclude that the geometric Mordell–Weil rank in characteristic zero is zero.

Example: Partial L-Function by Hand (ex-98b1c7)

In this example we calculate part of the L-function of an elliptic curve for which it is not feasible to compute the L-function completely. The simplest way to compute an L-function is to call `LFunction`, which counts points on the surface over certain constant field extensions. However, if the required extension fields are too big then `LFunction` will not terminate. One can determine the required extension degrees as follows.

```
> K<t> := FunctionField(GF(5));
> E := EllipticCurve([t^9+t^2, t^14+t^4+t^3]);
> h2 := BettiNumber(E, 2);
> N := FrobeniusActionOnTrivialLattice(E);
> [h2, h2 - NumberOfRows(N)];
[ 34, 21 ]
```

So the H^2 has dimension 34, of which a 13-dimensional piece is generated by the trivial lattice and a 21-dimensional piece is unknown. To determine the L-function one would have to at least count points over $\mathbb{F}_{5^{10}}$. As this is not feasible we will instead count points over extension degrees up to 5 and print the first few coefficients of the L-function.

```
> nop := NumbersOfPointsOnSurface(E, 5);
> traces := [ nop[i] - 1 - 25^i - 5^i*Trace(N^i) : i in [1..5] ];
```

(continues on next page)

(continued from previous page)

```
> CharacteristicPolynomialFromTraces(traces);
5750*T^5 + 875*T^4 - 40*T^3 - 40*T^2 + 1
```

134.11 Bibliography

- [Klo07] Remke Kloosterman. Elliptic K3 surfaces with geometric Mordell-Weil rank 15. *Canadian Mathematical Bulletin*, 50(2):215–226, 2007.
- [Kra77] K. Kramer. Two-Descent for Elliptic Curves in Characteristic Two. *Trans. Amer. Math. Soc.*, 232:279–295, 1977.
- [Rob07] David Roberts. *Explicit Descent On Elliptic Curves Over Function Fields*. PhD thesis, University of Nottingham, 2007.

Chapter 135

Models of Genus One Curves

135.1 Introduction

This chapter deals with curves of genus one that are given by equations in a particular form. Most of the functionality involves invariant theory of these models and applications of this theory to arithmetic problems concerning genus one curves over number fields.

Geometrically (viewed over an algebraically closed field), a genus one model of degree n is an elliptic curve embedded in $\mathbb{P}^{n-1}$ via the linear system $|n.O|$. In general, a genus one model of degree n is a principal homogeneous space for an elliptic curve (of order n in the Weil-Châtelet group) which is embedded in $\mathbb{P}^{n-1}$ in an analogous way. Such models are sometimes called genus one normal curves. Not every element of order n in the Weil-Châtelet group admits such an embedding, although it does if it is everywhere locally soluble.

Genus one models may be defined in Magma over any ring. The degree n can be 2, 3, 4, or 5. Genus one models have their own type in Magma: `ModelG1`, which is not a subtype of any other type. In particular, these objects are not curves or even schemes. The data that defines a genus one model is one of: a multivariate polynomial (for degree 2 or 3), a pair of multivariate polynomials (degree 4), or a matrix of linear forms (degree 5). Each of these are now described in detail.

A genus one model of degree 2 in Magma is defined either by a binary quartic $g(x, z)$ (referred to as a model without cross terms), or more generally by an equation $y^2 + f(x, z)y - g(x, z)$ where f and g are homogeneous of degrees 2 and 4 and the variables x, z, y are assigned weights 1, 1, 2 respectively. A binary quartic $g(x, z)$ defines the same model as the equation $y^2 - g(x, z)$. (The implicit map is the projection to $\mathbb{P}^1_{x,z}$, which in this case is not an embedding but rather has degree 2.)

A genus one model of degree 3 in Magma is defined by a cubic form in 3 variables (in other words, the equation of a projective plane cubic curve).

A genus one model of degree 4 in Magma is defined by a sequence of two homogeneous polynomials of degree 2 in 4 variables. This represents an intersection of two quadric forms in $\mathbb{P}^3$, and is the standard form in which Magma returns curves that are obtained by doing `FourDescent` on an elliptic curve.

A genus one model of degree 5 in Magma is defined by a 5×5 alternating matrix whose entries are linear forms in 5 variables. The associated subscheme of $\mathbb{P}^4$ is cut out by the 4×4 Pfaffians of the matrix. It is known that every genus one normal curve of degree 5 arises in this way.

Note: Degenerate cases are allowed, which means that the scheme associated to a genus one model is not always a smooth curve of genus 1 (or even a curve).

Some related functionality, dealing with models of degree 3, is described in Section *Three-Descent and Five-Descent*.

135.2 Creation of Genus One Models

`GenusOneModel(seq): [ RngElt ] -> ModelG1`

`GenusOneModel(n, seq): RngIntElt, [ RngElt ] -> ModelG1`

`GenusOneModel(R, n, seq): RngIntElt, [ RngElt ] -> ModelG1`

`GenusOneModel(n, str): RngIntElt, MonStgElt -> ModelG1`

The genus one model of degree n (where n is 2, 3, 4, or 5) determined by the coefficients in the given sequence or string. The coefficients may belong to any ring.

A sequence $[a, b, c, d, e]$ of length 5 is interpreted as the degree 2 model $ax^4+bx^3z+cx^2z^2+dxz^3+ez^4$. A sequence $[f, g, h, a, b, c, d, e]$ of length 8 is interpreted as the degree 2 model $y^2 + y(fx^2 + gxz + hz^2) - (ax^4 + bx^3z + cx^2z^2 + dxz^3 + ez^4)$.

A sequence $[a, b, c, d, e, f, g, h, i, j]$ of length 10 is interpreted as the degree 3 model $ax^3 + by^3 + cz^3 + dx^2y + ex^2z + fy^2x + gy^2z + hz^2x + iz^2y + jxyz$.

Sequences of lengths 20 or 50 are interpreted as models of degree 4 or 5 respectively; however, it is easier to create these by specifying matrices instead (see below).

The sequence of coefficients can be recovered by calling `Eltseq`.

`GenusOneModel(C): Crv -> ModelG1`

A genus one model that represents the given curve C.

For degree 2, C should either be a subscheme of a weighted projective space $\mathbb{P}(1, 1, 2)$, or a hyperelliptic curve. For degrees $n = 3$, 4, or 5, C should be a genus one normal curve of degree n; in other words, a plane cubic for $n = 3$, an intersection of two quadrics in $\mathbb{P}^3$ for $n = 4$, or an intersection of five quadrics in $\mathbb{P}^4$ for $n = 5$.

`GenusOneModel(f): RngMPolElt -> ModelG1`

`GenusOneModel(f): RngUPolElt -> ModelG1`

`GenusOneModel(seq): [ RngMPolElt ] -> ModelG1`

The genus one model given by the polynomial f or the sequence of equations `seq`.

`GenusOneModel(n, E): RngIntElt, CrvEll -> ModelG1, Crv, MapSch, MapSch`

A genus one model of degree n (where n is 2, 3, 4, or 5) representing the elliptic curve E embedded in $\mathbb{P}^{n-1}$ via the linear system $|n.O|$. Also returned are the image of the embedding as a curve C together with the maps of schemes $E \to C$ and $C \to E$.

`GenusOneModel(mat): Mtrx -> ModelG1`

The genus one model of degree 5 associated to the given 5×5 matrix.

`GenusOneModel(mats): [ AlgMatElt ] -> ModelG1`

The genus one model of degree 4 determined by the given pair of 4×4 symmetric matrices in the sequence *mats*. (The matrices can be recovered by calling `ModelToMatrices`).

`IsGenusOneModel(f): RngUPolElt -> BoolElt, ModelG1`

`IsGenusOneModel(f): RngMPolElt -> BoolElt, ModelG1`

`IsGenusOneModel(seq): [ RngMPolElt ] -> BoolElt, ModelG1`

`IsGenusOneModel(mat): Mtrx -> BoolElt, ModelG1`

Returns `true` if and only if the given polynomial, sequence of polynomials, or matrix determines a "genus one model" in the sense described in the introduction to this chapter. When true, the model is also returned.

Important note: This does *not* imply that the associated scheme is a curve of genus 1. Degenerate models are allowed.

`GenericModel(n): RngIntElt -> ModelG1`

The generic genus one model of degree n, where n is 2, 3, 4, or 5. The coefficients are indeterminates in a suitable polynomial ring.

RandomGenusOneModel(n): RngIntElt -> ModelG1

RandomModel(n): RngIntElt -> ModelG1

Size: RngIntElt	Default:

A random genus one model of degree n, where n is 2, 3, 4, or 5. The optional parameter **Size** is passed to **RandomSL** or **RandomGL**.

ChangeRing(model, R): ModelG1, Rng -> ModelG1

The genus one model defined over the ring R obtained by coercing the coefficients of the given genus one model into R.

CompleteTheSquare(model): ModelG1 -> ModelG1

Given a genus one model of degree 2, returns a simplified genus one model of degree 2 without cross terms; this is computed by completing the square on the multivariate polynomial defining the original model.

CubicFromPoint(E, P): CrvEll, PtEll -> RngMPolElt, MapSch, Pt

The 3-covering corresponding to the rational point P on an elliptic curve E. The 3-covering is returned as the equation of a projective plane cubic curve. Also returned are the covering map and a point that maps to P under the covering map.

HesseModel(n, seq): RngIntElt, [RngElt] -> ModelG1

A genus one model of degree n invariant under the standard representation of the Heisenberg group. The second argument should be a sequence of two ring elements.

DiagonalModel(n, seq): RngIntElt, [RngElt] -> ModelG1

A genus one model of degree n invariant under the diagonal action of μ_n. The second argument should be a sequence of n ring elements.

Example: Generic Genus One Model of Degree 5 (ex-bc04b8)

We construct the genus one model of degree 5 obtained from the generic elliptic curve $E_{a,b} : y^2 = x^3 + ax + b$ over $\mathbb{Q}(a, b)$. The model is the image of $E_{a,b}$ under the embedding in $\mathbb{P}^4$ given by the linear system $|5.O|$.

```
> K<a,b> := FunctionField(Rationals(), 2);
> Eab := EllipticCurve([a, b]);
> model := GenusOneModel(5, Eab);
> model;
[           0 -b*x1 - a*x2           x5           x4           x3]
[ b*x1 + a*x2            0           x4           x3           x2]
[         -x5          -x4            0          -x2            0]
[         -x4          -x3           x2            0           x1]
[         -x3          -x2            0          -x1            0]
```

From this matrix, which is the data storing the model, the equations of the curve in $\mathbb{P}^4$ can be computed; they are quadratic forms given by the 4×4 Pfaffians of the matrix.

```
> Equations(model);
[
    -x1*x4 + x2^2,
    x1*x5 - x2*x3,
    b*x1^2 + a*x1*x2 + x2*x4 - x3^2,
    -x2*x5 + x3*x4,
    -b*x1*x2 - a*x2^2 + x3*x5 - x4^2
]
```

Note that the degree 5 model has the same invariants $c4, c6, \Delta$ as $E_{a,b}$:

```
> Invariants(model);
-48*a
-864*b
-64*a^3 - 432*b^2
> cInvariants(Eab), Discriminant(Eab);
[
    -48*a,
    -864*b
]
-64*a^3 - 432*b^2
```

135.3 Attributes of Genus One Models

Degree(model): ModelG1 -> RngIntElt

The degree (2, 3, 4, or 5) of the given model.

DefiningEquations(model): ModelG1 -> [RngMPolElt]

A sequence containing the equations by which the given genus one model (which must have degree 2, 3, or 4) is defined.

Equations(model): ModelG1 -> [RngMPolElt]

A sequence containing equations for the scheme associated to the given genus one model (of any degree). For degree 2, 3, or 4 this is the same as **DefiningEquations**.

Matrix(model): ModelG1 -> Mtrx

The defining matrix of a genus one model of degree 5.

Curve(model): ModelG1 -> Crv

HyperellipticCurve(model): ModelG1 -> CrvHyp

QuadricIntersection(model): ModelG1 -> Crv

The curve associated to the given genus one model.

In the degree 2 case the curve is hyperelliptic of the form $y^2 + f(x,z)y - g(x,z) = 0$, but is only created explicitly as a hyperelliptic curve when **HyperellipticCurve** is called; otherwise it is created as a general curve in a weighted projective space in which the variables x, z, and y have weights 1, 1, and 2).

In the degree 4 case the curve is an intersection of two quadrics in $\mathbb{P}^3$.

An error results in degenerate cases where the equations of the model do not define a curve

Matrices(model): ModelG1 -> [AlgMatElt]

For a genus one model of degree 4 this function returns a sequence containing two 4×4 symmetric matrices representing the quadrics.

BaseRing(model): ModelG1 -> Rng

The coefficient ring of the given model.

PolynomialRing(model): ModelG1 -> RngMPol

The polynomial ring used to define the model.

ModelToSequence(model): ModelG1 -> [RngElt]

Eltseq(model): ModelG1 -> [RngElt]

A sequence containing the defining coefficients of the given genus one model.

ModelToString(model): ModelG1 -> MonStgElt

A string containing the defining coefficients of the given genus one model.

135.4 Transformations between Genus One Models

A transformation between two genus one models of degree $n = 2$, 3, 4, or 5 is an isomorphism of the underlying data which preserves some additional structure.

Transformations of genus one models have their own type `TransG1`. A transformation can be applied to a model using `'*'`. The syntax for creating transformations, and for extracting data from them, is described in this section.

For degree 2 models $q(x, z)$ without cross terms, or for degree 3 models, a transformation is a tuple $\langle k, S\rangle$, where k is an element of the coefficient ring; the transformed model is obtained by making the substitution of coordinate variables determined by S and multiplying the equation by k. For degree 2 models with cross terms a transformation is a tuple $\langle k, [A, B, C], S\rangle$, where k and S are as above and in addition $y + Ax^2 + Bxz + Cz^2$ is substituted for y in the transformation. For degree 4, the first element is a 2×2 matrix; a model of degree 4 is given by two quadric equations, and the 2×2 matrix determines a change of basis of the quadrics (acting on them from the left). A transformation of degree 5 models is a tuple $\langle T, S\rangle$ where T and S are both 5×5 matrices; a model of degree 5 is given by a 5×5 matrix of linear forms M, and the transformed model is given by TM_ST^{tr} where M_S is obtained from M by making the substitution of coordinate variables specified by S.

Two genus one models are said to be equivalent if they differ by such a transformation. Equivalent models have the same invariants up to scaling by the 4th and 6th powers of some element.

`IsTransformation(n, g): RngIntElt, Tup -> BoolElt, TransG1`

This returns `true` if and only if the tuple g represents a transformation of genus one models of degree n. If so, it also returns the transformation as an object of type `TransG1`.

`Tuple(g): TransG1 -> Tup`

This returns a tuple containing the data that defines the given transformation of genus one models.

`ChangeRing(g, R): TransG1, Rng -> TransG1`

For a transformation g of genus one models, this constructs the same transformation over the ring R.

`IdentityTransformation(n, R): RngIntElt, Rng -> TransG1`

The identity transformation of genus one models of degree n.

`RandomTransformation(n : parameters): RngIntElt -> TransG1`

```
Size      : RngIntElt                    Default: 5
Unimodular: BoolElt                      Default: false
CrossTerms: BoolElt                      Default: false
```

A random transformation of genus one models of degree n. When `Unimodular` is set to `true` then the returned transformation is integrally invertible.

The optional parameter `Size` is passed to `RandomSL` or `RandomGL`.

In degree 2, if `CrossTerms` is false then the returned transformation preserves the set of models with no cross terms.

`g * model: TransG1, ModelG1 -> ModelG1`

`ApplyTransformation(g, model): TransG1, ModelG1 -> ModelG1`

The result of applying the transformation g to the genus one model.

`g1 * g2: TransG1, TransG1 -> TransG1`

`ComposeTransformations(g1, g2): TransG1, TransG1 -> TransG1`

`MultiplyTransformations(g1, g2): TransG1, TransG1 -> TransG1`

The composition $g1 * g2$ of two transformations of genus one models. Transformations of genus one models act on the left: $(g1 * g2) * model = g1 * (g2 * model)$.

`Inverse(g): TransG1 -> TransG1`

`InverseTransformation(g): TransG1 -> TransG1`

`InverseTransformation(n, g): RngIntElt, TransG1 -> TransG1`

The inverse of the transformation g of genus one models.

`ScalingFactor(g): TransG1 -> RngElt`

`ScalingFactor(n, g): RngIntElt, TransG1 -> RngElt`

The scaling factor of the transformation g of genus one models. This an element λ such that if a genus one model has invariants c_4 and c_6 then the transformed model has invariants $\lambda^4 c_4$ and $\lambda^6 c_6$.

`Determinant(g): TransG1 -> RngElt`

The determinant of the matrix associated with the transformation g.

135.5 Equivalence of Genus One Models

`IsEquivalent(model1, model2): ModelG1, ModelG1 -> BoolElt, TransG1`

`IsEquivalent(cubic1, cubic2): RngMPolElt, RngMPolElt -> BoolElt, TransG1`

Return `true` if and only if the two given cubics (or genus one models of degree 3) are equivalent as genus one models. In other words, that there exists a linear transformation of the ambient projective space $\mathbb{P}^2_{\mathbb{Q}}$ which takes one cubic to the other, up to scaling. The algorithm is given in [Fis06a].

When true, the transformation is also returned as a tuple (the syntax is explained in Section *Transformations between Genus One Models*, on transformations, below).

135.6 Minimisation and Reduction

This section contains functions for obtaining a simpler model of a given genus one curve.

We use the following terminology, which has become fairly standard. Here a model will always mean a global integral model.

Minimisation refers to computing a model which is isomorphic to the original one (over its ground field) but possibly not integrally equivalent. A model is *minimal* if it is locally minimal at all primes; a model is locally minimal if the valuation of its discriminant is as small as possible among all integral models. For locally solvable models this minimal discriminant is equal to that of the associated elliptic curve.

Reduction refers to computing a model which is integrally equivalent to the original one (meaning there exist transformations in both directions over the ring of integers of the base field). Informally, a *reduced model* is one whose coefficients have small height. One approach to formulating this precisely is to attach to each model an *invariant* in some symmetric space: The model is then said to be reduced iff its invariant lies in some fixed fundamental domain. This idea is also the basis for algorithms used to obtain reduced models.

The algorithms used for models of degree 2, 3, and 4 over $\mathbb{Q}$ are described in [CFS10]. For degree 5 it is necessary to use some different techniques developed by Fisher. For models of degree 2 over number fields the algorithms use additional techniques developed by Donnelly and Fisher.

`Minimise(model : parameters): ModelG1 -> ModelG1, TransG1`

```
Transformation  : BoolElt                    Default: true
CrossTerms      : BoolElt                    Default: false
UsePrimes       : SeqEnum                    Default: []
ClassGroupPrimes: SeqEnum                    Default: []
Minimisation    : Verbose                    Default: 3
```

Given a genus one model (of degree 2, 3, 4, or 5) with coefficients in $\mathbb{Q}$, this returns a minimal model in the sense described above.

It is also implemented for genus one models of degree 2 without cross terms, and returns a model that is minimal among models without cross terms when this is possible. When this is not possible

(due to class group obstructions) it returns a model that is nearly minimal, in the sense that the extra factor appearing in the discriminant is chosen to have small norm. The primes dividing the extra factor may be specified via the optional argument `ClassGroupPrimes`.

The transformation taking the original model to the minimal model is also returned, unless the optional argument `Transformation` is set to `false`. (The syntax for transformations is explained in Section *Transformations between Genus One Models*).

For degree 2, when the optional argument `CrossTerms` is set to `false` then a model without cross terms is returned that is minimal among models of this kind.

The third value returned is the set of primes where the minimal model has positive level. These are the primes where the model is not soluble over $\mathbb{Q}_p^{nr}$, the maximal non-ramified extension of $\mathbb{Q}_p$ (except for $p = 2$ for models of degree 2 when `CrossTerms` is `false`).

The degree 5 routine is not yet proven to work in all cases.

`Minimise(f): RngMPolElt -> RngMPolElt, TransG1`

Given the equation f of a nonsingular projective plane cubic curve, this returns the equation of a minimal model of the curve and a tuple specifying the transformation (as explained in Section *Transformations between Genus One Models* below).

`pMinimise(f, p): RngMPolElt, RngIntElt -> RngMPolElt, AlgMatElt`

Given the equation f of a nonsingular projective plane cubic curve, this function returns a model of the curve which is minimal at p. Also returned is a matrix M giving the transformation; up to scaling, the minimised cubic is the original cubic evaluated at $M[x, y, z]^{tr}$.

`Reduce(model): ModelG1 -> ModelG1, TransG1`

`Reduce(f): RngMPolElt -> RngMPolElt, TransG1`

`Reduce: Verbose`	`Default: 3`

Given a genus one model of degree 2, 3, or 4 over $\mathbb{Q}$, this function computes a reduced model in the sense described above.

The model may instead be described by the appropriate polynomial f, which must be either a homogeneous polynomial that defines a genus one model of degree 2 or 3, or a univariate polynomial that defines a genus one model of degree 2 without cross terms.

The transformation taking the original model to the reduced model is also returned. (The syntax for transformations between genus one models is explained in Section *Transformations between Genus One Models*).

`ReduceQuadrics(seq): [RngMPolElt] -> [RngMPolElt], AlgMatElt, AlgMatElt`

`Reduce: Verbose`	`Default: 3`

This function computes a reduced basis for the space spanned by the given quadrics (which should be given as a sequence of homogeneous quadratic forms in variables $x_1, \ldots, x_n$). This means making a linear change of the homogeneous coordinates, and also finding a new basis for the space of forms, with the aim of making the resulting coefficients small. The second and third objects returned are matrices S and T giving the transformation. The change of homogeneous coordinate variables is given by S, while T specifies the change of basis of the space of forms. The returned forms are therefore obtained by substituting $[x_1, \ldots, x_n].S$ for $[x_1, \ldots, x_n]$ in the original forms, and then applying T to this basis of forms (where T acts from the left).

The current implementation of this routine is not optimal; nevertheless it is useful in some situations.

135.7 Local Solubility

Efficient routines are provided for testing local solubility for models of all degrees over $\mathbb{Q}$.

IsLocallySoluble(model, p): ModelG1, RngIntElt -> BoolElt, Any

Transformation:	BoolElt	Default: false
Random :	BoolElt	Default: false
LocSol :	Verbose	Default: 1

This determines whether the given genus one model (which must be defined over $\mathbb{Q}$ and nonsingular) has a $\mathbb{Q}_p$-rational point. When this is the case, then it also returns a $\mathbb{Q}_p$ point.

If the optional argument **Transformation** is **true**, then the second returned value is *not* a point but instead a transformation g such that **g*model** has a smooth point on its reduction modulo p.

If **Random** is **true**, then (for models of some degrees, including 4) the local point returned is not always the same, when the function is called several times with the same input.

IsLocallySoluble(model): ModelG1 -> BoolElt

This determines whether the given genus one model (which must be defined over $\mathbb{Q}$ and nonsingular) has a $\mathbb{Q}_p$-rational point for all primes p.

This is done by calling **IsLocallySoluble(model, p)** for all bad primes p.

135.8 Genus One Models as Coverings

The curve defined by a genus one model of degree n is a principal homogeneous space for some elliptic curve (namely the Jacobian of the curve). The data of the Jacobian, and the covering map of degree n^2, can be read from the invariants and covariants of the model.

Any two models with the same Jacobian can be added together as elements of the Weil-Châtelet group. Below are functions for adding two models of degree 3, and for "doubling" models of degree 4 or 5.

A related function for degree 3 models is **ThreeSelmerElement** (see Section *Three-Descent and Five-Descent*). For degree 4 models the maps can also be computed using **AssociatedEllipticCurve** and **AssociatedHyperellipticCurve** from the package on four descent (see Section *Four-Descent*).

Jacobian(model): ModelG1 -> CrvEll

Jacobian(C): Crv -> CrvEll

The Jacobian, returned as an elliptic curve, of the given genus one model, or of the curve C corresponding to a genus one model.

nCovering(model : parameters): ModelG1 -> Crv, CrvEll, MapSch

E:	CrvEll	Default:

The covering map from the given genus one model to its Jacobian. Three values are returned: the curve C of degree n corresponding to the given model, its Jacobian as an elliptic curve E, and a map of schemes $C \to E$.

If an elliptic curve E is given it must be isomorphic to the Jacobian and then this curve will be the image of the map.

AddCubics(cubic1, cubic2 : parameters): RngMPolElt, RngMPolElt -> RngMPolElt

AddCubics(model1, model2 : parameters): ModelG1, ModelG1 -> ModelG1

model1 + model2: ModelG1, ModelG1 -> ModelG1

E :	CrvEll	Default:
ReturnBoth:	BoolElt	Default: false

Given two ternary cubic polynomials, or two genus one models of degree 3, that both have the same invariants, returns the sum of the corresponding elements of $H^1(\mathbb{Q}, E[3])$.

An error results if the two cubics do not belong to the same elliptic curve E, See Section *Three-Descent and Five-Descent* for more information about `AddCubics`.

`DoubleGenusOneModel(model): ModelG1 -> ModelG1`

Given a genus one model of degree 4 or 5, this function computes twice the associated element in the Weil-Châtelet group and returns this as a genus one model (which will have degree 2 or 5, respectively).

`FourToTwoCovering(model : parameters): ModelG1 -> Crv, Crv, MapSch`
`FourToTwoCovering(C : parameters): Crv -> Crv, Crv, MapSch`

C2: Crv	Default:

Given a genus one model of degree 4 or an associated curve, this function returns three values: the curve C_4 in $\mathbb{P}^3$ corresponding to the model, a plane quartic curve C_2 representing twice the model in the Weil-Châtelet group, and the map of schemes $C_4 \to C_2$. Calling `AssociatedHyperellipticCurve(Curve(model))` provides the same information.

135.9 Families of Elliptic Curves with Prescribed n-Torsion

In [RS95], Rubin and Silverberg explicitly construct families of elliptic curves over $\mathbb{Q}$ which have the same Galois representation on the n-torsion subgroups as a given elliptic curve.

`RubinSilverbergPolynomials(n, J : parameters): RngIntElt, RngElt -> RngElt, RngElt`

Parameter: RngElt	Default:

Suppose that $n = 2$, 3, 4, or 5 and let $E : y^2 = x^3 + ax + b$ be an elliptic curve over the rationals with j-invariant $1728J$. This function returns polynomials $\alpha(t)$ and $\beta(t)$ that determine a family of elliptic curves with fixed n-torsion, in the following sense: Every nonsingular member F_t of the family $F : y^2 = x^3 + a\alpha(t)x + b\beta(t)$ has $F_t[n]$ isomorphic to $E[n]$ as $\mathbb{Z}[G]$-modules, where G is the absolute Galois group of $\mathbb{Q}$, and furthermore the isomorphisms between $F_t[n]$ and $E[n]$ preserve the Weil pairing. When n is 3, 4, or 5, all such "n-congruent" curves belong to the same family.

135.10 Invariants for Genus One Models

`aInvariants(model): ModelG1 -> [ RngElt ]`

The invariants $[a_1, a_2, a_3, a_4, a_6]$ of the given genus one model which must have degree 2, 3, or 4. The formulae in the degree 3 case come from [ARVT05].

`bInvariants(model): ModelG1 -> [ RngElt ]`

The invariants $[b_2, b_4, b_6, b_8]$ of the given genus one model which must have degree 2, 3, or 4. These are computed from the `aInvariants` in the standard way (as for elliptic curves).

`cInvariants(model): ModelG1 -> [ RngElt ]`

The invariants $[c_4, c_6]$ of the given genus one model. For $n = 2$, 3, or 4 these are the classical invariants, as can be found in [AKM+01]. For $n = 5$ the algorithm is described in [Fis08a].

`Invariants(model): ModelG1 -> RngElt, RngElt, RngElt`

The invariants c_4, c_6 and Δ (the discriminant) of the given genus one model.

`Discriminant(model): ModelG1 -> RngElt`

The discriminant Δ of the given genus one model.

`SL4Invariants(model): ModelG1 -> [ RngElt ]`

The SL_4-invariants of a genus one model of degree 4.

135.11 Covariants and Contravariants for Genus One Models

The functions in this section implement the invariant theory developed in [Fis06b].

Hessian(model): ModelG1 -> ModelG1

We write X_n for the (affine) space of genus one models of degree n. The module of covariants $X_n \to X_n$ is a free module of rank 2 over the ring of invariants. The generators are the identity map and a second covariant which we term the Hessian. (In the cases where $n = 2$ or 3 this is the determinant of a matrix of second partial derivatives.) This function evaluates the Hessian of the given genus one model.

CoveringCovariants(model): ModelG1 -> [RngMPolElt]

The covariants that define the covering map from the given genus one model to its Jacobian (this is the same as the defining equations of the nCovering).

Contravariants(model): ModelG1 -> ModelG1, ModelG1

We write X_n for the (affine) space of genus one models of degree n, and X_n^* for its dual. The module of contravariants $X_n \to X_n^*$ is a free module of rank 2 over the ring of invariants. This function evaluates the generators P and Q at the given genus one model.

HesseCovariants(model, r): ModelG1, RngIntElt -> ModelG1, ModelG1

Evaluates a pair of covariants, which depend on an integer r, at the genus one model of degree prime to r. The pencil spanned by these genus one models is a family of genus one curves invariant under the same representation of the Heisenberg group. (In other words, the universal family above a twist of $X(n)$.)

If $r \equiv 1 \pmod{n}$ then the covariants evaluated are the identity map and the Hessian. If $r \equiv -1 \pmod{n}$ then the covariants evaluated are the contravariants. If $n = 5$ there are two further possibilities. We identify $X_5 = \wedge^2 V \otimes W$ where V and W are 5-dimensional vector spaces. Then the covariants evaluated for $r \equiv 2, 3 \pmod{5}$ take values in $\wedge^2 W \otimes V^*$ and $\wedge^2 W^* \otimes V$ respectively.

HessePolynomials(n, r, invariants : parameters): RngIntElt, RngIntElt, [RngElt] -> RngElt, RngElt, RngElt

Variables: [RngMPolElt]	Default:

The Hesse polynomials $D(x, y), c_4(x, y), c_6(x, y)$. These polynomials give the invariants for the pencil of genus one models computed by HesseCovariants. The RubinSilverbergPolynomials are closely related to these formulae in the case $r \equiv 1 \pmod{n}$.

135.12 Examples

Example: A Cubic Violating the Hasse Principle (ex-1d0d2c)

We find a cubic which is a counterexample to the Hasse principle. The approach involves the idea of "visibility" of Tate–Shafarevich elements, which was introduced by Mazur (see [CM00b]).

The cubic will be a nontrivial element of the Tate–Shafarevich group of the curve 4343B1 in Cremona's tables, which we call E. The cubic will be obtained from a rational point on an auxiliary elliptic curve F.

First, we compute that E has rank 0, and F has rank 1:

```
> E := EllipticCurve([ 0, 0, 1, -325259, -71398995 ]);
> F := EllipticCurve([ 1, -1, 1, -24545, 1486216 ]);
> CremonaReference(E);
4343b1
> RankBounds(E);
0 0
```

(continues on next page)

(continued from previous page)

```
> RankBounds(F);
1 1
```

We take a plane cubic representing one of the nontrivial elements in the 3-Selmer group of F, which has order 3, so that its elements are all in the image of $F(\mathbb{Q})$ since $F(\mathbb{Q})$ has rank 1:

```
> SetClassGroupBounds("GRH");
> #ThreeSelmerGroup(F);
3
> coverings := ThreeDescent(F);
> coverings;
[
    Curve over Rational Field defined by
    -3*x^2*z - 3*x*y^2 - 27*x*y*z + 12*x*z^2 + 2*y^3 + 21*y^2*z + 3*y*z^2 - 4*z^3
]
> C := Equation(coverings[1]);
```

We now try to find a linear combination of C and its Hessian (which is also a plane cubic) that has j-invariant equal to the j-invariant of E. To find the right linear combination we may work geometrically (that is, with F instead of C since they are isomorphic over $\overline{\mathbb{Q}}$). We work with the family $tF + H$ where t is an indeterminate.

```
> B<t> := PolynomialRing(Rationals());
> F_BR := ChangeRing(Parent(Equation(F)), B);
> F_B := F_BR ! Equation(F);
> H_B := Hessian(F_B);
> c4,c6,Delta := Invariants(t*F_B + H_B);
```

Alternatively we could get these invariants as follows:

```
> D,c4,c6 := HessePolynomials(3, 1, cInvariants(F) : Variables := [t, 1] );
> Delta := Discriminant(F)*D^3;
// Solve c4(t)^3/Delta(t) = j(E) for t:
> jpoly := c4^3 - jInvariant(E)*Delta;
> Roots(jpoly);
[ <7479/7, 1> ]
```

So we take the following linear combination (and replace the equation by a nicer equation):

```
> C2raw := 7479/7*C + Hessian(C);
> C2 := Reduce(Minimise(C2raw));
> C2;
7*x^3 + 7*x^2*y + 3*x^2*z - 4*x*y^2 - 30*x*y*z + 12*x*z^2 - 13*y^3 - 2*y^2*z -
    15*y*z^2 - 17*z^3
```

The Jacobian of $C2$ is E, so $C2$ is a principal homogeneous space for E of index 3, and in fact it is everywhere locally soluble:

```
> IsIsomorphic(Jacobian(C2), E);
true
> PrimeDivisors(Integers()!Discriminant(GenusOneModel(C2)));
[ 43, 101 ]
> C2_crv := Curve(ProjectiveSpace(Parent(C2)), C2);
> IsLocallySolvable(C2_crv, 43);
true (7 + O(43) : 1 + O(43^50) : O(43))
> IsLocallySolvable(C2_crv, 101);
true (1 + O(101^50) : 32 + O(101) : 1 + O(101))
```

(continues on next page)

(continued from previous page)

```
// Find the preimage of the covering map C2 -> E:
> _, _, maptoE := nCovering(GenusOneModel(C2) : E := E);
> preimage := Pullback(maptoE, E!0);
> Points(preimage);   // Q-rational points
{@ @}
> TorsionSubgroup(E);
Abelian Group of order 1
```

We conclude that $C2$ has no rational points since, as $E(\mathbb{Q})$ is trivial, any rational points on $C2$ must map to O_E.

135.13 Bibliography

- [AKM+01] Sang Yook An, Seog Young Kim, David C. Marshall, Susan H. Marshall, William G. McCallum, and Alexander R. Perlis. Jacobians of genus one curves. *J. Number Theory*, 90(2):304–315, 2001.
- [ARVT05] Michael Artin, Fernando Rodriguez-Villegas, and John Tate. On the Jacobians of plane cubics. *Adv. Math.*, 198(1):366–382, 2005.
- [CFS10] J.E. Cremona, T.A Fisher, and M Stoll. Minimisation and reduction of 2-, 3- and 4-coverings of elliptic curves. *Algebra & Number Theory*, 4(6):763–820, 2010.
- [CM00b] John E. Cremona and Barry Mazur. Visualizing elements in the Shafarevich-Tate group. *Experiment. Math.*, 9(1):13–28, 2000.
- [Fis08a] T.A Fisher. The invariants of a genus one curve. *Proc. Lond. Math. Soc.*, 97(3):753–782, 2008.
- [Fis06a] T.A. Fisher. Testing equivalence of ternary cubics. In S. Pauli F. Hess and M.Pohst, editors, *ANTS VII*, volume 4076 of LNCS, 333–345. Springer-Verlag, 2006.
- [Fis06b] Tom Fisher. The Hessian of a genus one curve. preprint arXiv:math/0610403v2 [math.NT], 2006.
- [RS95] K. Rubin and A. Silverberg. Families of elliptic curves with constant mod p representations. In *Elliptic curves, modular forms, & Fermat's last theorem (Hong Kong, 1993)*, Ser. Number Theory, I, pages 148–161. Internat. Press, Cambridge, MA, 1995.

Chapter 136

Plane Quartics

136.1 Invariants

Isomorphisms of plane smooth quartics over an algebraically closed field k are induced by linear transformations of the ambient projective plane. Therefore, isomorphism classes are characterized by the weighted projective space associated to the ring of invariants of ternary quartic forms under the classical action of $\mathrm{SL}_3(k)$. When k is of characteristic 0, Dixmier [Dix87] gave a list of 7 invariants which form a homogeneous system of parameters. It was completed by [Ohn07], who furnished a list of 13 generators for the algebra $R(\mathbb{C})$.

The Dixmier-Ohno invariants are polynomials in the 15 coefficients of a ternary quartic form with coefficients in $\mathbb{Z}[1/6]$. They can be considered as a point in the weighted projective space with weights $(3, 6, 9, 9, 12, 12, 15, 15, 18, 18, 21, 21, 27)$. A list of generators of the invariants of smooth plane quartics in positive characteristic is not known, although it is suspected that the reduction of the Dixmier–Ohno invariants are generators when the characteristic is greater than 7. In [LLGR20] homogeneous systems of parameters are determined in all characteristics except 3, for which there is a conjectural HSOP that involves an invariant of degree 81. A call to `DixmierOhnoInvariants()` in general characteristic outputs a minimal set of invariants that generate the largest subring of invariants that is known so far.

Among the Dixmier–Ohno invariants of a form $f(x, y, z)$, the invariant I_{27} of degree 27 plays a particular role. It can be shown that $\frac{1}{2^{40}I_{27}}$ has integral coefficients: it defines the so-called discriminant of ternary quartic forms. Over any field, the zero locus of the discriminant is precisely the set of ternary quartic forms that define a singular plane quartic. The current calculation of this discriminant is based on the techniques developed in [BJ14] (Def.4.6, Prop.4.7).

The reader may also be interested in [GK06].

`DixmierOhnoInvariants(C): Crv -> SeqEnum, SeqEnum`

`DixmierOhnoInvariants(f): RngMPolElt -> SeqEnum, SeqEnum`

```
normalize            : BoolElt                    Default: false
IntegralNormalization: BoolElt                    Default: false
PrimaryOnly          : BoolElt                    Default: false
degmin               : RngIntElt                  Default: 1
degmax               : RngIntElt                  Default: Infinity()
PolynomialOnly       : BoolElt                    Default: true
```

Compute the Dixmier-Ohno invariants of a ternary quartic form f. When the characteristic of the coefficient ring is 0 or greater than 7, the returned invariants are 'I3', 'I6', 'I9', 'J9', 'I12', 'J12', 'I15', 'J15', 'I18', 'J18', 'I21', 'J21', and 'I27'. Weights of these invariants are returned as well. If `normalize` is set to `true`, then the invariants are normalized in the corresponding weighted projective space before being returned.

Setting `IntegralNormalization` to `true` multiplies the Dixmier-Ohno invariants by certain constants so that the invariants (as polynomials in the coefficients) are defined over $\mathbb{Z}$. Using the flags

`degmin` and `degmax` provides only a partial list of generators in the corresponding degrees. Setting the flag `PolynomialOnly` to `false` (only relevant in characteristic 3) provides additional invariants that come from non-integral expressions in the Dixmier–Ohno invariants.

Example: Dixmier-Ohno Invariants of the Klein Quartic (ex-11474d)

We calculate the Dixmier-Ohno invariants of the Klein quartic and one of its non-trivial twists over $\mathbb{Q}$.

```
> P<x,y,z> := PolynomialRing(Rationals(), 3);
> PP := ProjectiveSpace(P);
> f1 := x^3*y + y^3*z + z^3*x;
> f2 := x^4 + 7*x^3*z + 3*x^2*y^2 - 3*x^2*z^2 - 6*x*y*z^2 - 5*x*z^3 +
> 2*y^3*z + 3*y^2*z^2 + 2*y*z^3 - 4*z^4;
> C1 := Curve(PP, f1); DO1 := DixmierOhnoInvariants(C1 : normalize := true);
> C2 := Curve(PP, f2); DO2 := DixmierOhnoInvariants(C2 : normalize := true);
> DO1 eq DO2;
true
> IsIsomorphicPlaneQuartics(C1, C2);
false []
```

`DiscriminantOfTernaryQuartic(f): RngMPolElt -> Any`

Compute the discriminant of the ternary quartic form f.

Example: Discriminant of the Klein Quartic (ex-a1e8bd)

We compute the discriminant of the Klein Quartic over $\mathbb{F}_2$.

```
> P<x,y,z> := PolynomialRing(GF(2), 3);
> Q := x^3*y + y^3*z + z^3*x;
> DiscriminantOfTernaryQuartic(Q);
1
```

`DiscriminantFromDixmierOhnoInvariants(DO): SeqEnum -> Any`

Compute the discriminant of a ternary quartic form from the given Dixmier-Ohno invariants DO.

`DixmierOhnoInvariantsEqual(DO1, DO2): SeqEnum, SeqEnum -> BoolElt`

Check whether Dixmier-Ohno Invariants $DO1$ and $DO2$ of two quartics are equivalent.

`DixmierOhnoAlgebraicRelations(DOinv): SeqEnum -> SeqEnum`

Return generators of the ideal of relations between the Dixmier-Ohno invariants.

`CovariantHessian(Phi): RngMPolElt -> RngMPolElt`

Compute the Hessian covariant of the ternary quartic form Φ.

`ContravariantSigmaAndPsi(Phi): RngMPolElt -> RngMPolElt, RngMPolElt`

Compute the covariants Σ and Ψ of a ternary quartic form Φ, as defined in [Sal79, p. 78].

`QuarticCovariantsAndContravariants(Phi): RngMPolElt -> SeqEnum`

Computes generators of the covariant and contravariant algebra of the ternary quartic form Φ.

136.2 Creation from Invariants

Given the Dixmier-Ohno invariants of a generic plane smooth quartic over a field k of characteristic 0, the algorithms developed in [LRS16] allow the reconstruction of a model of this quartic, which is returned over k itself as long as the geometric automorphism group of C is not of order 2. The main function to this end is `PlaneQuarticFromDixmierOhnoInvariants`. It is proven to work only in characteristic 0 although it is likely to work in characteristic ≥ 11 as well (and one can check if the result is correct anyway).

In the above, "generic" means concretely that the Dixmier-Ohno invariant I_{12} is different from 0. If I_{12} is zero, other systems of co- or contra-variants may be chosen to perform the reconstruction. These

variants have not been implemented, and there are smooth plane quartics, like the Klein quartic, for which no such system exists. Regardless, for all non-trivial automorphism strata except for the cyclic group $\mathbb{Z}/2\mathbb{Z}$, as well as for $(\mathbb{Z}/2\mathbb{Z})^2$ in case $I_{12} = 0$, an *ad hoc* reconstruction is performed.

If the quartic curve has automorphism group of order 2, the field of moduli is not necessarily a field of definition and the reconstruction may happen over a quadratic extension only. Still, the algorithms will in practice often find a model over the field of moduli if it exists.

For more details, see [Els15, LRS18].

PlaneQuarticFromDixmierOhnoInvariants(DO): SeqEnum -> Crv, SeqEnum

TernaryQuarticFromDixmierOhnoInvariants(DO): SeqEnum -> RngMPolElt, SeqEnum

```
exact       : BoolElt                   Default: false
minimize    : BoolElt                   Default: true
descent     : BoolElt                   Default: true
search_point: BoolElt                   Default: true
```

Reconstructs a plane quartic from a given tuple of Dixmier-Ohno invariants *DO*.

If the flag `exact` is set to `true`, then a ternary forms is returned whose Dixmier-Ohno invariants exactly equal DO (instead of merely being equal in the corresponding weighted projective space).

If the flag `descent` is set to `true`, then the curve is descended to its base field.

If the flag `minimize` is set to `true`, then over the rationals an effort is made to return as small a model as possible.

If the flag `search_point` is set to `true`, then the algorithm tries to find a rational point on the Mestre conic of the associated binary form. This is required when reconstructing over the base field.

Example: Quartic from Dixmier-Ohno Invariants (ex-dbee6a)

We reconstruct a plane quartic from its invariants.

```
> P<x,y,z> := PolynomialRing(GF(31), 3);
> PP := ProjectiveSpace(P);
> f1 := x^4 + 3*y^4 + 5*z^4 + x^2*y*z + x*y*z^2 + x^2*y^2;
> C1 := Curve(PP, f1);
> I := DixmierOhnoInvariants(f1);
> C2 := Curve(PP, TernaryQuarticFromDixmierOhnoInvariants(I));
> IsIsomorphicPlaneQuartics(C1, C2);
true [
    [ 1 24  8]
    [ 8 27 20]
    [13 20 19]
]
```

136.3 Isomorphisms

Isomorphisms and automorphisms of plane quartics have been implemented following the covariant method due to van Rijnswou [vR01] that is also used in another form in the reconstruction algorithms. Let C_1 and C_2 be two plane quartic curves over a field k. The present algorithm first checks for equality of normalized Dixmier–Ohno invariants of C_1 and C_2, since if this equality does not hold, no isomorphisms can exist.

If this condition is satisfied, the algorithms first try to find the actual isomorphisms $C_1 \to C_2$ under the assumption that $I_{12} \neq 0$. In this case, [vR01] shows that the use of a suitable covariant reduces this question to finding transformations between certain binary forms associated to C_1 and C_2, which leads us to the same computation of elements in $\mathrm{GL}_2(k)$. In non-generic cases, a direct Gröbner basis method due to Michael Stoll is used.

Once again the algorithms admit both a version over the base field and a geometric version, with the latter finding the isomorphisms over the algebraic closure of k. Both versions are very efficient over finite fields, and the version over the base field is also reasonably fast for $k = \mathbb{Q}$. By contrast, finding geometric isomorphisms between plane quartic curves over the rationals can still take a fair amount of time. For more general fields, the implementation still takes too long, and our functions therefore restrict considerations to the cases where k is either finite or the rational field.

IsIsomorphicPlaneQuartics(X1, X2): CrvPln, CrvPln -> BoolElt, SeqEnum

IsIsomorphicTernaryQuartics(f1, f2): RngMPolElt, RngMPolElt -> BoolElt, SeqEnum

geometric: BoolElt	Default: false

Determine if the plane quartic curves X_1 and X_2 or the ternary quartics f_1 and f_2 are isomorphic, and return the isomorphism between them. If the flag `geometric` is set to `true`, then test is performed over the algebraic closure of the base field, over which the isomorphisms are then determined as well.

IsomorphismsOfPlaneQuartics(X1, X2): CrvPln, CrvPln -> SeqEnum

IsomorphismsOfTernaryQuartics(f1, f2): RngMPolElt, RngMPolElt -> SeqEnum

geometric: BoolElt	Default: false

Return the isomorphisms between the plane quartic curves X_1 and X_2 or between then ternary quartic forms f_1 and f_2. If the flag `geometric` is set to `true`, then the isomorphisms over the algebraic closure of the base field are returned.

AutomorphismsOfPlaneQuartic(X): CrvPln -> SeqEnum

AutomorphismsOfTernaryQuartic(f): RngMPolElt -> SeqEnum

geometric: BoolElt	Default: false

Return the automorphisms of the plane quartic curve X or the ternary quartic form f as matrices. If the flag `geometric` is set to `true`, then the automorphisms over the algebraic closure of the base field are returned.

AutomorphismGroupOfPlaneQuartic(X): CrvPln -> GrpPerm, Map

AutomorphismGroupOfTernaryQuartic(f): RngMPolElt -> GrpPerm, Map

geometric: BoolElt	Default: false
explicit : BoolElt	Default: false

Return the automorphisms of the plane quartic curve X or the ternary quartic form f as an abstract group, along with a map from said group to a matrix group if the flag `explicit` is set to `true`. If the flag `geometric` is set to `true`, then the automorphisms over the algebraic closure of the base field are returned.

Example: Automorphisms of the Klein Quartic (ex-153f49)

We determine the automorphisms of a plane quartic over the rationals.

```
> P<x,y,z> := PolynomialRing(Rationals(), 3);
> PP := ProjectiveSpace(P);
> C := Curve(PP, x^3*y+y^3*z+z^3*x);
> aut, phi := AutomorphismGroupOfPlaneQuartic(C : geometric:=true,
>                  explicit := true);
> aut;
Permutation group aut acting on a set of cardinality 8
Order = 168 = 2^3 * 3 * 7
    (2, 3, 4)(5, 8, 7)
```

(continues on next page)

(continued from previous page)

```
    (2, 4, 5)(3, 6, 7)
    (1, 2)(3, 7)(4, 5)(6, 8)
> GroupName(aut);
PSL(2,7)
> [phi(aut.i) : i in [1..Ngens(aut)]];
[
    [0 0 1]
    [1 0 0]
    [0 1 0],

    [0 0 -r1^5 - r1^4 - r1^3 - r1^2 - r1 - 1]
    [1 0 0]
    [0 r1^4 0],

    [1 r1^2 + r1 -r1^5 - r1^4 - r1^3]
    [-r1^4 - r1^3 - r1^2 - r1 - 1 r1^5 + r1^4 + r1^3 + r1^2 -r1^5 - r1^4 - r1^3
        - r1^2 - r1 - 1]
    [-r1^4 - r1^3 - r1^2 r1 r1^4 + r1^3]
]
```

AutomorphismGroupOfPlaneQuartic(X, Autos): CrvPln, SeqEnum -> GrpPerm, Map

AutomorphismGroupOfTernaryQuartic(f, Autos): RngMPolElt, SeqEnum -> GrpPerm, Map

```
explicit: BoolElt                    Default: false
```

Return the automorphisms Autos of the plane quartic curve X or the ternary quartic form f as an abstract group, along with a map from said group to a matrix group if the flag `explicit` is set to `true`.

GeometricAutomorphismGroup(C): Crv -> GrpPerm

Compute the geometric automorphism group of the elliptic, hyperelliptic, or plane quartic curve C.

136.4 Twists

Using classical reductions to compute the cohomology set $H^1(\mathrm{Gal}(\overline{k}/k), \mathrm{Aut}(C))$ over finite fields, the function `Twists` computes a list of representatives of all twists of a smooth plane quartic over a finite field. This relies on the prior computation of the geometric automorphism group of C, which the algorithms return as a second value when requested.

TwistsOfPlaneQuartic(C, Autos): Crv, SeqEnum -> SeqEnum[Crv], GrpPerm

```
AutomorphismGroup: BoolElt                    Default: false
```

Compute the twists of the plane quartic curve C from its geometric automorphism group *Autos*. If `AutomorphismGroup` is set to `true`, then the furnished automorphism group is additionally returned as an abstract group.

For more details, see [LRRS14, MT10].

Twists(C): Crv -> SeqEnum, GrpPerm

```
AutomorphismGroup: BoolElt                    Default: false
```

Compute the twists of the elliptic, hyperelliptic, or plane quartic curve C. If `AutomorphismGroup` is set to `true`, then the geometric automorphism group of C is additionally returned as an abstract group.

Example: Twists of the Klein Quartic (ex-ae635c)

We compute the twists of the Klein quartic over $\mathbb{F}_{31}$.

```
> P<x,y,z> := PolynomialRing(GF(31), 3);
> PP := ProjectiveSpace(P);
> f := x^3*y + y^3*z + z^3*x;
> C := Curve(ProjectiveSpace(P), f);
> #Twists(C);
4
```

136.5 Bibliography

- [BJ14] Laurent Busé and Jean-Pierre Jouanolou. On the discriminant scheme of homogeneous polynomials. *Math. Comput. Sci.*, 8(2):175–234, 2014.
- [Dix87] J. Dixmier. On the projective invariants of quartic plane curves. *Adv. in Math.*, 64:279–304, 1987.
- [Els15] Andreas-Stephan Elsenhans. Explicit computations of invariants of plane quartic curves. *J. Symbolic Comput.*, 68(part 2):109–115, 2015.
- [GK06] M. Girard and D. R. Kohel. Classification of genus 3 curves in special strata of the moduli space. In Hess, F. (ed.) et al., Algorithmic number theory. 7th international symposium, ANTS-VII, Berlin, Germany, July 23–28, 2006. Proceedings. Berlin: Springer. Lecture Notes in Computer Science 4076, 346-360 (2006)., 2006.
- [LLGR20] Reynald Lercier, Qing Liu, Elisa Lorenzo García, and Christophe Ritzenthaler. Reduction type of smooth quartics. 2020. arXiv:1803.05816.
- [LRRS14] Reynald Lercier, Christophe Ritzenthaler, Florent Rovetta, and Jeroen Sijsling. Parametrizing the moduli space of curves and applications to smooth plane quartics over finite fields. *LMS J. Comput. Math.*, 17(suppl. A):128–147, 2014.
- [LRS16] Reynald Lercier, Christophe Ritzenthaler, and Jeroen Sijsling. Explicit Galois obstruction and descent for hyperelliptic curves with tamely cyclic reduced automorphism group. *Math. Comp.*, 85(300):2011–2045, 2016.
- [MT10] Stephen Meagher and Jaap Top. Twists of genus three curves over finite fields. *Finite Fields Appl.*, 16(5):347–368, 2010.
- [Ohn07] T. Ohno. The graded ring of invariants of ternary quartics I. 2007. Unpublished.
- [Sal79] G. Salmon. *A Treatise on the Higher Plane Curves.* Hodges, Foster and Co, Dublin, 3rd edition, 1879.
- [vR01] Sander Matthijs van Rijnswou. *Testing the equivalence of planar curves.* PhD thesis, Technische Universiteit Eindhoven, Eindhoven, 2001.
- [LRS18] R. Lercier, C. Ritzenthaler, and J. Sijsling. Reconstructing plane quartics from their invariants. *Discrete & Computational Geometry*, pages 1–41, 2018.

Chapter 137

Hyperelliptic Curves

137.1 Introduction

This chapter contains descriptions of functions designed to perform calculations with hyperelliptic curves and their Jacobians. A hyperelliptic curve, which is taken to include the genus one case, is given by a nonsingular generalized Weierstrass equation

$$y^2 + h(x)y = f(x),$$

where $h(x)$ and $f(x)$ are polynomials over a field K. The curve is viewed as embedded in a weighted projective space, with weights 1, $g + 1$, and 1, in which the points at infinity are nonsingular.

Functionality for hyperelliptic curves includes optimized algorithms for working on genus two curves over $\mathbb{Q}$, including heights on the Jacobian, and a datatype for the Kummer surface of the Jacobian. For Jacobians of curves over finite fields, there exist specialized algorithms for computing the group structure of the set of rational points.

The category of hyperelliptic curves is `CrvHyp` and points on curves are of type `PtHyp`. Jacobians of hyperelliptic curves are of type `JacHyp` and points `JacHypPt`. Similarly, the Kummer surface of a genus two curve is of type `SrfKum` with points of type `SrfKumPt`.

The initial development of machinery for hyperelliptic curves was undertaken by Michael Stoll, supported by members of the Magma group.

137.2 Creation Functions

137.2.1 Creation of a Hyperelliptic Curve

A hyperelliptic curve C given by a generalized Weierstrass equation $y^2 + h(x)y = f(x)$ is created by specifying polynomials $h(x)$ and $f(x)$ over a field K. The class of hyperelliptic curves includes curves of genus one, and a hyperelliptic curve may also be constructed by type change from an elliptic curve E. Note that the ambient space of C is a weighted projective space in which the one or two points at infinity are nonsingular.

`HyperellipticCurve(f, h): RngUPolElt, RngUPolElt -> CrvHyp`

`HyperellipticCurve(f, h): RngElt, RngUPolElt -> CrvHyp`

`HyperellipticCurve(f, h): RngUPolElt, RngElt -> CrvHyp`

`HyperellipticCurve(f): RngUPolElt -> CrvHyp`

`HyperellipticCurve([f, h]): [ RngUPolElt ] -> CrvHyp`

Given two polynomials h and $f \in R[x]$ where R is a field or integral domain, this function returns the nonsingular hyperelliptic curve $C : y^2 + h(x)y = f(x)$. If $h(x)$ is not given, then it is taken as zero. If R is an integral domain rather than a field, the base field of the curve is taken to be the field of fractions of R. An error is returned if the given curve C is singular.

HyperellipticCurve(P, f, h): Prj, RngUPolElt, RngUPolElt -> CrvHyp

Create the hyperelliptic curve as described above using the projective space P as the ambient. The ambient P should have dimension 2.

HyperellipticCurveOfGenus(g, f, h): RngIntElt, RngUPolElt, RngUPolElt -> CrvHyp

HyperellipticCurveOfGenus(g, f, h): RngIntElt, RngElt, RngUPolElt -> CrvHyp

HyperellipticCurveOfGenus(g, f, h): RngIntElt, RngUPolElt, RngElt -> CrvHyp

HyperellipticCurveOfGenus(g, f): RngIntElt, RngUPolElt -> CrvHyp

HyperellipticCurveOfGenus(g, [f, h]): RngIntElt, [RngUPolElt] -> CrvHyp

Given a positive integer g and two polynomials h and $f \in R[x]$ where R is a field or integral domain, this function returns the nonsingular hyperelliptic curve of genus g given by $C : y^2 + h(x)y = f(x)$. If $h(x)$ is not given, then it is taken as zero. Before attempting to create C, the function checks that its genus will be g by testing various numerical conditions on f and g. If the genus is not correct, a runtime error is raised. If R is an integral domain rather than a field, the base field of C is taken to be the field of fractions of R. An error is returned if the curve C is singular.

HyperellipticCurve(E): CrvEll -> CrvHyp, Map

Returns the hyperelliptic curve C corresponding to the elliptic curve E, followed by the map from E to C.

137.2.2 Creation Predicates

IsHyperellipticCurve([f, h]): [RngUPolElt] -> BoolElt, CrvHyp

Given a sequence containing two polynomials h, $f \in R[x]$, where R is an integral domain, return **true** if and only if $C : y^2 + h(x)y = f(x)$ is a hyperelliptic curve. In this case, the curve is returned as a second value.

IsHyperellipticCurveOfGenus(g, [f, h]): RngIntElt, [RngUPolElt] -> BoolElt, CrvHyp

Given a positive integer g and a sequence containing two polynomials h, $f \in R[x]$ where R is an integral domain, return **true** if and only if $C : y^2 + h(x)y = f(x)$ is a hyperelliptic curve of genus g. In this case, the curve is returned as a second value.

Example: Creating a Hyperelliptic Curve (ex-1d23ed)

Create a hyperelliptic curve over the rationals:

```
> P<x> := PolynomialRing(RationalField());
> C := HyperellipticCurve(x^6+x^2+1);
> C;
Hyperelliptic Curve defined by y^2 = x^6 + x^2 + 1 over Rational Field
> C![0,1,1];
(0 : 1 : 1)
```

Now create the same curve over a finite field:

```
> P<x> := PolynomialRing(GF(7));
> C := HyperellipticCurve(x^6+x^2+1);
> C;
Hyperelliptic Curve defined by y^2 = x^6 + x^2 + 1 over GF(7)
> C![0,1,1];
(0 : 1 : 1)
```

137.2.3 Changing the Base Ring

BaseChange(C, K): Sch, Fld -> Sch

BaseExtend(C, K): Sch, Fld -> Sch

Given a hyperelliptic curve C defined over a field k, and a field K which is an extension of k, return a hyperelliptic curve C' over K using the natural inclusion of k in K to map the coefficients of C into elements of K.

BaseChange(C, j): Sch, Map -> Sch

BaseExtend(C, j): Sch, Map -> Sch

Given a hyperelliptic curve C defined over a field k and a ring map $j : k \to K$, return a hyperelliptic curve C' over K by applying j to the coefficients of E.

BaseChange(C, n): Sch, RngIntElt -> Sch

BaseExtend(C, n): Sch, RngIntElt -> Sch

If C is a hyperelliptic curve defined over a finite field k and a positive integer n, let K denote the extension of k of degree n. This function returns a hyperelliptic curve C' over K using the natural inclusion of k in K to map the coefficients of C into elements of K.

ChangeRing(C, K): Sch, Rng -> Sch

Given a hyperelliptic curve C defined over a field k, and a field K, return a hyperelliptic curve C' over K that is obtained from C by mapping the coefficients of C into K using the standard coercion map from k to K. This is useful when there is no appropriate ring homomorphism between k and K (e.g., when $k = \mathbb{Q}$ and K is a finite field).

Example: Reducing a Curve Modulo 101 (ex-8e4415)

We construct a curve C over the rationals and use **ChangeRing** to construct the corresponding curve $C1$ over $GF(101)$.

```
> P<x> := PolynomialRing(RationalField());
> C := HyperellipticCurve([x^9-x^2+57,x+1]);
> C1 := ChangeRing(C, GF(101));
> C1;
Hyperelliptic Curve defined by y^2 + (x + 1)*y = x^9 + 100*x^2 + 57
over GF(101)
> Q<t> := PolynomialRing(GF(101));
> HyperellipticPolynomials(C1);
t^9 + 100*t^2 + 57
t + 1
> C2, f := SimplifiedModel(C1);
> HyperellipticPolynomials(C2);
4*t^9 + 98*t^2 + 2*t + 27
0
> P1 := C1![31,30,1];
> P1;
(31 : 30 : 1)
> Q := P1@f; // evaluation
> Q;
(31 : 92 : 1)
> Q@@f; // pullback
(31 : 30 : 1)
```

An explanation of the syntax for isomorphisms of hyperelliptic curves and the functions for models is given below.

137.2.4 Models

SimplifiedModel(C): CrvHyp -> CrvHyp, MapIsoSch

Given a hyperelliptic curve C defined over a field of characteristic not equal to 2, this function returns an isomorphic hyperelliptic curve C' of the form $y^2 = f(x)$, followed by the isomorphism $C \to C'$.

`HasOddDegreeModel(C): CrvHyp -> BoolElt, CrvHyp, MapIsoSch`

Given a hyperelliptic curve C, this function returns `true` if C has a model C' of the form $y^2 = f(x)$, with f of odd degree. If so, C' is returned together with the isomorphism $C \to C'$.

`IntegralModel(C): CrvHyp -> CrvHyp, MapIsoSch`

```
Reduce: BoolElt                    Default: false
```

Given a hyperelliptic curve C defined over the rationals, this function returns an isomorphic curve C' given by polynomials with integral coefficients, together with the isomorphism $C \to C'$. If `Reduce` is set `true`, common divisors of the coefficients are eliminated as far as possible.

`MinimalWeierstrassModel(C): CrvHyp -> CrvHyp, MapIsoSch`

```
Bound        : RngIntElt                   Default: 0
CrvHypMinimal: Verbose                     Default: 3
```

Given a hyperelliptic curve C defined over the rationals, this function returns a globally minimal Weierstrass model C' of C. If `Bound` is set, it gives an upper bound for the bad primes that are checked. As this calculation uses trial division, `Bound` should not be set much larger than 10^7. The isomorphism $C \to C'$ is returned as a second value.

`pIntegralModel(C, p): CrvHyp, RngIntElt -> CrvHyp, MapIsoSch`
`pIntegralModel(C, p): CrvHyp, FldRatElt -> CrvHyp, MapIsoSch`
`pIntegralModel(C, p): CrvHyp, RngUPolElt -> CrvHyp, MapIsoSch`
`pIntegralModel(C, p): CrvHyp, FldFunRatUElt -> CrvHyp, MapIsoSch`
`pIntegralModel(C, p): CrvHyp, Infty -> CrvHyp, MapIsoSch`

```
CrvHypMinimal: Verbose                     Default: 3
```

Given a hyperelliptic curve C defined over $\mathbb{Q}$ or a rational function field, this function returns a model C' of the curve that is integral at the place p given as an integer, rational, polynomial, rational function, or ∞. The isomorphism $C \to C'$ is returned as a second value.

`pNormalModel(C, p): CrvHyp, RngIntElt -> CrvHyp, MapIsoSch`
`pNormalModel(C, p): CrvHyp, FldRatElt -> CrvHyp, MapIsoSch`
`pNormalModel(C, p): CrvHyp, RngUPolElt -> CrvHyp, MapIsoSch`
`pNormalModel(C, p): CrvHyp, FldFunRatUElt -> CrvHyp, MapIsoSch`
`pNormalModel(C, p): CrvHyp, Infty -> CrvHyp, MapIsoSch`

```
CrvHypMinimal: Verbose                     Default: 3
```

Given a hyperelliptic curve C defined over $\mathbb{Q}$ or a rational function field, this function returns a model C' of the curve that is normal at the place p given as an integer, rational, polynomial, rational function, or ∞. The isomorphism $C \to C'$ is returned as a second value.

`pMinimalWeierstrassModel(C, p): CrvHyp, RngIntElt -> CrvHyp, MapIsoSch`
`pMinimalWeierstrassModel(C, p): CrvHyp, FldRatElt -> CrvHyp, MapIsoSch`
`pMinimalWeierstrassModel(C, p): CrvHyp, RngUPolElt -> CrvHyp, MapIsoSch`
`pMinimalWeierstrassModel(C, p): CrvHyp, FldFunRatUElt -> CrvHyp, MapIsoSch`
`pMinimalWeierstrassModel(C, p): CrvHyp, Infty -> CrvHyp, MapIsoSch`

```
CrvHypMinimal: Verbose                     Default: 3
```

Given a hyperelliptic curve C defined over $\mathbb{Q}$ or a rational function field, this function returns a Weierstrass model C' of the curve that is minimal at the place p given as an integer, rational, polynomial, rational function, or ∞. The isomorphism $C \to C'$ is returned as a second value.

`ReducedModel(C): CrvHyp -> CrvHyp, MapIsoSch`

```
Smallest    : BoolElt                    Default: false
Height      : BoolElt                    Default: false
CrvHypReduce: Verbose                    Default: 3
```

Given a curve with integral coefficients, computes a model that is reduced with respect to the action of $\mathrm{SL}_2(\mathbb{Z})$. If `Smallest` is set, return a model whose `SimplifiedModel` is given by a binary form of minimal euclidean length (`Height := false`) or minimal height (`Height := true`).

`ReducedMinimalWeierstrassModel(C): CrvHyp -> CrvHyp, MapIsoSch`

```
Smallest     : BoolElt                    Default: false
Height       : BoolElt                    Default: false
CrvHypMinimal: Verbose                    Default: 3
CrvHypReduce : Verbose                    Default: 3
```

Given a hyperelliptic curve C defined over the rationals, this function returns a globally minimal integral Weierstrass model C' of C that is reduced with respect to the action of $\mathrm{SL}_2(\mathbb{Z})$, using `ReducedModel`, which makes use of the parameters `Smallest` and `Height`. The isomorphism $C \to C'$ is returned as a second value.

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

This sets the verbose printing level for the curve reduction algorithms of Stoll and Wamelen. The second argument can take integral values in the interval $[0, 3]$, or boolean values: `false` (equivalent to 0) and `true` (equivalent to 1).

137.2.5 Minimization and Reduction of Binary Forms

Intrinsics are provided to perform the minimization and reduction routines directly on binary forms over the rationals or integers. These are slightly more general, in that they apply to homogeneous forms whose associated polynomial may have repeated roots. In the case of non-square-free forms, the routines are applied to the square-free part and the reducing transformation is then applied to the original form.

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

This sets the verbose printing level for the minimization phases of the following intrinsics. The second argument can take integral values in the interval $[0, 2]$, or boolean values: `false` (equivalent to 0) and `true` (equivalent to 1).

`MinimizeAtP(f, p): RngMPolElt, RngIntElt -> RngMPolElt, AlgMatElt, RngIntElt`

The first argument, f, should be a homogeneous polynomial in two variables either defined over the integer ring or defined over the rational field with integral coefficients. The second argument p should be a prime.

Computes a transformation $f(x, y) \mapsto F(x, y) = p^{-r} f(ax+by, cx+dy)$ with $r \geq 0$ and $a, b, c, d \in \mathbb{Z}$, $ad - bc \neq 0$, such that F still has integral coefficients but whose discriminant has minimal p-part amongst all integral forms $GL_2(\mathbb{Q})$-equivalent to f.

Returns F, the transformation matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$, and the scaling factor $s = p^r$.

`MinRedBinaryForm(f): RngMPolElt -> RngMPolElt, AlgMatElt, RngIntElt`

The argument, f, should be a homogeneous polynomial in two variables defined over the rational field.

Computes a transformation $f(x, y) \mapsto F(x, y) = s^{-1} f(ax + by, cx + dy)$ with $1 \leq s \in \mathbb{Z}$ and $a, b, c, d \in \mathbb{Z}$, $ad - bc \neq 0$, such that F has integral coefficients and is minimized and reduced.

Here, if F_0 is the square-free part of F, minimized means that the absolute value of the discriminant of F_0 is minimal amongst all integral forms $GL_2(\mathbb{Q})$-equivalent to the square-free part of f. Reduced

means that F_0 is reduced up to $\mathrm{SL}_2(\mathbb{Z})$-equivalence following Stoll's reduction algorithm, as for the `ReducedModel` intrinsic on hyperelliptic curves.

Returns F, the transformation matrix $\begin{pmatrix} a & b \\ c & d \end{pmatrix}$, and the scaling factor s.

Note that the `CrvHypReduce` verbose level should be set to obtain verbose output for the reduction phase of this intrinsic.

`MinRedBinaryForm(f): RngUPolElt -> RngUPolElt, AlgMatElt, RngIntElt`

`degree: RngIntElt` Default: `Degree(f)`

Variant of the previous intrinsic, taking a univariate polynomial over $\mathbb{Q}$, f, and applying the minimization and reduction to the binary form f_h which is the homogenisation of f, the homogenising variable being the second one. If the parameter `degree` is set to an integer d greater than Degree(f) (it's default is Degree(f)), then f_h is taken as the degree d homogenisation.

The return values are the same as for the previous intrinsic, except that the first return value is the dehomogenised F computed for f_h, i.e. $F(x, 1)$.

Example: Minimizing and Reducing a Binary Form (ex-68a467)

```
> P<x,y> := PolynomialRing(Rationals(),2);
> f := 65537*x^8 + 131104*x^7*y + 115136*x^6*y^2 + 60928*x^5*y^3 +
> 35840*x^4*y^4 + 60928*x^3*y^5 + 115136*x^2*y^6 + 131104*x*y^7 + 65537*y^8;
> F,M,s := MinRedBinaryForm(f);
> F;M;s;
x^8 + y^8
[ 1  4]
[-4 -1]
2562890625
> s*F eq f^ChangeRing(M,Rationals());
true;
```

137.2.6 Predicates on Models

`IsSimplifiedModel(C): CrvHyp -> BoolElt`

Returns `true` if the hyperelliptic curve C is of the form $y^2 = f(x)$.

`IsIntegral(C): CrvHyp -> BoolElt`

Given a hyperelliptic curve C, the function returns `true` if C has integral coefficients, and `false` otherwise.

`IspIntegral(C, p): CrvHyp, RngIntElt -> BoolElt`

`IspIntegral(C, p): CrvHyp, RngUPolElt -> BoolElt`

`IspIntegral(C, p): CrvHyp, Infty -> BoolElt`

Given a hyperelliptic curve C defined over $\mathbb{Q}$ or a rational function field, this function returns `true` if the given model of C is integral at the given place.

`IspNormal(C, p): CrvHyp, RngIntElt -> BoolElt`

`IspNormal(C, p): CrvHyp, RngUPolElt -> BoolElt`

`IspNormal(C, p): CrvHyp, Infty -> BoolElt`

Given a hyperelliptic curve C defined over $\mathbb{Q}$ or a rational function field, this function returns `true` if the given model of C is normal at the given place.

`IspMinimal(C, p): CrvHyp, RngIntElt -> BoolElt, BoolElt`

`IspMinimal(C, p): CrvHyp, RngUPolElt -> BoolElt, BoolElt`

`IspMinimal(C, p): CrvHyp, Infty -> BoolElt, BoolElt`

Given a hyperelliptic curve C defined over $\mathbb{Q}$ or a rational function field, this function decides whether the given model of C is minimal at the given place. The returned values as follows:

- `false`, `false` if C is not an integral minimal model at the given place.
- `true`, `false` if C is integral and minimal at the given place, but not the unique minimal model (up to transformations that are invertible over the local ring).
- `true`, `false` if C is the unique integral minimal model at the given place, (up to transformations that are invertible over the local ring).

137.2.7 Type Change Predicates

`IsEllipticCurve(C): CrvHyp -> BoolElt, CrvEll, MapIsoSch, MapIsoSch`

The function returns `true` if and only if C is a genus one hyperelliptic curve of odd degree, in which case it also returns an elliptic curve E isomorphic to C followed by the isomorphism $C \to E$ and the inverse isomorphism $E \to C$.

137.3 Operations on Curves

137.3.1 Quadratic Twists

There are standard functions for quadratic twists of hyperelliptic curves in characteristic not equal to 2. In addition, from the new package [LRS21] (described in more detail in the next section) there are functions to return *all* twists of a genus 2 hyperelliptic curve over a finite field of any characteristic.

`QuadraticTwist(C, d): CrvHyp, RngElt -> CrvHyp`

Given a hyperelliptic curve C defined over a field k of characteristic not equal to 2 and an element d that is coercible into k, return the quadratic twist of C by d.

`QuadraticTwist(C): CrvHyp -> CrvHyp`

Given a hyperelliptic curve C defined over a finite field k, return the standard quadratic twist of C over the unique extension of k of degree 2. If the characteristic of k is odd, then this is the same as the twist of C by a primitive element of k.

`QuadraticTwists(C): CrvHyp -> SeqEnum`

Given a hyperelliptic curve C defined over a finite field k of odd characteristic, return a sequence containing the non-isomorphic quadratic twists of C.

`IsQuadraticTwist(C, D): CrvHyp, CrvHyp -> BoolElt, RngElt`

`CrvHypIso: Verbose` Default: 3

Given hyperelliptic curves C and D over a common field k having characteristic not equal to two, return `true` if and only if C is a quadratic twist of D over k. If so, the twisting factor is returned as the second value.

Example: Quadratic Twists over a Finite Field (ex-a5b80e)

We construct the quadratic twists of the hyperelliptic curve $y^2 = x^6 + x^2 + 1$ defined over $\mathbf{F}_7$.

```
> P<x> := PolynomialRing(GF(7));
> C := HyperellipticCurve(x^6+x^2+1);
> QuadraticTwists(C);
[
    Hyperelliptic Curve defined by y^2 = x^6 + x^2 + 1 over GF(7),
    Hyperelliptic Curve defined by y^2 = 3*x^6 + 3*x^2 + 3 over GF(7)
]
> IsIsomorphic($1[1],$1[2]);
false
```

Example: Quadratic Twist over the Rationals (ex-0bc2c3)

We take a hyperelliptic curve over the rationals and form a quadratic twist of it.

```
> P<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(x^6+x);
> C7 := QuadraticTwist(C, 7);
> C7;
Hyperelliptic Curve defined by y^2 = 7*x^6 + 7*x over Rational Field
```

We now use the function `IsIsomorphic` to verify that C and C_7 are nonisomorphic. We then extend the field of definition of both curves to $\mathbb{Q}(\sqrt{7})$ and verify that the curves become isomorphic over this extension.

```
> IsIsomorphic(C, C7);
false
> K<w> := ext< Rationals() | x^2-7 >;
> CK := BaseChange(C, K);
> C7K := BaseChange(C7, K);
> IsIsomorphic(CK, C7K);
true (x : y : z) :-> (x : -1/7*w*y : z)
```

137.3.2 Elementary Invariants

HyperellipticPolynomials(C): CrvHyp -> RngUPolElt, RngUPolElt

The univariate polynomials $f(x)$, $h(x)$, in that order, defining the hyperelliptic curve C by $y^2 + h(x)y = f(x)$.

Degree(C): CrvHyp -> RngIntElt

Degree(C): SetPtHyp -> RngIntElt

The degree of the hyperelliptic curve C or a pointset C of a hyperelliptic curve.

Discriminant(C): CrvHyp -> RngElt

The discriminant of the hyperelliptic curve C.

Genus(C): CrvHyp -> RngIntElt

The genus of the hyperelliptic curve C.

Conductor(C): CrvHyp -> RngIntElt

Conductor of a hyperelliptic curve C defined over $\mathbb{Q}$ or a number field.

Conductor(C, p): CrvHyp[FldRat], RngIntElt -> RngIntElt

Conductor(C, P): CrvHyp[FldNum], RngOrdIdl -> RngIntElt

Conductor exponent of a hyperelliptic curve $C/\mathbb{Q}$ at a prime p or a curve over a number field at a prime ideal P.

Example: Conductor over the Rationals (ex-9451e3)

```
> R<x>:=PolynomialRing(Rationals());
> C:=HyperellipticCurve(x^8+1,x); C;
Hyperelliptic Curve defined by y^2 + x*y = x^8 + 1 over Rational Field
> Factorization(Integers()!Discriminant(C));
[ <2, 4>, <109, 2>, <601, 2> ]
> Factorization(Conductor(C));                    // global
[ <2, 2>, <109, 2>, <601, 2> ]
> Conductor(C,2),Conductor(C,3),Conductor(C,5);   // local
2 0 0
```

ConductorExponent(C): CrvHyp[FldPad] -> RngIntElt

`Conductor(C): CrvHyp[FldPad] -> FldPadElt`

Conductor exponent and conductor ideal (uniformizer to the conductor exponent power) of a hyperelliptic curve over a p-adic field.

Example: Conductor over a p-Adic Field (ex-581c7e)

Here is a genus 4 example over $\mathbb{Q}_3$.

```
> K:=pAdicField(3,20);
> R<x>:=PolynomialRing(K);
> C:=HyperellipticCurve(x^9+1);
> ConductorExponent(C);
12
> Conductor(C);
3^12 + O(3^32)
```

`EulerFactor(C, p): CrvHyp[FldRat], RngIntElt -> RngUPolElt`

`EulerFactor(C, P): CrvHyp[FldNum], RngOrdIdl -> RngUPolElt`

`EulerFactor(C): CrvHyp[FldPad] -> RngUPolElt`

```
Degree: RngIntElt                    Default: Infinity()
```

The Euler factor (local polynomial) of a hyperelliptic curve defined over $\mathbb{Q}$, a number field, or a p-adic field at a prime p. If `Degree` is specified, it is computed only up to that degree.

`EulerFactor(C, p): CrvHyp[FldNum], RngIntElt -> RngUPolElt`

```
Degree: RngIntElt                    Default: Infinity()
```

The total Euler factor of a hyperelliptic curve defined over a number field over all primes above p. If `Degree` is specified, it is computed only up to that degree.

Example: Euler Factors and Conductors at 3 (ex-9f8b78)

We consider the genus 2 curve $C/\mathbb{Q} : y^2 = x^5 + x^2 + 3$ and compute its conductor at $p = 3$, and over $K = \mathbb{Q}(\sqrt[5]{3})$ at the unique prime ideal $P|3$.

```
> R<x>:=PolynomialRing(Rationals());
> C:=HyperellipticCurve(x^5+x^2+3);
> EulerFactor(C,3);                      // local factor /Q at 3
-T + 1
> K:=NumberField(x^5-3);
> P:=Ideal(Decomposition(K,5)[1,1]);
> EulerFactor(BaseChange(C,K),P);        // local factor /K at P
25*x^4 - 5*x^3 - x + 1
> EulerFactor(BaseChange(C,Completion(K,P)));   // same, computed over K_P
25*x^4 - 5*x^3 - x + 1
> EulerFactor(BaseChange(C,K),2);        // total factor /K over all P|2
1
```

137.3.3 Base Ring

`BaseField(C): Sch -> Fld`

`BaseRing(C): Sch -> Fld`

`CoefficientRing(C): Sch -> Fld`

The base field of the hyperelliptic curve C.

137.4 Function Field

137.4.1 Function Field and Polynomial Ring

`FunctionField(C): Sch -> FldFunG`

The function field of the hyperelliptic curve C.

`DefiningPolynomial(C): Sch -> RngMPolElt`

A weighted homogeneous polynomial for the hyperelliptic curve C.

`EvaluatePolynomial(C, a, b, c): CrvHyp, RngElt, RngElt, RngElt -> RngElt`

`EvaluatePolynomial(C, [a, b, c]): CrvHyp, [RngElt] -> RngElt`

Evaluates the homogeneous defining polynomial of the hyperelliptic curve C at the point (a, b, c).

137.5 Points

The hyperelliptic curve is embedded in a weighted projective space, with weights 1, $g + 1$, and 1, respectively on x, y, and z. Therefore point triples satisfy the equivalence relation $(x : y : z) = (\mu x : \mu^{g+1} y : \mu z)$, and the points at infinity are then normalized to take the form $(1 : y : 0)$.

137.5.1 Creation of Points

`C ! [x, y]: CrvHyp, [RngElt] -> PtHyp`

`C ! [x, y, z]: CrvHyp, [RngElt] -> PtHyp`

`elt< PS | x, y >: SetPtHyp, RngElt, RngElt, RngElt -> PtHyp`

`elt< PS | x, y, z >: SetPtHyp, RngElt, RngElt, RngElt -> PtHyp`

Returns the point on a hyperelliptic curve C specified by the coordinates (x, y, z). The `elt` constructor takes the pointset of a hyperelliptic curve as an argument. If z is not specified it is assumed to be 1.

`C ! P: CrvHyp, PtHyp -> PtHyp`

Given a point P on a hyperelliptic curve C_1, such that C is a base extension of C_1, this returns the corresponding point on the hyperelliptic curve C. The curve C can be, e.g., the reduction of C_1 to finite characteristic (i.e., base extension to a finite field) or the tautological coercion to itself.

`Points(C, x): CrvHyp, RngElt -> SetIndx`

`RationalPoints(C, x): CrvHyp, RngElt -> SetIndx`

`Points(C, x): CrvHyp, Infty -> SetIndx`

`RationalPoints(C, x): CrvHyp, Infty -> SetIndx`

The indexed set of all rational points on the hyperelliptic curve C that have the value x as their x-coordinate. (Rational points are those with coordinates in the coefficient ring of C). Note that points at infinity are considered to have ∞ as their x-coordinate.

`PointsAtInfinity(C): CrvHyp -> SetIndx`

The points at infinity for the hyperelliptic curve C returned as an indexed set of points.

`IsPoint(C, S): CrvHyp, SeqEnum -> BoolElt, PtHyp`

The function returns `true` if and only if the sequence S specifies a point on the hyperelliptic curve C, and if so, returns this point as the second value.

Example: Points at Infinity (ex-c9370e)

We look at the point at infinity on $y^2 = x^5 + 1$.

```
> P<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(x^5+1);
> PointsAtInfinity(C);
{@ (1 : 0 : 0) @}
```

There is only one, and to see that this really is a point on C it must be remembered that in Magma, all hyperelliptic curves are considered to live in weighted projective spaces:

```
> Ambient(C);
Projective Space of dimension 2
Variables : $.1, $.2, $.3
Gradings :
1       3       1
```

In fact, the point is nonsingular on C, as we now check. (It's worth remembering that all the functionality for curves, for instance `IsNonsingular`, applies to hyperelliptic curves as a special case.)

```
> pointAtInfinity := C![1,0,0];  // Entering the point by hand.
> IsNonsingular(pointAtInfinity);
true
```

137.5.2 Random Points

`Random(C): CrvHyp -> PtHyp`

Given a hyperelliptic curve C defined over a finite field, this returns a point chosen at random on the curve. If the set of all points on C has already been computed, this gives a truly random point, otherwise the ramification points have a slight advantage.

137.5.3 Predicates on Points

`P eq Q: PtHyp, PtHyp -> BoolElt`

Returns `true` if and only if the two points P and Q on the same hyperelliptic curve have the same coordinates.

`P ne Q: PtHyp, PtHyp -> BoolElt`

Returns `false` if and only if the two points P and Q on the same hyperelliptic curve have the same coordinates.

137.5.4 Access Operations

`P[i]: PtHyp, RngIntElt -> RngElt`

The i-th coordinate of the point P, for $1 \leq i \leq 3$.

`Eltseq(P): PtHyp -> SeqEnum`

`ElementToSequence(P): PtHyp -> SeqEnum`

Given a point P on a hyperelliptic curve, this returns a 3-element sequence consisting of the coordinates of the point P.

137.5.5 Arithmetic of Points

`- P: PtHyp -> PtHyp`

`Involution(P): PtHyp -> PtHyp`

Given a point P on a hyperelliptic curve, this returns the image of P under the hyperelliptic involution.

137.5.6 Enumeration and Counting Points

`NumberOfPointsAtInfinity(C): CrvHyp -> RngIntElt`

The number of points at infinity on the hyperelliptic curve C.

`PointsAtInfinity(C): CrvHyp -> SetIndx`

The points at infinity for the hyperelliptic curve C returned as an indexed set of points.

`# C: CrvHyp -> RngIntElt`

Given a hyperelliptic curve C defined over a finite field, this returns the number of rational points on C.

If the base field is small or there is no other good alternative, a naive point counting technique is used. However, if they are applicable, the faster p-adic methods described in the `#J` section are employed (which actually yield the full zeta function of C). As for `#J`, the verbose flag `JacHypCnt` can be used to output information about the computation.

`Points(C): CrvHyp -> SetIndx`

`RationalPoints(C): CrvHyp -> SetIndx`

`RationalPoints(C): SetPtHyp -> SetIndx`

```
Bound           : RngIntElt                    Default:
NPrimes         : RngIntElt                    Default: 30
DenominatorBound: RngIntElt                    Default: Bound
```

For a hyperelliptic curve C defined over a finite field, the function returns an indexed set of all rational points on C.

For a curve C over $\mathbb{Q}$ of the form $y^2 = f(x)$ with integral coefficients, it returns the set of points such that the naive height of the x-coordinate is less than `Bound`.

For a curve C over a number field, it returns the set of points in a search region which is controlled by the parameters `Bound` (which must be specified) and `DenominatorBound`. The algorithm is a sieve method, described in Appendix A of [Bru02]. The parameter `NPrimes` controls the number of primes to be used for the sieve.

`PointsGenus2(C): CrvHyp -> SetIndx, BoolElt, RngIntElt`

`RationalPointsGenus2(C): CrvHyp -> SetIndx, BoolElt, RngIntElt`

```
Bound1     : RngIntElt                         Default: 1000
Bound2     : RngIntElt                         Default: 20000
Fast       : BoolElt                           Default: false
RankBound  : RngIntElt                         Default: Infinity()
PrimeCutoff: RngIntElt                         Default: 10000
```

For a curve C over $\mathbb{Q}$ of the form $y^2 = f(x)$ try to determine the set of rational points

This returns a set of rational points on the curve, a flag indicating whether we know that these are all the points, and if this is not the case, a number such that all the points of (multiplicative x-coordinate) height up to that bound are included in the set.

`Bound1` is an argument passed to `RationalPoints`. Same is true for `Bound2`, which will be called after the first search for small points has failed, and the curve is everywhere locally soluble. If `Fast` is set, smaller bounds than the default ones are used in the call to `MordellWeilGroupGenus2` for the search on 2-coverings. This has the effect of making this function fail faster when otherwise a time-intensive search for rational points on 2-coverings would be triggered. `PrimeCutoff` is passed on to `TwoCoverDescent`.

This function uses a combination of techniques:

i) search for (relatively small) points,

ii) test for everywhere local solubility,

iii) two-cover descent,

iv) degree 2 and 3 elliptic subcovers of rank zero,

v) computation of generators of a finite-index subgroup of the Mordell-Weil group,

vi) Chabauty when the rank is at most 1,

vii) a check whether the curve has rational divisors of odd degree,

viii) if the rank is at least 2 and no points were found, a Mordell-Weil sieve computation.

The two possible failure modes are:

i) failure to determine the rank

ii) rank is at least 2 and rational points exist; in these cases the result is `false`, and it returns the set of rational points up to the height bound for the search.

`PointsKnown(C): CrvHyp -> BoolElt`

Returns `true` if and only if the points of the hyperelliptic curve C have been computed. This can especially be helpful when the curve is likely to have many points and when one does not wish to trigger the possibly expensive point computation.

Example: Points on a Curve of Diophantus (ex-814aee)

For the following classical curve of Diophantus' *Arithmetica*, it is proved by Joseph Wetherell [Wet97] that Diophantus found all positive rational points. The following Magma code enumerates the points on this curve.

```
> P<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(x^6+x^2+1);
> Points(C : Bound := 1);
{@ (1 : -1 : 0), (1 : 1 : 0), (0 : -1 : 1), (0 : 1 : 1) @}
> Points(C : Bound := 2);
{@ (1 : -1 : 0), (1 : 1 : 0), (0 : -1 : 1), (0 : 1 : 1), (-1 : -9 : 2),
(-1 : 9 : 2), (1 : -9 : 2), (1 : 9 : 2) @}
> Points(C : Bound := 4);
{@ (1 : -1 : 0), (1 : 1 : 0), (0 : -1 : 1), (0 : 1 : 1), (-1 : -9 : 2),
(-1 : 9 : 2), (1 : -9 : 2), (1 : 9 : 2) @}
```

137.5.7 Frobenius

`Frobenius(P, F): PtHyp, FldFin -> PtHyp`

Check: BoolElt	Default: true

Applies the Frobenius $x -> x^{(\#F)}$ to P. If `Check` is `true`, it verifies that the curve of which P is a point is defined over the finite field F.

`FrobeniusMatrix(C, p): CrvHyp, RngIntElt -> Mtrx`

Precision: RngIntElt	Default: 10

Given a hyperelliptic curve over the rationals and a good prime p, this computes the matrix corresponding to the action of Frobenius to the indicated precision. The basis used is $x^e(dx/y)$ for $0 \leq e < d-1$, where the curve is given by $y^2 = f(x)$ with f of degree d.

137.6 Jacobians

The Jacobian of a hyperelliptic curve is implemented as the divisor class group of the curve. In particular, no equations giving the Jacobian as a variety ever appear. The Jacobian of any hyperelliptic curve can be created, but most of the interesting functionality is over finite fields, or for genus 2 over number fields or $\mathbb{Q}$.

137.6.1 Creation of a Jacobian

`Jacobian(C): CrvHyp -> JacHyp`

The Jacobian of the hyperelliptic curve C.

137.6.2 Access Operations

`Curve(J): JacHyp -> CrvHyp`

The hyperelliptic curve from which the Jacobian J was constructed.

`Dimension(J): JacHyp -> RngIntElt`

The dimension of the Jacobian J as an algebraic variety, equal to the genus of the curve C of which J is the Jacobian.

137.6.3 Base Ring

`BaseField(J): JacHyp -> Fld`

`BaseRing(J): JacHyp -> Rng`

`CoefficientRing(J): JacHyp -> Rng`

The base field of the Jacobian J.

137.6.4 Changing the Base Ring

`BaseChange(J, F): JacHyp, Rng -> JacHyp`

`BaseExtend(J, F): JacHyp, Rng -> JacHyp`

The base extension of the Jacobian J to the field F.

`BaseChange(J, j): JacHyp, Map -> JacHyp`

`BaseExtend(J, j): JacHyp, Map -> JacHyp`

The base extension of the Jacobian J obtained by the map j, where j is a ring homomorphism with the base field of C as its domain.

`BaseChange(J, n): JacHyp, RngIntElt -> JacHyp`

`BaseExtend(J, n): JacHyp, RngIntElt -> JacHyp`

The base extension of the Jacobian J over a finite field to its degree n extension.

137.7 Richelot Isogenies

Let k be a field of characteristic different from 2. We consider a curve of genus 2, given by an equation

$$C : y^2 = f(x)$$

where $f(x)$ is a square-free polynomial of degree 5 or 6. Let J be the Jacobian of C. In this section we mean by a *Richelot isogeny* a polarized isogeny $\Phi : J \to A$ between principally polarized abelian surfaces, such that the kernel of Φ over the algebraic closure has group structure $\mathbb{Z}/2\mathbb{Z} \times \mathbb{Z}/2\mathbb{Z}$. We have that $J[\Phi] \subset J[2]$ is maximal isotropic with respect to the Weil-pairing on $J[2]$.

We can represent the points of $J[\Phi]$ over the algebraic closure as divisors in the following way. We write

$$f(x) = cQ_1(x)Q_2(x)Q_3(x),$$

where the Q_i are degree 2 polynomials if $\deg(f) = 6$. If $\deg(f) = 5$ then Q_2, Q_3 are of degree 2 and Q_1 is of degree 1 and is considered to represent a degree 2 with a root at $x = \infty$. Then

$$\{0, [Q_1(x) = 0] - [Q_2(x) = 0], [Q_1(x) = 0] - [Q_3(x) = 0], [Q_2(x) = 0] - [Q_3(x) = 0]\}$$

is the kernel of some Richelot-isogeny and, conversely, any Richelot kernel can be represented in this way.

The Q_i do not have to be defined over the ground field individually. One way of specifying such a kernel is to write

$$f(x) = c\mathrm{Norm}_{L[x]/k[x]}Q(x)$$

where $L = k[t]/(h(t))$ for some square free cubic polynomial h and $Q(x) \in L[x]$. If L is totally split and A is the Jacobian of a genus 2 curve then a description the genus 2 curve D such that $A = \mathrm{Jac}(D)$ is classically known. See [Smi05], Chapter 8 for an exposition that is relatively close to the description given here. See [BD09] for a description of D for general L.

In special cases, the codomain A can be a product of elliptic curves or the Weil-restriction of an elliptic curve with respect to a quadratic extension of k. In that case, the curve C has extra automorphisms that respects the representation $f(x) = c\mathrm{Norm}_{L[x]/k[x]}Q(x)$. and one can find the relevant elliptic curves as quotients of C.

A *double Richelot isogeny* φ between two principally polarized abelian surfaces A and A' (over some base field k) is the composition of two Richelot isogenies $A \to A'' \to A'$ that are individually not defined over k (but φ is). We exclude the trivial case that φ is multiplication by 2 (which occurs when the two Richelot isogenies are duals of each other). Then the kernel K of φ is isomorphic to $\mathbb{Z}/2\mathbb{Z} \times \mathbb{Z}/2\mathbb{Z} \times \mathbb{Z}/4\mathbb{Z}$ as an abelian group. Twice any of the points of order 4 in the kernel is a point P of order 2 on A, which must be k-rational (it is also the non-trivial element in the annihilator of $K \cap A[2]$ under the Weil pairing). There are three maximally isotropic subspaces of $A[2]$ (with respect to the Weil pairing) that contain P; each of them can be used as the kernel of the first Richelot isogeny. Since none of these Richelot isogenies is defined over k, the absolute Galois group of k has to permute these three spaces transitively. We now assume $\mathrm{char}(k) \neq 2$ and that $A = J$ is the Jacobian of a curve of the form $y^2 = f(x, z)$. Then

$$f(x, z) = q(x, z)h(x, z) \qquad \text{in } k[x]$$

with a quadratic q and a quartic h whose Galois group is not contained in D_4 (which is equivalent to saying that the three factorizations of h into two quadratics are all Galois-conjugate. These factorizations correspond to the maximal isotropic subspaces containing the point P given by q).

We note that if A' is doubly Richelot isogenous to J, then the intermediate abelian surface A'' must also be a Jacobian. Since the three possible Richelot isogenies $J \to A''$ are Galois-conjugate, the same is true for the three possible A'', so they are either all Jacobians or all products of two elliptic curves (at least geometrically). Working over a suitable field extension, we can assume that $h(x, z) = xz(x - z)(x - az)$. Recall that the condition for A'' to be split is that the three quadratic factors corresponding to the kernel of $J \to A''$ are linearly dependent. If all versions of A'' are split, this means that there is some quadratic $q(x, z) = rx^2 + sxz + tz^2$ such that the triples of vectors

- (r, s, t), $(0, 1, 0)$, $(1, -1 - a, a)$
- (r, s, t), $(1, -1, 0)$, $(0, 1, -a)$
- (r, s, t), $(1, -a, 0)$, $(0, 1, -1)$

are linearly dependent. Equivalently, the cross products of the last two vectors in each triple all lie in the plane orthogonal to (r, s, t); in particular, these cross products are linearly dependent. But

$$\begin{aligned}(0, 1, 0) \times (1, -1 - a, a) &= (a, 0, -1)\\ (1, -1, 0) \times (0, 1, -a) &= (a, a, 1)\\ (1, -a, 0) \times (0, 1, -1) &= (a, 1, 1)\end{aligned}$$

and

$$\begin{vmatrix} a & 0 & -1 \\ a & a & 1 \\ a & 1 & 1 \end{vmatrix} = 2a(a - 1) \neq 0\,.$$

So we obtain all surfaces doubly Richelot isogenous to J by taking a Richelot isogeny to another Jacobian over a cubic field and composing it with a further Richelot isogeny (over the same field) from that Jaocbian.

The `DoubleRichelot..` intrinsics below compute a sequence containing all double Richelot isogenous Jacobians (resp., the corresponding curves) for a given genus 2 Jacobian over the rational numbers such that the intermediate Richelot isogenous abelian surface (which will be defined over a cubic number field) is a Jacobian.

Let $\varphi\colon J \to J'$ be an isogeny between two Jacobians of genus 2 curves whose degree is a power of 2 (and such that the canonical principal polarizations are compatible via the isogeny). Then φ factors as a product of Richelot isogenies and double Richelot isogenies defined over the base field. So we can find all J' that are 2-power isogenous to a given J by constructing the connected component of the graph of Richelot and double Richelot isogenies that contains J. The `TwoPowerIsogenies` intrinsic is provided for this purpose.

Caveat: right now, we cannot guarantee that we really find all isogenous Jacobians (say), since the graph traversal stops at points where we reach a non-Jacobian abelian surface. It might be possible that a 2-power isogenous Jacobian can only be reached via an isogeny to a non-Jacobian.

`RichelotIsogenousSurfaces(J): JacHyp -> List, List`

`RichelotIsogenousSurfaces(C): CrvHyp -> List, List`

`Kernels: BoolElt` Default: `true`

Computes the richelot isogenies defined over the basefield of the given abelian varieties and returns a list of objects representing the codomains. If the codomain is the Jacobian of a genus 2 curve, then that Jacobian is returned or, if a curve is given instead of a Jacobian, the corresponding curve.

If the codomain is a product of elliptic curves, a Cartesian product of elliptic curves is returned. If the codomain is the Weil restriction of an elliptic curve relative to a quadratic extension, then the elliptic curve over the quadratic extension is returned.

If `Kernels` is specified then a second list is returned, consisting of quadratic polynomials over cubic algebras. Each describes the kernel of the relevant isogeny.

`RichelotIsogenousSurface(J, kernel): JacHyp, RngUPolElt[RngUPolRes] -> .`

`RichelotIsogenousSurface(C, kernel): CrvHyp, RngUPolElt[RngUPolRes] -> .`

Given a genus 2 Jacobian and a Richelot kernel, return the codomain. The genus 2 curve must be given by a model of the form $C : y^2 = f(x)$ and the kernel must be a quadratic polynomial $Q(x)$ over a cubic algebra L such that $\mathrm{Norm}_{L[x]/k[x]}Q(x) = cf(x)$. The elements of the second list returned by *RichelotIsogenousSurfaces* when given `Kernels:=true` are valid kernel descriptions. The codomain is returned using the same conventions as for *RichelotIsogenousSurfaces*.

Example: Three Richelot Isogenies (ex-194252)

We will determine the Richelot isogenies on the Jacobian of $y^2 = x^5 + x$. This Jacobian has the amusing property that there are 3 such isogenies and that each of the types of codomain (Jacobian, Weil restriction, product of elliptic curves) is represented.

```
> R<x>:=PolynomialRing(Rationals());
> C:=HyperellipticCurve(x^5+x);
> J:=Jacobian(C);
> RichelotIsogenousSurfaces(J);
[*
    Cartesian Product<Elliptic Curve defined by y^2 = x^3 + 5/32*x^2 -
    5/1024*x - 1/32768 over Rational Field, Elliptic Curve defined by
    y^2 = x^3 - 5/32*x^2 - 5/1024*x + 1/32768 over Rational Field>,
    Jacobian of Hyperelliptic Curve defined by y^2 = -2*x^5 - 2*x over
    Rational Field,
    Elliptic Curve defined by y^2 = x^3 + 5/32*$.1*x^2 + 5/1024*x +
        1/32768*$.1 over Number Field with defining polynomial x^2 + 1
    over the Rational Field
*]
```

We now illustrate how the kernels are represented.

```
> codomains,kernels:=RichelotIsogenousSurfaces(J:Kernels);
> Q:=kernels[1];
> LX<X>:=Parent(Q);
> L<alpha>:=BaseRing(LX);
> Q;
(-1/2*alpha^2 + 2*alpha)*X^2 + (-1/2*alpha^2 + alpha + 1)*X -
    1/2*alpha^2 + 2*alpha
> L;
Univariate Quotient Polynomial Algebra in alpha over Rational Field
with modulus alpha^3 - 4*alpha^2 + 2*alpha
```

Let us check that the norm of `Q` gives us $x^5 + x$ again and that calling *RichelotIsogenousSurface* allows us to recreate the corresponding codomain.

```
> _,swp:=SwapExtension(LX);
> Norm(swp(Q));
x^5 + x
```

We can use Q to recreate the corresponding codomain.

```
> codomains[1] eq RichelotIsogenousSurface(J,Q);
true
```

Finally, to verify that the computed abelian surfaces are all isogenous, we verify that their L-series over $\mathbb{Q}$ are equal. For each type of return value we have to create the L-Series in a slightly different way, but once done, we can easily check that their coefficients agree.

```
> LC:=LSeries(C : LocalData:="Ogg");
> myL:=func< A|
>     case<Type(A) | SetCart : LSeries(A[1])*LSeries(A[2]),
>                    JacHyp  : LSeries(Curve(A) : LocalData:="Ogg"),
>                    CrvEll  : LSeries(A),
>                    default : false>>;
> cfs:=[c: c in LGetCoefficients(LC,1000)];
> [[c: c in LGetCoefficients(myL(A),1000)] eq cfs : A in codomains];
[ true, true, true ]
```

`DoubleRichelotIsogenies(J): JacHyp -> SeqEnum`

Given a genus 2 Jacobian J, this finds Jacobians J' related to J by a "double Richelot isogeny", i.e., an isogeny with kernel isomorphic to $\mathbb{Z}/2 \times \mathbb{Z}/2 \times \mathbb{Z}/4$, but which does not factor into two Richelot isogenies over $\mathbb{Q}$. A sequence containing these Jacobians J' is returned.

`DoubleRichelotIsogenies(C): CrvHyp -> SeqEnum`

This finds curves C' such that the Jacobians of (genus 2) C and C' have a "double Richelot isogeny" between them, i.e., an isogeny with kernel isomorphic to $\mathbb{Z}/2 \times \mathbb{Z}/2 \times \mathbb{Z}/4$, but which does not factor into two Richelot isogenies over $\mathbb{Q}$. A sequence containing these curves C' is returned.

`TwoPowerIsogenies(J): JacHyp -> SeqEnum, SeqEnum, SeqEnum`

For a genus 2 Jacobian J over $\mathbb{Q}$, determine (hopefully) all principally polarized Jacobians and some other principally polyarized abelian surfaces over $\mathbb{Q}$ up to isomorphism that are isogenous to J (with compatible polarization) by an isogeny of degree a power of 2. This returns a sequence whose elements are isogenous Jacobians (other than J), a sequence whose elements are products of two elliptic curves over $\mathbb{Q}$ and a sequence whose elements are elliptic curves over quadratic fields (whose restriction of scalars down to $\mathbb{Q}$ are isogenous to J).

Example: Double Richelot Isogenies (ex-bdb4ad)

We consider the Jacobian of the curve $y^2 + (x^3 + 1)y = x^2 + x$ (this is curve 249.a.249.1 in the LMFDB). There are no Richelot isogenous abelian surfaces, but there are some double Richelot isogenous Jacobians.

```
> R<x>:=PolynomialRing(Rationals());
> C:=HyperellipticCurve(x^2+x, x^3+1);
> J:=Jacobian(C);
> RichelotIsogenousSurfaces(J);
[* *]
> DoubleRichelotIsogenies(J);
[
    Jacobian of Hyperelliptic Curve defined by y^2 = -15*x^6 - 220*x^5 - 960*x^4
        - 770*x^3 + 1780*x^2 - 780*x + 105 over Rational Field,
    Jacobian of Hyperelliptic Curve defined by y^2 = x^6 - 4*x^5 + 6*x^3 + 4*x^2
        + 12*x + 9 over Rational Field
]
```

Now we look at the curve $y^2 = (x-2)(x-1)x(x+1)(x+2)$ and its Jacobian. The Jacobian has full rational 2-torsion, so there are many Richelot isogenies. There are no double Richelot isogenies, but there are some additional 2-power isogenous abelian surfaces.

```
> R<x>:=PolynomialRing(Rationals());
> C:=HyperellipticCurve((x-2)*(x-1)*x*(x+1)*(x+2));
> J:=Jacobian(C);
> RichelotIsogenousSurfaces(J);
[*
    Jacobian of Hyperelliptic Curve defined by y^2 = -40*x^6 + 280*x^5 - 60*x^4
        - 1520*x^3 - 120*x^2 + 1120*x - 320 over Rational Field,
    Jacobian of Hyperelliptic Curve defined by y^2 = -10*x^6 + 70*x^5 - 60*x^4 -
        170*x^3 - 210*x^2 - 80*x + 40 over Rational Field,
    Jacobian of Hyperelliptic Curve defined by y^2 = -8*x^5 + 60*x^4 - 80*x^3 -
        120*x^2 - 32*x over Rational Field,
    Jacobian of Hyperelliptic Curve defined by y^2 = -35*x^6 + 112*x^5 + 105*x^4
        - 350*x^3 + 210*x^2 + 448*x - 280 over Rational Field,
    Jacobian of Hyperelliptic Curve defined by y^2 = -15*x^6 + 40*x^5 - 15*x^4 +
        10*x^3 + 30*x^2 + 160*x + 120 over Rational Field,
    Jacobian of Hyperelliptic Curve defined by y^2 = -5*x^6 - 20*x^5 + 105*x^4 -
        170*x^3 + 120*x^2 + 280*x + 80 over Rational Field,
    Elliptic Curve defined by y^2 = x^3 - 35/256*$.1*x^2 + 35/32768*x -
        1/8388608*$.1 over Number Field with defining polynomial x^2 - 2 over
    the Rational Field,
    Elliptic Curve defined by y^2 = x^3 + 5/256*$.1*x^2 + 45/32768*x +
        729/8388608*$.1 over Number Field with defining polynomial x^2 + 2 over
    the Rational Field,
    Jacobian of Hyperelliptic Curve defined by y^2 = -18*x^5 - 90*x^3 - 72*x
    over Rational Field,
    Jacobian of Hyperelliptic Curve defined by y^2 = 35*x^6 + 112*x^5 - 105*x^4
        - 350*x^3 - 210*x^2 + 448*x + 280 over Rational Field,
    Jacobian of Hyperelliptic Curve defined by y^2 = 15*x^6 + 40*x^5 + 15*x^4 +
        10*x^3 - 30*x^2 + 160*x - 120 over Rational Field,
    Jacobian of Hyperelliptic Curve defined by y^2 = 5*x^6 - 20*x^5 - 105*x^4 -
        170*x^3 - 120*x^2 + 280*x - 80 over Rational Field,
    Jacobian of Hyperelliptic Curve defined by y^2 = 40*x^6 + 280*x^5 + 60*x^4 -
        1520*x^3 + 120*x^2 + 1120*x + 320 over Rational Field,
    Jacobian of Hyperelliptic Curve defined by y^2 = 10*x^6 + 70*x^5 + 60*x^4 -
        170*x^3 + 210*x^2 - 80*x - 40 over Rational Field,
    Jacobian of Hyperelliptic Curve defined by y^2 = -8*x^5 - 60*x^4 - 80*x^3 +
        120*x^2 - 32*x over Rational Field
```

(continues on next page)

(continued from previous page)

```
*]
> DoubleRichelotIsogenies(J);
[]
> TwoPowerIsogenies(J);
[
    Jacobian of Hyperelliptic Curve defined by y^2 = -10*x^6 + 70*x^5 - 15*x^4 -
        380*x^3 - 30*x^2 + 280*x - 80 over Rational Field,
    Jacobian of Hyperelliptic Curve defined by y^2 = -10*x^6 + 70*x^5 - 60*x^4 -
        170*x^3 - 210*x^2 - 80*x + 40 over Rational Field,
    Jacobian of Hyperelliptic Curve defined by y^2 = -2*x^5 + 15*x^4 - 20*x^3 -
        30*x^2 - 8*x over Rational Field,
    Jacobian of Hyperelliptic Curve defined by y^2 = -35*x^6 + 112*x^5 + 105*x^4
        - 350*x^3 + 210*x^2 + 448*x - 280 over Rational Field,
    Jacobian of Hyperelliptic Curve defined by y^2 = -15*x^6 + 40*x^5 - 15*x^4 +
        10*x^3 + 30*x^2 + 160*x + 120 over Rational Field,
    Jacobian of Hyperelliptic Curve defined by y^2 = 5*x^6 - 35*x^5 + 30*x^4 +
        85*x^3 + 105*x^2 + 40*x - 20 over Rational Field,
    Jacobian of Hyperelliptic Curve defined by y^2 = x^5 + 5*x^3 + 4*x over
    Rational Field,
    Jacobian of Hyperelliptic Curve defined by y^2 = 35*x^6 + 112*x^5 - 105*x^4
        - 350*x^3 - 210*x^2 + 448*x + 280 over Rational Field,
    Jacobian of Hyperelliptic Curve defined by y^2 = 15*x^6 + 40*x^5 + 15*x^4 +
        10*x^3 - 30*x^2 + 160*x - 120 over Rational Field,
    Jacobian of Hyperelliptic Curve defined by y^2 = -5*x^6 - 35*x^5 - 30*x^4 +
        85*x^3 - 105*x^2 + 40*x + 20 over Rational Field,
    Jacobian of Hyperelliptic Curve defined by y^2 = 10*x^6 + 70*x^5 + 15*x^4 -
        380*x^3 + 30*x^2 + 280*x + 80 over Rational Field,
    Jacobian of Hyperelliptic Curve defined by y^2 = 10*x^6 + 70*x^5 + 60*x^4 -
        170*x^3 + 210*x^2 - 80*x - 40 over Rational Field,
    Jacobian of Hyperelliptic Curve defined by y^2 = -2*x^5 - 15*x^4 - 20*x^3 +
        30*x^2 - 8*x over Rational Field,
    Jacobian of Hyperelliptic Curve defined by y^2 = 105*x^6 + 20440*x^5 -
        6930*x^4 - 81760*x^3 - 13860*x^2 + 81760*x + 840 over Rational Field,
    Jacobian of Hyperelliptic Curve defined by y^2 = 45*x^6 - 680*x^5 + 2970*x^4
        - 2720*x^3 - 5940*x^2 - 2720*x - 360 over Rational Field,
    Jacobian of Hyperelliptic Curve defined by y^2 = -105*x^6 + 20440*x^5 +
        6930*x^4 - 81760*x^3 + 13860*x^2 + 81760*x - 840 over Rational Field,
    Jacobian of Hyperelliptic Curve defined by y^2 = -45*x^6 - 680*x^5 -
        2970*x^4 - 2720*x^3 + 5940*x^2 - 2720*x + 360 over Rational Field
]
[]
[
    Elliptic Curve defined by y^2 = x^3 - 35/256*$.1*x^2 + 35/32768*x -
        1/8388608*$.1 over Number Field with defining polynomial x^2 - 2 over
    the Rational Field,
    Elliptic Curve defined by y^2 = x^3 + 5/256*$.1*x^2 + 45/32768*x +
        729/8388608*$.1 over Number Field with defining polynomial x^2 + 2 over
    the Rational Field,
    Elliptic Curve defined by y^2 = x^3 + 1/64*(-635*$.1 - 285)*x^2 +
    1/4096*(-29350*$.1 + 41725)*x + 1/262144*(-625*$.1 + 875) over Number Field
    with defining polynomial x^2 - 2 over the Rational Field,
    Elliptic Curve defined by y^2 = x^3 + 1/128*(415*$.1 + 930)*x^2 +
    1/8192*(28650*$.1 + 45075)*x + 1/1048576*(455625*$.1 + 182250) over Number
    Field with defining polynomial x^2 + 2 over the Rational Field,
    Elliptic Curve defined by y^2 = x^3 + 1/128*(-1645*$.1 + 2310)*x^2 +
    1/8192*(-114170*$.1 - 161455)*x + 1/1048576*(-2401*$.1 - 3430) over Number
```

(continues on next page)

(continued from previous page)

```
    Field with defining polynomial x^2 - 2 over the Rational Field,
    Elliptic Curve defined by y^2 = x^3 - 5/256*$.1*x^2 - 45/32768*x +
        729/8388608*$.1 over Number Field with defining polynomial x^2 - 2 over
    the Rational Field,
    Elliptic Curve defined by y^2 = x^3 + 35/256*$.1*x^2 - 35/32768*x -
        1/8388608*$.1 over Number Field with defining polynomial x^2 + 2 over
    the Rational Field,
    Elliptic Curve defined by y^2 = x^3 + 1/128*(415*$.1 - 930)*x^2 +
    1/8192*(-28650*$.1 + 45075)*x + 1/1048576*(455625*$.1 - 182250) over Number
    Field with defining polynomial x^2 + 2 over the Rational Field
]
```

137.8 Points on the Jacobian

Points on $Jac(C)$ are represented as divisors on C. They can be specified simply by giving points on C, or divisors on C, or in the Mumford representation, which is the way Magma returns them (and which it uses to store and manipulate them). Points can be added and subtracted when the base field is an exact field.

Representation of points on $Jac(C)$: Let C be a hyperelliptic curve of genus g. A triple $\langle a(x), b(x), d\rangle$ specifies the divisor D of degree d on C defined by

$$A(x, z) = 0, \quad y = B(x, z)$$

where $A(x, z)$ is the degree d homogenisation of $a(x)$, and $B(x, z)$ is the degree $(g+1)$ homogenisation of $b(x)$. Note that the equation $y = B(x, z)$ makes sense projectively because y has weight $g+1$ (as always for hyperelliptic curves in Magma).

The *point on $Jac(C)$ corresponding to* $\langle a(x), b(x), d\rangle$ is then D minus a multiple of the divisor at infinity on C. So, when there is a single point P_∞ at infinity, we have $D - dP_\infty$. Otherwise, there is a $\mathbb{Q}$-rational divisor $P_{+\infty} + P_{-\infty}$ consisting of the two points at infinity; in this case d is required to be even and we have $D - (d/2)(P_{+\infty} + P_{-\infty})$.

All points on $Jac(C)$ can be expressed in this way, except when g is odd and there are no rational points at infinity (in which case the extra points can not be created in Magma, and arithmetic on points is not implemented). There is a uniquely determined "reduced" triple representing each point, which Magma uses to represent any point it encounters.

See the examples in the following section ("Creation of Points").

Technical details: In order to make sense, a triple $\langle a(x), b(x), d\rangle$ is required to satisfy:

(a) $a(x)$ is monic of degree at most g;

(b) $b(x)$ has degree at most $g + 1$, and $a(x)$ divides $b(x)^2 + h(x)b(x) - f(x)$, where $h(x)$ and $f(x)$ are the defining polynomials of C;

(c) d is a positive integer with $\deg(a(x)) \leq d \leq g + 1$, such that the degree of $b(x)^2 + h(x)b(x) - f(x)$ is less than or equal to $2g + 2 - d + \deg(a(x))$.

For uniqueness of representation, in the case of one point at infinity, (the odd degree case, in particular), we require that $d = \deg(a(x))$.

A triple $\langle a(x), b(x), d\rangle$ is reduced to a canonical representative as follows:

(a) If $d = \deg(a(x))$, we reduce b mod a, so $\deg(b(x)) < \deg(a(x))$.

(b) If $a = 1$, we assume that $\deg(B(1, z)) = d$.

(c) A unique representative for $b(x)$ is found such that the coefficient of x^k in b is zero for $\deg(a(x)) \leq k \leq \deg(a(x)) + g + 1 - d$.

For certain models of C, not all rational points can be represented in the above form (with the restrictions on d) or the representation is not unique. The bad cases are g odd and (i) 0 or (ii) 2 rational points at infinity.

In case (i) we would have to allow divisors with $\deg(a(x)) = g+1$, which give linear pencils of equivalent divisors, and in case (ii) a Jacobian point with $d = g + 1$ has precisely two canonical representatives.

In case (i), there is no obvious canonical representative for these additional Jacobian points and currently Magma does not deal with them. Consequently, arithmetic is not implemented in this case.

However, in case (ii), the two representatives correspond to the two distinguished elements of a linear pencil of divisors which include contributions from one or the other point at infinity. By arbitrarily selecting one of these two infinite points as the default, a unique representative can be chosen. This is now performed internally by Magma, the default point being chosen at the time of creation of the Jacobian. Arithmetic is also implemented. Note that in this case, extra work is generally required in point addition to keep track of points at infinity and final reduction to the unique representation. If very fast addition is crucial when C is of this type, it is generally wise to use an isomorphic model with exactly one point at infinity, if possible, by moving a rational Weierstrass point to infinity.

137.8.1 Creation of Points

`J ! 0: JacHyp, RngIntElt -> JacHypPt`

`Id(J): JacHyp -> JacHypPt`

`Identity(J): JacHyp -> JacHypPt`

The identity element on the Jacobian J.

`J ! [a, b]: JacHyp, [ RngUPolElt ] -> JacHypPt`

`elt< J | a, b >: JacHyp, RngUPolElt, RngUPolElt -> JacHypPt`

`elt< J | [a, b] >: JacHyp, [ RngUPolElt ] -> JacHypPt`

`elt< J | a, b, d >: JacHyp, RngUPolElt, RngUPolElt, RngIntElt -> JacHypPt`

`elt< J | [a, b], d >: JacHyp, [ RngUPolElt ], RngIntElt -> JacHypPt`

The point on the Jacobian J defined by the polynomials a and b and the positive integer d; if not specified then d is taken to be $\deg(a)$.

`P - Q: PtHyp, PtHyp -> JacHypPt`

`J ! [P, Q]: JacHyp, [PtHyp] -> JacHypPt`

`elt< J | P, Q >: JacHyp, PtHyp, PtHyp -> JacHypPt`

For points P and Q on a hyperelliptic curve, this constructs the image of the divisor class $[P - Q]$ as a point on its Jacobian J.

`J ! [S, T]: JacHyp, [SeqEnum] -> JacHypPt`

`elt< J | S, T >: JacHyp, [PtHyp], [PtHyp] -> JacHypPt`

Given two sequences $S = [P_i]$ and $T = [Q_i]$ of points on the hyperelliptic curve with Jacobian J, each of length n, this returns the image of the divisor class $\sum_i [P_i] - \sum_i [Q_i]$ as a point on the Jacobian J.

`JacobianPoint(J, D): JacHyp, DivCrvElt -> JacHypPt`

The point on the Jacobian J (of a hyperelliptic curve C) associated to the divisor D on C. If D does not have degree 0, then a suitable multiple of the divisor at infinity is subtracted. When the divisor at infinity on C has even degree, D is required to have even degree.

The function works for any divisor such that the corresponding point is definable in Magma. It is not implemented for characteristic 2.

`J ! P: JacHyp, JacHypPt -> JacHypPt`

Given a point P on a Jacobian J', construct the image of P on J, where J is a base extension of J'.

`Points(J, a, d): JacHyp, RngUPolElt, RngIntElt -> SetIndx`

`RationalPoints(J, a, d): JacHyp, RngUPolElt, RngIntElt -> SetIndx`

Find all points on the Jacobian J with first component a and degree d. Only implemented for genus 2 curves of the form $y^2 = f(x)$.

Example: Points on a Curve and Its Jacobian (ex-11fedb)

Points on $y^2 = x^6 - 3x - 1$ and their images on the Jacobian.

```
> _<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(x^6-3*x-1);
> C;
Hyperelliptic Curve defined by y^2 = x^6 - 3*x - 1 over Rational Field
> J := Jacobian(C);
> J;
Jacobian of Hyperelliptic Curve defined by y^2 = x^6 - 3*x - 1 over Rational␣
↪Field
```

Find some points on C and map them to J (using the first point to define the map $C \to J$):

```
> ptsC := Points(C : Bound := 100);
> ptsC;
{@ (1 : -1 : 0), (1 : 1 : 0), (-1 : -1 : 3), (-1 : 1 : 3) @}
> ptsJ := [ ptsC[i] - ptsC[1] : i in [2,3,4] ];
> ptsJ;
[ (1, x^3, 2), (x + 1/3, x^3, 2), (x + 1/3, x^3 + 2/27, 2) ]
```

We recreate the first of these, giving it in Magma's notation (which is read as "the divisor of degree 2 on C determined by $z^2 = y - x^3 = 0$").

```
> pt1 := elt< J | [1,x^3], 2 >;
> pt1 eq ptsJ[1];
true
```

The degree of the divisor must be specified here, otherwise it is assumed to be $\deg(a(x)) = 0$:

```
> pt1 := J! [1,x^3];
> pt1; pt1 eq J!0;
(1, 0, 0)
true
```

Example: Nontrivial 2-Torsion Point (ex-837c1c)

We define the nontrivial 2-torsion point on the Jacobian of $C : y^2 = (x^2 + 1)(x^6 + 7)$.

The divisor $(y = 0, x^2 + 1 = 0)$ should define the only nontrivial 2-torsion point in $J(\mathbb{Q})$.

```
> _<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve( (x^2+1)*(x^6+7) );
> J := Jacobian(C);
```

Define the point corresponding to the divisor given by $x^2 + 1 = 0, y = 0$ on C. We can use the simpler syntax, not specifying that the degree of the divisor is 2, because this equals the degree of $a(x) = x^2 + 1$.

```
> Ptors := J![x^2+1, 0];
> Ptors;
(x^2 + 1, 0, 2)
```

The alternative syntax would be as follows.

```
> Ptors1 := elt< J | [x^2+1, 0], 2 >;
> Ptors eq Ptors1;  // Are they the same?
true
```

Check that Ptors has order 2:

```
> 2*Ptors;
(1, 0, 0)
> $1 eq J!0;  // Is the previous result really the trivial point on J?
true
> Order(Ptors);  // Just to be absolutely sure ...
2
```

The other factor $x^6 + 7$ will give the same point on J:

```
> Ptors2 := J![x^6+7,0];
(x^2 + 1, 0, 2)
```

Note that Magma returned the unique reduced triple representing this point, which means we can easily check whether or not it is the same point as `Ptors`.

For alternative ways to obtain 2-torsion points, see the section on torsion.

Example: Jacobian Point Not from the Curve (ex-e19fa0)

A Jacobian with a point not coming from the curve.

Any curve containing the point $(\sqrt{2}, \sqrt{2})$ has a $\mathbb{Q}$-rational divisor

$$D := (\sqrt{2}, \sqrt{2}) + (-\sqrt{2}, -\sqrt{2}).$$

This will give a nontrivial point in $J(\mathbb{Q})$. For instance, take $y^2 = x^6 - 6$.

```
> _<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(x^6-6);
> J := Jacobian(C);
```

The divisor D has degree 2 and is given by $x^2 - 2 = y - x = 0$.

```
> pt := J![x^2-2, x];
> pt;  // What is pt?
(x^2 - 2, x, 2)
> Parent(pt);  // Where does pt live?
Jacobian of Hyperelliptic Curve defined by y^2 = x^6 - 6 over Rational Field
> pt eq J!0;  // Is pt equal to 0 on J?
false
> Order(pt);
0
```

This means that P has infinite order in $J(\mathbb{Q})$.

Alternatively we can construct the same point by first constructing the divisor, giving it as an ideal in the homogeneous coordinate ring in which C lives. (Note that Y has weight 3).

```
> P<X,Y,Z> := CoordinateRing(Ambient(C));
> D := Divisor(C, ideal<P | X^2-2*Z^2, Y-X*Z^2> );
> pt1 := J!D;
> pt eq pt1;
true
```

137.8.2 Random Points

`Random(J): JacHyp -> JacHypPt`

A random point on a Jacobian J of a hyperelliptic curve defined over a finite field.

137.8.3 Booleans and Predicates for Points

For each of the following functions, P and Q are points on the Jacobian of a hyperelliptic curve.

`P eq Q: JacHypPt, JacHypPt -> BoolElt`

Returns `true` if and only if the points P and Q on the same Jacobian are equal.

`P ne Q: JacHypPt, JacHypPt -> BoolElt`

Returns `false` if and only if the two points P and Q on the same Jacobian are equal.

`IsZero(P): JacHypPt -> BoolElt`

`IsIdentity(P): JacHypPt -> BoolElt`

Returns `true` if and only if P is the zero element of the Jacobian.

137.8.4 Access Operations

`P[i]: JacHypPt, RngIntElt -> RngElt`

Given an integer $1 \leq i \leq 2$, this returns the i-th defining polynomial for the Jacobian point P. For $i = 3$, this returns the degree d of the associated reduced divisor.

`Eltseq(P): PtHyp -> SeqEnum, RngIntElt`

`ElementToSequence(P): PtHyp -> SeqEnum, RngIntElt`

Given a point P on the Jacobian J of a hyperelliptic curve, the function returns firstly, a sequence containing the two defining polynomials for the divisor associated to P and secondly, the degree of the divisor.

137.8.5 Arithmetic of Points

For each of the following functions, P and Q are points on the Jacobian of a hyperelliptic curve. The arithmetic operators on points can be called for curves over any base field, but the computations will be precise only for exact fields (i.e., *not* reals, complexes, p-adics, or derivatives of these) so should really only be used for such fields.

`- P: JacHypPt -> JacHypPt`

The additive inverse of the point P on the Jacobian.

`P + Q: JacHypPt, JacHypPt -> JacHypPt`

The sum of the points P and Q on the same Jacobian.

`P +:= Q: JacHypPt, JacHypPt`

Given two points P and Q on the same Jacobian, set P equal to their sum.

`P - Q: JacHypPt, JacHypPt -> JacHypPt`

The difference of the points P and Q on the same Jacobian.

`P -:= Q: JacHypPt, JacHypPt`

Given two points P and Q on the same Jacobian, set P equal to the difference $P - Q$.

`n * P: RngIntElt, JacHypPt -> JacHypPt`

`P * n: JacHypPt, RngIntElt -> JacHypPt`

The n-th multiple of P in the group of rational points on the Jacobian.

`P *:= n: JacHypPt, RngIntElt`

Set P equal to the n-th multiple of itself.

137.8.6 Order of Points on the Jacobian

`Order(P): JacHypPt -> RngIntElt`

Returns the order of the point P on the Jacobian J of a hyperelliptic curve defined over a finite field, the rationals, or an algebraic number field, or 0 if P has infinite order. This first computes $\#J$ when J is defined over a finite field.

`Order(P, l, u): JacHypPt, RngIntElt, RngIntElt -> RngIntElt`

```
Alg          : MonStg                    Default: "Shanks"
UseInversion: BoolElt                    Default: true
```

Returns the order of the point P where u and l are bounds for the order of P or for the order of J the parent of P. This does not compute $\#J$.

If the parameter `Alg` is set to `"Shanks"` then the generic Shanks algorithm is used, otherwise, when `Alg` is `"PollardRho"`, a Pollard-Rho variant is used (see [GH]).

If `UseInversion` is `true` the search space is halved by using point negation.

`Order(P, l, u, n, m): JacHypPt, RngIntElt, RngIntElt, RngIntElt, RngIntElt -> RngIntElt`

```
Alg          : MonStg                    Default: "Shanks"
UseInversion: BoolElt                    Default: true
```

Returns the order of the point P where u and l are bounds for the order of P or for the order of J the parent of P and where n and m are such that the group order is n mod m. This does not compute $\#J$.

The two parameters `Alg` and `UseInversion` have the same use as in the previous function.

`HasOrder(P, n): JacHypPt, RngIntElt -> BoolElt`

Given a point P on the Jacobian J of a hyperelliptic curve and a positive integer n, this returns `true` if the order of the point is n.

137.8.7 Frobenius

`Frobenius(P, k): JacHypPt, FldFin -> JacHypPt`

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

Given a point P that lies on the Jacobian J of a hyperelliptic curve that is defined over the finite field $k = F_q$, determine the image of P under the Frobenius map $x \to x^q$. If `Check` is `true`, Magma verifies that the Jacobian of P is defined over k.

137.8.8 Weil Pairing

`WeilPairing(P, Q, m): JacHypPt, JacHypPt, RngIntElt -> RngElt`

Computes the Weil pairing of P and Q, where P and Q are m-torsion points on the 2-dimensional Jacobian J defined over a finite field.

Example: MOV Reduction on a Jacobian (ex-3d8114)

The following illustrates the use of the Weil Pairing in the MOV-reduction of the discrete logarithm problem on a Jacobian.

```
> PP<x>:=PolynomialRing(GF(2));
> h := PP!1;
> f := x^5 + x^4 + x^3 + 1;
> J := Jacobian(HyperellipticCurve(f,h));  // a supersingular curve
> Jext := BaseExtend(J, 41);
```

(continues on next page)

(continued from previous page)

```
> Factorization(#Jext);
[ <7, 1>, <3887047, 1>, <177722253954175633, 1> ]
> m := 177722253954175633;                  // some big subgroup order
> cofact := 3887047*7;
> P := cofact*Random(Jext);
> Q := 876213876263897634*P;                // Q in <P>
```

Say we want to recompute the logarithm of Q in base P.

```
> Jext2 := BaseExtend(Jext, 6);             // go to an ext of deg 6
> NJ := #Jext2;
>
> R := Random(Jext2);
> R *:= NJ div m^Valuation(NJ, m);
>
> eP := WeilPairing(Jext2!P, R, m);
> eQ := WeilPairing(Jext2!Q, R, m);
> assert eP^876213876263897634 eq eQ;
```

So the discrete log problem on the Jacobian has been reduced to a discrete log problem in a finite field.

137.9 Rational Points and Group Structure over Finite Fields

137.9.1 Enumeration of Points

Points(J): JacHyp -> SetIndx

RationalPoints(J): JacHyp -> SetIndx

Given a Jacobian J of a hyperelliptic curve defined over a finite field, determine all rational points on the Jacobian J.

137.9.2 Counting Points on the Jacobian

Several algorithms are used to compute the order of a Jacobian depending of its size, its genus, and its type. In particular, in genus 2 it includes all the techniques described in [GH]. The best current algorithms are the ones based on p-adic liftings. These are the defaults when they apply (see below) and the characteristic is not too large. In odd characteristic, Kedlaya's algorithm is used as described in [Ked01]. For characteristic 2, we have Mestre's canonical lift method as adapted by Lercier and Lubicz. Details can be found in [LL06]. We also now have an implementation of Vercauteren's characteristic 2 version of Kedlaya (described in [Ver02]). As an added bonus, the p-adic methods actually give the Euler factor (see below). A verbose flag can be set to see which strategy is chosen and the progress of the computation.

The latest p-adic methods, faster than Kedlaya in odd characteristic, are those of Alan Lauder based on deformations over parametrised families. Magma incorporates an implementation for hyperelliptic families by Hendrik Hubrechts (see [Hub06] for details). This has a slightly different interface to the other methods (which all run through the same top-level intrinsics) as it can be used to count points on several members of the family at once.

SetVerbose("JacHypCnt", v): MonStgElt, RngIntElt

Set the verbose printing level for the point-counting on the Jacobians. Currently the legal values for v are `true`, `false`, 0, 1, 2, 3, or 4 (`false` is the same as 0, and `true` is the same as 1).

J: JacHyp -> RngIntElt

Order(J): JacHyp -> RngIntElt

```
NaiveAlg          : BoolElt                    Default: false
ShanksLimit       : RngIntElt                  Default: 10^12
CartierManinLimit: RngIntElt                   Default: 5 × 10^5
UseSubexpAlg      : BoolElt                    Default: true
UseGenus2         : BoolElt                    Default: false
UseSchoof         : BoolElt                    Default: true
UseHalving        : BoolElt                    Default: true
```

Given the Jacobian J of a hyperelliptic curve defined over a finite field, determine the order of the group of rational points.

There are 4 optional parameters which concern every genus.

The parameter `NaiveAlg` can be set to one if one want to use the naive algorithm for point counting, consisting in counting points on the curve over the first g extensions, where g is the genus. Setting this parameter to `true` annihilates the effects of the others. By default, this strategy is used for small groups and Jacobians in characteristic 2 for which no group law is available.

In odd characteristic, Kedlaya's algorithm can be applied to curves of any genus and it is the default when the group is not too small or the characteristic too large.

In contrast, Mestre's characteristic 2 method has several conditions on its applicability. When these are satisfied and the genus is not too large or the group order too small, it is the default.

- Firstly, the Jacobian must be *ordinary*. This is equivalent to the defining polynomial $h(x)$ having degree g or $g+1$ and having distinct roots (ie non-zero discriminant).
- Next, there must be a group law available for a final elimination: the algorithm produces several possibilities for the characteristic polynomial of Frobenius. If no group law is available, we attempt to replace the curve with an isomorphic one for which one is. Note that a group law exists if there is a single point at infinity (degree of $h(x) = g$).

Warning : Even if these conditions are satisfied, the last phase of the algorithm may still fail for $g \geq 4$ in unusual cases. However, this is very rare. Finally, it should be noted that the main algorithm works over the smallest extension of the ground field K over which $h(x)$ splits. It is therefore much faster if $h(x)$ has all its roots in K.

When the genus is > 4 or the canonical lift method doesn't apply, Vercauteren's version of Kedlaya is now used in its place. It is also used for genus 4 when an extension of the base field is required for Mestre: ie, $h(x)$ doesn't split.

For genus 2 the methods mentioned at the start are applied if the parameter `UseGenus2` is set to `true`. Otherwise they are now only used for large characteristic.

If none of the above apply, the remaining methods are the subexponential algorithm, which uses the generic machinery of the function fields, and general group order algorithms of Shanks and Pollard. The subexponential algorithm will be the first choice if no group law is available or the genus is large and the base field small. It can be disabled by setting `UseSubexpAlg` to `false`.

The parameter `ShanksLimit` determines when we switch from a memory-consuming Shanks' algorithm to a slower Pollard's method. The value of the parameter `CartierManinLimit` is the maximal characteristic for which we may use the Cartier-Manin trick (see function `EulerFactorModChar` below).

The next 2 parameters are used only for the genus 2 methods in odd characteristic.

When the base field is large enough, it is better to firstly compute the cardinality of the Jacobian modulo some odd primes and some power of two. These two parameters allow the user to disable one or both of these methods.

Example: Direct Point Counting `(ex-be83bf)`

In the following sequence of examples the different point counting methods are illustrated. The first example uses direct counting.

```
> SetVerbose("JacHypCnt",true);
> P<x> := PolynomialRing(GF(31));
> f := x^8 + 3*x^7 + 2*x^6 + 23*x^5 + 5*x^4 + 21*x^3 + 29*x^2 + 12*x + 9;
> J := Jacobian(HyperellipticCurve(f));
> time #J;
Using naive counting algorithm.
20014
Time: 0.020
```

Example: Kedlaya's Algorithm (ex-ff9fd7)

For the second example, we apply Kedlaya's algorithm to a genus curve 2 over $GF(3^{20})$.

```
> K := FiniteField(3,20);
> P<x> := PolynomialRing(K);
> f :=  x^5 + x + K.1;
> J := Jacobian(HyperellipticCurve(f));
> #J;
Using Kedlaya's algorithm.
Applying Kedlaya's algorithm
Total time: 2.310
12158175772023384771
```

Example: Vercauteren's Algorithm (ex-23c874)

For the third example, we apply Vercauteren's algorithm to a non-ordinary genus curve 2 over $GF(2^{25})$..

```
> K := FiniteField(2,25);
> P<x> := PolynomialRing(K);
> f :=  x^5 + x^3 + x^2 + K.1;
> J := Jacobian(HyperellipticCurve([f,K!1]));
> time #J;
Using Kedlaya/Vercauteren's algorithm.
1125899940397057
Time: 0.680
```

Example: Mestre's Algorithm (ex-5fabdd)

For the third example, we apply Mestre's algorithm to a genus 3 ordinary curve over $GF(2^{25})$.

```
> K := FiniteField(2,25);
> P<x> := PolynomialRing(K);
> h := x*(x+1)*(x+K.1);
> f :=  x^7 + x^5 + x + K.1;
> C := HyperellipticCurve([f,h]);
> J := Jacobian(C);
> #J;
Using Mestre's method.
Total time: 0.440
3778001771268503789504 0
> SetVerbose("JacHypCnt",false);
```

As the full Euler factor of J has been computed and stored the number of points on C, the Zeta function for C, and the Euler factor for C are now immediate.

```
> time #C;
33555396
Time: 0.000
```

Example: Comparing Shanks and Pollard (ex-2b4829)

We compare the Shanks and Pollard methods for large prime fields.

```
> // Comparison between Shanks and Pollard:
> P<x> := PolynomialRing(GF(1000003));
> f := x^7 + 123456*x^6 + 123*x^5 + 456*x^4 + 98*x^3 + 76*x^2 + 54*x + 32;
> J := Jacobian(HyperellipticCurve(f));
> curr_mem := GetMemoryUsage();
> ResetMaximumMemoryUsage();
> time Order(J : ShanksLimit := 10^15);
1001800207033014252
Time: 19.140
> GetMaximumMemoryUsage() - curr_mem;
133583360
```

The computation took about 100 MB of central memory.

```
> delete J`Order;   // clear the result which has been stored
> curr_mem := GetMemoryUsage();
> ResetMaximumMemoryUsage();
> time Order(J : ShanksLimit := 0);
1001800207033014252
Time: 95.670
> GetMaximumMemoryUsage() - curr_mem;
0
```

Now it takes almost no memory, but it is slower (and runtime may vary a lot).

Example: Schoof and Halving in Genus 2 (ex-118ae9)

In the case of genus 2 curves, the parameters `UseSchoof` and `UseHalving` do help.

```
> // Using Schoof and Halving true is generally best in genus 2
> P<x> := PolynomialRing(GF(100000007));
> f := x^5 + 456*x^4 + 98*x^3 + 76*x^2 + 54*x + 32;
> J := Jacobian(HyperellipticCurve(f));
> time Order(J);
10001648178050390
Time: 7.350
> delete J`Order;
> time Order(J: UseSchoof := false, UseHalving := false);
10001648178050390
Time: 21.080
```

But if the Jacobian is known in advance to be highly non–cyclic, it may be slightly better to switch them off. The Jacobian below is the direct product of two copies of the same supersingular elliptic curve.

```
> // ... but not always for highly non-cyclic Jacobians
> P<x> := PolynomialRing(GF(500083));
> f := x^5 + 250039*x^4 + 222262*x^3 + 416734*x^2 + 166695*x + 222259;
> J := Jacobian(HyperellipticCurve(f));
> time Order(J);
250084007056
Time: 1.920
> delete J`Order;
> time Order(J : UseSchoof:=false, UseHalving := false);
250084007056
Time: 1.870
```

`FactoredOrder(J): JacHyp -> [ <RngIntElt, RngIntElt> ]`

Given the Jacobian J of a hyperelliptic curve defined over a finite field, the function returns the factorization of the order of the group of rational points.

`EulerFactor(J): JacHyp -> RngUPolElt`

Given the Jacobian J of a hyperelliptic curve defined over a finite field, the function returns the Euler factor J, i.e., the reciprocal of the characteristic polynomial of Frobenius acting on $H^1(J)$. (see also `ZetaFunction(C)` which is essentially the same function and has the same behaviour).

`EulerFactorModChar(J): JacHyp -> RngUPolElt`

Given the Jacobian J of a hyperelliptic curve defined over a finite field, the function returns the Euler factor J modulo the characteristic of the base field. This function should not be used in high characteristic (say p should be $\leq 10^6$).

`EulerFactor(J, K): JacHyp, FldFin -> RngUPolElt`

Given a Jacobian J of a hyperelliptic curve defined over the rationals and a finite field K at which J has good reduction, the function returns the Euler factor of the base extension of J to K.

137.9.3 Deformation Point Counting

`JacobianOrdersByDeformation(Q, Y): RngMPolElt, SeqEnum -> SeqEnum`

`EulerFactorsByDeformation(Q, Y): RngMPolElt, SeqEnum -> SeqEnum`

`ZetaFunctionsByDeformation(Q, Y): RngMPolElt, SeqEnum -> SeqEnum`

These functions compute the orders of Jacobians (*resp.* Euler factors, *resp.* Zeta functions) of one or more hyperelliptic curves in a 1-parameter family over a finite field using deformation methods.

The 2-variable polynomial $Q(x, z)$ should be in variables x and z over a finite field of odd characteristic k. Currently Q must be monic of odd degree as a polynomial in x. z is the parameter and the family of curves is given by $y^2 = Q(x, z)$.

The sequence Y should contain elements in a finite field extension K of k. The function then computes and returns the sequence of orders, Euler factors, or Zeta functions associated to the hyperelliptic curves over K obtained by specialising the z parameter of $Q(x, z)$ to the elements of Y.

Over a field of size p^n, for n not too large, it is efficient to compute the results for several curves in the family at once, as roughly half of the computation for an individual curve is taken up in computing a Frobenius matrix that is then specialised to the parameter value. For a sequence of input parameter values, this part of the computation is only performed once at the start. However, as n becomes larger, the time taken for the specialisation at a particular value starts to dominate.

The field k over which the family is defined should be of small degree over the prime field. This is because Kedlaya's algorithm is applied over k to a particular member of the family with the parameter specialised to a k-value to initialise the deformation.

Similarly, the algorithm is much faster when $Q(x, z)$ is of small degree in the parameter z. In practice, the degree of $Q(x, z)$ as a polynomial in z, probably shouldn't exceed 3 or 4 (linear is obviously best).

There are two further conditions on Q and Y which we will try to remove in the near future. The first is that no Y-value is zero. The second is that the generic discriminant of $\hat{Q}$ (resultant of $\hat{Q}$ and $\partial\hat{Q}/\partial x$ w.r.t. x) must have leading coefficient (as a polynomial in z) a unit in $W(k)$. Here, $W(k)$ is the ring of integers of the unramified extension of $\mathbb{Q}_p$ with residue class field k and $\hat{Q}$ is a lift of Q to a 2-variable polynomial over $W(k)$. By translating and scaling the parameter, the user may be able to effect these two conditions if they aren't initially satisfied.

As with the other point-counting functions, the verbose flag `JacHypCnt` can be used to output information during processing.

Example: Deformation Point Counting for a Family `(ex-9780bf)`

The following is a small example where the Euler factors for 4 random members of a linear family

of elliptic curves are computed with a single call. The base field is $\mathbb{F}_9$ and the parameter values are taken in $\mathbb{F}_{3^{40}}$ which is the field over which the counting occurs.

```
> Fp := FiniteField(3^2);
> R<X,Y> := PolynomialRing(Fp,2);
> Q  := X^3+X^2-2*X+Fp.1+Y*(X^2-X+1);
> Fq := ext<Fp | 20>;
> values := [Random(Fq) : i in [1..4]];
> EFs := EulerFactorsByDeformation(Q,values);
> EFs;
[
    1215766545905692880 1*$.1^2 + 852459373*$.1 + 1,
    1215766545905692880 1*$.1^2 + 1088717005*$.1 + 1,
    1215766545905692880 1*$.1^2 + 6911064226*$.1 + 1,
    1215766545905692880 1*$.1^2 - 607949990*$.1 + 1
]
```

137.9.4 Abelian Group Structure

Sylow(J, p): JacHyp, RngIntElt -> GrpAb, Map, Eseq

Given the Jacobian J of a hyperelliptic curve defined over a finite field and a prime p, this function returns the Sylow p-subgroup of the group of rational points of J, as an abstract abelian group A. The injection from A to J is also returned as well as the generators of the p-Sylow subgroup.

AbelianGroup(J): JacHyp -> GrpAb, Map

UseUserGenerators:	Bool	Default:	false
UserGenerators :	SetEnum	Default:	

Given the Jacobian J of a hyperelliptic curve defined over a finite field K, this function returns the group of rational points of J as an abstract abelian group A. The isomorphism from A to $J(K)$ is returned as a second value.

If UseUserGenerators is set then the group structure computation is achieved by extracting relations from the user-supplied set of generators in UserGenerators.

HasAdditionAlgorithm(J): JacHyp -> Bool

Returns true if and only if the Jacobian J has an addition algorithm. This is of interest when trying to construct the (abelian) group structure of J: When no user-supplied generators are given, such an algorithm must be present.

137.10 Jacobians over Number Fields or $\mathbb{Q}$

Some functions in this section work for general number fields (notably *TwoSelmerGroup*), while many are only implemented over $\mathbb{Q}$.

137.10.1 Searching For Points

Points(J): JacHyp -> SetIndx

RationalPoints(J): JacHyp -> SetIndx

Bound:	RngInt	Default:	0

Given a Jacobian J of a genus 2 hyperelliptic curve defined by an integral model over the rationals, determine all rational points on the Jacobian J whose naive height on the associated Kummer surface is less than or equal to Bound.

137.10.2 Torsion

`TwoTorsionSubgroup(J): JacHyp -> GrpAb, Map`

Given the Jacobian J of a hyperelliptic curve C which is either of genus 2 or has odd degree defined over a number field K, the function returns $J(K)[2]$ as an abstract group, together with a map sending elements of the abstract group to points on J. The curve C must be given in the simplified form $y^2 = f(x)$.

`TorsionBound(J, n): JacHyp, RngIntElt -> RngIntElt`

Given the Jacobian J of a hyperelliptic curve defined over the rationals or an algebraic number field, this function returns a bound on the size of the rational torsion subgroup of the Jacobian. The bound is obtained by examining the group $J(\mathbb{F}_p)$ for the first n good primes or prime ideals p.

`TorsionSubgroup(J): JacHyp -> GrpAb, Map`

Given the Jacobian J of a genus 2 curve defined over the rationals, this function returns the rational torsion subgroup of J, and the map from the group into J. The curve must have the form $y^2 = f(x)$ with integral coefficients.

Example: Torsion on a Jacobian (ex-a3bcbe)

For the curve

$$C : y^2 = (x+3)(x+2)(x+1)x(x-1)(x-2),$$

the only Q-rational torsion on the Jacobian is 2-torsion.

```
> _<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(&*[x-n : n in [-3..2]]);
> J := Jacobian(C);
> T, m := TwoTorsionSubgroup(J);
> T;
Abelian Group isomorphic to Z/2 + Z/2 + Z/2 + Z/2
Defined on 4 generators
Relations:
    2*P[1] = 0
    2*P[2] = 0
    2*P[3] = 0
    2*P[4] = 0
> [ m(T.i) : i in [1..4] ];
[ (x^2 - 3*x + 2, 0, 2), (x^2 - 2*x, 0, 2), (x^2 - x - 2, 0, 2),
(x^2 - 4, 0, 2) ]
> #T eq #TorsionSubgroup(J);
true
```

The Jacobian for the following curve has torsion subgroup $Z/24$ over $\mathbb{Q}$.

```
> C := HyperellipticCurve((2*x^2-2*x-1)*(2*x^4-10*x^3+7*x^2+4*x-4));
> J := Jacobian(C);
> T, m := TwoTorsionSubgroup(J);
> T;
Abelian Group isomorphic to Z/2
Defined on 1 generator
Relations:
    2*P[1] = 0
> m(T.1);
(x^2 - x - 1/2, 0, 2)
> A,h := TorsionSubgroup(J);
> #T eq #A;
false
```

(continues on next page)

(continued from previous page)

```
> A;
Abelian Group isomorphic to Z/24
Defined on 1 generator
Relations:
    24*P[1] = 0
> P := h(A.1);
> P;
(x^2 - 1/2, -1, 2)
> Order(P);
24
> 12 * P eq m(T.1);
true
```

137.10.3 Heights and Regulator

This section pertains to height functions on the Mordell–Weil group of the Jacobian of hyperelliptic curves over a number field k. However, naive heights and height constants are currently only implemented for Jacobians of genus 2 curves defined over $\mathbb{Q}$.

In the case of genus two curves defined over $\mathbb{Q}$, the canonical height is computed using the algorithm of Müller and Stoll [MS16]. This algorithm computes the canonical height using local error functions on the associated Kummer surface. It avoids factorization, which makes it more efficient than previously used algorithms.

In all other cases the algorithm described in chapter 5 of [Mul10a] is used. It is based on a theorem due to Faltings and Hriljac which expresses the canonical height pairing in terms of Arakelov intersection theory and works as follows:

We find divisors D_P and D_Q of degree zero on the curve representing P and Q, respectively. For this we use the canonical representative of P and Q (see Section *Points on the Jacobian*); if $P = Q$, then the canonical representatives for P and $-P$ are used. Since the canonical representatives have common support at infinity, we subtract the divisor of a function $x - \lambda$ from one of them. These ideas are due to David Holmes [Hol06]. If there are points at infinity in the positive support of P or Q, then we might have to also subtract the divisor of a function $x - \mu$ from the other representatives.

The actual Arakelov intersection computations are performed locally using regular models of the curve (see Section Regular Models of Arithmetic Surfaces) at the relevant primes and Groebner bases over p-adic quotient rings in the non-archimedean case. The algorithm requires factorisation of polynomials over non-archimedean local fields. For archimedean places the intersections multiplicities can be expressed using theta functions with respect to the analytic Jacobian. This relies heavily on several functions described in Section *Analytic Jacobians of Hyperelliptic Curves*.

If the genus is less than 4, the most expensive archimedean operations are usually applications of the Abel-Jacobi map ToAnalyticJacobian. In larger genus, the computation of theta functions, whose running time grows exponentially in the genus is more expensive.

Regarding the non-archimedean computations, the main bottleneck is integer factorisation which is required to find out which primes may yield non-trivial intersection multiplicities.

Applications: Heights are a useful tool for studying rational points on varieties. The most standard applications concerning points on Jacobians are

- proving independence of points in $J(k)/J_{tors}(k)$ (in particular, the regulator is defined in terms of the canonical height), and
- proving non-divisibility: given $P \in J(k)$ and $n \in \mathbb{Z}$, proving that P is not of the form nR for some $R \in J(k)$.

The heights in this section are logarithmic, and they measure the size of the coordinates of the image of P on the Kummer variety associated to J (embedded in $\mathbf{P}^{2^g-1}$, where g is the genus of the curve). The naive height h is simply the height of the image of P in $\mathbf{P}^{2^g-1}$ using an explicit embedding, which is

currently only available for $g = 2$. One can refine this, taking advantage of the group law on J, defining a *canonical height* which has nice properties with respect to the group law, for instance $\hat{h}(nP) = n^2\hat{h}(P)$. In particular, $\hat{h}(P) = 0$ if and only if P is a torsion point. The function $h - \hat{h}$ is bounded on $J(k)$. Computationally, one generally wants an upper bound on this, because then one can find all points up to a given canonical height by doing a search for points of bounded naive height.

`NaiveHeight(P): JacHypPt -> FldPrElt`

`NaiveHeight(P): SrfKumPt -> FldPrElt`

Given a point P on the Jacobian of a curve of genus 2 (or on the associated Kummer surface), the function returns the logarithmic height of the image of P in $\mathbf{P}^3$ under the maps $J \to K \to \mathbf{P}^3$.

`Height(P: parameters): JacHypPt -> FldReElt`

`Height(P: Precision): SrfKumPt -> FldReElt`

`CanonicalHeight(P: parameters): JacHypPt -> FldReElt`

`CanonicalHeight(P: Precision): SrfKumPt -> FldReElt`

```
lambda         : RngIntElt                    Default: 1
mu             : RngIntElt                    Default: 0
LocalPrecision: RngIntElt                     Default: 0
UseArakelov    : BoolElt                      Default: false
Precision      : RngIntElt                    Default: 0
```

The canonical height of a point P on the Jacobian of a hyperelliptic curve over a number field or over the rationals. If the genus is 2 and the ground field is $\mathbb{Q}$, then this computes the canonical height on the associated Kummer surface. Otherwise this function simply computes the height pairing of P with itself using Arakelov intersection theory.

`HeightConstant(J: parameters): JacHyp -> FldPrElt, FldPrElt`

```
Factor         : BoolElt                      Default: true
Steps          : RngIntElt                    Default: 10
Modified       : BoolElt                      Default: false
Fast           : BoolElt                      Default: false
LocalPrecBound: RngIntElt                     Default: 100
Effort         : RngIntElt                    Default: -1
```

If J is the Jacobian of a genus 2 curve defined over the rationals, of the form $y^2 = f(x)$ with integral coefficients, this computes a real number c such that $h_K(P) \leq \hat{h}(P) + c$ for all $P \in J(\mathbb{Q})$, where h_K is the naive logarithmic height obtained from the Kummer surface and $\hat{h}$ is the canonical height on J. The second value returned is a bound for $|\mu_\infty|$, the contribution from the infinite place.

If `Factor` is set, then the discriminant will be factored, and its prime divisors will be considered individually. This usually results in an improvement of the bound. If `Fast` is set, the cheap bound from [Sto99b] is used (and `Factor` is ignored). If `Modified` is set, the modified naive height is used as in [MS16]. `LocalPrecBound` is an upper bound for the precision used in the local computations. `Steps` is a technical parameter that is used in the computation of the contribution of the infinite place to the bound. `Effort` is present for compatibility with previous versions. `Effort` $= 0$ has the effect of setting `Fast` to `true`, `Effort` ≥ 1 has the effect of setting `Factor` to `true` and `Fast` to `false`.

`HeightPairing(P, Q: parameters): JacHypPt, JacHypPt -> FldReElt`

```
lambda         : RngIntElt                    Default: 1
mu             : RngIntElt                    Default: 0
LocalPrecision: RngIntElt                     Default: 0
UseArakelov    : BoolElt                      Default: false
Precision      : RngIntElt                    Default: 0
```

The value of the canonical height pairing for rational points P and Q on the Jacobian of a hyperelliptic curve defined over a number field. The pairing can be defined as $\langle P, Q\rangle := (\hat{h}(P+Q) - \hat{h}(P) - \hat{h}(Q))/2$ and if the genus is 2 and the ground field is $\mathbb{Q}$, the pairing is computed using this definition. Otherwise the pairing is computed using Arakelov intersection theory.

Sometime these fail due to insufficient precision; if this happens, the parameter `LocalPrecision` should be changed accordingly.

Changing the parameters `lambda` and `mu` (see the introduction above) can sometimes speed up the computations because some of the required integer factorisations might be significantly easier for some values of λ and μ than for others. Note that the parameter `mu` is only used if it is nonzero.

The parameter `UseArakelov` indicates whether the algorithm based on Arakelov intersection theory should be used in genus 2.

`HeightPairingMatrix(S: Precision): [JacHypPt] -> AlgMat`

```
Precision  : RngIntElt                    Default: 0
UseArakelov: BoolElt                      Default: false
```

Given a sequence $[P_1, ...P_n]$ of points on the Jacobian J of a hyperelliptic curve defined over a number field, this function returns the matrix with entries $\langle P_i, P_j\rangle$, where the latter denotes the canonical height pairing between P_i and P_j.

`Regulator(S: Precision): [JacHypPt] -> FldReElt`

```
Precision  : RngIntElt                    Default: 0
UseArakelov: BoolElt                      Default: false
```

Given a sequence S of points on the Jacobian J of a hyperelliptic curve defined over a number field k, the function returns the determinant of the height pairing matrix of S. The regulator is equal to zero when the points are dependent in the Mordell–Weil group, and otherwise is equal to the square of the volume of the parallelotope spanned by the points in the subgroup of the free quotient of $J(k)$ generated by S.

`ReducedBasis(S: Precision): [JacHypPt] -> SeqEnum, AlgMatElt, SeqEnum`

`ReducedBasis(S: Precision): {@JacHypPt@} -> SeqEnum, AlgMatElt, SeqEnum`

`ReducedBasis(S: Precision): {JacHypPt} -> SeqEnum, AlgMatElt, SeqEnum`

```
Precision   : RngIntElt                   Default: 0
Rels        : BoolElt                     Default: false
ReducedBasis: Verbose                     Default: 3
```

Given a sequence, indexed set, or set S of points on a genus 2 Jacobian, compute an LLL-reduced basis (with respect to the height pairing) of the subgroup generated by the points modulo torsion. The second return value is the height pairing matrix of the basis, computed with the given `Precision` (default precision when `Precision` $= 0$.) If `Rels` is set to true (and S is not a set), the third return value is a sequence of sequences of integers such that the corresponding input point differs from the specified linear combination of the basis by a torsion point.

Example: Heights on a Genus 2 Jacobian (ex-106026)

This example illustrates some basic properties of heights, and proves that a certain point in $J(\mathbb{Q})$ is not a nontrivial multiple of any other point in $J(\mathbb{Q})$. Let J be the Jacobian of $y^2 = x^6 + x^2 + 2$.

```
> P<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(x^6+x^2+2);
> J := Jacobian(C);
```

Find some small points on C and map them to J:

```
> ptsC := Points(C : Bound:=1000);
> ptsJ := [ ptsC[i] - ptsC[1] : i in [2,3,4,5,6] ];
> ptsJ;
[ (1, x^3, 2), (x + 1, x^3 - 1, 2), (x + 1, x^3 + 3, 2), (x - 1, x^3 - 3, 2),
(x - 1, x^3 + 1, 2) ]
```

The canonical heights of these five points:

```
> [ Height(P) : P in ptsJ ];
[ 0.47983979745040515202327954249 05, 0.0000000000000000000000000000000,
0.47983979745040515202327954249 06, 0.47983979745040515202327954249 06,
0.0000000000000000000000000000000 ]
```

We see that two of them are torsion ($\hat{h} = 0$), and the others are probably equal or inverse to each other modulo torsion, because they appear to have the same canonical height. If so, they would generate a subgroup of rank 1 in $J(\mathbb{Q})/J_{tors}(\mathbb{Q})$. The next command verifies this.

```
> ReducedBasis(ptsJ);
[ (1, x^3, 2) ]
[0.47983979745040515202327954249 05]
> P := ptsJ[1];
> P;
(1, x^3, 2)
```

So this point (which is the first point in our list) generates the others. We proceed to check that the other two non-torsion points in our list are equal to $P+T$ or $-P+T$ for some torsion point T.

```
> Jtors, maptoJ := TorsionSubgroup(J);
> {ptsJ[3], ptsJ[4]} subset { pt + maptoJ(T) : pt in {P,-P}, T in Jtors };
true
```

Now we check the property $\hat{h}(nP) = n^2\hat{h}(P)$ for $n = 23$.

```
> Height(23*P)/Height(P);
529.000000000000000000000000001
```

Of course, the naive height does not behave so nicely, but at least $h - \hat{h}$ should be bounded by the height constant.

```
> HC := HeightConstant(J : Effort:=2);
> HC;
4.263611035929762739947562346 07
```

In particular, all torsion points should have naive height less than this.

```
> for T in Jtors do
>     NaiveHeight(maptoJ(T));
> end for;
0.0000000000000000000000000000000
0.69314718055994530941723212145 8
0.69314718055994530941723212145 8
0.69314718055994530941723212145 8
0.69314718055994530941723212145 8
0.69314718055994530941723212145 8
0.69314718055994530941723212145 8
0.69314718055994530941723212145 8
> // Does the inequality hold for 23*P?
```

(continues on next page)

(continued from previous page)

```
> NaiveHeight(23*P) - Height(23*P) le HeightConstant(J);
true
```

Finally, we show that P is not a nontrivial multiple of another point in $J(\mathbb{Q})/J_{tors}(\mathbb{Q})$ (implying that P generates $J(\mathbb{Q})/J_{tors}(\mathbb{Q})$ if this has rank 1). For suppose that $P = nQ + T$ for some $Q \in J(\mathbb{Q}), T \in J_{tors}(\mathbb{Q})$. Then $\hat{h}(Q) = (1/n^2)\hat{h}(P) < \hat{h}(P)$, and the following search shows there is no Q satisfying this bound.

```
> LogarithmicBound := Height(P) + HeightConstant(J);  // Bound on the naive h(Q)
> AbsoluteBound := Ceiling(Exp(LogarithmicBound));
> PtsUpToAbsBound := Points(J : Bound:=AbsoluteBound );
> ReducedBasis( [ pt : pt in PtsUpToAbsBound ]);
[ (1, x^3, 2) ]

[0.47983979745040515202327954249O5]
```

If Q exists, it would have to be in the set `PtsUpToAbsBound`. But the results of the final command indicate that the group generated by `PtsUpToAbsBound` is also generated by a single point of canonical height 0.479839797450405152023279542491, so there are no nontorsion points Q in `PtsUpToAbsBound` with $\hat{h}(Q) < \hat{h}(P)$.

This example is continued in Example *Both Chabauty Implementations*, where we prove that $J(\mathbb{Q})$ has rank 1, and then use the fact that P generates $J(\mathbb{Q})/J_{tors}(\mathbb{Q})$ to find all rational points on C.

Example: Reduced Basis via Heights `(ex-3facfa)`

We compute a reduced basis for a set of points on the Jacobian J on the genus two curve $C : y^2 = x^6 + x^2 + 1$.

```
> P<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(x^6+x^2+1);
> J := Jacobian(C);
```

We construct some points on C:

```
> Z := PointsAtInfinity(C);
> Z;
{@ (1 : -1 : 0), (1 : 1 : 0) @}
> P1 := Z[1];
> P2 := Z[2];
> P3 := C![1/2,9/8,1];
> P4 := C![-1/2,9/8,1];
```

We now map them to J (for example $Q1$ is the divisor $P1 - P2$):

```
> Q1 := J![P1, P2];
> Q2 := J![P1, P3];
> Q3 := J![P3, P4];
> B, M := ReducedBasis([Q1, Q2, Q3] : Precision := 12);
> B;  // This will be a basis for <Q1, Q2, Q3>
[ (1, -x^3, 2), (x^2, 1, 2) ]
> M;  // The height pairing matrix for the new basis B
[0.2797317933687 5.684341886081E-14]
[5.684341886081E-14 0.9524462128097]
> Determinant(M);
0.2664294871964
```

Since $\mathrm{Det}(M)$ is nonzero, the two generators in B are independent.

137.10.4 Saturation

IsDivisibleBy(P, n): JacHypPt, RngIntElt -> BoolElt, JacHypPt

For a $\mathbb{Q}$-rational point P on the Jacobian of a curve over $\mathbb{Q}$, this returns whether P is divisible by n, i.e., whether there exists a $\mathbb{Q}$-rational point Q such that $nQ = P$. If so, the point Q is also returned.

Saturation(bas, p): [JacHypPt], RngIntElt -> [JacHypPt]

For a sequence of $\mathbb{Q}$-rational points on a Jacobian J, this finds the p-saturation in $J(\mathbb{Q})$ of the group generated by the points, i.e., the smallest group containing the points and closed in $J(\mathbb{Q})$ under division by p. The function returns a sequence of generators for the saturated group modulo its torsion subgroup.

137.10.5 The 2-Selmer Group

The principal functions in this section provide information about the Mordell–Weil group $J(K)$ of a hyperelliptic Jacobian defined over the rationals or a number field. Two descent provides an upper bound (since $J(K)/2J(K)$ embeds in the 2-Selmer group of J). Finer information is provided by **HasSquareSha**, which determines the parity of the 2-rank of the Shafarevich–Tate group, and **RankBounds** collects together all the information that is computable in Magma.

In Magma V2.13, a simpler interface was introduced for computing 2-Selmer groups. The two implementations that were previously accessible still exist internally. A third has been added since. When *TwoSelmerGroup* is called, it chooses which implementation to use, unless the user specifies this by setting parameter.

BadPrimes(C): CrvHyp -> SeqEnum

BadPrimes(J): JacHyp -> SeqEnum

Badness: RngIntElt	Default: 1

Given a hyperelliptic curve C with integral coefficients defined over a number field (or the Jacobian of such a curve), the function returns a sequence containing the primes where the given model has bad reduction.

HasSquareSha(J): JacHyp -> BoolElt

IsEven(J): JacHyp -> BoolElt

Given the Jacobian of a hyperelliptic curve over $\mathbb{Q}$ or a number field, and assuming that the Shafarevich-Tate group of J is finite, this returns **true** if and only if the order of the Shafarevich-Tate group is a square. (Otherwise, the order is twice a square, as shown by Poonen and Stoll in [PS99]). The order is square if and only if the number of "deficient" primes for C is even (see below).

IsDeficient(C, p): CrvHyp, RngIntElt -> BoolElt

For a genus 2 curve defined over $\mathbb{Q}$ or a number field, this returns **true** if C is "deficient" at p. By definition, the curve is deficient if there are no points on C defined over any extension of odd degree over $\mathbb{Q}_p$ (when p is a prime) or over $\mathbb{R}$ (when p is 0). Equivalently, C is deficient if there is no $\mathbb{Q}_p$-rational divisor of odd degree on C.

HasIndexOne(C, p): CrvHyp, RngIntElt -> BoolElt

HasIndexOne(C, p): CrvHyp, RngOrdIdl -> BoolElt

Given a hyperelliptic curve C over $\mathbb{Q}$ or a number field, this returns true if and only if there is a divisor of odd degree on C which is defined over the completion at the prime p. An even genus curve is deficient at p if and only if it does not have index one at p.

HasIndexOneEverywhereLocally(C): CrvHyp -> BoolElt

Returns **true** if and only if C has index one over all completions of its base field (including real completions).

`TwoSelmerGroup(J): JacHyp -> GrpAb, Map, Any, Any`

```
Al                    : MonStgElt                    Default:
Fields                : SetEnum                      Default: {}
ReturnFakeSelmerData: BoolElt                        Default: false
ReturnRawData         : BoolElt                      Default: false
Selmer                : Verbose                      Default: 3
```

The 2-Selmer group of the Jacobian of a hyperelliptic curve defined over $\mathbb{Q}$ or a number field. If the curve is not defined over $\mathbb{Q}$ and its genus is larger than two then it must have a single rational point at infinity (for curves in the simplified form $y^2 = f(x)$, this is equivalent to $f(x)$ having odd degree). The algorithm may work better when an integral model of the curve is given (or better yet, a minimal model).

The Selmer group is returned as a finite abelian group S, together with a map from S to some affine algebra A, which represents the standard map from the Selmer group to $A^*/(A^*)^2$ or to $A^*/(A^*)^2\mathbb{Q}^*$ (depending whether the degree of f is odd or even).

Three separate implementations exist internally in Magma. (Prior to V2.13 two of these were available as the intrinsics `TwoSelmerGroup` and `TwoSelmerGroupData`). The user can specify which implementation is used by setting the optional parameter `Al` to `"TwoSelmerGroupOld"`, `"TwoSelmerGroupNew"`, or to `"TwoSelmerGroupData"`; otherwise, an appropriate choice is made automatically.

Much of the computation time is usually spent on class group and unit calculations. These computations can be speeded up by using non-rigorous bounds, and there are two ways to control which bounds are used. The recommended way is to preset them using one of the intrinsics `SetClassGroupBounds` or `SetClassGroupBoundMaps` (see Setting the Class Group Bounds). The other way is to precompute the class groups of the fields involved (with the desired optional parameters in `ClassGroup`), and then pass these fields to `TwoSelmerGroup` via the optional parameter `Fields`. The class group data is stored, and if `TwoSelmerGroup` requires a field that is isomorphic to one of the given `Fields`, the stored data will automatically be used. The relevant fields are those given by the roots of f, where $y^2 = f(x)$ is the `SimplifiedModel` of the curve.

When called with `ReturnFakeSelmerData`, the program returns an additional item, which specifies the "fake Selmer group" in the terminology of [Sto01]. (This is only relevant to the even degree case.) The returned object is a tuple $\langle B_1, B_2, B_3\rangle$, where B_1 is a sequence of elements in A that span a subgroup S_1 of $A^*/(A^*)^2$, and B_2 and B_3 are sequences of integers that span subgroups S_2 and S_3 of $\mathbb{Q}^*/(\mathbb{Q}^*)^2$. The fake Selmer group S is then determined by the exact sequence $0 \to S_3 \to S_2 \to S_1 \to S \to 0$. See [Sto01] for a full explanation of this.

When called with `ReturnRawData`, the program additionally returns a third item `expvecs` and a fourth item `factorbase`. These specify the images in $A^*/(A^*)^2$ of the Selmer group generators (in unexpanded form). The `factorbase` is a sequence of elements f_j of A, and `expvecs` is a sequence of vectors in $\mathbb{Z}^F$, where F is the number of elements in the `factorbase`. The image in $A^*/(A^*)^2$ of the ith Selmer group generator is then the sum over j of $f_j^{e_j}$ where (e_j) is the ith exponent vector.

`RankBound(J): JacHyp -> RngIntElt`

`RankBounds(J): JacHyp -> RngIntElt, RngIntElt`

An upper, or a lower and an upper bound, on the rank of the Mordell–Weil group of J, which should be the Jacobian of a hyperelliptic curve over the rationals or a number field. `RankBound` can be computed provided both *TwoSelmerGroup* and *TwoTorsionSubgroup* are implemented for J, while `RankBounds` is only implemented for Jacobians defined over $\mathbb{Q}$.

For curves of even degree and odd genus it is not always possible to compute an upper bound for the rank. One can compute instead an upper bound for the rank of the subgroup of the Mordell–Weil group consisting of points which can be represented by rational divisors. When the curve has index one everywhere locally, every rational divisor class can be represented by a rational divisor. `RankBound` and `RankBounds` check if this condition is satisfied and throws an error when it does not hold.

An initial upper bound is furnished by computing the 2-Selmer group. For even degree hyperelliptic curves this can potentially be sharpened by determining if the torsor T parameterizing divisor classes of degree 1 on the curve represents a nontrivial element of Sha[2]. This can be achieved in two ways. If Sha does not have square order (see `HasSquareSha`) then T is nontrivial and the bound can be decreased by 1. If there are rational divisor classes of degree 1 everywhere locally, then T lies in Sha[2], and `RankBounds` will attempt to determine whether T is divisible by 2 in Sha using the algorithm described in [Cre12]. When T is not divisible by 2, the bound may be lowered by 2. Examples of both phenomena are given below.

The lower bound is computed by searching for points on the Jacobian, then determining their independence using either the height pairing machinery or by considering their images in $J(\mathbb{Q})/2J(\mathbb{Q})$.

Example: 2-Selmer Group of a Jacobian (ex-e48977)

In this example, we find out as much as we can about the Mordell–Weil group of the Jacobian of

$$y^2 = x(x + 1344^2)(x + 10815^2)(x + 5406^2)(x + 2700^2).$$

First define the curve and its Jacobian:

```
> _<x> := PolynomialRing(Rationals());
> pol := x*(x+1344^2)*(x+10815^2)*(x+5406^2)*(x+2700^2);
> C := HyperellipticCurve( pol );
> J := Jacobian(C);
> J;
Jacobian of Hyperelliptic Curve defined by
y^2 = x^5 + 155285397*x^4 + 4761213301312596*x^3 + 330186894143664707856000*x^2
+ 450122990999339719434240000000*x over Rational Field
```

We can search for points on C (with x-coordinate up to Bound):

```
> ptsC := Points(C : Bound := 10^6);
> ptsC;
{@ (1 : 0 : 0), (0 : 0 : 1), (43264 : -44828581639628800 : 1),
(43264 : 44828581639628800 : 1) @}
> pointAtInfinity := ptsC[1];
```

This `pointAtInfinity` $(1 : 0 : 0)$ might not seem to lie on C, but recall that in Magma, a hyperelliptic curve lives in a weighted projective plane. We can also search for points on the Jacobian, which takes longer:

```
> time ptsJ := Points(J : Bound := 2000);
Time: 0.670
> ptsJ;
{@ (1, 0, 0), (x, 0, 1) @}
```

The notation for points on J is explained in the section "Points on the Jacobian" in this chapter. The points appearing above are the trivial point on J, and the point corresponding to the divisor $(0 : 0 : 1) - (1 : 0 : 0)$ on C.

```
> ptsJ[1] eq J!0;  // Is the first point equal to 0 on J?
true
> Order( ptsJ[2] );
2
```

So far, we have only found some trivial points on J. In fact, we can see from the equation that J has full 2-torsion, which we now confirm.

```
> Jtors, map := TorsionSubgroup(J);
> Jtors;
```

(continues on next page)

(continued from previous page)

```
Abelian Group isomorphic to Z/2 + Z/2 + Z/2 + Z/4
Defined on 4 generators
Relations:
2*P[1] = 0
2*P[2] = 0
2*P[3] = 0
4*P[4] = 0
```

`Jtors` is an abstract group, and `map` converts elements of `Jtors` to actual points on J. Here are the generator of the $\mathbb{Z}/4$ and its inverse.

```
> map(Jtors.4);
> map(3*Jtors.4);
(x^2 + 32963094*x - 212161021632000, 94792247622*x - 2005558137487296000, 2)
(x^2 + 32963094*x - 212161021632000, -94792247622*x + 2005558137487296000, 2)
```

Looking at the points on C we found above, the third of them looks nontrivial, so we find its order on J:

```
> P := ptsC[3];
> PJ := J![ P, pointAtInfinity ];
> PJ;
(x - 43264, -44828581639628800, 1)
> Order(PJ);
0
```

This means PJ has infinite order on J. Now we do two-descent on J:

```
> #TwoSelmerGroup(J);
64
```

We already knew that $J(\mathbb{Q})[2]$ has order 16, and that the rank of $J(\mathbb{Q})$ is at least 1, so we now know that the rank is either 1 or 2. We now ask whether the order of $Sha(J)$ is a square or twice a square (assuming it is finite):

```
> HasSquareSha(J);
true
```

It follows (assuming $Sha(J)$ is finite) that $Sha(J)$ has square order, and that $J(\mathbb{Q})$ has rank 2, even though the other generator of $J(\mathbb{Q})$ is probably not easy to find, especially if does not come from a point on C.

For more examples of Selmer group computations, see the next section.

Example: Tate-Shafarevich Groups of Nonsquare Order `(ex-930a5d)`

We produce some Jacobians for which the order of the Tate–Shafarevich group is twice a square (assuming it is finite). We then observe how this affects the rank bounds.

A good source of examples are curves $y^2 = 3(x^6 - x^2 + a)$ for $a = 1 \bmod 3$, since these curves are "deficient" at 3 (for the definition, see above); this can be proved by elementary arguments. We now list those (for a up to 50) which have nonsquare Sha.

```
> _<x> := PolynomialRing(Rationals());
> for a := 1 to 50 do
>    if a mod 3 eq 1 then
>       Ca :=  HyperellipticCurve( 3*(x^6-x^2+a) );
>       assert IsDeficient(Ca,3);
>          // (This causes a failure if our assertion is wrong.)
```

(continues on next page)

(continued from previous page)

```
>       if not HasSquareSha(Jacobian(Ca)) then print Ca; end if;
>    end if;
> end for;
Hyperelliptic Curve defined by y^2 = 3*x^6 - 3*x^2 + 12 over Rational Field
Hyperelliptic Curve defined by y^2 = 3*x^6 - 3*x^2 + 21 over Rational Field
Hyperelliptic Curve defined by y^2 = 3*x^6 - 3*x^2 + 30 over Rational Field
Hyperelliptic Curve defined by y^2 = 3*x^6 - 3*x^2 + 48 over Rational Field
Hyperelliptic Curve defined by y^2 = 3*x^6 - 3*x^2 + 57 over Rational Field
Hyperelliptic Curve defined by y^2 = 3*x^6 - 3*x^2 + 66 over Rational Field
Hyperelliptic Curve defined by y^2 = 3*x^6 - 3*x^2 + 84 over Rational Field
Hyperelliptic Curve defined by y^2 = 3*x^6 - 3*x^2 + 93 over Rational Field
Hyperelliptic Curve defined by y^2 = 3*x^6 - 3*x^2 + 102 over Rational Field
Hyperelliptic Curve defined by y^2 = 3*x^6 - 3*x^2 + 129 over Rational Field
Hyperelliptic Curve defined by y^2 = 3*x^6 - 3*x^2 + 138 over Rational Field
```

Aside: we could have tried simply $y^2 = 3(x^6 + a)$ for $a = 1 \bmod 3$, but this won't produce any examples: it's a cute exercise to show that for these curves, the number of deficient primes is always even.

Now consider the Mordell–Weil rank of the second curve listed (where $a = 7$),

```
> C7 :=  HyperellipticCurve( 3*(x^6-x^2+7) );
> J := Jacobian(C7);
> #TwoTorsionSubgroup(J);
1
> #TwoSelmerGroup(J);
2
> RankBound(J);
0
```

There is no nontrivial 2-torsion in $J(\mathbb{Q})$, and the Selmer group has order 2. The program has checked that the order of the Tate–Shafarevich group is twice a square (hence the 2-rank of Sha is exactly 1), and therefore it has returned a `RankBound` indicating that $J(\mathbb{Q})$ has rank 0. This rank bound is unconditional. In fact, the computation proves that there is a subgroup $\mathbb{Z}/2$ in Sha modulo the subgroup of infinitely divisible elements; therefore one may also deduce unconditionally that the 2-power part of Sha is finite cyclic (since if there was an infinitely 2-divisible part, it would still contribute to the 2-Selmer group).

Example: Visibility of the Tate-Shafarevich Group `(ex-e6f6af)`

We demonstrate a non-trivial Tate–Shafarevich group on the Jacobian of a genus 2 curve. We consider the following Jacobian.

```
> P<x>:=PolynomialRing(Rationals());
> C:=HyperellipticCurve(-x^6 + 2*x^5 + 3*x^4 + 2*x^3 - x - 3);
> JC:=Jacobian(C);
```

We determine an upper bound on its Mordell–Weil rank

```
> RankBound(JC);
3
```

But we can only find one independent rational point on this Jacobian.

```
> V:=RationalPoints(JC,x^2 + 83/149*x + 313/596,2);
> B:=ReducedBasis(V);
> #B;
1
```

We test if there is a quadratic twist of our curve that does have high Mordell–Weil rank.

```
> d:=-1;
> Cd:=QuadraticTwist(C,d);
> JCd:=Jacobian(Cd);
> Vd:=RationalPoints(JCd:Bound:=100);
> Bd:=ReducedBasis(Vd);
> #Bd;
4
```

This shows that the Mordell–Weil rank of $\mathrm{Jac}(C_d)(\mathbb{Q})$ is at least four.

Note that $\mathrm{rank}\ \mathrm{Jac}(C)(\mathbb{Q}(\sqrt{d})) = \mathrm{rank}\ \mathrm{Jac}(C)(\mathbb{Q}) + \mathrm{rank}\ \mathrm{Jac}(C_d)(\mathbb{Q})$, so if we can find an upper bound on the Mordell–Weil rank of $\mathrm{Jac}(C)$ over $\mathbb{Q}(\sqrt{d})$ then we also find an upper bound on $\mathrm{rank}\ \mathrm{Jac}(C_d)(\mathbb{Q})$. We first try this with unproven class group data.

```
> SetClassGroupBounds("GRH");
> K:=QuadraticField(d);
> CK:=BaseChange(C,K);
> JCK:=Jacobian(CK);
> NumberOfGenerators(TwoSelmerGroup(JCK));
5
```

We find that $\mathrm{rank}\mathrm{Jac}(C)(\mathbb{Q}(\sqrt{d})) \leq 5$, which means that $\mathrm{Jac}(C)$ must indeed have Mordell–Weil rank 1 over $\mathbb{Q}$. To make the class group computation unconditional, we must check up to the Minkowski bound for each of the number fields utilized in the computation

```
> Aa:=AbsoluteAlgebra(JCK`Algebra);
> L:=Aa[1];
> MinkowskiBound(L);
3763009
```

The command `FactorBasisVerify(IntegerRing(L),1000,MinkowskiBound(L))` would perform the required check, but will take considerable time to complete.

Example: Better Rank Bounds by Torsor Descent (ex-97e7d9)

In this example we compute the Mordell–Weil rank of a genus 3 hyperelliptic Jacobian J with nontrivial 2-torsion in Sha using a different method. Namely we perform a 2-descent on the torsor parameterizing divisor classes of degree one on the curve as described in [Cre12].

```
> f := Polynomial([Rationals()| 30, 10, 30, 20, 10, 10, 10, 30, 10 ]);
> f;
10*x^8 + 30*x^7 + 10*x^6 + 10*x^5 + 10*x^4 + 20*x^3 + 30*x^2 + 10*x + 30
> X := HyperellipticCurve(f);
> J := Jacobian(X);
> SetClassGroupBounds("GRH");
> S := TwoSelmerGroup(J);
> #S;
4
> TorsionBound(J,5);
1
```

Here the 2-Selmer rank is 2 and there is no nontrivial 2-torsion. So the 2-descent on J gives an upper bound of 2 for the Mordell–Weil rank. However, one can do better.

```
> J`TwoSelmerSet;
{}
```

Calling *TwoSelmerGroup* also computes the 2-Selmer set of the torsor T parameterizing rational divisor classes of degree 1 (because C is everywhere locally solvable, the parameter `Al :=`

"TwoSelmerGroupNew" was used). This extra information allows us to deduce a better bound for the rank.

```
> RankBounds(J);
0 0
```

In this example the 2-Selmer set is empty, which implies that T is not divisible by 2 in Sha. Consequently Sha[2] has rank at least 2 (because of well known properties of the Cassels-Tate pairing). The theory behind these computations is described in [Cre12].

Example: Rank Bound Without Local Index One (ex-e005c5)

For hyperelliptic curves of even degree and odd genus it is not always possible to obtain an upper bound for the Mordell–Weil rank. When the curve does not have index one everywhere locally, it may not be the case that every rational divisor class can be represented by a rational divisor.

Here we provide an example of an even degree odd genus curve which does not have index one everywhere locally, but for which we can still compute an upper bound for the Mordell–Weil rank. The use of SetClassGroupBounds speeds up the computation, but is not necessary.

```
> f := Polynomial([Rationals()|-9, 8, 8, 1, -8, -8, -7, -2, -7 ]);
> C := HyperellipticCurve(f);
> Genus(C);
3
> Degree(C);
8
> J := Jacobian(C);
> SetClassGroupBounds("GRH");
> RankBound(J);
Runtime error: Upper bound is not obtainable.
Upper bound for the rank of Pic^0(X)/2*Pic^0(X) is 2.
```

The error occurs because C does not have index one everywhere locally. In fact it is easy to see C has no real points. So over the reals it has no rational divisors of odd degree. However C has points locally at all nonarchimedean primes. This means C fails to have index 1 at exactly one prime. The obstruction to a rational divisor class being represented by a rational divisor is an element of the Brauer group. It follows from reciprocity in the Brauer group that every $\mathbb{Q}$-rational divisor class is represented by a $\mathbb{Q}$-rational divisor. In particular $Pic^0(C) = J(\mathbb{Q})$. So we can disregard the warning.

```
> HasIndexOneEverywhereLocally(C);
false 0
> Roots(f,RealField(50));
[]
> Evaluate(f,0) lt 0;
true
> &and[ HasIndexOne(C,p) : p in BadPrimes(C) ];
true
```

137.10.6 The Mordell–Weil Group

The *Mordell–Weil group* of a Jacobian over $\mathbb{Q}$ is the group $J(\mathbb{Q})$ of rational points on J. The theorem of (Mordell-)Weil states that this group is a finitely generated abelian group. Magma contains functionality that attempts to determine this group when J is the Jacobian of a curve C of genus 2.

A variety of approaches are used for this.

- The computation of the 2-Selmer group of J (see TwoSelmerGroup) and the 2-Selmer set of Pic^1 of C to obtain an upper bound on the rank of $J(\mathbb{Q})$. See also RankBound.
- The computation of the torsion subgroup of $J(\mathbb{Q})$ (see TorsionSubgroup).

- Search for rational points on C and on J and computation of the subgroup of $J(\mathbb{Q})$ generated by the points found. This uses `ReducedBasis`, which in turn is based on height computations.
- Search for rational points on varieties associated to 2-covering spaces of J. This makes it possible to find larger points than by a direct search.
- Search for rational points on varieties associated to 2-covering spaces of Pic^1_C, if Pic^1_C corresponds to an element of the 2-Selmer group that may be in the image of $J(\mathbb{Q})$ and no rational point mapping to it has been found so far. This can find even larger points.
- Use isogenous abelian surfaces to obtain possibly better upper bounds. For this, see `TwoPowerIsogenies`.
- If J is isogenous to a product of two elliptic curves or to the restriction of scalars of an elliptic curve over a quadratic field, use functionality for their Mordell–Weil groups.
- Try to use *visualization* to improve the upper bound. If we can find a squarefree integer $d \neq 1$ such that we can determine the rank of the quadratic twist $J^{(d)}(\mathbb{Q})$ and a bound for the rank of $J(\mathbb{Q}\sqrt{d})$, then we can deduce a bound on the rank of $J(\mathbb{Q})$, since $\mathrm{rk}J(\mathbb{Q}) + \mathrm{rk}J^{(d)}(\mathbb{Q}) = \mathrm{rk}J(\mathbb{Q}(\sqrt{d}))$.
- If the rank can be determined (i.e., upper and lower bound agree), then the known subgroup is saturated (see `Saturation`) to obtain the full Mordell–Weil group.

The resulting procedure is usually quite fast (below one second) and successful. Longer running times can be caused by the class group computation triggered by the 2-Selmer group computation (this can be sped up by assuming GRH via `SetClassGroupBounds ("GRH")`), by the need to find large generators, or when the best upper bound on the rank that can be determined is not sharp.

`MordellWeilGroupGenus2(J): JacHyp -> GrpAb, Map, BoolElt, BoolElt, RngIntElt`

```
SearchBounds     : SeqEnum[RngIntElt]                    Default: [5,10,20,50,100,
↪300,1000]
SearchBounds2    : SeqEnum[RngIntElt]                    Default: [100, 200, 500,␣
↪1000]
SearchBounds3    : SeqEnum[RngIntElt]                    Default: [10, 20, 50,␣
↪100, 200]
MaxBound         : RngIntElt                             Default: 10000
BoundC           : RngIntElt                             Default: 10000
MaxIndex         : RngIntElt                             Default: 1000
maxp1            : RngIntElt                             Default: 29
maxp2            : RngIntElt                             Default: 13
Rankbound        : RngIntElt                             Default: Infinity()
RankOnly         : BoolElt                               Default: false
LogDiscBound     : FldReElt                              Default: 40.0
TwistSearchBound: RngIntElt                              Default: 200
TwistBound       : RngIntElt                             Default: 200
MordellWeilGroup: Verbose                                Default: 3
```

Given the Jacobian J of a genus 2 curve over the rationals, this attempts to compute the Mordell–Weil group $J(\mathbb{Q})$. It returns an abstract abelian group G and a map from G to $J(\mathbb{Q})$. The third value is a flag that indicates whether the subgroup found could be shown to be of finite index. The fourth value is another flag that indicates whether the result is proved to be correct. The last return value is the upper bound for the rank obtained from 2-descent and possibly visualization. If `RankOnly` is set to true, then saturation is not attempted, and the function returns as soon as the rank has been determined. If `RankBound` is provided, it will be used as an upper bound on the rank (e.g., one can pass the analytic rank here to get a result that is conditional on BSD). There are a number of further technical parameters that affect the effort spent during the various stages of the computation. `BoundC` is used as a (multiplicative) height bound for searching for points on the underlying curve. `SearchBounds` is an increasing sequence of positive integers specifying successive height bounds for a point search directly on J (phase 1). `SearchBounds2` is similar and used in the search for rational points on 2-covering spaces of J (phase 2); `SearchBounds3` is used analogously in the search for rational points of 2-covering spaces of Pic^1 of the underlying curve (phase 3).

`MaxBound` is an upper bound on the height used for searching for rational points on J during saturation. `MaxIndex` bounds the primes at which saturation is attempted. `maxp1` and `maxp2` are used to bound the primes p for which $\mathbb{F}_p$-points and the corresponding lattices are constructed for the point searches in phases 2 and 3, respectively. `LogDiscBound` is a bound on the logarithm of the discriminant of curves with isogenous Jacobians that are tried to find a better rank bound. `TwistSearchBound` and `TwistBound` are used in the visualization step; `TwistSearchBound` gives a bound for the point search on the 2-coverings that is used to detect possibly 'good' quadratic twists, and `TwistBound` is a bound on the absolute value of the squarefree integers d such that the quadratic twist by d is considered.

`MordellWeilGroup(J): JacHyp -> GrpAb, Map, BoolElt, BoolElt`

J is a hyperelliptic Jacobian. If the genus of the curve of J is at most 2 (and J is defined over the rationals when it is 2 or over a number field when it is 1), attempt to determine the group of rational points on J. When the base field is finite, this works in general. The first return value is an abstract abelian group, the second is a map from this group into J or into an elliptic curve isomorphic to J when the genus is 1. The third value is a flag that indicates whether the subgroup found could be shown to be of finite index. The fourth value is another flag that indicates whether the result is proved to be correct. Over finite fields, this calls `AbelianGroup`. For genus 2 over the rationals, this calls `MordellWeilGroupGenus2`. For genus 1 over the rationals or number fields, this calls `MordellWeilGroup` on an isomorphic elliptic curve.

Example: Mordell-Weil Group of a Genus 2 Jacobian (ex-7fa3d4)

We find the Mordell–Weil group of the Jacobian of a genus 2 curve.

```
> R<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(-x^6 + x^5 - 3*x^4 + 5*x^3 + 5*x^2 - 5*x + 4);
> J := Jacobian(C);
> SetVerbose("MordellWeilGroup", 1); // so we can see what is going on
> MordellWeilGroup(J);

------------------------------------------------------------------------
Mordell-Weil group of the Jacobian of
Hyperelliptic Curve defined by y^2 = -x^6 + x^5 - 3*x^4 + 5*x^3 + 5*x^2 - 5*x +
4 over Rational Field
------------------------------------------------------------------------

The torsion subgroup is trivial.
Computing rank bound...
Upper bound for the rank: 5

Found 9 points from points on the curve,
 generating a subgroup of rank 5

 ==> rank is 5 <==

Height difference bound is 1.0480.
The MW lattice has (squared) covering radius <= 2.2151.
Search for points up to height 26 to saturate...
Found 55 points.

The group we found is Z^5.

This could be shown to be the full Mordell-Weil group.

Abelian Group isomorphic to Z (5 copies)
Defined on 5 generators (free)
Mapping from: Abelian Group isomorphic to Z (5 copies)
Defined on 5 generators (free) to JacHyp: J given by a rule
```

(continues on next page)

(continued from previous page)

```
true true
```

We see that the group has rank 5 (and no torsion).

Example: Inverting the Mordell-Weil Map (ex-f78b70)

Here is another example; this is curve 968.a.1936.1 from the LMFDB. This example shows that the map from the abstract abelian group to the Jacobian has an inverse.

```
> R<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(x^6-x^4, 1);
> J := Jacobian(C);
> SetVerbose("MordellWeilGroup", 1); // so we can see what is going on
> MordellWeilGroup(J);

------------------------------------------------------------------------
Mordell-Weil group of the Jacobian of
Hyperelliptic Curve defined by y^2 + y = x^6 - x^4 over Rational Field
------------------------------------------------------------------------

The torsion subgroup is Z/5.
Computing rank bound...
Upper bound for the rank: 1

Found 7 points from points on the curve,
 generating a subgroup of rank 1

 ==> rank is 1 <==

Curve is bielliptic (of degree 2). Use elliptic curves.

MW group of first elliptic curve is Z  (verified).
MW group of second elliptic curve is Z/5  (verified).

Points on J generate a subgroup of rank 1

 ==> rank is 1 <==


The group we found is Z/5 + Z.

This could be shown to be the full Mordell-Weil group.

Abelian Group isomorphic to Z/5 + Z
Defined on 2 generators
Relations:
    5*$.1 = 0
Mapping from: Abelian Group isomorphic to Z/5 + Z
Defined on 2 generators
Relations:
    5*$.1 = 0 to JacHyp: J given by a rule
true true
> G, m := $1;
> pts := Points(J : Bound := 1); #pts;
7
> [pt @@ m : pt in pts];
[
```

(continues on next page)

(continued from previous page)

```
    0,
    G.1,
    4*G.1,
    2*G.2,
    -2*G.2,
    4*G.1 - 2*G.2,
    G.1 + 2*G.2
]
> forall{i : i in [1..#pts] | m($1[i]) eq pts[i]};
true
```

Here we have a non-trivial torsion subgroup, and we see a case that can be reduced to computing the Mordell–Weil groups of two elliptic curves.

137.11 Two-Selmer Set of a Curve

Let $C : y^2 = f(x)$ be a hyperelliptic curve of genus g over a number field k. We say $\pi_\delta : D_\delta \to C$ is a *two-cover* of C if, over an algebraic closure, π_δ is isomorphic as a cover to the pull-back of an embedding of C into its Jacobian along multiplication by 2. The two-Selmer set classifies the k-isomorphism classes of two-covers of C that have points everywhere locally.

The hyperelliptic involution acts on such covers by pull-back: $\iota^*(\pi_\delta) = \pi_\delta \circ \iota$ (apply ι after π_δ). The *fake* two-Selmer set is the set

$$\{\delta : D_\delta(k_v) \neq \emptyset \text{ for all places } v \text{ of } k\}/\iota^*$$

If this set is empty, then C has no k-rational points. This can happen even if C itself does have points everywhere locally. If the set is non-empty, it gives information about rational points on C. See [BS09] for the underlying theory as well as a description of an algorithm to compute the set.

TwoCoverDescent(C): CrvHyp -> SetEnum, Map, [Map, SeqEnum]

```
Bound      : RngIntElt                    Default: -1
Fields     : SetEnum                      Default: {}
Raw        : BoolElt                      Default: false
PrimeBound : RngIntElt                    Default: 0
PrimeCutoff: RngIntElt                    Default: 0
```

Computes the fake 2-Selmer set as an abstract set. The map returned as second value can be used to obtain a representation of these abstract elements as elements in an algebra, which allows explicit construction of the corresponding cover.

The optional parameters `Bound`, `Fields`, and `Raw` perform the same function as for *TwoSelmerGroup* and we refer to it for their description. The remaining optional parameters are specific to this routine and we describe them here.

`PrimeBound`: Two covers are of very high genus. Hence, according to the Weil bounds, they can have local obstructions at very large good primes. For instance, for genus 2 one should check all primes up to norm 1153 and for genus 3 one should check all primes up to norm 66553. This is very time consuming. One can use `PrimeBound` to restrict the good primes to be considered to only those whose norm do not exceed the given bound. In principle, this can result in a larger set being returned than the proper two-selmer set.

`PrimeCutoff`: If the curve has bad reduction at some large prime, it can be prohibitively expensive to check the local conditions at this prime. This bound allows a restriction on the norm of bad primes where local conditions are considered. Setting this bound can result in a larger set being returned than the proper two-selmer set.

Example: Two-Cover Descent on Genus 2 Curves (ex-4cc57e)

As an illustration of `TwoCoverDescent`, we give the Magma-code to perform the computations related to the examples in [BS09]. First we give some examples of genus 2 curves that have points everywhere locally, but have an empty 2-Selmer set and hence have no rational points.

```
> Q:=Rationals();
> Qx<x>:=PolynomialRing( Q );
> C:=HyperellipticCurve(2*x^6+x+2);
> Hk,AtoHk:=TwoCoverDescent(C);
> #Hk;
0
> C:=HyperellipticCurve(-x^6+2*x^5+3*x^4-x^3+x^2+x-3);
> Hk,AtoHk:=TwoCoverDescent(C);
> #Hk;
0
```

In the following we consider a curve of genus 2 that does have rational points. In fact, its Jacobian has Mordell–Weil rank 2. We compute its two-covers with points everywhere locally. There are two. They both cover an elliptic curve over a quadratic extension. We use `Chabauty` following [Bru02] to determine the rational points on C. First we check that we can represent the fake two-Selmer set of the curve with some nice elements.

```
> f:=2*x^6+x^4+3*x^2-2;
> C:=HyperellipticCurve(f);
> Hk,AtoHk:=TwoCoverDescent(C:PrimeBound:=30);
> A<theta>:=Domain(AtoHk);
> deltas:={-1-theta,1-theta};
> {AtoHk(d): d in deltas} eq Hk;
true
```

Next, we determine a factorisation of f into a quartic and a quadratic polynomial.

```
> L<alpha>:=NumberField(x^2+x+2);
> LX<X>:=PolynomialRing(L);
> g:=(X^2-1/2)*(X^2-alpha);
> h:=2*(X^2+alpha+1);
> g*h eq Evaluate(f,X);
true
```

For some $\gamma = \gamma(\delta)$, we have that a 2-cover D_δ covers the elliptic curve

$$E : y^2 = \gamma\, g(x).$$

This allows us to translate the question about rational points on D_δ into a question about L-rational points on E with the additional property that x is rational. We verify that the two values of δ we found above, correspond to $\gamma = (1 - \alpha)/2$.

```
> LTHETA<THETA>:=quo<LX|g>;
> j:=hom<A->LTHETA|THETA>;
> gamma:=1/2*(-alpha + 1);
> {Norm(j(delta)):delta in deltas} eq {gamma};
true
> E:=HyperellipticCurve( gamma * g );
> P1:=ProjectiveSpace(Rationals(),1);
> EtoP1:=map<E->P1|[E.1,E.3]>;
```

We present E explicitly as an elliptic curve to Magma

```
> P0:=E![1,(1-alpha)/2];
> Eprime,EtoEprime:=EllipticCurve(E,P0);
> Etilde:=EllipticCurve(X^3+(1-alpha)*X^2+(2-9*alpha)*X+(16-2*alpha));
> EprimeToEtilde:=Isomorphism(Eprime,Etilde);
> EtoEtilde:=EtoEprime*EprimeToEtilde;
> EtildeToP1:=Expand(Inverse(EtoEtilde)*EtoP1);
```

We determine a group of finite odd index in $E(L)$.

```
> success,MWgrp,MWmap:=PseudoMordellWeilGroup(Etilde);
> success;
true
> MWgrp;
Abelian Group isomorphic to Z/2 + Z
Defined on 2 generators
Relations:
    2*MWgrp.1 = 0
```

We determine the set of L-rational points on E that have a $\mathbb{Q}$-rational image under `EtildeToP1`.

```
> V,R:=Chabauty(MWmap,EtildeToP1:IndexBound:=2);
> V;
{
    0,
    MWgrp.1 - MWgrp.2,
    MWgrp.1,
    -MWgrp.2
}
> R;
4
```

Since we found earlier that in this case, there is only one value of γ to consider, we know that any rational point on C must correspond to one of these points on E. The correspondence is given by the fact that E and C both cover the x-line. We determine these points explicitly.

```
> CtoP1 := map< C -> P1 | [C.1,C.3] >;
> pi := Extend( EtildeToP1 );
> { pi( MWmap( v ) ) : v in V };
{ (1 : 1), (-1 : 1) }
> [ RationalPoints( p@@CtoP1 ): p in { P1(Q)| pi( MWmap( v ) ) : v in V } ];
[
    {@ (1 : -2 : 1), (1 : 2 : 1) @},
    {@ (-1 : -2 : 1), (-1 : 2 : 1) @}
]
```

137.12 Chabauty's Method

This is a method for finding the rational points on a curve C of genus at least 2, that applies when the Mordell–Weil group of $Jac(C)$ has rank less than the genus of C. It involves doing local calculations at some prime where C has good reduction.

For (hyperelliptic) curves of genus 2 over $\mathbb{Q}$, two separate routines are available in Magma. In the older one, the user specifies the prime, and the results usually do not determine the set of rational points precisely. In the new implementation, which is far more powerful, Chabauty calculations are combined with a "Mordell–Weil sieve" which puts together information obtained using many different primes, to determine precisely the set of rational points on the curve. However, this routine does not apply to curves with no rational points, in fact one rational point must be known, as this plays a role in the algorithm.

Both routines are applicable only when the Chabauty condition holds: for a genus 2 curve, the Jacobian must have Mordell–Weil rank 1. In addition a routine `Chabauty0` is provided for the easy case where the Jacobian has rank 0.

Brief description of Chabauty's method: Under the assumption that the rank of $J = Jac(C)$ is strictly less than the genus of C, the closure V of $J(\mathbb{Q})$ in $J(\mathbb{Q}_p)$ (in the p-adic topology) can be described as the locus where certain power series vanish. The dimension of V is at most the rank of $J(\mathbb{Q})$. The image of C in J (under a natural embedding) will intersect V in a finite set, and this set must contain $C(\mathbb{Q})$. By examining the power series that define this finite intersection, one obtains upper bounds for the number of points on $C(\mathbb{Q})$ in each congruence class modulo p (or any power of p).

`Chabauty0(J): JacHyp -> SetIndx`

Given the Jacobian of a hyperelliptic curve C over $\mathbb{Q}$, the function finds all rational points on C, under the assumption that $J(\mathbb{Q})$ has rank zero.

The assumption is not checked. When it does not hold, the function computes a subset of $C(\mathbb{Q})$ corresponding (in some sense) to the torsion in $J(\mathbb{Q})$.

`Chabauty(P : ptC): JacHypPt -> SetIndx`

ptC: Pt	Default:

For a curve C of genus 2 over $\mathbb{Q}$, this uses Chabauty's method combined with the Mordell–Weil sieve to find rational points on C. The argument P is a rational point of infinite order on the Jacobian J of C. The function returns the set of points on C whose images in $J(\mathbb{Q})$ lie in the saturation in $J(\mathbb{Q})$ of the group generated by P. In particular when $J(\mathbb{Q})$ has rank 1, all rational points on C are returned.

The algorithm makes use of one rational point on the curve. Such a point may be supplied as the optional argument `ptC`; otherwise, the function attempts to find a point by searching (if it fails to find a point, a runtime error occurs). In particular, this routine cannot be used to show that a curve has no rational points.

`Chabauty(P, p: Precision): JacHypPt, RngIntElt -> SetIndx`

Precision: RngIntElt	Default: 5

Given a point on the Jacobian of a hyperelliptic curve C over $\mathbb{Q}$, the function uses Chabauty's method (at the prime p) to bound the number of rational points on the curve C, The curve must have good reduction at the prime p, and the point P should generate a subgroup of index coprime to p in $J(\mathbb{Q})/J_{tors}(\mathbb{Q})$. The algorithm assumes that C is in the simplified form $y^2 = f(x)$, where f has integral coefficients.

The function returns an indexed set of tuples $\langle x, z, v, k\rangle$ such that there are at most k pairs of rational points on C whose image in P^1 under the x-coordinate map are congruent to $(x : z)$ modulo p^v, and such that the only rational points on C outside these congruences classes are Weierstrass points.

If the optional parameter `Precision` is supplied, then in the returned data each of the v's will be greater than or equal to `Precision`. (This might take some time because the algorithm is not optimised for this.)

Example: Both Chabauty Implementations (ex-9911f8)

In this example, which continues Example *Heights on a Genus 2 Jacobian*, we use both implementations of Chabauty to prove that the curve $y^2 = x^6 + x^2 + 2$ has only the obvious six rational points.

```
> _<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(x^6+x^2+2);
> ptsC := Points(C : Bound:=1000); ptsC;
{@ (1 : -1 : 0), (1 : 1 : 0), (-1 : -2 : 1),
```

(continues on next page)

(continued from previous page)

```
   (-1 : 2 : 1), (1 : -2 : 1), (1 : 2 : 1) @}
> J := Jacobian(C);
> RankBound(J);
1
```

So the Jacobian has rank at most 1. Next we ask whether any of the points on C give us points of infinite order on J.

```
> PJ1 := J! [ ptsC[3], ptsC[1] ];
> Order(PJ1);
8
> PJ2 := J! [ ptsC[5], ptsC[1] ];
> Order(PJ2);
0
```

This means $PJ2$ has infinite order in $J(\mathbb{Q})$. In Example *Heights on a Genus 2 Jacobian* we proved that $PJ2$ is not divisible in $J(\mathbb{Q})$. Since $J(\mathbb{Q})$ has rank 1, $PJ2$ is a generator of $J(\mathbb{Q})/J(\mathbb{Q})_{tors}$.

```
> all_pts := Chabauty(PJ2); all_pts;
{ (1 : -2 : 1), (-1 : -2 : 1), (1 : 2 : 1),
  (1 : -1 : 0), (1 : 1 : 0), (-1 : 2 : 1) }
```

It still finds all points even if we do not give a generator for the free part: a "saturation" step is built in. Naturally, this takes slightly longer.

```
> okay := all_pts eq Chabauty(12*PJ2);
> assert okay; // Check that it is okay
```

So we find that we already had the full set of rational points on C. We could have reached the same conclusion using the other implementation of Chabauty, where we specify that the method be applied at the prime 3 (the most natural choice, since it is the smallest prime of good reduction).

```
> BadPrimes(C);
[ 2, 7 ]
> Chabauty(PJ2, 3);
{@ <1, 1, 4, 1>, <-1, 1, 4, 1>, <1, 0, 4, 1> @}
```

This tells us that there are at most 3 pairs of rational points on C, apart from Weierstrass points. In fact, one can easily check that there are no rational Weierstrass points. Therefore, the 6 known points are the only rational points on C.

Example: Chabauty on a Rank 1 Jacobian (ex-19e0e5)

In this example we show that there are precisely 6 rational points on the curve $y^2 = x^6 + 8$.

```
> _<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(x^6+8);
> ptsC := Points(C : Bound:= 1000); ptsC;
{@ (1 : -1 : 0), (1 : 1 : 0), (-1 : -3 : 1),
   (-1 : 3 : 1), (1 : -3 : 1), (1 : 3 : 1) @}
> J := Jacobian(C);
> PJ := J! [ ptsC[5], ptsC[1]];
> Order(PJ);
0
> RankBound(J);
1
```

This shows that $J(\mathbb{Q})$ has rank 1, and PJ has infinite order. Now we check that PJ generates $J(\mathbb{Q})/J_{tors}(\mathbb{Q})$, although this is not necessary. (For more explanation, see the similar calculation

in Example *Heights on a Genus 2 Jacobian*).

```
> heightconst := HeightConstant(J : Effort:=2, Factor);
> LogarithmicBound := Height(PJ) + heightconst;  // Bound on h(Q)
> AbsoluteBound := Ceiling(Exp(LogarithmicBound));
> PtsUpToAbsBound := Points(J : Bound:=AbsoluteBound );
> ReducedBasis([ pt : pt in PtsUpToAbsBound ]);
[ (x^2 - x + 1, 3, 2) ]
[0.32661733877148842826007676859 0]
> Height(PJ);
0.32661733877148842826007676859 0
```

This shows there are no points in $J(\mathbb{Q})$ with canonical height smaller than that of PJ, so PJ is a generator modulo torsion.

```
> all_pts := Chabauty(PJ : ptC:=ptsC[1]); all_pts;
{@ (1 : -1 : 0), (-1 : -3 : 1), (1 : 3 : 1),
   (-1 : 3 : 1), (1 : -3 : 1), (1 : 1 : 0) @}
> assert all_pts eq Chabauty(2*PJ : ptC:=ptsC[1]);
```

Example: Points in a Rank 1 Subgroup (ex-a36c35)

In Example *2-Selmer Group of a Jacobian* (in the section on 2-Selmer groups) we found a curve of rank 2. Although Chabauty's method cannot be applied to get information about the full set of rational points on C, we can still find all points on C whose images on J lie in a given subgroup of rank 1.

```
> _<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve( x*(x+1344^2)*(x+10815^2)*(x+5406^2)*(x+2700^2) );
> J := Jacobian(C);
> ptsC := Points(C : Bound := 10^6); ptsC;
{@ (1 : 0 : 0), (0 : 0 : 1), (43264 : -44828581639628800 : 1),
   (43264 : 44828581639628800 : 1) @}
> PJ := J! [ ptsC[3], ptsC[1] ];
> Order(PJ);
0
```

Now consider the map from C into J that sends P to $P-$ `ptsC[1]`. We call Chabauty to find all points $P \in C(\mathbb{Q})$ whose image lies in the saturated subgroup spanned by `PJ`.

```
> pts := Chabauty(PJ : ptC:=ptsC[1]); pts;
{ (1 : 0 : 0), (0 : 0 : 1), (-29224836 : 0 : 1),
  (-7290000 : 0 : 1), (-1806336 : 0 : 1), (-116964225 : 0 : 1),
  (43264 : 44828581639628800 : 1), (43264 : -44828581639628800 : 1) }
> assert #pts eq 8;
```

Example: Chabauty for a Rank 0 Jacobian (ex-7f4710)

We find all rational points on $C : y^2 = x^6 + 1$. There are only finitely many rational points on the $Jac(C)$ which means we may use `Chabauty0`.

```
> _<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(x^6+1);
> J := Jacobian(C);
> RankBound(J);
0
> Chabauty0(J);
{@ (1 : -1 : 0), (1 : 1 : 0), (0 : 1 : 1), (0 : -1 : 1) @}
```

The rank bound was obtained by computing the 2-Selmer group, which involved class group and unit computations in the fields given by roots of $x^6 + 1$.

For this curve, there is an easier approach. There is an obvious map from C to the elliptic curve $y^2 = x^3 + 1$, which happens to have rank 0 over $\mathbb{Q}$. To define the map, note that C lives in a weighted projective space where Y has weight 3.

```
> E := EllipticCurve([0,1]);
> Pr2<X,Y,Z> := Ambient(C);
> CtoE := map< C -> E | [X^2*Z,Y,Z^3] >;
```

The map is well-defined (Magma checks this by default). Now we find the rational points on E.

```
> Rank(E);
0
> #TorsionSubgroup(E);
6
> ptsE := Points(E : Bound:=100 ); ptsE;
{@ (0 : 1 : 0), (-1 : 0 : 1), (0 : 1 : 1),
   (0 : -1 : 1), (2 : 3 : 1), (2 : -3 : 1) @}
```

So, these are all the rational points on E. We can see with the naked eye that only those with $X = 0$ pull back to rational points on C, giving us the same six points on C that we found before.

We may also ask Magma to pull them back:

```
> for P in ptsE do
>     preimageofP := P @@ CtoE;
>     Points(preimageofP);
> end for;
{@ (1 : -1 : 0), (1 : 1 : 0) @}
{@ @}
{@ (0 : 1 : 1) @}
{@ (0 : -1 : 1) @}
{@ @}
{@ @}
```

137.13 Cyclic Covers of $\mathbb{P}^1$

A q-cyclic cover of the projective line is a curve C which admits a degree q morphism to $\mathbb{P}^1$ such that the associated extension of function fields is (geometrically) cyclic. By Kummer theory, such a curve has an affine model of the form $y^q = f(x)$ (at least when the base field contains the q-th roots of unity). This class of curve includes hyperelliptic curves. Current functionality for cyclic covers includes testing local solubility, searching for points, and performing descents on curves and their Jacobians when they are defined over a number field.

Most functionality is implemented in the cases where $f(x)$ is a separable polynomial of arbitrary degree and q is prime. One notable exception is that the method of descent on the curve is available for any $f(x)$ that is q-th power free.

137.13.1 Points

A point on a cyclic cover $C : y^q = f_d x^d + f_{d-1} x^{d-1} + ... + f_0$ is specified by a triple representing a point in the appropriate weighted projective plane.

When $d = \text{Degree}(f(x))$ is divisible by q, we consider points the weighted projective plane $\mathbb{P}^2(1 : d/q : 1)$. In this case a point is represented by a triple $[X, Y, Z]$ satisfying

$$Y^q = f_d X^d + f_{d-1} X^{d-1} Z + ... + f_0 Z^d.$$

When $d = \text{Degree}(f(x))$ is not divisible by q, we consider points in the weighted projective plane $\mathbb{P}^2(q : d : 1)$. In this case a point is represented by a triple $[X, Y, Z]$ satisfying

$$Y^q = f_d X^d + f_{d-1} X^{d-1} Z^q + ... + f_0 Z^{dq}.$$

RationalPoints(f, q): RngUPolElt, RngIntElt -> SetIndx
Points(f, q): RngUPolElt, RngIntElt -> SetIndx

```
Bound           : RngIntElt                     Default: 0
DenominatorBound: RngIntElt                     Default: 0
NPrimes         : RngIntElt                     Default: 0
```

For a polynomial f defined over a number field (or $\mathbb{Q}$), this searches for rational points on the curve $y^q = f(x)$. The search region is controlled by the parameters `Bound` (which must be specified) and `DenominatorBound`. The function returns a set of triples (x, y, z) satisfying $y^q = F(x, z)$ where $F(x, z)$ is the homogenisation of $f(x)$ taking x to have weight LCM(q,Degree($f(x)$))/Degree($f(x)$).

The algorithm, and the parameters, are the same as for *RationalPoints* of a hyperelliptic curve.

HasPoint(f, q, v): RngUPolElt, RngIntElt, RngIntElt -> BoolElt, SeqEnum
HasPoint(f, q, v): RngUPolElt, RngIntElt, RngOrdIdl -> BoolElt, SeqEnum

For a polynomial f defined over a number field k, this checks if the curve defined by $y^q = f(x)$ has any points over the completion k_v. The second return value is a triple of elements in the completion of k at v giving a point on the curve. The triple (x, y, z) returned satisfies $y^q = F(x, z)$ where $F(x, z)$ is the homogenisation of $f(x)$ taking x to have weight LCM(q,Degree($f(x)$))/Degree($f(x)$).

The current implementation for testing local solubility of cyclic covers is polynomial time in the cardinality of the residue field, making it somewhat impractical for larger primes.

HasPointsEverywhereLocally(f, q): RngUPolElt, RngIntElt -> BoolElt

For a polynomial f defined over a number field k, this returns true if and only if the curve $y^q = f(x)$ has points over all nonarchimedean completions of k.

137.13.2 Descent

Let C be a curve with Jacobian J and suppose $\psi : A \to J$ is an isogeny of abelian varieties. We say that a morphism of curves $\pi : D \to C$ is a ψ-covering of C if, over an algebraic closure, π is isomorphic as a cover to the pull-back of an embedding of C into J along ψ.

Now suppose that $C : y^q = f(x)$ is a q-cyclic cover of the projective line defined over a number field k. Any primitive q-th root of unity ζ gives rise to an automorphism: ι sending a point (x, y) to $(x, \zeta y)$. This induces an automorphism of the Jacobian. Hence the endomorphism ring of the Jacobian contains the cyclotomic ring $\mathbb{Z}[\zeta]$, and in particular the endomorphism $\phi = 1 - \zeta$. The ϕ-Selmer set of C classifies isomorphism classes of ϕ-coverings of C which are everywhere locally soluble. There is an action of ι on these coverings. The quotient by this action is known as the fake ϕ-Selmer set of C. If this set is empty, then C has no k-rational points. This can happen even if C itself has points everywhere locally. Even when this set is non-empty, it gives information about rational points on C. When $q = 2$, ϕ is multiplication by 2, in which case the ϕ-Selmer set is the same as the 2-Selmer set described in `TwoCoverDescent`.

qCoverDescent(f, q): RngUPolElt, RngIntElt -> Set, Map

```
PrimeBound : RngIntElt                  Default: 0
PrimeCutOff: RngIntElt                  Default: 0
KnownPoints: Set                        Default: {}
CycCov     : Verbose                    Default: 3
```

Computes the fake ϕ-Selmer set of the cyclic cover $C : y^q = f(x)$ as an abstract set. The first return value is the fake ϕ-Selmer set given as a subset of the cartesian product of number fields

having defining polynomials the irreducible factors of f. The map returned is from this cartesian product into the abstract group $A(S, q)$ involved in the computation (the unramified outside S subgroup of the quotient of the etale algebra associated to the ramification points of C by the subgroup generated by scalars and q-th powers).

The input `f` must be a q-th power free polynomial defined over a number field k with class number 1.

The optional parameters `PrimeBound` and `PrimeCutoff` have the same meaning as those used the `TwoCoverDescent` for hyperelliptic curves. Only good primes up to `PrimeBound` and bad primes up to `PrimeCutoff` will be considered in the local computations. If either of these parameters is not set to default, the set returned might be strictly larger than the fake ϕ-Selmer set.

One can obtain a lower bound for the size of the ϕ-Selmer set by considering the images of known global points on C. This often speeds up the computation considerably as the first few primes might be sufficient to show that the ϕ-Selmer set is equal to the set of images of the known rational points. A set of known points may be passed in using the parameter `KnownPoints`. These should be given as triples of the type returned by `RationalPoints`.

Example: Descent Proving No Rational Points `(ex-66395c)`

The following computation shows that the genus 15 curve defined by the equation $y^7 = 2x^7 + 6$ has no rational points even though it is everywhere locally solvable.

```
> _<x> := PolynomialRing(Rationals());
> RationalPoints(3*x^7+6,7 :  Bound := 1000);
{@ @}
> HasPointsEverywhereLocally(3*x^7+6,7);
true
> time Sel := qCoverDescent(3*x^7+6,7 : PrimeBound := 1000);
Time: 0.450
> Sel;
{}
```

137.13.3 Monic Models

Some of the descent algorithms described below must work with a cyclic cover $y^q = f(x)$ where $f(x)$ is of the form $c \cdot h(x)$ where h is a monic integral polynomial and c is a constant (equal to 1 in the case that the degree d of f is not divisible by q). Such an model can be obtained using the intrinsic `MonicModel` described below. If the input is not of the required form, `MonicModel` will be called automatically.

`MonicModel(f, q): RngUPolElt, RngIntElt -> RngUPolElt, SeqEnum`

Returns a polynomial $g(x)$ of the form $c \cdot h(x)$ where h is monic and integral and c lies in the base ring of the input polynomial, f. If the degree of the input polynomial is not divisible by q then g will itself be monic and integral. The second return value is a pair of polynomials $a(x), b(y)$ giving a change of variables which determine an isomorphism between the cyclic covers $y^q = f(x)$ and $y^q = g(x)$.

137.13.4 Descent on the Jacobian

Suppose J is the Jacobian of the cyclic cover C defined by $y^q = f(x)$ over a field k containing the q-th roots of unity. If k is a number field, then the ϕ-Selmer group of J is a finite group which contains $J(k)/\phi(J(k))$. Computing the ϕ-Selmer group can be used to give an upper bound for the rank of $J(k)$. In the case $q = 2$, this reduces to the 2-Selmer group computed with *TwoSelmerGroup*.

For q prime the ϕ-Selmer group may be computed using an algorithm of Poonen-Schaefer [PS97b]. Their algorithm actually computes a fake Selmer group, which is a quotient of the true Selmer group by a subgroup of order dividing q.

When the base field does not contain the q-th roots of unity, one still obtains information by computing the ϕ-Selmer group over the cyclotomic extension. For details see [PS97b], or consider the example below.

PhiSelmerGroup(f, q): RngUPolElt, RngIntElt -> GrpAb, Map

ReturnRawData:	BoolElt	Default: false
Selmer :	Verbose	Default: 3

For a polynomial f over a number field k, this computes the (fake) ϕ-Selmer group of the Jacobian of the curve $y^q = f(x)$ over the field $k(\mu_q)$ obtained by adjoining the q-th roots of unity to k. The polynomial $f(x)$ must be separable and q must be prime.

There are two steps in the algorithm which may require a significant amount of time: the first is the computation of class groups of the extensions of $k(\mu_q)$ given by the irreducible factors of $f(x)$; the second is the computation of the local images of the descent map at the prime(s) above q.

The (fake) Selmer group is returned as a finite abelian group S, together with a map from S to some affine algebra A, which represents the standard map from the Selmer group to $A^*/k^*(A^*)^q$ or to $A^*/(A^*)^q$ correspondingly as the degree of $f(x)$ is or is not divisible by q.

When called with ReturnRawData the program will yield three additional return values, expvecs, factorbase, and selpic1. The first two are used to specify the images in $A^*/k^*(A^*)^q$ of the (fake) Selmer group generators (in unexpanded form). This is done exactly as for *TwoSelmerGroup*. selpic1 is an element of the supergroup containing the (fake) Selmer group, or it is the empty set. This specifies the coset of the (fake) Selmer group corresponding to the (fake) Selmer set of the Pic^1 torsor as described in [Cre12]. When the curve has points everywhere locally this determines whether or not the torsor is divisible by ϕ in Sha, which in turn yields more information on the rank.

Note: this function calls PicnDescent and repackages the information.

PicnDescent(f, q): RngUPolElt, RngIntElt -> RngIntElt, GrpAb, Tup, RngIntElt, Map, GrpAb

Safety:	RngIntElt	Default: 0
Fields:	SetEnum	Default: { }
Selmer:	Verbose	Default: 3

This function returns information at a slightly lower level than PhiSelmerGroup. It returns eight values, which we will call dimSel, SelfakeJ, SelfakePic1, ranktorsglobal, ASqmodktoA, muJktors, toVec, FB.

The return value ASqmodktoA is a map $m : G \to A$, where $A = k[x]/f(x)$ and G is an elementary abelian q-group. The group G is isomorphic to the unramified outside S subgroup of $A*$ modulo q-th powers and scalars. (Here S is the set of bad primes used for the computation.) The value dimSel is an upper bound for the rank of the ϕ-Selmer group of J/k. It is not always sharp. It tries to determine the kernel of the map from Selmer to fake Selmer but doesn't always succeed. SetVerbose("Selmer",1) should give some details about this. The group SelfakeJ is the subgroup of G corresponding to the fake-ϕ-Selmer group of J/k. The return value SelfakePic1 is <SelfakeJ,g> with g in G such that the coset $(g+\text{SelfakeJ})$ in G corresponds to the fake-ϕ-Selmer set of $Pic1/k$. The value ranktorsglobal is a lower bound for rank of $J(k)[\phi]$ (just coming from factorization of f over k, so not sharp in all cases). The group muJktors is the subgroup of G corresponding to the image under the descent map of the part of $J(k)[\phi]$ computed above. The return values toVec and FB are similar to expvecs and factorbase returned by *TwoSelmerGroup*.

There are optional parameters which have the following roles. The local images are determined by computing images of random points until they generate a group that is large enough. Setting Safety := N will force it to compute the images of N extra random points (and throw an error if they generate a larger group). The parameter Fields has the same role as in *TwoSelmerGroup*.

RankBound(f, q): RngUPolElt, RngIntElt -> RngIntElt

RankBounds(f, q): RngUPolElt, RngIntElt -> RngIntElt, RngIntElt

```
ReturnGenerators: BoolElt                    Default: false
Selmer          : Verbose                    Default: 3
```

An upper (or a lower and upper) bound for the rank of the Jacobian of the cyclic cover defined by $y^q = f(x)$. The upper bound is obtained by performing a descent as described above and incorporates the information on the Pic^1 torsor thus obtained. The lower bound is obtained by a naive (and rather inefficient) search for divisors on the curve.

If the optional parameter `ReturnGenerators` is set to `true`, then a third value will be returned. This will be a list of univariate polynomials defining divisors on $\mathbb{P}^1$ that lift to divisors on the curve. The images of these in the Mordell–Weil group generate a subgroup of rank at least as large as the lower bound provided. Independence of these points is determined by considering their images in the (fake) Selmer group.

Example: Rank and 3-Primary Sha by Descent (ex-ddca22)

In this example we use descents to determine both the rank of the Mordell–Weil group and the 3-primary part of Sha for the Jacobian J of a genus 4 curve C over the rationals.

```
> _<x> := PolynomialRing(Rationals());
> f := 3*(x^6 + x^4 + 4*x^3 + 2*x^2 + 4*x + 3);
> k := CyclotomicField(3);
> IsIrreducible(PolynomialRing(k)!f);
true
> SelJ,m,expvecs,fb,SelPic1 := PhiSelmerGroup(f,3 : ReturnRawData);
> Ilog(3,#SelJ);
2
```

In this example the map from the genuine Selmer group to the fake Selmer group has a kernel of order 3. This is because f has degree divisible by $q = 3$ and is irreducible. Hence the ϕ-Selmer group has 3-rank 3. The fact that f is irreducible also implies that there is no nontrivial ϕ-torsion on J defined over k. Thus the ϕ-Selmer group gives an upper bound of $2 * 3 = 6$ for the rank of J over k and an upper bound of 3 for the rank over $\mathbb{Q}$ (see [PS97b]). We can use the information about the Pic^1 torsor to get a better bound

```
> HasPointsEverywhereLocally(f,3);
true
> SelPic1;
{}
```

This means that the Pic^1 torsor represents a ϕ-torsion class in the Shafarevich-Tate group of J that is not divisible by ϕ. In particular it is nontrivial, and because the Cassels-Tate pairing is alternating this implies the ϕ-torsion subgroup of Sha contains a nontrivial subgroup with even 3-rank. Hence, we deduce that $J(k)/\phi(J(k))$ has 3-rank at most $3 - 2 = 1$, and consequently that $J(\mathbb{Q})$ has rank ≤ 1. Calling `RankBounds` will deduce these bounds, and search for generators to obtain a lower bound.

```
> r_l, r_u, gens := RankBounds(f,3 : ReturnGenerators);
> [r_l,r_u];
[1,1]
> gens;
[*
    x^3 - x^2 + 4*x + 4
*]
```

This polynomial defines a divisor on $\mathbb{P}^1$ that lifts to a degree 3 divisor D on $C : y^3 = f(x)$. One way to verify this is the following:

```
> K := NumberField(gens[1]);
> IsPower(Evaluate(f,K.1),3);
true 1/4*(-3*K.1^2 + 9*K.1 + 6)
```

Let D_0 denote the pullback of some point $P \in \mathbb{P}^1$ under the degree 3 map $C \to \mathbb{P}^1$. Then $D - D_0$ is a degree 0 defined over the rationals. According to `RankBounds` its class has infinite order in $J(\mathbb{Q})$. This was determined by considering its image under the composition, $J(\mathbb{Q}) \to J(k) \to Sel^{\phi}(J/k)$. Since there is no nontrivial ϕ-torsion, this amounts to saying that the image is nontrivial. We can verify this with the following computations (carried out internally by `RankBounds`).

```
> A<theta> := Domain(m); // the algebra representing Sel
> D := Evaluate(gens[1],theta);
> imD := m(D);
> assert imD ne SelJ!0;
> assert imD in SelJ;
```

Since the rank over $\mathbb{Q}$ has been computed unconditionally it follows that the ϕ-torsion subgroup of Sha is isomorphic to $(\mathbb{Z}/3)^2$, and since it contains a element not divisible by ϕ in Sha one can deduce that the 3-primary part of Sha is also isomorphic to $(\mathbb{Z}/3)^2$.

137.13.5 Partial Descent

The (fake) ϕ-Selmer set of the curve $C : y^q = f(x)$ defined over a number field k is computed using a map which sends a point (x, y) of C to $x - \theta$, where θ is a generic root of f. This map takes values in the product of the extensions of k defined by the irreducible factors of $f(x)$. More generally one can specify a partial descent map as follows. For each irreducible factor $f_i(x)$ of $f(x)$, fix an extension K_i/k and an irreducible factor $h_i(x) \in K_i[x]$ of $f_i(x)$. The map sending (x, y) to $(h_1(x), ..., h_n(x))$ induces a well defined map from $C(k)$ to an appropriate quotient of the units in the product of the K_i. A partial descent computes the set of elements which restrict into the image of $C(k_v)$ for every prime v of k. This set gives information on the set of rational points of C. If it is empty, then the curve has no rational points.

Geometrically this set can be understood as follows. The coverings parameterised by the ϕ-Selmer set are Galois covers of C with group isomorphic (as a Galois module) to the ϕ-torsion subgroup of the Jacobian. By Galois theory, any Galois submodule gives rise to intermediate coverings. A partial descent computes a set of everywhere locally solvable intermediate coverings corresponding to some Galois submodule of $J[\phi]$.

`qCoverPartialDescent(f, factors, q): RngUPolElt, [* RngUPolElt *], RngIntElt -> Set, Map`

```
PrimeBound : RngIntElt                  Default: 0
PrimeCutoff: RngIntElt                  Default: 0
KnownPoints: Set                        Default: {}
CycCov     : Verbose                    Default: 3
```

Performs a partial descent on the curve $y^q = f(x)$ with respect to the descent map given by `factors`. The map returned is from this cartesian product into the abstract group $A(S, q)$ involved in the computation (the unramified outside S subgroup of the quotient of the etale algebra associated to the orbits of ramification points of C determined by `factors` by the subgroup generated by scalars and q-th powers).

The input `f` must be a separable polynomial defined over a number field k with class number 1. For each irreducible factor $h_i \in k[x]$ of f, the list `factors` must contain an irreducible factor of h_i defined over some (possibly trivial) extension of k. The optional parameters are the same as for `qCoverDescent`.

Example: Partial Descent on a Cyclic Cover (ex-9ab341)

Consider the cyclic cover $D : y^3 = 3(x^2 + 1)(x^2 + 17)(x^2 - 17)$.

```
> _<x>:=PolynomialRing(Rationals());
> f := 3*(x^2+1)*(x^2+17)*(x^2-17);
> pts := RationalPoints(f,3 : Bound:=100);
> pts;
{@
    [ -1, -12, 1 ],
    [ 1, -12, 1 ]
@}
```

So this appears to have only two rational points. We now apply partial descent by only factorising the third factor as follows:

```
> K<th> := NumberField(x^2-17);
> factor := Factorisation(x^2-17,K)[1,1];
> time selmerset, algebra:=qCoverPartialDescent(
>  3*(x^2+1)*(x^2+17)*(x^2-17),[*x^2+1,x^2+17,factor*],3
>  : KnownPoints := pts, PrimeBound := 1000 );
Time: 0.530
> selmerset;
{
    <1, 9, $.1 + 9>,
    <1, 9, 2*$.1 + 2>
}
> delta := <1,9,2+2*th>;
```

Each element of `selmerset` corresponds to a covering of D and every rational point on D lifts to one of these two coverings. For example δ corresponds to a covering Y_δ of D with defining equations

$$\lambda^2 y_1^3 = x^2 + 1,$$

$$9\lambda^2 y_2^3 = x^2 + 17,$$

$(2\theta + 2)\lambda y_3^3 = x - \theta$, where θ is a square root of 17. This covering of D also evidently covers the genus one curve defined over $\mathbb{Q}(\theta)$, $C : (2\theta + 2)v^3 = (x^2 + 1)(x - \theta)$. One can now use the method of Elliptic Curve Chabauty to find all points of $Y_\delta(K)$ which map to points of $D(\mathbb{Q})$. Namely any such point gives rise to a point of $C(K)$ with x-coordinate in $\mathbb{Q}$. Provided we can obtain generators for $C(K)$, such points can be determined using `Chabauty`.

```
> P2<x,v,w>:=ProjectivePlane(K);
> C:=Curve(P2, -(delta[3])*v^3 + (x^2+w^2)*(x-th*w) );
> E,CtoE:=EllipticCurve(C);
> twotors := TorsionSubgroup(E);
> #twotors;
1
> covers,coverstoE:=TwoDescent(E);
> #covers;
1
> cover:=covers[1];
> covertoE:=coverstoE[1];
> pts:={@pt@covertoE : pt in RationalPoints(cover : Bound:=100)@};
> pts;
{@ (1/128*(-153*th + 1377) : 1/1024*(-8109*th + 28917) : 1),
(1/128*(-153*th + 1377) : 1/1024*(6885*th - 17901) : 1),
(1/194688*(-171346689*th + 708942857) : 1/60742656*(-4566609746937*th
    + 18826158509345) : 1), (1/194688*(-171346689*th + 708942857) :
1/60742656*(4566537140481*th - 18825505051241) : 1) @}
> gen := pts[1];
```

This shows that $E(K) \simeq \mathbb{Z}$, so `gen` generates a finite index subgroup.

```
> A:=FreeAbelianGroup(1);
> AtoE:=map<A -> E | elt :-> Eltseq(elt)[1]*gen >;
```

To apply Chabauty, we want to use a map $E \to \mathbb{P}^1$ such that the composition $C \to E \to \mathbb{P}^1$ sends (x, v, w) to x.

```
> P1:=ProjectiveSpace(Rationals(),1);
> CtoE;
Mapping from: CrvPln: C to CrvEll: E
with equations :
1/16*(9*th - 153)*v
1/128*(-729*th + 1377)*x
-1/17*th*x + w
and inverse
th*$.2
1/16*(-9*th + 153)*$.1
$.2 + 1/128*(1377*th - 12393)*$.3
> EtoP1:=map<E -> P1 | [th*E.2,E.2 + 1/128*(1377*th - 12393)*E.3]>;
> Chabauty(AtoE,EtoP1);
{
    -A.1
}
126
> Chabauty(AtoE,EtoP1 : IndexBound:=126);
{
    -A.1
}
126
> pointC:= (-A.1)@AtoE@@CtoE;
> pointC;
(-1 : -1 : 1)
```

Any preimage of this point on Y_δ has $x = -1$, and as such must lie above $(-1, -12) \in D(\mathbb{Q})$. A similar argument proves that the only rational point coming from the other element of the partial Selmer set is $(1, -12)$, proving $D(\mathbb{Q}) = \{(1, 12), (-1, -12)\}$.

137.14 Kummer Surfaces

The Kummer surface K associated to the Jacobian J of a genus 2 curve is the quotient of J by the inverse map. It can be embedded as a quartic hypersurface in projective 3-space whose only singularities are 16 ordinary double points. The Jacobian is a double cover of K ramified at these double points; they are the images of the two-torsion points on J. Resolving the singularities on K yields a $K3$-surface.

Currently, Kummer surfaces in Magma are not schemes, but are of type SrfKum. They can be used to perform arithmetic on the Jacobian without the need for reduction of divisors. The other nontrivial functionality that uses them is point searching.

The Kummer surface and arithmetic on it are implemented for Jacobians in arbitrary characteristic following [Mul10b] which extends earlier work by Flynn, described in chapter 3 of [CF96].

137.14.1 Creation of a Kummer Surface

`KummerSurface(J): JacHyp -> SrfKum`

The Kummer surface of the Jacobian J of a genus 2 curve.

137.14.2 Structure Operations

`DefiningPolynomial(K): SrfKum -> RngMPolElt`

The defining polynomial of the Kummer surface K.

137.14.3 Base Ring

`BaseField(K): SrfKum -> Fld`

`BaseRing(K): SrfKum -> Rng`

`CoefficientRing(K): SrfKum -> Rng`

The base field of the Kummer surface K.

137.14.4 Changing the Base Ring

`BaseChange(K, F): SrfKum, Rng -> SrfKum`

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

Extends the base field of the Kummer surface K to the field F.

`BaseChange(K, j): SrfKum, Map -> SrfKum`

`BaseExtend(K, j): SrfKum, Map -> SrfKum`

Extends the base field of the Kummer surface K by the map j, where j is a ring homomorphism with the base field of C as its domain.

`BaseChange(K, n): SrfKum, RngIntElt -> SrfKum`

`BaseExtend(K, n): SrfKum, RngIntElt -> SrfKum`

Extends the finite base field of the Kummer surface K over a finite field to the degree n extension.

137.15 Points on the Kummer Surface

Points are given by their projective coordinates, normalized depending on the base field.

137.15.1 Creation of Points

`K ! 0: SrfKum, RngIntElt -> SrfKumPt`

Returns the image of the identity element on the Kummer surface K, which is normalized to be the origin $(0 : 0 : 0 : 1)$.

`K ! [x1, x2, x3, x4]: SrfKum, [ RngElt ] -> SrfKumPt`

Returns the point on the Kummer surface K defined by the projective coordinates x_1, x_2, x_3, and x_4.

`K ! P: SrfKum, SrfKumPt -> SrfKumPt`

`K ! P: SrfKum, JacHypPt -> SrfKumPt`

Given a point P on the Jacobian of K, or on a Kummer surface for which K is a base extension, this returns the point on K.

`IsPoint(K, S): SrfKum, [RngElt] -> BoolElt, SrfKumPt`

Given a sequence $S = [x_1, x_2, x_3, x_4]$ of elements of the base field of K, the function returns `true` if the point specified by the sequence defines the homogeneous coordinates of a point on the Kummer surface K. If so, the corresponding point on K is returned as the second value.

`Points(K, [x1, x2, x3]): SrfKum, [RngElt] -> SetIndx`

Returns the indexed set of points on the Kummer surface K with first three coordinates given by the sequence $[x_1, x_2, x_3]$.

137.15.2 Access Operations

`P[i]: SrfKumPt, RngIntElt -> RngElt`

Returns the i-th coordinate of the point P, for $1 \leq i \leq 4$.

`Eltseq(P): SrfKumPt -> SeqEnum`

`ElementToSequence(P): PtHyp -> SeqEnum`

Given a point P on a Kummer surface, the function returns the coordinates of P as a sequence.

137.15.3 Predicates on Points

`P eq Q: SrfKumPt, SrfKumPt -> BoolElt`

Given two points on the same Kummer surface, this returns `true` if and only if the points P and Q are equal.

`P ne Q: SrfKumPt, SrfKumPt -> BoolElt`

Given two points on the same Kummer surface, this returns `false` if and only if the points P and Q are equal.

137.15.4 Arithmetic of Points

`- P: SrfKumPt -> SrfKumPt`

Returns the negation of the point P on the Kummer surface, equal to P itself.

`n * P: RngIntElt, SrfKumPt -> SrfKumPt`

`P * n: SrfKumPt, RngIntElt -> SrfKumPt`

Returns the n-th multiple of the point P on the Kummer surface K.

`Double(P): SrfKumPt -> SrfKumPt`

Returns the double $2 * P$ of the point P.

`PseudoAdd(P1, P2, P3): SrfKumPt, SrfKumPt, SrfKumPt -> SrfKumPt`

Let P and Q be points on the Jacobian J of a genus 2 curve. Given the images P_1, P_2, and P_3 on the Kummer surface of points P, Q, and $P - Q$ on J, the function returns the image of $P + Q$.

`PseudoAddMultiple(P1, P2, P3, n): SrfKumPt, SrfKumPt, SrfKumPt, RngIntElt -> SrfKumPt`

Let P and Q be points on the Jacobian J of a genus 2 curve. Given the images P_1, P_2, and P_3 on the Kummer surface of points P, Q, $P - Q$ on J, the function returns the image of $P + n * Q$.

137.15.5 Rational Points on the Kummer Surface

`RationalPoints(K, Q): SrfKum, [RngElt] -> SetIndx`

Given the Kummer surface of the Jacobian of a genus 2 hyperelliptic curve defined over a ring R and sequence Q of three elements of R, the function returns an indexed set containing those points on K whose first three coordinates correspond to the three terms of Q.

Example: Searching Points on a Kummer Surface (ex-51064e)

We search for some points on the Kummer surface of the hyperelliptic curve $y^2 = x^5 - 7$ defined over the rational field.

```
> P<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(x^5-7);
> Genus(C);
2
> J := Jacobian(C);
> K := KummerSurface(J);
> K;
Kummer surface of Jacobian of Hyperelliptic Curve defined by
```

(continues on next page)

(continued from previous page)

```
    y^2 = x^5 - 7 over Rational Field
> Points(K, [0,1,2]);
{@ (0 : 1 : 2 : 4) @}
> Points(K, [1,3,2]);
{@ @}
> Points(K, [0,1,3]);
{@ (0 : 1 : 3 : 9) @}
```

137.15.6 Pullback to the Jacobian

Points(J, P): JacHyp, SrfKumPt -> SetIndx

RationalPoints(J, P): JacHyp, SrfKumPt -> SetIndx

Given a point P on the Kummer surface associated to the Jacobian J (of a genus 2 curve), the function returns the indexed set of points on J mapping to P.

137.16 Analytic Jacobians of Hyperelliptic Curves

This section contains descriptions of functions pertaining to the creation and use of analytic Jacobians for hyperelliptic curves.

Suppose C is a curve of genus g defined over the complex numbers. The analytic Jacobian of the curve is an abelian torus and is constructed as follows: We view the complex points on the curve, $C(\mathbb{C})$, as a compact Riemann surface of genus g. It is known that the dimension of the vector space of holomorphic differentials is equal to the genus of the curve. So let ϕ_i, $i = 1, 2, \ldots, g$ be a basis for this vector space and set $\overline{\phi} = {}^t(\phi_1, \ldots, \phi_g)$, a vector of holomorphic 1-forms. For a hyperelliptic curve there is a natural choice of holomorphic differentials, namely $\phi_i = x^{i-1}dx/y$. We can now define a mapping $\mathrm{Int} : C \to \mathbb{C}^g$ by $P \mapsto \int_P^\infty \overline{\phi}$. To remove the dependency of this mapping on the path of integration, we define Λ to be the image of the map from the first homology group of C, $H_1(C, \mathbb{Z})$, to $\mathbb{C}^g$ which sends a closed path, $\sigma \in H_1(C, \mathbb{Z})$, on the Riemann surface to $\int_\sigma \overline{\phi}$. Then Λ turns out to be a lattice in $\mathbb{C}^g$ and so we get the complex torus $\mathbb{C}^g/\Lambda$. We then see that Int is a well-defined mapping from C into $\mathbb{C}^g/\Lambda$. We extend this mapping additively to the divisors of degree zero on C. The Abel part of the Abel-Jacobi theorem states that two divisors map to the same image under Int if and only if they are linearly equivalent. The Jacobi part of the theorem asserts that the map is onto. So there is a bijection between the points of $\mathbb{C}^g/\Lambda$ and the points of the (algebraic) Jacobian.

So far the analytic Jacobian $\mathbb{C}^g/\Lambda$ is just a torus, but it can be made into an abelian manifold by choosing a polarization. Let $A_1, \ldots, A_g, B_1, \ldots, B_g$ be a symplectic homology basis. That is, the A_i and B_i are closed paths on the Riemann surface such that all intersection numbers are 0 except that A_i intersects B_i with intersection number 1 for each i. Let ω_1 and ω_2 be the $g \times g$-matrices with entries

$$\omega_{1ij} = \int_{B_j} \phi_i,$$

and

$$\omega_{2ij} = \int_{A_j} \phi_i.$$

Then the columns of $P = (\omega_1, \omega_2)$ form a $\mathbb{Z}$-basis for the lattice $\Lambda = P\mathbb{Z}^{2g}$. The matrix P is called the (big) period matrix. The space of complex, symmetric $g \times g$ matrices with positive definite imaginary part is called the Siegel upper-half space of degree g. Because we chose a symplectic basis for the homology it follows that $\tau = \omega_2^{-1}\omega_1$ is an element of Siegel upper half-space. We will refer to τ as the small period matrix.

Set

$$J = \begin{pmatrix} 0 & \mathbf{1}_g \\ -\mathbf{1}_g & 0 \end{pmatrix}.$$

If we define $E(Px, Py) = {}^t xJy$, for any $x, y \in \mathbb{R}^{2g}$, then E is a Riemann form for the torus $\mathbb{C}^g/\Lambda$ and so the torus acquires a polarization. The existence of a Riemann form on a torus is a necessary and sufficient condition for the torus to be embeddable into projective space. The image is isomorphic to the (algebraic) Jacobian.

In order for the theory describing the map from the analytic Jacobian to the algebraic Jacobian (see Mumford [Mum84]) to work we actually need a special symplectic basis linked to a particular ordering of the roots of the hyperelliptic polynomial. For an example of such a basis see Mumford [Mum84, Chap III.5]. The basis used can be obtained using the function *HomologyBasis* and the roots in the corresponding order are stored as the attribute `A`Roots` (where `A` is an analytic Jacobian).

Note: many of the function with analytic Jacobians use complex approximations, and indelicacies involving these can cause the functions to fail. One common remedy is to first transform the hyperelliptic curve before applying the analytic Jacobian machinery, either by translating or by inverting $x \to 1/x$. This can often have the effect of changing the internal computations sufficiently that any errors are avoided.

137.16.1 Creation and Access Functions

`AnalyticJacobian(f): RngUPolElt -> AnHcJac`

Given $f \in \mathbb{C}[x]$ where $\mathbb{C}$ is a complex field (see Chapter Real and Complex Fields), this function returns the analytic Jacobian of the hyperelliptic curve defined by $y^2 = f(x)$. The polynomial must have degree at least 3 and the complex field precision at least 20 and, for the moment, less than 2000.

`HyperellipticPolynomial(A): AnHcJac -> RngUPolElt`

Returns the polynomial defining the hyperelliptic curve whose Jacobian is A.

`HomologyBasis(A): AnHcJac -> SeqEnum, SeqEnum, Mtrx`

The symplectic homology basis used for the computation of the period matrix of the Jacobian A. First assume that *HyperellipticPolynomial*`(A)` has odd degree. The function `HomologyBasis(A)` returns three values which we label `basepoints`, `loops`, and `S`. Then `basepoints` is a list of points in the complex plane. The return value`loops` is a list of $2g$ lists of indices into `basepoints`. So the list `[i1, i2, i3, ...]` corresponds to the loop consisting of joining the straight lines from `basepoints[in]` to `basepoints[in+1]` for `i` $= 1, 2, \ldots$. These closed loops form a homology basis for the curve but probably not a symplectic basis. The third return value, `S`, is a matrix giving the linear combinations of the loops that were used to form a symplectic homology basis and to compute the period matrix.

If *HyperellipticPolynomial*`(A)` has even degree then the returned homology basis is not for the Riemann surface for this curve, but for the curve of odd degree obtained by sending $a =$ `A`InfiniteRoot` to infinity through the linear fractional transformation given by $x \mapsto 1/(x - a)$. So one needs to apply the inverse transformation to the returned basis in order to get the basis that was used to compute the period matrix.

`Dimension(A): AnHcJac -> RngIntElt`

`Genus(A): AnHcJac -> RngIntElt`

The dimension of the Jacobian A as a complex abelian variety. This is equal to the genus of the curve C for which A is the Jacobian.

`BaseField(A): JacHyp -> Fld`

`BaseRing(A): JacHyp -> Rng`

`CoefficientRing(A): JacHyp -> Rng`

The base field of the Jacobian A.

137.16.2 Period Matrices

`SmallPeriodMatrix(A): AnHcJac -> AlgMatElt`

The small period matrix of the analytic Jacobian A. If A has dimension g or equivalently the hyperelliptic curve has genus g, then this is a symmetric $g \times g$ matrix with positive definite imaginary

part. If $P = (\omega_1, \omega_2)$ is the full period matrix (see *BigPeriodMatrix*) then the small period matrix is defined by $\omega_2^{-1}\omega_1$.

`BigPeriodMatrix(A): AnHcJac -> AlgMatElt`

The full period matrix of the hyperelliptic curve of the Jacobian A. If A has dimension g or equivalently the hyperelliptic curve has genus g, then this is a $g \times 2g$ matrix. The analytic Jacobian as a torus equals $\mathbb{C}^g/\Lambda$, where Λ is the $\mathbb{Z}$-lattice spanned by the columns of the period matrix. The period matrix is computed with respect to the holomorphic differentials $\phi_i = x^{i-1}dx/y$ and a symplectic basis for the homology that can be retrieved using *HomologyBasis*.

`PeriodMapping(A, n): ModSym, RngIntElt -> Map`

Precision: RngIntElt	Default: 0

The complex period mapping to precision `Precision`, computed using n terms of q-expansion and complex precision 30 if the optional parameter `Precision` is 0 (default) or using an appropriate number of terms if `Precision` is non-zero. The period map is a homomorphism $M \to C^d$, where d is the dimension of A.

The period mapping attached to the space of modular symbols A, computed using n terms of the q-expansions of modular forms associated to A. If the parameter `Precision` is set to some positive integer, the parameter n is ignored. Instead, the number of terms of the q-expansions needed to have a result of complex precision `Precision` (i.e., $< 10^{-\text{Precision}}$) is computed. The period mapping is a homomorphism from `AmbientSpace(A)` to a complex vector space of dimension equal to the number of newform conjugates.

`Periods(M, n): ModSym, RngIntElt -> SeqEnum`

Precision: RngIntElt	Default: 0

The complex period lattice associated to M using n terms of the q-expansions and complex precision 30 if the optional parameter `Precision` is 0 (default) or using an appropriate number of terms is `Precision` is non-zero.

The complex period lattice associated to the space of modular symbols M, computed using `Precision` terms of the q-expansions of modular forms associated to M. If the parameter `Precision` is set to some positive integer, the parameter n is ignored. Instead, the number of terms of the q-expansions needed to have a result of complex precision `Precision` (i.e., $< 10^{-\text{Precision}}$) is computed.

137.16.3 Maps between Jacobians

`ToAnalyticJacobian(x, y, A): FldComElt, FldComElt, AnHcJac -> Mtrx`

Let A be the analytic Jacobian of $y^2 = f(x)$. This function maps the point (x, y) on the curve to the analytic Jacobian. More precisely, let $a = \infty$ when *HyperellipticPolynomial* has odd degree and $a =$`A`InfiniteRoot` otherwise. Then it maps the divisor $(x, y) - (a, 0)$ to the analytic Jacobian. As any point on the algebraic Jacobian is simply a sum of such divisors, we can get its image by linearity of the map. The function returns a $g \times 1$ matrix. This should be thought of as an element of $\mathbb{C}^g/\Lambda$ where Λ is the $\mathbb{Z}$-lattice generated by the columns of the *BigPeriodMatrix*.

`FromAnalyticJacobian(z, A): Mtrx, AnHcJac -> SeqEnum`

Let A be an analytic Jacobian of $y^2 = f(x)$, with small period matrix τ. Let z be a $g \times 1$ complex matrix (thought of as an element of $\mathbb{C}^g/\Lambda$ where Λ is the $\mathbb{Z}$-lattice generated by the columns of the *BigPeriodMatrix*). This function returns a list of g (the dimension of A), or fewer, pairs $P_i = \langle x_i, y_i \rangle$ satisfying $y^2 = f(x)$. This is an element of the algebraic Jacobian when interpreted as the divisor $\sum_{i=1}^{g} P_i - g\infty$.

Example: Addition on the Analytic Jacobian (ex-56e605)

We give an example of moving between the algebraic and analytic Jacobians. We take two points on an algebraic Jacobian, map them to the analytic Jacobian, add them there, and map the answer

back to the algebraic Jacobian. We check that the answer corresponds to adding the points on the algebraic Jacobian.

```
> P<x> := PolynomialRing(RationalField());
> f := -8*x^5 - 14*x^4 + 28*x^3 - 30*x^2 + 6*x;
> C := HyperellipticCurve(f);
> J := Jacobian(C);
> pts := ReducedBasis(RationalPoints(J:Bound:=500)); pts;
[ (x^2 + 1, -3*x + 5, 2), (x^2 - 4/9*x + 1/9, -55/27*x + 43/27, 2) ]
```

This Jacobian has rank 2 and these are two generators. We will use them as our two points.

```
> P1 := pts[1]; P2 := pts[2];
> C<I> := ComplexField(50);
> K<x> := PolynomialRing(C);
> f2 := Evaluate(f,x);
> div1 := [r[1] : r in Roots(Evaluate(ElementToSequence(P1)[1],x))];
> div1 := [<d1,Evaluate(ElementToSequence(P1)[2],d1)> : d1 in div1];
> div2 := [r[1] : r in Roots(Evaluate(ElementToSequence(P2)[1],x))];
> div2 := [<d1,Evaluate(ElementToSequence(P2)[2],d1)> : d1 in div2];
> A := AnalyticJacobian(f2);
```

We are working with 50 decimal places. The sequences `div1` and `div2` are the two points on the algebraic Jacobian represented as pairs of $\langle x, y \rangle$ coordinates.

```
> pt1 := &+[ToAnalyticJacobian(d[1],d[2],A) : d in div1];
> pt2 := &+[ToAnalyticJacobian(d[1],d[2],A) : d in div2];
```

Now `pt1` and `pt2` are the images of the two points `P1` and `P2` on the analytic Jacobian (two 2×1 matrices).

```
> sum := FromAnalyticJacobian(pt1+pt2,A);
> sum;
[ <0.62500000000000000000000000000000000000000000000024 +
-1.49999999999999999999999999999999999999999999992*I,
1.73437500000000000000000000000000000000000000053 +
-11.71874999999999999999999999999999999999999999985*I>,
<0.62500000000000000000000000000000000000000000036 +
1.49999999999999999999999999999999999999999999992*I,
1.73437500000000000000000000000000000000000000075 +
11.71874999999999999999999999999999999999999999986*I> ]
```

Now we have mapped sum back to the algebraic Jacobian where it is given as a sequence of two $\langle x, y \rangle$ coordinates. We compare the polynomial having these two x-coordinates as roots, with the algebraic sum of `P1` and `P2`.

```
> clst := Coefficients((x-sum[1][1])*(x-sum[2][1]));
> xpol := P![BestApproximation(Re(c),1000) : c in clst];
> xpol;
x^2 - 5/4*x + 169/64
> P1+P2;
(x^2 - 5/4*x + 169/64, 125/16*x - 403/128, 2)
```

Isomorphisms, Isogenies, and Endomorphism Rings of Analytic Jacobians

In this section we discuss the functionality provided for finding isomorphisms and isogenies between different analytic Jacobians and also for computing the endomorphism ring of an analytic Jacobian.

Suppose we have two abelian varieties $A_1 = \mathbb{C}^g/\Lambda_1$ and $A_2 = C^g/\Lambda_2$ with a morphism $\phi : A_1 \mapsto A_2$. This map lifts to a map $\mathbb{C}^g \mapsto \mathbb{C}^g$, given by some complex $g \times g$ matrix, α. This is called the complex

representation of ϕ. Suppose Λ_i is spanned by the columns of the $g \times 2g$ matrix P_i. As $\phi(\Lambda_1) \subset \Lambda_2$, we see that there must exist an integral $2g \times 2g$ matrix M such that $\alpha P_1 = P_2 M$. M is called the rational representation of ϕ. If A_1 and A_2 are isomorphic and P_1 and P_2 are Frobenius bases, that is, they provide a basis with respect to which the polarization is given by the matrix $J = \begin{pmatrix} 0 & \mathbf{1}_g \\ -\mathbf{1}_g & 0 \end{pmatrix}$, then it follows that M must be a symplectic matrix. That is, M must be such that $MJ^tM = J$. The symplectic matrices act on Siegel upper half-space. If $M = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ is a symplectic matrix and τ an element of Siegel upper half-space, then the action is given by $\tau \mapsto (a\tau + b)(c\tau + d)^{-1}$. Note that if there is an isomorphism from A_1 to A_2 with rational representation M then τ_1 equals the result of letting ${}^tM$ (not M) act on τ_2.

In the case of finding isomorphisms between abelian surfaces, the code makes use of a fundamental domain for 2-dimensional upper half-space (see [Got59]). This often works well even with relatively low precision. The other functions all rely on the function `LinearRelation` and work with greater reliability if a high precision is chosen. This also means that it can happen that these functions miss finding a map. Even if a map is reported it might be an artifact of insufficient precision causing `LinearRelation` to identify a relation that disappears at higher precision. Of course, every effort was made to make these functions work as well as possible and no cases are known where sufficient precision fails to give correct results.

`To2DUpperHalfSpaceFundamentalDomain(z): Mtrx -> Mtrx, Mtrx`

Given a complex matrix in 2-dimensional Siegel upper half-space (that is a symmetric matrix with positive definite imaginary part), this function returns two matrices. The first is an element of the fundamental domain for 2-dimensional Siegel upper half-space described by Gottschling in [Got59] and the second is a symplectic matrix that takes the input to the first return value. Note that if the input is equivalent to an element on the border of the fundamental domain the returned value might depend heavily on the least significant digits of the input.

`AnalyticHomomorphisms(t1, t2): Mtrx, Mtrx -> SeqEnum`

Given two small period matrices t_1 and t_2, this function returns a basis for the $\mathbb{Z}$-module of $2g \times 2g$ integer matrices M such that there exists a complex $g \times g$ matrix α such that $\alpha(t_1, 1) = (t_2, 1)M$.

`IsIsomorphicSmallPeriodMatrices(t1, t2): Mtrx, Mtrx -> Bool, Mtrx`

For two small period matrices t_1 and t_2, this function finds a $2g \times 2g$ symplectic integer matrix M such that there exists a complex $g \times g$ matrix α such that $\alpha(t_1, 1) = (t_2, 1)M$. Such matrices define isomorphisms between the corresponding analytic Jacobians. The first return value is true if such a matrix is found, false otherwise. The second return value is the matrix, if found. Zero matrix otherwise.

`IsIsomorphicBigPeriodMatrices(P1, P2): Mtrx, Mtrx -> Bool, Mtrx, Mtrx`

For two big period matrices P_1 and P_2, this function finds a $2g \times 2g$ symplectic integer matrix M such that there exists a complex $g \times g$ matrix α such that $\alpha P_1 = P_2 M$. Such matrices define isomorphisms between the corresponding analytic Jacobians. The first return value is true if such a matrix is found, false otherwise. The second and third return values are M and α, if found.

`IsIsomorphic(A1, A2): AnHcJac, AnHcJac -> Bool, Mtrx, Mtrx`

For two analytic Jacobians A_1 and A_2 with big period matrices P_1 and P_2, this function finds a $2g \times 2g$ symplectic integer matrix M such that there exists a complex $g \times g$ matrix α such that $\alpha P_1 = P_2 M$. Such matrices define isomorphisms between the analytic Jacobians. The first return value is true if such a matrix is found, false otherwise. The second and third return values are M and α, if found.

`IsIsogenousPeriodMatrices(P1, P2): Mtrx, Mtrx -> Bool, Mtrx`

For two period matrices (small or big) P_1 and P_2, this function finds a nonsingular $2g \times 2g$ integer matrix M such that there exists a complex $g \times g$ matrix α such that $\alpha(P_1, 1) = (P_2, 1)M$, in case of small period matrices, or $\alpha P_1 = P_2 M$ for big period matrices. Such a matrix defines an isogeny between the corresponding analytic Jacobians. The first return value is true if such a matrix is found, false otherwise. The second return value is M, if found. In the case of big period matrices α is the third return value.

IsIsogenous(A1, A2): AnHcJac, AnHcJac -> Bool, Mtrx, Mtrx

For two analytic Jacobians A_1 and A_2 with big period matrices P_1 and P_2, this function finds a nonsingular $2g \times 2g$ integer matrix M such that there exists a complex $g \times g$ matrix α such that $\alpha P_1 = P_2 M$. Such a matrix defines an isogeny between the analytic Jacobians. The first return value is true if such a matrix is found, false otherwise. The second and third return values are M and α, if found.

EndomorphismRing(P): Mtrx -> AlgMat

This function returns the endomorphism ring, as a matrix algebra, of the analytic Jacobian associated to the given period matrix P. If a big period matrix, P, is given then it also returns a list of α-matrices such that $\alpha P = PM$, for each generator, M, of the matrix algebra.

EndomorphismRing(A): AnHcJac -> AlgMat, SeqEnum

This function returns the endomorphism ring, as a matrix algebra, of the given analytic Jacobian A. Suppose the analytic Jacobian has big period matrix P. The second return value is a list of α-matrices such that $\alpha P = PM$, for each generator, M, of the matrix algebra.

Example: Finding a Rational Isogeny (ex-e44b74)

We give an example of how Magma can be used to find rational isogenies between the Jacobians of genus 2 curves. Let us consider the two curves

$$y^2 = x^5 - 4x^4 + 8x^2 - 4x,$$

and

$$y^2 = x^5 + 4x^4 + 10x^3 + 12x^2 + x.$$

These are curves 1 and 3 in the twenty second isogeny class of Smart [Sma97]. We compute their analytic Jacobians to 100 decimal places.

```
> K<x> := PolynomialRing(RationalField());
> C<i> := ComplexField(100);
> KC<xc> := PolynomialRing(C);
> f1 := x^5 - 4*x^4 + 8*x^2 - 4*x;
> f1C := Evaluate(f1,xc);
> A1 := AnalyticJacobian(f1C);
> f2 := x^5 + 4*x^4 + 10*x^3 + 12*x^2 + x;
> f2C := Evaluate(f2,xc);
> A2 := AnalyticJacobian(f2C);
```

We now get a basis for the $\mathbb{Z}$-module of isogenies from `A1` to `A2`. Note that `IsIsogenous` returns `true` for these two Jacobians, but it does not return an isogeny defined over $\mathbb{Q}$.

```
> Mlst := AnalyticHomomorphisms(SmallPeriodMatrix(A1),SmallPeriodMatrix(A2));
> Mlst;
[
  [ 1  0 -1  0]
  [ 0  1  0  0]
  [ 1  1  1  0]
  [ 1  1 -1  2],

  [ 1  0  1  0]
  [ 0  0  0 -1]
  [-1  0  1 -1]
  [ 1  2  1 -1],

  [ 0  1  0 -1]
  [ 1  0  0  0]
```

(continues on next page)

(continued from previous page)

```
  [ 1  1  0  1]
  [ 1  1  2 -1],

  [ 0  1  0  1]
  [ 0  0 -1  0]
  [ 0 -1 -1  1]
  [ 2  1 -1  1]
]
>
```

For each of these four "M" matrices let us find the corresponding α matrices.

```
> P1 := BigPeriodMatrix(A1);
> P2 := BigPeriodMatrix(A2);
> alst := [Submatrix(P2*Matrix(C,M),1,1,2,2)
>          *Submatrix(P1,1,1,2,2)^-1 : M in Mlst];
> alst[1][1,1];
9.4985726731827932691644804899114347978913775491982427655786034864379794810594984815432574268642185 01E-101 + -1.4142135623730950488016887242096980785696718753769480731766797379907324784621070388503875343276415 73*i
>
```

So at least `alst[1]` does not correspond to a rational isogeny. In general none of the four α-matrices might correspond to a rational isogeny. But it is possible that some $\mathbb{Z}$-linear combination of them is defined over the rationals and we want to know whether this actually happens. We can find out by recognizing the entries of the α-matrices as algebraic numbers. This can be done using the `PowerRelation` function.

```
> SetDefaultRealFieldPrecision(100);
> pol := PowerRelation(C!alst[4][1,1],8:Al:="LLL");
> Evaluate(pol,x);
x^4 + 2*x^2 - 1
```

It turns out that every entry of each of the `alst` matrices is in the number field defined by the polynomial $x^8 + 12x^6 + 34x^4 + 52x^2 + 1$. We can use `LinearRelation` to write each entry as a linear combination of the elements of a power basis for this number field. For example:

```
> aroot := C!Roots(Evaluate(x^8 + 12*x^6 + 34*x^4 + 52*x^2 + 1,xc))[1][1];
> basis := [aroot^i : i in [0..7]];
> LinearRelation(Append(basis,alst[1][1,1]));
[ 0, 411, 0, 293, 0, 107, 0, 9, -40 ]
```

So we can write each α-matrix as an algebraic number in the number field defined by $x^8 + 12x^6 + 34x^4 + 52x^2 + 1$. It is then a simple matter to find a linear combination of the α-matrices that is defined over $\mathbb{Q}$. In this particular case we get lucky and `alst[3]` itself is already rational:

```
> alpha := alst[3];
> alpha;
[1.1429873912822749822157835483053409594519099948227986612151258432276326359067
  3819567544802186017203 0E-100 + 1.2392853976334069864667648932075686036279851 2
  5201250666217517621001351893530963357790883492962362769E-100*i
  -0.9999999999999999999999999999999999999999999999999999999999999999999999999999
  9999999999999999999999999 + -1.1867460295221457538279151206799346478636391298
  7256639374529369743809277710674524919691817563280250 7E-100*i]
[-2.0000000000000000000000000000000000000000000000000000000000000000000000000000
  0000000000000000000000 + 8.3347107997000908349034998848064280654648729937412
  2756873472630524543977843856788766310506327194212 5E-101*i
```

(continues on next page)

(continued from previous page)

```
  1.000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
  000000000000000000000000 + -1.290641867783003764771213651499250861488005819465331943514008039451662739730964143625816137036994307E-100*i]
> Matrix(2,2,[Round(Real(x)) : x in Eltseq(alpha)]);
[ 0  1]
[ 2 -1]
```

So the two Jacobians are $\mathbb{Q}$-isogenous. We can use `alpha` to map points from the one Jacobian to the other. We check that it maps rational points to rational points.

```
> C1 := HyperellipticCurve(f1);
> J1 := Jacobian(C1);
> pts1 := RationalPoints(J1:Bound:=500);
> pts1;
{@ (1, 0, 0), (x^2 - 2*x + 1, 1/2*x + 1/2, 2), (x^2 - 2*x + 1, -1/2*x - 1/2, 2),
(x, 0, 1), (x - 1, 1, 1), (x - 1, -1, 1), (x^2 - 2, 0, 2), (x^2 + 2, 4*x, 2),
(x^2 + 2, -4*x, 2), (x^2 - x, x, 2), (x^2 - x, -x, 2), (x^2 - 4*x - 2, 4*x, 2),
(x^2 - 4*x - 2, -4*x, 2), (x^2 - 4*x + 2, 0, 2), (x^2 - 9/4*x - 4, 3/8*x - 2,
2), (x^2 - 9/4*x - 4, -3/8*x + 2, 2), (x^2 - 52/9*x + 50/9, 152/27*x - 160/27,
2), (x^2 - 52/9*x + 50/9, -152/27*x + 160/27, 2), (x^2 + 16/25*x - 18/25,
  152/125*x - 96/125, 2), (x^2 + 16/25*x - 18/25, -152/125*x + 96/125, 2) @}
> P1 := pts1[8]; P1;
(x^2 + 2, 4*x, 2)
> div1 := [r[1] : r in Roots(Evaluate(ElementToSequence(P1)[1],xc))];
> div1 := [<d1,Evaluate(ElementToSequence(P1)[2],d1)> : d1 in div1];
> pt1 := &+[ToAnalyticJacobian(d[1],d[2],A1) : d in div1];
> P2 := FromAnalyticJacobian(alpha*pt1,A2);
> clst := Coefficients((xc-P2[1][1])*(xc-P2[2][1]));
> xpol := K![BestApproximation(Re(c),10^50) : c in clst];
> xpol;
x^2 + x + 1
> C2 := HyperellipticCurve(f2);
> J2 := Jacobian(C2);
> pts2 := RationalPoints(J2:Bound:=500);
> pts2;
{@ (1, 0, 0), (x^2 + 2*x + 1, x - 1, 2), (x^2 + 2*x + 1, -x + 1, 2), (x + 1, 2,
1), (x + 1, -2, 1), (x, 0, 1), (x^2 + x, 2*x, 2), (x^2 + x, -2*x, 2), (x^2 + x +
  1, 2*x - 1, 2), (x^2 + x + 1, -2*x + 1, 2), (x^2 - 23*x - 16, 122*x + 84, 2),
(x^2 - 23*x - 16, -122*x - 84, 2), (x^2 + 23/4*x + 9, 59/8*x + 57/2, 2), (x^2 +
  23/4*x + 9, -59/8*x - 57/2, 2), (x^2 - 106/9*x + 1/9, 2599/54*x - 19/54, 2),
(x^2 - 106/9*x + 1/9, -2599/54*x + 19/54, 2), (x^2 - 9/16*x - 1/16, 269/64*x +
  21/64, 2), (x^2 - 9/16*x - 1/16, -269/64*x - 21/64, 2) @}
```

`ToAnalyticJacobianMumford(pt, AJ): JacHypPt, AnHcJac -> Mtrx`

Given a point pt on the Jacobian of a hyperelliptic curve over $\mathbb{Q}$ with Analytic Jacobian AJ, return the image of pt under the Abel-Jacobi map of AJ, i.e., a complex $(g \times 1)$-column vector representing a point on the complex torus associated to AJ.

`ToAnalyticJacobianMumford(pt, AJ, conj): JacHypPt, AnHcJac, RngIntElt -> Mtrx`

Map the point on the Jacobian over a number field K to the analytic Jacobian using the conj'th embedding of K into the complex numbers. This function is the same as the previous one, except that the hyperelliptic curve can be defined over a number field K and that the third parameter gives the index of the embedding of K into the complex numbers.

`FromAnalyticJacobianProjective(z, A): Mtrx[FldCom], AnHcJac -> SeqEnum`

This returns a divisor consisting of points on the projective curve over the complex numbers with coordinates $(x : y : z)$ corresponding to z. This function is the same as `FromAnalyticJacobian`

except that it returns points in weighted projective space $\mathbf{P}_{\mathbf{1,g+1,1}}$ where g is the genus of the hyperelliptic curve X. In contrast to `FromAnalyticJacobian`, it also works if the divisor on X representing the point has a point lying on the line at infinity, i.e., $z \neq 1$.

137.16.4 From Period Matrix to Curve

`RosenhainInvariants(t): Mtrx -> Set`

Given a small period matrix t corresponding to an analytic Jacobian A of genus g, this function returns a set S of $2g-1$ complex numbers such that the hyperelliptic curve $y^2 = x(x-1)\prod_{s\in S}(x-s)$ has Jacobian isomorphic to A. The name of the function comes from the fact that a hyperelliptic curve in the form $y^2 = x(x-1)(x-\lambda_1)...(x-\lambda_{2g-1})$ is said to be in Rosenhain normal form.

Example: Finding a Genus 2 CM Curve `(ex-3d7636)`

We give an example of how Magma can be used to find the equation of a genus 2 curve whose Jacobian has Complex Multiplication by a given field. We use the field $\mathbb{Q}(\sqrt{-2+\sqrt{2}})$.

```
> SetSeed(1);
> Q := RationalField();
> P<x> := PolynomialRing(Q);
> R<s> := NumberField(x^2-2);
> PP<x> := PolynomialRing(R);
> RF := NumberField(x^2-(-2+s));
> CMF<t> := AbsoluteField(RF);
> O := MaximalOrder(CMF);
```

Now, for a CM type Φ, and $\mathcal{O}$ the ring of integers, $\mathbb{C}^2/\Phi(\mathcal{O})$ is a torus with Complex Multiplication by the maximal order of our field $\mathbb{Q}(\sqrt{-2+\sqrt{2}})$. In order for this to be an abelian variety we have to find a principal polarization. This can be done using Algorithm 1 in [Wam99]. In this case it turns out that a principal polarization is essentially given by a generator of the inverse different.

```
> D := Different(O);
> IsPrincipal(D);
true
> xi := -1/8*O.2; // a chosen generator of D^-1
> xi*O eq D^-1;
true
> auts := Automorphisms(CMF : Abelian := true);
> cc := auts[3];
> cc(cc(t)) eq t;
true
> cc(xi^2) eq xi^2;
true
```

So `cc` is complex multiplication and `xi` squared is in the real subfield. By Theorem 3 of [Wam99] we see that this gives a principal polarization. We now find a Frobenius basis for our lattice with respect to the non-degenerate Riemann form given by `xi`.

```
> Z := IntegerRing();
> E := Matrix(Z,4,4,[Trace(xi*cc(a)*b) : b in Basis(O), a in Basis(O)]);
> D, C := FrobeniusFormAlternating(E); D;
[ 0  0  1  0]
[ 0  0  0  1]
[-1  0  0  0]
[ 0 -1  0  0]
> newb := ElementToSequence(Matrix(O,C)*Matrix(O,4,1,Basis(O)));
> SetKantPrecision(O,100);
> Abs(Re(Conjugate(xi,2))) lt 10^-10 and Im(Conjugate(xi,2)) gt 0;
```

(continues on next page)

(continued from previous page)

```
true
> Abs(Re(Conjugate(xi,4))) lt 10^-10 and Im(Conjugate(xi,4)) gt 0;
true
```

The CM type Φ is given by the second and fourth complex embeddings. We can finally write down a big period matrix and find the element in the Siegel upper half-space corresponding to our CM Jacobian:

```
> C := ComplexField(100);
> BigPM := Matrix(C,2,4,[Conjugate(b,2) : b in newb] cat
>                       [Conjugate(b,4) : b in newb]);
> tau := Submatrix(BigPM,1,3,2,2)^-1*Submatrix(BigPM,1,1,2,2);
```

We can use *EndomorphismRing* to check that the analytic Jacobian corresponding to `tau` does have CM by the correct field:

```
> MA := EndomorphismRing(tau); Dimension(MA);
4
> MAGens := SetToSequence(Generators(MA)); MAGens;
[
    [-1  0  0  0]
    [ 0 -1  0  0]
    [ 0  0 -1  0]
    [ 0  0  0 -1],

    [  0   0 -17  -7]
    [  0   0  -7  -3]
    [  5 -12   0   0]
    [-12  29   0   0]
]
> MP := [MinimalPolynomial(g): g in MAGens];
> IsIsomorphic(NumberField(rep{f: f in MP | Degree(f) gt 1}), CMF);
true
```

Now let us find a curve with this Jacobian.

```
> S := RosenhainInvariants(tau);
> P<x> := PolynomialRing(C);
> f := x*(x-1)*&*{x-a : a in S};
> IC := IgusaClebschInvariants(f);
> ICp := [BestApproximation(Re(r),10^50) : r in
>          [IC[1]/IC[1], IC[2]/IC[1]^2, IC[3]/IC[1]^3, IC[4]/IC[1]^5]];
> ICp;
[ 1, 5/324, 31/5832, 1/1836660096 ]
> C1 := HyperellipticCurveFromIgusaClebsch(ICp);
> C2 := ReducedWamelenModel(C1);
> C2;
Hyperelliptic Curve defined by y^2 = -x^5 - 3*x^4 + 2*x^3 + 6*x^2 - 3*x - 1
over Rational Field
```

Let's check that this curve's analytic Jacobian is isomorphic to the one corresponding to `tau`.

```
> P<x> := PolynomialRing(C);
> f := -x^5 - 3*x^4 + 2*x^3 + 6*x^2 - 3*x - 1;
> A := AnalyticJacobian(f);
> IsIsomorphicSmallPeriodMatrices(tau,SmallPeriodMatrix(A));
true
```

(continues on next page)

(continued from previous page)

```
[ 1 -2  0  0]
[ 0  0  2  1]
[ 1 -2  5  2]
[ 2 -5  4  2]
```

137.16.5 Voronoi Cells

This section describes two functions that are concerned with the computation of Voronoi cells.

`Delaunay(sites): SeqEnum -> SeqEnum`

This function computes the Delaunay triangulation for the sites given by *sites*. The argument *sites* should be a sequence of pairs (of type `Tup`) of real numbers. The real numbers should all belong to the same real field. For n sites, a sequence of n sequences is returned. The ith sequence contains the list of indices of the sites to which site i should be joined to form the triangulation.

`Voronoi(sites): SeqEnum -> SeqEnum, SeqEnum, SeqEnum`

This function computes the Voronoi cells for the sites given by *sites*. A Voronoi cell of a site consists of all those points in the plane closer to that site than to any other. The argument *sites* should be a sequence of pairs (of type `Tup`) of real numbers. The real numbers can be either fixed precision or free reals, but they should all be in the same real field.

Three sequences, *siteedges*, *dualsites*, and *cells* are returned. The sequence *siteedges* is what the intrinsic *Delaunay* would have returned (so it defines the Delaunay triangulation). The sequence *dualsites* is a sequence of triples $\langle x, y, m\rangle$, interpreted as follows: If m is zero the triple represents the point x, y. If m is non-zero the triple represents a point "at infinity" in the direction of the vector x, y. For n sites, *cells* is a sequence of n sequences. The ith sequence is a list $[i1, i2, i3, ..., im]$ such that the cell around site i is formed by connecting *dualsites[i1]* to *dualsites[i2]*, … If a cell is infinite the first and last indices in the sequence point to infinite points. That is, the two infinite sides are given by the lines $(x_0, y_0) + t(x_1, y_1)$, for $t \geq 0$ where (x_0, y_0) equals *dualsites[i2][1], dualsites[i2][2]* and (x_1, y_1) equals *dualsites[i1][1], dualsites[i1][2]* or (x_0, y_0) equals *dualsites[im-1][1], dualsites[im-1][2]* and (x_1, y_1) equals *dualsites[im][1], dualsites[im][2].*

137.17 Invariants

Over an algebraically closed field k of characteristic p with $p \neq 2$, the isomorphism class of a genus g hyperelliptic curve $y^2 = f(x)$ with $f \in k[x]$ corresponds to the orbit of the binary form $z^{2g+2} f(x/z)$ under the classical action of $\mathrm{GL}_2(k)$. Working out generators for the algebra $R_g(\mathbb{C})$ was a popular pastime of nineteenth-century mathematicians.

For $g = 2$, Igusa [Igu60a] managed to give a "universal set of invariants", which works in every characteristic, including 2. These can be called by the function `IgusaInvariants()`. There are also other sets of invariants (`IgusaClebschInvariants()`, `ClebschInvariants()`) and absolute invariants (`G2Invariants()`), which are used for historical or practical reasons.

For $g = 3$, Shioda [Shi67] found invariants in characteristic 0. The reductions of the Shioda generators for $R_3(\mathbb{C})$ are still generators for $R_3(k)$ when the characteristic of k is greater than 7. For more information, we refer to [LRS21].

137.17.1 Igusa Invariants

The Clebsch, Igusa–Clebsch, and Igusa invariants may be computed for curves of genus 2.

The Magma package implementing the functions was written by Everett W. Howe (howe@alumni.caltech.edu) with some advice from Michael Stoll and is based on some gp routines written by Fernando Rodriguez–Villegas as part of the Computational Number Theory project funded by a TARP grant. The gp routines may be found at https://web.archive.org/web/20110515095750/http://www.ma.utexas.edu/users/villegas/cnt/inv.gp.

In addition, a package of functions written by Reynald Lercier and Christophe Ritzenthaler, and Jeroen Sijsling [LRS21] has been added. It contains, amongst other things, functionality for working with a different set of absolute (as opposed to weighted projective) invariants referred to as Cardona—Quer—Nart—Pujola invariants. These work in characteristic 2 as well as other characteristics, although the definitions are different in the two cases.

Rodriguez–Villegas's routines are based on a paper of Mestre [Mes91]. The first part of Mestre's paper summarizes work of Clebsch and Igusa, and is based on the classical theory of invariants. This package contains functions to compute three types of invariants of quintic and sextic polynomials f (or, perhaps more accurately, of binary sextic forms):

- The *Clebsch invariants* A, B, C, D of f, as defined on p. 317 of Mestre;
- The *Igusa–Clebsch invariants* A', B', C', D' of f, as defined on p. 319 of Mestre; and
- The *Igusa invariants* (or *Igusa J-invariants*, or J-*invariants*) J_2, J_4, J_6, J_8, J_{10} of f, as defined on p. 324 of Mestre.

The corresponding functions are `ClebschInvariants`, `IgusaClebschInvariants`, and `JInvariants`, respectively. For convenience, we use `IgusaInvariants` as a synonym for `JInvariants`.

Igusa invariants may be defined for a curve of genus 2 over any field and for polynomials of degree at most 6 over fields of characteristic not equal to 2. The Igusa invariants of the curve $y^2+hy = f$ are equal to the Igusa invariants of the polynomial $h^2 + 4 * f$ except in characteristic 2, where the latter are not defined. In practice, the functions below will not calculate the Igusa invariants of a polynomial unless 2 is a unit in the coefficient ring. However, Igusa invariants of curves are available for all coefficient rings. (But see below.)

Igusa invariants are given by a sequence $[J_2, J_4, J_6, J_8, J_{10}]$ of five elements of the coefficient ring of the polynomials defining the curve. This sequence should be thought of as living in weighted projective space, with weights 2, 4, 6, 8, and 10.

It should be noted that many of the people who work with genus 2 curves over the complex numbers prefer not to work with the real Igusa invariants, but rather work with some related numbers, $[I_2, I_4, I_6, I_{10}]$ (or $[A', B', C', D']$ in Mestre's terminology), that we call the *Igusa–Clebsch invariants*, of the curve. Once again, these live in weighted projective space. The Igusa–Clebsch invariants of a polynomial are defined in terms of certain nice symmetric polynomials in its roots, and, in characteristic zero, the J-invariants may be obtained from the I-invariants by some simple homogeneous transformations. In fact, many of the genus-2-curves-over-the-complex-numbers people refer to the elements $i1 := I_2^5/I_{10}$, $i2 := I_2^3 * I_4/I_{10}$, and $i3 := I_2^2 * I_6/I_{10}$ as the "invariants" of the curve. The problem with the Igusa–Clebsch invariants is that they do not work in characteristic 2 and it was for this reason that Igusa defined his J-invariants.

The coefficient ring of the polynomial f must be an algebra over a field of characteristic not equal to 2 or 3.

The Cardona—Quer—Nart—Pujola invariants are three absolute invariants g_1, g_2, g_3 which can be derived from the J-invariants and which provide an affine classification of all genus two curves over a basefield k up to isomorphism over $\bar{k}$. That is, there is a 1-1 correspondence between $\bar{k}$-isomorphism classes of such curves and triples (g_1, g_2, g_3) in k^3. There are also functions to construct a curve with given invariants and to find all twists of such a curve (*ie* representatives of the k-isomorphism classes in the given $\bar{k}$-isomorphism class), which will be described in later sections.

The invariants are different in the characteristic 2 and odd (or 0) characteristic cases. Details about the former case may be found in [CNP05a]. See [CQ05b] for the latter case. In the odd characteristic case, the formulae for $[g_1, g_2, g_3]$ in terms of the J-invariants are as follows:

$$[\frac{J_2^5}{J_{10}}, \frac{J_2^3 J_4}{J_{10}}, \frac{J_2^2 J_6}{J_{10}}] \quad J_2 \neq 0$$

$$[0, \frac{J_4^5}{J_{10}^2}, \frac{J_4 J_6}{J_{10}}] \quad J_2 = 0, J_4 \neq 0$$

$$[0, 0, \frac{J_6^5}{J_{10}^2}] \quad J_2 = J_4 = 0$$

In the characteristic 2 case, the field k must be perfect. Formulae for the invariants (labelled j_i rather than g_i) may be found on p. 191 of [CNP05a].

ClebschInvariants(C): CrvHyp -> SeqEnum

Given a hyperelliptic curve C having genus 2, compute the Clebsch invariants A, B, C, and D as described on p. 317 of [Mes91]. The base field of C may not have characteristic 2, 3, or 5. The invariants are found using Überschiebungen.

ClebschInvariants(f): RngUPolElt -> SeqEnum

Given a polynomial f of degree at most 6, compute the Clebsch invariants A, B, C, and D as described on p. 317 of [Mes91]. The coefficient ring of the polynomial f must be an algebra over a field of characteristic not equal to 2, 3, or 5. The invariants are found using Überschiebungen.

IgusaClebschInvariants(C: parameters): CrvHyp -> SeqEnum

```
Quick: BoolElt                    Default: false
```

Given a curve C of genus 2 defined over a field, the Igusa–Clebsch invariants A', B', C', and D' as described on p. 319 of [Mes91] are found. These will be all be zero in characteristic 2. If `Quick` is `true`, the base field of C may not have characteristic 2, 3 or 5 and a faster method using Überschiebungen is employed; otherwise, universal formulae are used.

IgusaClebschInvariants(f, h): RngUPolElt, RngUPolElt -> SeqEnum

Given a polynomial h having degree at most 3 and a polynomial f having degree at most 6, the Igusa–Clebsch invariants A', B', C', and D' of the curve $y^2 + hy - f = 0$ are found. These will be all be zero in characteristic 2.

IgusaClebschInvariants(f: parameters): RngUPolElt -> SeqEnum

```
Quick: BoolElt                    Default: false
```

Given a polynomial f having degree at most 6 and defined over a ring in which 2 is a unit, the Igusa–Clebsch invariants A', B', C', and D' of the polynomial f are found. These will be all be zero in characteristic 2. If `Quick` is `true`, the coefficient ring of f may not have characteristic 2, 3 or 5 and a faster method using Überschiebungen is employed; otherwise, universal formulae are used.

IgusaInvariants(C: parameters): CrvHyp -> SeqEnum, SeqEnum

JInvariants(C: parameters): CrvHyp -> SeqEnum

```
Quick    : BoolElt                Default: false
extend   : BoolElt                Default: false
normalize: BoolElt                Default: false
```

Given a curve C of genus 2 defined over a field, the function returns the Igusa invariants (or J-invariants) J_2, J_4, J_6, J_8, J_{10} as described on p. 324 of [Mes91]. If `Quick` is `true`, the base field of C must not have characteristic 2, 3 or 5 and a faster method using Überschiebungen is employed; otherwise, universal formulae are used.

The parameter `normalize` can give a normal form of the list of invariants, allowing easy comparisons.

Note that the Igusa invariants do not contain the invariant I_{15} which is one of the generator of the ring of binary sextic invariants but which is not useful to characterize genus 2 curves. However it may be useful to have it in the reconstruction process over the rationals. It can be stored using the parameter `extend`.

IgusaInvariants(f, h: parameters): RngUPolElt, RngUPolElt -> SeqEnum, SeqEnum

JInvariants(f, h: parameters): RngUPolElt, RngUPolElt -> SeqEnum, SeqEnum

```
Quick    : BoolElt                    Default: false
extend   : BoolElt                    Default: false
normalize: BoolElt                    Default: false
```

Given a polynomial h having degree at most 3 and a polynomial f having degree at most 6, this function returns the Igusa invariants (or J-invariants) J_2, J_4, J_6, J_8, J_{10} of the curve $y^2 + hy = f$. The coefficient ring R of the polynomials h and f must either (a) have characteristic 2, or (b) be a ring in which Magma can apply the operator `ExactQuotient(n,2)`. For example, R may be an arbitrary field, the ring of rational integers, a polynomial ring over a field or over the integers and so forth. However, R may not be a p-adic ring, for instance. If the desired coefficient ring does not meet either condition (a) or condition (b), then `ScaledIgusaInvariants` should be used and its invariants then scaled by the appropriate powers of $\frac{1}{2}$.

The parameter `normalize` can give a normal form of the list of invariants, allowing easy comparisons.

Note that the Igusa invariants do not contain the invariant I_{15} which is one of the generator of the ring of binary sextic invariants but which is not useful to characterize genus 2 curves. However it may be useful to have it in the reconstruction process over the rationals. It can be stored using the parameter `extend`.

`IgusaInvariants(f: parameters): RngUPolElt -> SeqEnum`

`IgusaInvariants(f: parameters): RngMPolElt -> SeqEnum`

`JInvariants(f: parameters): RngUPolElt -> SeqEnum`

```
Quick    : BoolElt                    Default: false
extend   : BoolElt                    Default: false
normalize: BoolElt                    Default: false
```

Given a polynomial f having degree at most 6 which is defined over a ring in which 2 is a unit, return the Igusa invariants (or J-invariants) J_2, J_4, J_6, J_8, J_{10} of f. If `Quick` is `true`, the coefficient ring of f may not have characteristic 2, 3 or 5 and a faster method using Überschiebungen is employed; otherwise, universal formulae are used.

The parameter normalize can give a normal form of the list of invariants, allowing easy comparison.

Note that the Igusa invariants do not contain the invariant I_{15} which is one of the generator of the ring of binary sextic invariants but which is not useful to characterize genus 2 curves. However it may be useful to have it in reconstruction process over the rational. It can be stored using the parameter `extend`.

`IgusaAlgebraicRelations(JI): SeqEnum -> SeqEnum`

Return generators of the ideal of relations between the Igusa invariants.

`IgusaInvariantsEqual(JI1, JI2): SeqEnum, SeqEnum -> BoolElt`

Check whether Igusa Invariants $JI1$ and $JI2$ of two genus 2 hyperelliptic curves or of two binary forms of degree 8 are equivalent.

`DiscriminantFromIgusaInvariants(JI): SeqEnum -> Any`

Compute the discriminant of a genus 2 hyperelliptic curve from the given Igusa Invariants.

`ScaledIgusaInvariants(f, h): RngUPolElt, RngUPolElt -> SeqEnum`

Given a polynomial h having degree at most 3 and a polynomial f having degree at most 6, return the Igusa J-invariants of the curve $y^2 + hy = f$, scaled by $[16, 16^2, 16^3, 16^4, 16^5]$.

`ScaledIgusaInvariants(f): RngUPolElt -> SeqEnum`

Given a polynomial f having degree at most 6 which is defined over a ring not of characteristic 2, return the Igusa J-invariants of f, scaled by $[16, 16^2, 16^3, 16^4, 16^5]$.

AbsoluteInvariants(C): CrvHyp -> SeqEnum

Given a curve C of genus 2 defined over a field, the function computes the ten absolute invariants of C as described on p. 325 of [Mes91].

ClebschToIgusaClebsch(Q): SeqEnum -> SeqEnum

Convert Clebsch invariants in the sequence Q to Igusa–Clebsch invariants.

IgusaClebschToIgusa(S): SeqEnum -> SeqEnum

Convert Igusa–Clebsch invariants in the sequence S to Clebsch invariants.

G2Invariants(C): CrvHyp -> SeqEnum

Compute and return the sequence of three Cardona—Quer—Nart—Pujola invariants (see the introduction above) for C of genus 2.

G2ToIgusaInvariants(GI): SeqEnum -> SeqEnum

Convert the sequence of Cardona—Quer—Nart—Pujola invariants (see the introduction above) to a corresponding sequence of Igusa J-invariants.

IgusaToG2Invariants(JI): SeqEnum -> SeqEnum

Convert the sequence of Igusa J-invariants to Cardona—Quer—Nart—Pujola invariants (see the introduction above).

137.17.2 Shioda Invariants

The Shioda invariants may be computed for curves of genus 3 in characteristic 0 or characteristic ≥ 11. There are 9 of these, the first 6 being algebraically independent and the last 3 being algebraic over the field generated by the other 6. The discriminant is not one of these invariants. For more details, see [LR12] or [Shi67]. These invariants have weights and naturally give a point in a weighted projective space. There are also intrinsics for computing the six Maeda invariants of the curve ([Mae90a]). In characteristic $2, 3$ and 7, a set of separating invariants is known [Bas15]. They are sufficient to characterize the geometric isomorphism classes of hyperelliptic curves of genus 3.

ShiodaInvariants(C): CrvHyp -> SeqEnum, SeqEnum

ShiodaInvariants(f): RngUPolElt -> SeqEnum, SeqEnum

ShiodaInvariants(f): RngMPolElt -> SeqEnum, SeqEnum

ShiodaInvariants(fh): SeqEnum -> SeqEnum, SeqEnum

```
normalize            : BoolElt                    Default: false
PrimaryOnly          : BoolElt                    Default: false
IntegralNormalization: BoolElt                    Default: false
degmin               : RngIntElt                  Default: 1
degmax               : RngIntElt                  Default: Infinity()
```

Returns a sequence containing the 9 Shioda invariants of a genus 3 hyperelliptic curve C along with a sequence containing the weight of the corresponding invariant (always $2, 3, 4, 5, 6, 7, 8, 9, 10$) when the base field of C is of characteristic 0 or ≥ 11. There are also versions that takes a single polynomial f of degree less than or equal to 8 as argument (corresponding to the curve $y^2 = f(x)$) or a sequence of two polynomials fh (corresponding to the curve $y^2 + h(x)y = f(x)$). In these cases if the degree of f (resp $f - h^2/4$) is less than 7, or if the discriminant is zero, the invariants will be invariants of the corresponding binary octic but no longer of a genus 3 hyperelliptic curve. Such sequences of invariants are referred to as *singular*.

The sequence of invariants can be considered as giving a point in a dimension 8 weighted projective space with the above weights. If the parameter `normalize` is set to `true`, the sequence is scaled to give a new sequence representing the same point in a normalized form. This means that two isomorphic curves will not necessarily give the same sequence of invariants, but they will always give the same sequence of normalized invariants.

When the characteristic of the field is 2, 3, or 7 a set of separating invariants is returned. They are sufficient to characterize isomorphism classes of hyperelliptic curves of genus 3. In characteristic

5, a minimal list of invariants that generates the largest subring of invariants that has been found so far is returned.

The lists of invariants can be normalized with the parameter `normalize`. It is also possible to use the parameter `IntegralNormalization` to obtain the all of these invariants as polynomials over the integers. The flag `PrimaryOnly` can be used to return a homogeneous system of parameters, which is proven in all characteristics (even 5). Finally, the parameters `degmin, degmax` are useful for obtaining only those invariants within some range.

`ShiodaInvariantsEqual(V1, V2): SeqEnum, SeqEnum -> BoolElt`

Returns whether two sequences of Shioda invariants $V1$ and $V2$ represent the same point in the natural weighted projective space. This is equivalent to asking whether their normalized versions (see previous intrinsic) are the same, which means that they represent the same isomorphism class of curves (if they are non-singular).

`DiscriminantFromShiodaInvariants(JI): SeqEnum -> RngElt`

Returns the corresponding discriminant from a sequence of Shioda invariants JI. The discriminant (of a binary octic) is just a polynomial in its 9 Shioda invariants. Note that is scaled if the invariants are scaled, so that the value is different for example, if applied to the results of calling `ShiodaInvariants` with `normalize` set to `true` or `false`. The significant thing though is whether this is non-zero or not. The value is zero if and only if JI are *singular* invariants (associated to a polynomial with multiple roots or of degree ≤ 6).

`ShiodaAlgebraicInvariants(FJI): SeqEnum -> SeqEnum`

```
ratsolve: BoolElt                    Default: true
```

As mentioned in the introduction, the first 6 Shioda invariants are algebraically independent forms on the genus 3 hyperelliptic moduli space and the remaining 3 are algebraic over them.

This intrinsic takes a sequence FJI of 6 elements of a field k that must be of characteristic 0 or ≥ 11. If these are considered to be the first 6 Shioda invariants (of a possibly singular set), this intrinsic returns a sequence containing all possible sequences of the full 9 invariants over k that have those of FJI as the first 6, if the parameter `ratsolve` is `true` (the default). If `ratsolve` is `false`, the sequence returned instead contains the 6 polynomials in 3 variables over k, which defines a dimension 0, degree 5 system whose solutions are the possible last 3 invariants.

Example: Shioda Invariants of a Genus 3 Curve (ex-ffb27f)

```
> k := GF(37);
> P<t> := PolynomialRing(k);
> C := HyperellipticCurve(t^8+33*t^7+27*t^6+29*t^5+4*t^4+
>         18*t^3+20*t^2+27*t+36);
> ShiodaInvariants(C);
[ 33, 16, 30, 31, 8, 18, 0, 30, 31 ]
[ 2, 3, 4, 5, 6, 7, 8, 9, 10 ]
> ShiodaAlgebraicInvariants([k|33,16,30,31,8,18]);
[
    [ 33, 16, 30, 31, 8, 18, 0, 30, 31 ],
    [ 33, 16, 30, 31, 8, 18, 14, 10, 25 ]
]
> ShiodaAlgebraicInvariants([k|33,16,30,31,8,18] : ratsolve := false);
[
    $.1^5 + 21*$.1^4 + 12*$.1^3 + 3*$.1^2 + 23*$.1,
    $.1^2 + 6*$.1 + 33*$.2 + 23*$.3 + 36,
    $.1*$.2 + 34*$.1 + 36*$.2 + 32*$.3,
    $.1*$.3 + 17*$.1 + 14*$.2^2 + 23*$.2 + 17*$.3 + 21,
    27*$.1 + $.2*$.3 + 36*$.2 + 27*$.3 + 2,
    8*$.1 + 27*$.2^2 + 2*$.2 + $.3^2 + 19*$.3 + 27
]
```

`MaedaInvariants(C): CrvHyp -> SeqEnum`

`MaedaInvariants(f): RngUPolElt -> SeqEnum`

Returns the six Maeda field invariants $(I2, I3, I4, I4p, I8, I9)$ of a genus 3 hyperelliptic curve C. Again, the base field of C must be of characteristic 0 or ≥ 11. For this intrinsic, the model of C must also be in simplified $y^2 = f(x)$ form. There is also a version with argument f, a univariate polynomial of degree less than or equal to 8, which returns the Maeda invariants of the binary octic given by homogenizing f (to degree 8). This is the same as asking for the invariants of $y^2 = f(x)$ when f has degree 7 or 8.

137.17.3 Creation from Invariants

The problem is to construct a curve of genus 2 from a given set of Igusa–Clebsch invariants defined over the same field as the field of moduli (simply the smallest field in which the invariants lie). Mestre [Mes91] shows that this is not always possible, even in theory. But over a finite field or the rationals he gives a method for deciding whether it is possible and if it is, for finding such a curve. Mestre's algorithm was implemented by P. Gaudry. Mestre's algorithm does not work when the curve has a split Jacobian. Cardona and Quer [CQ05b] showed that in this case one can always find a curve defined over the field of moduli and gave equations for such a curve. This works over any field of characteristic not 2, 3, or 5.

In any characteristic, the code package [LRS21] produces a genus 2 curve from a given set of Cardona—Quer—Nart—Pujola invariants (see Subsection *Igusa Invariants*). They use the work of Cardona and Quer cited in the previous paragraph in the odd characteristic case, with some extra work for characteristics 3 and 5. In characteristic 2, they use the models from [CNP05a].

Lercier, Ritzenthaler, and Sijsling [LRS21] have also contributed a package for genus 3 hyperelliptic curves that produces a curve from a given set of Shioda invariants (see Subsection *Shioda Invariants*). This works in characteristic not 5.

Over a field of characteristic zero, the equation of the curve returned by these methods can involve huge coefficients. For curves over the rationals, a minimization procedure can be applied. It uses the function `MinRedBinaryForm` but before that optimize the reconstruction process by choosing a "nice" conic. Strangely, the procedure is more obvious in genus 3. For genus 2, it may involve the invariant I_{15}. If this one is provided, it may speed up the computation.

`HyperellipticCurveFromIgusaClebsch(S): SeqEnum -> CrvHyp`

```
Reduce: BoolElt                    Default: false
```

This attempts to build a curve of genus 2 with the given Igusa-Clebsch invariants (see Subsection *Igusa Invariants*) by Mestre's algorithm (and Cardona's in case of non-hyperelliptic involutions) defined over the field F in which the invariants lie. Currently this field must be the rationals, a number field, or a finite field of characteristic greater than 5. If there exists no such curve defined over F, then either a curve over some quadratic extension is returned (when F is the rationals), or an error results (when F is a number field).

When the base field is the rationals, then the parameter `Reduce` invokes `ReducedWamelenModel`.

`HyperellipticCurveFromIgusaInvariants(S): SeqEnum -> CrvHyp, GrpPerm`

`HyperellipticCurveFromG2Invariants(S): SeqEnum -> CrvHyp, GrpPerm`

```
RationalModel: BoolElt                  Default: false
minimize     : BoolElt                  Default: false
```

From a given sequence of Igusa or Cardona—Quer—Nart—Pujola invariants (as described in Subsection *Igusa Invariants*) over a finite field or the rationals, return a genus 2 curve with these absolute invariants. This works in all characteristics as noted in the introduction to the section. The geometric automorphism group is also returned as a finitely-presented group.

`HyperellipticCurveFromShiodaInvariants(JI): SeqEnum -> CrvHyp, GrpPerm`

`HyperellipticPolynomialFromShiodaInvariants(JI): SeqEnum -> SeqEnum, GrpPerm`

```
RationalModel: BoolElt                    Default: false
minimize     : BoolElt                    Default: false
```

Given a sequence of 9 Shioda invariants JI over the rationals or a finite field k of characteristic different from 5, the first intrinsic returns a genus 3 hyperelliptic curve over k with these invariants. The abstract geometric automorphism group is also returned as a permutation group. This will cause an error if JI are singular Shioda invariants.

The second intrinsic also works for singular invariants. It returns a list of one polynomial f of degree ≤ 8 with the given invariants. If JI are non-singular, $y^2 = f(x)$ is a genus 3 curve with invariants JI.

If the field of definition is the rationals and when there is no obstruction, a rational model can be obtained using `RationalModel`. Moreover a minimization of the size of the coefficients of the curve can be obtained with the flag `minimize`.

For more details, see [Bas15, KLL+18, LR12].

Example: Curve from Igusa-Clebsch Invariants (ex-6da88d)

A typical run would look like:

```
> SetVerbose("Igusa",1);
> IgCl := [ Rationals() |
>   -549600, 8357701824, -1392544870972992, -3126674637319431000064 ];
> time C := HyperellipticCurveFromIgusaClebsch(IgCl);
Found conic point:
(63705669133950658705501560272896505986878769262593332850455\\
76123068237304687 5/18103398482659638797652756701644023680087 9\\
24350669804640627790086809805712049105263690909696, 611702243\\
78843623609866834721286550443257754798173427871477348775268 68\\
90697363615036010742187 5/36851195888684221102913488343636314 8\\
77715458043146715761966305043223626871732802305004524520180 84\\
2489569787904)
Time: 0.090
> // C is already only well defined up to twist
> time C := ReducedWamelenModel(C);
Time: 0.600
> HyperellipticPolynomials(C);
29*x^6 + 26*x^5 + 49*x^4 + 60*x^3 + 140*x^2 + 172*x + 92
0
```

An example in characteristic 2 from Cardona—Quer—Nart—Pujola invariants:

```
> k<t> := GF(16);
> g2_invs := [t^3,t^2,t];
> HyperellipticCurveFromG2Invariants(g2_invs);
Hyperelliptic Curve defined by y^2 + (x^2 + x)*y =
   t*x^5 + t*x^3 + t*x^2 + t*x over GF(2^4)

Finitely presented group on 3 generators
Relations
    $.2^2 = Id($)
    $.1^-3 = Id($)
    ($.1^-1 * $.2)^2 = Id($)
    $.3^2 = Id($)
    $.1 * $.3 = $.3 * $.1
    $.2 * $.3 = $.3 * $.2
```

(continues on next page)

(continued from previous page)

```
> _,auts := $1;
> #auts; // auts = D_12
12
```

A genus 3 example using Shioda invariants

```
> k := GF(37);
> FJI := [k| 30, 29, 13, 13, 16, 9];
> ShiodaAlgebraicInvariants(FJI);
[
    [ 30, 29, 13, 13, 16, 9, 14, 35, 0 ],
    [ 30, 29, 13, 13, 16, 9, 36, 32, 22 ],
    [ 30, 29, 13, 13, 16, 9, 36, 32, 23 ]
]
> JI := ($1)[1];
> HyperellipticCurveFromShiodaInvariants(JI);
Hyperelliptic Curve defined by y^2 = 19*x^8 + 13*x^7 + 29*x^6 +
    3*x^5 + 16*x^4 + 19*x^3 + x^2 + 27*x + 12
over GF(37)
Symmetric group acting on a set of cardinality 2
Order = 2
```

A last example over the rationals.

```
> P<x> := PolynomialRing(Rationals());
> f := x^8 + x^3 + 1;
> f2 := P ! (Evaluate(f, (x+1001)/(3*x+5))*(3*x+5)^8);
> f2;
6805*x^8 + 827242*x^7 + 765112882*x^6 + 306007035874*x^5 + 72331829214220*x^4 +
    5628736468095180 6*x^3 + 2816843187239852 9278*x^2 + 8056168289692716824758*x
    + 1008028056073190412776751
> I := ShiodaInvariants(f2);
> HyperellipticCurveFromShiodaInvariants(I);
Hyperelliptic Curve defined by y^2 = x^8 + x^3 + 1 over Rational Field
Symmetric group acting on a set of cardinality 2
Order = 2
```

137.18 Isomorphisms and Transformations

A hyperelliptic curve isomorphism is defined by a linear fractional transformation $t(x : z) = (ax + bz : cx + dz)$, a scale factor e, and a polynomial $u(x)$ of degree at most $g + 1$, where g is the genus of the curve. This data defines the isomorphism of weighted projective points

$$(x : y : z) \mapsto (ax + bz : ey + \tilde{u}(x, z) : cx + dz),$$

where $\tilde{u}$ is the degree $g+1$ homogenization of u. When not specified, the values of e and u are by default taken to be 1 and 0, respectively.

An isomorphism can be created from the parent structures by coercing a tuple $\langle [a, b, c, d], e, u\rangle$ into the structure of isomorphisms between two hyperelliptic curves, or by creating it as a transformation of a given curve, i.e., creating the codomain curve together with the isomorphism from the given data.

Note that due to the projective weighting of the ambient space of the curve, two equal isomorphisms may have different representations.

137.18.1 Creation of Isomorphisms

`Aut(C): CrvHyp -> PowAutSch`

Given a hyperelliptic curve C, this returns the structure of all automorphisms of the curve.

`Iso(C1, C2): CrvHyp, CrvHyp -> PowIsoSch`

Given hyperelliptic curves $C1$ and $C2$ of the same genus and base field, this returns the structure of all isomorphisms between them.

`Transformation(C, t): CrvHyp, [RngElt] -> CrvHyp, MapIsoSch`

`Transformation(C, u): CrvHyp, RngUPolElt -> CrvHyp, MapIsoSch`

`Transformation(C, e): CrvHyp, RngElt -> CrvHyp, MapIsoSch`

`Transformation(C, e, u): CrvHyp, RngElt, RngUPolElt -> CrvHyp, MapIsoSch`

`Transformation(C, t, e, u): CrvHyp, [RngElt], RngElt, RngUPolElt -> CrvHyp, MapIsoSch`

Returns the hyperelliptic curve C' which is the codomain of the isomorphism from the hyperelliptic curve C specified by the sequence of ring elements t, the ring element e, and the polynomial u, followed by the isomorphism to the curve.

Example: Transforming a Hyperelliptic Curve (ex-9a2a7d)

We create the hyperelliptic curve $y^2 = x^5 - 7$ and apply a transformation to it.

```
> P<x> := PolynomialRing(Rationals());
> H1 := HyperellipticCurve(x^5-7);
> H2, phi := Transformation(H1, [0,1,1,0], 1/2, x^2+1);
> H2;
Hyperelliptic Curve defined by y^2 + (-2*x^2 - 2)*y = -7/4*x^6 - x^4 - 2*x^2 +
    1/4*x - 1 over Rational Field
> phi;
(x : y : z) :-> (z : 1/2*y + x^2*z + z^3 : x)
> IsIsomorphic(H1, H2);
true (x : y : z) :-> (z : 1/2*y + x^3 + x*z^2 : x)
```

137.18.2 Invariants of Isomorphisms

`Parent(f): MapIsoSch -> PowIsoSch`

The "parent structure" of an isomorphism between two hyperelliptic curves (which contains all isomorphisms with the same domain and codomain as the given isomorphism).

`Domain(f): MapIsoSch -> CrvHyp`

The curve which is the domain of the given isomorphism.

`Codomain(f): MapIsoSch -> CrvHyp`

The curve which is the target of the given isomorphism.

137.18.3 Automorphism Group and Isomorphism Testing

This section details the features for computing isomorphisms between two curves and determining the group of automorphisms.

The genus 2 part of [LRS21] provides a function to determine the geometric automorphism group of a genus 2 curve in any characteristic by working with Cardona—Quer—Nart—Pujola invariants (see Subsection *Igusa Invariants*). This replaces the old function that only worked in odd (or 0) characteristics. They also provide a function that returns a list of all possible geometric automorphism groups for genus 2 curves over a given finite field and the number of isomorphism classes of curves with each possible group.

The genus 3 part of [LRS21] provides the same for genus 3 hyperelliptic curves, working with Shioda invariants. Here, the characteristic has to be 0 or greater than 7.

It is also possible to compute (geometric, reduced) isomorphisms and automorphisms by a fast direct computation in all characteristic different from 2 and all genus > 1.

IsIsomorphic(C1, C2): CrvHyp, CrvHyp -> BoolElt, MapIsoSch

```
CrvHypIso: Verbose                    Default: 3
```

This function returns **true** if and only if the hyperelliptic curves $C1$ and $C2$ are isomorphic *as hyperelliptic curves (i.e., preserving the involution)* over their common base field. If the curves are isomorphic, an isomorphism is returned.

AutomorphismGroup(C): CrvHyp -> GrpPerm, Map, Map

Given a hyperelliptic curve C of characteristic different from 2, the function returns a permutation group followed by an isomorphism to the group of automorphisms of the curve over its base ring. The curve must be of genus at least one, and the automorphism group is defined to consist of those automorphisms which commute with the hyperelliptic involution, i.e., which induce a well-defined automorphism of its quotient projective line. A third return value gives the action $C \times G \to C$.

Example: Automorphisms of a Genus One Curve (ex-faec5d)

We give an example of the computation of the automorphism group of a genus one hyperelliptic curve.

```
> P<x> := PolynomialRing(GF(3));
> C1 := HyperellipticCurve(x^3+x);
> G1, m1 := AutomorphismGroup(C1);
> #G1;
> [ m1(g) : g in G1 ];
[
    (x : y : z) :-> (x : y : z),
    (x : y : z) :-> (x : -y : z),
    (x : y : z) :-> (z : y : x),
    (x : y : z) :-> (z : -y : x)
]
```

We note that due to the weighted projective space, the same map may have a non-unique representation, however the equality function is able to identify equivalence on representations.

```
> f := m1(G1.3);
> f;
(x : y : z) :-> (z : y : x)
> g := Inverse(f);
> g;
(x : y : z) :-> (2*z : y : 2*x)
> f eq g;
true
```

We see that the geometric automorphism group is much larger. By base extending the curve to a quadratic extension, we find the remaining automorphisms of the curve.

```
> K<t> := GF(3,2);
> C2 := BaseExtend(C1, K);
> G2, m2 := AutomorphismGroup(C2);
> #G2;
48
> O := C2![1,0,0];
> auts := [ m2(g) : g in G2 ];
> [ f : f in auts | O@f eq O ];
[
```

(continues on next page)

(continued from previous page)

```
    (x : y : z) :-> (x : y : z),
    (x : y : z) :-> (x : -y : z),
    (x : y : z) :-> (x : t^2*y : 2*z),
    (x : y : z) :-> (x + t^6*z : t^2*y : 2*z),
    (x : y : z) :-> (x + t^6*z : y : z),
    (x : y : z) :-> (x + t^2*z : y : z),
    (x : y : z) :-> (x + t^2*z : t^2*y : 2*z),
    (x : y : z) :-> (x : t^6*y : 2*z),
    (x : y : z) :-> (x + t^6*z : t^6*y : 2*z),
    (x : y : z) :-> (x + t^6*z : -y : z),
    (x : y : z) :-> (x + t^2*z : -y : z),
    (x : y : z) :-> (x + t^2*z : t^6*y : 2*z)
]
> #$1;
12
```

Note that this curve is an example of a supersingular elliptic curve in characteristic 3. In the final computation we restrict to the automorphisms as an elliptic curve, i.e., those which fix the point at infinity — the identity element of the group law.

In the context of hyperelliptic curves of genus one, the group of automorphisms must stabilize the ramification points of the hyperelliptic involution. These are precisely the 2-torsion elements as an elliptic curve. So we have a group extension by the 2-torsion elements, acting by translation. Converting to an elliptic curve, we find that there are two 2-torsion elements over $\mathbb{F}_3$:

```
> E1 := EllipticCurve(C1);
> A1 := AbelianGroup(E1);
> A1;
Abelian Group isomorphic to Z/4
Defined on 1 generator
Relations:
    4*$.1 = 0
```

We see that two of the 2-torsion elements are defined over $\mathbb{F}_3$, and the remaining ones appear over the quadratic extension. So in the former case the automorphism group is an extension of the elliptic curve automorphism group (of order 2) by $\mathbb{Z}/2\mathbb{Z}$, and in latter case the automorphism group is an extension (of the group of order 12) by the abelian group isomorphic to $\mathbb{Z}/2\mathbb{Z} \times \mathbb{Z}/2\mathbb{Z}$.

Note that there exist other curve automorphisms given by translations by other torsion points (under the addition as an elliptic curve), but that do not commute with the hyperelliptic involution, hence do not enter into the hyperelliptic automorphism group.

IsIsomorphicHyperellipticCurves(X1, X2): CrvHyp, CrvHyp -> BoolElt, List

IsIsomorphicHyperellipticCurves(f1, f2): RngUPolElt, RngUPolElt -> BoolElt, List

```
geometric  : BoolElt                    Default: false
commonfield: BoolElt                    Default: true
covariant  : BoolElt                    Default: true
```

Returns a boolean indicating whether a matrix T and a scalar e exist that induce an isomorphism $X_1 \mapsto X_2$, or $y^2 = f_1(x) \mapsto y^2 = f_2(x)$, as well as a full list of all such pairs.

If `geometric` is set to `true`, then the set of isomorphisms over the algebraic closure of the base field is returned. If `commonfield` is set to `false`, then the isomorphisms that are returned may be defined over different fields. Of `covariant` is set to `false`, then the calculation of the isomorphisms is performed by a direct methods instead of applying the usual covariant reduction.

Example: Isomorphisms Between Two Curves (ex-83144d)

We determine the isomorphisms $C_1 \to C_2$ for $C_1 : y^2 = x^{12} - 1$ and $C_2 : y^2 = x^{12} + 1$. The code

and the resulting output are as follows.

```
> P<x> := PolynomialRing(Rationals());
> C1 := HyperellipticCurve(x^12 - 1);
> C2 := HyperellipticCurve(x^12 + 1);
> test, _ := IsIsomorphicHyperellipticCurves(C1, C2);
> test;
false
> test, _ := IsIsomorphicHyperellipticCurves(C1, C2 : geometric := true);
> test;
true
```

IsomorphismsOfHyperellipticCurves(X1, X2): CrvHyp, CrvHyp -> List

IsomorphismsOfHyperellipticCurves(f1, f2): RngUPolElt, RngUPolElt -> List

```
geometric  : BoolElt                    Default: false
commonfield: BoolElt                    Default: true
covariant  : BoolElt                    Default: true
```

Returns a full list of pairs of matrices T and scalars e that induce an isomorphism $X_1 \mapsto X_2$, or $y^2 = f_1(x) \mapsto y^2 = f_2(x)$.

If `geometric` is set to `true`, then the set of isomorphisms over the algebraic closure of the base field is returned. If `commonfield` is set to `false`, then the isomorphisms that are returned may be defined over different fields. Of `covariant` is set to `false`, then the calculation of the isomorphisms is performed by a direct methods instead of applying the usual covariant reduction.

For more details, see [LRS12].

AutomorphismsOfHyperellipticCurve(X): CrvHyp -> List

AutomorphismsOfHyperellipticCurve(f): RngUPolElt -> List

```
geometric  : BoolElt                    Default: false
commonfield: BoolElt                    Default: true
covariant  : BoolElt                    Default: true
```

Returns a full list of pairs of matrices T and scalars e that induce an automorphism of the curve X, or the curve $y^2 = f(x)$.

If `geometric` is set to `true`, then the set of isomorphisms over the algebraic closure of the base field is returned. If `commonfield` is set to `false`, then the isomorphisms that are returned may be defined over different fields. Of `covariant` is set to `false`, then the calculation of the isomorphisms is performed by a direct methods instead of applying the usual covariant reduction.

AutomorphismGroupOfHyperellipticCurve(X, Autos): CrvHyp, List -> GrpPerm, Map

AutomorphismGroupOfHyperellipticCurve(f, Autos): RngUPolElt, List -> GrpPerm, Map

```
explicit: BoolElt                     Default: false
```

Return the automorphisms group defined by the sequence *Autos*, as a permutation group (and its representation as automorphisms of the curve if `explicit` is set to `true`).

AutomorphismGroupOfHyperellipticCurve(X): CrvHyp -> GrpPerm, Map

AutomorphismGroupOfHyperellipticCurve(f): RngUPolElt -> GrpPerm, Map

```
geometric: BoolElt                    Default: false
explicit : BoolElt                    Default: false
```

Return the automorphisms group of the curve X or $y^2 = f(x)$, as a permutation group (and its representation if explicit is set to true).

If `geometric` is set to `true`, then the set of isomorphisms over the algebraic closure of the base field is returned.

Example: Automorphisms over a Finite Field (ex-aab802)

We determine automorphism group of the curve $C_1 : y^2 = x^{12} - 1$ over the finite field $\mathbb{F}_{11}$.

```
> P<x> := PolynomialRing(GF(11));
> C := HyperellipticCurve(x^12 - 1);
> aut, phi := AutomorphismGroupOfHyperellipticCurve(C : geometric := true,
>               explicit := true);
> aut;
Permutation group aut acting on a set of cardinality 2640
Order = 2640 = 2^4 * 3 * 5 * 11
> GroupName(aut);
C2.PSL(2,11).C2
> [phi(aut.i) : i in [1..Ngens(aut)]];
[
    <
        [      1       0]
        [      0       1],

        10
    >,

    <
        [      1  $.1^27]
        [ $.1^77  $.1^20],

        4
    >,

    <
        [      1  $.1^98]
        [ $.1^28  $.1^30],

        8
    >,

    <
        [      1 $.1^115]
        [  $.1^5      10],

        $.1^6
    >,

    <
        [      1  $.1^85]
        [ $.1^35      10],

        $.1^6
    >
]
```

GeometricAutomorphismGroup(C): CrvHyp -> GrpPerm

GeometricAutomorphismGroupFromIgusaInvariants(GI): SeqEnum -> SeqEnum, SeqEnum

Given a hyperelliptic curve C of genus 2 or 3 the function returns a finitely-presented group isomorphic to the geometric automorphism group, i.e., the automorphism group of the curve over an algebraic closure of its base field.

The method used for genus 2 is to compute the Cardona—Quer—Nart—Pujola invariants (see Subsection *Igusa Invariants*) of C and use the classification of the possible automorphism groups in terms of the invariants. See [SV01] and [CQ05b] for the odd (and 0) characteristic case and [CNP05a] for characteristic 2.

The method used for genus 3 is to compute the Shioda invariants (see Subsection *Shioda Invariants*) of C and use the classification of the possible automorphism groups in terms of the invariants. For this case, the base field must have characteristic 0 or ≥ 11.

There is also a genus 2 version where the argument is the sequence GI of Cardona—Quer—Nart—Pujola invariants of a curve rather than the actual curve. This avoids actually constructing the curve, in case the user is starting from the invariants.

The functions are part the package [LRS21] contributed by Reynald Lercier, Christophe Ritzenthaler, and Jeroen Sijsling. For more details, see [LRS12, MSSV02].

`GeometricAutomorphismGroupFromShiodaInvariants(JI): SeqEnum -> GrpPerm`

There is a variant of the last intrinsic for genus 3 curves where the argument is the sequence JI of Shioda invariants of a curve rather than the actual curve. This avoids actually constructing the curve, in case the user is starting from the invariants. The same restrictions on the characteristic of the base field apply.

Example: Geometric Automorphism Groups `(ex-37ebfa)`

We give examples of the computation of the geometric automorphism group of a genus two and a genus three hyperelliptic curve.

```
> P<x> := PolynomialRing(RationalField());
> f := x^6+x^3+13;
> C := HyperellipticCurve(f);
> time GeometricAutomorphismGroup(C);
Permutation group acting on a set of cardinality 6
Order = 12 = 2^2 * 3
    (1, 2, 3, 4, 5, 6)
    (1, 6)(2, 5)(3, 4)
Time: 0.010
> f := x^8-1;
> C1 := HyperellipticCurve(f);
> GeometricAutomorphismGroup(C1);
GrpPC of order 32 = 2^5
PC-Relations:
    $.1^2 = $.4,
    $.3^2 = $.5,
    $.2^$.1 = $.2 * $.3,
    $.3^$.1 = $.3 * $.5,
    $.3^$.2 = $.3 * $.5
```

Note that `AutomorphismGroup` can be used to retrieve the same (and more!) information but this can be much slower.

```
> aut := AutomorphismGroup(C);
> aut;
Symmetric group aut acting on a set of cardinality 2
Order = 2
  (1, 2)
  Id(aut)
```

We need to extend the field!

```
> Qbar := AlgebraicClosure();
> Cbar := BaseChange(C, Qbar);
> time autbar := AutomorphismGroup(Cbar);
Time: 332.290
> autbar;
Permutation group autbar acting on a set of cardinality 12
Order = 12 = 2^2 * 3
  (1, 2)(3, 4)(5, 6)(7, 8)(9, 10)(11, 12)
  Id(autbar)
  (1, 3)(2, 4)(5, 7)(6, 8)(9, 11)(10, 12)
  (1, 5, 9)(2, 6, 10)(3, 11, 7)(4, 12, 8)
  (1, 7)(2, 8)(3, 9)(4, 10)(5, 11)(6, 12)
  (1, 9, 5)(2, 10, 6)(3, 7, 11)(4, 8, 12)
  (1, 11)(2, 12)(3, 5)(4, 6)(7, 9)(8, 10)
> IdentifyGroup(autbar);
<12, 4>
```

GeometricAutomorphismGroupGenus2Classification(F): FldFin -> SeqEnum, SeqEnum

Given a finite field F (of any characteristic), the function returns two sequences. The first gives a list of all possible geometric automorphism groups for genus 2 curves defined over F and the second gives the corresponding number of isomorphism (over $\bar{F}$, the algebraic closure of F) classes of F-curves having the given automorphism group. The groups are represented as finitely-presented groups.

This function belong to the genus 2 part of [LRS21] and is based on the classification analysis in [Car03a] and [CNP05a].

GeometricAutomorphismGroupGenus3Classification(F): FldFin -> SeqEnum, SeqEnum

Given a finite field F of characteristic ≥ 11, the function returns two sequences. The first gives a list of all possible geometric automorphism groups for genus 3 curves defined over F and the second gives the corresponding number of isomorphism (over $\bar{F}$, the algebraic closure of F) classes of F-curves having the given automorphism group. The groups are represented as permutation groups.

This function belongs to the genus 3 part of [LRS21]. For more details, see also [LRS12].

Example: Classifying Genus 2 Automorphism Groups (ex-56c040)

We determine the possible (geometric) automorphism groups for genus 2 curves over $\mathbb{F}_2$.

```
> gps,ncls :=  GeometricAutomorphismGroupGenus2Classification(GF(2));
> [#gp : gp in gps];
[ 2, 12, 32, 160 ]
> ncls;
[ 5, 1, 1, 1 ]
```

137.18.4 Twisting Hyperelliptic Curves

TwistsOfHyperellipticPolynomials(f: parameters): RngUPolElt -> SeqEnum[RngUPolElt], GrpPerm

TwistsOfHyperellipticPolynomials(fh): SeqEnum[RngUPolElt] -> SeqEnum, GrpPerm

```
AutomorphismGroup: BoolElt                    Default: false
```

TwistedHyperellipticPolynomialsFromShiodaInvariants(JI: parameters): SeqEnum[FldFinElt] -> SeqEnum, GrpPerm

RationalModel: BoolElt Default: true

When f is a polynomial defining a hyperelliptic curve $C : y^2 = f(x)$ over a finite field, these functions return the list of polynomials g such that $y^2 = g(x)$ is a twist of C (i.e. a curve over k isomorphic to C over $\bar{k}$). Unless the genus of C is 2 or 3, the finite field is required to be of characteristic not 2, but in these cases it is also possible to compute the lists of twists of $y^2 + h(x)y = f(x)$ given f and h. In all cases, the geometric reduced automorphism groups is returned as an abstract group.

There is also a version where the argument is JI, the Shioda invariants of a genus 3 curve (see Section *Shioda Invariants*) over a finite field.

For more details, see [CN07].

Twists(C: parameters): CrvHyp -> SeqEnum[CrvHyp], GrpPerm

TwistsFromIgusaInvariants(JI: parameters): SeqEnum[FldFinElt] -> SeqEnum[CrvHyp], GrpPerm

TwistsFromG2Invariants(JI: parameters): SeqEnum[FldFinElt] -> SeqEnum[CrvHyp], GrpPerm

TwistsFromShiodaInvariants(JI: parameters): SeqEnum[FldFinElt] -> SeqEnum[CrvHyp], GrpPerm

AutomorphismGroup: BoolElt Default: false

For C a hyperelliptic curve over a finite field of characteristic not 2 (unless the genus of C is 2 or 3) or a smooth plane quartic over any finite field, the function Twists return the list of all twists of C and the reduced geometric automorphism group in the hyperelliptic case or the geometric automorphism group in the smooth plane quartic case, as an abstract group.

There is also a version where the argument is JI, the sequence of Igusa or Cardona—Quer—Nart—Pujola invariants of a genus 2 curve (see Section *Igusa Invariants*) or the sequence of Shioda invariants of a genus 3 curve (see Section *Shioda Invariants*) over a finite field, which returns the full set of isomorphism classes (twists) of curves over k with the given invariants.

Twists(C, Autos: parameters): Crv, SeqEnum -> SeqEnum[Crv], GrpPerm

AutomorphismGroup: BoolElt Default: false

Compute the twists of a curve over a finite field with respect to a subgroup generated by geometric automorphisms (given by a list of N x N matrices) which act on it linearly.

HyperellipticPolynomialsFromShiodaInvariants(JI): SeqEnum -> SeqEnum, GrpPerm

Computes and returns all twists of a genus 3 hyperelliptic curve and its geometric automorphism group corresponding to a sequence of Shioda invariants JI (see Section *Shioda Invariants*) over a finite field of characteristic different from 5. In fact the first return value is a sequence of polynomials $f(x)$ (or of couple f, h in characteristic 2) of degree 7 or 8 such that the twisted curves correspond to $y^2 = f(x)$ (or to $y^2 + h(x)y = f(x)$ in characteristic 2). The reason for this is that JI could be a *singular* set of invariants corresponding to polynomials f with discriminant zero. In that case, these do not correspond to hyperelliptic curves, but it might be useful to get the full set of twists anyway.

Example: Twists over a Finite Field (ex-d7106f)

We determine the twists of the hyperelliptic curve $C : y^2 = x^{12} - 1$ over the finite field $\mathbb{F}_{11}$.

```
> P<x> := PolynomialRing(GF(11));
> C := HyperellipticCurve(x^12 - 1);
> Ts, Aut := Twists(C : AutomorphismGroup := true);
> Ts;
[
```

(continues on next page)

(continued from previous page)

```
    Hyperelliptic Curve defined by y^2 = x^12 + 10 over GF(11),
    Hyperelliptic Curve defined by y^2 = x^12 + 10*x^11 + 5*x + 3 over GF(11),
    Hyperelliptic Curve defined by y^2 = x^12 + 3*x^11 + 8*x + 8 over GF(11),
    Hyperelliptic Curve defined by y^2 = x^12 + 6*x^11 + 3*x over GF(11),
    Hyperelliptic Curve defined by y^2 = x^12 + 5*x^11 + 7*x + 3 over GF(11),
    Hyperelliptic Curve defined by y^2 = 2*x^12 + 10*x^11 + 3*x + 6 over GF(11),
    Hyperelliptic Curve defined by y^2 = x^12 + 7*x^11 + 5*x over GF(11),
    Hyperelliptic Curve defined by y^2 = x^11 + 10*x over GF(11),
    Hyperelliptic Curve defined by y^2 = x^12 + 3*x^11 + 7*x + 4 over GF(11),
    Hyperelliptic Curve defined by y^2 = x^12 + 2*x^11 + 9*x + 5 over GF(11),
    Hyperelliptic Curve defined by y^2 = x^12 + 4*x^11 + 9*x + 5 over GF(11),
    Hyperelliptic Curve defined by y^2 = x^12 + 4*x^11 + 5*x + 4 over GF(11),
    Hyperelliptic Curve defined by y^2 = x^12 + 4*x^11 + 2*x + 3 over GF(11),
    Hyperelliptic Curve defined by y^2 = x^12 + 7*x^11 + 7*x + 9 over GF(11)
]
> Aut;
Permutation group Aut acting on a set of cardinality 12
Order = 1320 = 2^3 * 3 * 5 * 11
    (3, 4, 7, 5, 9)(6, 10, 8, 11, 12)
    (3, 6, 5, 11, 4, 10, 9, 12, 7, 8)
    (2, 3)(4, 5)(7, 12)(8, 10)(9, 11)
    (1, 2)(4, 9)(5, 7)(6, 8)(11, 12)
> GroupName(Aut);
PSL(2,11).C2
```

Example: Twists of a Supersingular Curve (ex-d1281d)

We find all the twists of a supersingular genus 2 curve over $\mathbb{F}_2$.

```
> P<x> := PolynomialRing(GF(2));
> C := HyperellipticCurve(x^5,P!1);
> C;
Hyperelliptic Curve defined by y^2 + y = x^5 over GF(2)
> tws,auts := Twists(C);
> tws;
[
    Hyperelliptic Curve defined by y^2 + y = x^5 over GF(2),
    Hyperelliptic Curve defined by y^2 + y = x^5 + x^4 over GF(2),
    Hyperelliptic Curve defined by y^2 + y = x^5 + x^4 + 1 over GF(2)
]
> #auts; // auts is the geometric automorphism group of C
160
```

137.19 Reduced Automorphism Group and Reduced Isomorphism Testing

The function `IsGL2Equivalent` plays a central role in the isomorphism testing, and is documented here due to its central role in these computations.

`IsGL2Equivalent(f, g, n): RngUPolElt, RngUPolElt, RngIntElt -> BoolElt, SeqEnum`

This function returns `true` if and only if f and g are in the same $\mathrm{GL}_2(k)$-orbit, where k is the coefficient field of their parent, modulo scalars. The polynomials are considered as homogeneous polynomials of degree n, where n must be at least 4. The second return value is the sequence of all matrix entries $[a, b, c, d]$ such that $g(x)$ is a constant times $f\left((ax+b)/(cx+d)\right)(cx+d)^n$.

`IsGL2EquivalentExtended(f1, f2, deg): RngUPolElt, RngUPolElt, RngIntElt -> BoolElt, List`

```
geometric   : BoolElt                    Default: false
commonfield: BoolElt                     Default: true
covariant   : BoolElt                    Default: true
```

Returns a boolean indicating whether a matrix T exists such that the change of variable induced on $f1$ by T, $f1 * T$, is a multiple of $f2$, as well as a full list of all such matrices.

If `geometric` is set to `true`, then the set of isomorphisms over the algebraic closure of the base field is returned. If `commonfield` is set to `false`, then the isomorphisms that are returned may be defined over different fields. Of `covariant` is set to `false`, then the calculation of the isomorphisms is performed by a direct methods instead of applying the usual covariant reduction.

For more details, see [LRS12].

IsReducedIsomorphicHyperellipticCurves(X1, X2): CrvHyp, CrvHyp -> BoolElt, List
IsReducedIsomorphicHyperellipticCurves(f1, f2): RngUPolElt, RngUPolElt -> BoolElt, List

```
geometric   : BoolElt                    Default: false
commonfield: BoolElt                     Default: true
covariant   : BoolElt                    Default: true
```

Returns a boolean indicating whether a matrix T exists that induces an isomorphism $f1(x) \to f2(x)$ ($f1$ and $f2$ resp. define $X1$ and $X2$), as well as a full list of all such matrices.

If `geometric` is set to `true`, then the set of isomorphisms over the algebraic closure of the base field is returned. If `commonfield` is set to `false`, then the isomorphisms that are returned may be defined over different fields. Of `covariant` is set to `false`, then the calculation of the isomorphisms is performed by a direct methods instead of applying the usual covariant reduction.

ReducedIsomorphismsOfHyperellipticCurves(X1, X2): CrvHyp, CrvHyp -> List
ReducedIsomorphismsOfHyperellipticCurves(f1, f2): RngUPolElt, RngUPolElt -> List

```
geometric   : BoolElt                    Default: false
commonfield: BoolElt                     Default: true
covariant   : BoolElt                    Default: true
```

Returns a full list of matrices T that induce an isomorphism $f1(x) \to f2(x)$ ($f1$ and $f2$ resp. define $X1$ and $X2$).

If `geometric` is set to `true`, then the set of isomorphisms over the algebraic closure of the base field is returned. If `commonfield` is set to `false`, then the isomorphisms that are returned may be defined over different fields. Of `covariant` is set to `false`, then the calculation of the isomorphisms is performed by a direct methods instead of applying the usual covariant reduction.

For more details, see [LRS12].

ReducedAutomorphismsOfHyperellipticCurve(X): CrvHyp -> List
ReducedAutomorphismsOfHyperellipticCurve(f): RngUPolElt -> List

```
geometric   : BoolElt                    Default: false
commonfield: BoolElt                     Default: true
covariant   : BoolElt                    Default: true
```

Return the automorphism group of the defining polynomial of X, as a full list of matrices T.

If `geometric` is set to `true`, then the set of isomorphisms over the algebraic closure of the base field is returned. If `commonfield` is set to `false`, then the isomorphisms that are returned may be defined over different fields. Of `covariant` is set to `false`, then the calculation of the isomorphisms is performed by a direct methods instead of applying the usual covariant reduction.

ReducedAutomorphismGroupOfHyperellipticCurve(X, Autos): CrvHyp, List -> GrpPerm, Map

ReducedAutomorphismGroupOfHyperellipticCurve(f, Autos): RngUPolElt, List -> GrpPerm, Map

```
explicit: BoolElt                    Default: false
```

Return the automorphisms group defined by the sequence *Autos*, as a permutation group (and its representation if `explicit` is set to `true`).

ReducedAutomorphismGroupOfHyperellipticCurve(X): CrvHyp -> GrpPerm, Map

ReducedAutomorphismGroupOfHyperellipticCurve(f): RngUPolElt -> GrpPerm, Map

Return the automorphisms group of the curve $y^2 = f(x)$, as a permutation group (and its representation if `explicit` is set to `true`).

137.20 Bibliography

- [Bas15] Romain Basson. *Arithmétique des espaces de modules des courbes hyperelliptiques de genre 3 en caractéristique positive*. PhD thesis, Université de Rennes 1, Rennes, 2015.
- [Bru02] N. R. Bruin. *Chabauty methods and covering techniques applied to generalized Fermat equations*. Volume 133 of CWI Tract. Stichting Mathematisch Centrum Centrum voor Wiskunde en Informatica, Amsterdam, 2002. ISBN 90-6196-508-X. Dissertation, University of Leiden, Leiden, 1999.
- [BD09] Nils Bruin and Kevin Doerksen. The arithmetic of genus two curves with (4,4)-split Jacobians. ArXiv preprint. http://arxiv.org/abs/0902.3480, 2009.
- [BS09] Nils Bruin and Michael Stoll. Two-cover descent on hyperelliptic curves. *Math. Comp.*, 78:2347–2370, 2009.
- [Car03a] G. Cardona. On the number of curves of genus 2 over a finite field. *Finite Fields and Their Applications*, 9(4):505–526, 2003.
- [CN07] G. Cardona and E. Nart. Zeta function and cryptographic exponent of supersingular curves of genus 2. In *Pairing-based cryptography—Pairing 2007*, volume 4575 of Lecture Notes in Comput. Sci., pages 132–151. Springer, Berlin, 2007.
- [CNP05a] G. Cardona, E. Nart, and J. Pujolas. Curves of genus two over fields of even characteristic. *Mathematische Zeitschrift*, 250:177–201, 2005.
- [CQ05b] Gabriel Cardona and Jordi Quer. Field of moduli and field of definition for curves of genus 2. *Lecture Notes Ser. Comput.*, 13:71–83, 2005.
- [CF96] J.W.S. Cassels and E.V. Flynn. *Prolegomena to a Middlebrow Arithmetic of Curves of Genus* 2. Cambridge University Press, Cambridge, 1996.
- [Cre12] Brendan Creutz. Explicit descent in the Picard group of a cyclic cover of the projective line. In Everett Howe and Kiran Kedlaya, editors, *ANTS X: Proceedings of the Tenth Algorithmic Number Theory Symposium*, volume 1 of OBS. Mathematics Sciences Publishers, 2012.
- [GH] P. Gaudry, R. Harley. "Counting Points on Hyperelliptic Curves over Finite Fields." 313–332.
- [Got59] E. Gottschling. Explizite Bestimmung der Randflächen des Fundamentalbereiches der Modulgruppe zweiten Grades. *Math. Annalen*, 138:103–124, 1959.
- [Hol06] David Holmes. Canonical heights on hyperelliptic curves. Preprint http://arxiv.org/abs/1004.4503, 2006.
- [Hub06] Hendrik Hubrechts. Point counting in families of hyperelliptic curves. Preprint http://arxiv.org/abs/math.NT/0601438, 2006.
- [Igu60a] J. Igusa. "Arithmetic variety of moduli for genus two." 72, 612–649. 1960.
- [Ked01] Kiran S. Kedlaya. Counting Points on Hyperelliptic Curves using Monsky-Washnitzer Cohomology. *J. Ramanujan Math. Soc.*, 16:323 – 338, 2001.

- [KLL+18] Pnar Klıçer, Hugo Labrande, Reynald Lercier, Christophe Ritzenthaler, Jeroen Sijsling, and Marco Streng. Plane quartics over $\mathbb{Q}$ with complex multiplication. *Acta Arith.*, 185(2):127–156, 2018.
- [LL06] R. Lercier and D. Lubicz. A Quasi-Quadratic Time Algorithm for Hyperelliptic Curve Point Counting. *to appear*, 2006.
- [LRS12] R. Lercier, C. Ritzenthaler, and J. Sijsling. Fast computation of isomorphisms of hyperelliptic curves and explicit descent. In E. W. Howe and K. S. Kedlaya, editors, *Proceedings of the Tenth Algorithmic Number Theory Symposium*, 463–486. Mathematical Sciences Publishers, 2012.
- [LR12] Reynald Lercier and Christophe Ritzenthaler. Hyperelliptic curves and their invariants: geometric, arithmetic and algorithmic aspects. *J. Algebra*, 372:595–636, 2012. http://dx.doi.org/10.1016/j.jalgebra.2012.07.054.
- [LRS21] Reynald Lercier, Christophe Ritzenthaler, and Jeroen Sijsling. Functionalities for genus 2 and 3 curves. https://arxiv.org/abs/2102.04372, 2021. arXiv:2102.04372.
- [MS16] J.S. Müller and M. Stoll. Canonical heights on genus two Jacobians. *Algebra & Number Theory*, 10:2153–2234, 2016. https://doi.org/10.2140/ant.2016.10.2153. doi:10.2140/ant.2016.10.2153.
- [Mae90a] T. Maeda. On the invariant fields of binary octavics. *Hiroshima Math. J.*, 20:619–632, 1990.
- [MSSV02] K. Magaard, T. Shaska, S. Shpectorov, and H. Völklein. The locus of curves with prescribed automorphism group. *Sūrikaisekikenkyūsho Kōkyūroku*, pages 112–141, 2002. Communications in arithmetic fundamental groups (Kyoto, 1999/2001).
- [Mes91] J.-F. Mestre. Construction de courbes de genre 2 à partir de leurs modules. In T. Mora and C. Traverso, editors, *Effective methods in algebraic geometry*, volume 94 of Progr. Math., 313–334. Birkhäuser, 1991. Proc. Congress in Livorno, Italy, April 17–21, 1990.
- [Mum84] David Mumford. *Tata Lectures on Theta II.* Volume 43 of Progress in Mathematics. Birkhäuser, 1984.
- [Mul10a] Jan Steffen Müller. *Computing canonical heights on Jacobians.* PhD Thesis, Universität Bayreuth, 2010. available at http://www.math.uni-hamburg.de/home/js.mueller/.
- [Mul10b] Jan Steffen Müller. Explicit Kummer surface formulas for arbitrary characteristic. *LMS J. Comput. Math.*, 4:47–64, 2010.
- [PS97b] Bjorn Poonen and Ed Schaefer. Explicit descent for Jacobians of cyclic covers of the projective line. *J. Reine Angew. Math.*, 488:141–188, 1997.
- [PS99] Bjorn Poonen and Michael Stoll. The Cassels-Tate pairing on polarized abelian varieties. *Ann. of Math. (2)*, 150(3):1109–1149, 1999.
- [SV01] Tony Shaska and Völklein. *Elliptic subfields and automorphisms of genus* 2 *function fields.* Springer, 2001, http://au.arxiv.org/abs/math.AG/0107142.
- [Shi67] T. Shioda. On the Graded Ring of Invariants of Binary Octavics. *Am. Jour. Math.*, 89(4):1022–1046, 1967.
- [Sma97] N. P. Smart. S-unit equations, binary forms and curves of genus 2. *Proc. London Math. Soc. (3)*, 75(2):271–307, 1997.
- [Smi05] Benjamin A. Smith. *Explicit endomorphisms and correspondences.* PhD thesis, University of Sydney, 2005. http://hdl.handle.net/2123/1066.
- [Sto99b] M. Stoll. On the height constant for curves of genus two. *Acta Arithmetica*, 90:183–201, 1999. https://doi.org/10.4064/aa-90-2-183-201. doi:10.4064/aa-90-2-183-201.
- [Sto01] Michael Stoll. Implementing 2-descent for Jacobians of hyperelliptic curves. *Acta Arith.*, 98(3):245–277, 2001.
- [Ver02] F. Vercauteren. Computing zeta functions of hyperelliptic curves over finite fields of characteristic 2. In *Advances in cryptology—CRYPTO 2002*, volume 2442 of LNCS, pages 369–384. Springer, Berlin, 2002.

- [Wam99] P. Van Wamelen. Examples of Genus Two CM Curves Defined over the Rationals. *Mathematics of Computation*, 68:307–320, 1999.
- [Wet97] J. L. Wetherell. *Bounding the number of rational points on certain curves of high rank*. PhD thesis, U.C. Berkeley, 1997.

Chapter 138

Hypergeometric Motives

138.1 Introduction

Let $\vec{\alpha}, \vec{\beta} \in \mathbf{C}^n$ be n-tuples (or multisets) of complex numbers. For arithmetic applications we will eventually take them to be rationals, and for purposes of monodromy will largely need only to consider them modulo 1.

Consider the generalised hypergeometric differential equation

$$z(\theta+\alpha_1)\cdots(\theta+\alpha_n)F(z) = (\theta+\beta_1-1)\cdots(\theta+\beta_n-1)F(z), \qquad \theta = z\frac{d}{dz},$$

whose only singularities are regular at 0, 1, and ∞. For simplicity of exposition, we assume that the β's are distinct modulo 1, when a basis of solutions around $z=0$ is given by

$$z^{1-\beta_i}{}_nF_{n-1}\left(\begin{matrix}\alpha_1-\beta_i+1,\ldots,\alpha_n-\beta_i+1\\ \beta_1-\beta_i+1,..\vee..,\beta_n-\beta_i+1\end{matrix}\,\middle|\, z\right)$$

for $i = 1\ldots n$, and the ith term $\beta_i-\beta_i+1$ is suppressed. The generalised hypergeometric function ${}_nF_{n-1}$ is given by

$${}_nF_{n-1}\left(\begin{matrix}a_1,\ldots,a_n\\ b_1,\ldots,b_{n-1}\end{matrix}\,\middle|\, z\right) = \sum_{k=0}^{\infty}\frac{(a_1)_k\cdots(a_n)_k}{(b_1)_k\cdots(b_{n-1})_k}\frac{z^k}{k!},$$

where the Pochhammer symbol is given by $(x)_k = (x)(x+1)\cdots(x+k-1)$; the $k!$ in the denominator of the above display can thus be thought of as $(1)_k$, which was the suppressed term. Note that shifting all the α and β by some fixed amount keeps the ${}_nF_{n-1}$ expression the same, while only modifying the $z^{1-\beta_i}$ term. Also, switching α and β can be envisaged in terms of the map $z \to 1/z$ that swaps 0 and ∞.

A theorem of Pochhammer says that the above differential equation has $(n-1)$ independent *holomorphic* solutions around $z=1$. Let G denote the fundamental group of the thrice punctured Riemann sphere, and $V_{\vec{\alpha},\vec{\beta}}$ the solution space around a base point. We have a monodromy representation $M : G \to GL_n(V_{\vec{\alpha},\vec{\beta}})$. Writing $g_0, g_1, g_\infty \in G$ for loops about 0,1,∞, we find that $M(g_0)$ has eigenvalues $e^{-2\pi i\beta_j}$ and $M(g_\infty)$ has eigenvalues $e^{2\pi i\alpha_j}$, implying that we are mainly concerned with $\vec{\alpha}$ and $\vec{\beta}$ only modulo 1. Indeed, one can note that if we take $F = {}_nF_{n-1}(\vec{a},\vec{b}|z)$ and $\vec{x} \in \mathbf{Z}^n, \vec{y} \in \mathbf{Z}^{n-1}$, then generically ${}_nF_{n-1}(\vec{a}+\vec{x},\vec{b}+\vec{y}|z)$ is a linear combination of rational functions times derivatives of F (this is a contiguity relation). Meanwhile, the above fact from Pochhammer implies that $M(g_1)$ must have $(n-1)$ eigenvalues equal to 1 (all with independent eigenvectors), and so this element is a pseudo-reflection.

It turns out that if $H \subseteq GL_n(\mathbf{C})$ is generated by A and B with AB^{-1} a pseudo-reflection, the H-action on $\mathbf{C}^n$ is irreducible if and only if A and B have disjoint sets of eigenvalues. This is equivalent to all the $\alpha_i - \beta_j$ being nonintegral. Moreover, in his 1961 Amsterdam thesis, Levelt showed that, given any eigenvalues, there are (up to conjugacy) unique A and B realising these eigenvalues with AB^{-1} a pseudo-reflection. (Much of the above comes from notes of Beukers.)

For arithmetic purposes, one usually also desires that the eigenvalues be roots of unity and the sets of them be Galois-invariant. Thus we can specify hypergeometric data H by (say) two products of cyclotomic polynomials, these products being coprime and of equal degree. Given such an H, Rodriguez-Villegas conjectures the existence of a family of pure motives (defined over $\mathbf{Q}$), for which the trace of Frobenius at good primes is given by a hypergeometric sum defined by Katz [Kat90] (see also [Kat96]). For each rational $t \neq 0, 1$, there should be a motive H_t whose L-function satisfies a functional equation of a prescribed type, with the Euler factors at good primes given in terms of Gauss sums (the bad Euler factors are less understood, and depend on deformation theory).

One can also relate such motives to more traditional objects in many cases. For instance, there is one hypergeometric datum in degree 1, which can be specified by $\alpha = [\frac{1}{2}]$ and $\beta = [0]$, these being rationals corresponding to the second and first cyclotomic polynomials respectively. The L-function here corresponds to the quadratic field $\mathbf{Q}(\sqrt{t(t-1)})$. In degree 2 there are 13 such data, of which 3 are of weight 0 (see below) and give Artin representations of number fields, while the other 10 are of weight 1, and yield elliptic curves (explicitly calculated by Cohen). An example in higher degree is $\alpha = [\frac{1}{5}, \frac{2}{5}, \frac{3}{5}, \frac{4}{5}]$ and $\beta = [0, 0, 0, 0]$, corresponding respectively to the 5th cyclotomic polynomial and the 4th power of the first cyclotomic polynomial, and this is associated to the Calabi-Yau quintic 3-fold given by $x_1^5 + x_2^5 + x_3^5 + x_4^5 + x_5^5 = 5tx_1x_2x_3x_4x_5$.

The weight w of a hypergeometric motive can be defined in terms of how much the α and β interlace (considered as roots of unity). In particular, if they are completely interlacing, then the weight is 0, and the resulting motive corresponds to an Artin representation. Write $D(x) = \#\{\alpha : \alpha \leq x\} - \#\{\beta : \beta \leq x\}$. Then $w + 1 = \max_x D(x) - \min_x D(x)$, so that the above 3-fold has weight 3 (from the four β's at 0). This weight controls how large the coefficients of the Euler factors will be.

The trace at q of a hypergeometric motive (for the parameter t) is given in terms of Gauss sums g_q over $\mathbf{F}_q$. Associated to hypergeometric data is a `GammaArray`, and one defines $G_q(r) = \prod_v g_q(-rv)^{\gamma_v}$, and also the `MValue` by $M = \prod_v v^{v\gamma_v}$. For primes p with $v_p(Mt) = 0$ the hypergeometric trace is then given by

$$U_q(t) = \frac{1}{1-q}\left(\sum_{r=0}^{q-2} \omega_p(Mt)^r Q_q(r)\right),$$

where ω_p is the Teichmüller character and $Q_q(r) = (-1)^{m_0} q^{D+m_0-m_r} G_q(r)$ where m_r is the multiplicity of $\frac{r}{q-1}$ in the β and D is a scaling parameter that involves the Hodge structure (one expression is $m_0 = w + 1 - 2D$). One uses p-adic Γ-functions to expedite the computation of the above Gauss sums (indeed, the above gives the hypergeometric trace as a p-adic number, which one recognises as an integer via sufficiently high precision). The Euler factor is given by the standard recipe $E_p(T) = \exp\left(-\sum_n U_{p^n}(t)T^n/n\right)$, and this is a polynomial that satisfies a local functional equation.

138.2 Functionality

138.2.1 Creation Functions

`HypergeometricData(A, B): SeqEnum, SeqEnum -> HypGeomData`

`HypergeometricData(F, G): RngUPolElt, RngUPolElt -> HypGeomData`

`Print: MonStgElt` `Default: "cyclotomic"`

These are two of the principal ways of specifying hypergeometric data. The first takes two sequences A and B (of the same length) of rationals, which must be disjoint upon reduction modulo 1, and each of which must be Galois-invariant when taking the corresponding roots of unity (for instance, if $\frac{1}{6}$ is specified, $\frac{5}{6}$ is also given). The second takes two products F and G of cyclotomic polynomials, these products being coprime and of the same degree. Previous Magma versions could switch A and B in some cases; this is no longer the case.

The default can be specified with the `Print` vararg, the other option currently being `"alpha_beta"`.

There is now also some functionality for non-disjoint A and B, which mainly manifests itself at the L-series level.

`HypergeometricData(G): SeqEnum[RngIntElt] -> HypGeomData`

This is a third way to specify hypergeometric data, by giving a sequence of integers G such that $\sum_v vG[v] = 0$. Here we have $P_\alpha(T)/P_\beta(T) = \prod_v (T^v - 1)^{G[v]}$, and the polynomials can be determined via Möbius inversion.

`HypergeometricData(L): List -> HypGeomData`

This is a fourth way to specify hypergeometric data, by giving a *list* L of nonzero integers (with repetition possible) corresponding to the sequence L of the previous intrinsic, with negative integers for those where $L[v]$ is negative. The sum of the members of the list must be 0.

`HypergeometricData(F, G): SeqEnum[RngIntElt], SeqEnum[RngIntElt] -> HypGeomData`

This is a fifth way to specify hypergeometric data, by giving two arrays F and G of integers, corresponding to the cyclotomic polynomials to be used.

`HypergeometricData(E): SeqEnum[SeqEnum] -> HypGeomData`

This is a utility intrinsic that take a sequence E of two sequences and then passes these two sequences to the intrinsics above.

`Twist(H): HypGeomData -> HypGeomData`

This intrinsic takes hypergeometric data H, and adds $1/2$ to every element in α and β, returning new hypergeometric data. Magma no longer (ever) switches α and β when twisting.

`PrimitiveData(H): HypGeomData -> HypGeomData`

Given hypergeometric data H, return its primitive associated data. This is most easily described in terms of `GammaList`, dividing all the elements by the gcd.

`PossibleHypergeometricData(d): RngIntElt -> SeqEnum`

```
Weight        : RngIntElt                    Default: false
TwistMinimal  : BoolElt                      Default: false
CyclotomicData: BoolElt                      Default: false
Primitive     : RngIntElt                    Default: 0
```

Given a degree d, generate all possible examples of hypergeometric data of that degree, returned as a sequence of pairs of sequences, each sequence therein having d rationals. If `Weight` is specified, restrict to data of this weight. If `TwistMinimal` is specified, only give twist-minimal data. If `CyclotomicData` is specified, return the sequences of cyclotomic data rather than rationals. If `Primitive` is `true`, only return data that are primitive; if `Primitive` is a positive integer, return the data that have this imprimitivity.

138.2.2 Access Functions

`Weight(H): HypGeomData -> RngIntElt`

The weight of the given hypergeometric data H.

`Degree(H): HypGeomData -> RngIntElt`

The degree of the given hypergeometric data H.

`DefiningPolynomials(H): HypGeomData -> RngUPolElt, RngUPolElt`

The (products of cyclotomic) polynomials corresponding to α and β corresponding to hypergeometric data H.

`Bezoutian(H): HypGeomData -> RngIntElt`

The resultant of the defining polynomials of the hypergeometric data.

`CyclotomicData(H): HypGeomData -> SeqEnum, SeqEnum`

Returns two arrays of integers, specifying which cyclotomic polynomials occur for α and β corresponding to hypergeometric data H. Thus, for example, $\Phi_3\Phi_4^2\Phi_6$ would be represented by [3,4,4,6]).

`AlphaBetaData(H): HypGeomData -> SeqEnum, SeqEnum`

Returns two arrays of rationals, giving the α and β of the hypergeometric data H.

`MValue(H): HypGeomData -> FldRatElt`

This returns the scaling parameter M of the given hypergeometric data H. This is defined by taking $M_n = \prod_{d|n} d^{d\mu(n/d)}$ for the nth cyclotomic polynomial, and combining these into the products for α and β, and then dividing these. Another definition is $M = \prod_v v^{v\gamma_v}$.

`GammaArray(H): HypGeomData -> SeqEnum`

This returns an array of integers corresponding to γ_v, where these are defined by $P_\alpha(T)/P_\beta(T) = \prod_v (T^v - 1)^{\gamma_v}$. We also have $\sum_v v\gamma_v = 0$.

`GammaList(H): HypGeomData -> List`

This returns a *list* of integers corresponding to γ_v, where $\mathrm{sign}(\gamma_v) \cdot v$ appears in the list $|\gamma_v|$ times.

`H1 eq H2: HypGeomData, HypGeomDat -> BoolElt`

`H1 ne H2: HypGeomData, HypGeomDat -> BoolElt`

Two instances of hypergeometric data $H1$ and $H2$ are equal if they have the same α and β.

`IsPrimitive(H): HypGeomData -> BoolElt, RngIntElt`

Returns `true` if the given hypergeometric data H is primitive, and the index of imprimitivity. The latter is the gcd of the elements in the `GammaList`.

138.2.3 Functionality with L-series and Euler Factors

`HypergeometricTrace(H, t, q): HypGeomData, RngQZElt, RngIntElt -> RngIntElt`

Given a hypergeometric datum H, a rational $t \neq 0$, and a prime power $q = p^f$ for which p is good or multiplicative, return the hypergeometric trace. The intrinsic also works more generally when $v_p(Mt) = 0$, even if p is wild.

`HypergeometricTraceK(A, B, t, q): SeqEnum, SeqEnum, RngQZElt, RngIntElt -> FldPadElt`

`HypergeometricTraceK(A, B, t, q): SeqEnum, SeqEnum, FldPadElt, RngIntElt -> FldPadElt`

`Precision: RngIntElt`	`Default: 5`

Given α's and β's associated to a not-necessarily Galois datum, and a rational $t \neq 0$, and a prime power $q = p^f$ for which $v_p(t) = 0$ and p divides no denominator of the α's and β's, return the hypergeometric trace according to the p-adic Γ-function definition, namely that

$$H_q(\alpha, \beta | t) = \frac{q^D}{1-q} \sum_{r=0}^{q-2} \omega_p(t)^r \frac{q^{m_0}}{q^{m_r}} \frac{X_q(r)}{X_q(0)} \frac{(-p)^{T_f(r)}}{(-p)^{T_f(0)}}$$

where ω_p is the Teichmüller as before, m_r is the multiplicity of $-r/(q-1)$ in B (modulo 1), while

$$X_q(r) = \prod_{i=0}^{f-1} \frac{\prod_j \Gamma_p(\{p^i(\alpha_j + r/(q-1))\})}{\prod_j \Gamma_p(\{p^i(\beta_j + r/(q-1))\})}$$

and

$$T_f(r) = \sum_j \big[S_f(\alpha_j + r/(q-1)) - S_f(\beta_j + r/(q-1)) \big] \quad \text{where} \quad S_f(x) = \sum_{i=0}^{f-1} \{p^i x\}.$$

In the version where t is a p-adic, it must be compatible with q. The precision of t should exceed that with the `Precision` parameter, those this is not mandated, and could cause an incompatibility problem later in the computation.

This intrinsic is (much) slower than the optimized version in the Galois case.

`EulerFactor(H, t, p): HypGeomData, RngQZElt, RngIntElt -> RngUPolElt`

`Degree: RngIntElt`	`Default: 0`
`Check : BoolElt`	`Default: false`
`Fake : BoolElt`	`Default: false`

This intrinsic is the heart of the hypergeometric motive package. It takes hypergeometric data H, a rational $t \neq 0, 1$, and a prime p, and computes the pth Euler factor of the hypergeometric motive at t. This uses p-adic Γ-functions, as indicated by Cohen. The `Degree` vararg specifies how many terms in the Euler factor should be computed – if this is 0, then the whole polynomial is computed. The `Check` vararg allows one to turn off the use of the (local) functional equation that is used to expedite the computation process.

The `Fake` vararg allows one to compute the hypergeometric trace(s) for t with $v_p(Mt) = 0$ (including wild primes). Whether or not this is the actual Euler factor depends on how inertia acts. The use of this vararg inhibits the use of the local functional equation, but one can curtail via `Degree`, and apply it manually (if known).

In general, the given prime must not be wild, that is, it must not divide the denominator of any of the α or β.

At other bad primes, the Euler factor will depend upon the relevant monodromy. The primes for which $v_p(t-1) > 0$ can perhaps be called multiplicative, in that p should divide the conductor only once (this is related to the pseudoreflection). Since $v_p(Mt) = 0$ here, the Euler factor (of degree $d-1$) can be recovered by the p-adic Γ-function methods (even when $p = 2$). Also it is often possible to relate the (presumed) hypergeometric motive to objects from a deformation. When $v_p(t-1)$ is even and the weight is also even, the prime p is actually good and has a degree d Euler factor, even though the hypergeometric trace only gives one of degree $(d-1)$. In such a case, the `EulerFactor` intrinsic with the `Fake` vararg will return the part from the hypergeometric trace.

The p with $v_p(1/t) > 0$ correspond to monodromy at ∞. The associated inertia is given by the roots of unity with the β, with maximal Jordan blocks when eigenvalues are repeated. The same is true for p with $v_p(t) > 0$, where the monodromy is around 0 (so that the α are used). In Example *The Quintic Threefold*), an instance is given where the inertia is trivialised due to having $\zeta^{v_p(t)} = 1$.

We can compute the Euler factors at such tame primes as follows. Suppose that $t = t_0 p^{vm}$ with $v > 0$, where m occurs as a denominator of the α (similarly with $v < 0$ and β). Then one takes the smallest $q = p^f$ that is 1 mod m, and from the hypergeometric trace formula extracts the terms

$$\omega_p(Mt_0)^{j(q-1)/m} Q_q\left(\frac{j(q-1)}{m}\right)$$

for $0 \leq j < m$ with $\gcd(j, m) = 1$. Denoting these by η, we then have that $\prod_\eta (1 - \eta T^f)$ is an fth power (due to repetitions in the above extraction), and the fth root of this is the desired Euler factor of degree $\phi(m)$.

When m does not divide $v_p(t)$, the Euler factor from it is trivial. One then multiplies together all such Euler factors corresponding to the m from the α and β. Each m is only considered once, even if it appears multiple times in the `CyclotomicData`, as the Jordan blocks of the eigenvalues are maximal. Note that the local functional equation is not used for tame primes, though the computation should not be too onerous unless $q = p^f$ is large.

`LSeries(H, t): HypGeomData, RngQZElt -> LSer, LSer`

```
BadPrimes     : SeqEnum                    Default: []
HodgeStructure: HodgeStruc                 Default: false
GAMMA         : SeqEnum                    Default: []
Identify      : BoolElt                    Default: true
Precision     : RngIntElt                  Default: 0
Weight01      : BoolElt                    Default: false
QuadraticTwist: Any                        Default: false
PoleOrder     : RngIntElt                  Default: 0
SaveEuler     : RngIntElt                  Default: false
```

Given hypergeometric data H and a rational $t \neq 0, 1$, try to construct the L-series of the associated motive. This will usually need the Euler factors at wild primes to be specified. Everything else, including tame/multiplicative Euler factors and γ-factors, can be computed automatically by Magma (these can also be given respectively via `BadPrimes`, and `GAMMA` and/or `HodgeStructure`).

The Identify vararg indicates whether an attempt should be made to identify motives of weight 0 as Artin representations, and similarly with (hyper)elliptic curves for weight 1. The Weight01 vararg when true will Translate the L-series (essentially a Tate twist) so that the weight is 0 or 1. Setting Weight01 to an (odd) integer r will Translate so that the (motivic) weight is r plus the number of zero entries in the AlphaBetaData. A typical choice is $r = -1$.

The QuadraticTwist vararg can be used to take the tensor product with the given quadratic Dirichlet character. This can be given as a nonzero rational or as a character, or alternatively can be set to true, when Magma will use a default twisting factor. However, this option can conflict with BadPrimes (Magma does not know whether to apply such primes to the original L-function or the twist), and so should be used sparingly.

The PoleOrder vararg allows the user to specify that the given power of the (shifted) Riemann ζ-function is expected to divide the L-series of the hypergeometric motive. The routines will then act accordingly, decomposing the L-series into a factorisation and moving the poles into the ζ-function part.

Finally, the SaveEuler option takes a nonnegative integer (or a boolean), and indicates how large of primes should have their EulerFactor saved when computed. This is useful when (say) dealing with such L-function constructs as Symmetrization, for which the underlying work with hypergeometric traces is being done on one L-function, and then used multiple times. This option will be (silently) ignored if Magma is able to Identify the L-function as coming from somewhere else.

The intrinsic actually returns two L-series, the first corresponding to the disjoint parts of A and B, and the second an Artin part (weight 0) corresponding to the common part. For most purposes the second can be ignored (it will typically be the trivial LSeries).

Identification of Hypergeometric Data as Other Objects

ArtinRepresentation(H, t): HypGeomData, RngQZElt -> ArtRep

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

Given hypergeometric data H of weight 0 and a rational $t \neq 0, 1$, try to determine the associated Artin representation. This is implemented for all Belyi cases where the GammaList of H or its Twist has size 3 (this includes all cases up through degree 3), and for isolated cases (most due to Bartosz Nasrecki) in higher degree. When Check is true, good primes up to 100 have their Euler factors checked for correctness. When Optimize is true, the number field representation will be optimized.

EllipticCurve(H): HypGeomData -> CrvEll

EllipticCurve(H, t): HypGeomData, RngQZElt -> CrvEll

Given hypergeometric data H of degree 2 and weight 1 (there are 10 such families) and a rational $t \neq 0, 1$, return the associated elliptic curve, as catalogued by Cohen. When t is not given, return the result over a function field.

For each of the 10 families, the same function can be called for the corresponding imprimitive data of index r, and the result will generically be an elliptic curve over an extension of degree r. However, when the $x^r - 1/Mt$ splits, the intrinsic will return an array of elliptic curves corresponding to this splitting.

HyperellipticCurve(H): HypGeomData -> CrvHyp

HyperellipticCurve(H, t): HypGeomData, RngQZElt -> CrvHyp

Given hypergeometric data H of degree 4 and weight 1 and a rational $t \neq 0, 1$, return the associated hyperelliptic curve, if this data is known to correspond to such. When t is not given, return the result over a function field. There are 18 cases where one gets a genus 2 curve from the CanonicalCurve (making 36 cases when twisting is considered), and a few others where CanonicalCurve gives a higher genus curve and there is a genus 2 quotient. In general, one can try to call IsHyperelliptic on the CanonicalCurve.

`Identify(H, t): HypGeomData, RngQZElt -> Any`

Given hypergeometric data H and a rational $t \neq 0, 1$, return any known associated object (else returns `false`). The return value can (currently) be: an Artin representation (weight 0); an elliptic curve over $\mathbf{Q}$ (weight 1 in degree 2); an elliptic curve over a number field (weight 1 in degree $2r$ with imprimitivity r), or possibly multiple such curves; or a hyperelliptic curve over $\mathbf{Q}$ (weight 1 in degree 4).

138.2.4 Associated Schemes and Curves

`CanonicalScheme(H): HypGeomData -> Sch`

`CanonicalScheme(H, t): HypGeomData, RngQZElt -> Sch`

Given hypergeometric data H, this constructs a canonical associated scheme. When the parameter t is given, the specialization is returned, otherwise the result returned will be a scheme over a function field.

The scheme is determined from the `GammaList`, with a variable (X_i or Y_j) for every element in the list. The scheme is the intersection of $\sum_i X_i = \sum_j Y_j = 1$ with

$$\prod_i X_i^{g_i^+} \prod_j Y_j^{g_j^-} = \frac{1}{Mt},$$

where the g_i^+ are the positive elements in the `GammaList` and the g_j^- are the negative ones (one usually moves the latter to the other side of the equation, to make the exponents positive).

`CanonicalCurve(H): HypGeomData -> Crv`

`CanonicalCurve(H, t): HypGeomData, RngQZElt -> Crv`

Given suitable hypergeometric datum H, this tries to construct an associated plane curve. When the parameter t is given, the specialization at t is returned, otherwise the return value will be a plane curve over a function field. The curve is constructed using the `GammaList` (which indicates the Jacobi sums that need to be taken). When this list has four elements, it is always possible to get a curve. When the list has six elements, it is sometimes possible, depending on whether the largest element (in absolute value) is the negation of the sum of two of the other elements. If it is not possible to construct such a curve, the intrinsic returns `false`.

`AssociatedSchemes(H): HypGeomData -> List, SeqEnum, RngIntElt`

`AssociatedSchemes(H, t): HypGeomData, RngElt -> List, SeqEnum, RngIntElt`

Given a hypergeometric datum H, construct the associated schemes in product projective space of minimal dimension. This involves finding maximal zero-sumset splittings of the `GammaList`. The first returned value is a list of schemes, the second is a corresponding list of splittings, and the third is the resulting dimension. In the second version, the parameter t is nominally a rational not 0 nor 1, though no checks are made. Similarly, the vararg in the first version allows one to define the parameter over a different function field than the rationals, if desired.

138.2.5 Utility Functions

`HypergeometricMotiveSaveLimit(n): RngIntElt`

`HypergeometricMotiveClearTable()`

These are utility intrinsics that will cache the pre-computation of p-adic Γ-functions. The first indicates to save all computed values when the prime power is less than n, and the second clears the table. The qth table entry will have $(q-1)$ elements in it.

`pPart(H, p): HypGeomMot, RngIntElt -> Tup`

`pParts(H): HypGeomMot -> List`

Given a hypergeometric datum, reduce the cyclotomic indices modulo either the given prime or all wild primes.

138.3 Examples

Example: Recognising Hypergeometric Motives (ex-15319e)

Our first example constructs some hypergeometric motives, and recognises them as being related to elliptic curves or Artin representations.

```
> H := HypergeometricData([1/2],[0]); // weight 0
> t := 3/5;
> A := ArtinRepresentation(H,t);
> D := Discriminant(Integers(Field(A))); // -24
> assert IsSquare(D/(t*(t-1))); // Q(sqrt(t(t-1)))
> R := ArtinRepresentationQuadratic(-24);
> assert A eq R;
> //
> H := HypergeometricData([1/4,3/4],[0,0]);
> Weight(H);
1
> DefiningPolynomials(H);
y^2 + 1, y^2 - 2*y + 1
> t := 3/2;
> E := EllipticCurve(H,t); E;
Elliptic Curve defined by y^2 + x*y = x^3 + 1/96*x over Q
> P := PrimesInInterval(5,100);
> &and[EulerFactor(E,p) eq EulerFactor(H,t,p) : p in P];
true
> //
> f := CyclotomicPolynomial(6)*CyclotomicPolynomial(2);
> g := CyclotomicPolynomial(1)^3;
> H := HypergeometricData(f,g); H;
Hypergeometric data given by [ 2, 6 ] and [ 1, 1, 1 ]
> Weight(H);
2
> GammaList(H);
[* -1, -1, -1, -3, 6 *]
> GammaArray(H);
[ -3, 0, -1, 0, 0, 1 ]
> [EulerFactor(H,4,p) : p in [5,7,11,13,17,19]];
[ 125*y^3 + 20*y^2 + 4*y + 1, 343*y^3 - 42*y^2 - 6*y + 1,
  -1331*y^3 - 22*y^2 + 2*y + 1, -2197*y^3 - 156*y^2 + 12*y + 1,
  4913*y^3 + 323*y^2 + 19*y + 1, 6859*y^3 - 57*y^2 - 3*y + 1 ]
> //
> _<u> := FunctionField(Rationals());
> H := HypergeometricData([-2,0,0,-1,0,1] : Print:="alpha_beta");
> H; // weight 1
Hypergeometric data given by [1/6,1/3,2/3,5/6] and [0,0,1/4,3/4]
> HyperellipticCurve(H); // defined over Q(u)
Hyperelliptic Curve defined by y^2 = 4*x^6 - 8*x^5 + 4*x^4 - 64/729/u
> t := 4;
> C := Specialization($1,t); // only works over Q(u)
> &and[EulerFactor(C,p) eq EulerFactor(H,t,p) : p in P];
true
> //
> H := HypergeometricData([0,-1,0,1,0,1,0,-1] : Print:="alpha_beta");
> H; // weight 1
Hypergeometric data given by [1/6,1/3,2/3,5/6] and [1/8,3/8,5/8,7/8]
> MValue(H);
```

(continues on next page)

(continued from previous page)

```
729/4096
> t := 3; // could alternatively specialize later
> E := EllipticCurve(H,t); aInvariants(E);
[ 0, 0, -s, -s, 0] where s^2 is 4096/2187
> &and[EulerFactor(E,p) eq EulerFactor(H,t,p) : p in P];
true
```

Example: Twisting Hypergeometric Data (ex-81c686)

This is a simple example of twisting hypergeometric data, showing that a related Artin motive is obtained for the given weight 0 data.

```
> f := CyclotomicPolynomial(6);
> g := CyclotomicPolynomial(1)*CyclotomicPolynomial(2);
> H := HypergeometricData(f,g); H; assert(Weight(H)) eq 0;
Hypergeometric data given by [ 6 ] and [ 1, 2 ]
> A := ArtinRepresentation(H,-4/5);
> K := OptimisedRepresentation(Field(A));
> DefiningPolynomial(K);
y^6 - 3*y^5 + 3*y^4 - y^3 + 3*y^2 - 3*y + 1
> T := Twist(H); T;
Hypergeometric data given by [ 3 ] and [ 1, 2 ]
> A := ArtinRepresentation(T,-4/5);
> L := OptimisedRepresentation(Field(A));
> IsSubfield(L,K), DefiningPolynomial(L);
true Mapping from: L to K,  y^3 + 3*y - 1
```

The same can be said for twisting for (hyper)elliptic curves.

```
> H := HypergeometricData([2,2],[3]); // Phi_2^2 and Phi_3
> E := EllipticCurve(H,3);
> T := EllipticCurve(Twist(H),3);
> IsQuadraticTwist(E,T);
true -4/9
> //
> H := HypergeometricData([5],[8]); // Phi_5 and Phi_8
> C := HyperellipticCurve(H);
> t := 7;
> S := Specialization(C,t);
> T := HyperellipticCurve(Twist(H),t);
> Q := QuadraticTwist(T,5*t); // get right parameter
> assert IsIsomorphic(Q,S);
```

Example: Primitivity of Hypergeometric Data (ex-34ea1e)

This example exercises the primitivity functionality.

```
> H := HypergeometricData([3],[4]); // Phi_3 and Phi_4
> GammaList(H);
[* -1, 2, 3, -4 *]
> H2 := HypergeometricData([* -2, 4, 6, -8 *]);
> IsPrimitive(H2);
false 2
> PrimitiveData(H2) eq H;
true
> H3 := HypergeometricData([* -3, 6, 9, -12 *]);
> IsPrimitive(H3);
```

(continues on next page)

(continued from previous page)

```
false 3
> PrimitiveData(H3) eq H;
true
> aInvariants(EllipticCurve(H));
[ 0, 0, -64/27/u, -64/27/u, 0 ] where u is FunctionField(Q).1
> aInvariants(EllipticCurve(H2));
[ 0, 0, -s, -s, 0 ] where s^2=(-64/27)^2/u
> aInvariants(EllipticCurve(H3));
[ 0, 0, -s, -s, 0 ] where s^3=(-64/27)^3/u
```

Example: Canonical Schemes and Curves (ex-77d133)

Here is an example with the canonical schemes and curves associated to various hypergeometric data.

```
> _<u> := FunctionField(Rationals());
> H := HypergeometricData([* -2, 3, 4, -5 *]); // degree 4
> C := CanonicalScheme(H);
> _<[X]> := Ambient(C); C;
Scheme over Univariate rational function field over Q defined by
X[1] + X[2] - 1, X[3] + X[4] - 1,
X[1]^2*X[2]^5 - 3125/1728/u*X[3]^3*X[4]^4
> Dimension(C), Genus(Curve(C)); // genus 2 curve
1 2
> assert IsHyperelliptic(Curve(C));
> CC := CanonicalCurve(H); _<x,y> := Ambient(CC); CC;
Curve over Univariate rational function field over Q defined by
x^7 - 2*x^6 + x^5 + 3125/1728/u*y^7 - 3125/576/u*y^6 +
 3125/576/u*y^5 - 3125/1728/u*y^4
> b, C2 := IsHyperelliptic(CC); assert b;
> HyperellipticCurve(H); // in the degree 4 catalogue
Hyperelliptic Curve defined over Univariate function field
 over Q by y^2 = 4*x^5 - 3125/432/u*x^3 + 9765625/2985984/u^2
> assert IsIsomorphic(HyperellipticCurve(H),C2);
> // and an example where the curve is reducible
> H := HypergeometricData([* 6,6,-8,-4 *]); // weight 1
> C := CanonicalCurve(H);
> A := AlgorithmicFunctionField(FunctionField(C));
> E<s> := ExactConstantField(A);
> CE := BaseChange(C,E);
> I := IrreducibleComponents(CE); assert #I eq 2;
> _<x,y> := Ambient(I[1]); I[1];
Scheme over E defined by [ where s^2 = 1048576/531441/u ]
 x^6 - 2*x^5 + x^4 - s*y^6 + 3*s*y^5 - 3*s*y^4 + s*y^3
> b, C2 := IsHyperelliptic(Curve(I[1])); assert b;
```

Example: Associated Schemes in Weighted Projective Space (ex-c4df99)

While canonical schemes are defined (in Magma) over affine spaces, they also have a natural placement in a product (weighted) projective space. Moreover, this can typically reduce the resulting dimension, by noting various subsets of the `GammaList` that sum to zero. This was first implemented in Magma by Nasrecki.

```
> G := [* -1,-1,-1,-1,-1, 2,2,2,2, 3,3,3, -6,-6 *]; // size 14
> H := HypergeometricData(G); // [2,2,3] [6,6], degree 4
> AssociatedSchemes(H); // two schemes in this case
[* Scheme over Univariate rational function field over Q by
```

(continues on next page)

(continued from previous page)

```
    y1+y6+y7+y11+y13, y10+y12+y14, y2+y3+y8, y4+y5+y9,
    -y1*y13^6*y14^6*y2*y3*y4*y5
       - $.1*y6^2*y7^2*y11^3*y10^3*y12^3*y8^2*y9^2,
  Scheme over Univariate rational function field over Q by
    y6+y7+y8+y13, y11+y12+y14, y1+y2+y3+y10, y4+y5+y9,
    -y13^6*y14^6*y1*y2*y3*y4*y5
       - $.1*y6^2*y7^2*y8^2*y11^3*y12^3*y10^3*y9^2  *]
[ [ [ -6,-1,2,2,3 ],[ -6,3,3 ],[ -1,-1,2 ],[ -1,-1,2 ] ],
  [ [ -6,2,2,2 ],[ -6,3,3 ],[ -1,-1,-1,3 ],[ -1,-1,2 ] ] ]
5 // the dimension
```

The number of `AssociatedSchemes` is typically not too large, and the partition into zerosum-subsets typically is somewhat nontrivial.

```
> PHD := [HypergeometricData(h) : h in PossibleHypergeometricData(6)];
> SCH := [<a,b,c> where a,b,c:=AssociatedSchemes(h) : h in PHD];
> Multiset([#s[1] : s in SCH]); // number of schemes
{* 1^^661, 2^^279, 3^^159, 4^^75, 5^^32, 6^^25,
   7^^8, 8^^7, 9^^6, 10, 11, 13^^2, 15 *}
> Multiset([#s[2][1] : s in SCH]); // size of splitting
{* 1^^204, 2^^487, 3^^320, 4^^155, 5^^60, 6^^27, 7^^4 *}
> Multiset([s[3] : s in SCH]); // resulting dimension
{* 0^^6, 1^^112, 2^^161, 3^^507, 4^^154, 5^^271, 6^^19, 7^^25, 8, 9 *}
```

Example: Motive Matching a Siegel Modular Form (ex-678dfc)

Here is an example in degree 4 and weight 3. It turns out that the motive from $t = -1$ has complex multiplication, and the L-series appears to be the same as that of a Siegel modular form given by [vGvS93, §8.7]. (This was found by Cohen and Rodriguez-Villegas). This L-series also appears in Example *Tensor Product and a Siegel Modular Form*.

```
> H := HypergeometricData([1/2,1/2,1/2,1/2],[0,0,0,0]);
> L := LSeries(H,-1 : BadPrimes:=[<2,9,1>]); // guessed
> CFENew(L);
-5.916456789157588540587964239 62E-31
> LGetCoefficients(L,100);
[* 1, 0, 0, 0, -4, 0, 0, 0, -6, 0, 0, 0, -84, 0, 0, 0, 36, 0, 0, 0, 0,
   0, 0, 0, 146, 0, 0, 0, 140, 0, 0, 0, 0, 0, 0, 0, 60, 0, 0, 0, -140,
   0, 0, 0, 24, 0, 0, 0, -238, 0, 0, 0, 924, 0, 0, 0, 0, 0, 0, 0, -820,
   0, 0, 0, 336, 0, 0, 0, 0, 0, 0, 0, -396, 0, 0, 0, 0, 0, 0, 0, -693,
   0, 0, 0, -144, 0, 0, 0, -300, 0, 0, 0, 0, 0, 0, 0, -252, 0, 0, 0 *]
> // compare to the Tensor product way of getting this example
> E := EllipticCurve("32a");
> NF := Newforms(ModularForms(DirichletGroup(32).1,3)); // wt 3 w/char
> L1 := LSeries(E); L2 := LSeries(ComplexEmbeddings(NF[1][1])[1][1]);
> TP := TensorProduct(L1, L2, [ <2, 9> ]); // conductor 2^9 (guessed)
> [Round(Real(x)) : x in LGetCoefficients(TP,100)];
[ 1, 0, 0, 0, -4, 0, 0, 0, -6, 0, 0, 0, -84, 0, 0, 0, 36, 0, 0, 0, 0,
  0, 0, 0, 146, 0, 0, 0, 140, 0, 0, 0, 0, 0, 0, 0, 60, 0, 0, 0, -140,
  0, 0, 0, 24, 0, 0, 0, -238, 0, 0, 0, 924, 0, 0, 0, 0, 0, 0, 0, -820,
  0, 0, 0, 336, 0, 0, 0, 0, 0, 0, 0, -396, 0, 0, 0, 0, 0, 0, 0, -693,
  0, 0, 0, -144, 0, 0, 0, -300, 0, 0, 0, 0, 0, 0, 0, -252, 0, 0, 0 ]
```

Example: Hypergeometric Traces for Non-Galois Data (ex-6d9b8f)

We go through an example working with hypergeometric traces when the underlying datum is not Galois. First we check that the relevant intrinsic `HypergeometricTraceK` gives the same answers as `HypergeometricTrace`.

```
> PHD := PossibleHypergeometricData(4);
> t := 2; p := 13;
> Z := Integers();
> for h in PHD do H:=HypergeometricData(h); A:=H`alpha; B:=H`beta;
>    assert HypergeometricTrace(H,t,p) eq Z!HypergeometricTraceK(A,B,t,p);
>    end for;
```

Next we consider the "Klein quartic" datum defined over $\mathbf{Q}(\sqrt{-7})$. For split primes, its trace will be a quadratic surd.

```
> A := [1/14,9/14,11/14];
> B := [1/4,3/4,1];
> p := 29; // a split prime in Q(sqrt(-7))
> e := HypergeometricTraceK(A,B,2,p : Precision:=10);
> PowerRelation(e,4);
x^2 - x + 2
```

In more generality, one can expect elements in cyclotomic fields.

```
> e := HypergeometricTraceK([1/8,7/8],[0,1/4],2,17 : Precision:=10);
> PowerRelation(e,4);
17*x^4 + 8*x^3 + 24*x^2 + 32*x + 16
> assert IsIsomorphic(NumberField($1),CyclotomicField(8));
> //
> e := HypergeometricTraceK([1/8,7/8],[0,1/4],2,97 : Precision:=10);
> PowerRelation(e,4);
97*x^4 + 56*x^3 - 232*x^2 + 224*x + 784
> assert IsIsomorphic(NumberField($1),CyclotomicField(8));
> // and a non-split prime, with q = 1 mod 8
> e := HypergeometricTraceK([1/8,7/8],[0,1/4],2,7^2 : Precision:=20);
> PowerRelation(e,2); // generates Q(sqrt(2))
7*x^2 - 4*x - 4
```

The K-trace property can also be used to compute at t-values which are not rational. For instance, work of Guillera [AG12, Gui12] implies that $t = 8/(15\sqrt{5} - 33)^3$ should give a pole for $(\Phi_2^3\Phi_3, \Phi_1^5)$, and we can verify numerically that the (normalized) traces have average 1 for small primes (and also that the second moment is approximately 2, indicating a splitting).

```
> H := HypergeometricData([2,2,2,3],[1,1,1,1,1]); // weight 4
> A := H`alpha; B := H`beta;
> Z := Integers();
> f := func<p|(1/((15*Sqrt(pAdicField(p,20)!5)-33)/2)^3)>;
> P := [p : p in PrimesUpTo(200) | p^2 mod 5 eq 1];
> R := [<p,Z!HypergeometricTraceK(A,B,Z!f(p),p)> : p in P];
> &+[r[2]/r[1]^2*1.0 : r in R]/#P;
1.13162808398732021466387060353
> &+[r[2]^2/r[1]^4*1.0 : r in R]/#P; // second moment
2.32631764414942310383344594579
```

Comparatively, most t-values have mean zero and second moment of 1, though again by [Gui12] we expect that $t = (4/3)^3$ is exceptional.

```
> MomentData(LSeries(H,5),P,2); // t = 5
0.14376 [ 0.84480 ]
> MomentData(LSeries(H,11),P,2); // t = 11
0.17961 [ 0.68179 ]
> MomentData(LSeries(H,(4/3)^3),P,2); // t = (4/3)^3
0.81211 [ 1.2554 ] // P is not that robust
```

(continues on next page)

(continued from previous page)

```
> MomentData(LSeries(H,(4/3)^3),PrimesUpTo(1000),2);
0.98494 [ 1.9196 ]
```

Finally, we demonstrate how to work with p-adic inputs for the t-parameter.

```
> A := [1/4]; B := [1/3];
> Qp := pAdicField(5,20);
> E := ext<Qp|Polynomial([-2,0,1])>; // x^2-2, unramified
> HypergeometricTraceK(A,B,4+E.1,5);
1190*E.1 + 1258 + O(5^5)
> HypergeometricTraceK(A,B,4-E.1,5);
-1190*E.1 + 1258 + O(5^5)
```

Example: Euler Factors at Bad Primes (ex-bd0b5d)

Here is an example showing how to handle bad primes in some cases. The Euler factors at $\{3, 5, 17\}$ [where $p|(t-1)$] were determined via a recipe from deformation theory by Rodriguez-Villegas, while at $p = 2$, Roberts suggested a t-value that would trivialise the conductor (from a number field analogy), and Tornaría then computed the full degree 4 factor (at $p = 2$) for $t = 2^8$.

```
> H := HypergeometricData([1/2,1/2,1/2,1/2],[0,0,0,0]);
> Lf := LSeries(Newforms(ModularForms(8,4))[1][1]);
> T := PolynomialRing(Integers()).1; // dummy variable
> f3 := EulerFactor(Lf,3 : Integral)*(1-3*T); // make it a poly
> f5 := EulerFactor(Lf,5 : Integral)*(1-5*T); // via Integral
> f17 := EulerFactor(Lf,17 : Integral)*(1-17*T);
> f2 := 1+T+6*T^2+8*T^3+64*T^4; // determined by Tornaria
> BP := [<2,0,f2>,<3,1,f3>,<5,1,f5>,<17,1,f17>];
> L := LSeries(H,256 : BadPrimes:=BP);
> Conductor(L);
255
> assert Abs(CFENew(L)) lt 10^(-28);
```

One need not specify all the bad prime information as in the above example. Here is a variation on it, with $t = 1/2^8$ (note that this actually gives the same L-series, as the data is a self-twist, with the character induced by twisting being trivial for this choice of t). Note that only the information at 2 is given to `LSeries`.

```
> H := HypergeometricData([1/2,1/2,1/2,1/2],[0,0,0,0]);
> MValue(H);
256
> t := 1/2^8; // makes v_2(Mt)=0
> f2 := EulerFactor(H,t,2 : Fake);
> f2;
64*T^4 + 8*T^3 + 6*T^2 + T + 1
> L := LSeries(H,t : BadPrimes:=[<2,0,f2>]);
> Conductor(L);
255
> assert Abs(CFENew(L)) lt 10^(-28);
```

Example: The Quintic Threefold (ex-87ce78)

Here is an example with the quintic 3-fold. The deformation theory at $p = 11$ here is related to the Grössencharacter example over $\mathbf{Q}(\zeta_5)$ given in Example Grossencharacter Central Value and Gamma Periods (ex-38ad25). The action on inertia at $p = 11$ involves ζ_5 when $11|t$, and here it is raised to the 5th power, thus trivialising it. As with previous example, the deformation theory also involves a weight 4 modular form, here of level 25.

```
> f := CyclotomicPolynomial(5); g := CyclotomicPolynomial(1)^4;
> H := HypergeometricData(f,g : Print:="alpha_beta");
> H, Weight(H); // weight 3
Hypergeometric data given by [1/5,2/5,3/5,4/5] and [0,0,0,0]
3
> t := 11^5; // 11 is now good, as is raised to 5th power
> T := PolynomialRing(Rationals()).1;
> f2 := (1-T+8*T^2)*(1+2*T); // could have Magma compute these
> f3221 := (1-76362*T+3221^3*T^2)*(1-3221*T); // wt 4 lev 25
> // degree 4 factor at 11 comes from Grossencharacter
> // in fact, this is the t=0 deformation: sum_i x_i^5 = 0
> K<z5> := CyclotomicField(5);
> p5 := Factorization(5*Integers(K))[1][1]; // ramified
> G := HeckeCharacterGroup(p5^2);
> psi := Grossencharacter(G.0,[[3,0],[1,2]]);
> f11 := EulerFactor(LSeries(psi),11 : Integral); f11;
1771561*x^4 - 118459*x^3 + 3861*x^2 - 89*x + 1
> BP := [<2,1,f2>,<5,4,1>,<11,0,f11>,<3221,1,f3221>];
> L := LSeries(H,t : BadPrimes:=BP);
> Conductor(L); // 2*5^4*3221, 5^4 is somewhat guessed
4026250
> LSetPrecision(L,5);
> LCfRequired(L); // approx with old CheckFunctionalEquation
12775              //
> time CFENew(L); // actually needs much fewer now
1.5259E-5
Time: 4.290
```

Again one need not specify all the bad prime information, as Magma can automatically compute it at multiplicative and tame primes (however, the local conductor at 5 must be specified).

```
> EulerFactor(H,t,11); // tame
1771561*T^4 - 118459*T^3 + 3861*T^2 - 89*T + 1
> EulerFactor(H,t,2); // multiplicative
16*T^3 + 6*T^2 + T + 1
> EulerFactor(H,t,3221); // multiplicative
-107637325775281*T^3 + 33663324863*T^2 - 79583*T + 1
```

One can also choose t so as to trivialise the wild prime 5.

```
> MValue(H); // 5^5;
3125
> t := 11^5/5^5;
> f5 := EulerFactor(H,t,5 : Fake); // v_5(Mt)=0
> f5;
15625*T^4 - 125*T^3 - 45*T^2 - T + 1
> L := LSeries(H,t : BadPrimes:=[<5,0,f5>]);
> Conductor(L); // 2*3*26321, Magma computes Euler factors
157926
> LSetPrecision(L,9); // about 4000 terms
> CFENew(L);
-2.32830644E-10
> t := -11^5/5^5; // another choice with v_5(Mt)=0
> f5 := EulerFactor(H,t,5 : Fake); // v_5(Mt)=0
> f5; // four possible Euler factors, one for each Mt mod 5
15625*T^4 + 1750*T^3 + 230*T^2 + 14*T + 1
> L := LSeries(H,t : BadPrimes:=[<5,0,f5>]);
```

(continues on next page)

(continued from previous page)

```
> Conductor(L); // 2*31*331, Magma computes Euler factors
20522
> LSetPrecision(L,9); // about 1300 terms
> CFENew(L);
4.65661287E-10
```

Example: Tame Primes and Artin Representations (ex-3f5d52)

Here is an example with tame primes. This derives from comments of Rodriguez-Villegas. The idea is to take hypergeometric data that has weight 0 or 1, and compare it to Artin representations or hyperelliptic curves.

```
> T := PolynomialRing(Rationals()).1; // dummy variable
> H := HypergeometricData([3,4,6,12],[1,1,5,5]); // degree 10
> b, HC := IsHyperelliptic(CanonicalCurve(H)); // genus 5
> assert b; Genus(HC);
5
> EulerFactor(Specialization(HC,13^12),13); // 13 becomes good
371293*T^10 - 285610*T^9 + 125229*T^8 - 31096*T^7 + 4810*T^6
 - 540*T^5 + 370*T^4 - 184*T^3 + 57*T^2 - 10*T + 1
> EulerFactor(H,13^12,13); // use hypergeometric methods
371293*T^10 - 285610*T^9 + 125229*T^8 - 31096*T^7 + 4810*T^6
 - 540*T^5 + 370*T^4 - 184*T^3 + 57*T^2 - 10*T + 1
> assert &and[EulerFactor(Specialization(HC,p^12),p)
>             eq EulerFactor(H,p^12,p) : p in [11,13,17,19]];
> assert &and[EulerFactor(Specialization(HC,t0*13^12),13)
>             eq EulerFactor(H,t0*13^12,13) : t0 in [1..12]];
```

One can take a smaller power than the 12th, but then the curve will not become completely good at the prime. However, the hypergeometric calculations will still be possible.

```
> EulerFactor(H,17^4,17);
17*T^2 + 2*T + 1
> EulerFactor(H,19^9,19); // takes the Phi_3 term
19*T^2 + 7*T + 1
> EulerFactor(H,19^6,19);
361*T^4 + 114*T^3 + 31*T^2 + 6*T + 1
> EulerFactor(H,1/11^5,11); // degree is phi(1)+phi(5)
-T^5 + 1
> EulerFactor(H,4/11^5,11); // degree is phi(1)+phi(5)
-T^5 + 5*T^4 - 10*T^3 + 10*T^2 - 5*T + 1
```

A similar exploration is possible with a weight 0 example. Here the Artin representation machinery is better able to cope with partially good primes.

```
> H := HypergeometricData([2,3,6],[1,5]); // degree 5
> Q := Rationals();
> EulerFactor(ArtinRepresentation(H,7^6),7 : R:=Q);
-T^5 - T^4 - T^3 + T^2 + T + 1
> EulerFactor(ArtinRepresentation(H,7^3),7 : R:=Q);
T^2 + T + 1
> EulerFactor(ArtinRepresentation(H,7^2),7 : R:=Q);
-T + 1
> EulerFactor(ArtinRepresentation(H,2/11^5),11 : R:=Q);
-T^5 + 5*T^4 - 10*T^3 + 10*T^2 - 5*T + 1
> EulerFactor(H,7^6,7); // compute it directly from H
-T^5 - T^4 - T^3 + T^2 + T + 1
```

(continues on next page)

(continued from previous page)

```
> EulerFactor(H,7^3,7);
T^2 + T + 1
> EulerFactor(H,7^2,7);
-T + 1
> EulerFactor(H,2/11^5,11);
-T^5 + 5*T^4 - 10*T^3 + 10*T^2 - 5*T + 1
```

Example: Tame Euler Factors and Grossencharacters (ex-00ce74)

In general one should be able to relate the tame Euler factors to Grössencharacters. This is partially considered (in a different guise) in [Wei74] and [Wei76]. In particular, for primes p that are 3 mod 4, we can take the hypergeometric data given by the squares modulo p, and find that at a prime l that is 1 mod p with a suitably normalised t-value whose valuation $v_l(t)$ at l is p, the Euler factor is a power (and Tate twist) of that for the canonical Grössencharacter of $\mathbf{Q}(\sqrt{-p})$.

```
> p := 11; assert p mod 4 eq 3 and p ne 3 and IsPrime(p);
> SQ := [-Integers()!x : x in GF(p) | IsSquare(x) and x ne 0];
> H := HypergeometricData([* x : x in [-&+SQ] cat SQ *]);
> GammaList(H);
[* -1, -3, -4, -5, -9, 22 *]
> Weight(H); assert Weight(H) eq (p-5)/2;
3
> K := QuadraticField(-p);
> I := Factorization(p*Integers(K))[1][1];
> G := HeckeCharacterGroup(I); // get Tate twist of canonical GR
> u := (Weight(H)+ClassNumber(K)) div 2;
> v := (Weight(H)-ClassNumber(K)) div 2; //u+v=wt, u-v=classno
> u,v;
2 1
> GR := Grossencharacter(G.0,[[u,v]]); // [6,3] for sqrt(-23)
> for l in [l : l in PrimesUpTo(1000) | l mod p eq 1] do
>   ef := EulerFactor(H,l^p/MValue(H),l);
>   F := Factorization(ef);
>   assert #F eq 1 and F[1][2] eq (p-1)/2;
>   assert F[1][1] eq EulerFactor(GR,l : Integral);
>   printf "%o %o\n",l,F[1][1];
>   end for;
23 12167*x^2 + 207*x + 1
67 300763*x^2 - 871*x + 1
89 704969*x^2 + 801*x + 1
199 7880599*x^2 + 3980*x + 1
```

Example: Enumerating Hypergeometric Data (ex-1d8d1b)

Here is an example of the use of `PossibleHypergeometricData`, enumerating the number of possibilities in small degree. A speed test is also done for the save-limit code.

```
> for d in [1..8] do
>    [#PossibleHypergeometricData(d : Weight:=w) : w in [0..d-1]];
>    end for;
[ 1 ]
[ 3, 10 ]
[ 3, 0, 10 ]
[ 11, 74, 30, 47 ]
[ 7, 0, 93, 0, 47 ]
[ 23, 287, 234, 487, 84, 142 ]
[ 21, 0, 426, 0, 414, 0, 142 ]
```

(continues on next page)

(continued from previous page)

```
[ 51, 1001, 1234, 3247, 894, 1450, 204, 363 ]
> D4w1 := PossibleHypergeometricData(4 : Weight:=1);
> D := [HypergeometricData(x) : x in D4w1]; // 12 are self-twists
> #[x : x in D | Twist(x) eq x or Twist(x) eq SwapAlphaBeta(x)];
12
> #PossibleHypergeometricData(4 : Weight:=1,TwistMinimal);
43
> #PossibleHypergeometricData(4 : Weight:=1,Primitive);
64
> // speed test for SaveLimit
> H := HypergeometricData([1/2,1/2,1/2,1/2],[0,0,0,0]);
> HypergeometricMotiveSaveLimit(2000);
> time _:=LGetCoefficients(LSeries(H,-1),2000);
Time: 1.040
> time _:=LGetCoefficients(LSeries(H,-1),2000);
Time: 0.540
> HypergeometricMotiveClearTable();
> time _:=LGetCoefficients(LSeries(H,-1),2000);
Time: 1.030
```

138.3.1 Special Hypergeometric Motives

As indicated partially in the previous description, one can specialize the (singular) parameter $t = 1$ in a family of a hypergeometric motives, and still get a arithmetic object. In doing this, the degree drops by 1, or 2 if the original datum has even degree and odd weight. However, the weight stays the same. All primes that are not involved with the cyclotomic data can be considered "multiplicative" in that $p|(t-1)$. Magma contains a "database" of bad Euler information for all HGMs up through degree 6. Some examples are given below.

Example: Special Motives of Degree 2 (ex-77313e)

Note that any degree 2 weight 1 hypergeometric datum will have a $t = 1$ specialization with a degree 0 L-function, that is, the trivial L-function that is identically 1. One of the degree 2 weight 0 data gives the Riemann ζ-function, and the other two give Artin representations (which could be written as Dirichlet characters).

```
> H := HypergeometricData([6],[1,2]);
> L := LSeries(H,1); L;
L-series of Artin rep C2: (1,-1) of ext<Q|x^2-3>, conductor 12
> H := HypergeometricData([3],[1,2]); // twist of above
> L := LSeries(H,1); L;
L-series of Riemann zeta function
> H := HypergeometricData([4],[1,2]);
> L := LSeries(H,1); L;
L-series of Artin rep C2: (1,-1) of ext<Q|x^2-2>, conductor 8
```

Magma automatically identifies the examples where the resulting degree is 2 or less, and also some additional Artin representations. However, for (Φ_5, Φ_6^2) and its twist, the resulting newform has level 5400, and as the identification in this space might be expensive, it is not made.

```
> H := HypergeometricData([12],[1,2,3]); // degree 4
> L := LSeries(H,1); L; // drops to degree 3
L-series of Artin rep S4: (3,-1,1,0,-1) of ext<Q|x^4-4*x-6>, cond 6912
> // both the next two drop to degree 2 (odd wt)
> H := HypergeometricData([2,2,6],[10]);
> L := LSeries(H,1); L; CremonaReference(L`parent);
L-series of Elliptic Curve y^2 + y = x^3 - 75*x + 256 over Q
```

(continues on next page)

(continued from previous page)

```
225e1
> H := HypergeometricData([2,2,3],[4,4]);
> L := LSeries(H,1); L; Conductor(L);
L-series of q - 2*q^5 + 12*q^7 - 60*q^11 + O(q^12)
288
```

In general, the returned object is just an L-series label, with the correct bad Euler data attached. There are also various examples (necessarily in even weight) where a translate of the Riemann ζ-function is a factor of L-series.

```
> H := HypergeometricData([3,8],[1,2,2,2,6]);
> Degree(H), Weight(H);
6 2
> L:=LSeries(H,1);
> Degree(L); // drops by 1
5
> L;
(L-function for param t=1 of HG data given by [3,8] and [1,2,2,2,6] /
 Translation by 1 of L-series of Riemann zeta function) *
(Translation by 1 of L-series of Riemann zeta function)
> BadPrimeData(L); // conductor 2^7 * 3^2
[
    <2, 7, 8*x^3 - 4*x^2 - 2*x + 1>,
    <3, 2, -27*x^3 - 3*x^2 + x + 1>
]
> CFENew(L); // check the bad Euler info
0.00000000000000000000000000000
```

In general, the local conductors bad Euler factors (possibly trivial) were found by a process of trial and error, aided by ideas from David Roberts and Fernando Rodriguez-Villegas.

```
> H := HypergeometricData([5,6],[1,1,2,2,3]); // deg 6 wt 1
> L := LSeries(H,1); // degree 4
> BadPrimeData(L);
[
    <2, 1, -2*x^3 + x + 1>, // (1-x)*(1+2x+2x^2)
    <3, 1, -3*x^3 + 2*x^2 + 1>, // (1-x)*(1+x+3x^2)
    <5, 4, 1>
]
> LSetPrecision(L,15);
> CFENew(L);
0.000000000000000
> //
> H := HypergeometricData([14],[1,1,1,2,3]); // deg 6 wt 2
> L := LSeries(H,1); // degree 5
> BadPrimeData(L); // cond too large to check here
[
    <2, 10, 1>,
    <3, 3, 3*x^2 - 4*x + 1>, // (1-x)(1-3x), mixed weight
    <7, 5, 1>
]
```

There are some exotic examples where the specialized L-function factors into degree 2 L-functions of modular forms (or elliptic curves), or at least has an identifiable factor of smaller degree.

```
> H := HypergeometricData([2,2,4,6],[3,10]);
> L := LSeries(H,1); // reduce to deg 4 and wt 1
```

(continues on next page)

(continued from previous page)

```
> E1 := EllipticCurve("400c");
> E2 := EllipticCurve("1200l");
> X := LSeries(E1)*LSeries(E2);
> &and[EulerFactor(L,p) eq EulerFactor(X,p) : p in PrimesUpTo(100)];
true
> //
> H := HypergeometricData([2,2,2,2,2,2],[1,1,1,1,1,1]);
> L := LSeries(H,1); // reduce to deg 4 and wt 5
> f4 := Newforms(CuspForms(8,4))[1][1]; // modular forms
> f6 := Newforms(CuspForms(8,6))[1][1]; // of level 8
> Y := Translate(LSeries(f4),1)*LSeries(f6);
> &and[EulerFactor(L,p) eq EulerFactor(Y,p) : p in PrimesUpTo(100)];
true
> //
> H := HypergeometricData([3,3,3,3],[6,6,6,6]);
> L := LSeries(H,1); // deg 6 weight 7
> P := PrimesInInterval(5,100);
> &+[1.0*Coefficient(EulerFactor(L,p : Degree:=1),1)^2/p^7 : p in P]/#P;
1.997443822975737201500504017 54 // imprimitive, 2 factors
> f6 := Newforms(CuspForms(36,6))[2][1]; // level 36 modwt 6
> LQ := Translate(LSeries(f6),1); // make it weight 7
> P30 := PrimesInInterval(5,30);
> &and[IsDivisibleBy(EulerFactor(L,p),EulerFactor(LQ,p)) : p in P30];
true
```

138.4 Jacobi Motives

A topic related to hypergeometric motives is that of Jacobi sum motives. These are indeed simpler, and in fact the tame prime information for hypergeometric motives can be determined from Jacobi motives, possibly twisted by Kummer and Tate characters.

The classical Jacobi sums were indicated by Weil to come from Grössencharacters [Wei52], and this functionality is also included, with it indeed being the preferred method to compute Euler factors and the L-series, once the reciprocity correspondence has been established and the Grössencharacter identified.

138.4.1 Background

Let $n_j \in \mathbf{Z}$ and $x_j \in \mathbf{Q}/\mathbf{Z}$ with $\theta = \sum_j n_j \langle x_j \rangle$ an element of the free group on $\mathbf{Q}/\mathbf{Z}$ with $\sum n_j x_j \in \mathbf{Z}$.

Letting m be the least common multiple of the denominators of the x_j, the field of definition K_θ is a subfield of $\mathbf{Q}(\zeta_m)$, corresponding by class field theory to quotienting out by $(\mathbf{Z}/m\mathbf{Z})^\star$ by elements which leave θ fixed when scaling by them. When scaling by -1 fixes θ this field K_θ is totally real, and otherwise it is a CM field.

For primes p with $\gcd(p, m) = 1$, we consider Gauss sums corresponding for prime ideals $\mathfrak{p}$ in $\mathbf{Q}(\zeta_m)$, defined by

$$G^{\psi}_{a/m}(\mathfrak{p}) = -\sum_{x \in \mathbf{F}_\mathfrak{p}^\star} \left(\frac{x}{\mathfrak{p}}\right)_m^a \psi\left(\mathrm{Tr}^{\mathbf{F}_\mathfrak{p}}_{\mathbf{F}_p} x\right),$$

where ψ is a nontrivial additive character on $\mathbf{F}_p$ and the power residue symbol takes values in the roots of unity of $\mathbf{Q}(\zeta_m)$ with

$$\left(\frac{x}{\mathfrak{p}}\right)_m^a \equiv x^{(q-1)a/m} \pmod{\mathfrak{p}}.$$

The associated Jacobi sum evaluation for θ at $\mathfrak{p}$ is then given by $\prod_j G_{x_j}(\mathfrak{p})^{n_j}$ with the result being independent of the choice of additive character ψ. This defines the Jacobi sum for good primes $\mathfrak{p}$ up to a choice of ζ_m into $\mathbf{C}$.

If one is just interested in Euler factors over $\mathbf{Q}$ and not K_θ, then a p-adic method using the Gross-Koblitz formula can also be used. There are known bounds on the conductor of the resulting L-function, the first being that of Weil [Wei52].

138.4.2 Kummer and Tate Twists

A Jacobi motive can also be Kummer twisted by t^ρ for some rational ρ and nonzero rational t. This can increase the field of definition so that m includes the denominator of ρ. This corresponds to multiplying the various Jacobi sum evaluations by suitable roots of unity.

Often one wants to Tate twist the Jacobi sum to gets its effective weight, and for this reason the full unit is sometimes called a Jacket motive (for Jacobi, Kummer, and Tate).

138.5 Jacobi Motive Functionality

138.5.1 Creation Functions

`JacobiMotive(A, B): SeqEnum, SeqEnum -> JacketMot`

```
Kummer: SeqEnum                    Default: [1,0]
Tate  : RngIntElt                  Default: 0
Weight: RngIntElt                  Default:
```

Given two sequences of rationals, corresponding to positive and negative elements in the free group on $\mathbf{Q}/\mathbf{Z}$, create the resulting Jacobi motive. This requires that the signed sum of the rationals is an integer. The optional `Kummer` vararg can specify a Kummer twist, and similarly with the `Tate` vararg. Alternatively, the desired Tate twist can be obtained by giving and integral argument to the `Weight` vararg, and the effective Tate twist can be obtained by setting `Weight` as `true`. A variant with only one argument (B is empty) is also available.

`JacketMotive(A, B, t, rho, j): SeqEnum, SeqEnum, RngQZElt, RngQZElt, RngIntElt -> JacketMot`

Similar to above, this intrinsic spells out the Jacobi summands (A, B), the Kummer twist t^ρ, and the Tate twist j explicitly.

`KummerTwist(J, t, rho): JacketMot, RngQZElt, RngQZElt -> JacketMot`

Given a Jacobi motive J, return its Kummer twist by t^ρ, where ρ is rational and t is a nonzero rational.

`TateTwist(J, j): JacketMot, RngIntElt -> JacketMot`

Given a Jacobi motive J and an integer j, return the jth Tate twist of J.

138.5.2 Operations

`J1 * J2: JacketMot, JacketMot -> JacketMot`

Given two Jacobi motives, take their tensor product, eliminating any rationals common to the positive and negative parts.

`J1 / J2: JacketMot, JacketMot -> JacketMot`

Given two Jacobi motives, take their tensor quotient, eliminating any rationals common to the positive and negative parts.

`J1 eq J2: JacketMot, JacketMot -> JacketMot`

`J1 ne J2: JacketMot, JacketMot -> JacketMot`

Whether two Jacobi motives are equal, that is, whether they have the same positive and negative parts, their t^ρ Kummer twists are the same, and they have the same Tate twist parameter.

`Scale(J, q): JacketMot, RngQZElt -> JacketMot`

Given a Jacobi motive, scale all the rational numbers defined the datum by the given rational q. The denominator of q must be coprime to m, and q must be invertible mod m. The resulting motive will be identical over $\mathbf{Q}$ but need only be conjugate over K_θ.

138.5.3 Attributes

`Field(J): JacketMot -> FldNum`

The field of definition of a Jacobi motive.

`Weight(J): JacketMot -> RngIntElt`

The motivic weight of a Jacobi motive.

`EffectiveWeight(J): JacketMot -> RngIntElt`

The effective motivic weight of a Jacobi motive, that is, the width of its Hodge structure.

`HodgeStructure(J): JacketMot -> HodgeStruc`

`HodgeVector(J): JacketMot -> HodgeStruc, RngIntElt`

`EffectiveHodgeStructure(J): JacketMot -> HodgeStruc`

The Hodge structure of a Jacobi motive.

138.5.4 L-function

`EulerFactor(J, p): JacketMot, RngIntElt -> RngUPolElt`

```
Degree: RngIntElt                    Default:
Roots : BoolElt                      Default: false
```

Given a good prime p, that is, one which is coprime to m and the Kummer twisting parameter t, compute its Euler factor. The `Roots` vararg also returns as a second argument the p-adic approximations to the roots (associated to the prime ideals above p). The `Degree` vararg can be used when the full Euler factor is not needed, though it is often still just as easily computed. It is often easier to first identify the Jacobi motive as a `Grossencharacter`, and then compute its Euler factors.

`ComplexEvaluation(J, P): JacketMot, RngOrdIdl -> FldComElt`

```
Precision: RngIntElt                 Default:
```

Given a Jacobi motive and a good degree 1 prime over K_θ that splits completely over the cyclotomic field, compute the associated Jacobi sum as a complex number. This is used to identify motives that are equivalent over $\mathbf{Q}$ but not over K_θ in some examples.

`Grossencharacter(J): JacketMot -> GrossenChar`

Given a Jacobi motive, identify it as a Grössencharacter. This uses the Weil bound on the conductor, and then tries enough good primes to distinguish the character. This is now the preferred way to compute the `LSeries` of a Jacobi motive (though the latter still exists).

138.6 Jacobi Motive Examples

Example: Jacobi Motives of the Fermat Cubic `(ex-1aeb77)`

Here are some simple examples of Jacobi motives. The first involves the Fermat cubic, and various twists.

```
> J := JacobiMotive([2/3,2/3],[1/3]); // CM elliptic curve conductor 27
> Weight(J);
1
> Field(J);
Number Field with defining polynomial y^2 - y + 1 over Q
> P := PrimesInInterval(11,100);
> E := EllipticCurve("27a");
> &and[EulerFactor(J,p) eq EulerFactor(E,p) : p in P];
true
> K := KummerTwist(J, 2, 1/3); // twist by 2^(1/3)
> Et := EllipticCurve("108a");
> &and[EulerFactor(K,p) eq EulerFactor(Et,p) : p in P];
true
> K4 := KummerTwist(J, 4, 1/3); // twist by 4^(1/3)
> E36 := EllipticCurve("36a");
> &and[EulerFactor(K4,p) eq EulerFactor(E36,p) : p in P];
true
> K2 := KummerTwist(J, -2, 1/2); // quadratic twist by -2
> Q := QuadraticTwist(E,-2);
> &and[EulerFactor(K2,p) eq EulerFactor(Q,p) : p in P];
true
```

Example: Jacobi Motives and the Klein Quartic (ex-475839)

The next example is related to the Klein quartic and the elliptic curves of conductor 49.

```
> J := JacobiMotive([1/7,2/7,4/7]);
> Scale(J,2) eq J; // scaling by 2 or 4 gives same motive
true
> Field(J);
Number Field with defining polynomial y^2 - y + 2 over the Rational Field
> Weight(J);
3
> EffectiveWeight(J);
1
> T := TateTwist(J,1); // this twist is weight 1
> Weight(T);
1
> P := PrimesInInterval(11,100);
> E := EllipticCurve("49a");
> &and[EulerFactor(T,p) eq EulerFactor(E,p) : p in P];
true
> Grossencharacter(J);
Grossencharacter of type [[ 1, 2 ]] for Hecke-Dirichlet pair (1,$.1)
with modulus of norm 7 over Number Field y^2 - y + 2
> TateTwist($1,1);
Grossencharacter of type [[ 0, 1 ]] for Hecke-Dirichlet pair (1,$.1)
with modulus of norm 7 over Number Field y^2 - y + 2
```

Example: Tensor Arithmetic of Jacobi Motives (ex-c0a387)

We next give an example with some tensor arithmetic.

```
> J := JacobiMotive([1/3,1/3,1/3]); // weight 3, effective wt 1
> J;
Jacobi motive given by 3*[1/3]
> Grossencharacter(J);
Grossencharacter of type [[ 1, 2 ]] for Hecke-Dirichlet pair
```

(continues on next page)

(continued from previous page)

```
(1,$.1*$.2^2) with modulus of norm 9 over Number Field y^2 - y + 1
> J^3;
Jacobi motive given by 9*[1/3]
> Grossencharacter(J^3); // norm 3
Grossencharacter of type [[ 3, 6 ]] for Hecke-Dirichlet pair
(1,$.1) with modulus of norm 3 over Number Field y^2 - y + 1
> Grossencharacter(J^6); // norm 1
Grossencharacter of type [[ 6, 12 ]] for Hecke-Dirichlet pair
(1,1) with modulus of norm 1 over Number Field y^2 - y + 1
> A := JacobiMotive([2/3,2/3],[1/3]);
> J*A; // one of the 1/3 cancels
Jacobi motive given by 2*[1/3]+2*[2/3]
> Grossencharacter(J*A); // trivial character
Tate twist by -2 of Hecke character 1 in Group of Hecke characters
of modulus of norm 1 over Number Field y - 1
> K := KummerTwist(J*A, 3, 1/3); K;
Jacobi motive given by 2*[1/3]+2*[2/3] with Kummer twisting
parameters (t,rho)=(3,1/3)
> Grossencharacter(K); // bad reduction at 3
Grossencharacter of type [[ 2, 2 ]] for Hecke-Dirichlet pair (1,1)
with modulus of norm 81 over Number Field y^2 - y + 1
> K2 := KummerTwist(JacobiMotive([]),3,1/3); K2;
Unital Jacobi motive with Kummer twisting parameters (t,rho)=(3,1/3)
> Grossencharacter(K2); // twist of the above
Grossencharacter of type [[ 0, 0 ]] for Hecke-Dirichlet pair
(1,$.1^2) with modulus of norm 81 over Number Field y^2 - y + 1
> K/K2 eq J*A; // removing the Kummer twist from K, gives J*A
true
```

Example: Motives Agreeing Only over Q (ex-46e756)

Here is an example with motives that are the same over $\mathbf{Q}$, but not over $\mathbf{Q}(\sqrt{-3})$. One of the Grössencharacters is seen to agree with the Jacobi motive at prime ideals.

```
> J := JacketMotive([], [], 2, 1/3, 0); J; // twist by 2^(1/3)
Unital Jacobi motive with Kummer twisting parameters (t,rho)=(2,1/3)
> GR := Grossencharacter(J); GR;
Grossencharacter of type [[ 0, 0 ]] for Hecke-Dirichlet pair (1,$.1)
with modulus of norm 36 over Number Field y^2 - y + 1
> ef := func<G,p | EulerFactor(G,p : Integral)>;
> P := PrimesUpTo(100);
> &and[ef(GR,p) eq ef(GR2,p) where GR2:=GR^2 : p in P]; // same over Q
true
> p7 := Ideal(Decomposition(Field(J),7)[1][1]); // norm 7, degree 1
> GR(p7);
-0.49999999999999999999999999998 - 0.86602540378443864676372317075 4*i
> (GR^2)(p7);
-0.50000000000000000000000000001 + 0.86602540378443864676372317075 2*i
> ComplexEvaluation(J,p7); // same as GR, not GR^2
-0.49999999999999999999999999998 - 0.86602540378443864676372317075 4*i
> PK := [p : p in PrimesUpTo(100,Field(J)) | Norm(p) ne 3];
> Max([Norm(GR(p)-ComplexEvaluation(J,p)) : p in PK | Degree(p) eq 1]);
1.0112199337457536249980455606 2E-59
```

Similarly, as John Voight indicates, one does not always have a unique Grossencharacter that is identifiable from degree 1 primes.

```
> J := JacobiMotive([3/4,1/8,5/8],[1/2]); // scaling by 5
> psi := Grossencharacter(J); psi;
WARNING: Didn't get unique kernel (size 2)
Grossencharacter of type [[ 1, 1 ]] with norm 64 over x^2 + 1
> G := HeckeCharacterGroup(64*Integers(J`K));
> TG := TargetRestriction(G,CyclotomicField(1));
> P := [p : p in PrimesUpTo(1000) | p mod 8 eq 5];
> A := [EulerFactor(psi,p : Integral) : p in P];
> tw := psi*TG.2;
> B := [EulerFactor(tw,p : Integral) : p in P];
> assert A eq B; // same L-function over Q
> // but they are different over primes that are 5 mod 8
> (TG.2)(Factorization(5*Integers(J`K))[1][1]);
-1
```

The point here is that only primes that are 5 mod 8 can distinguish this Grossencharacter, and Magma can not use these in `ComplexEvaluation`.

Example: Tame Primes via Jacobi Motives (ex-b87008)

Jacobi motives also can be used to determine the tame prime information for a hypergeometric motive. It is easiest to write the hypergeometric datum in its `GammaList` form when doing this. The appropriate Kummer-Tate twist of the Jacobi motive gives the tame hypergeometric Euler factor corresponding to trivialised inertia near $t = 0$ or $t = \infty$.

```
> H := HypergeometricData([* 1,2,3,3,6, -5,-5,-5 *]); H;
Hypergeometric data given by [ 1, 1, 2, 2, 3, 3, 3, 6 ] and [ 5, 5, 5 ]
> Weight(H);
3
> POS := [x/5 : x in GammaList(H) | x ge 0];
> NEG := [x/5 : x in GammaList(H) | x le 0];
> J:=JacobiMotive(POS,NEG); J; // weight 5
Jacobi motive given by 2*[1/5]+[2/5]+2*[3/5]
> p := 11; // check 5th powers for beta value
> EulerFactor(H,1/p^5/MValue(H),p); // weight 3-2
121*x^4 + 11*x^3 - 9*x^2 + x + 1
> EulerFactor(TateTwist(J,2),p); // weight 5-2*2
121*x^4 + 11*x^3 - 9*x^2 + x + 1
> [EulerFactor(H,u/p^5/MValue(H),p) : u in [1..p-1]];
[
    121*x^4 + 11*x^3 - 9*x^2 + x + 1,
    121*x^4 + 121*x^3 + 51*x^2 + 11*x + 1,
    121*x^4 - 44*x^3 + 6*x^2 - 4*x + 1,
    121*x^4 - 99*x^3 + 41*x^2 - 9*x + 1,
    121*x^4 + 11*x^3 + 21*x^2 + x + 1,
    121*x^4 + 11*x^3 + 21*x^2 + x + 1,
    121*x^4 - 99*x^3 + 41*x^2 - 9*x + 1,
    121*x^4 - 44*x^3 + 6*x^2 - 4*x + 1,
    121*x^4 + 121*x^3 + 51*x^2 + 11*x + 1,
    121*x^4 + 11*x^3 - 9*x^2 + x + 1
]
> [EulerFactor(JacketMotive(POS,NEG,u,1/5,2),p) : u in [1..p-1]];
[
    121*x^4 + 11*x^3 - 9*x^2 + x + 1,
    121*x^4 + 121*x^3 + 51*x^2 + 11*x + 1,
    121*x^4 - 44*x^3 + 6*x^2 - 4*x + 1,
    121*x^4 - 99*x^3 + 41*x^2 - 9*x + 1,
    121*x^4 + 11*x^3 + 21*x^2 + x + 1,
```

(continues on next page)

(continued from previous page)

```
    121*x^4 + 11*x^3 + 21*x^2 + x + 1,
    121*x^4 - 99*x^3 + 41*x^2 - 9*x + 1,
    121*x^4 - 44*x^3 + 6*x^2 - 4*x + 1,
    121*x^4 + 121*x^3 + 51*x^2 + 11*x + 1,
    121*x^4 + 11*x^3 - 9*x^2 + x + 1
]
> p := 31;
> &and[EulerFactor(H,u/p^5/MValue(H),p) eq
>      EulerFactor(JacketMotive(POS,NEG,u,1/5,2),p) : u in [1..p-1]];
true
```

The same can be done for the α parameters, with positive valuation in the t-values. Here there can be various contributions depending which of the positive γ-values (1,2,3,3,6) divide the valuation.

```
> p := 7; // again take a prime that is 1 mod 6
> d := 6;
> D := Divisors(d);
> POS := [[x/e : x in GammaList(H) | x ge 0] : e in D];
> NEG := [[-x/e : x in GammaList(H) | x le 0] : e in D];
> A, B := CyclotomicData(H);
> for e in D do e,Multiplicity(A cat B,e); end for;
1 2 // weight should be 3 + 1 - 2
2 2 // weight should be 3 + 1 - 2
3 3 // weight should be 3 + 1 - 3
6 1 // weight should be 3 + 1 - 1
> EulerFactor(H,p^2/MValue(H),p);
49*x^2 - 14*x + 1
> f1 := EulerFactor(JacobiMotive(POS[1],NEG[1] : Weight:=2),p);
> f2 := EulerFactor(JacobiMotive(POS[2],NEG[2] : Weight:=2),p);
> f1 * f2;
49*x^2 - 14*x + 1
> function ef(t0,v) ans:=PolynomialRing(Integers())!1;
>   for i in [1..#D] do
>     e := D[i]; if Gcd(v,e) ne e then continue; end if;
>     w := Weight(H)+1-Multiplicity(A cat B,e);
>     J := JacobiMotive(POS[i],NEG[i] : Kummer:=[t0,1/e], Weight:=w);
>     ans := ans*EulerFactor(J,p); end for;
>   return ans; end function;
> for i in [1..100] do v:=Random([1..12]); t0:=Random([1..p-1]);
>   assert EulerFactor(H,t0*p^v/MValue(H),p) eq ef(t0,v); end for;
```

138.7 Bibliography

- [AG12] Gert Almkvist and Jesús Guillera. Ramanujan-like series for $1/\pi^2$ and String Theory. *Experimental Math.*, 21:223–234, 2012. URL: http://dx.doi.org/10.1080/10586458.2012.656059.
- [Gui12] 2012.
- [Kat90] N. M. Katz. *Exponential Sums and Differential Equations.* Volume 124. Annals of Math. Studies., 1990.
- [Kat96] N. M. Katz. *Rigid Local Systems.* Volume 139. Annals of Math. Studies., 1996.
- [vGvS93] B. van Geemen and D. van Straten. The cusp forms of weight 3 on $\Gamma_2(2,4,8)$. *Math. Comp.*, 61(204):849–872, 1993.
- [Wei52] A. Weil. Jacobi Sums as "Grössencharaktere". *Trans. AMS*, 73:487–495, 1952.
- [Wei74] A. Weil. Sommes de Jacobi et caractères de Hecke. *Göttingen Nachr.*, 1:1–14, 1974.

- [Wei76] A. Weil. Sur les périodes des intégrales abéliennes. *Comm. Pure Appl. Math.*, XXIX:813–819, 1976.

Chapter 139

L-Functions

139.1 Overview

A large variety of ζ-functions and L-functions occur in number theory and algebraic geometry. Some well-known L-functions include the Riemann ζ-function, the Dedekind ζ-function of a number field, Dirichlet series associated to characters, and L-series of curves (e.g., elliptic curves) over the rationals. Magma provides functionality for constructing such L-functions and computing their values in the complex plane. A typical calculation might go as follows:

```
> L := LSeries(EllipticCurve([0, -1, 1, 0, 0]));
> Evaluate(L,2);
0.546048036215013518334126660433
```

The first line defines an L-series $L(E, s)$ of the elliptic curve

$$E : y^2 + xy = x^3 - x^2$$

while the second line computes its value at $s = 2$. An impatient reader may wish simply to type `LSeries;` at the prompt and look at the various `LSeries` signatures and mimic the code above, thus getting access to much of the functionality.

Topics covered in this chapter include:

- The built-in L-series which include the Riemann ζ-function, the Dedekind ζ-function of a number field, Dirichlet series associated to characters, Artin representations, modular forms, and L-series of elliptic curves;
- The calculation of values, derivatives, and Taylor expansions of L-series at a complex point s_0 to desired accuracy;
- A technical description of the L-series object in Magma, together with a description of how to construct user-defined L-series with any number of gamma factors, provided that the L-series satisfies a functional equation of the standard type;
- Operations such as division, multiplication, and the tensor product of two L-series.

The reader is referred to Manin-Panchishkin [Sha95] Chapter 4, Serre [Ser65], and articles in [JKS94] for a background on L-functions. The algorithms mostly follow Dokchitser [Dok04] and the Pari implementation ComputeL [Dok02]. See also Lavrik [Lav67], Tollis [Tol97], and the exposition in Cohen [Coh00], 10.3.

139.2 Built-in L-series

An *L-series* or an *L-function* is an infinite sum $L(s) = \sum_{n=1}^{\infty} a_n/n^s$ in the complex variable s with complex coefficients a_n. Such functions arise in many places in mathematics and they are usually

naturally associated with some kind of mathematical object, for instance a character, a number field, a curve, a modular form, or a cohomology group of an algebraic variety. The coefficients a_n are certain invariants associated with that object. For example, in the case of a character $\chi : (\mathbb{Z}/m\mathbb{Z})^* \to \mathbb{C}^*$ they are simply its values $a_n = \chi(n)$ when $\gcd(n, m) = 1$ and 0 otherwise.

Magma is able to associate an L-series to various types of object. The intrinsic which provides access to such pre-defined L-series is

`LSeries(`*object: optional parameters*`)`

Every such function returns a variable of type `LSer`. A range of functions may now be applied to this L-series object as described in the following sections (and in fact more are available than listed), and these are independent of the object to which the L-series was originally associated. In fact, an object of type `LSer` only "remembers" its origin for printing purposes.

`RiemannZeta() -> LSer`

Precision: RngIntElt	Default:

The Riemann zeta function $\zeta(s)$ is returned.

The number of digits of precision to which the values $\zeta(s)$ are to be computed may be specified using the `Precision` parameter. If it is omitted, the precision of the default real field will be used.

Example: Riemann Zeta Function at 2 (ex-b6ea59)

Check that $\zeta(2)$ agrees numerically with $\pi^2/6$.

```
> L := RiemannZeta( : Precision:=40);
> Evaluate(L,2);
1.6449340668482264364724151666460251892l9
> Pi(RealField(40))^2/6;
1.6449340668482264364724151666460251892l9
```

`LSeries(K): FldNum -> LSer`

Method : MonStgElt		Default: "Default"
ClassNumberFormula: BoolElt		Default: false
Precision : RngIntElt		Default:

Create the Dedekind zeta function $\zeta(K, s)$ of a number field K. The series is defined by $\sum_I \mathrm{Norm}_{K/\mathbb{Q}}(I)^{-s}$, where the sum is taken over the non-zero ideals I of the maximal order of K. For $K = \mathbb{Q}$, the series coincides with the Riemann zeta function.

The optional parameter `Method` may be `"Artin"`, `"Direct"` or `"Default"` and specifies whether the zeta function should be computed as a product of L-series of Artin representations or directly, by counting prime ideals. (The default behaviour depends upon the field.)

For the `"Direct"` method, the Dedekind zeta function has a simple pole at $s = 1$ whose residue must be known in order to compute the L-values. The class number formula gives an expression for this residue in terms of the number of real/complex embeddings of K, the regulator, the class number, and the number of roots of unity in K. If the optional parameter `ClassNumberFormula` is set to `true`, then these quantities are computed on initialization (using Magma's functions `Signature(K), Regulator(K), #ClassGroup(MaximalOrder(K))` and `#TorsionSubgroup(UnitGroup(K))`) and it might take some time if the discriminant of K is large. If `ClassNumberFormula` is `false` (default) then the residue is computed numerically from the functional equation. This is generally faster, unless the discriminant of K is small and the precision is set to be very high.

The number of digits of precision to which the values $\zeta(K, s)$ are to be computed may be specified using the `Precision` parameter. If it is omitted the precision is taken to be that of the default real field.

Example: Dedekind Zeta Function of Q(i) (ex-a7a8a2)

This code computes the value of $\zeta(\mathbb{Q}(i), s)$ at $s = 2$.

```
> P<x> := PolynomialRing(Integers());
> K := NumberField(x^2+1);
> L := LSeries(K);
> Evaluate(L, 2);
1.5067030099229850308865650481 8
```

Example: Dedekind Zeta of a Degree 12 Field (ex-e875eb)

The code computes $\zeta(F, 2)$ for $F = \mathbb{Q}(\sqrt[12]{3})$.

```
> R<x> := PolynomialRing(Rationals());
> F := NumberField(x^12-3);
> L := LSeries(F: Method:="Direct");
> Conductor(L), LCfRequired(L);
1579460446107205632 92968955438
```

The set-up time for the direct method is negligible, but the L-value computation will take days for this number of coefficients. On the other hand, the normal closure of F is not too large and has only representations of small dimension:

```
> G := GaloisGroup(F);
> #G, [Degree(ch): ch in CharacterTable(G)];
24 [ 1, 1, 1, 1, 2, 2, 2, 2, 2 ]
> time L := LSeries(F : Method:="Artin");
Time: 0.340
```

It took longer to define the L-series, but the advantage is that it is a product of L-series with very small conductors, and the L-value calculations are almost instant:

```
> [Conductor(f[1]) : f in Factorisation(L)];
[ 1, 12, 3888, 243, 576, 15552, 15552 ]
> time Evaluate(L, 2);
1.639254271936468828359907088 20
Time: 0.840
```

Example: Dedekind Zeta Vanishing at the Centre (ex-f2ca95)

This code follows an example of Serre and Armitage (see [Arm71, Fri76, Ser71]) where the ζ-function of a field vanishes at the central point.

```
> _<x> := PolynomialRing(Rationals());
> K<s5> := NumberField( x^2-5 );
> L<s205> := NumberField( x^2-205 );
> C := Compositum(K,L);
> e1 := C!(5+s5);
> e2 := C!(41+s205);
> E := ext<C | Polynomial( [ -e1*e2, 0, 1] )>;
> A := AbsoluteField(E);
> DefiningPolynomial(A);
x^8 - 820*x^6 + 223040*x^4 - 24206400*x^2 + 871430400
> Signature(A); // totally real
8 0
> L := LSeries(A);
> LCfRequired(L); // approx value, with old CFE
2739
```

(continues on next page)

(continued from previous page)

```
> CFENew(L); // new vers of CheckFunctionalEquation
0.00000000000000000000000000000
> Evaluate(L, 1/2); // zero as expected
0.00000000000000000000000000000
```

So the evaluation of L at $1/2$ is zero as expected. In fact, L is a product, and one factor has odd sign:

```
> L`prod;
[
    <L-series of Riemann zeta function, 1>,
    <L-series of Artin representation Q8: (1,1,-1,1,-1) of
    ext<Q|x^8-820*x^6+223040*x^4-24206400*x^2+871430400>, conductor 5, 1>,
    <L-series of Artin representation Q8: (1,1,1,-1,-1) of
    ext<Q|x^8-820*x^6+223040*x^4-24206400*x^2+871430400>, conductor 41, 1>,
    <L-series of Artin representation Q8: (1,1,-1,-1,1) of
    ext<Q|x^8-820*x^6+223040*x^4-24206400*x^2+871430400>, conductor 205, 1>,
    <L-series of Artin representation Q8: (2,-2,0,0,0) of
    ext<Q|x^8-820*x^6+223040*x^4-24206400*x^2+871430400>, conductor 42025, 2>
]
> [ ComplexField(9)!Sign(x[1]) : x in $1 ];
[ 1.00000000, 1.00000000, 1.00000000, 1.00000000, -1.00000000 ]
> Sign(L`prod[5][1]);
-1.00000000000000000000000000000
```

This phenomenon happens fairly generally with symplectic representations, and quaternionic Galois groups give the most direct examples. The example of smallest conductor $2^8 3^2$ appears to be for the field $x^8 + 12x^6 + 36x^4 + 36x^2 + 9$, while $x^8 - x^7 + x^6 - 4x^5 + 5x^4 - 8x^3 + 4x^2 - 8x + 16$ has Galois group $\mathbf{SL}_2(\mathbf{F}_3)$ and conductor 163^2.

`LSeries(A): ArtRep -> LSer`

```
Precision: RngIntElt                    Default:
```

Creates the L-series of an Artin representation A. (For information about Artin representations see Chapter Artin Representations.)

Example: Artin L-Series over Q(i) `(ex-e5651d)`

L-series of the two characters of $\mathrm{Gal}(\mathbb{Q}(i)/\mathbb{Q}) \cong C_2$.

```
> K := QuadraticField(-1);
> triv,sign := Explode(ArtinRepresentations(K));
> Evaluate(LSeries(triv), 2);    // zeta(2)=pi^2/6
1.64493406684822643647241516665
> Evaluate(LSeries(sign), 2);
0.91596559417721901505460351493
```

Example: Artin Representation of A7 `(ex-c074bd)`

We take a polynomial over $\mathbb{Q}$ with Galois group A_7 and the unique 6-dimensional irreducible Artin representation a of this group

```
> load galpols;
> f:=PolynomialWithGaloisGroup(7,6); // Alt(7)
> K:=NumberField(f);
> A:=ArtinRepresentations(K);
> a:=A[2];a;
```

(continues on next page)

(continued from previous page)

```
Artin representation A7: (6,2,3,0,0,1,-1,-1,-1) of ext<Q|x^7-2*x^6-7*x^5+11*x^4+
   16*x^3-14*x^2-11*x+2>
```

Its L-series with the default precision of 30 digits needs a lot of coefficients to compute with.

```
> L:=LSeries(a);
> LCfRequired(L); // approx value
1830809
```

We decrease the precision to 8 digits

```
> L:=LSeries(a: Precision:=8);
> LCfRequired(L); // approx value with CheckFunctionalEquation
52584
```

Now it takes under 15 seconds to verify the functional equation of $L(a, s)$ and to compute its value at $s = 2$.

```
> CFENew(L); // new vers of CheckFunctionalEquation
1.4901161E-8
> Evaluate(L,2);
1.1105972
```

`LSeries(E): CrvEll -> LSer`

```
Precision: RngIntElt                    Default:
```

Create the L-series $L(E, s)$ of an elliptic curve E defined over $\mathbb{Q}$ or over a number field.

The number of digits of precision to which the values $L(E, s)$ are to be computed may be specified using the `Precision` parameter. If it is omitted the precision is taken to be that of the default real field.

Note that the computation time for evaluating an L-series grows roughly like the square root of the conductor (or its norm to $\mathbb{Q}$ if the base field is a number field). Therefore an evaluation might take an unreasonable amount of time if the conductor of E is much larger than, say, 10^{10} or so. If only the leading term at $s = 1$ is required, over $\mathbb{Q}$ it is faster to use `AnalyticRank` or `ConjecturalRegulator`.

Note also for general number fields it is only conjectured that $L(E/K, s)$ has a meromorphic continuation to $\mathbb{C}$ and also possesses a functional equation. This conjecture is implicitly used in the computations.

Example: L-Series of a Rank 1 Curve `(ex-f74234)`

Consider the curve $E : y^2 + y = x^3 + x^2$ over $\mathbb{Q}$ of conductor 43. It has Mordell–Weil rank equal to 1, so we expect $L(E/\mathbb{Q}, 1) = 0$ and $L'(E/\mathbb{Q}, 1) \neq 0$ by the Birch-Swinnerton-Dyer conjecture.

```
> E := EllipticCurve([0,1,1,0,0]);
> Conductor(E);
43
> L:=LSeries(E);
> Evaluate(L, 1);
0.00000000000000000000000000000000
> Evaluate(L, 1 : Derivative:=1);
0.34352397461847823061807116392 2
```

Now base change E to $K = \mathbb{Q}(i)$. The Mordell–Weil rank of E over K is 2:

```
> Rank(E) + Rank(QuadraticTwist(E,-1));
2
```

So we expect $L(E/K, s)$ to have a zero of order 2:

```
> K := QuadraticField(-1);
> EK := BaseChange(E, K);
> L := LSeries(EK);
> Evaluate(L, 1);
0.00000000000000000000000000000000
> Evaluate(L, 1 : Derivative:=1) lt 10^-20;
true
> Evaluate(L, 1 : Derivative:=2);
1.623995450256000307225469103 44
```

`LSeries(E, K): CrvEll, FldNum -> LSer`

```
Method   : MonStgElt                    Default: "Default"
Precision: RngIntElt                    Default:
```

Given an elliptic curve E defined over the rationals and a number field K, create the L-series $L(E/K, s)$ associated with E/K. Note that in general it is only conjectured that $L(E/K, s)$ has an analytic continuation to $\mathbb{C}$ and possesses a functional equation. This conjecture is implicitly used in the computations.

Technically, the resulting L-series is the tensor product of two l-adic representations, the one associated to $E/\mathbb{Q}$ and the one associated to $K/\mathbb{Q}$. `Method` specifies how `LSeries(K)` should be defined. It is the same parameter as for `LSeries(FldNum)`. Note that the conductor of the L-series $L(E/K, s)$ usually increases very rapidly with the discriminant of K. Consequently, if the used method is `"Direct"` or the irreducible constituents of `PermutationCharacter(K)` have large dimension, the computation time may be quite substantial.

The number of digits of precision to which the values $L(E/K, s)$ are to be computed may be specified using the `Precision` parameter. If it is omitted the precision is taken to be that of the default real field.

Example: Base Change of an Elliptic Curve (ex-9cb4a0)

We take the curve $E : y^2 = x^3 + x$ over the rationals and apply base change to obtain a curve over $\mathbb{Q}(\sqrt{5})$. The resulting L-series $L(E, K, s)$ is in fact the product of $L(E/\mathbb{Q}, s)$ and $L(F/\mathbb{Q}, s)$ where F is E twisted by 5.

```
> E := EllipticCurve([0, 0, 0, 1, 0]);
> F := QuadraticTwist(E, 5);
> L := LSeries(E, QuadraticField(5));
> Evaluate(L, 1);
1.5373382847036052245896606919 5
> Evaluate(LSeries(E),1) * Evaluate(LSeries(F),1);
1.5373382847036052245896606919 5
```

Here is another example over a cyclotomic field. The L-value computations are fast, as here they only involve one-dimensional twists:

```
> E := EllipticCurve([0, 0, 0, 1, 0]);
> L := LSeries(E, CyclotomicField(11));
> time Evaluate(L, 1);
0.00000000000000000000000000000000
Time: 1.560
```

`LSeries(E, A): CrvEll, ArtRep -> LSer`

Precision: RngIntElt	Default:

Twisted L-series of an elliptic curve $E/\mathbb{Q}$ by an Artin representation A.

Example: Twisting by Cyclotomic Characters (ex-50ee8e)

We take the elliptic curve 11A3 and twist it by the characters of $\mathbb{Q}(\zeta_5)/\mathbb{Q}$:

```
> E := EllipticCurve(CremonaDatabase(),"11A3");
> K := CyclotomicField(5);
> art := ArtinRepresentations(K);
> for A in art do Evaluate(LSeries(E,A),1); end for;
0.25384186085591068433775892335 1
0.68597671458851643816988951422 3 + 1.10993363969520543571381847366*$.1
2.83803828204429619496466743332 2
0.68597671458851643816988951422 3 - 1.10993363969520543571381847366*$.1
```

All the L-values are non-zero, so according to the Birch-Swinnerton-Dyer conjecture E has rank 0 over $\mathbb{Q}(\zeta_5)$. Indeed:

```
> #TwoSelmerGroup(BaseChange(E,K));
1
```

Example: Twist by a Quaternion Representation (ex-e42206)

As a higher-dimensional example, we twist $E = X_1(11)/\mathbb{Q}$ by a 2-dimensional Artin representation that factors through a quaternion Galois group.

```
> load galpols;
> E:=EllipticCurve("11a3");           // X_1(11)
> f:=PolynomialWithGaloisGroup(8,5);  // Quaternion Galois group
> K:=NumberField(f);
> A:=ArtinRepresentations(K);
> assert exists(a){a: a in A | Degree(a) eq 2};a;
Artin representation Q8: (2,-2,0,0,0) of ext<Q|x^8-12*x^6+36*x^4-36*x^2+9>
> L:=LSeries(E,a: Precision:=10);
> LCfRequired(L);
208818
> time Evaluate(L,1);
1.678012769
Time: 7.470
> Sign(L);
1.000000000
```

`LSeries(C): CrvHyp[FldRat] -> LSer`

Precision: RngIntElt	Default:
LocalData: List	Default: [* *]

Returns the L-series of a hyperelliptic curve $C/\mathbb{Q}$.

The number of digits of precision to which the values $L(C, s)$ are to be computed may be specified using the `Precision` parameter. If it is omitted the precision is taken to be that of the default real field.

If the conductor exponents and the local factors at (some of) the bad primes are known in advance, they can be passed as a list of tuples `<prime,conductor exponent,local factor>`, e.g. LocalData:=$[*\langle 2, 11, 1 - x\rangle *]$.

Current implementation can only compute the conductor at 2 when $v_2(\Delta) < 12$, using Ogg's formula. If this is not the case, either the local factor at 2 must be supplied in LocalData, or the use of Ogg's formula at 2 forced with `LocalData:="Ogg"` or `[* <2,"Ogg"> *]`.

Example: L-Series of a Hyperelliptic Curve (ex-8fc046)

We take the hyperelliptic curve $y^2 = x^5 + 1$

```
> R<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(x^5+1);
> L := LSeries(C: Precision:=18);
> LCfRequired(L);   // need this number of coefficients
1809
> Evaluate(L,1);  // L(C,1)
1.03140710417331776
> Sign(L); // sign in the functional equation
1.00000000000000000
```

The L-value is non-zero, indicating that the Jacobian should have rank 0. In fact, it does:

```
> RankBound(Jacobian(C));
0
```

`LSeries(C, K): CrvHyp[FldRat], FldNum -> LSer`

`Precision: RngIntElt`	`Default:`
`LocalData: List`	`Default: [* *]`

The L-series associated to a hyperelliptic curve $C/\mathbb{Q}$ base changed to a number field K.

The number of digits of precision to which the values $L(C, s)$ are to be computed may be specified using the `Precision` parameter. If it is omitted the precision is taken to be that of the default real field.

Current implementation can only compute the conductor at 2 when $v_2(\Delta) < 12$, using Ogg's formula. If this is not the case, the use of Ogg's formula at 2 can be forced with `LocalData:="Ogg"`. Also, the local Galois representation machinery is used in the computations, and it is currently not implemented for some of the reduction types.

Example: Base Change of a Hyperelliptic Curve (ex-1a6295)

```
> R<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(x^5-x+1);
> K := CyclotomicField(4);
> L := LSeries(C,K: Precision:=12);
> CheckFunctionalEquation(L);
0.000000000000
```

`LSeries(C): CrvHyp[FldNum] -> LSer`

`Precision: RngIntElt`	`Default:`
`LocalData: List`	`Default: [* *]`

The L-series of a hyperelliptic curve C defined over a number field K.

The number of digits of precision to which the values $L(C/K, s)$ are to be computed may be specified using the `Precision` parameter. If it is omitted the precision is taken to be that of the default real field.

If the conductor exponents and the local factors at (some of) the bad primes are known in advance, they can be passed as a list of tuples `<prime ideal,conductor exponent,local factor>`, e.g. LocalData:=$[*\langle P, 11, 1 - x\rangle *]$.

Current implementation can only compute the conductor at primes $P|2$ when $v_P(\Delta) < 12$, using Ogg's formula. If this is not the case, either the local factor at 2 must be supplied in LocalData, or the use of Ogg's formula at 2 forced with `LocalData:="Ogg"` or `[* <P,"Ogg"> *]`.

A word of warning: as is it the case for most L-functions, the number of Dirichlet coefficients necessary to compute with $L(C/K, s)$ behaves roughly like the square root of the conductor N of the L-function. For a curve C/K of genus g and conductor N_C (an ideal in O_K), this conductor is given by the formula

$$N = |\mathrm{Norm}_{K/\mathbb{Q}}(N_C)| \cdot |\Delta_{K/\mathbb{Q}}|^{2g},$$

It grows very fast, and so in practice the use of these L-series is limited to small genus and small degree $[K : \mathbb{Q}]$, even for curves of small conductor.

Example: Genus 2 Curve over Q(i) (ex-73aef0)

We take a hyperelliptic curve C of genus 2 over $\mathbb{Q}(i)$,

$$C/\mathbb{Q}(i): \quad y^2 + (x^3 + x^2 + x + 1)y = -(i+1)x^3 - (i+1)x^2 - ix,$$

define its L-series $L(C, s)$ and compute $L(C, 1)$ and $L'(C, 1)$.

```
> K<i>:=CyclotomicField(4);
> R<x>:=PolynomialRing(K);
> C:=HyperellipticCurve((-i-1)*x^3+(-i-1)*x^2-i*x,x^3+x^2+x+1);
> L:=LSeries(C: Precision:=8);
> LCfRequired(L);                 // need this many coefficients
7855
> CheckFunctionalEquation(L);     // zero to correct precision
-7.4505806E-9
> Evaluate(L,1);                  // L(C,1) is zero
6.4853393E-10
> Evaluate(L,1: Derivative:=1);   // L'(C,1) is non-zero
0.31311835
```

According to the Birch–Swinnerton-Dyer conjecture, the Jacobian of $C/\mathbb{Q}(i)$ should have rank 1. We can verify that it has indeed trivial 2-torsion, 2-Selmer group of size 2, and that there are non-trivial rational points on C:

```
> f:=HyperellipticPolynomials(SimplifiedModel(C));
> J:=Jacobian(HyperellipticCurve(f));
> #TwoTorsionSubgroup(J),#TwoSelmerGroup(J);
1 2
> Points(C: Bound:=10);
{@ (1 : 0 : 0), (1 : -1 : 0), (0 : -1 : 1), (0 : 0 : 1), (i - 1 : -1 : 1),
   (i - 1 : -i - 1 : 1), (-i : 1 : 1), (-i : -1 : 1) @}
```

`LSeries(Chi): GrpDrchElt -> LSer`

```
Precision: RngIntElt                    Default:
```

Given a primitive dirichlet character $\chi : (\mathbb{Z}/m\mathbb{Z})^* \to \mathbb{C}^*$, create the associated Dirichlet L-series $L(\chi, s) = \sum_{n=1}^{\infty} \chi(n)/n^s$. The character χ must be defined so that its values fall in either the ring of integers, the rational field, or a cyclotomic field.

The number of digits of precision to which the values $L(\chi, s)$ are to be computed may be specified using the `Precision` parameter. If it is omitted the precision is taken to be that of the default real field.

For information on Dirichlet characters, see Section Dirichlet Characters.

Example: Dirichlet L-Function Modulo 37 (ex-ac2d65)

We define a primitive character $\chi : (\mathbb{Z}/37\mathbb{Z})^* \to \mathbb{C}^*$ and construct the associated Dirichlet L-function.

```
> G<Chi> := DirichletGroup(37, CyclotomicField(36));
> L := LSeries(Chi);
> Evaluate(L,1); // depends on the chosen generator of G
1.65325576836885655776002342451 - 0.55160789892291080587553771593 5*$.1
```

`LSeries(hmf): ModFrmHilElt -> LSer`

```
Precision: RngIntElt                    Default:
```

Given a cuspidal newform in a space of Hilbert modular forms, this creates the associated L-series. Note that this is **not** implemented for Bianchi modular forms, as the computations of Hecke eigenvalues (particularly at bad primes) is not always feasible in that case.

Example: L-Series of a Hilbert Modular Form (ex-846e3d)

```
> K := NumberField(x^2 - 5) where x is PolynomialRing(Rationals()).1;
> H := HilbertCuspForms(K,7*Integers(K),[2,2]);
> f := NewformDecomposition(NewSubspace(H))[1];
> L := LSeries(Eigenform(f));
> LSetPrecision(L,9);
> LCfRequired(L); // approximate value, with old CFE
198
> time CFENew(L); // new vers of CheckFunctionalEquation
0.000000000
Time: 9.690
```

`LSeries(omf): ModFrmAlgElt -> LSer`

```
Precision: RngIntElt                    Default: 0
UseAuto  : BoolElt                      Default: true
LowMemory: BoolElt                      Default: false
ThetaPrec: RngIntElt                    Default: 25
```

Given a cuspidal newform in a space of orthogonal modular forms for a quadratic space of rank 3 or 5, this creates the associated L-series. Note that this is currently **not** implemented for other spaces of algebraic modular forms, particularly unitary modular forms, as the computations of Hecke eigenvalues at bad primes is not always implemented in that case. The parameters `UseAuto`, `LowMemory`, and `ThetaPrec` are for computing the Hecke operators, as in `HeckeOperator` for algebraic modular forms.

Example: L-Series of an Orthogonal Modular Form (ex-cc704a)

```
> L := QuinaryQuadraticLattices(61)[1][1];
> M := OrthogonalModularForms(L);
> fs := HeckeEigenforms(M);
> f := fs[2];
> L := LSeries(f);
> LSetPrecision(L,10);
> LCfRequired(L); // approximate value, with old CFE
66
> time CFENew(L); // new vers of CheckFunctionalEquation
0.000000000
Time: 355.350
```

`LSeries(psi): GrpHeckeElt -> LSer`

`LSeries(psi): GrossenChar -> LSer`

Precision: RngIntElt	Default:

Given a primitive Hecke (Grössen)character on ideals, construct the associated L-series.

For more information on these see Section Hecke Grössencharacters and their L-functions.

`LSeries(f): ModFrmElt -> LSer`

Embedding: Map/UserProgram	Default:
Precision: RngIntElt	Default:

Given a modular form f, construct the L-series $L(f,s) = \sum_{n=1}^{\infty} a_n/n^s$, where f has the q-expansion $\sum_{n=0}^{\infty} a_n q^n$. It is assumed that $L(f,s)$ satisfies a functional equation of the standard kind (see Section *Terminology* for the precise form of the functional equation).

The optional parameter `embedding` specifies a map which embeds the coefficients of f into the complex field. By default this is the identity map, so that the coefficients of f must be coercible into $\mathbb{C}$. Otherwise, the value of the parameter must either be an object of type `Map` or a user-defined function $e(x)$ each having domain the base ring of f and codomain the complex field (or values than can be coerced into the complex field).

The number of digits of precision to which the values $L(f,s)$ are to be computed may be specified using the `Precision` parameter. If it is omitted the precision is taken to be that of the default real field.

Example: Newform over the Gaussian Integers (ex-0f42a0)

We define a newform of weight 2 and conductor 16. It is not defined over the integers but rather over $B = \mathbb{Z}[i]$.

```
> f := Newforms("G1N16k2A")[1]; f;
q + (-a - 1)*q^2 + (a - 1)*q^3 + 2*a*q^4 + (-a - 1)*q^5 + 2*q^6 - 2*a*q^7 +
   (-2*a + 2)*q^8 + a*q^9 + 2*a*q^10 + (a + 1)*q^11 + O(q^12)
> B:=BaseRing(f); B;
Equation Order with defining polynomial x^2 + 1 over its ground order
```

The two distinct embeddings of B into the complex numbers give rise to two modular forms, which can be accessed using the `ComplexEmbeddings` function.

```
> f1, f2 := Explode(ComplexEmbeddings(f)[1]);
> Coefficient(f,2), Coefficient(f1,2), Coefficient(f2,2);
-a - 1
-1.000000000000000000000000000000 + 1.000000000000000000000000000000*$.1
   -1.000000000000000000000000000000 - 1.000000000000000000000000000000*$.1
```

Thus, f_1 and f_2 have genuine complex coefficients and we can construct the associated L-series and compute their L-values, for instance at $s = 1$.

```
> L1 := LSeries(f1);
> L2 := LSeries(f2);
> CFENew(L1); // new vers of CheckFunctionalEquation
4.733165431326070832470371391 70E-30
> CFENew(L2);
4.733165431326070832470371391 70E-30
> v1 := Evaluate(L1,1); v2 := Evaluate(L2,1); v1,v2;
0.359306437003505684066327207778 + 0.071470493999117268658845806 6909*$.1
   0.359306437003505684066327207778 - 0.071470493999117268658845806 6909*$.1
```

If instead we invoke `LSeries(f)`, Magma will note that f is defined over a number field and complain that the coefficients of f are not well-defined complex numbers.

```
> L := LSeries(f);
Runtime error: For f over a number field, you have to specify a complex
   embedding
```

Instead of using `ComplexEmbeddings`, one can instead explicitly specify an embedding of the coefficients of B into the complex numbers using the parameter `Embedding` with the function `LSeries` The following statements define the same L-function as L_2 above.

```
> C<i> := ComplexField();
> L2A := LSeries(f: Embedding:=hom< B -> C | i > );
> L2B := LSeries(f: Embedding:=func< x | Conjugates(x)[1] > );
> L2C := LSeries(f1: Embedding:=func< x | ComplexConjugate(x) > );
```

Finally, we illustrate the very important fact that Magma expects, but does *not* check that the L-function associated to a modular form satisfies a functional equation.

```
> L := LSeries(f1+f2);  // or L:=LSeries(f: Embedding:=func<x|Trace(B!x)>);
```

Although Magma is happy with this definition, it is in fact illegal. The modular form f has a character whose values lie in the field of the 4-th roots of unity.

```
> Order(DirichletCharacter(f));
4
```

The two embeddings f_1 and f_2 of f have *different* (complex conjugate) characters and $f_1 + f_2$ does not satisfy a functional equation of the standard kind. Magma will suspect this when it tries to determine the sign in the functional equation and thereby print a warning:

```
> Evaluate(L,1);
 |Sign| is nowhere near 1, wrong functional equation?
0.36370614319615175049306340784 3
> CFENew(L); // new vers of CheckFunctionalEquation
0.29356085273807234095668455673 9
```

The function `CheckFunctionalEquation` or `CFENew` should return 0 (to current precision), so the functional equation is not satisfied, and the result of evaluating L will be a random number. So it is the *user's* responsibility to ensure that the modular form does satisfy a functional equation as described in Section *Terminology*.

`LSeries(S): ModSym -> LSer`

```
Precision: RngIntElt                    Default:
```

Given a 1-dimensional cuspidal modular symbol space, construct the L-series determined by its Hecke operators (for the Euler factors). In fact, the dual Hecke operators are used for efficiency purposes.

Example: L-Series from Modular Symbols (ex-52ab37)

We give two examples, the first of a largish level where the computations might otherwise be difficult, and the second for one where the space has a nontrivial character.

```
> S := CuspidalSubspace(ModularSymbols(864,4,+1)); // wt 4
> x := PolynomialRing(Integers()).1;
> K := Kernel([<5,x-19>,<7,x-13>],S); // 19 at p=5, 13 at p=7
> assert Dimension(K) eq 1;
> L := LSeries(K);
> LCfRequired(L); // approx for old CheckFunctionalEquation
467
```

(continues on next page)

(continued from previous page)

```
> CFENew(L);
0.00000000000000000000000000000000
> time DualHeckeOperator(K,997);
[31536]
Time: 0.040
```

```
> chi := DirichletGroup(4).1; // nontrivial character modulo 4
> S := CuspidalSubspace(ModularSymbols(chi,5,+1)); // weight 5
> L := LSeries(S);
> CFENew(L);
0.00000000000000000000000000000000
> EulerFactor(L,2);
4*x + 1
> DualHeckeOperator(S,2);
[-4]
```

139.3 Computing L-values

Once an L-series $L(s)$ has been constructed using either a standard L-function (Section *Built-in L-series*), a user defined L-function (Section *Constructing a General L-Series*) or constructed from other L-functions (Section *Arithmetic with L-series*), Magma can compute values $L(s_0)$ for complex s_0, values for the derivatives $L^{(k)}(s_0)$, and Taylor expansions.

Evaluate(L, s0): LSer, FldComElt -> FldComElt

```
Derivative: RngIntElt                    Default: 0
Leading   : BoolElt                      Default: false
```

Given the L-series L and a complex number s_0, the intrinsic computes either $L(s_0)$, or if $D > 0$, the value of the derivative $L^{(D)}(s_0)$. If $D > 0$ and it is known that all the lower derivatives vanish,

$$L(s_0) = L'(s_0) = ... = L^{(D-1)}(s_0) = 0\,,$$

the computation time can be substantially reduced by setting `Leading:=true`. This is useful if it is desired to determine experimentally the order of vanishing of $L(s)$ at s_0 by successively computing the first few derivatives.

CentralValue(L): LSer -> FldComElt

Given an L-function of motivic weight $2k-1$, the value of L is computed at $s = k$.

LStar(L, s0): LSer, FldComElt -> FldComElt

```
Derivative: RngIntElt                    Default: 0
```

Given the L-series L and a complex number s_0, the intrinsic computes either the value $L^*(s_0)$ or, if $D > 0$, the value of the derivative $L^{*(D)}(s_0)$. Here $L^*(s) = \gamma(s)L(s)$ is the modified L-function that satisfies the functional equation (cf. Section *Terminology*)

$$L^*(s) = \text{sign} \cdot \bar{L}^*(\text{weight} - s)$$

(cf. Section *Terminology*).

LTaylor(L, s0, n): LSer, FldComElt, RngIntElt -> FldComElt

```
ZeroBelow: RngIntElt                     Default: 0
```

Compute the first $n+1$ terms of the Taylor expansion of the L-function about the point $s = s_0$, where s_0 is a complex number:

$$L(s_0) + L'(s_0)x + L''(s_0)x^2/2! + \ldots + L^{(n)}(s_0)x^n/n! + O(x^{n+1}) \ .$$

If the first few terms $L(s_0), ..., L^{(k)}(s_0)$ of this expansion are known to be zero, the computation time can be reduced by setting `ZeroBelow:=k+1`.

Example: Derivatives of a Rank 3 L-Series `(ex-7d80f5)`

We define an elliptic curve E of conductor 5077 and compute derivatives at $s = 1$ until a non-zero value is reached:

```
> E := EllipticCurve([0, 0, 1, -7, 6]);
> L := LSeries(E : Precision:=15);
> Evaluate(L, 1);
0.000000000000000
> Evaluate(L, 1 : Derivative:=1, Leading:=true);
1.87710082755801E-24
> Evaluate(L, 1 : Derivative:=2, Leading:=true);
-6.94957228421048E-24
> Evaluate(L, 1 : Derivative:=3, Leading:=true);
10.3910994007158
```

This suggests that $L(E, s)$ has a zero of order 3 at $s = 1$. In fact, E is the elliptic curve (over $\mathbf{Q}$) of smallest conductor with Mordell–Weil rank 3:

```
> Rank(E);
3
```

Consequently, a zero of order 3 is predicted by the Birch–Swinnerton-Dyer conjecture. We can also compute a few terms of the Taylor expansion about $s = 1$, with or without specifying that the first three terms vanish.

```
> time LTaylor(L, 1, 5 : ZeroBelow:=3);
1.73184990011930*$.1^3 - 3.20590558844390*$.1^4 + 2.80009237167013*$.1^5 +
   O($.1^6)
Time: 0.800
> time LTaylor(L, 1, 5);
1.87710082755801E-24*$.1 - 3.47478614210524E-24*$.1^2 + 1.73184990011930*$.1^3
   - 3.20590558844390*$.1^4 + 2.80009237167013*$.1^5 + O($.1^6)
Time: 1.530
```

And this is the leading derivative, with the same value as `Evaluate(L,1:D:=3)`.

```
> c := Coefficient($1,3)*Factorial(3);c;
10.3910994007158
```

Finally, we compute the 3rd derivative of the modified L-function $L^*(s) = \gamma(s)L(s)$ at $s = 1$. For an elliptic curve over the rationals, $\gamma(s) = (N/\pi^2)^{s/2}\Gamma(s/2)\Gamma((s+1)/2)$, where N is the conductor. So, by the chain rule, $L^{*'''}(1) = \gamma(1)L'''(1) = \sqrt{N/\pi}L'''(1)$.

```
> LStar(L, 1 : Derivative:=3);
417.724689268266
> c*Sqrt(Conductor(E)/Pi(RealField(15)));
417.724689268267
```

`DedekindZetaExact(K, z): Fld, RngIntElt -> FldRatElt`

```
Relative: BoolElt                    Default: false
```

The special value of the Dedekind zeta function of K at the negative integer z. Returns $\zeta_K(z)$ or $\frac{\zeta_K(z)}{\zeta_F(z)} = L(\chi_{K/F}, z)$ if K is a quadratic extension of F and `Relative` is `true`. In the latter case, can also compute the value at $z = 0$.

139.4 General L-series

In addition to the built-in L-series for the standard objects, Magma provides machinery that allows the user to define L-series for arbitrary objects, provided that they admit a meromorphic continuation to the whole complex plane and satisfy a functional equation of the standard kind. To describe how this may be done, we need to introduce some terminology.

139.4.1 Terminology

Recall that an *L-series* is a sum

$$L(s) = \sum_{n=1}^{\infty} \frac{a_n}{n^s},$$

where the coefficients a_n are complex numbers, known as the Dirichlet coefficients of the L-series. For example, the Dirichlet coefficients of the classical Riemann zeta function are $a_n = 1$ for all n. We assume that, as in the case of Riemann zeta function, every $L(s)$ has a meromorphic continuation to the complex plane and has a functional equation. Technically, our assumptions are as follows:

Assumption 1. The defining series for $L(s)$ converges for $\mathrm{Re}(s)$ sufficiently large. Equivalently, the *coefficients* a_n grow at worst as a polynomial function of n.

Assumption 2. $L(s)$ admits a meromorphic continuation to the entire complex plane.

Assumption 3. The following exist: real positive *weight*, complex *sign* of absolute value 1, real positive *conductor*, and the Γ-*factor*

$$\gamma(s) = \Gamma\Big(\frac{s+\lambda_1}{2}\Big) \cdots \Gamma\Big(\frac{s+\lambda_d}{2}\Big)$$

of *dimension* $d \geq 1$ and rational γ-*shifts* $\lambda_1, \ldots \lambda_d$, such that

$$L^*(s) = \Big(\frac{\text{conductor}}{\pi^d}\Big)^{s/2} \gamma(s)\, L(s)$$

satisfies the *functional equation*

$$L^*(s) = \text{sign} \cdot \bar{L}^*(\text{weight} - s).$$

Here $\bar{L}$ is the *dual* L-series with complex conjugate coefficients $L(s) = \sum_{n=1}^{\infty} \overline{a_n}/n^s$. Note that the "weight" here is one more than the usual motivic notion of weight.

Assumption 4. The series $L^*(s)$ has finitely many simple *poles* and no other singularities.

Assumption 1 will almost certainly be true for any naturally arising L-function and Assumptions 2–4 are expected (but not proven) to be satisfied for most of the L-functions arising in geometry and number theory. In fact, there is a large class of so-called *motivic L-functions* for which all of the above assumptions are conjectured to be true. Essentially this class consists of L-functions associated to cohomology groups of varieties over number fields. This is an extremely large class of L-functions that includes all of the standard examples.

139.4.2 Constructing a General L-Series

When computing the values $L(s)$ for a complex number s, Magma relies heavily on the functional equation. This means that the *emphasized* parameters in Assumptions 1–4 (coefficients, weight, conductor, poles, etc.) must be known before the computations can be carried out.

The generic LSeries function allows the user to construct an L-series for a new object by specifying values for these parameters. In fact, this function is used to construct all of the built-in L-series in Magma and some of these will be used as examples to illustrate its use (see the advanced examples section).

```
LSeries(weight, gamma, conductor, cffun): FldReElt, [FldRatElt], FldReElt, Any -> LSer
```

```
Sign              : FldComElt                    Default: 0
Poles             : SeqEnum                      Default: []
Residues          : SeqEnum                      Default: []
Parent            : Any                          Default:
CoefficientGrowth: UserProgram                   Default:
Precision         : RngIntElt                    Default:
ImS               : FldReElt                     Default: 0
Asymptotics       : BoolElt                      Default: true
```

The function takes four arguments: real positive *weight*, sequence of rational numbers *gamma*, real positive *conductor*, and a coefficient function *cffun* together with a number of optional parameters. It constructs an L-series with specified coefficients and the form of the functional equation.

Compulsory arguments (see Section *Terminology* for terminology):

weight — weight of the L-series.

Specifies the functional equation; for instance, it is 1 for the Riemann zeta function, 2 for curves over the rationals, and Weight(f) for modular forms f.

Note that the "weight" in this sense is the w such that $s \to w - s$ satisfies a functional equation. The usual motivic notion of weight is thus one less than this – for instance, a weight k modular form has weight k in Magma, but weight $k-1$ for motives, as its coefficients are typically of size $(\sqrt{p})^{k-1}$.

gamma — gamma shifts/parameters.

List (of type SeqEnum) of rational numbers that specify the gamma factor. For instance, this is [0] for the Riemann zeta function, [0,1] for an elliptic curve over $\mathbb{Q}$ and [0,...,0,1,...,1] with $r_1 + r_2$ zeros and r_2 ones for the Dedekind zeta function of a number field K with r_1 real and r_2 pairs of complex embeddings.

conductor — conductor of the L-series.

Part of the exponential factor $(\frac{\text{conductor}}{\pi^d})^{s/2}$ in the functional equation. It is usually an integer and is 1 for the Riemann zeta function, the level for a modular form, |Discriminant(K)| for the Dedekind zeta of a number field K and the conductor of the Jacobian for an algebraic curve over the rationals.

cffun — specifies the coefficients a_n. The following possibilities are allowed:

- A finite sequence $[a_1, ..., a_n]$;
- A function $f(n)$ that returns a_n;
- A function $f(p, d)$ that computes the inverse of the local factor at p up to degree d;
- The value 0, signifying that the coefficients will be provided later with a call to LSetCoefficients.

See Section *Setting the Coefficients* for a detailed explanation and examples.

Optional parameters for Lseries::

Sign — the sign appearing in the functional equation.

This is the sign that enters the functional equation and is usually ± 1. It must be a complex number of absolute value 1. Alternatively, this can be set to 0 (default), in which case Magma will determine it numerically from the functional equation.

Instead of providing the actual sign, the user may also use an option `Sign:=s` where s is the name of a function $s(p)$ or $s(L, p)$ that computes the sign to precision p.

`Poles` — the poles z of $L^*(s)$ with Re $z \geq$weight/2 and

`Residues` — the residues at these poles

If $L^*(s)$ happens to have poles, there are only finitely many of them and each must be simple by Assumption 4 of the previous subsection. The poles have to be known and specified here when the L-function is created. The poles are symmetric about the point $s =$weight/2 by the functional equation and only the "right half" of the set of them has to be supplied in the form of a sequence as the value of the parameter `Poles`. The default setting is that $L^*(s)$ has no poles.

The residues at the poles of $L^*(s)$ are also required and they can be specified using the `Residues` parameter, which also takes a sequence as its value. It can be either a sequence of the same length as `Poles` or left to its default setting `[]`. In that case, Magma will attempt to determine the residues numerically using the functional equation. However, this is only possible if `Sign` is known and will not work to full precision if there is more than one pair of poles.

As an example, the Riemann zeta function has weight 1 and the modified function $\zeta^*(s)$ has poles at $s = 0$ and $s = 1$ with residues 1 and -1 respectively. So the parameter assignments `Poles:=[1], Residues:=[-1]` would be used in its definition.

As in the case of `Sign`, instead of providing the actual residues, the user may give the name of a function $r(p)$ or $r(L, p)$ that computes the residues up to precision p and returns them as a sequence.

`Parent` — any Magma object.

This is only used for printing purposes. When a variable type `LSer` is printed, Magma prints "L-series of" and then prints the parent object.

`Precision` — the precision to which L-values are to be computed (default is 0, use current precision).

`CoefficientGrowth` — name f of a function $f(x)$ or $f(L, x)$ such that $|a_n| < f(n)$.

`ImS` — the largest imaginary part of s for which $L(s)$ will be evaluated.

`Asymptotics` — whether to use asymptotic expansions (default is yes).

These four settings are related to the precision to which values are calculated in L-series computations. They are the same as for the `SetPrecision` function and are described in detail in Section *Precision*.

As an illustration, the following code creates the Riemann zeta function from its invariants. This is essentially what `RiemannZeta()` does:

```
> Z := LSeries(1, [0], 1, func<n|1>
>                         : Sign:=1, Poles:=[1], Residues:=[-1]);
> CFENew(Z);
0.00000000000000000000000000000000
```

`LSeries(HS, N, cffun): HodgeStruc, RngIntElt, Any -> LSer`

This is an alternative way to specify a generic L-series, with all the varargs of before. Instead of giving a real number for the functional equation reflection together with γ-shifts, one passes a Hodge structure to the LSeries intrinsic instead.

For more examples, see the "Advanced Examples" Section *Advanced Examples*.

It is strongly advised that whenever a generic L-series is defined the user apply one of the functions `CFENew` or `CheckFunctionalEquation`. If one of the specified parameters (conductor, sign, etc.) happens to be incorrect, the resulting L-series will not have a functional equation and the values returned by the evaluation functions will be nonsense. Only by checking the functional equation will the user have an indication that something is wrong.

`CheckFunctionalEquation(L): LSer -> FldComElt`

`CFENew(L): LSer -> FldReElt`

```
t: FldReElt                    Default: 1.2
```

Given an L-series L, this intrinsic tests the functional equation numerically and should ideally return 0 (to the current precision), meaning that the test was passed.

The new preferred version is `CFENew`, which in many cases of reasonable interest can often require signficantly fewer terms (using a dynamic criterion, rather than a static one), sometimes by a factor of 5 or more. It also is much superior at keeping track of internal precision. However, it does not completely replicate the old machinery (particularly when poles/residues are present), and so `CheckFunctionalEquation` is also still available.

If the value returned is a significant distance from 0, either the L-function does not have a functional equation or some of the defining parameters (conductor, poles etc.) have been incorrectly specified or not enough coefficients have been given. In the latter case, `CFENew` or `CheckFunctionalEquation` is likely to return a number reasonably close to 0 that measures the accuracy of computations.

As already mentioned, Magma can only work with L-functions that satisfy the functional equation as in Section *Terminology*. Whenever an L-function is constructed, its functional equation is implicitly used in all the L-series evaluations, even when L is evaluated in the region where the original Dirichlet series is absolutely convergent.

If either the sign in the functional equation or the residues of $L^*(s)$ have not yet been computed, this function will first compute them. If the sign was undefined, it returns $|\texttt{Sign}| - 1$ which, again, must be 0.

The optional technical parameter `t` is a point on the real line where Magma evaluates the two Theta functions associated to the L-series and subtracts one from the other to get the value returned by `CFENew` or `CheckFunctionalEquation`.

Example: Checking a Functional Equation (ex-37a053)

We attempt to define a truncated L-series of a quadratic character mod 3 (so $\chi(n) = 0, 1, -1$ if n is 0, 1, or 2 mod 3 respectively.)

```
> L := LSeries(1, [0], 3, [1,-1,0,1,-1,0] : Sign:=-1);
> CFENew(L);
0.22234625740398941924564655806 9
```

This does not look right. In fact, the γ-shifts are 0 for even quadratic characters, but 1 for odd ones,

```
> L := LSeries(1, [1], 3, [1,-1,0,1,-1,0] : Sign:=-1);
> CheckFunctionalEquation(L);
1.063726235879220898712968226 24
> CFENew(L);
7.47942058598558334301138589732E-18
```

This still does not look right (though `CFENew` will ignore `Sign`). We gave the wrong sign in the functional equation.

```
> L := LSeries(1, [1], 3, [1,-1,0,1,-1,0] : Sign:=1);
> CFENew(L);
7.47942058598558334301138589732E-18
```

This certainly looks better but it indicates that we did not give enough coefficients to our L-series. We determine how many coefficients are needed to do computations with default precision (30 digits),

```
> LCfRequired(L); // approx for old CheckFunctionalEquation
11
```

So 11 coefficients are needed and we provide them in the code below.

```
> L := LSeries(1, [1], 3, [1,-1,0,1,-1,0,1,-1,0,1] : Sign:=1);
> CFENew(L);
0.00000000000000000000000000000000
```

This a correct way to define our L-series. Even better is the following variant:

```
> L := LSeries(1, [1], 3, func<n|((n+1) mod 3)-1> : Sign:=1);
> CFENew(L);
0.00000000000000000000000000000000
```

This allows Magma to calculate as many a_n as it deems necessary using the provided function.

139.4.3 Setting the Coefficients

`LSetCoefficients(L, cffun): LSer, Any -> {}`

This function defines the coefficients a_n of the L-series L. The argument *cffun* can be one of the following:

- A sequence $[a_1, ..., a_n]$;
- A function $f(n)$ that returns a_n;
- A function $f(p, d)$ that computes the inverse of the local factor at p up to degree d.

When a user-defined L-series is constructed by invoking the function

```
> L := LSeries(weight,gamma,conductor,cffun: <optional parameters>);
```

this is actually equivalent to invoking the pair of functions

```
> L := LSeries(weight,gamma,conductor,0: <optional parameters>);
> LSetCoefficients(L,cffun);
```

where the first line indicates that the coefficients will be supplied later and the second line actually specifies the coefficients. So the following description of the *cffun* argument for the function `LSetCoefficients(L,cffun)` applies to the main `LSeries` signature as well.

The first two ways to specify the a_n are either to give a pre-computed sequence of coefficients up to a certain bound or to give the name of a function $f(n)$ that computes the a_n. (The other two ways are described in the two subsections that follow.) For instance, the following both define the Riemann zeta function with weight 1, one γ-shift 0, conductor 1, sign 1, pole at $s = 1$ with residue -1, all $a_n = 1$:

```
> V := [ 1 : k in [1..100] ];
> L := LSeries(1, [0], 1, V : Sign:=1, Poles:=[1], Residues:=[-1]);
```

or

```
> f := func<n|1>;
> L := LSeries(1, [0], 1, f : Sign:=1, Poles:=[1], Residues:=[-1]);
```

Of the two possibilities, the second one is safer to use in a sense that it lets Magma decide how many coefficients it needs in order to perform the calculations to the required precision. However, if the computation of a_n is costly and it involves previous coefficients, then it is probably better to write a function that computes a_n recursively, storing them in a sequence and then passing it to `LSeries`.

139.4.4 Specifying the Coefficients Later

When specifying a finite list of coefficients it is necessary to know in advance how many coefficients have to be computed. For this, Magma provides a function `LCfRequired(L)` (see Section *Accessing the Invariants*).

Example: Riemann Zeta with Deferred Coefficients (ex-147e8a)

We define L to be our own version of the Riemann zeta function (weight 1, one γ-shift 0, conductor 1, sign 1, pole at $s = 1$ with residue -1), but tell Magma that we will specify the coefficients later with the 4th parameter set to 0. Then we ask how many coefficients it needs to perform computations:

```
> L := LSeries(1, [0], 1, 0: Sign:=1, Poles:=[1], Residues:=[-1]);
> N := LCfRequired(L); N;
6
```

Now we compute the coefficient vector $[a_1, ..., a_N]$.

```
> vec := [1, 1, 1, 1, 1, 1];
> LSetCoefficients(L,vec);
```

Now we can evaluate our ζ-function

```
> Evaluate(L,2);
1.64493406684822643647241516665
> Pi(RealField())^2/6;
1.64493406684822643647241516665
```

If we provide fewer coefficients, the computations will not have full precision but we can use `CFENew` or `CheckFunctionalEquation` to get an indication of the resulting accuracy.

```
> LSetCoefficients(L, [1,1]);
> CFENew(L);
1.49504837351354351195383363489E-10
> Evaluate(L, 2);
1.64493406707833102002503055804
> $1 - Pi(RealField(28))^2/6;
2.30104583552615391305838556E-10
```

Note that such good accuracy with just two coefficients is a singular phenomenon that only happens for L-functions with very small conductors. Normally, at least hundreds of coefficients are needed to compute L-values to this precision.

Note also that the last value `Evaluate(L,2)` in the example is much more precise than what one would get with the truncated version of the original defining series $\zeta_{trunc}(s) = 1 + 1/2^s$ at $s = 2$. The reason for this is that even in the region where the original Dirichlet series converges, the use of the functional equation usually speeds up the convergence.

139.4.5 Generating the Coefficients from Local Factors

The final and, in many cases, mathematically the most natural way to supply the coefficients is by specifying the so-called *local factors*. Recall that the coefficients for most L-series are *weakly multiplicative*, meaning that $a_{mn} = a_m a_n$ for m coprime to n. For such L-series, $L(s)$ admits a *product formula*

$$L(s) = \prod_{p \text{ prime}} \frac{1}{F_p(p^{-s})},$$

where $F_p(x)$ is a formal power series in x with complex coefficients. For example, if $L(s)$ arises from a variety over a number field (and this, as we already mentioned, essentially covers everything), then such a product formula holds. In this case the $F_p(x)$ are polynomials of degree d for those primes p not

dividing the conductor and are of smaller degree otherwise. Moreover, their coefficients lie in the ring of integers of the field of definition of the variety.

For L-functions with weakly multiplicative coefficients, Magma allows the coefficients to be defined by specifying the local factors, using the function `LSetCoefficients(L,f)` where f is a user-defined function with two arguments p and d that computes $F_p(x)$, either as a full polynomial in x or as a power series in x of precision $O(x^{d+1})$. For example, the Riemann zeta function has $F_p(x) = 1 - x$ for all p, so we can define it as follows,

```
> Z := LSeries(1, [0], 1, 0 : Poles:=[1], Residues:=[-1], Sign:=1);
> P<x> := PolynomialRing(Integers());
> LSetCoefficients(Z, func<p,d | 1-x> );
```

or as

```
> P<x> := PowerSeriesRing(Integers());
> Z := LSeries(1, [0], 1, func<p,d|1-x+O(x^(d+1))> :
               Poles:=[1], Residues:=[-1], Sign:=1);
```

139.5 Accessing the Invariants

`LCfRequired(L): LSer -> RngIntElt`

The number of Dirichlet coefficients a_n that have to be calculated in order to compute the values $L(s)$. This function can be also used with a user-defined L-series before its coefficients are set, see Section *Specifying the Coefficients Later*.

`LGetCoefficients(L, N): LSer, RngIntElt -> List`

Compute the vector of first N coefficients `[*` $a_1, ..., a_N$ `*]` of the L-series given by L.

`EulerFactor(L, p): LSer, RngIntElt -> RngElt`

```
Degree   : RngIntElt                    Default:
Precision: RngIntElt                    Default:
```

Given an L-series and a prime p, this computes the pth Euler factor, either as a polynomial or a power series. The optional parameter `Degree` will truncate the series to that length, and the optional parameter `Precision` is of use when the series is defined over the complex numbers.

`Degree(L): LSer -> RngIntElt`

The degree of an L-function.

`Conductor(L): LSer -> RngElt`

Conductor of the L-series (real number, usually an integer). This invariant enters the functional equation and measures the 'size' of the object to which the L-series is associated. Evaluating an L-series takes time roughly proportional to the square root of the conductor.

`Sign(L): LSer -> RngElt`

Sign in the functional equation of the L-series. This is a complex number of absolute value 1, or 0 if the sign has not been computed yet. (Calling `CFENew` or `CheckFunctionalEquation(L)` or any evaluation function sets the sign.)

`MotivicWeight(L): LSer -> RngIntElt`

The motivic weight of an L-function.

`GammaFactors(L): LSer -> SeqEnum`

A sequence of Gamma factors $\lambda_1, \ldots, \lambda_d$ for $L(s)$. Each one represents a factor $\Gamma((s + \lambda_i)/2)$ entering the functional equation of the L-function.

With the previous two intrinsics, see also `HodgeStructure`.

`LSeriesData(L): LSer -> Info`

Given an L-series L, this function returns the weight, the conductor, the list of γ-shifts, the coefficient function, the sign, the poles of $L^*(s)$, and the residues of $L^*(s)$ as a tuple of length 7. If Sign $= 0$, this means it has not been computed yet. Residues= [] means they have not yet been computed. From this data, L can be re-created with a general `LSeries` call (see Section *Constructing a General L-Series*).

`BadPrimeData(L): LSer -> SeqEnum`

Given an L-series L, this returns an array of bad prime data, each entry being a 3-tuple $\langle p, v, U\rangle$ where p is the prime, v is the conductor valuation at p, and U is the (possibly trivial) Euler factor at p.

Example: Coefficients of a Modular Form L-Series (ex-f37c5f)

For a modular form, the q-expansion coefficients are the same as the Dirichlet coefficients of the associated L-series:

```
> f := Newforms("30k2")[1,1];
> qExpansion(f,10);
q - q^2 + q^3 + q^4 - q^5 - q^6 - 4*q^7 - q^8 + q^9 + O(q^10)
> Lf := LSeries(f);
> LGetCoefficients(Lf,20);
[* 1, -1, 1, 1, -1, -1, -4, -1, 1, 1, 0, 1, 2, 4, -1, 1, 6, -1, -4, -1 *]
```

The elliptic curve of conductor 30 that corresponds to f has, of course, the same L-series.

```
> E := EllipticCurve(f); E;
Elliptic Curve defined by y^2 + x*y + y = x^3 + x + 2 over Rational Field
> LE := LSeries(E);
> LGetCoefficients(LE,20);
[* 1, -1, 1, 1, -1, -1, -4, -1, 1, 1, 0, 1, 2, 4, -1, 1, 6, -1, -4, -1 *]
```

Now we change the base field of E to a number field K and evaluate the L-series of E/K at $s = 2$.

```
> P<x> := PolynomialRing(Integers());
> K := NumberField(x^3-2);
> LEK := LSeries(E,K);
> i := LSeriesData(LEK); i;
<2, [ 0, 0, 0, 1, 1, 1 ], 8748000, function(p, d [ Precision ]) ... end
   function, 1, [], []>
```

The conductor of this L-series (second entry) is not that small and this is an indication that the calculations of $L(E/K, 2)$ to the required precision (30 digits) will take some time. We can also ask how many coefficients will be used in this calculation.

```
> LCfRequired(LEK);
24636
```

Decreasing the precision will help somewhat.

```
> LSetPrecision(LEK,9);
> LCfRequired(LEK);
3364
```

Magma now automatically does a factorization of the L-series (see the Arithmetic section).

```
> LEK`prod;
[
    <L-series of twist of Elliptic Curve defined by y^2 + x*y + y = x^3 + x + 2
```

(continues on next page)

(continued from previous page)

```
    over Rational Field by Artin representation S3: (1,1,1) of ext<Q|x^3-2>, 1>,
    <L-series of twist of Elliptic Curve defined by y^2 + x*y + y = x^3 + x + 2
    over Rational Field by Artin representation S3: (2,0,-1) of ext<Q|x^3-2>,
    conductor 108, 1>
]
```

Factorization(L): LSer -> SeqEnum[Tup]

Factorisation(L): LSer -> SeqEnum[Tup]

If an L-series is represented internally as a product of other L-series, say $L(s) = \prod_i L_i(s)^{n_i}$, return the sequence [...<L_i,n_i>...].

Example: Factorization of L-Series (ex-52681e)

```
> L := RiemannZeta();
> Factorization(L);
[ <L-series of Riemann zeta function, 1> ]
> R<x> := PolynomialRing(Rationals());
> K := SplittingField(x^3-2);
> L := LSeries(K);
> Factorization(L);
[
    <L-series of Riemann zeta function, 1>,
    <L-series of Artin representation S3: (1,-1,1) of ext<Q|x^6+108>, conductor
3, 1>,
    <L-series of Artin representation S3: (2,0,-1) of ext<Q|x^6+108>, conductor
108, 2>
]
```

139.6 Modifying the L-function

There is some capability to modify an L-function, for instance if one were trying to experimentally determine bad Euler factors, you might not want to recompute the known (good) Euler factors repeatedly with a new L-function.

ChangeEulerFactor(L, p, f): LSer, RngIntElt, RngUPolElt -> LSer

Given an L-series that computes its coefficients as Euler factors, a prime, and a suitable polynomial, create an L-function with the given Euler factor at said prime. This copies over the known coefficients.

Be warned that the new L-series has a function to compute Euler Factors which consists of one line to check if the prime is the special one, and then calls the old function. So recursively calling this function is not wise. It is (much) superior to create a new L-series with this intrinsic, compute with the new L-series, and then use CopyCoefficients to copy back to the first L-series, then creating more new L-series from it (and not from the second one).

ChangeLocalInformation(L, p, d, f): LSer, RngIntElt, RngIntElt, RngUPolElt -> LSer

Similar to ChangeEulerFactor, but include the new local conductor as the third argument.

ChangeLocalInformation(L, bp): LSer, List -> LSer

Similar to ChangeEulerFactor, but now the third argument is a list containing bad prime information, each entry being a tuple consisting of the prime, the local conductor, and the Euler factor (as a polynomial).

CopyCoefficients(L, M): LSer, LSer

Copy the known coefficients array of the first L-series to the second. If the latter has more known coefficients, then those are retained at the end.

Example: Changing Local Information (ex-05dc99)

We give a short example of how changing local information can be done. We create the L-series hypergeometric motive at the inverse value of the M-parameter, where the wild primes can often become good.

```
> H := HypergeometricData([2,3],[1,6]);
> MValue(H);
1/4
> L := LSeries(H,4);
WARNING: Guessing wild prime information
> BadPrimeData(L);
[ <2, 2, 4*x^2 + 2*x + 1>,  <3, 3, -3*x + 1> ]
> LCfRequired(L); // approx for old CheckFunctionalEquation
287
> CFENew(L);
0.09937654841436797096340869695555
```

At this point Magma really has no knowledge of the bad prime information at 3. Its guess of $(1-3x)$ turns out to be incorrect, as is the conductor exponent at 3. By trial and error, we find that the Euler factor at 3 is trivial, and the local conductor is 4. We can change the old L-series to a new one with `ChangeLocalInformation`, and then check the functional equation. Finally (and importantly when one works via trial-and-error), we copy any extra coefficients computed with the second L-series back to the first.

```
> M := ChangeLocalInformation(L,[* <3,4,1> *]);
> LCfRequired(M);
503
> CFENew(M);
0.00000000000000000000000000000000
> #L`cfvec; // length 260
260
> CopyCoefficients(M,L); // M back to L
> #L`cfvec; // now length 450
450
```

139.7 Precision

The values of an L-series are computed numerically and the precision required in these computations can be specified using the `Precision` parameter when an L-function is created. For example,

```
> K := CyclotomicField(3);
> L := LSeries(K: Precision:=60);
> Evaluate(L,2);
1.285190955484149402917511798699574603969178397028921243951 33
```

computes $\zeta(K,2)$ to 60 digits precision. The default value `Precision:=0` is equivalent to the precision of the default real field at the time of the `LSeries` initialization call. If the user only wants to check numerically whether an L-function vanishes at a given point s and the value itself is not needed, it might make sense to decrease the precision (to 9 digits, say) to speed up the computations.

```
> E := EllipticCurve([0,0,1,-7,6]);
> RootNumber(E);
-1
> L := LSeries(E: Precision:=9);
> Evaluate(L, 1: Derivative:=1);
1.50193628E-11
```

(continues on next page)

(continued from previous page)

```
> Rank(E);
3
```

This example checks numerically that $L'(E,1) = 0$ for the elliptic curve E of conductor 5077 from Example *Derivatives of a Rank 3 L-Series*.

The precision may be changed at a later time using `LSetPrecision`.

`LSetPrecision(L, precision): LSer, RngIntElt -> {}`

```
CoefficientGrowth: UserProgram                    Default:
ImS              : FldReElt                       Default: 0
Asymptotics      : BoolElt                        Default: true
```

Change the number of digits to which the L-values are going to be computed to *precision*. The parameter `CoefficientGrowth` is described below in Section *L-series with Unusual Coefficient Growth*.

139.7.1 L-series with Unusual Coefficient Growth

The parameter `CoefficientGrowth` is the name f of a function $f(x)$ (or $f(L,x)$, where L is an L-series) which is an increasing function of a real positive variable x such that $|a_n| \leq f(n)$. This is used to truncate various infinite series in computations. It is set by default to the function $f(x) = 1.5 \cdot x^{\rho-1}$ where ρ is the largest real part of a pole of $L^*(s)$ if $L^*(s)$ has poles and $f(x) = 2x^{(weight-1)/2}$ if $L^*(s)$ has no poles. The user will most likely leave this setting untouched.

139.7.2 Computing $L(s)$ when Im(s) is Large (ImS Parameter)

If s is a complex number having a large imaginary part, a great deal of cancellation occurs while computing $L(s)$, resulting in a loss of precision. (The time when all the precision-related parameters are pre-computed is when the function `LSeries` is invoked, and at that time Magma has no way of knowing whether $L(s)$ is to be evaluated for complex numbers s having large imaginary part.) If this happens, a message is printed, warning of a precision loss. To avoid this, the user may specify the largest Im(s) for which the L-values are to be calculated as the value of the `ImS` parameter at the time of the L-series initialization or, later, with a call to `LSetPrecision`:

```
> C<i> := ComplexField();
> L := RiemannZeta();
> Evaluate(L, 1/2+40*i);            // wrong
Warning: Loss of 13 digits due to cancellation
0.79304495256192867198212885104 5 - 1.04127461465106502007452195086*i
> LSetPrecision(L, 30: ImS:=40);
> Evaluate(L, 1/2+40*i);            // right
0.79304495256192867196489258889 8 - 1.04127461465106502005189059539*i
```

139.7.3 Implementation of L-series Computations (Asymptotics Parameter)

The optional parameter `Asymptotics` in `LSetPrecision` and in the general `LSeries` function specifies the method by which the special functions needed for the L-series evaluation are computed.

If set to `false`, Magma will use only Taylor expansions at the origin and these are always known to be convergent. With the default behaviour (`Asymptotics:=true`) Magma will use both the Taylor expansions and the continued fractions of the asymptotic expansions at infinity. This is much faster, but these continued fractions are not proved to be convergent in all cases.

139.8 Verbose Printing

There is one verbose printing variable, called "LSeries" that controls verbose printing for all of the functions in this chapter.

```
> SetVerbose("LSeries",n);
```

It accepts verbosity levels n such that $0 \leq n \leq 3$. The values are as follows:

$n = 0$ is the (default) quiet mode. The functions do not print anything apart from loss-of-precision warnings.

$n = 1$ is possibly the most useful setting. The sign and the residues of $L^*(s)$ are printed whenever they are determined from the functional equation.

$n = 2$ generates messages that allow the user to keep track of what is currently happening. So messages are printed when a new L-function is constructed, when coefficients are generated, or when expansions of special functions are computed.

$n = 3$ is a "keep-alive" setting that basically informs the user that the computer has not crashed yet and, yes, something is still happening. For L-functions of large conductor or very high precision that require a large number of coefficients, Magma will print a "progress indicator" every time 1000 new coefficients are generated and every 5000 terms in the series computations that test the functional equation or compute the L-values. (The frequencies 1000 and 5000 are stored in the attributes L`vprint_coeffs and L`vprint_series of a variable L of type LSeries and may be changed at the user's desire.)

139.9 Arithmetic with L-series

With the exception of some modular forms, all the built-in L-series have weakly multiplicative coefficients, so that $L(s) = \sum a_n/n^s$ with $a_{mn} = a_m a_n$ for m, n coprime. For two such L-series, Magma allows the user to construct their product and, provided that it makes sense, their quotient.

There are also various methods to construct a new L-function from old L-functions, namely tensor products and symmetrizations. The notion of Hodge structure is also useful to have.

L1 * L2: LSer, LSer -> LSer

```
Poles     : SeqEnum                    Default: []
Residues  : SeqEnum                    Default: []
Precision: RngIntElt                   Default:
```

Let $L_1(s)$ and $L_2(s)$ be two L-series of the same weight whose coefficients are weakly multiplicative, that is, they satisfy $a_{mn} = a_m a_n$ for m, n coprime. This function constructs their product $L(s) = L_1(s)L_2(s)$.

If one of the L-series has zeros that cancel the poles of the other L-series, the user should specify the list of poles for $L_1^*(s)L_2^*(s)$ using the Poles parameter and the corresponding residues using the Residues parameter. See Section *Terminology* for the terminology and Section *Constructing a General L-Series* for the format of the poles and residues parameters.

The number of digits of precision to which the values $L(s)$ are to be computed may be specified using the Precision parameter. If it is omitted, the precision is taken to be that of the default real field.

L1 / L2: LSer, LSer -> LSer

```
Poles     : SeqEnum                    Default: []
Residues  : SeqEnum                    Default: []
Precision: RngIntElt                   Default:
```

Let $L_1(s)$ and $L_2(s)$ be two L-series whose coefficients a_{mn} are weakly multiplicative, that is, they satisfy $a_{mn} = a_m a_n$ for m, n coprime. This function constructs their quotient $L(s) = L_1(s)/L_2(s)$.

This function assumes (but does not check!) that this quotient exists and is a genuine L-function with finitely many poles.

If $L_2(s)$ happens to have zeros that give poles in the quotient, the user must specify the list of poles of $L_1^*(s)/L_2^*(s)$ using the `Poles` parameter and the corresponding residues using the `Residues` parameter. See Section *Terminology* for the terminology and Section *Constructing a General L-Series* for the format of `Poles` and `Residues`.

The number of digits of precision to which the values $L(s)$ are to be computed may be specified using the `Precision` parameter. If it is omitted, the precision is taken to be minimum of that of the inputs.

139.9.1 Hodge Structure

Magma can compute with Hodge structures of an L-series. In the sense of Deligne [Del79, Table 5.3], this is printed as a series of $\langle p, q\rangle$ pairs and $\langle p, p, \epsilon\rangle$ triples, where the former have $p < q$ and correspond to $H^{p,q}$ and $H^{q,p}$, and with the latter $F_\infty = (-1)^{p+\epsilon}$ is the action of complex conjugation on $H^{p,p}$. An alternative printing (of the `HodgeVector`) can be obtained from the vararg `PrintHodgeVector` or `PHV` when creating a Hodge structure.

Hodge structures of the same weight (possibly negative) can be added or subtracted (though virtual Hodge structures are not currently allowed). They can also be tensored and inverted (the latter negates all p, q) via the multiplication operator and `Dual`.

`HasHodgeStructure(L): LSer -> BoolElt, HodgeStruc`

`HodgeStructure(L): LSer -> HodgeStruc`

Given an L-series, determine if it has a Hodge structure. The `HodgeStructure` intrinsic can also be applied to various other objects.

`HodgeStructure(X): SeqEnum -> HodgeStruc`

Given a sequence (or other aggregate) of tuples corresponding to Hodge (p, q)'s, construct the given Hodge structure. Here both (p, q) and (q, p) must be given (for redundancy), and (p, p, ϵ) must have its sign given as a third element in the tuple.

`HodgeStructure(w, G): RngIntElt, SeqEnum -> HodgeStruc`

Given a weight and a sequence of γ-shifts, return the Hodge structure.

`Dual(HS): HodgeStruc -> HodgeStruc`

Given a Hodge structure, negate all the (p, q)'s.

`TateTwist(HS, k): HodgeStruc, RngIntElt -> HodgeStruc`

Given a Hodge structure and an integer, take the Tate twist of the structure. This subtracts k from all the p and q, decreasing the weight by $2k$. This corresponds to translating the L-function in the complex plane.

`Translate(L, z): LSer, RngIntElt -> LSer`

`Translate(L, z): LSer, FldRatElt -> LSer`

Given an L-series, translate it by the given integer (or rational).

`GammaFactors(HS): HodgeStruc -> SeqEnum`

`GammaShifts(HS): HodgeStruc -> SeqEnum`

`Degree(HS): HodgeStruc -> RngIntElt`

`Weight(HS): HodgeStruc -> RngIntElt`

The γ-factors (or shifts), degree, and (motivic) weight of a Hodge structure.

`EffectiveHodgeStructure(HS): HodgeStruc -> HodgeStruc, RngIntElt`

The Hodge structure obtained by shifting so that all $p, q \geq 0$ with at least one of them equal to zero. Also returns the shift amount.

RootNumber(HS): HodgeStruc -> FldCycElt

The root number at ∞ of a Hodge structure, which is a 4th root of unity.

Note that this is the reciprocal of that given by Deligne in [Del79, Table 5.3].

TensorProduct(H1, H2): HodgeStruc, HodgeStruc -> HodgeStruc

This is the standard operation on Hodge structures, also accessible via the '*' operator.

SymmetricPower(HS, m): HodgeStruc, RngIntElt -> HodgeStruc

The mth symmetric power of a Hodge structure.

Determinant(HS): HodgeStruc -> HodgeStruc

The determinant of a Hodge structure.

AlternatingSquare(HS): HodgeStruc -> HodgeStruc

The alternating square of Hodge structure. The general symmetrization intrinsic for Hodge structures has not been implemented at this time, but these last examples are probably the more interesting cases.

HodgeVector(HS): HodgeStruc -> SeqEnum, RngIntElt

Given a Hodge structure, return its Hodge vector, and the Tate twist to make it effective. This is an array of $h^{p,q}$ starting which the first nonzero.

CriticalPoints(HS): HodgeStruc -> SeqEnum

CriticalPoints(L): LSer -> SeqEnum

Given a Hodge structure or an L-series with a Hodge structure, return its critical points. In the case that the list is infinite, it will be truncated at height 100.

ImaginaryTwist(HS): HodgeStruc -> HodgeStruc

Take the tensor product with the nontrivial weight 0 Hodge structure.

Example: Hodge Structures of L-Series (ex-271a08)

Some examples of Hodge structures and operations.

```
> LE := LSeries(EllipticCurve("11a"));
> HE := HodgeStructure(LE); HE;
Hodge structure of weight 1 given by <0,1>
> Lf := LSeries(Newforms(CuspForms(9,4))[1][1]);
> Hf := HodgeStructure(Lf); Hf;
Hodge structure of weight 3 given by <0,3>
> PR := Translate(LE,1)*Lf;
> HodgeStructure(PR);
Hodge structure of weight 3 given by <0,3> <1,2>
> TateTwist(HE,-1)+Hf; // direct sum
Hodge structure of weight 3 given by <0,3> <1,2>
> TP := TensorProduct(LE,Lf);
> HodgeStructure(TP);
Hodge structure of weight 4 given by <0,4> <1,3>
> HE*Hf; // tensor product
Hodge structure of weight 4 given by <0,4> <1,3>
> RootNumber(HodgeStructure([[0,0,1]])); // imag quad field
-zeta_4
> RootNumber(HE); // elliptic curve
-1
> RootNumber(Hf); // wt 4 modform
1
> assert SymmetricPower(HE,2) eq HodgeStructure(SymmetricPower(LE,2));
> CriticalPoints(SymmetricPower(Hf,2));
[ 1, 3, 4, 6 ]
```

(continues on next page)

(continued from previous page)

```
> CriticalPoints(ImaginaryTwist(SymmetricPower(Hf,2)));
[ 2, 5 ]
```

139.9.2 Tensor Products

The above discussion with Hodge structure folds nicely into the `TensorProduct` machinery.

`TensorProduct(L1, L2, ExcFactors): LSer, LSer, [<>] -> LSer`

`TensorProduct(L1, L2, ExcFactors, K): LSer, LSer, [<>], FldNum -> LSer`

`TensorProduct(L1, L2): LSer, LSer -> LSer`

`TensorProduct(L1, L2, K): LSer, LSer, FldNum -> LSer`

```
BadPrimes: SeqEnum                    Default: []
Precision: RngIntElt                  Default:
Sign     : FldComElt                  Default:
```

Let L_1 and L_2 be L-functions such that $L_1(s) = L(V_1, s)$ and $L_2(s) = L(V_2, s)$ are associated to systems of l-adic representations V_1 and V_2 (à la Serre). This function computes their tensor product $L(s) = L(V_1 \otimes V_2, s)$. This can be used, for example, to twist an L-function by characters or higher-dimensional Artin representations (see Examples *Tensor Product of l-Adic L-Series*, *BSD in a Non-Abelian Extension*).

Note that, in particular, both $L_1(s)$ and $L_2(s)$ must have integer conductor, weakly multiplicative coefficients, and an underlying Hodge structure (which is computed from the γ-shifts). The argument `ExcFactors` (or the vararg `BadPrimes`) is a list of tuples of the form $\langle p, v\rangle$ or $\langle p, v, F_p(x)\rangle$ that give, for each of the primes p where V_1 and V_2 both have bad reduction, the valuation v of the conductor of $V_1 \otimes V_2$ at p and the inverse local factor at p. If the data is not provided for such a prime p, Magma will attempt to compute the local factors by assuming that the inertia invariants behave well at p,

$$(V_1 \otimes V_2)^{I_p} = V_1^{I_p} \otimes V_2^{I_p}.$$

It will also compute the conductor exponents by predicting the tame and wild degrees from the degrees of the local factors, but this does not work if both V_1 and V_2 are wildly ramified at p.

The sign in the functional equation of $L(V_1 \otimes V_2, s)$ cannot be determined from the signs of the factors, so it will be calculated numerically from the functional equation. If the sign is known, the user may specify it by means of the `Sign` parameter.

The number of digits of precision to which the values $L(s)$ are to be computed may be specified using the `Precision` parameter. If it is omitted, the precision is taken to be the minimal precision among the tensor parts.

Some tensor products can induce poles, particularly when tensoring an L-series with itself. Also in some cases, the tensor product will be taken over $\mathbf{Q}$ unless the user specifies otherwise.

However, if L_1 and L_2 have Euler products over the same field K, this field can be given as an additional argument, and the tensor product will be taken with respect to that field. Namely, we take two L-functions with Euler products over a field K:

$$L_1(M, s) = \prod_{\mathfrak{p}} \prod_{i=1}^{m} \left(1 - \alpha_i(\mathfrak{p})/\mathrm{N}\mathfrak{p}^s\right)^{-1} \quad \text{and} \quad L_2(N, s) = \prod_{\mathfrak{p}} \prod_{j=1}^{n} \left(1 - \beta_j(\mathfrak{p})/\mathrm{N}\mathfrak{p}^s\right)^{-1},$$

where M and N are of degrees m and n respectively, and we ignore bad Euler factors in the above. Their tensor product is an L-function of degree mn given by

$$L(A \otimes B, s) = \prod_{\mathfrak{p}} \prod_{i=1}^{m} \prod_{j=1}^{n} \left(1 - \alpha_i(\mathfrak{p})\beta_j(\mathfrak{p})/\mathrm{N}\mathfrak{p}^s\right)^{-1}.$$

It is not clear in general how to compute bad Euler factors, and even the gamma factors can be tricky. Internally, the functionality tries to compute a "Hodge structure" corresponding to the gamma factors of each object, with the "Hodge structure" for the tensor product then following via combinatorics, before passing back to the gamma factors.

Example: Tensor Product of Two Elliptic Curves (ex-8a6681)

A tensor product of two elliptic curves over **Q**.

```
> E1 := EllipticCurve("11a");
> E2 := EllipticCurve("17a");
> L1 := LSeries(E1);
> L2 := LSeries(E2);
> L := TensorProduct(L1, L2, []);
> LSeriesData(L); // level is 11^2 * 17^2
<3, [ -1, 0, 0, 1 ], 34969, function(p, d [ Precision ]) ... end function, 0,
   [], []>
> CFENew(L);
0.00000000000000000000000000000000
```

Example: Tensor Product of Level 1 Modular Forms (ex-0e486d)

A tensor product of two modular forms of level 1.

```
> f1 := ModularForms(1,12).2;
> f2 := ModularForms(1,26).2;
> L1 := LSeries(f1);
> L2 := LSeries(f2);
> L := TensorProduct(L1, L2, []);
WARNING: Modular form is not known to be a newform
WARNING: Modular form is not known to be a newform
> LSeriesData(L); // weight is (12-1)+(26-1)+1 -- motivic weight is 11+25
<37, [ -11, -10, 0, 1 ], 1, function(p, d [ Precision ]) ... end function, 0,
   [], []>
> CFENew(L);
0.00000000000000000000000000000000
> Pi(RealField(30))^2 * Evaluate(L,24) / Evaluate(L,25);
9.87142857142857142857142857142
> 691/70.; // Ramanujan congruence
9.87142857142857142857142857142
```

Example: Tensor Product and a Siegel Modular Form (ex-f0736e)

An example related to Siegel modular forms (see [vGvS93, §8.7]).

```
> E := EllipticCurve("32a"); // congruent number curve
> chi := DirichletGroup(32).1; // character of conductor 4 lifted
> MF := ModularForms(chi, 3); // weight 3 modular forms on Gamma1(32,chi)
> NF := Newforms(MF);
> NF[1][1]; // q-expansion of the desired form
q + a*q^3 + 2*q^5 - 2*a*q^7 - 7*q^9 - a*q^11 + O(q^12)
> Parent(Coefficient(NF[1][1], 3)); // defined over Q(i)
Number Field with defining polynomial x^2 + 16 over the Rational Field
> f1, f2 := Explode(ComplexEmbeddings(NF[1][1])[1]);
> L1 := LSeries(E);
> L2 := LSeries(f1); // first complex embedding
> L := TensorProduct(L1, L2, [ <2, 9> ]); // conductor 2^9 (guessed)
WARNING: Modular form is not known to be a newform
> time CFENew(L);
```

(continues on next page)

(continued from previous page)

```
3.155443620884047221646914261 13E-30
Time: 1.900
```

Example: Tensor Product over a Quadratic Field (ex-f991ed)

An example of a tensor product over a field larger than the rationals.

```
> K<s> := QuadraticField(-3);
> I := Factorization(3 * IntegerRing(K))[1][1];
> H := HeckeCharacterGroup(I^2);
> G := Grossencharacter(H.0, [[1, 0]]); // canonical character
> E := EllipticCurve([1, (3+s)/2, 0, (1+s)/2, 0]);
> Norm(Conductor(E));
73
> LG := LSeries(G);
> LE := LSeries(E);
> TP := TensorProduct(LE, LG, [<I, 5>], K); // ensure 3-part correct
> LSetPrecision(TP, 9); // there is another factor of 3 from K
> LCfRequired(TP); // approx for old CheckFunctionalEquation
1642
> CFENew(TP);
0.000000000
```

Example: Consani-Scholten Tensor Product (ex-1ad661)

Another example of a tensor product over a field larger than the rationals. This comes from work of Consani and Scholten [CS01b], and the motivic realisation was later proven by Dieufelait, Pacetti, and Schütt [DPS12].

```
> K := NumberField(x^2 - 5) where x is PolynomialRing(Rationals()).1;
> SetStoreModularForms(K, true); // to save computation time
> f5 := Factorization(5*Integers(K))[1][1];
> H := HilbertCuspForms(K,2*3*f5,[2,4]);
> N := NewformDecomposition(NewSubspace(H)); // 4-dimensional
> f11 := Factorization(11*Integers(K))[1][1]; // get right form
> f := [f : f in N | HeckeEigenvalue(Eigenform(f),f11) in
>                                          {2*K.1-58,-2*K.1-58}][1];
> Lf := LSeries(Eigenform(f));
> psi := HeckeCharacterGroup(f5,[1,2]).1;
> Lpsi := LSeries(psi);
> R := PolynomialRing(Integers());
> TP := TensorProduct(Lf,Lpsi,[<f5,3,R!1>],K); // 5^4 in all, 1 from K
```

The above constructs the L-series of a specific Hilbert modular form of weight $[2, 4]$ of level $6\sqrt{5}$ over $\mathbf{Q}(\sqrt{5})$, and then twists it by a nontrivial Hecke character. This corresponds to twisting the Fourier coefficients by a Dirichlet character of this field, which yields a Hilbert modular form at level $6(\sqrt{5})^2$. We construct this latter form below.

We then compare the L-series data (to ensure that Magma correctly worked with the Hodge structure over the real quadratic field in the tensor product), and Euler factors up to 100 (a weak check). The use of `CFENew` or `CheckFunctionalEquation` to (say) 6 digits takes a few minutes, so we omit it here.

```
> H := HilbertCuspForms(K, 2*3*f5^2, [2,4]);
> time N := NewformDecomposition(NewSubspace(H));
Time: 1.410
> g := [g : g in N | Dimension(g) eq 1 and
>                  HeckeEigenvalue(Eigenform(g),f11) in {2*K.1-58,-2*K.1-58}][1];
```

(continues on next page)

(continued from previous page)

```
> Lg := LSeries(Eigenform(g));
> assert forall{ i : i in [1..3] |
>                LSeriesData(TP)[i] eq LSeriesData(Lg)[i] };
> // weak check of Euler factors
> assert forall{ p : p in PrimesUpTo(100) |
>                EulerFactor(TP, p : Degree:=1, Integral) eq
>                EulerFactor(Lg, p : Degree:=1, Integral) };
> // f and g may be Galois conjugates, so compare the coefficients
> // as multisets over primes above each rational prime.
> PK := PrimesUpTo(100, K);
> res := func< p | Characteristic(ResidueClassField(p)) >;
> assert forall{ c : c in {res(p) : p in PK} |
>                Multiset([Round(Coefficient(EulerFactor(TP, p), 1))
>                          : p in PK | res(p) eq c]) eq
>                Multiset([Round(Coefficient(EulerFactor(Lg, p), 1))
>                          : p in PK | res(p) eq c]) };
>
```

139.9.3 Symmetric Powers

Symmetric power L-functions form a natural analogue to tensor products. Here we essentially tensor an L-function with itself repeatedly, but remove redundant factors.

In the case of $GL(1)$, the kth symmetric power is simply the L-function associated to the kth power of the underlying character, though we must be careful to ensure primitivity (this disregards the bad primes). Explicitly, we have

$$L(\mathrm{Sym}^k \psi, s) = \prod_{\mathfrak{p}} \left(1 - \psi(\mathfrak{p})^k / \mathrm{N}\mathfrak{p}^s\right)^{-1},$$

(again ignoring bad primes) and so the eigenvalues are just the kth powers of the original. It is relatively easy to compute the bad Euler factors, given those for $L(\psi, s)$.

In the case of $GL(2)$, the kth symmetric power is an L-function of degree $(k+1)$ over the field of definition, given by

$$L(\mathrm{Sym}^k A, s) = \prod_{\mathfrak{p}} \prod_{i=0}^{k} \left(1 - \alpha_1(\mathfrak{p})^{k-i} \alpha_2(\mathfrak{p})^i / \mathrm{N}\mathfrak{p}^s\right)^{-1},$$

where $\alpha_1(\mathfrak{p})$ and $\alpha_2(\mathfrak{p})$ are the eigenvalues at the prime $\mathfrak{p}$. In the case of elliptic curves, one can use the fact that $\alpha_1(p)\alpha_2(p) = p$ to rewrite these via symmetric functions.

A similar definition can be made for higher degree L-functions, yielding that the kth symmetric power of an L-function of degree d will have degree $\binom{k+d-1}{d-1}$.

If the object itself has a natural powering operation (as with, say, Hecke characters), Magma will simply take the primitivization therein. The bad Euler factors have been explicitly calculated for elliptic curves over the rationals in [MW06] and [DMW09]. In other cases, the user will likely have to provide them. Furthermore, the full ability to take symmetric powers over fields other than the rationals is not yet fully implemented.

More general symmetrizations are also possible, with the determinant and the alternating square perhaps being the most useful.

`Determinant(L): LSer -> LSer`

```
Translate: BoolElt                    Default: false
```

This computes the determinant L-series, returned as a translate of an L-function of a Dirichlet character.

`SymmetricPower(L, m): LSer, RngIntElt -> LSer`

```
BadPrimes: SeqEnum                    Default: []
```

Return the L-series corresponding to the mth symmetric power of L. The `BadPrimes` vararg consists of $\langle p, f, E\rangle$ triples, where p is a prime, f the conductor exponent, and E a polynomial that gives the Euler factor at that prime.

Example: Symmetric Powers of a Dirichlet Character (ex-2b6618)

Some basic code, showing the special cases where Magma will just take the power of the underlying object.

```
> G := FullDirichletGroup(3*5*7);
> chi := G.1*G.2*G.3;
> L := LSeries(chi);
> LS3 := SymmetricPower(L, 3);
> Lc3 := LSeries(chi^3);
> Evaluate(LS3, 1);
0.84396449805350779437298478447 2 + 0.19923299213711678303364580375 3*i
> Evaluate(Lc3, 1);
0.84396449805350779437298478447 2 + 0.19923299213711678303364580375 3*i
```

Example: Symmetric Powers of a Grossencharacter (ex-833dcd)

In this example we take symmetric powers of the L-function of a Grossencharacter.

```
> K := QuadraticField(-23);
> I := Factorization(23 * IntegerRing(K))[1][1];
> G := HeckeCharacterGroup(I);
> psi := G.1^14;
> L := LSeries(psi);
> LS5 := SymmetricPower(L, 5);
> Lp5 := LSeries(psi^5);
> Evaluate(LS5, 1);
0.87080188482438164758363114981 4 + 0.62258929156395483122972653081 3*i
> Evaluate(Lp5, 1);
0.87080188482438164758363114981 4 + 0.62258929156395483122972653081 3*i
> GR := Grossencharacter(psi, [[1,0]]);
> L := LSeries(GR);
> LS4 := SymmetricPower(L, 4);
> Lp4 := LSeries(GR^4);
> Evaluate(LS4, 3);
4.4860849121343336627832724307 1 + 0.42480936796734327076566791945 0*i
> Evaluate(Lp4, 3);
4.4860849121343336627832724307 1 + 0.42480936796734327076566791945 0*i
```

Example: Symmetric Powers of an Elliptic Curve (ex-b9aa14)

An example with symmetric powers of elliptic curves. In the case of the symmetric square of 389A, the `ModularDegree` code will do the same calculation, but more efficiently.

```
> E := EllipticCurve("389a");
> L := LSeries(E);
> L2 := SymmetricPower(L, 2);
> LSeriesData(L2);
<3, [ 0, 1, 0 ], 151321, function(p, d) ... end function, 1, [], []>
> LSetPrecision(L2, 9);
> Evaluate(L2, 2);
3.17231145
```

(continues on next page)

(continued from previous page)

```
> ($1 * Conductor(E)) / (2 * Pi(RealField()) * FundamentalVolume(E));
40.0000000
> ModularDegree(E);
40
```

Example: Symmetric Power with Central Vanishing (ex-5c1fea)

This is an example where a higher symmetric power has a vanishing central value. It comes from work of Buhler, Schoen, and Top [BST97], who also compute that the observed analytic rank of the symmetric cube of 2379b is 4.

```
> E := EllipticCurve("73a"); // conductor 73
> L := LSeries(E);           // note this is < 389 (!)
> L3 := SymmetricPower(L, 3);
> LSeriesData(L3); // Magma knows the Sign is +1
<4, [ 0, -1, 1, 0 ], 389017, function(p, d) ... end function, 1, [], []>
> LSetPrecision(L3, 9);
> CentralValue(L3); // analytic rank 2
-1.14998623E-16
```

`Symmetrization(L, p): LSer, SeqEnum -> LSer`

```
BadPrimes: SeqEnum                    Default: []
PoleOrder: RngIntElt                  Default: 0
Induction: BoolElt                    Default: true
```

The `Symmetrization` intrinsic is the general workhorse. It takes an L-function and a partition (weakly decreasing sequence of nonnegative integers) as inputs. The `BadPrimes` vararg is as with other cases, while `PoleOrder` specifies that an appropriate translate of the Riemann ζ-function will be a factor of the result (Magma knows about this is some cases). Finally, the `Induction` vararg controls how $GL(1)$ objects over fields larger than $\mathbf{Q}$ are handled.

Magma will reduce the symmetrization, so that, for example, the $\{6,4\}$-symmetrization of a degree 2 L-function will just be the translate by 4 of its symmetric square. However, the practical symmetrizations that can be computed are still rather limited. Thus, for reasons of efficiency, Magma contains hardcoded formulas for small degree examples. For degree 2 these include all symmetric powers. For degree 3 it includes the symmetric cube, the alternating square ($\{1,1\}$-symmetrization), and $\{2,1\}$-symmetrization, not to mention the determinant (which is always available). For degree 4 Magma has the symmetric and alternating squares, and for degree 5 the alternating square is available. Furthermore, for objects such as Artin representations, `Symmetrization` can be applied to the object itself, and the resulting L-function is (highly) imprimitive, easing computations.

`IsOrthogonal(L): LSer -> BoolElt`

`IsSymplectic(L): LSer -> BoolElt`

```
AssumeTrue : BoolElt                  Default: false
AssumeFalse: BoolElt                  Default: false
```

Given an L-function return whether it is orthogonal (respectively symplectic). Firstly this means it is self-dual, that is it has real coefficients (a check is done for the first few values). If the motivic weight is odd, then it is symplectic but not orthogonal. If the degree is odd, then it is orthogonal but not symplectic. If both the degree and motivic weight are even, then currently the user must specify `AssumeTrue` or `AssumeFalse` for a result to be given.

`OrthogonalSymmetrization(L, p): LSer, SeqEnum -> LSer`

`SymplecticSymmetrization(L, p): LSer, SeqEnum -> LSer`

```
BadPrimes: SeqEnum                          Default: []
PoleOrder: RngIntElt                        Default: 0
```

Given an orthogonal L-function and a suitable partition p, return the orthogonal symmetrization. Referring to Section Symmetrization, the symmetrization of L can factor into smaller parts. In particular, one can throw out "old" factors, and concentrate on those that are truly new. In general, one should have that the degree exceeds twice the number of parts in the partition (see the "modification rules" in the above reference), as these form a basis for the representation space (up to twisting).

In degree 2, the relevant partitions are of the form (m), and all the orthogonal symmetrizations themselves have degree 2. In degree 3, again the relevant partitions are singletons, and the orthogonal mth power has degree $2m+1$. Magma allows one to consider $m \leq 4$ via hand-coded formulas. In degree 4 the practical partitions are (2) and $(2,2)$, these leading to degree 9 and 10 L-functions respectively.

Similarly one can compute the symplectic symmetrization of a symplectic L-function. The only (interesting) available case is for degree 4 L-functions, when the $(1,1)$ partition gives a L-function of degree 5.

Example: General Symmetrizations of a Modular Form (ex-631189)

Here we consider some general symmetrizations. We first take a modular form of level 25 and weight 4, so a degree 2 L-function of motivic weight 3, and take various symmetrizations of it.

```
> x := PolynomialRing(Integers()).1;
> Lf := LSeries(Newforms(ModularForms(25,4))[1][1]);
> S2 := Symmetrization(Lf,[2] : BadPrimes:=[<5,2,1+125*x>]);
> CFENew(S2);
0.00000000000000000000000000000000
> S31 := Symmetrization(Lf,[3,1] : BadPrimes:=[<5,2,1+5^7*x>]);
> CFENew(S31);
0.00000000000000000000000000000000
> LSetPrecision(Lf,10);
> S3 := Symmetrization(Lf,[3] : BadPrimes:=[<5,4,1>]);
> CFENew(S3);
0.0000000000
> S4 := Symmetrization(Lf,[4] : BadPrimes:=[<5,4,1-125^2*x>]);
> CFENew(S4);
5.820766091E-11
```

Another (unitary) example in degree 2 is from a weight 2 newform with complex character. Here we take a "semistable" example, as for such L-functions it is relatively easy to compute the bad prime info.

```
> f := Newforms(CuspForms(FullDirichletGroup(13).1^2,2))[1][1];
> L := LSeries(f : Embedding:=func<x|Conjugates(x)[1]>);
> BP := [<13,2,Polynomial([1,-(-Coefficient(EulerFactor(L,13),1))^2])>];
> S2 := Symmetrization(L,[2] : BadPrimes:=BP);
> CFENew(S2); // deg 3, conductor 169
0.00000000000000000000000000000000
> LSetPrecision(L,15);
> BP := [<13,3,Polynomial([1,-(-Coefficient(EulerFactor(L,13),1))^3])>];
> S3 := Symmetrization(L,[3] : BadPrimes:=BP);
> CFENew(S3); // deg 4, conductor 13^3
8.88178419700125E-16
> LSetPrecision(L,5);
> BP := [<13,4,Polynomial([1,-(-Coefficient(EulerFactor(L,13),1))^4])>];
> S4 := Symmetrization(L,[4] : BadPrimes:=BP);
```

(continues on next page)

(continued from previous page)

```
> CFENew(S4); // deg 5, conductor 13^4
0.00000
```

Example: Orthogonal Symmetric Cube (ex-401e29)

We next take a degree 3 L-function of (motivic) weight 2, and compute its symmetric cube. Note that this is a degree 10 L-function, but when the input is orthogonal this original L-function (translated appropriately) appears as a factor.

```
> H := HypergeometricData([1,1,1],[2,2,2]);
> L := LSeries(H,2 : BadPrimes:=[<2,8,1>],SaveEuler:=10000);
> LSetPrecision(L,5);
> assert IsOrthogonal(L); // degree 3 weight 2
> S3 := Symmetrization(L,[3] : BadPrimes:=[<2,26,1>]);
> [Degree(x[1]) : x in Factorization(S3)];
[ 8, 3 ]
> CFENew(S3); // 2^18 for deg 7 factor
0.045944
```

It would be more efficient to compute the above information using `OrthogonalSymmetrization`. Note that the `BadPrimes` vararg in the first form referred to the whole L-functions, while when using this second form we center on the degree 7 constituent.

```
> O3 := OrthogonalSymmetrization(L,[3] : BadPrimes:=[<2,18,1>]);
> Degree(O3);
7
> CFENew(O3); // L-coeffs already computed, thanks to SaveEuler
0.00000
```

The $\{2,1\}$-symmetrization of a degree 3 L-function has degree 8, and when the input is orthogonal there is a degree 3 constituent corresponding to the L-function.

```
> LSetPrecision(L,20);
> S21 := Symmetrization(L,[2,1] : BadPrimes:=[<2,20,1>]); // deg 8
> CFENew(S21);
0.00000000000000000000
> LT := Translate(L,2);
> CFENew(S21/LT);
0.00000000000000000000
```

The degree 5 constituent is the orthogonal symmetric square twisted by the determinant.

```
> D := Determinant(L); D;
Translation by 3 of L-series of Kronecker character -4
> O2 := OrthogonalSymmetrization(L,[2] : BadPrimes:=[<2,10,1>]);
> CFENew(O2);
0.00000000000000000000
> TP := TensorProduct(O2,D : BadPrimes:=[<2,12,1>]);
> Q := Translate(S21/LT,2);
> P := PrimesUpTo(100);
> assert &and[EulerFactor(TP,p) eq EulerFactor(Q,p) : p in P];
```

Be careful here! It turns out that `CFENew` works to 12 digits for the quotient of S_{21} with the `Determinant`, but in fact there is no such factorization at the level of L-functions (the issue is likely that the lowest height zero on the critical line for the `Determinant` is rather large, and thus the poles in the L-quotient do not contribute much).

```
> CFENew(S21/LT/D);       // the product of these
2.5742151368679484303E-6
> CFENew(LT/D);           // last two is ~ 10^(-12)
6.3865388810908029459E-5
```

A superior way to detect constituents is through taking a Selberg inner product; here this would just be looking at the average c_p value when twisting by the appropriate Dirichlet character.

The $\{1,1\}$-symmetrization (or alternating square) of a degree 3 L-function is actually a translate of the original L-function tensored with its determinant.

```
> S := Symmetrization(L,[1,1] :  BadPrimes:=[<2,8,1>]);
> DL := TensorProduct(D,L : BadPrimes:=[<2,8,1>]);
> T := Translate(DL,-2);
> assert &and[EulerFactor(T,p)  eq EulerFactor(S,p) : p in P];
```

Example: Symplectic Symmetrizations in Degree 4 (ex-d40acc)

We consider a degree 4 symplectic L-function. Note that when the base L-function has degree 4, one needs to compute the quadratic term of the Euler factor to be able to compute the symmetrizations, and depending on the type of L-function, this can be quite time-consuming. Our first example is identified by the internal Magma machinery as coming from an elliptic curve over a quadratic field, which allows faster computations of the coefficients. We take its alternating square, which yields a degree 6 L-function that has a translated Riemann ζ-function as a factor.

```
> x := PolynomialRing(Integers()).1;
> H := HypergeometricData([4,4],[1,1,2,2]);
> L := LSeries(H,2 : Precision:=10);
> assert IsSymplectic(L); // degree 4 weight 1
> L`parent;
Elliptic Curve defined by y^2 + x*y + $.1*y = x^3 + $.1*x^2 over Number Field
   with defining polynomial x^2 - 1/512 over the Rational Field
> S := Symmetrization(L,[1,1] : BadPrimes:=[<2,16,1-2*x>]);
> Degree(S);
6
> CFENew(S);
0.0000000000
> Factorization(S)[2];
<Translation by 1 of L-series of Riemann zeta function, 1>
```

Alternatively, we can compute this via the symplectic symmetrization, when we don't have to specify the bad prime information corresponding to the ζ-function.

```
> S := SymplecticSymmetrization(L,[1,1] :  BadPrimes:=[<2,16,1>]);
> Degree(S);
5
> CFENew(S);
0.0000000000
```

Another example comes from hypergeometric motives, where here the conductor is sufficiently small so that the slowdown from computing quadratic coefficients is not too problematic.

```
> H := HypergeometricData([4,2,2,2,2],[1,1,1,1,1,1]);
> L := LSeries(H,1 : Precision:=10); // t=1 degeneration
> Conductor(L); // degree 4, conductor 2^8, motivic wt 5
256
> S := SymplecticSymmetrization(L,[1,1] : BadPrimes:=[<2,8,1>]);
> time CFENew(S);
```

(continues on next page)

(continued from previous page)

```
0.0000000000
Time: 2.300
```

An additional source of such L-functions could be Grössencharacters, though these are inductions from $GL(1)$.

```
> K<zeta5> := CyclotomicField(5);
> p5 := Factorization(5*Integers(K))[1][1];
> psi := HeckeCharacterGroup(p5^2).0;
> GR := Grossencharacter(psi,[[3,0],[1,2]]);
> L := LSeries(GR :  Precision:=15);
> S := SymplecticSymmetrization(L,[1,1] : BadPrimes:=[<5,6,1>]);
> CFENew(S); // S is imprimitive
0.000000000000000
> Q := S/Translate(LSeries(KroneckerCharacter(5)),3);
> CFENew(Q);
0.000000000000000
```

Example: Higher Orthogonal Symmetrizations (ex-5e5bc4)

Here are some examples involving the higher orthogonal symmetrizations. Again we note that computing the quadratic term of the underlying L-function can sometimes be quite time-consuming.

```
> _<x> := PolynomialRing(Integers());
> H := HypergeometricData([2,2,2],[1,1,1]);
> L := LSeries(H,2 : BadPrimes:=[<2,9,1>],Precision:=5);
> BP := [<2,18,1-16*x>];
> O4 := OrthogonalSymmetrization(L,[4] : BadPrimes:=BP);
> CFENew(O4); // OSym^4(deg3) // deg 9
0.00000
> H := HypergeometricData([2,2,2,2,2],[1,1,1,4]);
> L := LSeries(H,1); // degree 4, weight 4, cond 2^7
> BP := [<2,15,1-16*x>];
> O2 := OrthogonalSymmetrization(L,[2] : BadPrimes:=BP);
> LSetPrecision(O2,5); // only 450 terms required, but slow!
> time CFENew(O2); // OSym^2(deg4) // deg 9
0.00000
Time: 21.420
```

With this latter selection of L, one can verify (using about 1000 terms to get 3 digits, taking 5 minutes) that the orthogonal symmetrization $L^{[2,2]}$ has conductor 2^{21} and trivial Euler factor at 2. This is fully in line with a computation from (wild) Swan slopes.

Example: Imprimitive Orthogonal Symmetrization (ex-8645f8)

We are able to give a computed example of an $L^{[2,2]}$ orthogonal symmetrization of a degree 4 L-function. It turns out to be imprimitive (in fact, with a factor of ζ, but factors even further it seems), though L itself is primitive.

```
> _<x> := PolynomialRing(Integers());
> H := HypergeometricData([2,2,2,2,2],[1,1,1,1,1]);
> L := LSeries(H,1 : SaveEuler:=10^4); // degree 4, weight 4, cond 2^8
> LSetPrecision(L,5);
> BP := [<2,12,(1-2^8*x)^2*(1+2^8*x)^2>]; // Swan conductor 16-10 or 6
> O22 := OrthogonalSymmetrization(L,[2,2] : BadPrimes:=BP,PoleOrder:=1);
> time CFENew(O22); // OSym^[2,2](deg4) // deg 10
1.5259E-5
Time: 6.910
```

(continues on next page)

(continued from previous page)

```
> BP := [<2,12,(1+2^8*x^2)*(1+2^4*x)>]; // Swan conductor 15-9 or 6
> O2 := OrthogonalSymmetrization(L,[2] : BadPrimes:=BP);
> time CFENew(O2); // OSym^2(deg4) // deg 9
0.00000
Time: 0.220
```

It is difficult to give an example of a *primitive* L-function of degree 5 and nonzero weight for which the $\{1,1\}$-symmetrization is computable reasonably fast; one example here is the $t = 1$ hypergeometric motive from $[4,4,4]$ and $[1,2,2,2,2,2]$, whose $L^{\{1,1\}}$ has conductor 2^{24} and trivial Euler factor.

Having no better example, we rely on the 4th symmetric power of an elliptic curve L-function, which is primitive though not wholly generic. Indeed, the resulting $\{1,1\}$-symmetrization is itself imprimitive. Importantly, the coefficients can be computed fast.

```
> _<x> := PolynomialRing(Integers());
> E := EllipticCurve("128a");
> L := LSeries(E);
> S4 := SymmetricPower(L,4 : Precision:=5);
> Degree(S4), Conductor(S4); // 2^10, slopes 5/3,5/3,5/3,0,0
5 1024
> BP :=[<2,24,1+16*x>]; // Swan conductor (5/3)*9
> S11 := Symmetrization(S4,[1,1] : BadPrimes:=BP);
> time CFENew(S11); // Sym^[1,1](deg5) // deg 10
0.00000
Time: 1.910
```

139.10 Advanced Examples

139.10.1 Handmade L-series of an Elliptic Curve

Example: Elliptic Curve L-Series from Scratch (ex-84defa)

This is an example of how the general `LSeries` function can be used to define the L-series for elliptic curves. This is essentially what `LSeries(E)` does:

```
> E := EllipticCurve([1,2,3,4,5]);
> N := Conductor(E);N;
10351
> P<x> := PolynomialRing(Integers());
```

The easiest way to define the coefficients is to provide the local factors.

```
> cf := func< p,d|1 - TraceOfFrobenius(E,GF(p,1))*x
>                  +(N mod p ne 0 select p else 0)*x^2 >;
> L := LSeries(2,[0,1],N,cf : Parent:=E, Sign:=RootNumber(E));
```

Compare this with the built-in function `LSeries(E)`

```
> Evaluate(L,2);
0.97743186689450050867903912764 7
> Evaluate(LSeries(E),2);
0.97743186689450050867903912764 7
```

139.10.2 Self-made Dedekind Zeta Function

Example: Dedekind Zeta Function from Scratch (ex-aa77ea)

This is an example of how the general `LSeries` function may be used to define the Dedekind zeta function of a number field K. This is essentially what the function `LSeries(K)` does:

```
> function DedekindZeta(K)
>   M := MaximalOrder(K);
>   r1,r2 := Signature(K);
>   gamma := [0: k in [1..r1+r2]] cat [1: k in [1..r2]];
>   disc := Abs(Discriminant(M));
>   P<x> := PolynomialRing(Integers());
```

The coefficients are defined by means of local factors; for a prime p we take the product of $1 - x^{f_k}$ where the f_i are the residue degrees of primes above p. Note that this local factor has (maximal) degree $[K : \mathbb{Q}]$ if and only if p is unramified or, equivalently, if and only if p does not divide the discriminant of K which is (up to sign) the conductor of our zeta-function.

```
>   cf := func<p,d|&*[1-x^Degree(k[1]): k in Decomposition(M,p)]>;
```

Finally, the Dedekind zeta function has a pole at $s = 1$ and we need its residue (or, rather, the residue of $\zeta^*(s)$) which we compute using the class number formula.

```
>   h := #ClassGroup(M);
>   reg := Regulator(K);
>   mu := #TorsionSubgroup(UnitGroup(M));
>   return LSeries(1, gamma, disc, cf: Parent:=K, Sign:=1, Poles:=[1],
>     Residues := [-2^(r1+r2)*Pi(RealField())^(r2/2)*reg*h/mu]);
> end function;
> Z := DedekindZeta(CyclotomicField(5)); Z;
L-series of Cyclotomic Field of order 5 and degree 4
> Evaluate(Z,1);
Runtime error: L*(s) has a pole at s = 1
> L := Z/RiemannZeta();
> Evaluate(L,1);
0.33983727824052353546427878 1159
```

139.10.3 Handmade L-series of a Hyperelliptic Curve

Example: Genus 2 L-Series from Scratch (ex-d8bc35)

We construct an L-series of a hyperelliptic curve of genus 2.

```
> P<x> := PolynomialRing(Integers());
> C := HyperellipticCurve([x^5+x^4,x^3+x+1]); C;
Hyperelliptic Curve defined by y^2 + (x^3 + x + 1)*y = x^5 + x^4
   over Rational Field
```

There is an L-series attached to $\mathrm{H}^1(C)$ or, equivalently, the H^1 of the Jacobian J of C. To define this L-series, we need its local factors which, for primes p of good reduction of C, are given by `EulerFactor(J,GF(p,1))`. Let us look at the primes of bad reduction:

```
> Abs(Discriminant(C));
169
> Factorization(Integers()!$1);
[ <13, 2> ]
```

There is one prime $p = 13$ where the curve has bad reduction. There the fibre is a singular curve whose normalization is elliptic.

```
> A<X,Y> := AffineSpace(GF(13,1),2);
> C13 := Curve(A,Equation(AffinePatch(C,1)));
> GeometricGenus(C), GeometricGenus(C13);
2 1
> SingularPoints(C13);
{@ (9, 1) @}
> p := $1[1]; IsNode(p), IsCusp(p);
false true
> C13A := Translation(C13,p)(C13);
> C13B := Blowup(C13A); C13B;
Curve over GF(13) defined by
X^3 + 12*X^2*Y + 7*X^2 + 12*X*Y + 12*Y^2 + 3*Y + 1
> E := EllipticCurve(ProjectiveClosure(C13B));
```

The local factor of C at $p = 13$ is given by the Euler factor of this elliptic curve

```
> EulerFactor(E);
13*x^2 + 5*x + 1
```

The conductor of $L(C, s)$ is 13^2, the same as the discriminant in this case. So, we define $L(C, s)$:

```
> J := Jacobian(C);
> loc := func<p,d|p eq 13 select EulerFactor(E) else EulerFactor(J, GF(p,1))>;
> L := LSeries(2,[0,0,1,1],13^2,loc);
```

Now we check the functional equation and compute a few L-values. (Most of the execution time will be spent generating coefficients, so the first call takes some time. After that, the computations are reasonably fast.)

```
> CFENew(L);
0.00000000000000000000000000000000
> Evaluate(L,1);
0.09049039083242962911358975725 81
> Evaluate(L,2);
0.36428634294436406815429145 0139
```

This is roughly what the built-in function `LSeries(C)` does:

```
> LC:=LSeries(C);
> CFENew(LC);
0.00000000000000000000000000000000
> Evaluate(LC,1);
0.09049039083242962911358975725 81
> Evaluate(LC,2);
0.36428634294436406815429145 0139
```

139.10.4 Experimental Mathematics for Small Conductor

Example: Recovering a Conductor 11 L-Series (ex-c04c9d)

This example shows how one may construct the first 20 coefficients of the L-series of an elliptic curve of conductor 11 without knowing anything about either the curve or modular form theory.

The point is that for an L-series $L(s) = \sum a_n/n^s$, the associated theta function (used in `CFENew` or `CheckFunctionalEquation`) is a series with a_n as coefficients and terms that decrease very rapidly with n. This means that if we truncate the series to, for instance, just $a_1 + a_2/2^s$ and call `LSeries` with the same parameters (weight, sign, etc.) as those for $L(s)$ but just $[a_1, a_2, 0, 0, 0, 0, ...]$ as the coefficient vector, then `CheckFunctionalEquation` will return a number reasonably close to 0, because the coefficients a_n for $n > 2$ only make a small contribution.

With this in mind, we create an L-series that looks like that of an elliptic curve with conductor 11 and sign 1, that is, we take weight = 2, conductor = 1, gamma = [0,1], sign = 1, and no poles:

```
> L := LSeries(2, [0,1], 11, 0 : Sign:=1);
```

Then taking $a_1 = 1$, and recalling the requirement that all the coefficients must be weakly multiplicative, we try various a_2 to see which of the truncated L-series $1 + a_2/2^s$ is closest to satisfying a functional equation. Using the knowledge that a_2 (and every other a_n) is an integer together with the Hasse-Weil bound, $|a_p| < 2\sqrt{p}$, we find just 5 choices:

```
> for a_2 := -2 to 2 do
>   LSetCoefficients(L,[1,a_2]);
>   print a_2, CFENew(L);
> end for;
-2 0.015003701009749031717353004899 6
-1 0.101579424986956534846398098015
0 0.157218613580238064317855126848
1 0.195989746558550757309403554281
2 0.224553905951474457074100971432
```

It seems that $a_2 = -2$ is the best choice, so we set it. Note that this also determines a_4, a_8, etc. We next proceed to find a_3 in the same way. It might come as a surprise, but in this way we can find the first 30 or so coefficients correctly. (Actually, by using careful bounds, it is even possible to prove that these are indeed uniquely determined.) Here is code that finds $a_1, ..., a_{20}$:

```
> N := LCfRequired(L); N;
48
> V := [0 : k in [1..N] ];   // keep a_p in here
> P<x> := PolynomialRing(Integers());
> function Try(V,p,a_p)  // set V[p]=a_p and try functional equation
>   V[p] := a_p;
>   LSetCoefficients(L, func<p,d | 1-V[p]*x+(p eq 11 select 0 else p)*x^2 >);
>   return Abs(CFENew(L));
> end function;
> for p in PrimesUpTo(20) do  // try a_p in Hasse-Weil range and find best one
>   hasse := Floor(2*Sqrt(p));
>   _,V[p] := Min([Try(V,p, a_p): a_p in [-hasse..hasse]]);
>   V[p] -:= hasse+1;
> end for;
> LSetCoefficients(L, func<p,d | 1-V[p]*x+(p eq 11 select 0 else p)*x^2 >);
```

We list the coefficients that we found and apply `CheckFunctionalEquation` to them:

```
> LGetCoefficients(L,20);
[* 1, -2, -1, 2, 1, 2, -2, 0, -2, -2, 1, -2, 4, 4, -1, -4, -2, 4, 0, 2 *]
> CFENew(L);
5.624167335010766890508460613 01E-17
```

Compare this with the actual truth:

```
> qExpansion(Newforms("11A")[1],21);
q - 2*q^2 - q^3 + 2*q^4 + q^5 + 2*q^6 - 2*q^7 - 2*q^9 - 2*q^10 + q^11 - 2*q^12
   + 4*q^13 + 4*q^14 - q^15 - 4*q^16 - 2*q^17 + 4*q^18 + 2*q^20 + O(q^21)
```

Such methods have been used by Stark in his experiments with L-functions of number fields, and by Mestre to find restrictions on possible conductors of elliptic curves. In fact, by modifying our example (changing 11 to $1 \ldots 10$ and sign = 1 to sign = ± 1) one can show that for $N < 11$ there are no modular elliptic curves of conductor N.

139.10.5 Tensor Product of L-series Coming from l-adic Representations

Example: Tensor Product of l-Adic L-Series (ex-3815af)

This is an example of using the tensor product for L-functions coming from l-adic representations. This is what `LSeries(E,K)` uses to construct L-series of elliptic curves over number fields.

```
> E := EllipticCurve([ 0, 0, 0, 0, 1]); // Mordell curve
> P<x> := PolynomialRing(Integers());
> K := NumberField(x^3-2);
> LE := LSeries(E);
> LK := LSeries(K);
```

We have now two L-functions, one associated to an elliptic curve and one to a number field. They both come from l-adic representations, and we can try to construct their tensor product. Actually, the function `TensorProduct` requires us to specify exponents of the conductor and the bad local factors of the tensor product representation, but we can try and let Magma do it, hoping that nothing unusual happens. We take the tensor product of the two L-series and divide it by $L(E, s)$ to speed up the computations. Remember that $L(K, s)$ has a copy of the Riemann zeta function as a factor.

```
> L := TensorProduct(LE, LK, []) / LE;
WARNING: 2 is ramified in each part of tensor prod
WARNING: 3 is ramified in each part of tensor prod
> CFENew(L);
0.023406257100607514301535636259
```

The resulting L-function does not satisfy the required functional equation, so something unusual does happen at a prime where both E and K have bad reduction, which in this case must be either $p = 2$ or $p = 3$.

Let us check $p = 2$. We change the base field of E to K and look at its reduction at the unique prime above 2:

```
> EK := BaseChange(E,K);
> p := Decomposition(MaximalOrder(K),2)[1,1];
> LocalInformation(E,2);
<2, 4, 2, 3, IV, true>
> loc, model:=LocalInformation(EK,p);loc,model;
<Prime Ideal
Two element generators:
[2, 0, 0]
[0, 1, 0], 0, 0, 1, I0, true>
Elliptic Curve defined by y^2 + y = x^3 over K
```

We see that E has acquired good reduction at $p \mid 2$. This means that the inertia invariants at 2 of the l-adic representation associated to L, namely $\rho_E \otimes (\rho_K - \rho_{\mathbb{Q}})$ are *larger* than the tensor product of the corresponding inertia invariants, which is zero. Thus the local factor of L at 2 is not $F_p(x) = 1$ (which is what `TensorProduct` assumed), but a polynomial of higher degree. In fact, it is the characteristic polynomial of Frobenius of the reduced curve E/K mod p, so $F_p(x) = 1 - a_2x + 2x^2$ with a_2 given by

```
> TraceOfFrobenius(Reduction(model, p));
0
```

This is now the correct L-function (at $p = 3$ the default procedure works) and the functional equation is satisfied

```
> L := TensorProduct(LE,LK,[<2,4,1+2*x^2>])/LE;
WARNING: 3 is ramified in each part of tensor prod
```

(continues on next page)

(continued from previous page)

```
> CFENew(L);
7.88860905221011805411728565283E-31
```

In fact, our L-series is the same as that constructed by `LSeries(E,K)/LE`.

139.10.6 Non-abelian Twist of an Elliptic Curve

Example: BSD in a Non-Abelian Extension (ex-3cebc3)

In this example we illustrate how to use `LSeries(E,K)` to experiment with the Birch-Swinnerton-Dyer conjecture for elliptic curves in non-abelian extensions of $\mathbb{Q}$.

```
> E := EllipticCurve([0, 0, 0, 0, 1]);
> P<x> := PolynomialRing(Integers());
> K := NumberField(x^3-2);
> L := LSeries(E,K) / LSeries(E);
> lval := Evaluate(L, 1); lval;
0.655371552164974435712378842491
```

The series L is the L-function of non-abelian twist of E and it appears that $L(1)$ is non-zero. In other words, the analytic rank of $L(E/K, s)$ at $s = 1$ is the same as that of $L(E/\mathbb{Q}, s)$. Tate's version of the Birch-Swinnerton-Dyer conjecture then predicts that the Mordell–Weil rank of E/K is the same as the rank over $\mathbb{Q}$, so the elliptic curve has not acquired new independent points. This is something that can be confirmed by a Selmer group computation:

```
> EK := BaseChange(E,K);
> twoE := MultiplicationByMMap(E, 2);
> #SelmerGroup(twoE);
2
> twoEK := MultiplicationByMMap(EK, 2);
> #SelmerGroup(twoEK);
2
```

Indeed, the 2-Selmer rank is 1 in both cases, as expected. Also, the second part of the Birch-Swinnerton-Dyer conjecture predicts that the following quotient should be rational:

```
> p1, p2 := Explode(Periods(E));
> lval*Sqrt(Abs(Discriminant(K))) / (p1*Im(p2));
1.33333333333333333333333333334
```

Finally, the same L-series can be defined via the Artin representation machinery:

```
> triv,sign,rho:=Explode(ArtinRepresentations(K));
> L:=LSeries(E,rho);
> CentralValue(L);
0.655371552164974435712378842491
```

139.10.7 Unitary Twist with Central Vanishing

Example: Rohrlich's Central Vanishing Twist (ex-a7958b)

In this example we go through Exercise 5.5 of Rohrlich's Park City notes [Roh11], illustrating an example of a nonorthogonal L-function that has vanishing central value.

We will be interested in the twist of the elliptic curve 49a by an element d in the quadratic field $\mathbf{Q}(\sqrt{-7})$. By construction, this twist will have a point on it. Namely, we write 49a as $y^2 = x^3 - 35x - 98$, and take d as the evaluation of the right-side cubic polynomial at $x = (1 + \sqrt{-7})/2$.

```
> K<s7> := QuadraticField(-7);
> f := func<x|x^3-35*x-98>;
> d := f((1+s7)/2); // -118 - 18*s7, norm 2^6 * 11 * 23
```

We then take the quadratic Hecke character corresponding to this element, and multiply it by the canonical Grössencharacter for 49a. The central value of the resulting L-series will then vanish, even though the root number is non-orthogonal.

```
> eta := QuadraticCharacter(d);
> xi := eta*Grossencharacter(EllipticCurve("49a"));
> L := LSeries(xi); // conductor 7^2 * 11 * 23
> CentralValue(L); Sign(L);
-5.525476622753405916701874254 83E-30
0.943041920192897214648941373358 - 0.33267391956523024173849624 9575*i
```

We can also work with the associated elliptic curve, using the point that we already know on it.

```
> E := EllipticCurve([K|-35,-98]); // a model for 49a
> Q := QuadraticTwist(E,d);
> pt := Points(Q,(1+s7)*d/2)[1];
> Height(pt); // nonzero
1.28216334015607284474482974
```

While it is easy to get two independent points on the twisted curve (indeed, a second point could be obtained from this first via the isogeny of multiplication by $(1+\sqrt{-7})/2$), it is actually non-trivial to prove a sufficient saturation bound to show this is really the full Mordell–Weil group.

```
> b,G,m := PseudoMordellWeilGroup(Q : SearchBound:=10);
> PTS := [m(G.3),m(G.4)]; // G.1 and G.2 are 2-torsion
> [Height(P) : P in PTS]; // differ by multiple of 2
[ 2.56432668031214568948965948, 1.28216334015607284474482974 ]
```

There is no Gross-Zagier theorem known in this case, but one can numerically verify the relevant Birch–Swinnerton-Dyer analogue.

```
> fd := Coefficient(LTaylor(L,1,2),1); // L'(1)
> OM := Periods(Q);
> h := Height(pt : Precision:=10);
> [fd/p/h : p in &cat OM];
[ 7.041379671 + 19.96043517*i, 29.92586360 + 0.6653478392*i,
  1.34017E-28 + 21.16601049*i, 28.00000000 - 10.58300524*i ]
> [Norm(x) : x in $1];
[ 448.0000000, 895.9999999, 448.0000000, 896.0000001 ]
```

Other examples of unitary L-functions with vanishing central value can come from modular forms (this was first noted/computed by W. A. Stein). We take the quadratic character of conductor 61 and modulus 122, and then compute the desired modular form and its associated modular symbols. This calculation already (rigorously) gives that the central value vanishes, and we can check this numerically also (and that the sign is nonorthogonal).

```
> chi := DirichletGroup(122).1;
> f := Newforms(CuspForms(chi))[1][1]; // 1st isogeny class
> LRatio(ModularSymbols(f),1);
0
> L := LSeries(f : Embedding:=func<x|Conjugates(x)[1]>);
> CentralValue(L), Sign(L);
0.00000000000000000000000000000000
-0.76822127959737584204554649 8599 - 0.64018439966447986837128874 8833*i
```

The above is the smallest example with quadratic character. Already at level 61 (likely a numerical coincidence) there is an example with sextic character.

```
> chi := DirichletGroup(61,CyclotomicField(6)).1;
> f := Newforms(CuspForms(chi))[1][1]; // 1st isogeny class
> LRatio(ModularSymbols(f),1);
0
> L := LSeries(f : Embedding:=func<x|Conjugates(x)[1]>);
> CentralValue(L), Sign(L);
0.00000000000000000000000000000
-0.85515172874206975518053864819 9 + 0.51837777810150151976680679199 1*i
```

139.11 Bibliography

- [JKS94] Uwe Jannsen, Steven Kleiman, and Jean-Pierre Serre, editors. *Motives*, volume 55 of Proceedings of Symposia in Pure Mathematics, Providence, RI, 1994. American Mathematical Society.
- [Arm71] J. V. Armitage. Zeta functions with a zero at $s = 1/2$. *Invent. Math.*, 15(3):199–205, 1971.
- [BST97] J. Buhler, C. Schoen, and J. Top. Cycles, L-functions and triple products of elliptic curves. *J. Reine. Angew. Math.*, 492:93–133, 1997.
- [Coh00] Henri Cohen. *Advanced Topics in Computational Number Theory*. Springer, Berlin–Heidelberg–New York, 2000.
- [CS01b] C. Consani and J. Scholten. Arithmetic on a quintic threefold. *Internat. J. Math*, 12/8:943–972, 2001.
- [Del79] P. Deligne. Valeurs de fonctions L et périodes d'intégrales. *Proc. Symposia Pure Math.*, 33/2:313–346, 1979.
- [DPS12] L. Dieulefait, A. Pacetti, and M. Schütt. Modularity of the Consani-Scholten quintic. *Documenta Mathematica*, 17:953–987, 2012.
- [Dok02] Tim Dokchitser. ComputeL, pari package to compute motivic L-functions. http://www.maths.bris.ac.uk/∼matyd/computel/, 2002.
- [Dok04] Tim Dokchitser. Computing special values of motivic L-functions. *Experimental Mathematics*, 13(2):137–149, 2004.
- [DMW09] N. Dummigan, P. Martin, and M. Watkins. Euler factors and local root numbers for symmetric powers of elliptic curves. *Pure and Appl. Math. Qu.*, 5(4):1311–1341, 2009.
- [Fri76] J. B. Friedlander. On the class numbers of certain quadratic extensions. *Acta Arith.*, 28(4):391–393, 1975/76.
- [Lav67] A. F. Lavrik. An approximate functional equation for the Hecke zeta-function of an imaginary quadratic field. *Mat. Zametki*, 2:475–482, 1967.
- [MW06] P. Martin and M. Watkins. Symmetric powers of elliptic curve L-functions. In F. Hess, S. Pauli, and M. Pohst, editors, *Algorithmic Number Theory*, volume 4076 of Lecture Notes in Computer Science, 377–392. Springer, 2006.
- [Roh11] D. E. Rohrlich. Root Numbers. In K. Rubin C. Popescu and A. Silverberg, editors, *Arithmetic of L-functions*, volume 18 of IAS / Park City Math. Ser., pages 353–448. Amer. Math. Soc., 2011.
- [Ser71] J.-P. Serre. Conducteurs d'Artin des caractères réels. *Invent. Math.*, 14(3):173–183, 1971.
- [Ser65] Jean-Pierre Serre. Zeta and L functions. In *Arithmetical Algebraic Geometry (Proc. Conf. Purdue Univ., 1963)*, pages 82–92. Harper & Row, New York, 1965.

- [Sha95] I. R. Shafarevich, editor. *Number theory I.* Volume 49 of Encyclopaedia of Mathematical Sciences. Springer-Verlag, Berlin, 1995. Fundamental problems, ideas and theories, A translation of Number theory 1 (Russian), Akad. Nauk SSSR, Vsesoyuz. Inst. Nauchn. i Tekhn. Inform., Moscow, 1990, Translation edited by A. N. Parshin and I. R. Shafarevich.
- [Tol97] Emmanuel Tollis. Zeros of Dedekind zeta functions in the critical strip. *Math. Comp.*, 66(219):1295–1321, 1997.
- [vGvS93] B. van Geemen and D. van Straten. The cusp forms of weight 3 on $\Gamma_2(2,4,8)$. *Math. Comp.*, 61(204):849–872, 1993.

References

[JKS94] Uwe Jannsen, Steven Kleiman, and Jean-Pierre Serre, editors. *Motives*, volume 55 of Proceedings of Symposia in Pure Mathematics, Providence, RI, 1994. American Mathematical Society.

[AG12] Gert Almkvist and Jesús Guillera. Ramanujan-like series for $1/\pi^2$ and String Theory. *Experimental Math.*, 21:223–234, 2012. URL: http://dx.doi.org/10.1080/10586458.2012.656059.

[AKM+01] Sang Yook An, Seog Young Kim, David C. Marshall, Susan H. Marshall, William G. McCallum, and Alexander R. Perlis. Jacobians of genus one curves. *J. Number Theory*, 90(2):304–315, 2001.

[Arm71] J. V. Armitage. Zeta functions with a zero at $s = 1/2$. *Invent. Math.*, 15(3):199–205, 1971.

[ARVT05] Michael Artin, Fernando Rodriguez-Villegas, and John Tate. On the Jacobians of plane cubics. *Adv. Math.*, 198(1):366–382, 2005.

[Bas15] Romain Basson. *Arithmétique des espaces de modules des courbes hyperelliptiques de genre 3 en caractéristique positive*. PhD thesis, Université de Rennes 1, Rennes, 2015.

[Bla05] I. Blake, G. Serrousi and N. Smart, editor. *Advances in Elliptic Curve Cryptography*. Volume 317 of LMS LNS. Cambridge University Press, Cambridge, 2005.

[BC84] A. Bremner and J. W. S. Cassels. On the equation $Y^2 = X(X^2 + p)$. *Math. Comp.*, 42(165):257–264, 1984.

[Bru02] N. R. Bruin. *Chabauty methods and covering techniques applied to generalized Fermat equations*. Volume 133 of CWI Tract. Stichting Mathematisch Centrum Centrum voor Wiskunde en Informatica, Amsterdam, 2002. ISBN 90-6196-508-X. Dissertation, University of Leiden, Leiden, 1999.

[Bru] N. Bruin. "Some ternary Diophantine equations of signature $(n, n, 2)$."

[Bru03] Nils Bruin. Chabauty methods using elliptic curves. *J. reine angew. Math.*, 562:27–49, 2003.

[BD09] Nils Bruin and Kevin Doerksen. The arithmetic of genus two curves with (4,4)-split Jacobians. ArXiv preprint. http://arxiv.org/abs/0902.3480, 2009.

[BS09] Nils Bruin and Michael Stoll. Two-cover descent on hyperelliptic curves. *Math. Comp.*, 78:2347–2370, 2009.

[BS10] Nils Bruin and Michael Stoll. The Mordell-Weil sieve: Proving non-existence of rational points on curves. *LMS J. Comput. Math.*, 13:272–306, 2010.

[BST97] J. Buhler, C. Schoen, and J. Top. Cycles, L-functions and triple products of elliptic curves. *J. Reine. Angew. Math.*, 492:93–133, 1997.

[BJ14] Laurent Busé and Jean-Pierre Jouanolou. On the discriminant scheme of homogeneous polynomials. *Math. Comput. Sci.*, 8(2):175–234, 2014.

[Car03a] G. Cardona. On the number of curves of genus 2 over a finite field. *Finite Fields and Their Applications*, 9(4):505–526, 2003.

[CN07] G. Cardona and E. Nart. Zeta function and cryptographic exponent of supersingular curves of genus 2. In *Pairing-based cryptography—Pairing 2007*, volume 4575 of Lecture Notes in Comput. Sci., pages 132–151. Springer, Berlin, 2007.

[CNP05a] G. Cardona, E. Nart, and J. Pujolas. Curves of genus two over fields of even characteristic. *Mathematische Zeitschrift*, 250:177–201, 2005.

[CQ05b] Gabriel Cardona and Jordi Quer. Field of moduli and field of definition for curves of genus 2. *Lecture Notes Ser. Comput.*, 13:71–83, 2005.

[Cas66] J. W. S. Cassels. Diophantine equations with special reference to elliptic curves. *J. London Math. Soc.*, 41:150–158, 1966.

[Cas78] J. W. S. Cassels. *Rational Quadratic Forms*. Academic Press, London–New York–San Francisco, 1978.

[Cas91] J. W. S. Cassels. *Lectures on elliptic curves*. Volume 24 of London Mathematical Society Student Texts. Cambridge University Press, Cambridge, 1991. ISBN 0-521-41517-9; 0-521-42530-1.

[CF96] J.W.S. Cassels and E.V. Flynn. *Prolegomena to a Middlebrow Arithmetic of Curves of Genus 2*. Cambridge University Press, Cambridge, 1996.

[Coh93] Henri Cohen. *A Course in Computational Algebraic Number Theory*. Volume 138 of Graduate Texts in Mathematics. Springer, Berlin–Heidelberg–New York, 1993.

[Coh00] Henri Cohen. *Advanced Topics in Computational Number Theory*. Springer, Berlin–Heidelberg–New York, 2000.

[Con99] I. Connell. *Elliptic Curve Handbook*. McGill University, 1999.

[CS01b] C. Consani and J. Scholten. Arithmetic on a quintic threefold. *Internat. J. Math*, 12/8:943–972, 2001.

[Cre97] J. E. Cremona. *Algorithms for modular elliptic curves*. Cambridge University Press, Cambridge, second edition, 1997. ISBN 0-521-59820-6.

[CFO+08] J. E. Cremona, T. A. Fisher, C. O'Neil, D. Simon, and M. Stoll. Explicit n-descent on elliptic curves. I. Algebra. *J. Reine Angew. Math.*, 615:121–155, 2008.

[CR03] J. E. Cremona and D. Rusin. Efficient solution of rational conics. *Mathematics of Computation*, 72(243):1417–1441, 2003.

[CR06] J. E. Cremona and D. Rusin. Solving conics over function fields. *Journal de Theorie des Nombres de Bordeaux*, 18:595–606, 2006.

[CFO+] J.E. Cremona, T.A Fisher, C. O'Neil, D. Simon, and M Stoll. Explicit n-Descent On Elliptic Curves, III. Algorithms. ArXiv preprint. http://arxiv.org/abs/1107.3516.

[CFO+09] J.E. Cremona, T.A Fisher, C. O'Neil, D. Simon, and M Stoll. Explicit n-Descent On Elliptic Curves, II. Geometry. *J. reine angew. Math.*, 632:63–84, 2009.

[CFS10] J.E. Cremona, T.A Fisher, and M Stoll. Minimisation and reduction of 2-, 3- and 4-coverings of elliptic curves. *Algebra & Number Theory*, 4(6):763–820, 2010.

[Cre99] John Cremona. Reduction of binary cubic and quartic forms. *LMS JCM*, 2:62–92, 1999.

[Cre01] John Cremona. Classical invariants and 2-descent on elliptic curves. *J. Symbolic Comp.*, 31:71–87, 2001.

[CM00b] John E. Cremona and Barry Mazur. Visualizing elements in the Shafarevich-Tate group. *Experiment. Math.*, 9(1):13–28, 2000.

[CM12] B. Creutz and R.L. Miller. Second isogeny descents and the Birch and Swinnerton-Dyer conjectural formula. *J.Algebra*, 372:673–701, 2012.

[Cre10] Brendan Creutz. *Explicit second p-descents on elliptic curves*. PhD Thesis, Jacobs University Bremen, 2010.

[Cre12] Brendan Creutz. Explicit descent in the Picard group of a cyclic cover of the projective line. In Everett Howe and Kiran Kedlaya, editors, *ANTS X: Proceedings of the Tenth Algorithmic Number Theory Symposium*, volume 1 of OBS. Mathematics Sciences Publishers, 2012.

[Del79] P. Deligne. Valeurs de fonctions L et périodes d'intégrales. *Proc. Symposia Pure Math.*, 33/2:313–346, 1979.

[DPS12] L. Dieulefait, A. Pacetti, and M. Schütt. Modularity of the Consani-Scholten quintic. *Documenta Mathematica*, 17:953–987, 2012.

[Dix87] J. Dixmier. On the projective invariants of quartic plane curves. *Adv. in Math.*, 64:279–304, 1987.

[Dok02] Tim Dokchitser. ComputeL, pari package to compute motivic L-functions. http://www.maths.bris.ac.uk/∼matyd/computel/, 2002.

[Dok04] Tim Dokchitser. Computing special values of motivic L-functions. *Experimental Mathematics*, 13(2):137–149, 2004.

[DMW09] N. Dummigan, P. Martin, and M. Watkins. Euler factors and local root numbers for symmetric powers of elliptic curves. *Pure and Appl. Math. Qu.*, 5(4):1311–1341, 2009.

[Elk] N. Elkies. "Rational Points Near Curves and Small Nonzero $|x^3 - y^2|$ via Lattice Reduction." 33–63.

[Els15] Andreas-Stephan Elsenhans. Explicit computations of invariants of plane quartic curves. *J. Symbolic Comput.*, 68(part 2):109–115, 2015.

[Fis08a] T.A Fisher. The invariants of a genus one curve. *Proc. Lond. Math. Soc.*, 97(3):753–782, 2008.

[Fis06a] T.A. Fisher. Testing equivalence of ternary cubics. In S. Pauli F. Hess and M.Pohst, editors, *ANTS VII*, volume 4076 of LNCS, 333–345. Springer-Verlag, 2006.

[Fisb] (no details available)

[Fis00] Tom Fisher. *On 5 and 7 descents for elliptic curves*. PhD thesis, University of Cambridge, 2000.

[Fis01] Tom Fisher. Some examples of 5 and 7 descent for elliptic curves over Q. *J. Eur. Math. Soc.*, 3(Issue 2):169–201, 2001.

[Fis06b] Tom Fisher. The Hessian of a genus one curve. preprint arXiv:math/0610403v2 [math.NT], 2006.

[Fis08b] Tom Fisher. Finding rational points on elliptic curves using 6-descent and 12-descent. *J. Algebra*, 320(2):853–884, 2008.

[Fri76] J. B. Friedlander. On the class numbers of certain quadratic extensions. *Acta Arith.*, 28(4):391–393, 1975/76.

[Gau02] P. Gaudry. A Comparison and a Combination of SST and AGM Algorithms for Counting Points on Elliptic Curves in Characteristic 2. In Y. Zheng, editor, *Advances in Cryptology—AsiaCrypt 2002*, volume 2501 of LNCS, 311–327. Springer-Verlag, 2002. Proc. 8th International Conference on the Theory and Applications of Cryptology and Information Security, Dec. 1–5 2002, Queenstown, New Zealand.

[GH] P. Gaudry, R. Harley. "Counting Points on Hyperelliptic Curves over Finite Fields." 313–332.

[GK06] M. Girard and D. R. Kohel. Classification of genus 3 curves in special strata of the moduli space. In Hess, F. (ed.) et al., Algorithmic number theory. 7th international symposium, ANTS-VII, Berlin, Germany, July 23–28, 2006. Proceedings. Berlin: Springer. Lecture Notes in Computer Science 4076, 346-360 (2006)., 2006.

[Got59] E. Gottschling. Explizite Bestimmung der Randflächen des Fundamentalbereiches der Modulgruppe zweiten Grades. *Math. Annalen*, 138:103–124, 1959.

[GZ86] Gross and Zagier. Heegner Points and Derivatives of L-series. *Invent. Math.*, 84:225–320, 1986.

[Gui12] 2012.

[Hara] R. Harley. Web posting. Under November 2002 entry at http://listserv.nodak.edu/archives/nmbrthry.html.

[Har08] D. Harvey. Efficient computation of p-adic heights. *LMS J. Comput. Math.*, 11:40–59, 2008.

[Hol06] David Holmes. Canonical heights on hyperelliptic curves. Preprint http://arxiv.org/abs/1004.4503, 2006.

[Hub06] Hendrik Hubrechts. Point counting in families of hyperelliptic curves. Preprint http://arxiv.org/abs/math.NT/0601438, 2006.

[Hub07] Hendrik Hubrechts. Quasi-quadratic elliptic curve point counting using rigid cohomology. *to appear in the Journal of Symbolic Computation*, 2007. http://wis.kuleuven.be/algebra/hubrechts/.

[Igu60a] J. Igusa. "Arithmetic variety of moduli for genus two." 72, 612–649. 1960.

[Kat90] N. M. Katz. *Exponential Sums and Differential Equations*. Volume 124. Annals of Math. Studies., 1990.

[Kat96] N. M. Katz. *Rigid Local Systems*. Volume 139. Annals of Math. Studies., 1996.

[Ked01] Kiran S. Kedlaya. Counting Points on Hyperelliptic Curves using Monsky-Washnitzer Cohomology. *J. Ramanujan Math. Soc.*, 16:323 – 338, 2001.

[Klo07] Remke Kloosterman. Elliptic K3 surfaces with geometric Mordell-Weil rank 15. *Canadian Mathematical Bulletin*, 50(2):215–226, 2007.

[Koh03] David R. Kohel. The AGM-$X_0(N)$ Heegner Point Lifting Algorithm and Elliptic Curve Point Counting. In *Advances in Cryptology—AsiaCrypt 2003*, number 2894 in LNCS. Berlin, 2003. Springer.

[Kra77] K. Kramer. Two-Descent for Elliptic Curves in Characteristic Two. *Trans. Amer. Math. Soc.*, 232:279–295, 1977.

[Kra81] K. Kramer. Arithmetic of elliptic curves upon quadratic extension. *Trans. Amer. Math. Soc.*, 264(1):121–135, 1981.

[KLL+18] Pnar Klıçer, Hugo Labrande, Reynald Lercier, Christophe Ritzenthaler, Jeroen Sijsling, and Marco Streng. Plane quartics over $\mathbb{Q}$ with complex multiplication. *Acta Arith.*, 185(2):127–156, 2018.

[Lam73] T. Y. Lam. *The Algebraic Theory of Quadratic Forms*. W. A. Benjamin, Inc., Reading, MA, 1973.

[Lav67] A. F. Lavrik. An approximate functional equation for the Hecke zeta-function of an imaginary quadratic field. *Mat. Zametki*, 2:475–482, 1967.

[LL03] R. Lercier and D. Lubicz. Counting Points on Elliptic Curves over Finite Fields of Small Characteristic in Quasi Quadratic Time. In *Advances in Cryptology—EuroCrypt 2003*, volume 2656 of LNCS, pages 360–373. Springer, 2003.

[LL06] R. Lercier and D. Lubicz. A Quasi-Quadratic Time Algorithm for Hyperelliptic Curve Point Counting. *to appear*, 2006.

[LRS12] R. Lercier, C. Ritzenthaler, and J. Sijsling. Fast computation of isomorphisms of hyperelliptic curves and explicit descent. In E. W. Howe and K. S. Kedlaya, editors, *Proceedings of the Tenth Algorithmic Number Theory Symposium*, 463–486. Mathematical Sciences Publishers, 2012.

[LLGR20] Reynald Lercier, Qing Liu, Elisa Lorenzo García, and Christophe Ritzenthaler. Reduction type of smooth quartics. 2020. arXiv:1803.05816.

[LR12] Reynald Lercier and Christophe Ritzenthaler. Hyperelliptic curves and their invariants: geometric, arithmetic and algorithmic aspects. *J. Algebra*, 372:595–636, 2012. http://dx.doi.org/10.1016/j.jalgebra.2012.07.054.

[LRRS14] Reynald Lercier, Christophe Ritzenthaler, Florent Rovetta, and Jeroen Sijsling. Parametrizing the moduli space of curves and applications to smooth plane quartics over finite fields. *LMS J. Comput. Math.*, 17(suppl. A):128–147, 2014.

[LRS16] Reynald Lercier, Christophe Ritzenthaler, and Jeroen Sijsling. Explicit Galois obstruction and descent for hyperelliptic curves with tamely cyclic reduced automorphism group. *Math. Comp.*, 85(300):2011–2045, 2016.

[LRS21] Reynald Lercier, Christophe Ritzenthaler, and Jeroen Sijsling. Functionalities for genus 2 and 3 curves. https://arxiv.org/abs/2102.04372, 2021. arXiv:2102.04372.

[MS16] J.S. Müller and M. Stoll. Canonical heights on genus two Jacobians. *Algebra & Number Theory*, 10:2153–2234, 2016. https://doi.org/10.2140/ant.2016.10.2153. doi:10.2140/ant.2016.10.2153.

[Mae90a] T. Maeda. On the invariant fields of binary octavics. *Hiroshima Math. J.*, 20:619–632, 1990.

[MSSV02] K. Magaard, T. Shaska, S. Shpectorov, and H. Völklein. The locus of curves with prescribed automorphism group. *Sūrikaisekikenkyūsho Kōkyūroku*, pages 112–141, 2002. Communications in arithmetic fundamental groups (Kyoto, 1999/2001).

[MW06] P. Martin and M. Watkins. Symmetric powers of elliptic curve L-functions. In F. Hess, S. Pauli, and M. Pohst, editors, *Algorithmic Number Theory*, volume 4076 of Lecture Notes in Computer Science, 377–392. Springer, 2006.

[MST06] B. Mazur, W. Stein, and J. Tate. Computation of p-adic heights and log convergence. *Documenta Mathematica*, Extra:577–614, 2006.

[MT91] B. Mazur and J. Tate. The p-adic sigma function. *Duke Math Journal*, 62(3):663–688, 1991.

[MT10] Stephen Meagher and Jaap Top. Twists of genus three curves over finite fields. *Finite Fields Appl.*, 16(5):347–368, 2010.

[MSS96] J. R. Merriman, S. Siksek, and N. P. Smart. Explicit 4-descents on an elliptic curve. *Acta Arith.*, 77(4):385–404, 1996.

[Mes91] J.-F. Mestre. Construction de courbes de genre 2 à partir de leurs modules. In T. Mora and C. Traverso, editors, *Effective methods in algebraic geometry*, volume 94 of Progr. Math., 313–334. Birkhäuser, 1991. Proc. Congress in Livorno, Italy, April 17–21, 1990.

[Mum84] David Mumford. *Tata Lectures on Theta II.* Volume 43 of Progress in Mathematics. Birkhäuser, 1984.

[Mul10a] Jan Steffen Müller. *Computing canonical heights on Jacobians.* PhD Thesis, Universität Bayreuth, 2010. available at http://www.math.uni-hamburg.de/home/js.mueller/.

[Mul10b] Jan Steffen Müller. Explicit Kummer surface formulas for arbitrary characteristic. *LMS J. Comput. Math.*, 4:47–64, 2010.

[Nag28] Trygve Nagell. Sur les propriétés arithmétiques des cubiques planes du premier genre. *Acta Math.*, 52:93–126, 1928.

[Ohn07] T. Ohno. The graded ring of invariants of ternary quartics I. 2007. Unpublished.

[PGS00] F. Heß P. Gaudry and N. P. Smart. Constructive and destructive facets of Weil descent on elliptic curves. *J. Cryptology*, 15(1):19–46, 2000.

[PS97b] Bjorn Poonen and Ed Schaefer. Explicit descent for Jacobians of cyclic covers of the projective line. *J. Reine Angew. Math.*, 488:141–188, 1997.

[PS99] Bjorn Poonen and Michael Stoll. The Cassels-Tate pairing on polarized abelian varieties. *Ann. of Math. (2)*, 150(3):1109–1149, 1999.

[Rob07] David Roberts. *Explicit Descent On Elliptic Curves Over Function Fields.* PhD thesis, University of Nottingham, 2007.

[Roh11] D. E. Rohrlich. Root Numbers. In K. Rubin C. Popescu and A. Silverberg, editors, *Arithmetic of L-functions*, volume 18 of IAS / Park City Math. Ser., pages 353–448. Amer. Math. Soc., 2011.

[RS95] K. Rubin and A. Silverberg. Families of elliptic curves with constant mod p representations. In *Elliptic curves, modular forms, & Fermat's last theorem (Hong Kong, 1993)*, Ser. Number Theory, I, pages 148–161. Internat. Press, Cambridge, MA, 1995.

[Sal79] G. Salmon. *A Treatise on the Higher Plane Curves*. Hodges, Foster and Co, Dublin, 3rd edition, 1879.

[SS04] Edward F. Schaefer and Michael Stoll. How to do a p-descent on an elliptic curve. *Trans. Amer. Math. Soc.*, 356(3):1209–1231 (electronic), 2004.

[Ser71] J.-P. Serre. Conducteurs d'Artin des caractères réels. *Invent. Math.*, 14(3):173–183, 1971.

[Ser65] Jean-Pierre Serre. Zeta and L functions. In *Arithmetical Algebraic Geometry (Proc. Conf. Purdue Univ., 1963)*, pages 82–92. Harper & Row, New York, 1965.

[Sha95] I. R. Shafarevich, editor. *Number theory I*. Volume 49 of Encyclopaedia of Mathematical Sciences. Springer-Verlag, Berlin, 1995. Fundamental problems, ideas and theories, A translation of Number theory 1 (Russian), Akad. Nauk SSSR, Vsesoyuz. Inst. Nauchn. i Tekhn. Inform., Moscow, 1990, Translation edited by A. N. Parshin and I. R. Shafarevich.

[SV01] Tony Shaska and Völklein. *Elliptic subfields and automorphisms of genus* 2 *function fields*. Springer, 2001, http://au.arxiv.org/abs/math.AG/0107142.

[Shi67] T. Shioda. On the Graded Ring of Invariants of Binary Octavics. *Am. Jour. Math.*, 89(4):1022–1046, 1967.

[Sik95] Samir Siksek. Infinite descent on elliptic curves. *Rocky Mountain J. Math.*, 25(4):1501–1538, 1995.

[Sil86] J. Silverman. *The arithmetic of elliptic curves*. Volume 106 of Graduate Texts in Mathematics. Springer-Verlag, New York, 1986.

[Sim05b] Denis Simon. Solving quadratic equations using reduced unimodular quadratic forms. *Math. Comp.*, 74(251):1531–1543 (electronic), 2005.

[Sma97] N. P. Smart. S-unit equations, binary forms and curves of genus 2. *Proc. London Math. Soc. (3)*, 75(2):271–307, 1997.

[Smi05] Benjamin A. Smith. *Explicit endomorphisms and correspondences*. PhD thesis, University of Sydney, 2005. http://hdl.handle.net/2123/1066.

[SW] (no details available)

[Sto99b] M. Stoll. On the height constant for curves of genus two. *Acta Arithmetica*, 90:183–201, 1999. https://doi.org/10.4064/aa-90-2-183-201. doi:10.4064/aa-90-2-183-201.

[Sto01] Michael Stoll. Implementing 2-descent for Jacobians of hyperelliptic curves. *Acta Arith.*, 98(3):245–277, 2001.

[ST96] R. J. Stroeker and Tzanakis T. Solving elliptic diophantine equations by estimating linear forms in elliptic logarithms, the quartic case. *Acta Arith.*, 75:165–190, 1996.

[ST94] R. J. Stroeker and N. Tzanakis. Solving elliptic diophantine equations by estimating linear forms in elliptic logarithms. *Acta Arith.*, 67:177–196, 1994.

[Tol97] Emmanuel Tollis. Zeros of Dedekind zeta functions in the critical strip. *Math. Comp.*, 66(219):1295–1321, 1997.

[vGvS93] B. van Geemen and D. van Straten. The cusp forms of weight 3 on $\Gamma_2(2,4,8)$. *Math. Comp.*, 61(204):849–872, 1993.

[vR01] Sander Matthijs van Rijnswou. *Testing the equivalence of planar curves*. PhD thesis, Technische Universiteit Eindhoven, Eindhoven, 2001.

[Ver02] F. Vercauteren. Computing zeta functions of hyperelliptic curves over finite fields of characteristic 2. In *Advances in cryptology—CRYPTO 2002*, volume 2442 of LNCS, pages 369–384. Springer, Berlin, 2002.

[Vig80] M.-F. Vignéras. *Arithmétique des Algèbres de Quaternions*. Volume 800 of Lecture Notes in Mathematics. Springer-Verlag, Berlin, 1980.

[Wam99] P. Van Wamelen. Examples of Genus Two CM Curves Defined over the Rationals. *Mathematics of Computation*, 68:307–320, 1999.

[Wat02] M. Watkins. Computing the modular degree of an elliptic curve. *Experimental Mathematics*, 11(4):487–502, 2002.

[Wei52] A. Weil. Jacobi Sums as "Grössencharaktere". *Trans. AMS*, 73:487–495, 1952.

[Wei74] A. Weil. Sommes de Jacobi et caractères de Hecke. *Göttingen Nachr.*, 1:1–14, 1974.

[Wei76] A. Weil. Sur les périodes des intégrales abéliennes. *Comm. Pure Appl. Math.*, XXIX:813–819, 1976.

[Wet97] J. L. Wetherell. *Bounding the number of rational points on certain curves of high rank*. PhD thesis, U.C. Berkeley, 1997.

[LRS18] R. Lercier, C. Ritzenthaler, and J. Sijsling. Reconstructing plane quartics from their invariants. *Discrete & Computational Geometry*, pages 1–41, 2018.

Index

Symbols

C (*operation*), 169
D (*operation*), 87
E (*operation*), 50
G (*operation*), 33
H (*operation*), 50
J (*operation*), 183
- P (*operation*), 43, 168, 181, 220
2-Isogeny Descents with Full 2-Torsion (*example*), 108
2-Selmer Group of a Jacobian (*example*), 197

A

A Classic Curve from Kramer (*example*), 109
A Cubic Violating the Hasse Principle (*example*), 149
Abelian Group of Points (*example*), 56
AbelianGroup(E) (*function*), 55
AbelianGroup(E: parameters) (*function*), 99
AbelianGroup(H) (*function*), 55
AbelianGroup(H: parameters) (*function*), 99
AbelianGroup(J) (*function*), 188
Absolute Algebra of an Etale Algebra (*example*), 95
AbsoluteAlgebra(A) (*function*), 94
AbsoluteInvariants(C) (*function*), 234
Action of the Frobenius Map (*example*), 41
AddCubics(cubic1, cubic2 : parameters) (*function*), 119, 147
AddCubics(model1, model2 : parameters) (*function*), 147
Addition on the Analytic Jacobian (*example*), 223
aInvariants(E) (*function*), 27
aInvariants(model) (*function*), 148
All Integral Points on a Curve (*example*), 84
AlphaBetaData(H) (*function*), 255
AlternatingSquare(HS) (*function*), 306
Analytic Rank of a Rank 6 Curve (*example*), 82
AnalyticHomomorphisms(t1, t2) (*function*), 225
AnalyticInformation(E) (*function*), 134
AnalyticInformation(E, e) (*function*), 134
AnalyticInformation(E, L) (*function*), 134
AnalyticJacobian(f) (*function*), 222
AnalyticRank(E) (*function*), 82, 95, 134
ApplyTransformation(g, model) (*function*), 144
Artin L-Series over Q(i) (*example*), 282
Artin Representation of A7 (*example*), 282
ArtinRepresentation(H, t) (*function*), 258
Associated Schemes in Weighted Projective Space (*example*), 262
AssociatedEllipticCurve(C) (*function*), 103
AssociatedEllipticCurve(f) (*function*), 103
AssociatedEllipticCurve(qi) (*function*), 113
AssociatedHyperellipticCurve(qi) (*function*), 113
AssociatedSchemes(H) (*function*), 259
AssociatedSchemes(H, t) (*function*), 259
AteqPairing(P, Q, m, q) (*function*), 58
AteTPairing(Q, P, n, q) (*function*), 58
Aut(C) (*function*), 240
Automorphism(C, a) (*function*), 16
Automorphism(C, S, T) (*function*), 15
Automorphism(E, [r, s, t, u]) (*function*), 37
AutomorphismGroup(C) (*function*), 241
AutomorphismGroup(E) (*function*), 41
AutomorphismGroupOfHyperellipticCurve(f) (*function*), 243
AutomorphismGroupOfHyperellipticCurve(f, Autos) (*function*), 243
AutomorphismGroupOfHyperellipticCurve(X) (*function*), 243
AutomorphismGroupOfHyperellipticCurve(X, Autos) (*function*), 243
AutomorphismGroupOfPlaneQuartic(X) (*function*), 155
AutomorphismGroupOfPlaneQuartic(X, Autos) (*function*), 156
AutomorphismGroupOfTernaryQuartic(f) (*function*), 155
AutomorphismGroupOfTernaryQuartic(f, Autos) (*function*), 156
Automorphisms of a Conic Without Points (*example*), 16
Automorphisms of a Genus One Curve (*example*), 241
Automorphisms of the Klein Quartic (*example*), 155
Automorphisms over a Finite Field (*example*), 244
Automorphisms(E) (*function*), 41
AutomorphismsOfHyperellipticCurve(f) (*function*), 243
AutomorphismsOfHyperellipticCurve(X) (*function*), 243
AutomorphismsOfPlaneQuartic(X) (*function*), 155
AutomorphismsOfTernaryQuartic(f) (*function*),

155

B

Bad Primes via Norm Residue Symbols (*example*), 8
BadPlaces(E) (*function*), 89, 129
BadPlaces(E, L) (*function*), 89
BadPrimeData(L) (*function*), 300
BadPrimes(C) (*function*), 7, 195
BadPrimes(E) (*function*), 65
BadPrimes(J) (*function*), 195
Base Change of a Hyperelliptic Curve (*example*), 286
Base Change of an Elliptic Curve (*example*), 284
BaseChange(C, j) (*function*), 160
BaseChange(C, K) (*function*), 159
BaseChange(C, n) (*function*), 160
BaseChange(E, h) (*function*), 23
BaseChange(E, K) (*function*), 23
BaseChange(E, n) (*function*), 23
BaseChange(J, F) (*function*), 171
BaseChange(J, j) (*function*), 171
BaseChange(J, n) (*function*), 171
BaseChange(K, F) (*function*), 219
BaseChange(K, j) (*function*), 219
BaseChange(K, n) (*function*), 219
BaseExtend(C, j) (*function*), 160
BaseExtend(C, K) (*function*), 159
BaseExtend(C, n) (*function*), 160
BaseExtend(E, h) (*function*), 23
BaseExtend(E, K) (*function*), 23
BaseExtend(E, n) (*function*), 23
BaseExtend(J, F) (*function*), 171
BaseExtend(J, j) (*function*), 171
BaseExtend(J, n) (*function*), 171
BaseExtend(K, F) (*function*), 219
BaseExtend(K, j) (*function*), 219
BaseExtend(K, n) (*function*), 219
BaseField(A) (*function*), 222
BaseField(C) (*function*), 3, 166
BaseField(J) (*function*), 171
BaseField(K) (*function*), 219
BaseRing(A) (*function*), 222
BaseRing(C) (*function*), 3, 166
BaseRing(E) (*function*), 29
BaseRing(G) (*function*), 33
BaseRing(J) (*function*), 171
BaseRing(K) (*function*), 219
BaseRing(model) (*function*), 143
Basis(S) (*function*), 131
Basis(S, r, disc) (*function*), 131
Better Rank Bounds by Torsor Descent (*example*), 200
BettiNumber(E, i) (*function*), 135
Bezoutian(H) (*function*), 255
BigPeriodMatrix(A) (*function*), 223
bInvariants(E) (*function*), 27
bInvariants(model) (*function*), 148
Both Chabauty Implementations (*example*), 208
BSD in a Non-Abelian Extension (*example*), 322

C

C ! P (*operation*), 167
C ! [x, y, z] (*operation*), 167
C ! [x, y] (*operation*), 167
Canonical Heights of Generators (*example*), 69
Canonical Schemes and Curves (*example*), 262
CanonicalCurve(H) (*function*), 259
CanonicalCurve(H, t) (*function*), 259
CanonicalHeight(P: parameters) (*function*), 69, 191
CanonicalHeight(P: Precision) (*function*), 191
CanonicalScheme(H) (*function*), 259
CanonicalScheme(H, t) (*function*), 259
Cassels-Tate Pairing with Nontrivial Sha (*example*), 111
CasselsMap(phi) (*function*), 106
CasselsTatePairing(C, D) (*function*), 111
Category(C) (*function*), 3
Category(E) (*function*), 29
Category(G) (*function*), 32
Category(H) (*function*), 35
Category(P) (*function*), 43
CentralValue(L) (*function*), 291
CFENew(L) (*function*), 295
Chabauty for a Rank 0 Jacobian (*example*), 210
Chabauty on a Rank 1 Jacobian (*example*), 209
Chabauty(MWmap, Ecov) (*function*), 91
Chabauty(MWmap, Ecov, p) (*function*), 92
Chabauty(P : ptC) (*function*), 208
Chabauty(P, p: Precision) (*function*), 208
Chabauty0(J) (*function*), 208
ChangeEulerFactor(L,p,f) (*function*), 301
ChangeLocalInformation(L,bp) (*function*), 301
ChangeLocalInformation(L,p,d,f) (*function*), 301
ChangeRing(C, K) (*function*), 160
ChangeRing(E, K) (*function*), 23
ChangeRing(g, R) (*function*), 144
ChangeRing(model, R) (*function*), 142
Changing Local Information (*example*), 301
CharacteristicPolynomialFromTraces(traces) (*function*), 135
CharacteristicPolynomialFromTraces(traces, d, q, i) (*function*), 135
CheckFunctionalEquation(L) (*function*), 295
Checking a Functional Equation (*example*), 296
cInvariants(E) (*function*), 28
cInvariants(model) (*function*), 148
Classifying Genus 2 Automorphism Groups (*example*), 246
ClebschInvariants(C) (*function*), 233
ClebschInvariants(f) (*function*), 233

ClebschToIgusaClebsch(Q) (*function*), 235
CMEndomorphism(E, alpha) (*function*), 32
Codomain(f) (*function*), 240
CoefficientRing(A) (*function*), 222
CoefficientRing(C) (*function*), 166
CoefficientRing(E) (*function*), 29
CoefficientRing(G) (*function*), 33
CoefficientRing(J) (*function*), 171
CoefficientRing(K) (*function*), 219
Coefficients of a Modular Form L-Series (*example*), 300
Coefficients(E) (*function*), 27
Comparing Shanks and Pollard (*example*), 185
Comparing Silverman and Siksek Bounds (*example*), 70
CompleteTheSquare(model) (*function*), 142
Complex Rank by Reduction Modulo p (*example*), 137
ComplexEvaluation(J, P) (*function*), 273
ComposeTransformations(g1, g2) (*function*), 144
Conductor over a p-Adic Field (*example*), 166
Conductor over the Rationals (*example*), 165
Conductor(C) (*function*), 165
Conductor(C, P) (*function*), 165
Conductor(C, p) (*function*), 165
Conductor(E) (*function*), 65, 89, 97, 129
Conductor(L) (*function*), 299
ConductorExponent(C) (*function*), 165
ConductorRange(D) (*function*), 87
Conductors in the Cremona Database (*example*), 88
Conic Model of a Rational Curve (*example*), 13
Conic(C) (*function*), 13
Conic(coeffs) (*function*), 2
Conic(M) (*function*), 2
Conic(X, f) (*function*), 2
Conjectural Regulator of a Curve (*example*), 83
ConjecturalRegulator(E) (*function*), 82, 95
ConjecturalRegulator(E, v) (*function*), 82
ConjecturalSha(E, Pts) (*function*), 96
Consani-Scholten Tensor Product (*example*), 309
Contravariants(model) (*function*), 149
ContravariantSigmaAndPsi(Phi) (*function*), 153
Converting Isogenies and Isomorphisms (*example*), 37
CopyCoefficients(L,M) (*procedure*), 301
CovariantHessian(Phi) (*function*), 153
CoveringCovariants(model) (*function*), 149
CPSHeightBounds(E) (*function*), 91
Creating a Conic from a Curve (*example*), 2
Creating a Hyperelliptic Curve (*example*), 159
CremonaDatabase(: parameters) (*function*), 87
CremonaReference(D, E) (*function*), 87
CremonaReference(E) (*function*), 87
CriticalPoints(HS) (*function*), 306
CriticalPoints(L) (*function*), 306
CubicFromPoint(E, P) (*function*), 142
Curve from Igusa-Clebsch Invariants (*example*), 238
Curve of Moderately Large Rank (*example*), 100
Curve(G) (*function*), 32
Curve(H) (*function*), 35
Curve(J) (*function*), 171
Curve(model) (*function*), 143
Curve(P) (*function*), 43
Curves with Kodaira Symbol I0* (*example*), 67
CyclotomicData(H) (*function*), 255

D

Dedekind Zeta Function from Scratch (*example*), 318
Dedekind Zeta Function of Q(i) (*example*), 280
Dedekind Zeta of a Degree 12 Field (*example*), 281
Dedekind Zeta Vanishing at the Centre (*example*), 281
DedekindZetaExact(K, z) (*function*), 292
DefiningEquations(model) (*function*), 143
DefiningIdeal(C) (*function*), 3
DefiningPolynomial(C) (*function*), 3, 167
DefiningPolynomial(E) (*function*), 30
DefiningPolynomial(K) (*function*), 219
DefiningPolynomials(H) (*function*), 255
DefiningSubschemePolynomial(G) (*function*), 33
Deformation Point Counting for a Family (*example*), 187
Degree(C) (*function*), 165
Degree(H) (*function*), 255
Degree(HS) (*function*), 305
Degree(I) (*function*), 40
Degree(L) (*function*), 299
Degree(model) (*function*), 143
Delaunay(sites) (*function*), 231
Derivatives of a Rank 3 L-Series (*example*), 292
Descent Proving No Rational Points (*example*), 213
DescentInformation(E: parameters) (*function*), 101
DescentMaps(phi) (*function*), 106
Determinant(g) (*function*), 145
Determinant(HS) (*function*), 306
Determinant(L) (*function*), 310
Diagonal Equations for Conics (*example*), 5
DiagonalModel(n, seq) (*function*), 142
Dimension(A) (*function*), 222
Dimension(J) (*function*), 171
Direct Point Counting (*example*), 184
Dirichlet L-Function Modulo 37 (*example*), 287
Discriminant of the Klein Quartic (*example*), 153

Discriminant(C) (*function*), 6, 165
Discriminant(E) (*function*), 28
Discriminant(model) (*function*), 148
DiscriminantFromDixmierOhnoInvariants(DO) (*function*), 153
DiscriminantFromIgusaInvariants(JI) (*function*), 234
DiscriminantFromShiodaInvariants(JI) (*function*), 236
DiscriminantOfTernaryQuartic(f) (*function*), 153
Division Polynomials and Torsion Points (*example*), 30
Division Polynomials for Complex Multiplication (*example*), 32
DivisionPoints(P, n) (*function*), 44
DivisionPolynomial(E, a) (*function*), 31
DivisionPolynomial(E, alpha) (*function*), 31
DivisionPolynomial(E, n) (*function*), 30
DivisionPolynomial(E, n, g) (*function*), 30
Dixmier-Ohno Invariants of the Klein Quartic (*example*), 153
DixmierOhnoAlgebraicRelations(DOinv) (*function*), 153
DixmierOhnoInvariants(C) (*function*), 152
DixmierOhnoInvariants(f) (*function*), 152
DixmierOhnoInvariantsEqual(DO1, DO2) (*function*), 153
Domain(f) (*function*), 240
Double Richelot Isogenies (*example*), 174
Double(P) (*function*), 220
DoubleGenusOneModel(model) (*function*), 148
DoubleRichelotIsogenies(C) (*function*), 174
DoubleRichelotIsogenies(J) (*function*), 174
Dual of an Isogeny (*example*), 39
Dual(HS) (*function*), 305
DualIsogeny(phi) (*function*), 39

E

E ! 0 (*operation*), 41
E ! [x, y, z] (*operation*), 41
E eq F (*operation*), 29
E ne F (*operation*), 29
E(L) (*literal*), 35
E(m) (*literal*), 35
EffectiveHodgeStructure(HS) (*function*), 305
EffectiveHodgeStructure(J) (*function*), 273
EffectiveWeight(J) (*function*), 273
EightDescent(C : parameters) (*function*), 115
EisensteinTwo(E, p) (*function*), 72
ElementToSequence(E) (*function*), 27
ElementToSequence(P) (*function*), 43, 168, 181, 220
Elliptic Curve Chabauty over a Number Field (*example*), 92
Elliptic Curve Discrete Logarithm (*example*), 63
Elliptic Curve from a Pencil of Cubics (*example*), 136
Elliptic Curve L-Series from Scratch (*example*), 317
Elliptic Exponential and Logarithm (*example*), 81
Elliptic Model via Riemann-Roch (*example*), 22
EllipticCurve(C) (*function*), 20
EllipticCurve(C, P) (*function*), 21
EllipticCurve(C, pl) (*function*), 21
EllipticCurve(D, N, I, J) (*function*), 87
EllipticCurve(D, N, S, J) (*function*), 87
EllipticCurve(D, S) (*function*), 87
EllipticCurve(f) (*function*), 20
EllipticCurve(f, h) (*function*), 20
EllipticCurve(H) (*function*), 258
EllipticCurve(H, t) (*function*), 258
EllipticCurve(S) (*function*), 87
EllipticCurve([a, b]) (*function*), 20
EllipticCurve([a1, a2, a3, a4, a6]) (*function*), 20
EllipticCurveDatabase(: parameters) (*function*), 87
EllipticCurveFromjInvariant(j) (*function*), 20
EllipticCurveFromPeriods(om: parameters) (*function*), 79
EllipticCurves(D) (*function*), 88
EllipticCurves(D, N) (*function*), 88
EllipticCurves(D, N, I) (*function*), 88
EllipticCurves(D, N, S) (*function*), 88
EllipticCurves(D, S) (*function*), 88
EllipticCurveSearch(N, Effort) (*function*), 96, 130
EllipticCurveWithGoodReductionSearch(N, Effort) (*function*), 96
EllipticCurveWithGoodReductionSearch(S, Effort) (*function*), 96
EllipticCurveWithjInvariant(j) (*function*), 20
EllipticExponential(E, k, z) (*function*), 80
EllipticExponential(E, S) (*function*), 80
EllipticExponential(E, z) (*function*), 80
EllipticLogarithm(E, S) (*function*), 80
EllipticLogarithm(P) (*function*), 80
EllipticLogarithm(P, k) (*function*), 80
elt< E | x, y, z > (*constructor*), 41
elt< H | x, y, z > (*constructor*), 41
elt< J | a, b > (*constructor*), 178
elt< J | a, b, d > (*constructor*), 178
elt< J | P, Q > (*constructor*), 178
elt< J | S, T > (*constructor*), 178
elt< J | [a, b] > (*constructor*), 178
elt< J | [a, b], d > (*constructor*), 178
elt< PS | x, y > (*constructor*), 167
elt< PS | x, y, z > (*constructor*), 167
Eltseq(E) (*function*), 27
Eltseq(model) (*function*), 143
Eltseq(P) (*function*), 43, 168, 181, 220

EndomorphismRing(A) (*function*), 226
EndomorphismRing(P) (*function*), 226
Enumerating Hypergeometric Data (*example*), 268
Equations(model) (*function*), 143
EtaqPairing(P, Q, n, q) (*function*), 57
EtaTPairing(P, Q, n, q) (*function*), 57
Euler Factors and Conductors at 3 (*example*), 166
Euler Factors at Bad Primes (*example*), 265
EulerFactor(C) (*function*), 166
EulerFactor(C, P) (*function*), 166
EulerFactor(C, p) (*function*), 166
EulerFactor(H, t, p) (*function*), 256
EulerFactor(J) (*function*), 187
EulerFactor(J, K) (*function*), 187
EulerFactor(J, p) (*function*), 273
EulerFactor(L, p) (*function*), 299
EulerFactorModChar(J) (*function*), 187
EulerFactorsByDeformation(Q, Y) (*function*), 187
Evaluate(L, s0) (*function*), 291
EvaluatePolynomial(C, a, b, c) (*function*), 167
EvaluatePolynomial(C, [a, b, c]) (*function*), 167
Extending the Base Field (*example*), 23

F

FactoredOrder(E) (*function*), 50
FactoredOrder(G) (*function*), 33
FactoredOrder(H) (*function*), 50
FactoredOrder(J) (*function*), 186
FactoredOrder(P) (*function*), 46
Factorisation(L) (*function*), 301
Factorization of L-Series (*example*), 301
Factorization(L) (*function*), 301
FakeIsogenySelmerSet(C,E1,E2) (*function*), 123
FakeIsogenySelmerSet(C,P) (*function*), 123
FakeIsogenySelmerSet(C,phi) (*function*), 123
FaltingsHeight(E) (*function*), 68
Field(J) (*function*), 273
Finding a Genus 2 CM Curve (*example*), 229
Finding a Rational Isogeny (*example*), 226
Finding Heegner Points (*example*), 76
Finding Points on a Conic (*example*), 11
FormalGroupHomomorphism(phi, prec) (*function*), 34
FormalGroupLaw(E, prec) (*function*), 34
FormalLog(E) (*function*), 34
Four-Descent and Mapping Points (*example*), 114
Four-Descent Proving Rank 0 (*example*), 112
FourCoverPullback(C, pt) (*function*), 114
FourDescent(C : parameters) (*function*), 112
FourDescent(f : parameters) (*function*), 112
FourDescent(S : parameters) (*function*), 112
FourToTwoCovering(C : parameters) (*function*), 148
FourToTwoCovering(model : parameters) (*function*), 148
Frobenius(P, F) (*function*), 170
Frobenius(P, k) (*function*), 182
Frobenius(P, q) (*function*), 136
FrobeniusActionOnPoints(S, q : parameters) (*function*), 136
FrobeniusActionOnReducibleFiber(L) (*function*), 136
FrobeniusActionOnTrivialLattice(E) (*function*), 136
FrobeniusMap(E) (*function*), 41
FrobeniusMap(E, i) (*function*), 40
FrobeniusMatrix(C, p) (*function*), 170
FrobeniusMatrix(E, p) (*function*), 73
FromAnalyticJacobian(z, A) (*function*), 223
FromAnalyticJacobianProjective(z, A) (*function*), 228
FunctionField(C) (*function*), 167

G

g * model (*operation*), 144
g1 * g2 (*operation*), 144
G1 eq G2 (*operation*), 33
G1 ne G2 (*operation*), 33
G2Invariants(C) (*function*), 235
G2ToIgusaInvariants(GI) (*function*), 235
GammaArray(H) (*function*), 256
GammaFactors(HS) (*function*), 305
GammaFactors(L) (*function*), 299
GammaList(H) (*function*), 256
GammaShifts(HS) (*function*), 305
General Symmetrizations of a Modular Form (*example*), 313
Generators(E) (*function*), 56, 99, 132
Generators(H) (*function*), 56, 99
Generic Genus One Model of Degree 5 (*example*), 142
Generic Maps Between Curves (*example*), 38
Generic Point on an Elliptic Surface (*example*), 45
Generic(G) (*function*), 32
GenericModel(n) (*function*), 141
Genuine and Fake Selmer Sets (*example*), 125
Genus 2 Curve over Q(i) (*example*), 287
Genus 2 L-Series from Scratch (*example*), 318
Genus(A) (*function*), 222
Genus(C) (*function*), 165
GenusOneModel(C) (*function*), 141
GenusOneModel(f) (*function*), 141
GenusOneModel(mat) (*function*), 141
GenusOneModel(mats) (*function*), 141
GenusOneModel(n, E) (*function*), 141
GenusOneModel(n, seq) (*function*), 141
GenusOneModel(n, str) (*function*), 141
GenusOneModel(R, n, seq) (*function*), 141

GenusOneModel(seq) (*function*), 141
Geometric Automorphism Groups (*example*), 245
GeometricAutomorphismGroup(C) (*function*), 156, 244
GeometricAutomorphismGroupFromIgusa Invariants(GI) (*function*), 244
GeometricAutomorphismGroupFromShioda Invariants(JI) (*function*), 245
GeometricAutomorphismGroupGenus2 Classification(F) (*function*), 246
GeometricAutomorphismGroupGenus3 Classification(F) (*function*), 246
GeometricMordellWeilLattice(E) (*function*), 132
GeometricTorsionBound(E) (*function*), 131
Grossencharacter(J) (*function*), 273
Group Order and Trace of Frobenius (*example*), 53

H

H ! 0 (*operation*), 41
H ! [x, y, z] (*operation*), 41
h eq k (*operation*), 67
h ne k (*operation*), 67
H1 eq H2 (*operation*), 35, 256
H1 ne H2 (*operation*), 35, 256
HasAdditionAlgorithm(J) (*function*), 188
HasComplexMultiplication(E) (*function*), 67, 90
HasHodgeStructure(L) (*function*), 305
HasIndexOne(C, p) (*function*), 195
HasIndexOneEverywhereLocally(C) (*function*), 195
HasOddDegreeModel(C) (*function*), 160
HasOrder(P, n) (*function*), 182
HasPoint(f, q, v) (*function*), 212
HasPointsEverywhereLocally(f, q) (*function*), 212
HasRationalPoint(C) (*function*), 9
HasSquareSha(J) (*function*), 195
Heegner Points for a Given Discriminant (*example*), 78
Heegner Points over a Class Field (*example*), 78
Heegner Points Using an Isogeny (*example*), 77
Heegner Points via Two-Descent (*example*), 77
HeegnerDiscriminants(E,lo,hi) (*function*), 75
HeegnerForms(E,D : parameters) (*function*), 75
HeegnerForms(N,D : parameters) (*function*), 76
HeegnerPoint(C : parameters) (*function*), 74
HeegnerPoint(E : parameters) (*function*), 74
HeegnerPoint(f : parameters) (*function*), 74
HeegnerPoints(E, D : parameters) (*function*), 76
HeegnerTorsionElement(E, Q) (*function*), 76
Height(P : parameters) (*function*), 90
Height(P) (*function*), 130
Height(P: parameters) (*function*), 69, 191
Height(P: Precision) (*function*), 191
HeightConstant(J: parameters) (*function*), 191
HeightDifferenceBounds(E) (*function*), 91
HeightDifferenceLowerBound(E) (*function*), 91
HeightDifferenceUpperBound(E) (*function*), 91
HeightPairing(P, Q) (*function*), 131
HeightPairing(P, Q: parameters) (*function*), 69, 191
HeightPairingLattice(S) (*function*), 131
HeightPairingMatrix(E: parameters) (*function*), 69
HeightPairingMatrix(P : parameters) (*function*), 90
HeightPairingMatrix(S) (*function*), 131
HeightPairingMatrix(S: parameters) (*function*), 69
HeightPairingMatrix(S: Precision) (*function*), 192
Heights on a Genus 2 Jacobian (*example*), 192
HesseCovariants(model, r) (*function*), 149
HesseModel(n, seq) (*function*), 142
HessePolynomials(n, r, invariants : parameters) (*function*), 149
Hessian(model) (*function*), 149
Higher Orthogonal Symmetrizations (*example*), 316
HilbertSymbol(a, b, p : parameters) (*function*), 8
HilbertSymbol(a, b, p) (*function*), 8
Hodge Structures of L-Series (*example*), 306
HodgeStructure(J) (*function*), 273
HodgeStructure(L) (*function*), 305
HodgeStructure(w, G) (*function*), 305
HodgeStructure(X) (*function*), 305
HodgeVector(HS) (*function*), 306
HodgeVector(J) (*function*), 273
HomologyBasis(A) (*function*), 222
HyperellipticCurve(E) (*function*), 159
HyperellipticCurve(f) (*function*), 158
HyperellipticCurve(f, h) (*function*), 158
HyperellipticCurve(H) (*function*), 258
HyperellipticCurve(H, t) (*function*), 258
HyperellipticCurve(model) (*function*), 143
HyperellipticCurve(P, f, h) (*function*), 158
HyperellipticCurve([f, h]) (*function*), 158
HyperellipticCurveFromG2Invariants(S) (*function*), 237
HyperellipticCurveFromIgusaClebsch(S) (*function*), 237
HyperellipticCurveFromIgusaInvariants(S) (*function*), 237
HyperellipticCurveFromShiodaInvariants(JI) (*function*), 237
HyperellipticCurveOfGenus(g, f) (*function*), 159
HyperellipticCurveOfGenus(g, f, h) (*function*), 159
HyperellipticCurveOfGenus(g, [f, h])

(*function*), 159
HyperellipticPolynomial(A) (*function*), 222
HyperellipticPolynomialFromShioda Invariants(JI) (*function*), 237
HyperellipticPolynomials(C) (*function*), 165
HyperellipticPolynomials(E) (*function*), 28
HyperellipticPolynomialsFromShioda Invariants(JI) (*function*), 247
Hypergeometric Traces for Non-Galois Data (*example*), 263
HypergeometricData(A, B) (*function*), 254
HypergeometricData(E) (*function*), 255
HypergeometricData(F, G) (*function*), 254, 255
HypergeometricData(G) (*function*), 254
HypergeometricData(L) (*function*), 255
HypergeometricMotiveClearTable() (*procedure*), 259
HypergeometricMotiveSaveLimit(n) (*procedure*), 259
HypergeometricTrace(H, t, q) (*function*), 256
HypergeometricTraceK(A, B, t, q) (*function*), 256

I

I eq J (*operation*), 40
Id(E) (*function*), 41
Id(H) (*function*), 41
Id(J) (*function*), 178
Identify(H, t) (*function*), 258
Identity(E) (*function*), 41
Identity(H) (*function*), 41
Identity(J) (*function*), 178
IdentityIsogeny(E) (*function*), 40
IdentityMap(E) (*function*), 40
IdentityTransformation(n, R) (*function*), 144
IgusaAlgebraicRelations(JI) (*function*), 234
IgusaClebschInvariants(C: parameters) (*function*), 233
IgusaClebschInvariants(f, h) (*function*), 233
IgusaClebschInvariants(f: parameters) (*function*), 233
IgusaClebschToIgusa(S) (*function*), 235
IgusaInvariants(C: parameters) (*function*), 233
IgusaInvariants(f, h: parameters) (*function*), 233
IgusaInvariants(f: parameters) (*function*), 234
IgusaInvariantsEqual(JI1, JI2) (*function*), 234
IgusaToG2Invariants(JI) (*function*), 235
ImaginaryTwist(HS) (*function*), 306
Imprimitive Orthogonal Symmetrization (*example*), 316
IndependentGenerators(points) (*function*), 131
Integral and Minimal Models (*example*), 25
Integral and S-Integral Points (*example*), 47
Integral Points on a Quartic (*example*), 86
IntegralModel(C) (*function*), 161
IntegralModel(E) (*function*), 24
IntegralPoints(E) (*function*), 84, 91
IntegralQuarticPoints(Q) (*function*), 86
IntegralQuarticPoints(Q, P) (*function*), 86
Invariants of a Generic Curve (*example*), 28
Invariants of an Elliptic Curve (*example*), 28
Invariants(model) (*function*), 148
Inverse(g) (*function*), 144
InverseTransformation(g) (*function*), 144
InverseTransformation(n, g) (*function*), 144
Inverting the Mordell-Weil Map (*example*), 204
Involution(P) (*function*), 168
IsConic(S) (*function*), 2
IsConstant(I) (*function*), 40
IsConstantCurve(E) (*function*), 130
IsDeficient(C, p) (*function*), 195
IsDivisibleBy(P, n) (*function*), 44, 195
IsEllipticCurve(C) (*function*), 22, 164
IsEllipticCurve([a, b]) (*function*), 22
IsEllipticCurve([a1, a2, a3, a4, a6]) (*function*), 22
IsEquivalent(cubic1, cubic2) (*function*), 145
IsEquivalent(f,g) (*function*), 105
IsEquivalent(model1, model2) (*function*), 145
IsEven(J) (*function*), 195
IsGenusOneModel(f) (*function*), 141
IsGenusOneModel(mat) (*function*), 141
IsGenusOneModel(seq) (*function*), 141
IsGL2Equivalent(f, g, n) (*function*), 248
IsGL2EquivalentExtended(f1, f2, deg) (*function*), 248
IsHyperellipticCurve([f, h]) (*function*), 159
IsHyperellipticCurveOfGenus(g, [f, h]) (*function*), 159
IsId(P) (*function*), 46
IsIdentity(P) (*function*), 46, 181
IsIntegral(C) (*function*), 163
IsIntegral(P) (*function*), 46
IsIntegralModel(E) (*function*), 24
IsIntegralModel(E, P) (*function*), 25
IsIsogenous(A1, A2) (*function*), 225
IsIsogenous(E, F) (*function*), 30
IsIsogenousPeriodMatrices(P1, P2) (*function*), 225
IsIsomorphic(A1, A2) (*function*), 225
IsIsomorphic(C1, C2) (*function*), 241
IsIsomorphic(E, F) (*function*), 29
IsIsomorphicBigPeriodMatrices(P1, P2) (*function*), 225
IsIsomorphicHyperellipticCurves(f1, f2) (*function*), 242
IsIsomorphicHyperellipticCurves(X1, X2) (*function*), 242
IsIsomorphicPlaneQuartics(X1, X2) (*function*), 155

IsIsomorphicSmallPeriodMatrices(t1, t2) (*function*), 225
IsIsomorphicTernaryQuartics(f1, f2) (*function*), 155
IsIsomorphism(I) (*function*), 37
IsLinearlyDependent(points) (*function*), 131
IsLinearlyIndependent(P, Q) (*function*), 71
IsLinearlyIndependent(P, Q, n) (*function*), 47
IsLinearlyIndependent(points) (*function*), 131
IsLinearlyIndependent(S) (*function*), 71
IsLinearlyIndependent(S, n) (*function*), 47
IsLocallySoluble(model) (*function*), 147
IsLocallySoluble(model, p) (*function*), 147
IsLocallySolvable(C) (*function*), 8
IsMinimalModel(E) (*function*), 25
Iso(C1, C2) (*function*), 240
Isogenous Curves via Division Polynomials (*example*), 68
IsogenousCurves(E) (*function*), 67
IsogenyFromKernel(E, psi) (*function*), 39
IsogenyFromKernel(G) (*function*), 38
IsogenyFromKernelFactored(E, psi) (*function*), 39
IsogenyFromKernelFactored(G) (*function*), 38
IsogenyMapOmega(I) (*function*), 40
IsogenyMapPhi(I) (*function*), 40
IsogenyMapPhiMulti(I) (*function*), 40
IsogenyMapPsi(I) (*function*), 40
IsogenyMapPsiMulti(I) (*function*), 40
IsogenyMapPsiSquared(I) (*function*), 40
Isomorphism(E, F) (*function*), 37
Isomorphism(E, F, [r, s, t, u]) (*function*), 36
IsomorphismData(I) (*function*), 37
Isomorphisms Between Two Curves (*example*), 242
IsomorphismsOfHyperellipticCurves(f1, f2) (*function*), 243
IsomorphismsOfHyperellipticCurves(X1, X2) (*function*), 243
IsomorphismsOfPlaneQuartics(X1, X2) (*function*), 155
IsomorphismsOfTernaryQuartics(f1, f2) (*function*), 155
IsomorphismToIsogeny(I) (*function*), 37
IsOrder(P, m) (*function*), 46
IsOrdinary(E) (*function*), 49
IsOrthogonal(L) (*function*), 312
IspIntegral(C, p) (*function*), 163
IspMinimal(C, p) (*function*), 163
IspNormal(C, p) (*function*), 163
IsPoint(C, S) (*function*), 167
IsPoint(E, S) (*function*), 42
IsPoint(E, x) (*function*), 42
IsPoint(H, S) (*function*), 42
IsPoint(H, x) (*function*), 42
IsPoint(K, S) (*function*), 219
IsPrimitive(H) (*function*), 256
IsProbablySupersingular(E) (*function*), 49
IsQuadraticTwist(C, D) (*function*), 164
IsQuadraticTwist(E, F) (*function*), 26
IsQuadricIntersection(C) (*function*), 113
IsRationalCurve(S) (*function*), 2
IsReduced(p) (*function*), 10
IsReducedIsomorphicHyperellipticCurves(f1, f2) (*function*), 249
IsReducedIsomorphicHyperellipticCurves(X1, X2) (*function*), 249
IsSimplifiedModel(C) (*function*), 163
IsSimplifiedModel(E) (*function*), 24
IsSIntegral(P, S) (*function*), 47
IsSupersingular(E : parameters) (*function*), 49
IsSymplectic(L) (*function*), 312
IsTransformation(n, g) (*function*), 144
IsTwist(E, F) (*function*), 26
IsWeierstrassModel(E) (*function*), 24
IsZero(I) (*function*), 40
IsZero(P) (*function*), 46, 181
Iterating over the Cremona Database (*example*), 88

J

J ! 0 (*operation*), 178
J ! P (*operation*), 178
J ! [a, b] (*operation*), 178
J ! [P, Q] (*operation*), 178
J ! [S, T] (*operation*), 178
J1 * J2 (*operation*), 272
J1 / J2 (*operation*), 272
J1 eq J2 (*operation*), 272
J1 ne J2 (*operation*), 272
JacketMotive(A, B, t, rho, j) (*function*), 272
Jacobi Motives and the Klein Quartic (*example*), 274
Jacobi Motives of the Fermat Cubic (*example*), 273
Jacobian Point Not from the Curve (*example*), 180
Jacobian(C) (*function*), 119, 147, 171
Jacobian(model) (*function*), 147
JacobianOrdersByDeformation(Q, Y) (*function*), 187
JacobianPoint(J, D) (*function*), 178
JacobiMotive(A, B) (*function*), 272
jInvariant(E) (*function*), 28
JInvariants(C: parameters) (*function*), 233
JInvariants(f, h: parameters) (*function*), 233
JInvariants(f: parameters) (*function*), 234

K

K ! 0 (*operation*), 219
K ! P (*operation*), 219
K ! [x1, x2, x3, x4] (*operation*), 219
Kedlaya's Algorithm (*example*), 185
Kernel(I) (*function*), 40

KodairaSymbol(E, p) (*function*), 66
KodairaSymbol(s) (*function*), 66
KodairaSymbols(E) (*function*), 66, 130
KummerSurface(J) (*function*), 218
KummerTwist(J, t, rho) (*function*), 272

L

L1 * L2 (*operation*), 304
L1 / L2 (*operation*), 304
L-Function of a Rank 2 Curve (*example*), 135
L-Series from Modular Symbols (*example*), 290
L-Series of a Hilbert Modular Form (*example*), 288
L-Series of a Hyperelliptic Curve (*example*), 286
L-Series of a Rank 1 Curve (*example*), 283
L-Series of an Orthogonal Modular Form (*example*), 288
Large Generator via 5-Descent (*example*), 124
LargestConductor(D) (*function*), 87
LCfRequired(L) (*function*), 299
LegendreModel(C) (*function*), 6
LegendrePolynomial(C) (*function*), 6
LFunction(E) (*function*), 134
LFunction(E, e) (*function*), 134
LFunction(E, S) (*function*), 134
LGetCoefficients(L, N) (*function*), 299
LiftDescendant(C) (*function*), 105
Linear Independence of Points (*example*), 71
Local Data and Mordell-Weil Group (*example*), 100
LocalHeight(P, p) (*function*), 69
LocalHeight(P, Pl : parameters) (*function*), 90
LocalHeight(P, Pl) (*function*), 130
LocalInformation(E) (*function*), 66, 90, 97, 130
LocalInformation(E, f) (*function*), 129
LocalInformation(E, P) (*function*), 89
LocalInformation(E, p) (*function*), 65
LocalInformation(E, Pl) (*function*), 129
LocalTwoSelmerMap(A, P) (*function*), 95
LocalTwoSelmerMap(P) (*function*), 95
Log(Q, P) (*function*), 63
Log(Q, P, t) (*function*), 63
LSeries(A) (*function*), 282
LSeries(C) (*function*), 285, 286
LSeries(C,K) (*function*), 286
LSeries(Chi) (*function*), 287
LSeries(E) (*function*), 283
LSeries(E, A) (*function*), 284
LSeries(E, K) (*function*), 284
LSeries(f) (*function*), 289
LSeries(H, t) (*function*), 257
LSeries(hmf) (*function*), 288
LSeries(HS, N, cffun) (*function*), 295
LSeries(K) (*function*), 280
LSeries(omf) (*function*), 288
LSeries(psi) (*function*), 288
LSeries(S) (*function*), 290
LSeries(weight, gamma, conductor, cffun) (*function*), 294
LSeriesData(L) (*function*), 299
LSetCoefficients(L,cffun) (*function*), 297
LSetPrecision(L,precision) (*function*), 303
LStar(L, s0) (*function*), 291
LTaylor(L,s0,n) (*function*), 291

M

MaedaInvariants(C) (*function*), 236
MaedaInvariants(f) (*function*), 236
ManinConstant(E) (*function*), 76
Matrices(model) (*function*), 143
Matrix(model) (*function*), 143
Mestre's Algorithm (*example*), 185
Minimal Model of a Conic (*example*), 7
Minimal Quadratic Twist (*example*), 27
MinimalDegreeModel(E) (*function*), 130
MinimalModel(C) (*function*), 6
MinimalModel(E) (*function*), 24, 130
MinimalModel(E, P : parameters) (*function*), 24
MinimalModel(E, p) (*function*), 24
MinimalQuadraticTwist(E) (*function*), 27
MinimalWeierstrassModel(C) (*function*), 161
Minimise(f) (*function*), 146
Minimise(model : parameters) (*function*), 145
MinimizeAtP(f, p) (*function*), 162
Minimizing and Reducing a Binary Form (*example*), 163
MinRedBinaryForm(f) (*function*), 162, 163
model1 + model2 (*operation*), 147
ModelToSequence(model) (*function*), 143
ModelToString(model) (*function*), 143
Modular Degree of an Elliptic Curve (*example*), 83
ModularDegree(E) (*function*), 83
ModularParametrisation(E, F : parameters) (*function*), 74
ModularParametrisation(E, f : parameters) (*function*), 74
ModularParametrisation(E, F, B : parameters) (*function*), 74
ModularParametrisation(E, f, B : parameters) (*function*), 74
ModularParametrisation(E, Z : parameters) (*function*), 74
ModularParametrisation(E, z : parameters) (*function*), 74
ModularParametrisation(E, Z, B : parameters) (*function*), 74
ModularParametrisation(E, z, B : parameters) (*function*), 74
ModularParametrization(E, F : parameters) (*function*), 74
ModularParametrization(E, f : parameters) (*function*), 74

ModularParametrization(E, F, B : parameters) (*function*), 74
ModularParametrization(E, f, B : parameters) (*function*), 74
ModularParametrization(E, Z : parameters) (*function*), 74
ModularParametrization(E, z : parameters) (*function*), 74
ModularParametrization(E, Z, B : parameters) (*function*), 74
ModularParametrization(E, z, B : parameters) (*function*), 74
MonicModel(f, q) (*function*), 213
Mordell-Weil Group of a Genus 2 Jacobian (*example*), 203
MordellWeilGroup(E : parameters) (*function*), 132
MordellWeilGroup(E: parameters) (*function*), 99
MordellWeilGroup(H: parameters) (*function*), 99
MordellWeilGroup(J) (*function*), 203
MordellWeilGroupGenus2(J) (*function*), 202
MordellWeilLattice(E) (*function*), 132
MordellWeilRank(E: parameters) (*function*), 99
MordellWeilRank(H: parameters) (*function*), 99
MordellWeilRankBounds(E: parameters) (*function*), 98
MordellWeilRankBounds(H: parameters) (*function*), 98
MordellWeilShaInformation(E: parameters) (*function*), 101
Motive Matching a Siegel Modular Form (*example*), 263
Motives Agreeing Only over Q (*example*), 275
MotivicWeight(L) (*function*), 299
MOV Reduction on a Jacobian (*example*), 182
MOV Reduction with the Weil Pairing (*example*), 60
MultiplicationByMMap(E, m) (*function*), 40
MultiplicationPolynomial(E, alpha) (*function*), 31
MultiplyTransformations(g1, g2) (*function*), 144
MValue(H) (*function*), 255

N

n * P (*operation*), 43, 181, 220
Nagell's Algorithm for a Plane Cubic (*example*), 21
NaiveHeight(P) (*function*), 68, 90, 130, 191
nCovering(model : parameters) (*function*), 147
NegationMap(E) (*function*), 40
Newform over the Gaussian Integers (*example*), 289
Ngens(E) (*function*), 56, 99
Ngens(H) (*function*), 56, 99
NineDescent(C : parameters) (*function*), 121
NineSelmerSet(C) (*function*), 121
Nonquadratic Twists over a Finite Field (*example*), 26
Nontrivial 2-Torsion Point (*example*), 179
NormResidueSymbol(a, b, p) (*function*), 8
NumberOfComponents(K) (*function*), 130
NumberOfCurves(D) (*function*), 87
NumberOfCurves(D, N) (*function*), 87
NumberOfCurves(D, N, i) (*function*), 87
NumberOfGenerators(E) (*function*), 56, 99
NumberOfGenerators(H) (*function*), 56, 99
NumberOfIsogenyClasses(D, N) (*function*), 87
NumberOfPointsAtInfinity(C) (*function*), 168
NumberOfPointsOnSurface(E, e) (*function*), 135
NumbersOfPointsOnSurface(E, e) (*function*), 135

O

Order(E) (*function*), 50
Order(E, r) (*function*), 53
Order(G) (*function*), 33
Order(H) (*function*), 50
Order(H, r) (*function*), 53
Order(J) (*function*), 183
Order(P) (*function*), 46, 182
Order(P, l, u) (*function*), 182
Order(P, l, u, n, m) (*function*), 182
Orthogonal Symmetric Cube (*example*), 314
OrthogonalSymmetrization(L, p) (*function*), 312

P

P * n (*operation*), 43, 181, 220
P *:= n (*operation*), 43, 181
P + Q (*operation*), 43, 181
P +:= Q (*operation*), 43, 181
P / n (*operation*), 44
P /:= n (*operation*), 44
P - Q (*operation*), 43, 178, 181
P -:= Q (*operation*), 43, 181
P eq Q (*operation*), 46, 168, 181, 220
P in E (*operation*), 46
P in H (*operation*), 46
P ne Q (*operation*), 46, 168, 181, 220
p-Adic Heights of Points (*example*), 73
pAdicEllipticLogarithm(P, p: parameters) (*function*), 80
pAdicHeight(P, p) (*function*), 72
pAdicRegulator(S, p) (*function*), 72
Pairings on a BN-Curve (*example*), 58
Parametrisation Matrix of a Conic (*example*), 13
Parametrised Family of 2-Isogenies (*example*), 36
Parametrising a Rational Curve (*example*), 14
Parametrization(C) (*function*), 14
Parametrization(C, P) (*function*), 14
Parametrization(C, p) (*function*), 14

Parametrization(C, p, P) (*function*), 14
ParametrizationMatrix(C) (*function*), 13
ParametrizeOrdinaryCurve(C) (*function*), 14
ParametrizeOrdinaryCurve(C, p) (*function*), 14
ParametrizeOrdinaryCurve(C, p, I) (*function*), 14
ParametrizeRationalNormalCurve(C) (*function*), 14
Parent(f) (*function*), 240
Parent(P) (*function*), 43
Partial Descent on a Cyclic Cover (*example*), 216
Partial L-Function by Hand (*example*), 138
PeriodMapping(A, n) (*function*), 223
Periods with Real Embeddings (*example*), 81
Periods(E, k) (*function*), 79
Periods(E: parameters) (*function*), 79
Periods(M, n) (*function*), 223
PhiSelmerGroup(f, q) (*function*), 213
PicnDescent(f, q) (*function*), 214
pIntegralModel(C, p) (*function*), 161
pIsogenyDescent(C,E1,E2) (*function*), 123
pIsogenyDescent(C,P) (*function*), 123
pIsogenyDescent(C,phi) (*function*), 123
pIsogenyDescent(E,P) (*function*), 123
pIsogenyDescent(E,p) (*function*), 123
pIsogenyDescent(lambda,p) (*function*), 123
PlaneQuarticFromDixmierOhnoInvariants(DO) (*function*), 154
pMinimalWeierstrassModel(C, p) (*function*), 161
pMinimise(f, p) (*function*), 146
pNormalModel(C, p) (*function*), 161
Point Sets over a Finite Field (*example*), 35
Points at Infinity (*example*), 167
Points in a Rank 1 Subgroup (*example*), 210
Points on a Curve and Its Jacobian (*example*), 179
Points on a Curve of Diophantus (*example*), 170
Points on Three Legendre Conics (*example*), 10
Points over a Large Finite Field (*example*), 46
Points(C : parameters) (*function*), 10, 133
Points(C) (*function*), 169
Points(C, x) (*function*), 167
Points(E) (*function*), 42, 55
Points(E, x) (*function*), 42
Points(f, q) (*function*), 212
Points(G) (*function*), 33
Points(H) (*function*), 42, 55
Points(H, x) (*function*), 42
Points(J) (*function*), 183, 188
Points(J, a, d) (*function*), 178
Points(J, P) (*function*), 221
Points(K,[x1, x2, x3]) (*function*), 219
PointsAtInfinity(C) (*function*), 167, 168
PointsAtInfinity(E) (*function*), 42
PointsAtInfinity(H) (*function*), 42
PointSet(E, L) (*function*), 35
PointSet(E, m) (*function*), 35
PointsGenus2(C) (*function*), 169
PointsKnown(C) (*function*), 170
PointsQI(C, B : parameters) (*function*), 113
PointsQI(C, H) (*function*), 133
PolynomialRing(model) (*function*), 143
PossibleHypergeometricData(d) (*function*), 255
pPart(H, p) (*function*), 259
pParts(H) (*function*), 259
pPowerTorsion(E, p) (*function*), 98
PrimitiveData(H) (*function*), 255
Primitivity of Hypergeometric Data (*example*), 261
pSelmerGroup(A, p, S) (*function*), 94
PseudoAdd(P1, P2, P3) (*function*), 220
PseudoAddMultiple(P1, P2, P3, n) (*function*), 220
PushThroughIsogeny(I, G) (*function*), 39
PushThroughIsogeny(I, v) (*function*), 39
P[i] (*literal*), 43, 168, 181, 220

Q

qCoverDescent(f, q) (*function*), 212
qCoverPartialDescent(f, factors, q) (*function*), 216
Quadratic Twist over the Rationals (*example*), 164
Quadratic Twists over a Finite Field (*example*), 25, 164
QuadraticTwist(C) (*function*), 164
QuadraticTwist(C, d) (*function*), 164
QuadraticTwist(E) (*function*), 25
QuadraticTwist(E, d) (*function*), 25
QuadraticTwists(C) (*function*), 164
QuadraticTwists(E) (*function*), 25
QuadricIntersection(C) (*function*), 113
QuadricIntersection(E) (*function*), 113
QuadricIntersection(F) (*function*), 113
QuadricIntersection(model) (*function*), 143
QuadricIntersection(P, F) (*function*), 113
Quartic from Dixmier-Ohno Invariants (*example*), 154
QuarticCovariantsAndContravariants(Phi) (*function*), 153
QuarticG4Covariant(q) (*function*), 105
QuarticG6Covariant(q) (*function*), 105
QuarticHSeminvariant(q) (*function*), 105
QuarticIInvariant(q) (*function*), 105
QuarticJInvariant(q) (*function*), 105
QuarticMinimise(q) (*function*), 105
QuarticMinimize(f) (*function*), 133
QuarticNumberOfRealRoots(q) (*function*), 105
QuarticPSeminvariant(q) (*function*), 105
QuarticQSeminvariant(q) (*function*), 105
QuarticReduce(q) (*function*), 105

QuarticRSeminvariant(q) (*function*), 105
QuaternionAlgebra(C) (*function*), 16

R

Random(C : parameters) (*function*), 9
Random(C) (*function*), 168
Random(D) (*function*), 87
Random(E) (*function*), 55
Random(H) (*function*), 55
Random(J) (*function*), 181
RandomGenusOneModel(n) (*function*), 141
RandomModel(n) (*function*), 141
RandomTransformation(n : parameters) (*function*), 144
Rank 0 via 5-Isogeny Descent (*example*), 124
Rank 2 Curve of Conductor 389 (*example*), 102
Rank 2 Curve over a Function Field (*example*), 132
Rank and 3-Primary Sha by Descent (*example*), 215
Rank Bound Without Local Index One (*example*), 201
Rank via the 2-Selmer Group (*example*), 108
Rank(E: parameters) (*function*), 99
Rank(H: parameters) (*function*), 99
RankBound(E) (*function*), 99, 132
RankBound(f, q) (*function*), 214
RankBound(J) (*function*), 196
RankBounds(E) (*function*), 132
RankBounds(E: parameters) (*function*), 98
RankBounds(f, q) (*function*), 214
RankBounds(H: parameters) (*function*), 98
RankBounds(J) (*function*), 196
RationalCurve(X, f) (*function*), 2
RationalMap(i, t) (*function*), 38
RationalPoint(C) (*function*), 9
RationalPoints(C : parameters) (*function*), 10
RationalPoints(C) (*function*), 169
RationalPoints(C, x) (*function*), 167
RationalPoints(E) (*function*), 42, 55
RationalPoints(f, q) (*function*), 212
RationalPoints(G) (*function*), 33
RationalPoints(H) (*function*), 42, 55
RationalPoints(J) (*function*), 183, 188
RationalPoints(J, a, d) (*function*), 178
RationalPoints(J, P) (*function*), 221
RationalPoints(K, Q) (*function*), 220
RationalPointsGenus2(C) (*function*), 169
RealPeriod(E: parameters) (*function*), 80
Recognising Hypergeometric Motives (*example*), 260
Recovering a Conductor 11 L-Series (*example*), 319
Reduce(f) (*function*), 146
Reduce(model) (*function*), 146
Reduced Basis via Heights (*example*), 194
Reduced Points on a Conic (*example*), 11
ReducedAteTPairing(Q, P, n, q) (*function*), 58
ReducedAutomorphismGroupOfHyperelliptic Curve(f) (*function*), 250
ReducedAutomorphismGroupOfHyperelliptic Curve(f, Autos) (*function*), 249
ReducedAutomorphismGroupOfHyperelliptic Curve(X) (*function*), 250
ReducedAutomorphismGroupOfHyperelliptic Curve(X, Autos) (*function*), 249
ReducedAutomorphismsOfHyperelliptic Curve(f) (*function*), 249
ReducedAutomorphismsOfHyperellipticCurve(X) (*function*), 249
ReducedBasis(S) (*function*), 71
ReducedBasis(S: Precision) (*function*), 192
ReducedEtaTPairing(P, Q, n, q) (*function*), 57
ReducedIsomorphismsOfHyperelliptic Curves(f1, f2) (*function*), 249
ReducedIsomorphismsOfHyperellipticCurves(X1, X2) (*function*), 249
ReducedLegendreModel(C) (*function*), 6
ReducedLegendrePolynomial(C) (*function*), 6
ReducedMinimalWeierstrassModel(C) (*function*), 162
ReducedModel(C) (*function*), 161
ReducedTatePairing(P, Q, n) (*function*), 57
ReduceQuadrics(seq) (*function*), 146
Reducing a Curve Modulo 101 (*example*), 160
Reducing a Set of Generators (*example*), 44
Reduction(E, p) (*function*), 90
Reduction(p) (*function*), 11
ReductionType(E, p) (*function*), 66
Regulator(E) (*function*), 69
Regulator(S) (*function*), 69
Regulator(S: Precision) (*function*), 192
RichelotIsogenousSurface(C, kernel) (*function*), 173
RichelotIsogenousSurface(J, kernel) (*function*), 173
RichelotIsogenousSurfaces(C) (*function*), 173
RichelotIsogenousSurfaces(J) (*function*), 173
Riemann Zeta Function at 2 (*example*), 280
Riemann Zeta with Deferred Coefficients (*example*), 298
RiemannZeta() (*function*), 280
Ring(H) (*function*), 35
Rohrlich's Central Vanishing Twist (*example*), 322
RootNumber(E) (*function*), 82, 95, 97
RootNumber(E, P) (*function*), 95
RootNumber(E, p) (*function*), 82
RootNumber(HS) (*function*), 305
RosenhainInvariants(t) (*function*), 229
RubinSilverbergPolynomials(n, J : parameters) (*function*), 148

S

S-Integral Points on a Desboves Curve (*example*), 86

S-Integral Points on a Rank 2 Curve (*example*), 85
Saturation(bas, p) (*function*), 195
Saturation(points) (*function*), 100
Saturation(points, n) (*function*), 99
Scale(J, q) (*function*), 272
ScaledIgusaInvariants(f) (*function*), 234
ScaledIgusaInvariants(f, h) (*function*), 234
ScalingFactor(g) (*function*), 145
ScalingFactor(n, g) (*function*), 145
Scheme(H) (*function*), 35
Scheme(P) (*function*), 43
Schoof and Halving in Genus 2 (*example*), 186
SEA Point Counting in Characteristic 2 (*example*), 51
SEA(E : parameters) (*function*), 50
SEA(H : parameters) (*function*), 50
Searching Points on a Kummer Surface (*example*), 220
Selmer's Famous Cubic Curve (*example*), 116
SelmerGroup(phi) (*function*), 107
SetBufferSize(D, n) (*procedure*), 87
SetVerbose("CrvHypReduce", v) (*procedure*), 162
SetVerbose("JacHypCnt", v) (*procedure*), 183
SetVerbose("Minimize", v) (*procedure*), 162
SetVerbose("SEA", v) (*procedure*), 53
Shioda Invariants of a Genus 3 Curve (*example*), 236
ShiodaAlgebraicInvariants(FJI) (*function*), 236
ShiodaInvariants(C) (*function*), 235
ShiodaInvariants(f) (*function*), 235
ShiodaInvariants(fh) (*function*), 235
ShiodaInvariantsEqual(V1, V2) (*function*), 236
Sign(L) (*function*), 299
SiksekBound(E: parameters) (*function*), 70
SiksekBound(H: parameters) (*function*), 70
SilvermanBound(E) (*function*), 70
SilvermanBound(H) (*function*), 70
SilvermanHeightBounds(E) (*function*), 91
SimplifiedModel(C) (*function*), 160
SimplifiedModel(E) (*function*), 24
Singular Curve of Genus Zero (*example*), 3
SIntegralDesbovesPoints(Q, S) (*function*), 86
SIntegralLjunggrenPoints(Q, S) (*function*), 86
SIntegralPoints(E, S) (*function*), 84
SIntegralQuarticPoints(Q, S) (*function*), 86
SixDescent(C2, C3) (*function*), 120
SixDescent(model2, model3) (*function*), 120
SL4Invariants(model) (*function*), 148
SmallPeriodMatrix(A) (*function*), 222
Special Motives of Degree 2 (*example*), 269
StableFaltingsHeight(E) (*function*), 68
Subgroup Schemes of a Curve (*example*), 33
SubgroupScheme(G, f) (*function*), 32
SupersingularEllipticCurve(K) (*function*), 21
SupersingularPolynomial(p) (*function*), 49
Sylow(J, p) (*function*), 188
Symmetric Power with Central Vanishing (*example*), 312
Symmetric Powers of a Dirichlet Character (*example*), 311
Symmetric Powers of a Grossencharacter (*example*), 311
Symmetric Powers of an Elliptic Curve (*example*), 311
SymmetricPower(HS, m) (*function*), 306
SymmetricPower(L, m) (*function*), 310
Symmetrization(L, p) (*function*), 312
Symplectic Symmetrizations in Degree 4 (*example*), 315
SymplecticSymmetrization(L, p) (*function*), 312

T

TamagawaNumber(E, p) (*function*), 65
TamagawaNumbers(E) (*function*), 65
Tame Euler Factors and Grossencharacters (*example*), 268
Tame Primes and Artin Representations (*example*), 267
Tame Primes via Jacobi Motives (*example*), 276
Tate-Shafarevich Group via the L-Function (*example*), 134
Tate-Shafarevich Groups of Nonsquare Order (*example*), 198
TatePairing(P, Q, n) (*function*), 57
TateTwist(HS, k) (*function*), 305
TateTwist(J, j) (*function*), 272
Tensor Arithmetic of Jacobi Motives (*example*), 274
Tensor Product and a Siegel Modular Form (*example*), 308
Tensor Product of l-Adic L-Series (*example*), 321
Tensor Product of Level 1 Modular Forms (*example*), 308
Tensor Product of Two Elliptic Curves (*example*), 308
Tensor Product over a Quadratic Field (*example*), 309
TensorProduct(H1, H2) (*function*), 306
TensorProduct(L1, L2) (*function*), 307
TensorProduct(L1, L2, ExcFactors) (*function*), 307
TensorProduct(L1, L2, ExcFactors, K) (*function*), 307
TensorProduct(L1, L2, K) (*function*), 307
TernaryQuarticFromDixmierOhnoInvariants(DO) (*function*), 154
Testing a Curve for Rationality (*example*), 4
Testing Coefficients for Elliptic Curves (*example*), 23
Testing Curves for Isomorphism (*example*), 37
The Quintic Threefold (*example*), 265

Three Constructions of One Curve (*example*), 20
Three Richelot Isogenies (*example*), 173
Three-Descent by Isogeny (*example*), 119
Three-Division Points (*example*), 44
ThreeDescent(E : parameters) (*function*), 116
ThreeDescentByIsogeny(E) (*function*), 118
ThreeDescentCubic(E, α : parameters) (*function*), 117
ThreeIsogenyDescent(E : parameters) (*function*), 118
ThreeIsogenyDescentCubic(φ, α) (*function*), 118
ThreeIsogenySelmerGroups(E : parameters) (*function*), 118
ThreeSelmerElement(C) (*function*), 119
ThreeSelmerElement(E, C) (*function*), 119
ThreeSelmerGroup(E : parameters) (*function*), 117
ThreeTorsionMatrices(E, C) (*function*), 120
ThreeTorsionPoints(E : parameters) (*function*), 120
ThreeTorsionType(E) (*function*), 120
To2DUpperHalfSpaceFundamentalDomain(z) (*function*), 225
ToAnalyticJacobian(x, y, A) (*function*), 223
ToAnalyticJacobianMumford(pt, AJ) (*function*), 228
ToAnalyticJacobianMumford(pt, AJ, conj) (*function*), 228
Torsion on a Jacobian (*example*), 189
TorsionBound(E, n) (*function*), 98, 131
TorsionBound(E, n, B) (*function*), 131
TorsionBound(J, n) (*function*), 189
TorsionSubgroup(E) (*function*), 55, 98, 131
TorsionSubgroup(H) (*function*), 55, 98
TorsionSubgroup(J) (*function*), 189
TorsionSubgroupScheme(G, n) (*function*), 32
Trace(E) (*function*), 53
Trace(E, r) (*function*), 53
Trace(H) (*function*), 53
Trace(H, r) (*function*), 53
TraceOfFrobenius(E) (*function*), 53
TraceOfFrobenius(E, p) (*function*), 130
TraceOfFrobenius(E, r) (*function*), 53
TraceOfFrobenius(H, r) (*function*), 53
TraceOfFrobeniusDirect(E, p) (*function*), 66
Traces of Frobenius (*example*), 66
TracesOfFrobenius(E, B) (*function*), 66
TracesOfFrobenius(E, B0, B1) (*function*), 66
TracesOfFrobenius(E, Ps) (*function*), 66
Transformation(C, e) (*function*), 240
Transformation(C, e, u) (*function*), 240
Transformation(C, t) (*function*), 240
Transformation(C, t, e, u) (*function*), 240
Transformation(C, u) (*function*), 240
Transforming a Hyperelliptic Curve (*example*), 240
Translate(L, z) (*function*), 305
TranslationMap(E, P) (*function*), 38
Tuple(g) (*function*), 144
TwelveDescent(C3, C4) (*function*), 121
TwelveDescent(model3, model4) (*function*), 121
Twist by a Quaternion Representation (*example*), 285
Twist(H) (*function*), 255
TwistedHyperellipticPolynomialsFromShioda Invariants(JI: parameters) (*function*), 246
Twisting by Cyclotomic Characters (*example*), 285
Twisting Hypergeometric Data (*example*), 261
Twists and Trace of Frobenius (*example*), 54
Twists Isomorphic over an Extension (*example*), 30
Twists of a Supersingular Curve (*example*), 248
Twists of the Klein Quartic (*example*), 156
Twists over a Finite Field (*example*), 247
Twists(C) (*function*), 156
Twists(C, Autos: parameters) (*function*), 247
Twists(C: parameters) (*function*), 247
Twists(E) (*function*), 25
TwistsFromG2Invariants(JI: parameters) (*function*), 247
TwistsFromIgusaInvariants(JI: parameters) (*function*), 247
TwistsFromShiodaInvariants(JI: parameters) (*function*), 247
TwistsOfHyperellipticPolynomials(f: parameters) (*function*), 246
TwistsOfHyperellipticPolynomials(fh) (*function*), 246
TwistsOfPlaneQuartic(C, Autos) (*function*), 156
Two-Cover Descent on Genus 2 Curves (*example*), 205
Two-Descent on an Elliptic Curve (*example*), 104
Two-Selmer Groups and Two-Covers (*example*), 110
TwoCover(e) (*function*), 104
TwoCoverDescent(C) (*function*), 205
TwoCoverPullback(f, pt) (*function*), 114
TwoCoverPullback(H, pt) (*function*), 114
TwoDescendantsOverTwoIsogenyDescendant(C) (*function*), 105
TwoDescent(E) (*function*), 133
TwoDescent(E: parameters) (*function*), 103
TwoIsogeny(P) (*function*), 38
TwoIsogenyDescent(E : parameters) (*function*), 104
TwoIsogenySelmerGroups(E) (*function*), 134
TwoPowerIsogenies(J) (*function*), 174
TwoPowerIsogenyDescentRankBound(E : parameters) (*function*), 122

TwoPowerIsogenyDescentRankBound(E, T : parameters) (*function*), 122
TwoSelmerGroup(E) (*function*), 107, 133
TwoSelmerGroup(J) (*function*), 195
TwoTorsionPolynomial(E) (*function*), 30
TwoTorsionSubgroup(E) (*function*), 98
TwoTorsionSubgroup(J) (*function*), 189
Type(C) (*function*), 3
Type(E) (*function*), 29
Type(G) (*function*), 32
Type(H) (*function*), 35
Type(P) (*function*), 43

V

Vercauteren's Algorithm (*example*), 185
Visibility of the Tate-Shafarevich Group (*example*), 199
Voronoi(sites) (*function*), 231

W

WeierstrassModel(E) (*function*), 24
Weight(H) (*function*), 255
Weight(HS) (*function*), 305
Weight(J) (*function*), 273
Weil Descent to a Genus 31 Curve (*example*), 62
Weil Pairing of 3-Torsion Points (*example*), 47
WeilDescent(E, k, c) (*function*), 61
WeilDescentDegree(E, k, c) (*function*), 62
WeilDescentGenus(E, k, c) (*function*), 62
WeilHeight(P) (*function*), 68
WeilPairing(P, Q, m) (*function*), 182
WeilPairing(P, Q, n) (*function*), 47, 56

Z

Zeta Function of an Elliptic Curve (*example*), 55
ZetaFunction(E) (*function*), 55
ZetaFunctionsByDeformation(Q, Y) (*function*), 187