

HANDBOOK OF MAGMA FUNCTIONS

Volume 10

Arithmetic Geometry and Number Theory

John Cannon Wieb Bosma

Claus Fieker Allan Steel

Editors

Version 2.19

Sydney

December 17, 2012

HANDBOOK OF MAGMA FUNCTIONS

Editors:

John Cannon Wieb Bosma Claus Fieker Allan Steel

Handbook Contributors:

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

Production Editors:

Wieb Bosma Claus Fieker Allan Steel Nicole Sutherland

HTML Production:

Claus Fieker Allan Steel

VOLUME 10: OVERVIEW

XVI	ARITHMETIC GEOMETRY	3901
119	RATIONAL CURVES AND CONICS	3903
120	ELLIPTIC CURVES	3927
121	ELLIPTIC CURVES OVER FINITE FIELDS	3969
122	ELLIPTIC CURVES OVER \mathbf{Q} AND NUMBER FIELDS	3993
123	ELLIPTIC CURVES OVER FUNCTION FIELDS	4075
124	MODELS OF GENUS ONE CURVES	4093
125	HYPERELLIPTIC CURVES	4111
126	HYPERGEOMETRIC MOTIVES	4215
127	L-FUNCTIONS	4235

VOLUME 10: CONTENTS

XVI	ARITHMETIC GEOMETRY	3901
119	RATIONAL CURVES AND CONICS	3903
119.1	<i>Introduction</i>	3905
119.2	<i>Rational Curves and Conics</i>	3906
119.2.1	Rational Curve and Conic Creation	3906
119.2.2	Access Functions	3907
119.2.3	Rational Curve and Conic Examples	3908
119.3	<i>Conics</i>	3911
119.3.1	Elementary Invariants	3911
119.3.2	Alternative Defining Polynomials	3911
119.3.3	Alternative Models	3912
119.3.4	Other Functions on Conics	3912
119.4	<i>Local-Global Correspondence</i>	3913
119.4.1	Local Conditions for Conics	3913
119.4.2	Norm Residue Symbol	3913
119.5	<i>Rational Points on Conics</i>	3915
119.5.1	Finding Points	3915
119.5.2	Point Reduction	3917
119.6	<i>Isomorphisms</i>	3919
119.6.1	Isomorphisms with Standard Models	3919
119.7	<i>Automorphisms</i>	3923
119.7.1	Automorphisms of Rational Curves	3923
119.7.2	Automorphisms of Conics	3924
119.8	<i>Bibliography</i>	3926
120	ELLIPTIC CURVES	3927
120.1	<i>Introduction</i>	3931
120.2	<i>Creation Functions</i>	3932
120.2.1	Creation of an Elliptic Curve	3932
120.2.2	Creation Predicates	3935
120.2.3	Changing the Base Ring	3936
120.2.4	Alternative Models	3937
120.2.5	Predicates on Curve Models	3938
120.2.6	Twists of Elliptic Curves	3939
120.3	<i>Operations on Curves</i>	3942
120.3.1	Elementary Invariants	3942
120.3.2	Associated Structures	3945
120.3.3	Predicates on Elliptic Curves	3945
120.4	<i>Polynomials</i>	3946
120.5	<i>Subgroup Schemes</i>	3947
120.5.1	Creation of Subgroup Schemes	3947
120.5.2	Associated Structures	3948
120.5.3	Predicates on Subgroup Schemes	3948
120.5.4	Points of Subgroup Schemes	3948
120.6	<i>The Formal Group</i>	3949
120.7	<i>Operations on Point Sets</i>	3950
120.7.1	Creation of Point Sets	3950
120.7.2	Associated Structures	3951

120.7.3	Predicates on Point Sets	3951
120.8	<i>Morphisms</i>	3952
120.8.1	Creation Functions	3952
120.8.2	Predicates on Isogenies	3957
120.8.3	Structure Operations	3957
120.8.4	Endomorphisms	3958
120.8.5	Automorphisms	3959
120.9	<i>Operations on Points</i>	3959
120.9.1	Creation of Points	3959
120.9.2	Creation Predicates	3960
120.9.3	Access Operations	3961
120.9.4	Associated Structures	3961
120.9.5	Arithmetic	3961
120.9.6	Division Points	3962
120.9.7	Point Order	3965
120.9.8	Predicates on Points	3965
120.9.9	Weil Pairing	3967
120.10	<i>Bibliography</i>	3968
121	ELLIPTIC CURVES OVER FINITE FIELDS	3969
121.1	<i>Supersingular Curves</i>	3971
121.2	<i>The Order of the Group of Points</i>	3972
121.2.1	Point Counting	3972
121.2.2	Zeta Functions	3978
121.2.3	Cryptographic Elliptic Curve Domains	3979
121.3	<i>Enumeration of Points</i>	3980
121.4	<i>Abelian Group Structure</i>	3981
121.5	<i>Pairings on Elliptic Curves</i>	3982
121.5.1	Weil Pairing	3982
121.5.2	Tate Pairing	3982
121.5.3	Eta Pairing	3983
121.5.4	Ate Pairing	3984
121.6	<i>Weil Descent in Characteristic Two</i>	3988
121.7	<i>Discrete Logarithms</i>	3990
121.8	<i>Bibliography</i>	3991
122	ELLIPTIC CURVES OVER \mathbb{Q} AND NUMBER FIELDS	3993
122.1	<i>Introduction</i>	3997
122.2	<i>Curves over the Rationals</i>	3997
122.2.1	Local Invariants	3997
122.2.2	Kodaira Symbols	3999
122.2.3	Complex Multiplication	4000
122.2.4	Isogenous Curves	4000
122.2.5	Mordell–Weil Group	4001
122.2.6	Heights and Height Pairing	4007
122.2.7	Two-Descent and Two-Coverings	4013
122.2.8	The Cassels-Tate Pairing	4016
122.2.9	Four-Descent	4018
122.2.10	Eight-Descent	4022
122.2.11	Three-Descent	4023
122.2.12	Nine-Descent	4030
122.2.13	p -Isogeny Descent	4031
122.2.14	Heegner Points	4035
122.2.15	Analytic Information	4042

122.2.16	Integral and S -integral Points	4047
122.2.17	Elliptic Curve Database	4050
122.3	<i>Curves over Number Fields</i>	4054
122.3.1	Local Invariants	4054
122.3.2	Complex Multiplication	4055
122.3.3	Mordell–Weil Groups	4055
122.3.4	Heights	4056
122.3.5	Two Descent	4057
122.3.6	Selmer Groups	4057
122.3.7	The Cassels–Tate Pairing	4063
122.3.8	Elliptic Curve Chabauty	4063
122.3.9	Auxiliary Functions for Etale Algebras	4067
122.3.10	Analytic Information	4068
122.3.11	Elliptic Curves of Given Conductor	4069
122.4	<i>Curves over p-adic Fields</i>	4070
122.4.1	Local Invariants	4070
122.5	<i>Bibliography</i>	4071
123	ELLIPTIC CURVES OVER FUNCTION FIELDS	4075
123.1	<i>An Overview of Relevant Theory</i>	4077
123.2	<i>Local Computations</i>	4079
123.3	<i>Elliptic Curves of Given Conductor</i>	4080
123.4	<i>Heights</i>	4081
123.5	<i>The Torsion Subgroup</i>	4082
123.6	<i>The Mordell–Weil Group</i>	4082
123.7	<i>Two Descent</i>	4084
123.8	<i>The L-function and Counting Points</i>	4085
123.9	<i>Action of Frobenius</i>	4088
123.10	<i>Extended Examples</i>	4088
123.11	<i>Bibliography</i>	4091
124	MODELS OF GENUS ONE CURVES	4093
124.1	<i>Introduction</i>	4095
124.2	<i>Related Functionality</i>	4096
124.3	<i>Creation of Genus One Models</i>	4096
124.4	<i>Predicates on Genus One Models</i>	4099
124.5	<i>Access Functions</i>	4099
124.6	<i>Minimisation and Reduction</i>	4100
124.7	<i>Genus One Models as Coverings</i>	4102
124.8	<i>Families of Elliptic Curves with Prescribed n-Torsion</i>	4104
124.9	<i>Transformations between Genus One Models</i>	4104
124.10	<i>Invariants for Genus One Models</i>	4105
124.11	<i>Covariants and Contravariants for Genus One Models</i>	4106
124.12	<i>Examples</i>	4107
124.13	<i>Bibliography</i>	4109

125	HYPERELLIPTIC CURVES	4111
125.1	<i>Introduction</i>	4115
125.2	<i>Creation Functions</i>	4115
125.2.1	Creation of a Hyperelliptic Curve	4115
125.2.2	Creation Predicates	4116
125.2.3	Changing the Base Ring	4117
125.2.4	Models	4118
125.2.5	Predicates on Models	4120
125.2.6	Twisting Hyperelliptic Curves	4121
125.2.7	Type Change Predicates	4123
125.3	<i>Operations on Curves</i>	4123
125.3.1	Elementary Invariants	4124
125.3.2	Igusa Invariants	4124
125.3.3	Shioda Invariants	4128
125.3.4	Base Ring	4130
125.4	<i>Creation from Invariants</i>	4130
125.5	<i>Function Field</i>	4133
125.5.1	Function Field and Polynomial Ring	4133
125.6	<i>Points</i>	4133
125.6.1	Creation of Points	4133
125.6.2	Random Points	4135
125.6.3	Predicates on Points	4135
125.6.4	Access Operations	4135
125.6.5	Arithmetic of Points	4135
125.6.6	Enumeration and Counting Points	4136
125.6.7	Frobenius	4137
125.7	<i>Isomorphisms and Transformations</i>	4138
125.7.1	Creation of Isomorphisms	4138
125.7.2	Arithmetic with Isomorphisms	4139
125.7.3	Invariants of Isomorphisms	4140
125.7.4	Automorphism Group and Isomorphism Testing	4140
125.8	<i>Jacobians</i>	4145
125.8.1	Creation of a Jacobian	4145
125.8.2	Access Operations	4145
125.8.3	Base Ring	4145
125.8.4	Changing the Base Ring	4146
125.9	<i>Richelot Isogenies</i>	4146
125.10	<i>Points on the Jacobian</i>	4149
125.10.1	Creation of Points	4150
125.10.2	Random Points	4153
125.10.3	Booleans and Predicates for Points	4153
125.10.4	Access Operations	4154
125.10.5	Arithmetic of Points	4154
125.10.6	Order of Points on the Jacobian	4155
125.10.7	Frobenius	4155
125.10.8	Weil Pairing	4156
125.11	<i>Rational Points and Group Structure over Finite Fields</i>	4157
125.11.1	Enumeration of Points	4157
125.11.2	Counting Points on the Jacobian	4157
125.11.3	Deformation Point Counting	4162
125.11.4	Abelian Group Structure	4163
125.12	<i>Jacobians over Number Fields or \mathbf{Q}</i>	4164
125.12.1	Searching For Points	4164
125.12.2	Torsion	4164
125.12.3	Heights and Regulator	4166
125.12.4	The 2-Selmer Group	4171

125.13	<i>Two-Selmer Set of a Curve</i>	4179
125.14	<i>Chabauty's Method</i>	4182
125.15	<i>Cyclic Covers of \mathbf{P}^1</i>	4187
125.15.1	Points	4187
125.15.2	Descent	4188
125.15.3	Descent on the Jacobian	4189
125.15.4	Partial Descent	4192
125.16	<i>Kummer Surfaces</i>	4195
125.16.1	Creation of a Kummer Surface	4195
125.16.2	Structure Operations	4195
125.16.3	Base Ring	4195
125.16.4	Changing the Base Ring	4196
125.17	<i>Points on the Kummer Surface</i>	4196
125.17.1	Creation of Points	4196
125.17.2	Access Operations	4197
125.17.3	Predicates on Points	4197
125.17.4	Arithmetic of Points	4197
125.17.5	Rational Points on the Kummer Surface	4198
125.17.6	Pullback to the Jacobian	4198
125.18	<i>Analytic Jacobians of Hyperelliptic Curves</i>	4199
125.18.1	Creation and Access Functions	4200
125.18.2	Maps between Jacobians	4201
125.18.3	From Period Matrix to Curve	4208
125.18.4	Voronoi Cells	4210
125.19	<i>Bibliography</i>	4211
126	HYPERGEOMETRIC MOTIVES	4215
126.1	<i>Introduction</i>	4217
126.2	<i>Functionality</i>	4219
126.2.1	Creation Functions	4219
126.2.2	Access Functions	4220
126.2.3	Functionality with L -series and Euler Factors	4221
126.2.4	Associated Schemes and Curves	4224
126.2.5	Utility Functions	4224
126.3	<i>Examples</i>	4225
126.4	<i>Bibliography</i>	4233
127	L-FUNCTIONS	4235
127.1	<i>Overview</i>	4237
127.2	<i>Built-in L-series</i>	4238
127.3	<i>Computing L-values</i>	4249
127.4	<i>Arithmetic with L-series</i>	4251
127.5	<i>General L-series</i>	4252
127.5.1	Terminology	4253
127.5.2	Constructing a General L -Series	4254
127.5.3	Setting the Coefficients	4258
127.5.4	Specifying the Coefficients Later	4258
127.5.5	Generating the Coefficients from Local Factors	4260
127.6	<i>Accessing the Invariants</i>	4260
127.7	<i>Precision</i>	4263
127.7.1	L -series with Unusual Coefficient Growth	4264
127.7.2	Computing $L(s)$ when $\text{Im}(s)$ is Large (ImS Parameter)	4264
127.7.3	Implementation of L -series Computations (Asymptotics Parameter)	4264
127.8	<i>Verbose Printing</i>	4265

127.9	<i>Advanced Examples</i>	4265
127.9.1	Handmade L -series of an Elliptic Curve	4265
127.9.2	Self-made Dedekind Zeta Function	4266
127.9.3	L -series of a Genus 2 Hyperelliptic Curve	4266
127.9.4	Experimental Mathematics for Small Conductor	4268
127.9.5	Tensor Product of L -series Coming from l -adic Representations	4269
127.9.6	Non-abelian Twist of an Elliptic Curve	4270
127.9.7	Other Tensor Products	4271
127.9.8	Symmetric Powers	4273
127.10	<i>Weil Polynomials</i>	4276
127.11	<i>Bibliography</i>	4279

PART XVI

ARITHMETIC GEOMETRY

119	RATIONAL CURVES AND CONICS	3903
120	ELLIPTIC CURVES	3927
121	ELLIPTIC CURVES OVER FINITE FIELDS	3969
122	ELLIPTIC CURVES OVER \mathbf{Q} AND NUMBER FIELDS	3993
123	ELLIPTIC CURVES OVER FUNCTION FIELDS	4075
124	MODELS OF GENUS ONE CURVES	4093
125	HYPERELLIPTIC CURVES	4111
126	HYPERGEOMETRIC MOTIVES	4215
127	L-FUNCTIONS	4235

119 RATIONAL CURVES AND CONICS

119.1 Introduction	3905	<code>NormResidueSymbol(a, b, p)</code>	3913
119.2 Rational Curves and Conics	3906	<code>HilbertSymbol(a, b, p : -)</code>	3914
<i>119.2.1 Rational Curve and Conic Creation</i>	<i>3906</i>	119.5 Rational Points on Conics .	3915
<code>Conic(coeffs)</code>	3906	<i>119.5.1 Finding Points</i>	<i>3915</i>
<code>Conic(M)</code>	3906	<code>HasRationalPoint(C)</code>	3916
<code>Conic(X, f)</code>	3906	<code>RationalPoint(C)</code>	3916
<code>IsConic(S)</code>	3906	<code>Random(C : -)</code>	3916
<code>RationalCurve(X, f)</code>	3906	<code>Points(C : -)</code>	3916
<code>IsRationalCurve(S)</code>	3906	<code>RationalPoints(C : -)</code>	3916
<i>119.2.2 Access Functions</i>	<i>3907</i>	<i>119.5.2 Point Reduction</i>	<i>3917</i>
<code>DefiningPolynomial(C)</code>	3907	<code>IsReduced(p)</code>	3917
<code>DefiningIdeal(C)</code>	3907	<code>Reduction(p)</code>	3917
<code>BaseRing BaseField</code>	3907	119.6 Isomorphisms	3919
<code>Category Type</code>	3907	<i>119.6.1 Isomorphisms with Standard Mod-</i>	
<i>119.2.3 Rational Curve and Conic Exam-</i>		<i>els</i>	<i>3919</i>
<i>ples</i>	<i>3908</i>	<code>Conic(C)</code>	3920
119.3 Conics	3911	<code>ParametrizationMatrix(C)</code>	3920
<i>119.3.1 Elementary Invariants</i>	<i>3911</i>	<code>Parametrization(C)</code>	3921
<code>Discriminant(C)</code>	3911	<code>Parametrization(C, P)</code>	3921
<i>119.3.2 Alternative Defining Polynomials</i>	<i>3911</i>	<code>Parametrization(C, p)</code>	3921
<code>LegendrePolynomial(C)</code>	3911	<code>Parametrization(C, p, P)</code>	3921
<code>ReducedLegendrePolynomial(C)</code>	3911	<code>ParametrizeOrdinaryCurve(C)</code>	3922
<i>119.3.3 Alternative Models</i>	<i>3912</i>	<code>ParametrizeOrdinaryCurve(C, p)</code>	3922
<code>LegendreModel(C)</code>	3912	<code>ParametrizeOrdinaryCurve(C, p, I)</code>	3922
<code>ReducedLegendreModel(C)</code>	3912	<code>ParametrizeRationalNormalCurve(C)</code>	3922
<i>119.3.4 Other Functions on Conics</i> . . .	<i>3912</i>	119.7 Automorphisms	3923
<code>MinimalModel(C)</code>	3912	<i>119.7.1 Automorphisms of Rational Curves</i>	<i>3923</i>
119.4 Local-Global Correspondence	3913	<code>Automorphism(C, S, T)</code>	3923
<i>119.4.1 Local Conditions for Conics</i> . .	<i>3913</i>	<i>119.7.2 Automorphisms of Conics</i> . . .	<i>3924</i>
<code>BadPrimes(C)</code>	3913	<code>QuaternionAlgebra(C)</code>	3924
<i>119.4.2 Norm Residue Symbol</i>	<i>3913</i>	<code>Automorphism(C, a)</code>	3924
		119.8 Bibliography	3926

Chapter 119

RATIONAL CURVES AND CONICS

119.1 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 \mathbf{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 [Sim05] 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.

119.2 Rational Curves and Conics

The general tools for constructing and analysing curves are described in Chapter 114. We do not repeat them here, but rather give some examples in Section 119.2.3 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.

119.2.1 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)

Ambient

SCH

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)

Ambient

SCH

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)

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

IsConic(S)

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)

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

IsRationalCurve(S)

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

Example H119E1

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*y^2 - z^2
> bool, C1 := IsConic(C0);
```

Clearly this is a nonsingular degree two curve, so `bool` must be `true` and we have created a new curve C_1 in the same ambient space P_2 but of conic type.

```
> 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
> C0 eq C1;
false
> Type(C0);
CrvPln
> Type(C1);
CrvCon
```

The equality test return `false` because the two objects are of different MAGMA type.

119.2.2 Access Functions

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

<code>DefiningPolynomial(C)</code>

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

<code>DefiningIdeal(C)</code>

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

<code>BaseRing(C)</code>

<code>BaseField(C)</code>

Returns the base ring of the curve C .

<code>Category(C)</code>

<code>Type(C)</code>

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`).

119.2.3 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 H119E2

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 ];
[
{@ (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 119.6.

```
> 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 114.9 of Chapter 114 for background on places of curves). We can verify this explicitly.

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

Example H119E3

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 \mathbf{Q} and use efficient algorithms of Section 119.5.1 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;
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 H119E4

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 - 186042605936505203884941*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
```

119.3 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.

119.3.1 Elementary Invariants

Discriminant(C)

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

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

119.3.2 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)

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)

Returns the reduced Legendre polynomial of the conic C , which must be defined over \mathbf{Q} or \mathbf{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.

119.3.3 Alternative Models

`LegendreModel(C)`

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)`

Returns the reduced Legendre model of the conic C , which must be defined over \mathbf{Q} or \mathbf{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.

119.3.4 Other Functions on Conics

`MinimalModel(C)`

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 [Sim05], as used in `HasRationalPoint`. A map from the conic to C is also returned.

Example H119E5

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
```

119.4 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 \mathbf{Q} .

119.4.1 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)

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/\mathbf{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.

119.4.2 Norm Residue Symbol

Hilbert’s norm residue symbol logically pertains to the theory of quadratic forms and lattices, and to that of quadratic fields and their norm equations. We express it here for its relevance to determining conditions for the existence of local points on conics over \mathbf{Q} . The norm residue symbol gives a precise condition for a quadratic form to represent zero over \mathbf{Q}_p , or more generally over any local field; this is equivalent to the condition that a conic has a point 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)

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 \mathbf{Q}_p and returns -1 otherwise.

HilbertSymbol(a, b, p : parameters)

A1

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 $a, b \in \mathbf{Q}$) or a prime ideal. For $a, b \in \mathbf{Q}$, 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 `A1` to "Evaluate".

Example H119E6

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 ]
```

119.5 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 $C(\mathbf{Q})$.

Over the rationals (or integers), the algorithm used to find rational points is due to D. Simon [Sim05]. 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/\mathbf{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 \mathbf{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 .

119.5.1 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)

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)

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)

Bound	RNGINTELT	<i>Default</i> : 10^9
Reduce	BOOLELT	<i>Default</i> : <code>false</code>

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 119.5.2 for more information about point reduction.)

Points(C : parameters)

RationalPoints(C : parameters)

Bound	RNGINTELT
--------------	-----------

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 H119E7

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 119.6, an alternative approach which is guaranteed to find points on the third curve would be to construct a rational parametrisation.

119.5.2 Point Reduction

If a conic C/\mathbf{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 \mathbf{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)`

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)`

Returns a Holzer-reduced point derived from p .

Example H119E8

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 H119E9

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
```

```

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

```

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)

```

119.6 Isomorphisms

119.6.1 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 \mathbf{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.

An isomorphism — provided that one exists — of the projective line with a conic can be described as follows. The 2-uple embedding $\phi : \mathbf{P}^1 \rightarrow \mathbf{P}^2$ defined by $(u : v) \mapsto (u^2 : uv : v^2)$

gives an isomorphism of \mathbf{P}^1 with the conic C_0 with defining equation $y^2 = xz$. The inverse isomorphism $C_0 \rightarrow \mathbf{P}^1$ is defined by the maps

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

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 \mathbf{P}^1 it is then necessary and sufficient to give a change of variables which maps $C_0 = \phi(\mathbf{P}^1)$ onto the conic C_1 . This matrix is called the *parametrisation matrix* and is stored with C_1 once a rational point is found.

Conic(C)

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

Example H119E10

We demonstrate the function `Conic` on the curve of Example H119E2 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)

This function is an optimised routine for parametrising a conic C defined over \mathbf{Z} or \mathbf{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 H119E11

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)

Parametrization(C, P)

Parametrization(C, p)

Parametrization(C, p, P)

Given a conic curve C over a general field, these functions return a parametrisation as an isomorphism of schemes $P \rightarrow C$. 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)

ParametrizeOrdinaryCurve(C, p)

ParametrizeOrdinaryCurve(C, p, I)

ParametrizeRationalNormalCurve(C)

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

When C is a plane curve with only ordinary singularities (see subsection 114.3.6) 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., non-singular 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 H119E12

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
> Image(phi) eq C;
false
> DefiningIdeal(Image(phi)) eq DefiningIdeal(C);
true
> phi(P1![1, 0]);
(2 : 33 : 1)
```

119.7 Automorphisms

119.7.1 Automorphisms of Rational Curves

Automorphisms of the projective line \mathbf{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)

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_∞ .

119.7.2 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 $\mathbf{P}^2(L)$ with the projectivisation of the trace zero part

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

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

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

in $\mathbf{P}^2(L)$. This allows us to identify the automorphism group $\text{Aut}_K(C)$ 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 $A \cong M_2(K)$.

We note that while the advanced features of finding rational points (and hence finding parametrisations) and determining isomorphism only exist over \mathbf{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)`

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

`Automorphism(C, a)`

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

Example H119E13

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]) ];
[ 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 $\mathbf{P}^2(K)$.

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 \mathbf{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.

119.8 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.
- [Sim05] 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.

120 ELLIPTIC CURVES

120.1 Introduction	3931		
120.2 Creation Functions	3932		
120.2.1 <i>Creation of an Elliptic Curve</i>	3932		
EllipticCurve([a, b])	3932		
EllipticCurve([a1, a2, a3, a4, a6])	3932		
EllipticCurve(f)	3932		
EllipticCurve(f, h)	3932		
EllipticCurveFromjInvariant(j)	3932		
EllipticCurveWithjInvariant(j)	3932		
EllipticCurve(C)	3933		
EllipticCurve(C, P)	3934		
EllipticCurve(C, pl)	3934		
SupersingularEllipticCurve(K)	3934		
120.2.2 <i>Creation Predicates</i>	3935		
IsEllipticCurve([a, b])	3935		
IsEllipticCurve([a1, a2, a3, a4, a6])	3935		
IsEllipticCurve(C)	3936		
120.2.3 <i>Changing the Base Ring</i>	3936		
BaseChange(E, K)	3936		
BaseExtend(E, K)	3936		
ChangeRing(E, K)	3936		
BaseChange(E, h)	3936		
BaseExtend(E, h)	3936		
BaseChange(E, n)	3937		
BaseExtend(E, n)	3937		
120.2.4 <i>Alternative Models</i>	3937		
WeierstrassModel(E)	3937		
IntegralModel(E)	3937		
SimplifiedModel(E)	3938		
MinimalModel(E)	3938		
MinimalModel(E, p)	3938		
MinimalModel(E, P : -)	3938		
120.2.5 <i>Predicates on Curve Models</i>	3938		
IsWeierstrassModel(E)	3938		
IsIntegralModel(E)	3938		
IsSimplifiedModel(E)	3938		
IsMinimalModel(E)	3938		
IsIntegralModel(E, P)	3939		
120.2.6 <i>Twists of Elliptic Curves</i>	3939		
QuadraticTwist(E, d)	3939		
QuadraticTwist(E)	3939		
QuadraticTwists(E)	3940		
Twists(E)	3940		
IsTwist(E, F)	3940		
IsQuadraticTwist(E, F)	3940		
MinimalQuadraticTwist(E)	3942		
120.3 Operations on Curves	3942		
120.3.1 <i>Elementary Invariants</i>	3942		
aInvariants(E)	3942		
Coefficients(E)	3942		
ElementToSequence(E)	3942		
Eltseq(E)	3942		
bInvariants(E)	3943		
cInvariants(E)	3943		
Discriminant(E)	3943		
jInvariant(E)	3943		
HyperellipticPolynomials(E)	3943		
120.3.2 <i>Associated Structures</i>	3945		
Category(E)	3945		
Type(E)	3945		
BaseRing(E)	3945		
CoefficientRing(E)	3945		
120.3.3 <i>Predicates on Elliptic Curves</i>	3945		
eq	3945		
ne	3945		
IsIsomorphic(E, F)	3945		
IsIsogenous(E, F)	3945		
120.4 Polynomials	3946		
DefiningPolynomial(E)	3946		
DivisionPolynomial(E, n)	3946		
DivisionPolynomial(E, n, g)	3946		
TwoTorsionPolynomial(E)	3946		
120.5 Subgroup Schemes	3947		
120.5.1 <i>Creation of Subgroup Schemes</i>	3947		
SubgroupScheme(G, f)	3947		
TorsionSubgroupScheme(G, n)	3947		
120.5.2 <i>Associated Structures</i>	3948		
Category(G)	3948		
Type(G)	3948		
Curve(G)	3948		
Generic(G)	3948		
BaseRing(G)	3948		
CoefficientRing(G)	3948		
DefiningSubschemePolynomial(G)	3948		
120.5.3 <i>Predicates on Subgroup Schemes</i>	3948		
eq	3948		
ne	3948		
120.5.4 <i>Points of Subgroup Schemes</i>	3948		
#	3948		
Order(G)	3948		
FactoredOrder(G)	3948		
Points(G)	3948		
RationalPoints(G)	3948		
120.6 The Formal Group	3949		
FormalGroupLaw(E, prec)	3949		
FormalGroupHomomorphism(phi, prec)	3950		
FormalLog(E)	3950		

120.7 Operations on Point Sets	3950	<i>120.8.5 Automorphisms</i>	<i>3959</i>
<i>120.7.1 Creation of Point Sets</i>	<i>3950</i>	AutomorphismGroup(E)	3959
E(L)	3950	Automorphisms(E)	3959
PointSet(E, L)	3950	120.9 Operations on Points	3959
E(m)	3950	<i>120.9.1 Creation of Points</i>	<i>3959</i>
PointSet(E, m)	3950	! elt	3959
<i>120.7.2 Associated Structures</i>	<i>3951</i>	! elt	3959
Category(H)	3951	! Id Identity	3959
Type(H)	3951	! Id Identity	3959
Scheme(H)	3951	Points(H, x)	3960
Curve(H)	3951	Points(E, x)	3960
Ring(H)	3951	Points RationalPoints	3960
<i>120.7.3 Predicates on Point Sets</i>	<i>3951</i>	Points RationalPoints	3960
eq	3951	PointsAtInfinity(H)	3960
ne	3951	PointsAtInfinity(E)	3960
120.8 Morphisms	3952	<i>120.9.2 Creation Predicates</i>	<i>3960</i>
<i>120.8.1 Creation Functions</i>	<i>3952</i>	IsPoint(H, S)	3960
Isomorphism(E, F, [r, s, t, u])	3953	IsPoint(E, S)	3960
Isomorphism(E, F)	3953	IsPoint(H, x)	3961
Automorphism(E, [r, s, t, u])	3953	IsPoint(E, x)	3961
IsomorphismData(I)	3953	<i>120.9.3 Access Operations</i>	<i>3961</i>
IsIsomorphism(I)	3954	P[i]	3961
IsomorphismToIsogeny(I)	3954	ElementToSequence(P)	3961
TranslationMap(E, P)	3955	Eltseq(P)	3961
RationalMap(i, t)	3955	<i>120.9.4 Associated Structures</i>	<i>3961</i>
TwoIsogeny(P)	3955	Category(P)	3961
IsogenyFromKernel(G)	3955	Type(P)	3961
IsogenyFromKernelFactored(G)	3955	Parent(P)	3961
IsogenyFromKernel(E, psi)	3956	Scheme(P)	3961
IsogenyFromKernelFactored(E, psi)	3956	Curve(P)	3961
PushThroughIsogeny(I, v)	3956	<i>120.9.5 Arithmetic</i>	<i>3961</i>
PushThroughIsogeny(I, G)	3956	-	3961
DualIsogeny(phi)	3956	+	3961
<i>120.8.2 Predicates on Isogenies</i>	<i>3957</i>	+=	3962
IsZero(I)	3957	-	3962
IsConstant(I)	3957	-=	3962
eq	3957	*	3962
<i>120.8.3 Structure Operations</i>	<i>3957</i>	*:=	3962
IsogenyMapPsi(I)	3957	<i>120.9.6 Division Points</i>	<i>3962</i>
IsogenyMapPsiMulti(I)	3957	/	3962
IsogenyMapPsiSquared(I)	3957	/:=	3962
IsogenyMapPhi(I)	3957	DivisionPoints(P, n)	3962
IsogenyMapPhiMulti(I)	3957	IsDivisibleBy(P, n)	3962
IsogenyMapOmega(I)	3957	<i>120.9.7 Point Order</i>	<i>3965</i>
Kernel(I)	3957	Order(P)	3965
Degree(I)	3957	FactoredOrder(P)	3965
<i>120.8.4 Endomorphisms</i>	<i>3958</i>	<i>120.9.8 Predicates on Points</i>	<i>3965</i>
MultiplicationByMMap(E, m)	3958	IsId(P)	3965
IdentityIsogeny(E)	3958	IsIdentity(P)	3965
IdentityMap(E)	3958	IsZero(P)	3965
NegationMap(E)	3958	eq	3966
FrobeniusMap(E, i)	3958	ne	3966
FrobeniusMap(E)	3958	in	3966
		in	3966

IsOrder(P, m)	3966	WeilPairing(P, Q, n)	3967
IsIntegral(P)	3966	IsLinearlyIndependent(S, n)	3967
IsSIntegral(P, S)	3966	IsLinearlyIndependent(P, Q, n)	3967
120.9.9 Weil Pairing	3967	120.10 Bibliography	3968

Chapter 120

ELLIPTIC CURVES

120.1 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 121),
- curves over the rationals or number fields (Chapter 122),
- curves over univariate function fields (Chapter 123).

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 \mathbf{Q} , and for point counting and the determination of the group structure over \mathbf{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 \mathbf{Z} or \mathbf{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 112 (Schemes) and 114 (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/\mathbf{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 121; Chapter 122 presents the wide range of techniques available for determining information about the group of rational points for curves over \mathbf{Q} and over number fields, while elliptic curves over function fields are discussed in Chapter 123.

120.2 Creation Functions

120.2.1 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 \mathbf{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])
```

```
EllipticCurve([a1, a2, a3, a4, a6])
```

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 \mathbf{Q} . The given coefficients must define a nonsingular curve; that is, the discriminant of the curve must be nonzero.

```
EllipticCurve(f)
```

```
EllipticCurve(f, h)
```

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)
```

```
EllipticCurveWithjInvariant(j)
```

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 H120E1

We create one particular elliptic curve over \mathbf{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)

Verbose

EllModel

Maximum : 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)

Verbose

EllModel

Maximum : 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)

Verbose

EllModel

Maximum : 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)

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

Example H120E2

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 $y^2 = x^3 - 5/16x^2 + 1/32x + 15/1024$ 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 $y^2 + 1024y = x^3 + 16x^2$ 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 H120E3

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

```

120.2.2 Creation Predicates

IsEllipticCurve([a, b])

IsEllipticCurve([a1, a2, a3, a4, a6])

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)

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 H120E4

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)
```

120.2.3 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)**BaseExtend(E, K)**

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)

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 $k = \mathbf{Q}$ and K is a finite field).

BaseChange(E, h)**BaseExtend(E, h)**

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

BaseChange(E , n)

BaseExtend(E , n)

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 H120E5

```
> 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);
```

Runtime error in 'BaseExtend': Coercion of equations is not possible

```
> assert E1 eq ChangeRing(E2, K1); // but this is OK
```

120.2.4 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 $E \rightarrow E'$ and $E' \rightarrow E$.

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)

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)

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

SimplifiedModel(E)

A simplified model of the elliptic curve E is returned. If E is defined over \mathbf{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)

Given an elliptic curve E defined over \mathbf{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 \mathbf{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 $K = \mathbf{Q}$); 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)**MinimalModel(E, P : parameters)**

`UseGeneratorAsUniformiser` `BOOLELT` *Default : 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 \mathbf{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.

120.2.5 Predicates on Curve Models**IsWeierstrassModel(E)**

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

IsIntegralModel(E)

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

IsSimplifiedModel(E)

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

IsMinimalModel(E)

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

IsIntegralModel(E, P)

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 H120E6

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
```

120.2.6 Twists of Elliptic Curves

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

QuadraticTwist(E, d)

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)

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

QuadraticTwists(E)

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)

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 H120E7

In this example we compute the quadratic twists of an elliptic curve over the finite field \mathbf{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)

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)

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 $\text{QuadraticTwist}(E, d)$.

Example H120E8

In this example we take curves of j -invariant 0 and $12^3 = 12$ over the finite field \mathbf{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),
  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)

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 H120E9

```
> 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
```

120.3 Operations on Curves

120.3.1 Elementary Invariants

aInvariants(E)**Coefficients(E)****ElementToSequence(E)****Eltseq(E)**

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)

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)

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)

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)

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)

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 H120E10

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 H120E11

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,
  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

```

120.3.2 Associated Structures

`Category(E)`

`Type(E)`

Returns the category of elliptic curves, `CrvEll`.

`BaseRing(E)`

`CoefficientRing(E)`

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 .

120.3.3 Predicates on Elliptic Curves

`E eq F`

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`

The logical negation of `eq`.

`IsIsomorphic(E, F)`

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)`

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 H120E12

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
```

120.4 Polynomials

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

<code>DefiningPolynomial(E)</code>

Returns the homogeneous defining polynomial for the elliptic curve E .

<code>DivisionPolynomial(E, n)</code>

<code>DivisionPolynomial(E, n, g)</code>
--

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 . If n is even this polynomial has multiplicity two at the nonzero 2-torsion points of the curve. The second return value is the n -th division 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.

<code>TwoTorsionPolynomial(E)</code>

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

Example H120E13

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
```

120.5 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 $\psi = 0$.

120.5.1 Creation of Subgroup Schemes

SubgroupScheme(G , f)

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)

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

120.5.2 Associated Structures

`Category(G)`

`Type(G)`

Returns the category of elliptic curve subgroup schemes, `SchGrpE11`.

`Curve(G)`

`Generic(G)`

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

`BaseRing(G)`

`CoefficientRing(G)`

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

`DefiningSubschemePolynomial(G)`

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

120.5.3 Predicates on Subgroup Schemes

`G1 eq G2`

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`

The logical negation of `eq`.

120.5.4 Points of Subgroup Schemes

`#G`

`Order(G)`

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

`FactoredOrder(G)`

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

`Points(G)`

`RationalPoints(G)`

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

Example H120E14

We construct a curve over \mathbf{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  $x^3 + 6x^2 + 4x + 6$ 
> 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
```

120.6 The Formal Group

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

<code>FormalGroupLaw(E, prec)</code>

This function returns a polynomial in two variables, $T_1 + T_2 + \dots$, 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)`

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)`

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 .

120.7 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 invariants (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.)

120.7.1 Creation of Point Sets

`E(L)`

`PointSet(E, L)`

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)`

`PointSet(E, m)`

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.

120.7.2 Associated Structures

Category(H)

Type(H)

Given a point set H of an elliptic curve, returns the category `SetPtE11` of point sets of elliptic curves.

Scheme(H)

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

Curve(H)

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

Ring(H)

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

120.7.3 Predicates on Point Sets

H1 eq H2

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

The logical negation of `eq`.

Example H120E15

We create an elliptic curve E over $\text{GF}(5)$ 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
> 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)
```

120.8 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 $E(\mathbf{Q}) \rightarrow F(\mathbf{Q})$ 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 .

120.8.1 Creation Functions

Example H120E16

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])

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 $E \rightarrow F$ 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)

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])

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 $E \rightarrow E$ 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)

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

Example H120E17

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)

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)

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

Example H120E18

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]);
> 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)

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

RationalMap(i, t)

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

TwoIsogeny(P)

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 H120E19

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)

Let G be a subgroup scheme of an elliptic curve E . There is a separable isogeny $f : E \rightarrow E_f$ 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)

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)`

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)`

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)`

`PushThroughIsogeny(I, G)`

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)`

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

Example H120E20

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

Of course, a simpler way to verify this is just to check equality:

```
> f_dual eq DualIsogeny(f);
true
```

120.8.2 Predicates on Isogenies

`IsZero(I)`

`IsConstant(I)`

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

`I eq J`

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

120.8.3 Structure Operations

`IsogenyMapPsi(I)`

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

`IsogenyMapPsiMulti(I)`

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

`IsogenyMapPsiSquared(I)`

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)`

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

`IsogenyMapPhiMulti(I)`

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

`IsogenyMapOmega(I)`

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

`Kernel(I)`

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

`Degree(I)`

Returns the degree of the morphism I .

120.8.4 Endomorphisms

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

The endomorphism ring of E always contains a subring isomorphic to \mathbf{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)

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

IdentityIsogeny(E)

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

IdentityMap(E)

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

NegationMap(E)

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

FrobeniusMap(E, i)

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)

Equivalent to `FrobeniusMap(E, 1)`.

Example H120E21

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
```

120.8.5 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.

<code>AutomorphismGroup(E)</code>

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).

<code>Automorphisms(E)</code>

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

120.9 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 $y = 1$.

120.9.1 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.

<code>H ! [x, y, z]</code>

<code>elt< H x, y, z ></code>

<code>E ! [x, y, z]</code>

<code>elt< E x, y, z ></code>

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.

<code>H ! 0</code>

<code>Id(H)</code>

<code>Identity(H)</code>

<code>E ! 0</code>

<code>Id(E)</code>

<code>Identity(E)</code>

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

Points(H, x)

Points(E, x)

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

Points(H)

RationalPoints(H)

Points(E)

RationalPoints(E)

Bound	RNGINTELT	<i>Default</i> : 0
DenominatorBound	RNGINTELT	<i>Default</i> : Bound
Denominators	SETQ	<i>Default</i> :
Max	RNGINTELT	<i>Default</i> :
NPrimes	RNGINTELT	<i>Default</i> : 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 \mathbf{Q} or a number field, a positive value for the parameter **Bound** must be given. Over \mathbf{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)

PointsAtInfinity(E)

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

120.9.2 Creation Predicates

IsPoint(H, S)

IsPoint(E, S)

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)

IsPoint(E, x)

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.

120.9.3 Access Operations

$P[i]$

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

ElementToSequence(P)

Eltseq(P)

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

120.9.4 Associated Structures

Category(P)

Type(P)

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

Parent(P)

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

Scheme(P)

Curve(P)

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

120.9.5 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 $O = (0 : 1 : 0)$.

$-P$

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

$P + Q$

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

`P += Q`

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

`P - Q`

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

`P -= Q`

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

`n * P`

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

`P *= n`

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

120.9.6 Division Points

`P / n`

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`

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)`

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)`

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 H120E22

```
> 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 H120E23

We construct some points in a certain elliptic curve over \mathbf{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;
> 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 H120E24

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)
```

120.9.7 Point Order

Order(P)

Given a point on an elliptic curve defined over \mathbf{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)

Given a point on an elliptic curve defined over \mathbf{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 \mathbf{Q} and P has infinite order.

Example H120E25

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> ]
```

120.9.8 Predicates on Points

IsId(P)

IsIdentity(P)

IsZero(P)

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

P eq Q

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

The logical negation of **eq**.

P in H

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

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)

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)

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

IsSIntegral(P, S)

Given a point P on an elliptic curve defined over \mathbf{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 H120E26

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) ]
```

120.9.9 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)`

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)`

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 $\mathbf{Z}/n\mathbf{Z}$.

`IsLinearlyIndependent(P, Q, n)`

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 H120E27

We compute the Weil pairing of two 3-torsion points on the curve $y^2 = x^3 - 3$ over \mathbf{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);
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

120.10 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. 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.
- [**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.
- [**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.

121 ELLIPTIC CURVES OVER FINITE FIELDS

121.1 Supersingular Curves . . . 3971	121.4 Abelian Group Structure . . 3981
IsSupersingular(E : -) 3971	AbelianGroup(E) 3981
SupersingularPolynomial(p) 3971	TorsionSubgroup(E) 3981
IsOrdinary(E) 3972	Generators(H) 3981
IsProbablySupersingular(E) 3972	Generators(E) 3981
121.2 The Order of the Group of Points 3972	NumberOfGenerators(H) 3981
<i>121.2.1 Point Counting 3972</i>	NumberOfGenerators(E) 3981
# 3972	Ngens(H) 3981
Order(E) 3972	Ngens(E) 3981
FactoredOrder(E) 3972	121.5 Pairings on Elliptic Curves . 3982
SEA(H : -) 3972	<i>121.5.1 Weil Pairing 3982</i>
SEA(E : -) 3972	WeilPairing(P, Q, n) 3982
SetVerbose("SEA", v) 3976	<i>121.5.2 Tate Pairing 3982</i>
Order(E, r) 3976	TatePairing(P, Q, n) 3982
Trace(E) 3976	ReducedTatePairing(P, Q, n) 3982
TraceOfFrobenius(E) 3976	<i>121.5.3 Eta Pairing 3983</i>
Trace(E, r) 3976	EtaTPairing(P, Q, n, q) 3983
TraceOfFrobenius(E, r) 3976	ReducedEtaTPairing(P, Q, n, q) 3983
<i>121.2.2 Zeta Functions 3978</i>	EtaqPairing(P, Q, n, q) 3983
ZetaFunction(E) 3978	<i>121.5.4 Ate Pairing 3984</i>
<i>121.2.3 Cryptographic Elliptic Curve Do- mains 3979</i>	AteTPairing(Q, P, n, q) 3984
CryptographicCurve(F) 3979	ReducedAteTPairing(Q, P, n, q) 3984
ValidateCryptographicCurve(E, P, ordP, h) 3979	AteqPairing(P, Q, m, q) 3984
SetVerbose("ECDom", v) 3979	121.6 Weil Descent in Characteristic Two 3988
121.3 Enumeration of Points . . . 3980	WeilDescent(E, k, c) 3988
Points(E) 3980	WeilDescentGenus(E, k, c) 3989
Points(H) 3980	WeilDescentDegree(E, k, c) 3989
RationalPoints(E) 3980	121.7 Discrete Logarithms 3990
RationalPoints(H) 3980	Log(Q, P) 3990
Random(E) 3980	Log(Q, P, t) 3990
Random(H) 3980	121.8 Bibliography 3991

Chapter 121

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 120. 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. These tools are used as the basis for functions that search for curves suitable for cryptographic applications.

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.

121.1 Supersingular Curves

IsSupersingular(E : <i>parameters</i>)
--

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)

Given a prime p , returns the separable monic polynomial over \mathbf{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)

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)

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.

121.2 The Order of the Group of Points**121.2.1 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.

#E**Order(E)**

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

FactoredOrder(E)

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)**SEA(E : parameters)**

Al	MONSTGELT	<i>Default</i> : "Default"
MaxSmooth	RNGINTELT	<i>Default</i> : ∞
AbortLevel	RNGINTELT	<i>Default</i> : 0
UseSEA	BOOLELT	<i>Default</i> : 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 `A1` can be set to one of `"Enumerate"`, `"BSGS"`, or `"Default"`. Setting `A1` 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 `A1` to `"BSGS"` is currently equivalent to setting `A1` 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` on page 4-1268) 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 ([Har]) 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 \mathbf{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 H121E1

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 := FiniteField(2, 160); // finite field of size 2^160
> E := EllipticCurve([K!1, 0, 0, 0, K.1]);
> time #E;
1461501637330902918203686141511652467686942715904
Time: 0.100
> //example with a GNB of Type 1
> HasGNB(pAdicRing(2, 1), 162, 1);
true
> K := FiniteField(2, 162); // finite field of size 2^162
> E := EllipticCurve([K!1, 0, 0, 0, K.1]);
> time #E;
5846006549323611672814738806268278193573757976576
Time: 0.020
> //example with a GNB of Type 2
> K := FiniteField(2, 173); // finite field of size 2^173
> E := EllipticCurve([K!1, 0, 0, 0, K.1]);
> time #E;
11972621413014756705924586057649971923911742202056704
Time: 0.090
```

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: 0.020

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: 1.580
```

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 (see also `CryptographicCurve` on page 3979):

```
> 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  $y^2 = x^3 + 29x + 1$  over GF(1000000007)
> #E;
999998501

```

`SetVerbose("SEA", v)`

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(E, r)`

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(E)`

`TraceOfFrobenius(E)`

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(E, r)`

`TraceOfFrobenius(E, r)`

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

Example H121E2

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.

```
> FF<w> := GF(2^133);
> E := EllipticCurve([1, 0, 0, 0, w]);
> time #E;
10889035741470030830941160923712974513152
Time: 8.830
> FactoredOrder(E);
[ <2, 10>, <3, 2>, <150959, 1>, <654749, 1>, <11953995917236774217401867, 1> ]
> time TraceOfFrobenius(E);
-113173485896391746559
Time: 0.000
> time TraceOfFrobenius(E, 3);
2247497466279379392112525914232899060001042788725924296315905
Time: 0.000
```

Example H121E3

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  $\text{GF}(101^2)$ ,
  Elliptic Curve defined by  $y^2 = x^3 + (61*w + 91)$  over  $\text{GF}(101^2)$ ,
  Elliptic Curve defined by  $y^2 = x^3 + (98*w + 16)$  over  $\text{GF}(101^2)$ ,
  Elliptic Curve defined by  $y^2 = x^3 + (9*w + 66)$  over  $\text{GF}(101^2)$ ,
  Elliptic Curve defined by  $y^2 = x^3 + (59*w + 76)$  over  $\text{GF}(101^2)$ ,
  Elliptic Curve defined by  $y^2 = x^3 + (87*w + 81)$  over  $\text{GF}(101^2)$ 
]
> 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 ]

```

121.2.2 Zeta Functions

ZetaFunction(E)

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 H121E4

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

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

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))) + 0(t^7);
16*t + 128*t^2 + 1264*t^3 + 14848*t^4 + 160976*t^5 + 1769600*t^6 + 0(t^7)
> [ Order(E, n) : n in [1..6] ];
[ 16, 128, 1264, 14848, 160976, 1769600 ]

```

121.2.3 Cryptographic Elliptic Curve Domains

This section describes functions for generating/validating good cryptographic Elliptic Curve Domains. The basic data of such a domain is an elliptic curve E over a finite field together with a point P on E such that the order of P is a large prime and the pair (E, P) satisfies the standard security conditions for being resistant to MOV and Anomalous attacks (see [JM00]).

<code>CryptographicCurve(F)</code>

<code>ValidateCryptographicCurve(E, P, ordP, h)</code>
--

<code>OrderBound</code>	<code>RNGINTELT</code>	<i>Default</i> : 2^{160}
<code>Proof</code>	<code>BOOLELT</code>	<i>Default</i> : <code>true</code>
<code>UseSEA</code>	<code>BOOLELT</code>	<i>Default</i> : <code>false</code>

Given a finite field F , the first function finds an Elliptic Curve Domain over F ; it generates random elliptic curves over F until a suitable one is found. For each generated curve E , the order $\#E$ is calculated and a check is made as to whether sieving $\#E$ by small primes leaves a strong pseudoprime

$$p \geq \max(\text{OrderBound}, 4\sqrt{\#F}).$$

If so, the security conditions (which only depend on the order of P) are checked for p . Finally, if `Proof` is `true`, p is proven to be prime. If all of the above conditions hold true then a random point P on E of order p is found and $E, P, p, \#E/p$ are returned.

If the Schoof–Elkies–Atkins method is used for computing $\#E$ then the early abort mechanism (see `SEA` on page 3972) can help to shortcut the process when `OrderBound` is close to $\#F$. However, when they apply, it is generally still quicker overall to use the much faster p -adic point-counting methods for large fields. To force the use of the SEA method for point-counting set `UseSEA` to `true`. This is not recommended!

Given data of the form returned by the first function as input, the second function verifies that it is valid data for a secure EC Domain with `ordP` satisfying the above inequality for p . If `Proof` is `true` (*resp.* `false`), `ordP` is proven to be prime (*resp.* subjected to a strong pseudoprimality test).

NB: For the first function, it is required that $\#F \geq \text{OrderBound}$ (or $2 * \text{OrderBound}$ if F has characteristic 2 when all ordinary elliptic curves have even order).

<code>SetVerbose("ECDom", v)</code>

Set the verbose printing level for progress output when running the above functions. Currently the legal values for v are `true`, `false`, 0, 1, and 2 (`false` is the same as 0, and `true` is the same as 1).

Example H121E5

For a bit of extra speed, we look for curves over $F = GF(2^{196})$ where a Type 1 GNB exists for p -adic point-counting (see SEA on page 3972). We begin by looking for a curve over F with a point whose order is greater than the default `OrderBound`.

```
> SetVerbose("ECDom", 1);
> F := FiniteField(2, 196);
> E,P,ord,h := CryptographicCurve(F);
Finished! Tried 2 curves.
Total time: 1.721
> ord; h;
12554203470773361527671578846370731873587186564992869565421
8
```

Now we search for a curve whose order is twice a prime (the best we can do in characteristic 2!). This can be accomplished by setting `OrderBound` to half the field size.

```
> E,P,ord,h := CryptographicCurve(F : OrderBound := 2^195);
Finished! Tried 102 curves.
Total time: 22.049
> ord; h;
50216813883093446110686315385797359941852852162929185668319
2
> ValidateCryptographicCurve(E, P, ord, h);
Verifying primality of the order of P...
true
```

121.3 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)

Points(H)

RationalPoints(E)

RationalPoints(H)

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)

Random(H)

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.

121.4 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(E)

TorsionSubgroup(E)

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)

Generators(E)

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)

NumberOfGenerators(E)

Ngens(H)

Ngens(E)

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 H121E6

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

```

121.5 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.

121.5.1 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)

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.

121.5.2 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)

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)

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 $\#K - 1$.

121.5.3 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 \mathbf{F}_q and $n \mid \#E(\mathbf{F}_q)$. The security multiplier k is the smallest positive integer such that $q^k \equiv 1 \pmod n$. Here, in the supersingular case, k divides 6. Now, $E[n] \subseteq E(\mathbf{F}_{q^k})$ is a free $\mathbf{Z}/n\mathbf{Z}$ -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 $E(\mathbf{F}_q)[n]$. If R is an arbitrary n -torsion point in $E(\mathbf{F}_{q^k})$, 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 $\#E(\mathbf{F}_q) = q + 1 - t$ 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 \mathbf{F}_{q^k} .

EtaTPairing(P, Q, n, q)

CheckCurve	BOOLELT	<i>Default : false</i>
CheckPoints	BOOLELT	<i>Default : false</i>

Given a supersingular elliptic curve E/\mathbf{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 $\#E(\mathbf{F}_q) = q + 1 - t$. The result is non-reduced and thus a random representative of a whole coset. Both points need to be in the same point set $E(\mathbf{F}_{q^k})$ and q should be the size of the base field.

ReducedEtaTPairing(P, Q, n, q)

CheckCurve	BOOLELT	<i>Default : false</i>
CheckPoints	BOOLELT	<i>Default : false</i>

The reduced version of the EtaTPairing function.

EtaqPairing(P, Q, n, q)

CheckCurve	BOOLELT	<i>Default : false</i>
CheckPoints	BOOLELT	<i>Default : false</i>

Given a supersingular elliptic curve E/\mathbf{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 $E(\mathbf{F}_{q^k})$ 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.

121.5.4 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 $\#E(\mathbf{F}_q) = q + 1 - t$ or $T = q$.

<code>AteTPairing(Q, P, n, q)</code>

`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 $\#E(\mathbf{F}_q) = q + 1 - t$. The result is non-reduced and thus a random representative of a whole coset. Both points need to be in the same point set $E(\mathbf{F}_{q^k})$ and q should be the size of the base field.

<code>ReducedAteTPairing(Q, P, n, q)</code>

`CheckPoints`

BOOLELT

Default : false

The reduced version of the `AteTPairing` function.

<code>AteqPairing(P, Q, m, q)</code>

`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 $E(\mathbf{F}_{q^k})$ 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 H121E7

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 \mathbf{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; // some random start value
> repeat
>   z := z+1;
>   p := Evaluate(ps, z);
>   n := Evaluate(ns, z);
> until IsProbablePrime(p) and IsProbablePrime(n);
> p;
2497857711095780713403056606399151275099020724723
> n;
2497857711095780713403055025937917618634473494829
> 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(2497857711095780713403056606399151275099020724723)
> #E eq n; // just to verify
true
> t := p + 1 - n;
> t;
1580461233656464547229895
> 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 : $\text{Tr}(Q) = \sum_{i=0}^{k-1} F^i(Q)$ 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 \mathbf{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 H121E8

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 \mathbf{F}_p for $p > 3$ can be $\mathbf{Z}/n\mathbf{Z}$ or $\mathbf{Z}/2\mathbf{Z} \times \mathbf{Z}/(n/2)\mathbf{Z}$, where $n = p + 1$, and the group structure over a degree 2 extension is $\mathbf{Z}/n\mathbf{Z} \times \mathbf{Z}/n\mathbf{Z}$. The nontrivial Weil pairing on this is the basis for this reduction.

```
> p := NextPrime(2^131);
> n := p + 1;
> n;
2722258935367507707706996859454145691688
> Factorization(n);
[ <2, 3>, <3, 2>, <37809151880104273718152734159085356829, 1> ]
> E0 := SupersingularEllipticCurve(GF(p));
> G<x>, f := AbelianGroup(E0);
> G;
Abelian Group isomorphic to Z/2722258935367507707706996859454145691688
Defined on 1 generator
Relations:
    2722258935367507707706996859454145691688*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
```

121.6 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 $\text{Jac}(C)(k)$ [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 [Gau00] or the chapter by F. Heß in [Bla05]), which constructs such a C along with the divisor map from E to C .

<code>WeilDescent(E, k, c)</code>

`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, $\text{Jac}(C)$. 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 $\text{Jac}(C)(k)$ 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`. 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)`

Returns the genus of the Weil descent curve C produced by `WeilDescent` for input E, k, c .

`WeilDescentDegree(E, k, c)`

Returns the degree in the second variable of the plane Weil descent curve C produced by `WeilDescent` for input E, k, c .

Example H121E9

The following example is similar to one from the article of F. Heß referred to above. E is defined over $\mathbf{F}_{2^{155}}$ and C is a hyperelliptic curve of genus 31 over \mathbf{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];
> 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]

121.7 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)

The discrete log of P to the base Q (an integer n satisfying $n * Q = P$ where $0 \leq n < \text{Order}(Q)$). 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)

The discrete log of P to the base Q (an integer n satisfying $n * Q = P$ where $0 \leq n < \text{Order}(Q)$). 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 H121E10

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.

```
> GetSeed();
1020 132
> K := GF(RandomPrime(40));
> E := EllipticCurve([Random(K), Random(K)]);
> E;
Elliptic Curve defined by  $y^2 = x^3 + 456271502613x + 504334195864$ 
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;
597156621194
> m*Q - P;
(0 : 1 : 0)
```

121.8 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.
- [Gau00] P. Gaudry, F. Heß and N. P. Smart. Constructive and destructive facets of Weil descent on elliptic curves. *J. Cryptology*, 15(1):19–46, 2000.
- [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*, pages 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.
- [Har] R. Harley. Web posting. Under November 2002 entry at URL:<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. URL:<http://wis.kuleuven.be/algebra/hubrechts/>.

- [**JM00**] D. Johnson and A. Menezes. The Elliptic Curve Digital Signature Algorithm (ECDSA). Technical report, Univ. Waterloo, 2000. Available at URL:<http://www.cacr.math.uwaterloo.ca/>.
- [**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.

122 ELLIPTIC CURVES OVER \mathbb{Q} AND NUMBER FIELDS

122.1 Introduction	3997		
122.2 Curves over the Rationals .	3997		
122.2.1 <i>Local Invariants</i>	<i>3997</i>		
Conductor(E)	3997	Regulator(E)	4008
BadPrimes(E)	3997	SilvermanBound(E)	4009
TamagawaNumber(E, p)	3997	SiksekBound(E: -)	4009
TamagawaNumbers(E)	3997	IsLinearlyIndependent(P, Q)	4010
LocalInformation(E, p)	3998	IsLinearlyIndependent(S)	4010
LocalInformation(E)	3998	ReducedBasis(S)	4010
ReductionType(E, p)	3998	pAdicHeight(P, p)	4011
FrobeniusTraceDirect(E, p)	3998	pAdicRegulator(S, p)	4012
TracesOfFrobenius(E, B)	3998	EisensteinTwo(E, p)	4012
122.2.2 <i>Kodaira Symbols</i>	<i>3999</i>	122.2.7 <i>Two-Descent and Two-Coverings</i>	<i>4013</i>
KodairaSymbol(E, p)	3999	TwoDescent(E: -)	4013
KodairaSymbols(E)	3999	AssociatedEllipticCurve(f)	4014
KodairaSymbol(s)	3999	AssociatedEllipticCurve(C)	4014
eq	3999	TwoIsogenyDescent(E: -)	4015
ne	3999	LiftDescendant(C)	4015
122.2.3 <i>Complex Multiplication</i>	<i>4000</i>	QuarticIInvariant(q)	4015
HasComplexMultiplication(E)	4000	QuarticJInvariant(q)	4015
122.2.4 <i>Isogenous Curves</i>	<i>4000</i>	QuarticG4Covariant(q)	4015
IsogenousCurves(E)	4000	QuarticG6Covariant(q)	4015
FaltingsHeight(E)	4000	QuarticHSeminvariant(q)	4015
122.2.5 <i>Mordell–Weil Group</i>	<i>4001</i>	QuarticPSeminvariant(q)	4015
MordellWeilShaInformation(E: -)	4002	QuarticQSeminvariant(q)	4015
DescentInformation(E: -)	4002	QuarticRSeminvariant(q)	4015
TorsionSubgroup(E)	4003	QuarticNumberOfRealRoots(q)	4016
Rank(E: -)	4004	QuarticMinimise(q)	4016
MordellWeilRank(E: -)	4004	QuarticReduce(q)	4016
RankBounds(E: -)	4004	IsEquivalent(f,g)	4016
MordellWeilRankBounds(E: -)	4004	122.2.8 <i>The Cassels-Tate Pairing</i>	<i>4016</i>
MordellWeilGroup(E: -)	4004	CasselsTatePairing(C, D)	4017
AbelianGroup(E: -)	4004	CasselsTatePairing(C, D)	4017
Generators(E)	4005	122.2.9 <i>Four-Descent</i>	<i>4018</i>
NumberOfGenerators(E)	4005	FourDescent(f: -)	4018
Ngens(E)	4005	FourDescent(C: -)	4018
Saturation(points, n)	4005	AssociatedEllipticCurve(qi)	4020
122.2.6 <i>Heights and Height Pairing</i>	<i>4007</i>	AssociatedHyperellipticCurve(qi)	4020
NaiveHeight(P)	4007	QuadricIntersection(F)	4020
WeilHeight(P)	4007	QuadricIntersection(P, F)	4020
Height(P: -)	4007	QuadricIntersection(E)	4020
CanonicalHeight(P: -)	4007	QuadricIntersection(C)	4020
LocalHeight(P, p)	4007	IsQuadricIntersection(C)	4020
HeightPairing(P, Q: -)	4008	PointsQI(C, B: -)	4020
HeightPairingMatrix(S: -)	4008	TwoCoverPullback(H, pt)	4020
HeightPairingMatrix(E: -)	4008	TwoCoverPullback(f, pt)	4020
Regulator(S)	4008	FourCoverPullback(C, pt)	4021
		122.2.10 <i>Eight-Descent</i>	<i>4022</i>
		EightDescent(C: -)	4022
		122.2.11 <i>Three-Descent</i>	<i>4023</i>
		ThreeDescent(E: -)	4023

ThreeSelmerGroup(E : -)	4025	pAdicEllipticLogarithm(P, p: -)	4043
ThreeDescentCubic(E, α : -)	4025	RootNumber(E)	4044
ThreeIsogenyDescent(E : -)	4026	RootNumber(E, p)	4044
ThreeIsogenySelmerGroups(E : -)	4026	AnalyticRank(E)	4044
ThreeIsogenyDescentCubic(ϕ , α)	4027	ConjecturalRegulator(E)	4045
ThreeDescentByIsogeny(E)	4027	ConjecturalRegulator(E, v)	4045
Jacobian(C)	4028	ModularDegree(E)	4046
ThreeSelmerElement(E, C)	4028	<i>122.2.16 Integral and S-integral Points . . .</i>	<i>4047</i>
ThreeSelmerElement(C)	4028	IntegralPoints(E)	4047
AddCubics(cubic1, cubic2 : -)	4028	SIntegralPoints(E, S)	4047
ThreeTorsionType(E)	4029	IntegralQuarticPoints(Q)	4049
ThreeTorsionPoints(E : -)	4029	IntegralQuarticPoints(Q, P)	4049
ThreeTorsionMatrices(E, C)	4029	SIntegralQuarticPoints(Q, S)	4049
SixDescent(C2, C3)	4029	SIntegralLjunggrenPoints(Q, S)	4050
SixDescent(model2, model3)	4029	SIntegralDesbovesPoints(Q, S)	4050
TwelveDescent(C3, C4)	4030	<i>122.2.17 Elliptic Curve Database</i>	<i>4050</i>
TwelveDescent(model3, model4)	4030	EllipticCurveDatabase(: -)	4050
<i>122.2.12 Nine-Descent</i>	<i>4030</i>	CremonaDatabase(: -)	4050
NineDescent(C : -)	4030	SetBufferSize(D, n)	4051
NineSelmerSet(C)	4031	LargestConductor(D)	4051
<i>122.2.13 p-Isogeny Descent</i>	<i>4031</i>	ConductorRange(D)	4051
pIsogenyDescent(E,P)	4031	#	4051
pIsogneyDescent(E,p)	4032	NumberOfCurves(D)	4051
pIsogenyDescent(lambda,p)	4032	NumberOfCurves(D, N)	4051
pIsogenyDescent(C,phi)	4032	NumberOfCurves(D, N, i)	4051
pIsogenyDescent(C,E1,E2)	4032	NumberOfIsogenyClasses(D, N)	4051
pIsogenyDescent(C,P)	4032	EllipticCurve(D, N, I, J)	4051
FakeIsogenySelmerSet(C,phi)	4032	EllipticCurve(D, N, S, J)	4051
FakeIsogenySelmerSet(C,E1,E2)	4032	EllipticCurve(D, S)	4051
FakeIsogenySelmerSet(C,P)	4032	EllipticCurve(S)	4051
<i>122.2.14 Heegner Points</i>	<i>4035</i>	Random(D)	4052
HeegnerPoint(E : -)	4035	CremonaReference(D, E)	4052
HeegnerPoint(C : -)	4036	CremonaReference(E)	4052
HeegnerPoint(f : -)	4036	EllipticCurves(D, N, I)	4053
ModularParametrization(E, z, B : -)	4037	EllipticCurves(D, N, S)	4053
ModularParametrization(E, z : -)	4037	EllipticCurves(D, N)	4053
ModularParametrization(E, Z, B : -)	4037	EllipticCurves(D, S)	4053
ModularParametrization(E, Z : -)	4037	EllipticCurves(D)	4053
ModularParametrization(E, f, B : -)	4037	122.3 Curves over Number Fields .	4054
ModularParametrization(E, f : -)	4037	<i>122.3.1 Local Invariants</i>	<i>4054</i>
ModularParametrization(E, F, B : -)	4037	Conductor(E)	4054
ModularParametrization(E, F : -)	4037	BadPlaces(E)	4054
HeegnerDiscriminants(E,lo,hi)	4037	BadPlaces(E, L)	4054
HeegnerForms(E,D : -)	4037	LocalInformation(E, P)	4054
HeegnerForms(N,D : -)	4038	LocalInformation(E)	4054
ManinConstant(E)	4038	Reduction(E, p)	4055
HeegnerTorsionElement(E)	4038	<i>122.3.2 Complex Multiplication</i>	<i>4055</i>
HeegnerPoints(E, D : -)	4038	HasComplexMultiplication(E)	4055
<i>122.2.15 Analytic Information</i>	<i>4042</i>	<i>122.3.3 Mordell–Weil Groups</i>	<i>4055</i>
Periods(E: -)	4042	TorsionBound(E, n)	4055
EllipticCurveFromPeriods(om: -)	4042	pPowerTorsion(E, p)	4055
RealPeriod(E: -)	4042	TorsionSubgroup(E)	4055
EllipticExponential(E, z)	4043	MordellWeilShaInformation(E: -)	4055
EllipticExponential(E, S)	4043	DescentInformation(E: -)	4055
EllipticLogarithm(P: -)	4043	RankBound(E)	4056
EllipticLogarithm(E, S)	4043		

<i>122.3.4 Heights</i>	4056	<code>AbsoluteAlgebra(A)</code>	4067
<code>NaiveHeight(P)</code>	4056	<code>pSelmerGroup(A, p, S)</code>	4067
<code>Height(P : -)</code>	4056	<code>LocalTwoSelmerMap(P)</code>	4067
<code>HeightPairingMatrix(P : -)</code>	4056	<code>LocalTwoSelmerMap(A, P)</code>	4067
<code>LocalHeight(P, P1 : -)</code>	4056	<i>122.3.10 Analytic Information</i>	4068
<i>122.3.5 Two Descent</i>	4057	<code>RootNumber(E, P)</code>	4068
<code>TwoDescent(E:-)</code>	4057	<code>RootNumber(E)</code>	4068
<code>TwoCover(e)</code>	4057	<code>AnalyticRank(E)</code>	4068
<code>TwoCover(e)</code>	4057	<code>ConjecturalRegulator(E)</code>	4069
<i>122.3.6 Selmer Groups</i>	4057	<code>ConjecturalSha(E, Pts)</code>	4069
<code>DescentMaps(phi)</code>	4058	<i>122.3.11 Elliptic Curves of Given Conductor</i>	4069
<code>CasselsMap(phi)</code>	4058	<code>EllipticCurveSearch(N, Effort)</code>	4069
<code>SelmerGroup(phi)</code>	4059	<code>EllipticCurveWith</code>	
<code>TwoSelmerGroup(E)</code>	4060	<code>GoodReductionSearch(S, Effort)</code>	4069
<i>122.3.7 The Cassels-Tate Pairing</i>	4063	122.4 Curves over p-adic Fields	4070
<i>122.3.8 Elliptic Curve Chabauty</i>	4063	<i>122.4.1 Local Invariants</i>	4070
<code>Chabauty(MWmap, Ecov)</code>	4063	<code>Conductor(E)</code>	4070
<code>Chabauty(MWmap, Ecov, p)</code>	4064	<code>LocalInformation(E)</code>	4071
<i>122.3.9 Auxiliary Functions for Etale Algebras</i>	4067	<code>RootNumber(E)</code>	4071
		122.5 Bibliography	4071

Chapter 122

ELLIPTIC CURVES OVER \mathbf{Q} AND NUMBER FIELDS

122.1 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 however, the functionality available over \mathbf{Q} is far ahead of that available for general number fields, both in the range of functions and in the efficiency of implementations.

Functions declared for number fields may also be used over \mathbf{Q} : to do so, it is usually necessary to construct \mathbf{Q} as `RationalsAsNumberField()` rather than `Rationals()`.

122.2 Curves over the Rationals

122.2.1 Local Invariants

`Conductor(E)`

The conductor of the elliptic curve E defined over \mathbf{Q} .

`BadPrimes(E)`

Given an elliptic curve E defined over \mathbf{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)`

Given an elliptic curve E defined over \mathbf{Q} and a prime number p , this function returns the local Tamagawa number of E at p , which is the index in $E(\mathbf{Q}_p)$ of the subgroup $E^0(\mathbf{Q}_p)$ of points with nonsingular reduction modulo p . For any prime p that is of good reduction for E , this function returns 1.

`TamagawaNumbers(E)`

Given an elliptic curve E defined over \mathbf{Q} , this function returns the sequence of Tamagawa numbers at each of the bad primes of E , as defined above.

LocalInformation(E, p)

Given an elliptic curve E defined over \mathbf{Q} and a prime number p , this function returns the local information at the prime p as a tuple of the form $\langle P, vpd, fp, c_p, K, split \rangle$, 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)

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)

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.

FrobeniusTraceDirect(E, p)

Given a rational elliptic curve E and a prime p , this function provides an efficient way to directly compute the trace of Frobenius (without having to create $GF(p)$, coercing E into this field, etc). The argument p is **not** checked to be prime.

TracesOfFrobenius(E, B)

The sequence of traces of Frobenius $a_p(E)$, of the reduction mod p of E , for all primes p up to B . This function is very carefully optimised.

Example H122E1

```
> 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
```

122.2.2 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)`

Given an elliptic curve E defined over \mathbf{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)`

Given an elliptic curve E defined over \mathbf{Q} , this function returns the reduction types of E modulo the bad primes in the form of a sequence of Kodaira symbols.

`KodairaSymbol(s)`

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`

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`

The logical negation of `eq`.

Example H122E2

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> ]
```

122.2.3 Complex Multiplication

`HasComplexMultiplication(E)`

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.

122.2.4 Isogenous Curves

`IsogenousCurves(E)`

The set of curves \mathbf{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)`

Precision

RNGINTELT

Default :

Compute the Faltings height of a rational elliptic curve. This is given by $-\frac{1}{2} \log \text{Vol}(E)$ where $\text{Vol}(E)$ is the volume of the fundamental parallelogram.

Example H122E3

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

122.2.5 Mordell–Weil Group

The Mordell–Weil theorem states that for an elliptic curve E defined over \mathbf{Q} , the set $E(\mathbf{Q})$ of points on E with \mathbf{Q} -rational coordinates forms a finitely generated abelian group, known as the *Mordell–Weil group*.

To compute Mordell–Weil groups in MAGMA, there are several ways to proceed:

- (i) A large array of relevant tools (various descents, and other techniques) are described elsewhere in this chapter, and may be applied “by hand”.
- (ii) The routine `MordellWeilShaInformation` automates the process: it applies all relevant tools in a suitable order.
- (iii) The older routines `Rank` and `MordellWeilGroup` use an (independent) implementation of an algorithm similar to the famous program `mwrnk` by John Cremona. These routines do 2-descent or 2-isogeny descent, but use no additional techniques.

Warning: No algorithm has been implemented that is guaranteed to determine the Mordell–Weil rank! In some cases the functions `Rank` and `MordellWeilGroup` return a lower bound which is not known to be the true rank (or group). In cases where the rank is not proven, a warning message is printed the first time either `Rank` or `MordellWeilGroup` is called for a particular elliptic curve; on subsequent calls, the unproven result will be

returned without any warning. It is recommended to use `RankBounds` instead of `Rank`, to avoid confusion.

MordellWeilShaInformation(E: <i>parameters</i>)
--

DescentInformation(E: <i>parameters</i>)

RankOnly	BOOLELT	<i>Default : false</i>
ShaInfo	BOOLELT	<i>Default : false</i>
Silent	BOOLELT	<i>Default : false</i>

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 \mathbf{Q} . The tools used include 2-descent, 4-descent, Cassels-Tate pairings, 3-descent, and also analytic routines (rank and Heegner points) when the conductor of E is not too large. 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, the collected information is returned in three sequences.

The first sequence returned contains lower and upper bounds on the Mordell-Weil rank of $E(\mathbf{Q})$, 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 $\mathbf{Z}/n\mathbf{Z}$ is contained in $Sha(E)$, the tuple $\langle n, [1, u] \rangle$ 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.

Example H122E4

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]
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
```

TorsionSubgroup(E)

Given an elliptic curve E defined over \mathbf{Q} , this function 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. 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 $\{1..4\}$). If there are two generators then the first generator returned in this case will have order 2.

Rank(E: <i>parameters</i>)

MordellWeilRank(E: <i>parameters</i>)
--

Bound	RNGINTELT	<i>Default : 150</i>
-------	-----------	----------------------

Given an elliptic curve E defined over \mathbf{Q} , this function returns the rank of the Mordell–Weil group of E .

The general procedure to calculate the rank involves searching for rational points on certain homogeneous spaces represented by equations of the form $y^2 = ax^4 + bx^3 + cx^2 + dx + e$. The parameter **Bound** is a bound on the numerator and denominator of x that will be used when searching for such points.

RankBounds(E: <i>parameters</i>)

MordellWeilRankBounds(E: <i>parameters</i>)
--

Bound	RNGINTELT	<i>Default : 150</i>
-------	-----------	----------------------

Given an elliptic curve E defined over \mathbf{Q} , this computes and returns lower and upper bounds on the rank of the Mordell–Weil group of E . The parameter **Bound** is as described in the intrinsic **Rank**. This function will not generate the usual warning message if the upper and lower bounds are not equal.

MordellWeilGroup(E: <i>parameters</i>)

AbelianGroup(E: <i>parameters</i>)

Bound	RNGINTELT	<i>Default : 150</i>
-------	-----------	----------------------

HeightBound	RNGINTELT	<i>Default : 15</i>
-------------	-----------	---------------------

The Mordell–Weil group of an elliptic curve E defined over \mathbf{Q} . 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 parameter **Bound** is used during the rank computation part of the algorithm, and has the same meaning as described in the intrinsic **Rank**, above. The group computation involves searching for points on E up to a certain (naive) height. The parameter **HeightBound** is used to limit this search as the rigorous bounds may not be feasible. When the **HeightBound** is less than the rigorous bound, a warning message is printed. In this situation, the user may wish to also use **Saturation** (see below).

As explained above for **Rank**, the computed rank may be less than the actual rank. In cases where the rank is not proven, a warning message is printed (the first time). The best way to check whether the rank has been proven is to use **RankBounds**.

Generators(E)

Given an elliptic curve E defined over \mathbf{Q} , this function returns generators for the Mordell–Weil group of E , in the form of a sequence 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 `MordellWeilGroup`; the generators of the torsion subgroup will come first (in the order described in `TorsionSubgroup`), followed by the points of infinite order.

NumberOfGenerators(E)**Ngens(E)**

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

Saturation(points, n)

<code>TorsionFree</code>	BOOLELT	<i>Default</i> : false
<code>OmitPrimes</code>	[RNGINTELT]	<i>Default</i> : []
<code>Check</code>	BOOLELT	<i>Default</i> : true

Given a sequence of points on an elliptic curve E over the rationals, and an integer n , this function returns a sequence of points generating a subgroup of $E(\mathbf{Q})$ 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.

Example H122E5

```
> 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 H122E6

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 .

122.2.6 Heights and Height Pairing

These functions require that the corresponding elliptic curve has integral coefficients.

`NaiveHeight(P)`

`WeilHeight(P)`

Given a point $P = (a/b, c/d, 1)$ on an elliptic curve E defined over \mathbf{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)`

`CanonicalHeight(P: parameters)`

Precision

RNGINTELT

Default :

Given a point P on an elliptic curve E defined over \mathbf{Q} with integral coefficients, this function returns the canonical height $\hat{h}(P)$.

One definition of $\hat{h}(P)$ is $\hat{h}(P) = \lim_{n \rightarrow \infty} 4^{-n} h(2^n P)$, 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)`

Precision

RNGINTELT

Default :

Check

BOOLELT

Default : false

Renormalization

BOOLELT

Default : false

Given a point P on an elliptic curve E defined over \mathbf{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** vararg determines whether to check if the second argument is prime. The **Renormalization** vararg 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: <i>parameters</i>)

Precision RNGINTELT Default :

Given two points P, Q on the same elliptic curve defined over \mathbf{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: <i>parameters</i>)
--

HeightPairingMatrix(E: <i>parameters</i>)
--

Precision RNGINTELT Default :

Given a sequence of points on an elliptic curve defined over \mathbf{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)

Precision RNGINTELT Default :

Given a sequence of points S on an elliptic curve E over \mathbf{Q} , this function returns the determinant of the Néron–Tate height pairing matrix of the sequence.

Regulator(E)

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 H122E7

```
> E := EllipticCurve([0,0,1,-7,6]);
> P1, P2, P3 := Explode(Generators(E));
> Height(P1);
0.6682051656519279350331420509
> 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);
-0.655035947159686182497278069814
> LocalHeight(P2, 5077); // 0 + Log(5077)/6
```

```

1.42207930249123238829272871637
> LocalHeight(P2, 0 : Renormalization);
0.767043355331546205795450646552
> LocalHeight(P2, 5077 : Renormalization);
0.00000000000000000000000000000000
> Height(P2);
0.7670433553315462057954506466

```

The above shows that the local height at a bad prime may still be zero, at least in the renormalised value.

SilvermanBound(E)

Given an elliptic curve E over \mathbf{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(E: parameters)

Torsion

BOOLELT

Default : false

Given an elliptic curve E over \mathbf{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 H122E8

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);
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);
15.70818965430290161142481294545

```

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);
11.0309876231179839831829512652688
```

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)

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)

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)

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 $E(\mathbf{Q})/E_{tors}(\mathbf{Q})$ as points of S .

Example H122E9

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, 11690998742798553808334900]);
> P0 := E! [ 39860582, 2818809365988 ];
> P1 := E! [ 144658748, -946639447182 ];
> P2 := E! [ 180065822, 569437198932 ];
> P3 := E! [ -339374593, 2242867099638 ];
> P4 := E! [ -3492442669/25, 590454479818404/125 ];
> 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])));
467.6598587040659411808117253
> Log(Abs(Denominator(Q[1])));
449.2554587727840583949442765

```

<code>pAdicHeight(P, p)</code>

Precision	RNGINTELT	Default : 0
E2	FLDPADELTA	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)

Precision	RNGINTELT	Default : 0
E2	FLDPADELTA	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)

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.

Example H122E10

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 + 0(5^20)
5495832406058373*7 + 0(7^20)
-12403985313704524674*11 + 0(11^20)
616727731594753360389*13 + 0(13^20)
53144975867434108754867*17 + 0(17^20)
651478754033733420744924*19 + 0(19^20)
1382894029404692415656094*23^2 + 0(23^20)
-2615503441214747688800993518*29 + 0(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 + 0(5^20)
-35022392779944725 + 0(7^20)
-331747503271490921584 + 0(11^20)
246446095809126124462 + 0(13^20)
-1528527915797227515067270 + 0(17^20)
333884275695653846729970*19 + 0(19^20)
412978398356115570280760602 + 0(23^20)
-45973362301048046561945193743 + 0(29^20)
> pAdicRegulator([P1, P2, P3], 23); // dependent points
```

```

0(23^20)
> eisen_two := EisensteinTwo(E, 13 : Precision:=40); eisen_two;
25360291252705414983472710631099471724343012 + 0(13^40)
> pAdicRegulator([P1, P2], 13 : Precision:=40, E2:=eisen_two);
85894629213918025547455609490608727790695215 + 0(13^40)

```

122.2.7 Two-Descent and Two-Coverings

The two-descent process determines the locally soluble two-coverings of an elliptic curve defined over a number field K , giving them as hyperelliptic curves $C : y^2 = f(x)$ with f of degree four. To be practical, K must be of rather small degree, and for various parts of 2-descent to run (particularly reduction), K must have certain properties (such as being totally real). Also, the algorithms over the rationals have now been revisited, and are now faster than when using the general number field descent machinery.

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 provides some tools for manipulating such curves, including generators for some rings of invariants of quartic forms, minimisation and reduction of such forms, and the maps back to the associated elliptic curve. This functionality overlaps with the package for genus one models (see 124). There is a straightforward interface to the 2-Selmer group machinery, which also works over number fields.

TwoDescent(E: <i>parameters</i>)		
RemoveTorsion	BOOLELT	Default : false
RemoveGens	{PTELL}	Default : {}
WithMaps	BOOLELT	Default : true
Verbose	TwoDescent	Maximum : 1

Given an elliptic curve E over the rationals 2-descent can be used to construct a sequence of hyperelliptic curves (quartics) representing the elements of the 2-Selmer group. The group structure is not preserved by this function: the intrinsic `TwoSelmerGroup` should be used if this is desired. If `RemoveTorsion` is `true`, the generators of the torsion subgroup are factored out from the set. If `RemoveGens` is nonempty, the image of the specified points is factored out from the set. If `AppendMaps` is `true`, then the maps from the covers to the curve are appended as a second argument.

AssociatedEllipticCurve(f)

AssociatedEllipticCurve(C)

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 .

Example H122E11

```
> SetSeed(1); // results may depend slightly on the seed
> E := EllipticCurve([0, 1, 0, -7, 6]);
> S := TwoDescent(E);
> S;
[
  Hyperelliptic Curve defined by  $y^2 = x^4 + 4x^3 - 2x^2 - 20x + 9$  over
  Rational Field,
  Hyperelliptic Curve defined by  $y^2 = x^4 - x^3 - 2x^2 + 2x + 1$  over
  Rational Field,
  Hyperelliptic Curve defined by  $y^2 = 2x^4 - 4x^3 - 8x^2 + 4x + 10$  over
  Rational Field
]
```

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.
> pt_on_S2 := Points(S[2] : Bound:=10 )[1];
> _, phi := AssociatedEllipticCurve(S[2] : E:=E );
> phi( pt_on_S2 );
(-3 : 3 : 1)
```

122.2.7.1 Two Descent Using Isogenies

`TwoIsogenyDescent(E : parameters)`

<code>Isogeny</code>	<code>MAPSCH</code>	<i>Default :</i>
<code>TwoTorsionPoint</code>	<code>PTELL</code>	<i>Default :</i>

Given an elliptic curve E over \mathbf{Q} admitting a 2-isogeny $\phi : E' \rightarrow E$, this function computes 2-coverings representing the nontrivial elements of the Selmer groups of ϕ and of the dual isogeny $\phi' : E \rightarrow E'$. These coverings are given as hyperelliptic curves $C : y^2 = \text{quartic}(x)$. Six objects are returned: (i) the sequence of coverings C of E for ϕ ; (ii) the corresponding list of maps $C \rightarrow E$; (iii) and (iv) the coverings C' and maps $C' \rightarrow E'$ for ϕ' ; (v) and (vi) the isogenies ϕ and ϕ' that were used.

It's advisable to give a model for E that is close to minimal.

`LiftDescendant(C)`

This routine performs a higher descent on curves arising in `TwoIsogenyDescent(E)` on an elliptic curve E over \mathbf{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 $D \rightarrow E$ 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 $D \rightarrow C$, and lastly the covering map $C \rightarrow E$ from the given curve to some model of its associated elliptic curve.

122.2.7.2 Invariants

`QuarticIInvariant(q)`

`QuarticJInvariant(q)`

`QuarticG4Covariant(q)`

`QuarticG6Covariant(q)`

`QuarticHSeminvariant(q)`

`QuarticPSeminvariant(q)`

`QuarticQSeminvariant(q)`

`QuarticRSeminvariant(q)`

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)`

Using invariant theory, compute the number of real roots of a real quartic polynomial q .

`QuarticMinimise(q)`

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 124. The algorithm can be found in [CFS10].

`QuarticReduce(q)`

Given a quartic q , the algorithm of [Cre99] is applied to find the reduced quartic and the matrix that reduced it.

`IsEquivalent(f,g)`

Determines if the quartics f and g are equivalent.

122.2.8 The Cassels-Tate Pairing

The Tate–Shafarevich group of any elliptic curve E admits an alternating bilinear form on $\text{III}(E)$ with values in \mathbf{Q}/\mathbf{Z} , known as the Cassels-Tate pairing. The key property is that if $\text{III}(E)$ 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 $\text{III}(E)[2]/2\text{III}(E)[4]$, or equivalently on $\text{Sel}^2(E)$ modulo the image of $\text{Sel}^4(E)$, with values in $\mathbf{Z}/2\mathbf{Z}$.

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 $\text{III}(E)$, 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 $\text{III}(E)[4]/2\text{III}(E)[8]$, the values of the pairing between this element and all elements in $\text{III}(E)[2]/2\text{III}(E)[4]$ provides the same information as performing an 8-descent on C . These elements may be represented by a 4-covering $C \rightarrow E$ and a 2-covering $D \rightarrow E$ respectively.

In MAGMA the pairing between 2-coverings is implemented over \mathbf{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 \mathbf{Q} . A new, very efficient implementation of pairing on 2-coverings over \mathbf{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 \mathbf{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`.

122.2.8.1 Verbose Information

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 $n = 2$.

<code>CasselsTatePairing(C, D)</code>

Verbose

CasselsTate

Maximum : 2

This evaluates the Cassels-Tate pairing on 2-coverings of an elliptic curve over \mathbf{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 $\mathbf{Z}/2\mathbf{Z}$ (returned as elements of \mathbf{Z}).

<code>CasselsTatePairing(C, D)</code>

Verbose

CasselsTate

Maximum : 2

This evaluates the Cassels-Tate pairing between a 4-covering C and a 2-covering D of the same elliptic curve over \mathbf{Q} . The arguments should be curves over \mathbf{Q} , with C an intersection of two quadrics in \mathbf{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 \mathbf{Q} (otherwise the pairing is not defined).

The pairing takes values in $\mathbf{Z}/2\mathbf{Z}$ (returned as elements of \mathbf{Z}).

Example H122E12

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  $y^2 = -4x^4 + 60x^3 - 232x^2 + 52x - 3$ 
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 $\text{III}(E)$ is precisely $\mathbf{Z}/2 \times \mathbf{Z}/2$. We could have reached the same conclusion using 4-descent:

```

> time FourDescent(covers[1]);
[]
Time: 0.460

```

122.2.9 Four-Descent

In a 1996 paper [MSS96], Merriman, Siksek and Smart described a four-descent algorithm for elliptic curves over \mathbf{Q} . This section describes the MAGMA implementation of that algorithm. Another useful reference is Tom Womack’s PhD thesis [Wom03]. Four-descent is performed on a two-cover, that is a hyperelliptic curve defined by a polynomial of degree four, with the assumption that the quartic of this descendant does not have a rational root. The terminology “four-descent” is thus slightly incorrect; MAGMA actually performs a second descent on a specific two-cover, and does not try to combine such information from all two-covers into the 4-Selmer group.

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 \mathbf{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 .

<code>FourDescent(f : parameters)</code>
--

<code>FourDescent(C : parameters)</code>
--

<code>RemoveTorsion</code>	<code>BOOLELT</code>	<i>Default : false</i>
<code>IgnoreRealSolubility</code>	<code>BOOLELT</code>	<i>Default : false</i>
<code>RemoveTorsion</code>	<code>BOOLELT</code>	<i>Default : false</i>
<code>RemoveGensEC</code>	<code>{PTELL}</code>	<i>Default : {}</i>
<code>RemoveGensHC</code>	<code>{PTHYP}</code>	<i>Default : {}</i>

Verbose	FourDescent	Maximum : 3
Verbose	LocalQuartic	Maximum : 2
Verbose	MinimiseFD	Maximum : 2
Verbose	QISearch	Maximum : 1
Verbose	ReduceFD	Maximum : 2
Verbose	QuotientFD	Maximum : 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 H122E13

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 $(\mathbf{Z}/2\mathbf{Z})^2$.

```
> 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 III, and the Mordell–Weil rank must be zero.

AssociatedEllipticCurve(qi)

AssociatedHyperellipticCurve(qi)

E

CRV_{ELL}*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)

QuadricIntersection(P, F)

Given a pair of symmetric 4×4 matrices F , this function returns the associated quadric intersection in $P = P^3$.

QuadricIntersection(E)

QuadricIntersection(C)

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)

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 : <i>parameters</i>)

OnlyOne

BOOLELT

Default : false

ExactBound

BOOLELT

Default : false

Verbose

QISearch

Maximum : 1

Given a quadric intersection C , this function searches, by a reasonably efficient method due to Elkies [Elk00], for a point on C of naïve height up to B ; the asymptotic running time is $O(B^{2/3})$.

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 \mathbf{Z} will be found first. If **ExactBound** is set to **true**, then points that are found with height larger than B will be ignored.

TwoCoverPullback(H, pt)

TwoCoverPullback(f, pt)

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)

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 H122E14

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 `mwrnk`; 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 :
20326609223460937753264735467/1050397899358266605692672489 : 1)
> Height($1);
44.19679596739622477261983370
```

122.2.10 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 \mathbf{Q} . These are nonsingular intersections of two quadrics in \mathbf{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 \mathbf{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 Stamerger.

<code>EightDescent(C : parameters)</code>

<code>BadPrimesHypothesis</code>	<code>BOOLELT</code>	<i>Default : false</i>
<code>DontTestLocalSolvabilityAt</code>	<code>{RNGINTELT}</code>	<i>Default : {}</i>
<code>StopWhenFoundPoint</code>	<code>BOOLELT</code>	<i>Default : false</i>
<code>Verbose</code>	<code>EightDescent</code>	<i>Maximum : 4</i>
<code>Verbose</code>	<code>LegendresMethod</code>	<i>Maximum : 3</i>

For a curve C obtained from `FourDescent` on some elliptic curve E over \mathbf{Q} , this performs a further 2-descent on C . It returns a sequence of curves D , together with a sequence containing maps $D \rightarrow C$ 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 "LegendresMethod" to 2.

122.2.11 Three-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.

There is also a separate implementation of “descent by three isogenies”, for elliptic curves over the rationals which admit a \mathbf{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 \mathbf{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 $H^1(\mathbf{Q}, E[3])$ (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 $H^1(\mathbf{Q}, E[3])$ 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.

<code>ThreeDescent(E : parameters)</code>

Method	MONSTGELT	Default : “HessePencil”
Verbose	Selmer	Maximum : 3
Verbose	ThreeDescent	Maximum : 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

$C \rightarrow E$. 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 $C \rightarrow E$ differ by composition with the negation map $E \rightarrow E$.)

The function simply calls `ThreeSelmerGroup(E)`, and then `ThreeDescentCubic` on the Selmer elements. Note that if `ThreeSelmerGroup(E)` has already been computed, it will not be recomputed, because the results are stored in the attribute `E.ThreeSelmerGroup`.

For more information see the description of `ThreeDescentCubic` below.

Example H122E15

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 $\mathbf{Z}/3\mathbf{Z} \oplus \mathbf{Z}/3\mathbf{Z}$, 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 124.

ThreeSelmerGroup(E : *parameters*)

ThreeTorsPts	TUP	<i>Default :</i>
MethodForFinalStep	MONSTGELT	<i>Default : “UseSUnits”</i>
CompareMethods	BOOLELT	<i>Default : false</i>
Verbose	Selmer	<i>Maximum : 3</i>

Given an elliptic curve E over the rationals, this function returns its 3-Selmer group as an abelian group, together with a map to the natural affine algebra.

The parameter **MethodForFinalStep** can be either “Heuristic” (which is usually much faster in large examples), “FindCubeRoots”, or “UseSUnits” (the default, which has the advantage that it usually takes roughly the same amount of time as the S -unit computation that has already been performed). To try all three methods and compare the times, set **CompareMethods** to **true**.

Most of the computation time is usually spent on class group and unit group computations. 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 Section 37.6.1). The other way is to precompute the class groups of the fields involved, setting the optional parameters in **ClassGroup** as desired. (The relevant fields can be obtained as the fields over which the points in **ThreeTorsionPoints**(E) are defined.) The fields and their class group data would then be stored internally, and automatically used when **ThreeSelmerGroup**(E) is called (even when no optional parameters are provided).

When **ThreeTorsPts** is specified, it determines the fields used in the computation, and the algebra used to express the answers.

ThreeDescentCubic(E , α : *parameters*)

ThreeTorsPts	TUP	<i>Default :</i>
Method	MONSTGELT	<i>Default : “HessePencil”</i>
Verbose	ThreeDescent	<i>Maximum : 3</i>

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 $C \rightarrow E$. The cubic is a principal homogeneous space for E , and the covering map $C \rightarrow E$ 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).

<code>ThreeIsogenyDescent(E : parameters)</code>		
--	--	--

<code>Isog</code>	<code>MAPSCH</code>	<i>Default :</i>
<code>Verbose</code>	<code>Selmer</code>	<i>Maximum : 3</i>
<code>Verbose</code>	<code>ThreeDescent</code>	<i>Maximum : 3</i>

Given an elliptic curve E over \mathbf{Q} that admits a \mathbf{Q} -rational isogeny $E \rightarrow E_1$ 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 $Sel(E \rightarrow E_1)$, a corresponding list of covering maps to E_1 , a list of curves for the dual isogeny Selmer group $Sel(E_1 \rightarrow E)$, a corresponding list of covering maps to E , and finally the isogeny $E \rightarrow E_1$.

This function works simply by calling `ThreeIsogenySelmerGroups(E)`, and then calling `ThreeIsogenyDescentCubic` for each Selmer element.

The isogeny $E \rightarrow E_1$ 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.

<code>ThreeIsogenySelmerGroups(E : parameters)</code>		
---	--	--

<code>Isog</code>	<code>MAPSCH</code>	<i>Default :</i>
<code>Verbose</code>	<code>Selmer</code>	<i>Maximum : 3</i>

Given an elliptic curve E over the rationals that admits a \mathbf{Q} -rational isogeny $E \rightarrow E_1$ of degree 3, the function computes the Selmer groups associated to the isogeny, and to its dual isogeny $E_1 \rightarrow E$. 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 $E \rightarrow E_1$, the group, and the map, for $E_1 \rightarrow E$, and finally the isogeny $E \rightarrow E_1$.

A bound for the rank of $E(\mathbf{Q})$ 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 $E \rightarrow E_1$ 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 étale 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 \mathbf{Q} .

ThreeIsogenyDescentCubic(ϕ, α)

Verbose

ThreeDescent*Maximum : 3*

Given an isogeny ϕ of degree 3 between elliptic curves over \mathbf{Q} , and any element α of $H^1(\mathbf{Q}, E[\phi])$, this function returns a plane cubic curve C representing α , together with a covering map $C \rightarrow E$ 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`. $H^1(\mathbf{Q}, E[\phi])$ is represented as the subgroup of $A^\times / (A^\times)^3$ consisting of elements whose norm is a cube.

ThreeDescentByIsogeny(E)

Verbose

Selmer*Maximum : 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 H122E16

```
> 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 III 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 III (E/\mathbf{Q}).

```
> time Sel3 := ThreeDescentByIsogeny(E);
```

```
Time: 5.280
> #Sel3*2+1;
9
```

This shows that $\text{III}(E/\mathbf{Q})[3] = 0$ (since we know the rank is 2). Alternatively one could get the same by calling `ThreeDescent`. This would require computing class and unit group information in the sextic field below (and would be much slower).

```
> E3reps := ThreeTorsionPoints(E);
> Parent(E3reps[2,1]);
Number Field with defining polynomial x^6 + 14174028 over the Rational Field
> time Sel3_slow := ThreeDescent(E);
forever!
```

Jacobian(C)

Given the equation of a nonsingular projective plane cubic curve C over the rationals, this function returns the Jacobian of C (over the rationals) as an elliptic curve. Note that C will be a principal homogeneous space of E .

ThreeSelmerElement(E, C)

ThreeSelmerElement(C)

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 $H^1(\mathbf{Q}, E[3])$ that is represented by a covering $C \rightarrow E$ that takes the flex points of C to O . Note that α is only determined up to inverse in $H^1(\mathbf{Q}, E[3])$.

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)

<code>E</code>	<code>CRVELL</code>	<i>Default :</i>
<code>ReturnBoth</code>	<code>BOOLELT</code>	<i>Default : false</i>

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 $H^1(\mathbf{Q}, E[3])$. The sum is returned as another plane cubic, if possible (and otherwise an error results).

An element of $H^1(\mathbf{Q}, E[3])$ 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 $H^1(\mathbf{Q}, E[3])$ as given by `ThreeSelmerElement`, adding the cocycles, and then converting the result to a cubic.

Note that a cubic only determines an element of $H^1(\mathbf{Q}, E[3])$ 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)`

For an elliptic curve E over the rationals, this function classifies the Galois action on $E[3]$. The possibilities are “Generic”, “2Sylow”, “Dihedral”, “Generic3Isogeny”, “ $\mathbf{Z}/3\mathbf{Z}$ -nonsplit”, “ μ_3 -nonsplit”, “Diagonal” and “ $\mu_3 + \mathbf{Z}/3\mathbf{Z}$ ”.

`ThreeTorsionPoints(E : parameters)`

`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)`

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 \mathbf{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.

122.2.11.1 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 \mathbf{P}^5 , and of degree 12 in \mathbf{P}^1 , respectively).

The algorithms are described in [Fis08].

`SixDescent(C2, C3)`

`SixDescent(model2, model3)`

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 $C6 \rightarrow C3$ is also returned.

TwelveDescent(C3, C4)

TwelveDescent(model3, model4)

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 $C12 \rightarrow C4$ are also returned.

122.2.12 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 \mathbf{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 étale 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 III).

NineDescent(C : parameters)

ExtraReduction	RNGINTELT	Default : 10
Verbose	Selmer	Maximum : 3
Verbose	ComputeL	Maximum : 2

Given a plane cubic curve C over \mathbf{Q} this returns a sequence of curves in \mathbf{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 $C(\mathbf{Q}) = \emptyset$ 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)

Computes the 3-rank of the 3-Selmer set of plane cubic curve C defined over \mathbf{Q} . The value -1 is returned when the 3-Selmer set is empty. In this case $C(\mathbf{Q}) = \emptyset$ 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.

122.2.13 p -Isogeny Descent

Given an isogeny $\phi : E_1 \rightarrow E_2$ 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 ϕ^\vee . In the case of elliptic curves, performing ϕ - and ϕ^\vee -descents can be used to bound the Mordell-Weil rank and get information on III. For general genus one curves descent by isogeny can be used to rule out divisibility in III 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 \mathbf{Q} -rational point of order $p \in \{5, 7\}$ the dimensions of the ϕ - and ϕ^\vee -Selmer groups can be computed using an algorithm of Fisher [Fis00] and [Fis01]. This method also produces models for the everywhere locally solvable ϕ^\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 ϕ^\vee -Selmer set of C can be computed using an algorithm of Creutz described in [Cre10] and [CM12]. Computing the ϕ^\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 \mathbf{P}^{p-1} . If the kernel of the isogeny is generated by a \mathbf{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)

`pIsogneyDescent(E,p)`

`pIsogenyDescent(lambda,p)`

This performs a ϕ -descent on an elliptic curve over \mathbf{Q} using the algorithm of Fisher. The descent requires an isogeny ϕ whose kernel is generated by a \mathbf{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 \mathbf{Q} -rational point on the modular curve $X_1(p)$. In the final case the choice for the coordinate λ on $X_1(p)(\mathbf{Q})$ is as described in [Fis00].

The return values are the p -ranks of the ϕ - and ϕ^\vee -Selmer groups, a sequence of genus one normal curves of degree p representing the inverse pairs of nontrivial elements of the ϕ^\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)`

`pIsogenyDescent(C,E1,E2)`

`pIsogenyDescent(C,P)`

Verbose

Selmer

Maximum : 3

This performs an isogeny descent on the genus one normal curve C of degree $p \in \{3, 5\}$ as described in [CM12]. C must be a projective plane cubic (in case $p = 3$) or an intersection of 5 quadrics in \mathbf{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 \mathbf{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)`

`FakeIsogenySelmerSet(C,E1,E2)`

`FakeIsogenySelmerSet(C,P)`

Verbose

Selmer

Maximum : 3

This determines the \mathbf{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 H122E17

We use a 5-isogeny descent to show that the elliptic curve with Cremona reference 57013 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 $E_2(\mathbf{Q})/\phi(E_1(\mathbf{Q})) \simeq 0$, while the ϕ^\vee -Selmer group gives an exact sequence: $0 \rightarrow E_2(\mathbf{Q})/\phi^\vee(E_1(\mathbf{Q})) \subset (\mathbf{Z}/5\mathbf{Z})^3 \rightarrow \text{III}(E_2/\mathbf{Q})[\phi^\vee] \rightarrow 0$. From this we conclude that the rank (for both curves) is 0, and so $\text{III}(E_2/\mathbf{Q})[\phi^\vee]$ has \mathbf{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 $\text{III}(E_1/\mathbf{Q})$ 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 III .

```
> time DimSelC := FakeIsogenySelmerSet(C,E1,E2);
Time: 2.950
> DimSelC;
-1
```

Example H122E18

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);
242.013813460415708000138268958 1
Time: 70.180
> Conductor(E);
271976526950
> ThreeTorsionType(E);
Generic
```

This shows that $E(\mathbf{Q}) \simeq \mathbf{Z} \times \mathbf{Z}/5\mathbf{Z}$ and that (assuming $\text{III}(E/\mathbf{Q})$ 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;
(18742046893875394386310714878805837118945751672332464430219783625592
23533610794664163106345898123969229661/991466716729115905824131485387
000945412007497180441812893340644055454270710030810641619997008843128
1 : 71245662543288432230853647434377610311616709098508077082874898143
715069493690489458124709392518651254352942841744887961253603235705564
184725054480767436761578/98722742040021206194215985208166099089985806
405143789926209543140373470096170812446567353837866650186446834616709
6101096776119973657047069330277618071 : 1)
> CanonicalHeight(P2);
242.013813460415708000138268957
```

Example H122E19

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));
true
> time FakeIsogenySelmerSet(C,phidual);
2
Time: 0.620
> time SelC := pIsogenyDescent(C,phidual);
Time: 1.160
> Ilog(3,#SelC);
1
```

122.2.14 Heegner Points

For an elliptic curve of rank 1, it is possible to compute the 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, so in practice it works best for curves contrived to have small conductors.

The calculation proceeds in three stages: choosing the quadratic field $\mathbf{Q}(\sqrt{-d})$, 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 : <i>parameters</i>)		
NaiveSearch	RNGINTELT	Default : 1000
Discriminant	RNGINTELT	Default :
Cover	CRV	Default :
DescentPossible	BOOLELT	Default : true

IsogenyPossible	BOOLELT	<i>Default : true</i>
Traces	SEQENUM	<i>Default : []</i>
Verbose	Heegner	<i>Maximum : 1</i>

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**. The parameter **NaiveSearch** indicates to what height the algorithm will first do a search (using **Points**) before turning to the analytic method. 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)

HeegnerPoint(f : parameters)

NaiveSearch	RNGINTELT	<i>Default : 10000</i>
Discriminant	RNGINTELT	<i>Default :</i>
Traces	SEQENUM	<i>Default : []</i>
Verbose	Heegner	<i>Maximum : 3</i>

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 \mathbf{P}^3 . These functions call **HeegnerPoint** on the underlying elliptic curve. They then map the computed point back to the given covering curve. The rational reconstruction step of the **HeegnerPoint** algorithm can be quite time-consuming for some coverings, especially if the index is large. Also, if a cover corresponds to an element of the Tate–Shafarevich group, the algorithm will likely enter an infinite loop. These functions have not been as extensively tested as the ordinary **HeegnerPoint** function, and thus occasionally might fail due to unforeseen problems.

ModularParametrization(E, z, B : <i>parameters</i>)		
ModularParametrization(E, z : <i>parameters</i>)		
ModularParametrization(E, Z, B : <i>parameters</i>)		
ModularParametrization(E, Z : <i>parameters</i>)		
Traces	SEQENUM	Default : []
ModularParametrization(E, f, B : <i>parameters</i>)		
ModularParametrization(E, f : <i>parameters</i>)		
ModularParametrization(E, F, B : <i>parameters</i>)		
ModularParametrization(E, F : <i>parameters</i>)		
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)		
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 : <i>parameters</i>)		
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 $E(\mathbf{Q})$. 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)

`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)

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)

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)

`ReturnPoint` BOOLELT *Default* : false
`Precision` RNGINTELT *Default* : 100
`Verbose` Heegner *Maximum* : 1

Given an elliptic curve E over \mathbf{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 $\mathbf{Q}(\sqrt{D})$, and the class number $h(D)$ must equal either $m \deg(p_D)$ or $2m \deg(p_D)$.

The conductor of the order $\mathbf{Q}(\sqrt{D})$ 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 H122E20

```
> 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 (11003637829478432203592984661004129/1048524607168660222036584535396 :
-1004181871409718654255966342764883958203316448270339/10736628855783147946
99393270058310986889998056 : 1)
Time: 1.380
```

Example H122E21

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);
373.478661569142379884077480412
```

Example H122E22

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  $y^2 = -2896x^4 - 57928x^3 - 202741x^2$ 
+  $868870x - 651725$  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;
(26935643239674824824611869793601774003477303945223677741244455058753
924460587724041316046901475913814274843012625394997597714766923750555
669810857471061235926971161094484197799515212721830555528087646969545
65/837619099331786545303500955405701693904657590793269053332825491870
645799230089011267382251975387071464373622714525213870033427638236014
2288012504288439077587496501436920044109243898570190931925860244164 :
410189886613094359515065087530226951251222017120181558101276037601794
678635473792334597914052557787798899390943937170051840037860568689664
871351793404398352109768736545284569745952273382778021947446766526118
621612003004627401344216069791103330332004546699363266079516476593864
98538303998567379869974143259174395/766601839981278884919411893932635
388671474045058772153268571977045787439290021029065740244648077391646
579430077900352635000328805236464794479797855430820809227462762666048
009219642700664370632930446228302292313415544188481623273194456758440
446505454248062292834244615276350323795396202280072306278575288 : 1)
> Height(PT);
476.811182818720336949724781780
> ConjecturalRegulator(E : Precision := 5);
476.81 1
```

Example H122E23

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 $E(\mathbf{Q})$. 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 $\mathbf{Q}(\sqrt{-327})$, 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 H122E24

```

> E := EllipticCurve([0,-1,0,-116,-520]);
> Conductor(E);
1460
> ConjecturalRegulator(E);
4.48887474770666173576726806264 1
> HeegnerDiscriminants(E,-200,-100);
[ -119, -111 ]
> P := HeegnerPoints(E,-119);
> P;
<1227609449333996689*t^10 - 106261506377143984603*t^9 +
  2459826667693945203684*t^8 - 12539974356047058417320*t^7 -
  298524708823654411408343*t^6 + 4440876684434597926161175*t^5 +
  7573549099120618979833241*t^4 - 393938048860406386108113130*t^3 -
  215318107135691628298668863*t^2 + 13958289016298162706904004974*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 $\mathbf{Q}(-119)$ 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
```

122.2.15 Analytic Information

Periods(E : *parameters*)

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, though the exact normalisation is not always given.

EllipticCurveFromPeriods(om : *parameters*)

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 vararg **Epsilon** indicating how close to integers the computed $-27c_4$ and $-54c_6$ need to be.

RealPeriod(E : *parameters*)

Precision

RNGINTELT

Default :

Returns the real period of the Weierstrass \wp -function associated to the elliptic curve E to **Precision** digits.

EllipticExponential(E, z)

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. 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, S)

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: parameters)**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 $-\omega_1/2 \leq \text{Re}(\phi(P)) < \omega_1/2$ and $-\omega_2/2 \leq \text{Im}(\phi(P)) < \omega_2/2$. The value is returned to **Precision** digits. As with **Periods**, a non-minimal model can be given, and the **EllipticLogarithm** will be computed with respect to it. The algorithm is again an AGM-trick due to Mestre.

EllipticLogarithm(E, S)**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)**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 H122E25

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;
[ 1.656947605210186238868298175 + -1.785440180067681418618695947*I,
```


ModularDegree(E)

Verbose

ModularDegree

Maximum : 1

Determine the modular degree of a rational elliptic curve E . The algorithm used is described in [Wat02]. One computes the special value $L(\text{Sym}^2 E, 2)$ of the motivic symmetric-square L -function of the elliptic curve, and uses the formula

$$\text{deg}(\text{phi}) = L(\text{Sym}^2 E, 2) / (2 * \text{Pi} * \Omega) * (Nc^2) * E_p(2)$$

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 [SW02], 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 H122E28

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 +
0(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
```

122.2.16 Integral and S -integral Points

Let E be an elliptic curve defined over the rational numbers \mathbf{Q} and denote by $S = \{p_1, \dots, p_{s-1}, \infty\}$ 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.

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)

FBasis	[PTELL]	<i>Default :</i>
SafetyFactor	RNGINTELT	<i>Default :</i>

Given an elliptic curve E over the \mathbf{Q} , this returns a sequence containing all the integral points on E , modulo negation. Secondly, a sequence is returned containing representations of the same points in terms of a fixed basis of the Mordell-Weil group (each such representation is a sequence of tuples of the form $[\langle P_i, n_i \rangle]$).

The algorithm involves first computing generators of the Mordell-Weil group, by calling `MordellWeilShaInformation` which accesses the appropriate tools available in MAGMA. 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 height bound is increased in such a way that the search region expands by roughly the specified factor.)

SIntegralPoints(E, S)

FBasis	[PTELL]	<i>Default :</i>
SafetyFactor	RNGINTELT	<i>Default :</i>

Given an elliptic curve E over the rationals and a set S of finite primes, returns the sequence of all S -integral points, modulo negation. The second return value, and the optional arguments `FBasis` and `SafetyFactor`, are the same as in `IntegralPoints`.

Example H122E29

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 H122E30

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)
(1534/81 : -42020/729 : 1)
(8516/117649 : -1163623840/40353607 : 1)
```

IntegralQuarticPoints(Q)

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)

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)

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 H122E31

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)

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)

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 H122E32

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 ]
]
```

122.2.17 Elliptic Curve Database

MAGMA includes John Cremona's database of all elliptic curves over \mathbf{Q} of small conductor (up to 200 000 as of December 2011). 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)

CremonaDatabase(: parameters)

BufferSize

RNGINTEL

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)`

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)`

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)`

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`

`NumberOfCurves(D)`

Returns the number of elliptic curves stored in the database.

`NumberOfCurves(D, N)`

Returns the number of elliptic curves stored in the database for conductor N .

`NumberOfCurves(D, N, i)`

Returns the number of elliptic curves stored in the database in the i -th isogeny class for conductor N .

`NumberOfIsogenyClasses(D, N)`

Returns the number of isogeny classes stored in the database for conductor N .

`EllipticCurve(D, N, I, J)`

`EllipticCurve(D, N, S, J)`

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)`

`EllipticCurve(S)`

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)

Returns a random curve from the database.

CremonaReference(D, E)

CremonaReference(E)

Returns the database reference to the minimal model for E (e.g., "101a1"). E must be defined over \mathbf{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 H122E33

```
> D := CremonaDatabase();
> #D;
847550
> minC, maxC := ConductorRange(D);
> minC, maxC;
1 130000
> &+[ NumberOfCurves(D, C) : C in [ minC .. maxC ] ];
847550
```

These numbers agree (which is nice). The conductor in that range with the most curves is 100800.

```
> S := [ NumberOfCurves(D, C) : C in [ minC .. maxC ] ];
> cond := maxval + minC - 1 where _,maxval := Max(S);
> cond;
100800
> NumberOfCurves(D, cond);
924
> NumberOfIsogenyClasses(D, cond);
418
```

The unique curve of conductor 5077 has rank 3.

```
> NumberOfCurves(D, 5077);
1
> E := EllipticCurve(D, 5077, 1, 1);
> E;
Elliptic Curve defined by  $y^2 + y = x^3 - 7x + 6$  over Rational Field
> CremonaReference(D, E);
5077a1
> Rank(E);
3
```

EllipticCurves(D, N, I)

EllipticCurves(D, N, S)

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)

The sequence of elliptic curves for conductor N from the database.

EllipticCurves(D, S)

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)

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 H122E34

Here are two ways to iterate through the database:

```
> D := CremonaDatabase();
25508696848625044861003843159331265666415824
Time: 21.130
> sum := 0;
> time for E in D do sum += Discriminant(E); end for;
> sum;
25508696848625044861003843159331265666415824
Time: 20.060
```

Now we create a random curve and find the other curves in its isogeny class.

```
> E := Random(D);
> E;
Elliptic Curve defined by  $y^2 = x^3 - 225x$  over Rational Field
> Conductor(E);
7200
> CremonaReference(E);
7200bg1
> EllipticCurves(D, "7200bg");
[
  Elliptic Curve defined by  $y^2 = x^3 - 225x$  over Rational Field,
  Elliptic Curve defined by  $y^2 = x^3 - 2475x - 47250$  over Rational Field,
  Elliptic Curve defined by  $y^2 = x^3 - 2475x + 47250$  over Rational Field,
  Elliptic Curve defined by  $y^2 = x^3 + 900x$  over Rational Field
]
```

122.3 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 \mathbf{Q} , with functions having similar names and arguments.

The main items implemented are Tate's algorithm, 2-descent and descent by 2-isogenies, the Cassels-Tate pairing, height machinery, and analytic tools.

122.3.1 Local Invariants

The routines listed here, when nontrivial, are based on an implementation of Tate's algorithm.

Conductor(E)

The conductor is part of the data computed by **LocalInformation** (described below).

BadPlaces(E)

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)

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)

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 than P , the returned model will still be integral or minimal if the given curve was.

LocalInformation(E)

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)`

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.

122.3.2 Complex Multiplication

`HasComplexMultiplication(E)`

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.

122.3.3 Mordell–Weil Groups

`TorsionBound(E, n)`

Given an elliptic curve E defined over a number field, returns a bound on the size of the torsion subgroup of E by looking at the first n non-inert primes of sufficiently low ramification degree and good reduction.

`pPowerTorsion(E, p)`

Bound

RNGINTELT

Default : -1

Given an elliptic curve E defined over a number field or \mathbf{Q} , returns the p -power torsion of E over its base field as an abelian group, together with the map from the group into the curve. One can specify a bound on the size of the p -power torsion, which may help the computation.

`TorsionSubgroup(E)`

Given an elliptic curve E defined over a number field, returns the torsion subgroup of E .

The algorithm involves using `TorsionBound` (described above).

`MordellWeilShaInformation(E: parameters)`

`DescentInformation(E: parameters)`

RankOnly

BOOLELT

Default : `false`

ShaInfo

BOOLELT

Default : `false`

Silent

BOOLELT

Default : `false`

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 elliptic curve E . The tools used include 2-descent and

the Cassels-Tate pairing on the 2-Selmer group; analytic routines may also be used when the conductor has small norm.

The arguments and returned values are the same as for curves over \mathbf{Q} , and are described in Section 122.2.5.

Note: The function `PseudoMordellWeilGroup` is obsolete and should not be used; this function should be used instead.

RankBound(E)

Isogeny

MAP

Default :

The upper bound on the rank of $E(K)$, for an elliptic curve over a number field K (or \mathbf{Q}), obtained by computing the Selmer group(s) associated to some isogeny on E . The isogeny is either multiplication by 2, or a 2-isogeny if any exist, and may be specified by the optional parameter **Isogeny**.

122.3.4 Heights

NaiveHeight(P)

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)

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)

Compute the height pairing matrix for an array of points. Same parameters as above.

LocalHeight(P, Pl : parameters)

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.

122.3.5 Two Descent

The functions here have similar syntax to the corresponding ones for curves over \mathbf{Q} . The current implementation from 2011 is the first complete implementation of 2-descent over number fields. Previously there have been several implementations that are effective for computing rank bounds, but not for obtaining nice models of the 2-coverings which is necessary in order to search for points. Some of the key ingredients here are the routine for solving conics over number fields (see `HasRationalPoint`), and techniques for minimisation and reduction over number fields developed by Donnelly and Fisher.

<code>TwoDescent(E:parameters)</code>

<code>RemoveTorsion</code>	BOOLELT	<i>Default : false</i>
<code>RemoveGens</code>	{PTELL}	<i>Default : {}</i>
<code>WithMaps</code>	BOOLELT	<i>Default : true</i>
<code>MinRed</code>	BOOLELT	<i>Default :</i>
<code>Verbose</code>	<code>TwoDescent</code>	<i>Maximum : 1</i>

This function has the same arguments and return values as the corresponding function for curves over \mathbf{Q} . It returns the 2-coverings as a sequence of hyperelliptic curves, and a corresponding list of maps.

Additionally, it returns a third object which specifies the group structure on the 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` are specified.

Currently, there is an additional optional argument `MinRed`, which controls whether the coverings are minimised and reduced. (This may be expensive for various reasons, especially when the field discriminant is not small: for one thing, a large integer may need to be factored.)

<code>TwoCover(e)</code>

<code>TwoCover(e)</code>

The purpose of this function is to obtain 2-covers arising in `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` in the next section).

122.3.6 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 $\phi : E' \rightarrow E$ 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 ,

$$0 \rightarrow E'[\phi] \rightarrow E' \rightarrow E \rightarrow 0,$$

we obtain

$$E'(K) \rightarrow E(K) \rightarrow H^1(K, E'[\phi]) \rightarrow H^1(K, E').$$

For the middle map, we write $\mu : E(K) \rightarrow H^1(K, E'[\phi])$. 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 through all places of K . It fits in the exact sequence

$$0 \rightarrow S^{(\phi)}(E/K) \rightarrow H^1(K, E'[\phi]) \rightarrow \prod_p H^1(K_p, E')$$

The main application of Selmer groups is that they provide the bound:

$$\#E(K)/\phi E'(K) \leq \#S^{(\phi)}(E/K).$$

If $\phi^* : E \rightarrow E'$ is the isogeny dual to ϕ , then $\phi \circ \phi^* : E \rightarrow E$ is multiplication by $d = \text{Deg}(\phi)$. 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 $E(K) \rightarrow K^*$. 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

$$H^1(K, E[2]) \sim \{\delta \in A^*/A^{*2} \mid N_{A/K}(\delta) \in K^{*2}\}.$$

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 $\tau_\delta : T_\delta \rightarrow E$ over K , modulo isomorphy over K , that are isomorphic to $\phi : E' \rightarrow E$ 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)
```

```
CasselsMap(phi)
```

Fields

SETENUM

Default : {}

Given an isogeny $\phi : E \rightarrow E_1$ of elliptic curves over a number field K (or \mathbf{Q}), the function returns the connecting homomorphism

$$\mu : E_1(K) \rightarrow H^1(K, E[\phi]),$$

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 $T_\delta \rightarrow \mathbf{P}^1$. (This covering is a twist of the covering $E \rightarrow E \rightarrow \mathbf{P}^1$ given by $P \mapsto 2P \mapsto x(2P)$.)

SelmerGroup(phi)

Hints	SETENUM	<i>Default</i> : {}
Raw	BOOLELT	<i>Default</i> : false
Bound	RNGINTELT	<i>Default</i> : -1
Verbose	Selmer	<i>Maximum</i> : 2

Given an isogeny $\phi : E \rightarrow E_1$ 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 $AtoS : A \rightarrow S$, 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” $S \rightarrow A$ provides the mathematically meaningful injection $S \hookrightarrow A^*/(A^*)^2$. The standard map

$$E_1(K) \rightarrow E_1(K)/\phi E(K) \rightarrow Sel(\phi)$$

is given by the composition of μ with $AtoS$, where $\mu : E_1(K) \rightarrow H^1(K, E[\phi])$ 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).

<code>TwoSelmerGroup(E)</code>

<code>Hints</code>	<code>SETENUM</code>	<i>Default : {}</i>
<code>Raw</code>	<code>BOOLELT</code>	<i>Default : false</i>
<code>Bound</code>	<code>RNGINTELT</code>	<i>Default : -1</i>
<code>Verbose</code>	<code>EllSelmer</code>	<i>Maximum : 2</i>

The 2-Selmer group of an elliptic curve defined over \mathbf{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 H122E35

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 $E(\mathbf{Q})$, 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  $y^2 = 9x^4 - 4x^3 - 18x^2 + 4x + 13$ 
```

The next command finds all \mathbf{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 H122E36

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 $E(\mathbf{Q})$:

```
> 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 H122E37

The next example is a classic one from [Kra81].

```
> E := EllipticCurve([0, 977, 0, 976, 0]);
> twoE := MultiplicationByMMap(E, 2);
> muE := CasselsMap(twoE);
> SE, toSE := SelmerGroup(twoE);
> rkEQ := RankBound(E);
> ptE := [ E | [-4, 108], [4, 140] ]; // ignore torsion
> IsLinearlyIndependent (ptE);
true
> // So E is really of rank 2
> d := 109;
> Ed := QuadraticTwist(E, d);
> Ed![-976, -298656, 1];
(-976 : -298656 : 1)
> // So Ed is of rank at least 1
> rkEdQ := RankBound(Ed);
> _<x> := PolynomialRing(Rationals());
> K := NumberField(x^2 - d);
> EK := BaseChange(E, K);
> rkEK := RankBound(EK);
> rkEQ;
2
> rkEdQ;
3
> rkEK;
3
```

We know that the rank of $E_d(\mathbf{Q})$ must be the rank of $E(K)$ minus the rank of $E(\mathbf{Q})$; thus $E_d(\mathbf{Q})$ has rank 1. This is smaller than the bound of 3 we get from a 2-descent on E_d alone.

Example H122E38

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  $y^2 = 3x^4 - 10x^3 + 10x + 1$  over
Rational Field
> G, m := TwoSelmerGroup(E);
> G.1 @@ m;
theta^2 - 12*theta + 33
> Parent($1); // Modulus has  $y^2 = \text{modulus}$  isomorphic to E
Univariate Quotient Polynomial Algebra in theta over Rational Field
with modulus  $\text{theta}^3 - 112\text{theta} + 400$ 
> TwoCover( (G.1 + G.2) @@ m);
Hyperelliptic Curve defined by  $y^2 = 3x^4 - 10x^3 + 10x + 1$  over
Rational Field
> TwoCover( Domain(m) ! 1 );
Hyperelliptic Curve defined by  $y^2 = 2x^3 - 12x^2 - 32x + 196$  over
Rational Field

```

122.3.7 The Cassels-Tate Pairing

The pairing between elements of the 2-Selmer group is implemented, in the same way as for curves over the rationals. This is described in Section 122.2.8.

122.3.8 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 principally by Nils Bruin (see [Bru03] or [Bru04]). 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)		
-----------------------	--	--

InertiaDegreeBound	RNGINTELT	Default : 20
SmoothBound	RNGINTELT	Default : 50
PrimeBound	RNGINTELT	Default : 30
IndexBound	RNGINTELT	Default : -1
InitialPrimes	RNGINTELT	Default : 50
Verbose	EllChab	Maximum : 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 \mathbf{Q} -rational image under a given map $E \rightarrow \mathbf{P}^1$.

The arguments are as follows:

- a map $\text{MWmap}: A \rightarrow E$ from an abstract abelian group into $E(K)$ (for instance, the map returned by `PseudoMordellWeilGroup`),
- A map $\text{Ecov}: E \rightarrow \mathbf{P}^1$ 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 \mathbf{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 \mathbf{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 at which the group order of the rational points on the reduction of E are **SmoothBound**-smooth are used.

The Mordell-Weil sieving process only uses information from $E(K)/BE(K)$, where B is **PrimeBound**-smooth.

If **IndexBound** is set to a value different from -1 , then the returned value R will only contain prime divisors from **IndexBound**. Setting this parameter may make it impossible for the routine to return.

Chabauty(MWmap , Ecov , p)

Cosets	TUP	<i>Default :</i>
Aux	SETENUM	<i>Default :</i>
Precision	RNGINTELT	<i>Default :</i>
Bound	RNGINTELT	<i>Default :</i>
Verbose	EllChab	<i>Maximum : 3</i>

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 $E \rightarrow \mathbf{P}^1$ are \mathbf{Q} -rational.

The arguments are as follows:

- a map **MWmap** : $A \rightarrow E$ from an abstract group into $E(K)$ (for instance, the map returned by **PseudoMordellWeilGroup**),
- a map of varieties **Ecov** : $E \rightarrow \mathbf{P}^1$ 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) $\in \mathbf{P}^1(\mathbf{Q})$ (note that N can be ∞).
- V is a set of elements of A found by the program that have images in $\mathbf{P}^1(\mathbf{Q})$,
- 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) $\in \mathbf{P}^1(\mathbf{Q})$, and
- L is a collection of cosets in A such that $(\bigcup L) \cup V$ contains all elements of A that have images in $\mathbf{P}^1(\mathbf{Q})$.

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].

For examples and a more elaborate discussion, see [Bru04].

Example H122E39

This example is motivated and explained in great detail in [Bru04] (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 $K := \mathbf{Q}(\zeta)$ 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

$$u : E \rightarrow \mathbf{P}^1 : (X : Y : Z) \rightarrow d(-X + (\zeta^3 - 1)Z : X + d(-\zeta^3 - \zeta)Z)$$

takes values in $\mathbf{P}^1(\mathbf{Q})$.

```
> 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 \mathbf{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 \mathbf{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 .

122.3.9 Auxiliary Functions for Etale Algebras

This section contains machinery for number fields (and more generally “etale algebras”, i.e. algebras of the form $\mathbf{Q}[x]/p(x)$), 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)

Fields	SETENUM	<i>Default</i> : {}
Verbose	EtaleAlg	<i>Maximum</i> : 1

Given a separable commutative algebra over \mathbf{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)

Verbose	EtaleAlg	<i>Maximum</i> : 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** on page 3-1004. If an algebra over the rationals is required, create a degree 1 extension by

```
RationalsAsNumberField();
```

LocalTwoSelmerMap(P)

Let K be the number field associated with the prime ideal P . The map returned is $K^* \rightarrow K_P^*/K_P^{*2}$, where K_P is the completion of K at P . The codomain is represented as a finite abelian group.

LocalTwoSelmerMap(A, P)

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 $A^* \rightarrow A^*/A^{*2} \otimes K_P$, 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 H122E40

```
> 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>
```

122.3.10 Analytic Information

RootNumber(E, P)

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)

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)

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)

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)

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 $\text{III}(E/K)$. This function computes the product of the regulator and III 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 $n^2 \cdot |\text{III}|$.

122.3.11 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 particularly effective when some traces of Frobenius of the desired are specified. The obvious application here is to find an elliptic curve that matches a known modular form.

These functions can be used to find elliptic curves over \mathbf{Q} , as well as number fields, by using `RationalsAsNumberField()`.

EllipticCurveSearch(N, Effort)**EllipticCurveWithGoodReductionSearch(S, Effort)**

Full	<code>BOOLELT</code>	<i>Default</i> : <code>false</code>
Max	<code>RNGINTELT</code>	<i>Default</i> :
Primes	<code>SEQENUM</code>	<i>Default</i> :
Traces	<code>SEQENUM</code>	<i>Default</i> :
Verbose	<code>ECSearch</code>	<i>Maximum</i> : 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 return a sequence containing all such curves that were found. This sequence will not contain curves that are isomorphic to each other. It may contain isogenous curves, however no attempt is made to find full isogeny classes.

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 which tries all available techniques (however this is subject to change).

There are two ways to specify an early stopping condition:

- (i) If the optional argument **Max** is set to some positive integer, the routine will return whenever this many non-isogenous curves have been found.
- (ii) The optional arguments **Primes** and **Traces**, which must correspond to each other, may be used 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 $[a_P : P \text{ in } \text{Primes}]$. Alternatively, to search for several curves with different traces, **Traces** may instead be a sequence of sequences of that form.

If an early stopping condition has been specified, the routine initially tries with lower effort, in order to succeed quickly for easier input; the effort is incremented exponentially up to the specified **Effort** level. (This strategy is desirable because the algorithm involves performing a large number of independent searches in alternation.) However if the option **Full** is set, the full specified **Effort** is used from the beginning.

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.

122.4 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.

122.4.1 Local Invariants

Conductor(E)

The conductor of the elliptic curve E defined over a p -adic field.

LocalInformation(E)

Implements Tate's algorithm for the elliptic curve E over a p -adic 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 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)

The local root number of the elliptic curve E (defined over a p -adic field).

122.5 Bibliography

- [BC04] W. Bosma and J. Cannon, editors. *Discovering Mathematics with Magma*. Springer-Verlag, Heidelberg, 2004.
- [Bos00] Wieb Bosma, editor. *ANTS IV*, volume 1838 of *LNCS*. Springer-Verlag, 2000.
- [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. Dissertation, University of Leiden, Leiden, 1999.
- [Bru03] Nils Bruin. Chabauty methods using elliptic curves. *J. reine angew. Math.*, 562:27–49, 2003.
- [Bru04] Nils Bruin. Some ternary Diophantine equations of signature $(n, n, 2)$. In Bosma and Cannon [BC04].
- [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.
- [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. URL:<http://arxiv.org/abs/1107.3516>.
- [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⁺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.
- [CM12] B. Creutz and R.L. Miller. Second isogeny descents and the Birch and Swinnerton-Dyer conjectural formula. *J.Algebra*, 372:673–701, 2012.

- [Coh93] Henri Cohen. *A Course in Computational Algebraic Number Theory*, volume 138 of *Graduate Texts in Mathematics*. Springer, Berlin–Heidelberg–New York, 1993.
- [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.
- [Cre10] Brendan Creutz. *Explicit second p -descents on elliptic curves*. PhD Thesis, Jacobs University Bremen, 2010.
- [Elk00] N. Elkies. Rational Points Near Curves and Small Nonzero $|x^3 - y^2|$ via Lattice Reduction. In Bosma [Bos00], pages 33–63.
- [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 \mathbb{Q} . *J. Eur. Math. Soc.*, 3(Issue 2):169–201, 2001.
- [Fis08] Tom Fisher. Finding rational points on elliptic curves using 6-descent and 12-descent. *J. Algebra*, 320(2):853–884, 2008.
- [FK02] Claus Fieker and David R. Kohel, editors. *ANTS V*, volume 2369 of *LNCS*. Springer-Verlag, 2002.
- [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.
- [MSS96] J. R. Merriman, S. Siksek, and N. P. Smart. Explicit 4-descents on an elliptic curve. *Acta Arith.*, 77(4):385–404, 1996.
- [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.
- [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.
- [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.
- [SW02] W. A. Stein and M. Watkins. A New Database of Elliptic Curves—First Report. In Fieker and Kohel [FK02].
- [Wat02] M. Watkins. Computing the modular degree of an elliptic curve. *Experimental Mathematics*, 11(4):487–502, 2002.

- [Wom03] T. Womack. *Explicit descent on elliptic curves*. PhD thesis, University of Nottingham, 2003.

123 ELLIPTIC CURVES OVER FUNCTION FIELDS

<p>123.1 An Overview of Relevant Theory 4077</p> <p>123.2 Local Computations 4079</p> <p>BadPlaces(E) 4079</p> <p>Conductor(E) 4079</p> <p>LocalInformation(E, P1) 4079</p> <p>LocalInformation(E, f) 4079</p> <p>LocalInformation(E) 4079</p> <p>KodairaSymbols(E) 4079</p> <p>NumberOfComponents(K) 4080</p> <p>MinimalModel(E) 4080</p> <p>MinimalDegreeModel(E) 4080</p> <p>IsConstantCurve(E) 4080</p> <p>TraceOfFrobenius(E, p) 4080</p> <p>123.3 Elliptic Curves of Given Conductor 4080</p> <p>EllipticCurveSearch(N, Effort) 4080</p> <p>123.4 Heights 4081</p> <p>NaiveHeight(P) 4081</p> <p>Height(P) 4081</p> <p>LocalHeight(P, P1) 4081</p> <p>HeightPairing(P, Q) 4081</p> <p>HeightPairingMatrix(S) 4081</p> <p>HeightPairingLattice(S) 4081</p> <p>Basis(S) 4081</p> <p>Basis(S, r, disc) 4081</p> <p>IsLinearlyDependent(points) 4081</p> <p>IsLinearlyIndependent(points) 4081</p> <p>IndependentGenerators(points) 4081</p> <p>123.5 The Torsion Subgroup 4082</p> <p>TorsionSubgroup(E) 4082</p> <p>TorsionBound(E, n) 4082</p> <p>TorsionBound(E, n, B) 4082</p> <p>GeometricTorsionBound(E) 4082</p> <p>123.6 The Mordell–Weil Group 4082</p> <p>RankBounds(E) 4082</p>	<p>RankBound(E) 4082</p> <p>MordellWeilGroup(E : -) 4083</p> <p>MordellWeilLattice(E) 4083</p> <p>GeometricMordellWeilLattice(E) 4083</p> <p>Generators(E) 4083</p> <p>123.7 Two Descent 4084</p> <p>TwoSelmerGroup(E) 4084</p> <p>TwoDescent(E) 4084</p> <p>QuarticMinimize(f) 4084</p> <p>Points(C : -) 4085</p> <p>PointsQI(C, H) 4085</p> <p>TwoIsogenySelmerGroups(E) 4085</p> <p>123.8 The <i>L</i>-function and Counting Points 4085</p> <p>LFunction(E) 4085</p> <p>LFunction(E, S) 4085</p> <p>LFunction(E, e) 4086</p> <p>AnalyticRank(E) 4086</p> <p>AnalyticInformation(E) 4086</p> <p>AnalyticInformation(E, L) 4086</p> <p>AnalyticInformation(E, e) 4086</p> <p>NumberOfPointsOnSurface(E, e) 4087</p> <p>NumbersOfPointsOnSurface(E, e) 4087</p> <p>BettiNumber(E, i) 4087</p> <p>CharacteristicPolynomial FromTraces(traces) 4087</p> <p>CharacteristicPolynomial FromTraces(traces, d, q, i) 4087</p> <p>123.9 Action of Frobenius 4088</p> <p>Frobenius(P, q) 4088</p> <p>FrobeniusActionOnPoints(S, q : -) 4088</p> <p>FrobeniusActionOnReducibleFiber(L) 4088</p> <p>FrobeniusActionOnTrivialLattice(E) 4088</p> <p>123.10 Extended Examples 4088</p> <p>123.11 Bibliography 4091</p>
---	---

Chapter 123

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.

123.1 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 $\pi : \mathcal{E} \rightarrow C$ (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 $s : C \rightarrow \mathcal{E}$ such that $\pi \circ s = \text{Id}_C$.) 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 $\text{NS}(\mathcal{E}_{\bar{k}})$ 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 $\text{NS}(\mathcal{E}_{\bar{k}})$ 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 $\text{NS}(\mathcal{E}_{\bar{k}})$. 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 $\text{NS}(\mathcal{E}_{\bar{k}})/N$ is generated by images of sections of π , and that $\text{NS}(\mathcal{E}_{\bar{k}})/N \cong E(\bar{k}(C))$ (via the identification of sections with points). In particular, this implies the Shioda–Tate formula

$$\text{rank}(E(\bar{k}(C))) + 2 + \sum_{v \in C(\bar{k})} (m_v - 1) = \text{rank}(\text{NS}(\mathcal{E}_{\bar{k}}))$$

where m_v denotes the number of components of the fibre $\pi^{-1}(v)$.

The Galois group $G = G_{\bar{k}/k}$ acts on $\text{NS}(\mathcal{E}_{\bar{k}})$, and it maps N to itself. Moreover, after extending scalars to \mathbf{Q} one can split the Galois representation. That is, there exists $M \subset \text{NS}(\mathcal{E}_{\bar{k}}) \otimes \mathbf{Q}$ such that $\text{NS}(\mathcal{E}_{\bar{k}}) \otimes \mathbf{Q} \cong M \oplus (N \otimes \mathbf{Q})$ and $M \cong E(\bar{k}(C)) \otimes \mathbf{Q}$ as G -modules. In particular, $M^G \cong E(K) \otimes \mathbf{Q}$. 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 $\text{NS}(\mathcal{E}_{\bar{k}})$ as a G -module, one can embed it in the ℓ -adic cohomology group $H^2(\mathcal{E}_{\bar{k}}, \mathbf{Q}_{\ell})$. To get a G -equivariant map one must slightly change the G -action on $H^2(\mathcal{E}_{\bar{k}}, \mathbf{Q}_{\ell})$. Let $H^2(\mathcal{E}_{\bar{k}}, \mathbf{Q}_{\ell})(1)$ denote the (1)-Tate twist of $H^2(\mathcal{E}_{\bar{k}}, \mathbf{Q}_{\ell})$. The main property that we need to know about this twist is that it transforms the q -eigensubspace in $H^2(\mathcal{E}_{\bar{k}}, \mathbf{Q}_{\ell})$ of some q -Frobenius element to the 1-eigenspace of this Frobenius in $H^2(\mathcal{E}_{\bar{k}}, \mathbf{Q}_{\ell})(1)$. The cycle class map then yields a G -equivariant embedding

$$\text{NS}(\mathcal{E}_{\bar{k}}) \otimes \mathbf{Q}_{\ell} \hookrightarrow H^2(\mathcal{E}_{\bar{k}}, \mathbf{Q}_{\ell})(1).$$

It is conjectured by Tate that the image of $(\text{NS}(\mathcal{E}_{\bar{k}}) \otimes \mathbf{Q}_{\ell})^G$ under this map exactly equals $H^2(\mathcal{E}_{\bar{k}}, \mathbf{Q}_{\ell})(1)^G$.

In the case that k is a finite field one can in principle determine $H^2(\mathcal{E}_{\bar{k}}, \mathbf{Q}_{\ell})(1)^G$ via the Lefschetz trace formula. Suppose that $k \subset \mathbf{F}_q$ and let F_q denote the q -th power Frobenius map. Then

$$\#\mathcal{E}(\mathbf{F}_q) = 1 + q^2 - (1 + q) \text{Trace}(F_q | H^1(C_{\bar{k}}, \mathbf{Q}_{\ell})) + \text{Trace}(F_q | H^2(\mathcal{E}_{\bar{k}}, \mathbf{Q}_{\ell})).$$

The trace on $H^1(C_{\bar{k}}, \mathbf{Q}_{\ell})$ is zero if C is a rational curve, and it can be determined by counting \mathbf{F}_q -rational points on C in the general case. Hence one can determine $\text{Trace}(F_q | H^2(\mathcal{E}_{\bar{k}}, \mathbf{Q}_{\ell}))$ by counting \mathbf{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 $H^2(\mathcal{E}_{\bar{k}}, \mathbf{Q}_{\ell})$, 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

$$\det(1 - T \cdot F_q | H^2(\mathcal{E}_{\bar{k}}, \mathbf{Q}_{\ell}) / \text{im}(N \otimes \mathbf{Q}_{\ell})),$$

where $\text{im}(N \otimes \mathbf{Q}_{\ell})$ stands for the image of $N \otimes \mathbf{Q}_{\ell}$ 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 $\text{rank}(\text{NS}(\mathcal{E}_{\bar{k}})) = 10$. In this case $\text{rank}(E(\bar{k}(C))) = 10 - \text{rank}(N)$ 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.

123.2 Local Computations

`BadPlaces(E)`

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)`

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)`

`LocalInformation(E, f)`

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)`

Returns a sequence of tuples as described above for all places of bad reduction of the elliptic curve E .

`KodairaSymbols(E)`

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)

The number of components of a fibre with the Kodaira symbol K .

MinimalModel(E)

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)

A model of the elliptic curve E (defined over a rational function field) which minimises the quantity $\text{Max}([\text{Degree}(a_i)/i])$, where a_1, a_2, a_3, a_4, a_6 are the Weierstrass coefficients.

IsConstantCurve(E)

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)

The trace of Frobenius a_p for the reduction of E at the place p , specified as an element of the base field.

123.3 Elliptic Curves of Given Conductor

EllipticCurveSearch(N, Effort)

Full	BOOLELT	<i>Default : false</i>
Max	RNGINTELT	<i>Default :</i>
Primes	SEQENUM	<i>Default :</i>
Traces	SEQENUM	<i>Default :</i>
Verbose	ECSearch	<i>Maximum : 2</i>

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.

123.4 Heights

NaiveHeight(P)

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)

The Néron–Tate height $\hat{h}(P)$ of the given point P on an elliptic curve defined over a function field.

LocalHeight(P, Pl)

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)

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)

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)

The height pairing lattice of a sequence of independent points on an elliptic curve defined over a function field.

Basis(S)

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)

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)

IsLinearlyIndependent(points)

IndependentGenerators(points)

These functions are available for elliptic curves over function fields, and behave the same way as for elliptic curves over the rationals.

123.5 The Torsion Subgroup

`TorsionSubgroup(E)`

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)`

`TorsionBound(E, n, B)`

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)`

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.

123.6 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)`

`RankBound(E)`

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)

A1

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 **A1** 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)

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)

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)

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 H123E1

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
> 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

```

123.7 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 $E \rightarrow E^{frob} \rightarrow E$. 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)

This function computes the 2-Selmer group of an elliptic curve E defined over a rational function field $\mathbf{F}_q(t)$ 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)

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 $C \rightarrow E$. The elliptic curve should be defined over a rational function field $\mathbf{F}_q(t)$ 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)

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)**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 $\mathbf{F}_q(t)$. 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)**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 $\mathbf{F}_q(t)$. 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)

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 $E \rightarrow E^{frob} : (x, y) \mapsto (x^2, y^2)$ and the dual isogeny $E^{frob} \rightarrow E$ (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 $S^{sep} \rightarrow k(t)/\Phi(k(t))$ and $S^{Frob} \rightarrow k(t)^*/(k(t)^*)^2$. 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].

123.8 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 H123E6).

LFunction(E)**LFunction(E, S)****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)

The L -function for the base change of the elliptic curve E by a constant field extension of degree e .

AnalyticRank(E)

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)

AnalyticInformation(E, L)

AnalyticInformation(E, e)

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 H123E2

The curve $y^2 = x^3 + 2t^2x + (t^3 + 2t^2 + 2t)$ over $\mathbf{F}_3(t)$ has rank 0 and its Tate–Shafarevich group contains $\mathbf{Z}/3 \oplus \mathbf{Z}/3$. 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 $E(\overline{\mathbf{F}_3}(t))$ is 4.

Example H123E3

We consider again the curve $y^2 = x^3 + (t^4 + 2t^2)x + t^3 + 2t$ over $\mathbf{F}_3(t)$, which was found to have rank 2 and regulator 12 in Example H123E1.

```
> 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)

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)

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 \mathbf{F}_q of degree up to e , together with the number of points on the base curve over these extensions.

BettiNumber(E, i)

Returns the i -th Betti number of the Kodaira–Néron model of the elliptic surface corresponding to the elliptic curve E .

CharacteristicPolynomialFromTraces(traces)

Given a sequence $traces = [t_1, t_2, \dots, t_d]$, 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)

Given a sequence of field elements $[t_1, t_2, \dots, t_k]$ 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 $\text{Floor}(d/2)$. 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 $\text{Ceiling}(d/2)$ 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.

123.9 Action of Frobenius

`Frobenius(P, q)`

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 \mathbf{F}_q .

`FrobeniusActionOnPoints(S, q : parameters)`

`gram`

ALGMAT`ELT`

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)`

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)`

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.

123.10 Extended Examples

Example H123E4

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 \mathbf{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 \mathbf{F}_{1831} , in this case up to \mathbf{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 H123E5

In this example we determine the $\mathbf{C}(t)$ -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));
[ <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 $\mathbf{F}_{11^2}(t)$ the rank equals the geometric rank of 1, and the height of a generator is congruent to $\frac{35}{2}$ modulo \mathbf{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 $\mathbf{F}_{13^2}(t)$ the rank equals the geometric rank of 1. The height of a generator is congruent to $\frac{121}{2}$ modulo \mathbf{Q}^2 . As the quotient of the heights of generators in characteristics 11 and 13 is not a square in \mathbf{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 H123E6

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 $\mathbf{F}_{5^{10}}$. 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] ];
> CharacteristicPolynomialFromTraces(traces);
5750*T^5 + 875*T^4 - 40*T^3 - 40*T^2 + 1
```

123.11 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.

124 MODELS OF GENUS ONE CURVES

124.1 Introduction	4095	<code>Reduce(f)</code>	4102
124.2 Related Functionality . . .	4096	<code>ReduceQuadrics(seq)</code>	4102
124.3 Creation of Genus One Models	4096	124.7 Genus One Models as Coverings	4102
<code>GenusOneModel(n, seq)</code>	4096	<code>Jacobian(model)</code>	4103
<code>GenusOneModel(seq)</code>	4096	<code>Jacobian(C)</code>	4103
<code>GenusOneModel(n, str)</code>	4096	<code>nCovering(model : -)</code>	4103
<code>GenusOneModel(C)</code>	4096	<code>AddCubics(cubic1, cubic2 : -)</code>	4103
<code>GenusOneModel(f)</code>	4096	<code>AddCubics(model1, model2 : -)</code>	4103
<code>GenusOneModel(seq)</code>	4096	<code>+</code>	4103
<code>GenusOneModel(n, E)</code>	4096	<code>DoubleGenusOneModel(model)</code>	4103
<code>GenusOneModel(mat)</code>	4097	<code>FourToTwoCovering(model : -)</code>	4103
<code>GenusOneModel(mats)</code>	4097	<code>FourToTwoCovering(C : -)</code>	4103
<code>CompleteTheSquare(model)</code>	4097	124.8 Families of Elliptic Curves with Prescribed n-Torsion .	4104
<code>RandomGenusOneModel(n)</code>	4097	<code>RubinSilverbergPolynomials(n, J : -)</code>	4104
<code>RandomModel(n)</code>	4097	124.9 Transformations between Genus One Models	4104
<code>GenericModel(n)</code>	4097	<code>IsTransformation(n, g)</code>	4104
<code>ChangeRing(model, B)</code>	4097	<code>RandomTransformation(n : -)</code>	4105
<code>CubicFromPoint(E, P)</code>	4097	<code>*</code>	4105
<code>HesseModel(n, seq)</code>	4097	<code>ApplyTransformation(g, model)</code>	4105
<code>DiagonalModel(n, seq)</code>	4097	<code>*</code>	4105
124.4 Predicates on Genus One Models	4099	<code>ComposeTransformations(g1, g2)</code>	4105
<code>IsGenusOneModel(f)</code>	4099	<code>ScalingFactor(g)</code>	4105
<code>IsGenusOneModel(seq)</code>	4099	124.10 Invariants for Genus One Models	4105
<code>IsGenusOneModel(mat)</code>	4099	<code>aInvariants(model)</code>	4105
<code>IsEquivalent(model1, model2)</code>	4099	<code>bInvariants(model)</code>	4105
<code>IsEquivalent(cubic1, cubic2)</code>	4099	<code>cInvariants(model)</code>	4105
124.5 Access Functions	4099	<code>Invariants(model)</code>	4106
<code>Degree(model)</code>	4099	<code>Discriminant(model)</code>	4106
<code>DefiningEquations(model)</code>	4099	<code>SL4Invariants(model)</code>	4106
<code>Equations(model)</code>	4099	124.11 Covariants and Contravariants for Genus One Models . . .	4106
<code>Matrix(model)</code>	4099	<code>Hessian(model)</code>	4106
<code>Curve(model)</code>	4100	<code>CoveringCovariants(model)</code>	4106
<code>HyperellipticCurve(model)</code>	4100	<code>Contravariants(model)</code>	4106
<code>QuadricIntersection(model)</code>	4100	<code>HesseCovariants(model, r)</code>	4106
<code>Matrices(model)</code>	4100	<code>HessePolynomials(n, r, invariants : -)</code>	4107
<code>BaseRing(model)</code>	4100	124.12 Examples	4107
<code>PolynomialRing(model)</code>	4100	124.13 Bibliography	4109
<code>Eltseq(model)</code>	4100		
<code>ModelToString(model)</code>	4100		
124.6 Minimisation and Reduction	4100		
<code>Minimise(model : -)</code>	4101		
<code>Minimise(f)</code>	4101		
<code>pMinimise(f, p)</code>	4101		
<code>Reduce(model)</code>	4102		

Chapter 124

MODELS OF GENUS ONE CURVES

124.1 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 \mathbf{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 \mathbf{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 $\mathbf{P}_{x,z}^1$, 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 \mathbf{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 \mathbf{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).

124.2 Related Functionality

Section 122.2.11, which concerns three descent on elliptic curves, is relevant for studying rational points on plane cubics (genus one models of degree 3) over the rationals.

A very efficient point search for schemes is available; for details see Section 112.13.4. This will find all rational points up to absolute height H on the curve associated to a given genus one model in time $O(H^{2/d})$, where d is the dimension of the ambient projective space.

124.3 Creation of Genus One Models

`GenusOneModel(n, seq)`

`GenusOneModel(seq)`

`GenusOneModel(n, str)`

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)`

A genus one model that represents the given curve C .

For degree 2, C should either be a subscheme of a weighted projective space $\mathbf{P}(1, 1, 2)$, 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 \mathbf{P}^3 for $n = 4$, or an intersection of five quadrics in \mathbf{P}^4 for $n = 5$.

`GenusOneModel(f)`

`GenusOneModel(seq)`

The genus one model given by the polynomial f or the sequence of equations `seq`.

`GenusOneModel(n, E)`

A genus one model of degree n (where n is 2, 3, 4, or 5) representing the elliptic curve E embedded in \mathbf{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 $E \rightarrow C$ and $C \rightarrow E$.

`GenusOneModel(mat)`

The genus one model of degree 5 associated to the given 5×5 matrix.

`GenusOneModel(mats)`

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`).

`CompleteTheSquare(model)`

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.

`RandomGenusOneModel(n)`

`RandomModel(n)`

Size

RNGINTELT

Default :

A random genus one model of degree n , where n is 2, 3, 4, or 5.

`GenericModel(n)`

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.

`ChangeRing(model, B)`

The genus one model defined over the ring B obtained by coercing the coefficients of the given genus one model into B .

`CubicFromPoint(E, P)`

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)`

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)`

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 H124E1

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 $\mathbf{Q}(a, b)$. The model is the image of $E_{a,b}$ under the embedding in \mathbf{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 \mathbf{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 c_4, c_6, Δ 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
```

124.4 Predicates on Genus One Models

`IsGenusOneModel(f)`

`IsGenusOneModel(seq)`

`IsGenusOneModel(mat)`

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, as degenerate cases are allowed!

`IsEquivalent(model1, model2)`

`IsEquivalent(cubic1, cubic2)`

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 $\mathbf{P}_{\mathbb{Q}}^2$ which takes one cubic to the other, up to scaling. The algorithm is given in [Fis06].

When true, the transformation is also returned as a tuple (the syntax is explained in Section 124.9, on transformations, below).

124.5 Access Functions

`Degree(model)`

The degree (2, 3, 4, or 5) of the given model.

`DefiningEquations(model)`

A sequence containing the equations by which the given genus one model (which must have degree 2, 3, or 4) is defined.

`Equations(model)`

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)`

The defining matrix of a genus one model of degree 5.

`Curve(model)`

`HyperellipticCurve(model)`

`QuadricIntersection(model)`

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 \mathbf{P}^3 .

An error results in degenerate cases where the equations of the model do not define a curve

`Matrices(model)`

For a genus one model of degree 4 this function returns a sequence containing two 4×4 symmetric matrices representing the quadrics.

`BaseRing(model)`

The coefficient ring of the given model.

`PolynomialRing(model)`

The polynomial ring used to define the model.

`Eltseq(model)`

`ModelToString(model)`

A sequence or string containing the coefficients of the defining data of the given genus one model (which is either a polynomial, a pair of polynomials, or a matrix). The model may be recovered by using `GenusOneModel`.

124.6 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 \mathbf{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.

<code>Minimise(model : parameters)</code>		
---	--	--

<code>Transformation</code>	<code>BOOLELT</code>	<i>Default : true</i>
<code>CrossTerms</code>	<code>BOOLELT</code>	<i>Default : false</i>
<code>UsePrimes</code>	<code>SEQENUM</code>	<i>Default : []</i>
<code>ClassGroupPrimes</code>	<code>SEQENUM</code>	<i>Default : []</i>
<code>Verbose</code>	<code>Minimise</code>	<i>Maximum : 3</i>

Given a genus one model (of degree 2, 3, 4, or 5) with coefficients in \mathbf{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 124.9).

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 \mathbf{Q}_p^{nr} , the maximal non-ramified extension of \mathbf{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.

<code>Minimise(f)</code>

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 124.9 below).

<code>pMinimise(f, p)</code>

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)

Reduce(f)

Verbose

Reduce

Maximum : 3

Given a genus one model of degree 2, 3, or 4 over \mathbf{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 124.9).

ReduceQuadrics(seq)

Verbose

Reduce

Maximum : 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, \dots, 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 $[x_1, \dots, x_n].S$ for $[x_1, \dots, x_n]$ 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.

124.7 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 122.2.11). For degree 4 models the maps can also be computed using `AssociatedEllipticCurve` and `AssociatedHyperellipticCurve` from the package on four descent (see Section 122.2.9).

Jacobian(model)

Jacobian(C)

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)

E

CRV _{ELL}

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 $C \rightarrow E$.

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)
--

AddCubics(model1, model2 : parameters)
--

model1 + model2

E

CRV _{ELL}

Default :

ReturnBoth

BOOLE _{LT}

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 $H^1(\mathbf{Q}, E[3])$.

An error results if the two cubics do not belong to the same elliptic curve E , See Section 122.2.11 for more information about `AddCubics`.

DoubleGenusOneModel(model)

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)

FourToTwoCovering(C : parameters)

C ₂

CRV

Default :

Given a genus one model of degree 4 or an associated curve, this function returns three values: the curve C_4 in \mathbf{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 $C_4 \rightarrow C_2$. Calling `AssociatedHyperellipticCurve(Curve(model))` provides the same information.

124.8 Families of Elliptic Curves with Prescribed n -Torsion

In [RS95], Rubin and Silverberg explicitly construct families of elliptic curve over \mathbf{Q} which have the same Galois representation on the n -torsion subgroups as a given elliptic curve.

<code>RubinSilverbergPolynomials(n, J : parameters)</code>
--

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 $\mathbf{Z}[G]$ -modules, where G is the absolute Galois group of \mathbf{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.

124.9 Transformations between Genus One Models

A transformation between two genus one models of degree $n = 2, 3, 4,$ or 5 is a tuple consisting of two elements. The second element is an $n \times n$ matrix determining a linear transformation of the projective coordinates (acting on them from the right). The first element is a “rescaling”, which is either a matrix or a scalar.

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 $TM_S T^{tr}$ 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 one of these transformations. Equivalent models have the same invariants up to scaling by the 4th and 6th powers of some element, given by `ScalingFactor`.

<code>IsTransformation(n, g)</code>

Return `true` if and only if the tuple g represents a transformation between genus one models of degree n .

`RandomTransformation(n : parameters)`

Size	RNGINTELT	<i>Default : 5</i>
Unimodular	BOOLELT	<i>Default : false</i>
CrossTerms	BOOLELT	<i>Default : false</i>

A tuple that represents a random transformation between 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`

`ApplyTransformation(g, model)`

The result of applying the transformation g to the second argument, which should be a genus one model.

`g1 * g2`

`ComposeTransformations(g1, g2)`

The composition $g1 * g2$ of two transformations of genus one models. Transformations of genus one models act on the left, so $(g1 * g2) * f = g1 * (g2 * f)$.

`ScalingFactor(g)`

The scaling factor of a transformation g between genus one models is 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 $\lambda^6 c_6$.

124.10 Invariants for Genus One Models

`aInvariants(model)`

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)`

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)`

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 [Fis08].

Invariants(model)

The invariants c_4, c_6 and Δ (the discriminant) of the given genus one model.

Discriminant(model)

The discriminant Δ of the given genus one model.

SL4Invariants(model)

The SL_4 -invariants of a genus one model of degree 4.

124.11 Covariants and Contravariants for Genus One Models

The functions in this section implement the invariant theory developed in [Fis].

Hessian(model)

We write X_n for the (affine) space of genus one models of degree n . The module of covariants $X_n \rightarrow X_n$ 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)

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)

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 $X_n \rightarrow X_n^*$ 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)

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(<i>n</i> , <i>r</i> , invariants : <i>parameters</i>)
--

Variables	[RINGMPOLELT]	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 $r \equiv 1 \pmod{n}$.

124.12 Examples

Example H124E2

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 [CM00]).

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
> 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 $F(\mathbf{Q})$ since $F(\mathbf{Q})$ has rank 1:

```
> SetClassGroupBounds(500);
> #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 $\overline{\mathbf{Q}}$). 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 + 0(43) : 1 + 0(43^50) : 0(43))
> IsLocallySolvable(C2_crv, 101);
true (1 + 0(101^50) : 32 + 0(101) : 1 + 0(101))
// 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 $E(\mathbf{Q})$ is trivial, any rational points on $C2$ must map to O_E .

124.13 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.
- [CM00] John E. Cremona and Barry Mazur. Visualizing elements in the Shafarevich-Tate group. *Experiment. Math.*, 9(1):13–28, 2000.
- [Fis] Tom Fisher. The Hessian of a genus one curve.
- [Fis06] T.A. Fisher. Testing equivalence of ternary cubics. In S. Pauli F. Hess and M. Pohst, editors, *ANTS VII*, volume 4076 of *LNCS*, pages 333–345. Springer-Verlag, 2006.
- [Fis08] T.A Fisher. The invariants of a genus one curve. *Proc. Lond. Math. Soc.*, 97(3):753–782, 2008.
- [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.

125 HYPERELLIPTIC CURVES

125.1 Introduction	4115	<code>IsEllipticCurve(C)</code>	4123
125.2 Creation Functions	4115	125.3 Operations on Curves	4123
125.2.1 <i>Creation of a Hyperelliptic Curve</i> 4115		125.3.1 <i>Elementary Invariants</i>	4124
<code>HyperellipticCurve(f, h)</code>	4115	<code>HyperellipticPolynomials(C)</code>	4124
<code>HyperellipticCurve(f)</code>	4115	<code>Degree(C)</code>	4124
<code>HyperellipticCurve([f, h])</code>	4115	<code>Discriminant(C)</code>	4124
<code>HyperellipticCurve(P, f, h)</code>	4116	<code>Genus(C)</code>	4124
<code>HyperellipticCurveOfGenus(g, f, h)</code>	4116	125.3.2 <i>Igusa Invariants</i>	4124
<code>HyperellipticCurveOfGenus(g, f)</code>	4116	<code>ClebschInvariants(C)</code>	4125
<code>HyperellipticCurve</code>		<code>ClebschInvariants(f)</code>	4126
<code>OfGenus(g, [f, h])</code>	4116	<code>IgusaClebschInvariants(C: -)</code>	4126
<code>HyperellipticCurve(E)</code>	4116	<code>IgusaClebschInvariants(f, h)</code>	4126
125.2.2 <i>Creation Predicates</i>	4116	<code>IgusaClebschInvariants(f: -)</code>	4126
<code>IsHyperellipticCurve([f, h])</code>	4116	<code>IgusaInvariants(C: -)</code>	4126
<code>IsHyperellipticCurve</code>		<code>JInvariants(C: -)</code>	4126
<code>OfGenus(g, [f, h])</code>	4116	<code>IgusaInvariants(f, h)</code>	4127
125.2.3 <i>Changing the Base Ring</i>	4117	<code>JInvariants(f, h)</code>	4127
<code>BaseChange(C, K)</code>	4117	<code>IgusaInvariants(f: -)</code>	4127
<code>BaseExtend(C, K)</code>	4117	<code>JInvariants(f: -)</code>	4127
<code>BaseChange(C, j)</code>	4117	<code>ScaledIgusaInvariants(f, h)</code>	4127
<code>BaseExtend(C, j)</code>	4117	<code>ScaledIgusaInvariants(f)</code>	4127
<code>BaseChange(C, n)</code>	4117	<code>AbsoluteInvariants(C)</code>	4127
<code>BaseExtend(C, n)</code>	4117	<code>ClebschToIgusaClebsch(Q)</code>	4127
<code>ChangeRing(C, K)</code>	4117	<code>IgusaClebschToIgusa(S)</code>	4127
125.2.4 <i>Models</i>	4118	<code>G2Invariants(C)</code>	4127
<code>SimplifiedModel(C)</code>	4118	<code>G2ToIgusaInvariants(GI)</code>	4128
<code>HasOddDegreeModel(C)</code>	4118	<code>IgusaToG2Invariants(JI)</code>	4128
<code>IntegralModel(C)</code>	4119	125.3.3 <i>Shioda Invariants</i>	4128
<code>MinimalWeierstrassModel(C)</code>	4119	<code>ShiodaInvariants(C)</code>	4128
<code>pIntegralModel(C, p)</code>	4119	<code>ShiodaInvariantsEqual(V1, V2)</code>	4128
<code>pNormalModel(C, p)</code>	4119	<code>DiscriminantFromShiodaInvariants(JI)</code>	4129
<code>pMinimalWeierstrassModel(C, p)</code>	4119	<code>ShiodaAlgebraicInvariants(FJI)</code>	4129
<code>ReducedModel(C)</code>	4120	<code>MaedaInvariants(C)</code>	4130
<code>ReducedMinimalWeierstrassModel(C)</code>	4120	125.3.4 <i>Base Ring</i>	4130
<code>SetVerbose("CrvHypReduce", v)</code>	4120	<code>BaseField(C)</code>	4130
125.2.5 <i>Predicates on Models</i>	4120	<code>BaseRing(C)</code>	4130
<code>IsSimplifiedModel(C)</code>	4120	<code>CoefficientRing(C)</code>	4130
<code>IsIntegral(C)</code>	4120	125.4 Creation from Invariants	4130
<code>IspIntegral(C, p)</code>	4120	<code>HyperellipticCurveFrom</code>	
<code>IspNormal(C, p)</code>	4120	<code>IgusaClebsch(S)</code>	4131
<code>IspMinimal(C, p)</code>	4121	<code>HyperellipticCurveFrom</code>	
125.2.6 <i>Twisting Hyperelliptic Curves</i>	4121	<code>G2Invariants(S)</code>	4131
<code>QuadraticTwist(C, d)</code>	4121	<code>HyperellipticCurveFrom</code>	
<code>QuadraticTwist(C)</code>	4121	<code>ShiodaInvariants(JI)</code>	4131
<code>QuadraticTwists(C)</code>	4121	<code>HyperellipticPolynomialFrom</code>	
<code>IsQuadraticTwist(C, D)</code>	4121	<code>ShiodaInvariants(JI)</code>	4131
<code>Twists(C)</code>	4121	125.5 Function Field	4133
<code>HyperellipticPolynomials</code>		125.5.1 <i>Function Field and Polynomial</i>	
<code>FromShiodaInvariants(JI)</code>	4122	<code>Ring</code>	4133
125.2.7 <i>Type Change Predicates</i>	4123	<code>FunctionField(C)</code>	4133

DefiningPolynomial(C)	4133	@	4139
EvaluatePolynomial(C, a, b, c)	4133	Evaluate(f,P)	4139
EvaluatePolynomial(C, [a, b, c])	4133	@@	4139
125.6 Points 4133		Pullback(f,P)	4139
125.6.1 Creation of Points 4133		eq	4139
!	4133	125.7.3 Invariants of Isomorphisms 4140	
!	4133	Parent(f)	4140
elt< >	4133	Domain(f)	4140
elt< >	4133	Codomain(f)	4140
!	4133	125.7.4 Automorphism Group and	
Points(C, x)	4134	Isomorphism Testing 4140	
RationalPoints(C, x)	4134	IsGL2Equivalent(f, g, n)	4140
PointsAtInfinity(C)	4134	IsIsomorphic(C1, C2)	4140
IsPoint(C, S)	4134	AutomorphismGroup(C)	4141
125.6.2 Random Points 4135		GeometricAutomorphismGroup(C)	4142
Random(C)	4135	GeometricAutomorphismGroupFrom	
125.6.3 Predicates on Points 4135		ShiodaInvariants(JI)	4143
eq	4135	GeometricAutomorphismGroup	
ne	4135	Genus2Classification(F)	4144
125.6.4 Access Operations 4135		GeometricAutomorphismGroup	
P[i]	4135	Genus3Classification(F)	4144
Eltseq(P)	4135	125.8 Jacobians 4145	
ElementToSequence(P)	4135	125.8.1 Creation of a Jacobian 4145	
125.6.5 Arithmetic of Points 4135		Jacobian(C)	4145
-	4135	125.8.2 Access Operations 4145	
Involution(P)	4135	Curve(J)	4145
125.6.6 Enumeration and Counting Points 4136		Dimension(J)	4145
NumberOfPointsAtInfinity(C)	4136	125.8.3 Base Ring 4145	
PointsAtInfinity(C)	4136	BaseField(J)	4145
#	4136	BaseRing(J)	4145
Points(C)	4136	CoefficientRing(J)	4145
RationalPoints(C)	4136	125.8.4 Changing the Base Ring 4146	
PointsKnown(C)	4136	BaseChange(J, F)	4146
ZetaFunction(C)	4137	BaseExtend(J, F)	4146
ZetaFunction(C, K)	4137	BaseChange(J, j)	4146
125.6.7 Frobenius 4137		BaseExtend(J, j)	4146
Frobenius(P, F)	4137	BaseChange(J, n)	4146
125.7 Isomorphisms and		BaseExtend(J, n)	4146
Transformations 4138		125.9 Richelot Isogenies 4146	
125.7.1 Creation of Isomorphisms 4138		RichelotIsogenousSurfaces(J)	4147
Aut(C)	4138	RichelotIsogenousSurfaces(C)	4147
Iso(C1, C2)	4138	RichelotIsogenousSurface(J, kernel)	4147
Transformation(C, t)	4138	RichelotIsogenousSurface(C, kernel)	4147
Transformation(C, u)	4138	125.10 Points on the Jacobian 4149	
Transformation(C, e)	4138	125.10.1 Creation of Points 4150	
Transformation(C, e, u)	4138	!	4150
Transformation(C, t, e, u)	4138	Id(J)	4150
125.7.2 Arithmetic with Isomorphisms 4139		Identity(J)	4150
*	4139	!	4150
Inverse(f)	4139	elt< >	4150
in	4139	elt< >	4150
		elt< >	4150
		elt< >	4150
		-	4150

!	4150	JacobianOrdersByDeformation(Q, Y)	4162
elt< >	4150	EulerFactorsByDeformation(Q, Y)	4162
!	4150	ZetaFunctionsByDeformation(Q, Y)	4162
elt< >	4150	125.11.4 Abelian Group Structure	4163
D)	4151	Sylow(J, p)	4163
!	4151	AbelianGroup(J)	4163
Points(J, a, d)	4151	HasAdditionAlgorithm(J)	4163
RationalPoints(J, a, d)	4151	125.12 Jacobians over Number Fields	
125.10.2 Random Points	4153	or Q	4164
Random(J)	4153	125.12.1 Searching For Points	4164
125.10.3 Booleans and Predicates for Points	4153	Points(J)	4164
eq	4153	RationalPoints(J)	4164
ne	4153	125.12.2 Torsion	4164
IsZero(P)	4153	TwoTorsionSubgroup(J)	4164
IsIdentity(P)	4153	TorsionBound(J, n)	4164
125.10.4 Access Operations	4154	TorsionSubgroup(J)	4164
P[i]	4154	125.12.3 Heights and Regulator	4166
Eltseq(P)	4154	NaiveHeight(P)	4167
ElementToSequence(P)	4154	Height(P: -)	4167
125.10.5 Arithmetic of Points	4154	CanonicalHeight(P: -)	4167
-	4154	HeightConstant(J: -)	4167
+	4154	HeightPairing(P, Q: -)	4167
+=	4154	HeightPairingMatrix(S: Precision)	4168
-	4154	Regulator(S: Precision)	4168
-=	4154	ReducedBasis(S: Precision)	4168
*	4154	125.12.4 The 2-Selmer Group	4171
*	4154	BadPrimes(C)	4171
*:=	4154	BadPrimes(J)	4171
125.10.6 Order of Points on the Jacobian .	4155	HasSquareSha(J)	4171
Order(P)	4155	IsEven(J)	4171
Order(P, l, u)	4155	IsDeficient(C, p)	4172
Order(P, l, u, n, m)	4155	HasIndexOne(C, p)	4172
HasOrder(P, n)	4155	HasIndexOne(C, p)	4172
125.10.7 Frobenius	4155	HasIndexOneEverywhereLocally(C)	4172
Frobenius(P, k)	4155	TwoSelmerGroup(J)	4172
125.10.8 Weil Pairing	4156	RankBound(J)	4173
WeilPairing(P, Q, m)	4156	RankBounds(J)	4173
125.11 Rational Points and Group		125.13 Two-Selmer Set of a Curve .	4179
Structure over Finite Fields .	4157	TwoCoverDescent(C)	4179
125.11.1 Enumeration of Points	4157	125.14 Chabauty's Method	4182
Points(J)	4157	Chabauty0(J)	4183
RationalPoints(J)	4157	Chabauty(P : ptC)	4183
125.11.2 Counting Points on the Jacobian	4157	Chabauty(P, p: Precision)	4183
SetVerbose("JacHypCnt", v)	4157	125.15 Cyclic Covers of P¹	4187
#	4157	125.15.1 Points	4187
Order(J)	4157	RationalPoints(f, q)	4187
FactoredOrder(J)	4161	HasPoint(f, q, v)	4187
EulerFactor(J)	4161	HasPoint(f, q, v)	4187
EulerFactorModChar(J)	4161	HasPointsEverywhereLocally(f, q)	4188
EulerFactor(J, K)	4162	125.15.2 Descent	4188
125.11.3 Deformation Point Counting . .	4162	qCoverDescent(f, q)	4188
		125.15.3 Descent on the Jacobian	4189

PhiSelmerGroup(f,q)	4190	PseudoAdd(P1, P2, P3)	4197
RankBound(f,q)	4190	PseudoAddMultiple(P1, P2, P3, n)	4197
RankBounds(f,q)	4190	125.17.5 Rational Points on the Kummer Surface	4198
125.15.4 Partial Descent	4192	RationalPoints(K, Q)	4198
qCoverPartialDescent(f,factors,q)	4192	125.17.6 Pullback to the Jacobian	4198
125.16 Kummer Surfaces	4195	Points(J, P)	4198
125.16.1 Creation of a Kummer Surface .	4195	RationalPoints(J, P)	4198
KummerSurface(J)	4195	125.18 Analytic Jacobians of Hyperel- liptic Curves	4199
125.16.2 Structure Operations	4195	125.18.1 Creation and Access Functions .	4200
DefiningPolynomial(K)	4195	AnalyticJacobian(f)	4200
125.16.3 Base Ring	4195	HyperellipticPolynomial(A)	4200
BaseField(K)	4195	SmallPeriodMatrix(A)	4200
BaseRing(K)	4195	BigPeriodMatrix(A)	4200
CoefficientRing(K)	4195	HomologyBasis(A)	4200
125.16.4 Changing the Base Ring	4196	Dimension(A)	4201
BaseChange(K, F)	4196	Genus(A)	4201
BaseExtend(K, F)	4196	BaseField(A)	4201
BaseChange(K, j)	4196	BaseRing(A)	4201
BaseExtend(K, j)	4196	CoefficientRing(A)	4201
BaseChange(K, n)	4196	125.18.2 Maps between Jacobians	4201
BaseExtend(K, n)	4196	ToAnalyticJacobian(x, y, A)	4201
125.17 Points on the Kummer Surface	4196	FromAnalyticJacobian(z, A)	4201
125.17.1 Creation of Points	4196	To2DUpperHalfSpace	
!	4196	FundamentalDomian(z)	4203
!	4196	AnalyticHomomorphisms(t1, t2)	4204
!	4196	IsIsomorphic	
IsPoint(K, S)	4196	SmallPeriodMatrices(t1,t2)	4204
Points(K,[x1, x2, x3])	4196	IsIsomorphic	
125.17.2 Access Operations	4197	BigPeriodMatrices(P1, P2)	4204
P[i]	4197	IsIsomorphic(A1, A2)	4204
Eltseq(P)	4197	IsIsogenousPeriodMatrices(P1, P2)	4204
ElementToSequence(P)	4197	IsIsogenous(A1, A2)	4204
125.17.3 Predicates on Points	4197	EndomorphismRing(P)	4205
eq	4197	EndomorphismRing(A)	4205
ne	4197	125.18.3 From Period Matrix to Curve . .	4208
125.17.4 Arithmetic of Points	4197	RosenhainInvariants(t)	4208
-	4197	125.18.4 Voronoi Cells	4210
*	4197	Delaunay(sites)	4210
*	4197	Voronoi(sites)	4210
Double(P)	4197	125.19 Bibliography	4211

Chapter 125

HYPERELLIPTIC CURVES

125.1 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 \mathbf{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.

125.2 Creation Functions

125.2.1 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)
```

```
HyperellipticCurve(f)
```

```
HyperellipticCurve([f, h])
```

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)`

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)`

`HyperellipticCurveOfGenus(g, f)`

`HyperellipticCurveOfGenus(g, [f, h])`

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)`

Returns the hyperelliptic curve C corresponding to the elliptic curve E , followed by the map from E to C .

125.2.2 Creation Predicates

`IsHyperellipticCurve([f, h])`

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])`

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 H125E1

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)

```

125.2.3 Changing the Base Ring

BaseChange(C, K)

BaseExtend(C, K)

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)

BaseExtend(C, j)

Given a hyperelliptic curve C defined over a field k and a ring map $j : k \rightarrow K$, return a hyperelliptic curve C' over K by applying j to the coefficients of E .

BaseChange(C, n)

BaseExtend(C, n)

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)

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 $k = \mathbf{Q}$ and K is a finite field).

Example H125E2

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  $y^2 + (x + 1)y = x^9 + 100x^2 + 57$ 
over GF(101)
> Q<t> := PolynomialRing(GF(101));
> HyperellipticPolynomials(C1);
 $t^9 + 100t^2 + 57$ 
 $t + 1$ 
> C2, f := SimplifiedModel(C1);
> HyperellipticPolynomials(C2);
 $4t^9 + 98t^2 + 2t + 27$ 
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.

125.2.4 Models**SimplifiedModel(C)**

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 $C \rightarrow C'$.

HasOddDegreeModel(C)

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 $C \rightarrow C'$.

ReducedModel(C)

Simple	BOOLELT	<i>Default : false</i>
Al	MONSTGELT	<i>Default : "Stoll"</i>
Verbose	CrvHypReduce	<i>Maximum : 3</i>

Given a hyperelliptic curve C with integral coefficients, this computes a reduced model C' . If the Stoll algorithm is used (default) then the curve argument must have integral coefficients and reduction is performed with respect to the action of $\mathrm{SL}_2(\mathbf{Z})$ on the (x, z) -coordinates. If the Wamelen algorithm is used, (`Al := "Wamelen"`), the curve must have genus 2. The isomorphism $C \rightarrow C'$ is currently returned only for the algorithm of Stoll.

ReducedMinimalWeierstrassModel(C)

Simple	BOOLELT	<i>Default : false</i>
Verbose	CrvHypMinimal	<i>Maximum : 3</i>
Verbose	CrvHypReduce	<i>Maximum : 3</i>

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 $\mathrm{SL}_2(\mathbf{Z})$, using Stoll's algorithm in `ReducedModel`. The isomorphism $C \rightarrow C'$ is returned as a second value.

SetVerbose("CrvHypReduce", v)

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).

125.2.5 Predicates on Models**IsSimplifiedModel(C)**

Returns `true` if the hyperelliptic curve C is of the form $y^2 = f(x)$.

IsIntegral(C)

Given a hyperelliptic curve C , the function returns `true` if C has integral coefficients, and `false` otherwise.

IspIntegral(C, p)

Given a hyperelliptic curve C defined over \mathbf{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)

Given a hyperelliptic curve C defined over \mathbf{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)`

Given a hyperelliptic curve C defined over \mathbf{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, true` if C is the unique integral minimal model at the given place, (up to transformations that are invertible over the local ring).

125.2.6 Twisting Hyperelliptic Curves

There are standard functions for quadratic twists of hyperelliptic curves in characteristic not equal to 2. In addition, from the new package of Lercier and Ritzenthaler (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)`

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)`

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)`

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)`

Verbose

`CrvHypIso`

Maximum : 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.

`Twists(C)`

For C a genus 2 or 3 hyperelliptic curve over a finite field k , returns the sequence of *all* twists of C (*ie*, a set of representatives of all isomorphism classes of curves over k isomorphic to C over \bar{k}) along with the abstract geometric automorphism group of C (and all of its twists) as a permutation group.

For genus 2, k can be any characteristic. For genus 3, the characteristic of k must be at least 11 and the model of C of the form $y^2 = f(x)$.

There is also a version where the argument is GI , the sequence of Cardona-Quer-Nart-Pujola invariants of a genus 2 curve over a finite field (see Section 125.3.2) which returns the full set of isomorphism classes (twists) of curves over k with the given invariants.

These functions are part of the package contributed by Lercier and Ritzenthaler which is more fully described in the Igusa invariants section.

HyperellipticPolynomialsFromShiodaInvariants(JI)

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 125.3.3) over a finite field of characteristic at least 11. In fact the first return value is a sequence of polynomials $f(x)$ of degree 7 or 8 such that the twisted curves correspond to $y^2 = f(x)$. 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.

This function comes from the genus 3 package contributed by Lercier and Ritzenthaler.

Example H125E3

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 H125E4

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 $\mathbf{Q}(\sqrt{7})$ 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)

```

Example H125E5

We find all the twists of a supersingular genus 2 curve over \mathbf{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

```

125.2.7 Type Change Predicates

IsEllipticCurve(C)

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 $C \rightarrow E$ and the inverse isomorphism $E \rightarrow C$.

125.3 Operations on Curves

125.3.1 Elementary Invariants

HyperellipticPolynomials(C)

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)

The degree of the hyperelliptic curve C or a pointset C of a hyperelliptic curve.

Discriminant(C)

The discriminant of the hyperelliptic curve C .

Genus(C)

The genus of the hyperelliptic curve C .

125.3.2 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 (however@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 <http://www.ma.utexas.edu/users/villegas/cnt/inv.gp>.

In addition, a package of functions written by Reynald Lercier and Christophe Ritzenhaler has been added which 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 [CNP05]. See [CQ05] 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:

$$\begin{aligned} & \left[\frac{J_2^5}{J_{10}}, \frac{J_2^3 J_4}{J_{10}}, \frac{J_2^2 J_6}{J_{10}} \right] \quad J_2 \neq 0 \\ & \left[0, \frac{J_4^5}{J_{10}^2}, \frac{J_4 J_6}{J_{10}} \right] \quad J_2 = 0, J_4 \neq 0 \\ & \left[0, 0, \frac{J_6^5}{J_{10}^2} \right] \quad J_2 = J_4 = 0 \end{aligned}$$

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 [CNP05].

ClebschInvariants(C)

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)

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)

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)

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)

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)

JInvariants(C: parameters)

Quick

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 may not have characteristic 2, 3 or 5 and a faster method using Überschiebungen is employed; otherwise, universal formulae are used.

IgusaInvariants(f, h)

JInvariants(f, h)

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 $\frac{1}{2}$.

IgusaInvariants(f: parameters)

JInvariants(f: parameters)

Quick

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.

ScaledIgusaInvariants(f, h)

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)

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)

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)

Convert Clebsch invariants in the sequence Q to Igusa–Clebsch invariants.

IgusaClebschToIgusa(S)

Convert Igusa–Clebsch invariants in the sequence S to Clebsch invariants.

G2Invariants(C)

Compute and return the sequence of three Cardona-Quer-Nart-Pujola invariants (see the introduction above) for C of genus 2.

G2ToIgusaInvariants(GI)

Convert the sequence of Cardona-Quer-Nart-Pujola invariants (see the introduction above) to a corresponding sequence of Igusa J-invariants.

IgusaToG2Invariants(JI)

Convert the sequence of Igusa J-invariants to Cardona-Quer-Nart-Pujola invariants (see the introduction above).

125.3.3 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 intrinsic for computing the 6 Maeda invariants of the curve ([Mae90]).

This functionality comes from a package contributed by Lercier and Ritzenthaler.

ShiodaInvariants(C)**normalize**

BOOLELT

Default : false

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). The base field of C must be 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 not of a genus 3 hyperelliptic curve anymore. 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 fashion. 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.

ShiodaInvariantsEqual(V1,V2)

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 normized 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)

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)

`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 H125E6

```
> 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)

Returns the six Maeda field invariants ($I_2, I_3, I_4, I_4p, I_8, I_9$) 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.

125.3.4 Base Ring**BaseField(C)****BaseRing(C)****CoefficientRing(C)**

The base field of the hyperelliptic curve C .

125.4 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 [CQ05] 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 of Lercier and Ritzenthaler produces a genus 2 curve from a given set of Cardona-Quer-Nart-Pujola invariants (see Subsection 125.3.2). 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 [CNP05].

Lercier and Ritzenthaler have also contributed a package for genus 3 hyperelliptic curves that produces a curve from a given set of Shioda invariants (see Subsection 125.3.3). This works in characteristic not 2, 3, 5 or 7.

Over a field of characteristic zero, the equation of the curve returned by these methods can involve huge coefficients. For curves over the rationals, this curve can be processed by the algorithm of P. Wamelen [Wam99] and [Wam01]. While, in practice, this algorithm often produces a curve with much smaller coefficients, for certain curves the algorithm may not significantly reduce their size.

HyperellipticCurveFromIgusaClebsch(S)

Reduce

BOOLELT

Default : false

This attempts to build a curve of genus 2 with the given Igusa-Clebsch invariants (see Subsection 125.3.2) 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, the parameter `Reduce` invokes Wamelen's reduction algorithm, and is equivalent to a call to `ReducedModel` with the algorithm specified as "Wamelen".

HyperellipticCurveFromG2Invariants(S)

From a given sequence of Cardona-Quer-Nart-Pujola invariants (as described in Subsection 125.3.2) 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)
--

HyperellipticPolynomialFromShiodaInvariants(JI)

Given a sequence of 9 Shioda invariants JJ over the rationals or a finite field k , 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 JJ are singular Shioda invariants.

The second intrinsic also works for singular invariants. It returns a polynomial f of degree ≤ 8 with the given invariants. If JJ are non-singular, $y^2 = f(x)$ is a genus 3 curve with invariants JJ .

Example H125E7

A typical run would look like:

```
> SetVerbose("Igusa",1);
> IgCl := [ Rationals() |
>   -549600, 8357701824, -1392544870972992, -3126674637319431000064 ];
> time C := HyperellipticCurveFromIgusaClebsch(IgCl);
Found conic point:
(-64822283782462146583672682837123736006679080996161747198790\
94839597076141357421875/1363791104410093031413327266775768846\
086857295667582286386963073756856153402049457884003686477312,\
1018800933596853119179482113258441387813354952965788880045011\
3779781952464580535888671875/107297694738676628355433722672369\
86876862256732919126043768293539723478848063808819839532581156\
```

```

5610468983296)
Time: 0.990
> time C := ReducedModel(C : A1 := "Wamelen");
Time: 161.680
> HyperellipticPolynomials(C);
-23*x^6 + 52*x^5 - 55*x^4 + 40*x^3 - 161*x^2 + 92*x - 409
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
> _,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 +
  Ψ3*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

```

125.5 Function Field

125.5.1 Function Field and Polynomial Ring

`FunctionField(C)`

The function field of the hyperelliptic curve C .

`DefiningPolynomial(C)`

A weighted homogeneous polynomial for the hyperelliptic curve C .

`EvaluatePolynomial(C, a, b, c)`

`EvaluatePolynomial(C, [a, b, c])`

Evaluates the homogeneous defining polynomial of the hyperelliptic curve C at the point (a, b, c) .

125.6 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 $(1 : y : 0)$.

125.6.1 Creation of Points

`C ! [x, y]`

`C ! [x, y, z]`

`elt< PS | x, y >`

`elt< PS | x, y, z >`

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`

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)`

`RationalPoints(C, x)`

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)`

The points at infinity for the hyperelliptic curve C returned as an indexed set of points.

`IsPoint(C, S)`

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 H125E8

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

```

125.6.2 Random Points

`Random(C)`

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.

125.6.3 Predicates on Points

`P eq Q`

Returns `true` if and only if the two points P and Q on the same hyperelliptic curve have the same coordinates.

`P ne Q`

Returns `false` if and only if the two points P and Q on the same hyperelliptic curve have the same coordinates.

125.6.4 Access Operations

`P[i]`

The i -th coordinate of the point P , for $1 \leq i \leq 3$.

`Eltseq(P)`

`ElementToSequence(P)`

Given a point P on a hyperelliptic curve, this returns a 3-element sequence consisting of the coordinates of the point P .

125.6.5 Arithmetic of Points

`-P`

`Involution(P)`

Given a point P on a hyperelliptic curve, this returns the image of P under the hyperelliptic involution.

125.6.6 Enumeration and Counting Points

`NumberOfPointsAtInfinity(C)`

The number of points at infinity on the hyperelliptic curve C .

`PointsAtInfinity(C)`

The points at infinity for the hyperelliptic curve C returned as an indexed set of points.

`#C`

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)`

`RationalPoints(C)`

<code>Bound</code>	<code>RNGINTELT</code>	<i>Default :</i>
<code>NPrimes</code>	<code>RNGINTELT</code>	<i>Default : 30</i>
<code>DenominatorBound</code>	<code>RNGINTELT</code>	<i>Default : Bound</i>

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 \mathbf{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, looks for points using a sieve method, described in Appendix A of [Bru02]. The parameter `NPrimes` controls the number of primes which are used and `DenominatorBound` the size of the denominators used.

`PointsKnown(C)`

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.

ZetaFunction(C)

Given a hyperelliptic curve C defined over a finite field, this function computes the zeta function of C . The zeta function is returned as an element of the function field in one variable over the integers.

If the base field is small or there is no other good alternative, the method used is a naive point count on the curve over extensions of degree $1, \dots, g$ of the base field. However, if they are applicable, the faster p -adic methods described in the #J section are employed. As for #J, the verbose flag `JacHypCnt` can be used to output information about the computation.

ZetaFunction(C, K)

Given a hyperelliptic curve C defined over the rationals, this function computes the zeta function of the base extension of C to K . The curve C must have good reduction at the characteristic of K .

Example H125E9

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) @}
```

125.6.7 Frobenius**Frobenius(P, F)****Check**

BOOLELT

Default : true

Applies the Frobenius $x \mapsto x^{(\#F)}$ to P . If **Check** is **true**, it verifies that the curve of which P is a point is defined over the finite field F .

125.7 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.

125.7.1 Creation of Isomorphisms

Aut(C)

Given a hyperelliptic curve C , this returns the structure of all automorphisms of the curve.

Iso(C1, C2)

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)

Transformation(C, u)

Transformation(C, e)

Transformation(C, e, u)

Transformation(C, t, e, u)

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 the isomorphism to the curve.

Example H125E10

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)

```

125.7.2 Arithmetic with Isomorphisms

Hyperelliptic curve isomorphisms can be evaluated, inverted, and composed on points as right operators. Note that the functional notation $f(P)$ is not presently available for these maps of curves. However the available map syntax \circ is more consistent with functions as operating on the right (which determines and is apparent from the way composition is defined).

$f * g$

The composition of the maps f and g as right operators.

$\text{Inverse}(f)$

The inverse of the hyperelliptic curve isomorphism f .

$f \text{ in } M$

Given an isomorphism of hyperelliptic curves f , and the structure of isomorphisms M between two hyperelliptic curves, this returns **true** if and only if they share the same domains and codomains.

$P \circ f$

$\text{Evaluate}(f, P)$

Returns the evaluation of f at a point P in its domain. The same functions apply equally to points in the Jacobian of the domain curve and, in the case of genus 2, to points on the associated Kummer surface.

$P \circ\circ f$

$\text{Pullback}(f, P)$

Returns the inverse image of the isomorphism f at a point P in its codomain. The same functions apply equally to points in the Jacobian of the codomain curve and, in the case of genus 2, to points on the associated Kummer surface.

$f \text{ eq } g$

Given isomorphisms f and g of hyperelliptic curves having the same domain and codomain, this function returns **true** if and only if they are equal. Note that two isomorphisms may be equal even if their defining data are distinct (see example below).

125.7.3 Invariants of Isomorphisms

Parent(f)

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)

The curve which is the domain of the given isomorphism.

Codomain(f)

The curve which is the target of the given isomorphism.

125.7.4 Automorphism Group and Isomorphism Testing

This section details the features for computing isomorphisms between two curves and determining the group of automorphisms. The function `IsGL2Equivalent` plays a central role in the isomorphism testing, and is documented here due to its central role in these computations.

The genus 2 package of Lercier and Ritzenthaler 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 125.3.2). 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.

Lercier and Ritzenthaler’s genus 3 package provides the same for genus 3 hyperelliptic curves, working with Shioda invariants. Here, the characteristic has to be 0 or ≥ 11 .

IsGL2Equivalent(f, g, n)

This function returns `true` if and only if f and g are in the same $GL_2(k)$ -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 $f((ax + b)/(cx + d))(cx + d)^n$.

IsIsomorphic(C1, C2)

Verbose

CrvHypIso

Maximum : 3

This function returns `true` if and only if the hyperelliptic curves $C1$ and $C2$ are isomorphic over their common base field. If the curves are isomorphic, an isomorphism is returned.

AutomorphismGroup(C)

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 $C \times G \rightarrow C$.

Example H125E11

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
> 0 := C2![1,0,0];
> auts := [ m2(g) : g in G2 ];
> [ f : f in auts | 0@f eq 0 ];
[
  (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 an 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 \mathbf{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 \mathbf{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 $\mathbf{Z}/2\mathbf{Z}$, and in latter case the automorphism group is an extension (of the group of order 12) by the abelian group isomorphic to $\mathbf{Z}/2\mathbf{Z} \times \mathbf{Z}/2\mathbf{Z}$. 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.

GeometricAutomorphismGroup(C)

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 125.3.2) of C and use the classification of the possible automorphism groups in terms of the invariants. See [SV01] and [CQ05] for the odd (and 0) characteristic case and [CNP05] for characteristic 2.

The method used for genus 3 is to compute the Shioda invariants (see Subsection 125.3.3) 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 of packages for genus 2 and 3 curves contributed by Reynald Lercier and Christophe Ritzenthaler.

`GeometricAutomorphismGroupFromShiodaInvariants(JI)`

There is a variant of the last intrinsic for genus 3 curves where the argument is the sequence JJ 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 H125E12

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)

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 is part of a package for genus 2 curves contributed by Reynald Lercier and Christophe Ritzenthaler and is based on the classification analysis in [Car03] and [CNP05].

GeometricAutomorphismGroupGenus3Classification(F)

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 is part of a package for genus 3 curves contributed by Reynald Lercier and Christophe Ritzenthaler.

Example H125E13

We determine the possible (geometric) automorphism groups for genus 2 curves over \mathbf{F}_2 .

```
> gps,ncls := GeometricAutomorphismGroupGenus2Classification(GF(2));
> [#gp : gp in gps];
[ 2, 12, 32, 160 ]
> ncls;
[ 5, 1, 1, 1 ]
```

125.8 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 \mathbf{Q} .

125.8.1 Creation of a Jacobian

Jacobian(C)

The Jacobian of the hyperelliptic curve C .

125.8.2 Access Operations

Curve(J)

The hyperelliptic curve from which the Jacobian J was constructed.

Dimension(J)

The dimension of the Jacobian J as an algebraic variety, equal to the genus of the curve C of which J is the Jacobian.

125.8.3 Base Ring

BaseField(J)

BaseRing(J)

CoefficientRing(J)

The base field of the Jacobian J .

125.8.4 Changing the Base Ring

BaseChange(J, F)

BaseExtend(J, F)

The base extension of the Jacobian J to the field F .

BaseChange(J, j)

BaseExtend(J, j)

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)

BaseExtend(J, n)

The base extension of the Jacobian J over a finite field to its degree n extension.

125.9 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 $\Phi : J \rightarrow A$ between principally polarized abelian surfaces, such that the kernel of Φ over the algebraic closure has group structure $\mathbf{Z}/2\mathbf{Z} \times \mathbf{Z}/2\mathbf{Z}$. 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

$$f(x) = c\text{Norm}_{L[x]/k[x]}Q(x)$$

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 $A = \text{Jac}(D)$ 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 $f(x) = c\text{Norm}_{L[x]/k[x]}Q(x)$. and one can find the relevant elliptic curves as quotients of C .

<code>RichelotIsogenousSurfaces(J)</code>

<code>RichelotIsogenousSurfaces(C)</code>

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.

<code>RichelotIsogenousSurface(J, kernel)</code>
--

<code>RichelotIsogenousSurface(C, kernel)</code>
--

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 $\text{Norm}_{L[x]/k[x]}Q(x) = cf(x)$. 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 H125E14

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  $y^2 = -2x^5 - 2x$  over
Rational Field,
Elliptic Curve defined by  $y^2 = x^3 + 5/32x^2 + 5/1024x + 1/32768$  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 \mathbf{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 : ExcFactors:="Ogg");
> myL:=func< A |
>   case<Type(A) | SetCart : LSeries(A[1])*LSeries(A[2]),
>   JacHyp : LSeries(Curve(A) : ExcFactors:="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 ]

```

125.10 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.

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 $y = B(x, z)$ make 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 \mathbf{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 $b \bmod a$, 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.

125.10.1 Creation of Points

```
J ! 0
```

```
Id(J)
```

```
Identity(J)
```

The identity element on the Jacobian J .

```
J ! [a, b]
```

```
elt< J | a, b >
```

```
elt< J | [a, b] >
```

```
elt< J | a, b, d >
```

```
elt< J | [a, b], d >
```

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
```

```
J ! [P, Q]
```

```
elt< J | P, Q >
```

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]
```

```
elt< J | S, T >
```

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)`

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`

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)`

`RationalPoints(J, a, d)`

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 H125E15

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; // Magma didn't do much work to create 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 -> 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 H125E16

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 $J(\mathbf{Q})$.

```
> _<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 H125E17

A Jacobian with a point not coming from the curve.

Any curve containing the point $(\sqrt{2}, \sqrt{2})$ has a \mathbf{Q} -rational divisor

$$D := (\sqrt{2}, \sqrt{2}) + (-\sqrt{2}, -\sqrt{2}).$$

This will give a nontrivial point in $J(\mathbf{Q})$. 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 $J(\mathbf{Q})$.

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
```

125.10.2 Random Points

`Random(J)`

A random point on a Jacobian J of a hyperelliptic curve defined over a finite field.

125.10.3 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`

Returns `true` if and only if the points P and Q on the same Jacobian are equal.

`P ne Q`

Returns `false` if and only if the two points P and Q on the same Jacobian are equal.

`IsZero(P)`

`IsIdentity(P)`

Returns `true` if and only if P is the zero element of the Jacobian.

125.10.4 Access Operations

$P[i]$

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.

$Eltseq(P)$

$ElementToSequence(P)$

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.

125.10.5 Arithmetic of Points

For each of the following functions, P and Q are points on the Jacobian of a hyperelliptic curve.

$-P$

The additive inverse of the point P on the Jacobian.

$P + Q$

The sum of the points P and Q on the same Jacobian.

$P += Q$

Given two points P and Q on the same Jacobian, set P equal to their sum.

$P - Q$

The difference of the points P and Q on the same Jacobian.

$P -= Q$

Given two points P and Q on the same Jacobian, set P equal to the difference $P - Q$.

$n * P$

$P * n$

The n -th multiple of P in the group of rational points on the Jacobian.

$P *:= n$

Set P equal to the n -th multiple of itself.

125.10.6 Order of Points on the Jacobian

Order(P)

Returns the order of the point P on the Jacobian J of a hyperelliptic curve defined over a finite field or the rationals, or 0 if P has infinite order. This first computes $\#J$ when J is defined over a finite field.

Order(P, l, u)

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 [GH00]).

If UseInversion is true the search space is halved by using point negation.

Order(P, l, u, n, m)

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 $n \bmod m$. This does not compute $\#J$.

The two parameters Alg and UseInversion have the same use as in the previous function.

HasOrder(P, n)

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 .

125.10.7 Frobenius

Frobenius(P, k)

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 $x \rightarrow x^q$. If Check is true, MAGMA verifies that the Jacobian of P is defined over k .

125.10.8 Weil Pairing

WeilPairing(P , Q , m)

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 H125E18

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);
> 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.

125.11 Rational Points and Group Structure over Finite Fields

125.11.1 Enumeration of Points

`Points(J)`

`RationalPoints(J)`

Given a Jacobian J of a hyperelliptic curve defined over a finite field, determine all rational points on the Jacobian J .

125.11.2 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 [GH00]. 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 [LL]. 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)`

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`

`Order(J)`

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.

<code>NaiveAlg</code>	BOOLELT	<i>Default</i> : <code>false</code>
<code>ShanksLimit</code>	RNGINTELT	<i>Default</i> : 10^{12}
<code>CartierManinLimit</code>	RNGINTELT	<i>Default</i> : 5×10^5
<code>UseSubexpAlg</code>	BOOLELT	<i>Default</i> : <code>true</code>

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 H125E19

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 H125E20

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 H125E21

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 H125E22

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
37780017712685037895040
> 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 H125E23

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 H125E24

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)

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)

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)

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)

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 .

125.11.3 Deformation Point Counting

JacobianOrdersByDeformation(Q, Y)

EulerFactorsByDeformation(Q, Y)

ZetaFunctionsByDeformation(Q, Y)

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.

$Q(x, z)$ should be a 2-variable polynomial 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 $y^2 = Q(x, z)$.

Y should be a sequence of 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 \mathbf{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 H125E25

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 \mathbf{F}_9 and the parameter values are taken in \mathbf{F}_{340} 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;
[
  12157665459056928801*$.1^2 + 852459373*$.1 + 1,
  12157665459056928801*$.1^2 + 1088717005*$.1 + 1,
  12157665459056928801*$.1^2 + 6911064226*$.1 + 1,
  12157665459056928801*$.1^2 - 607949990*$.1 + 1
]
```

125.11.4 Abelian Group Structure**Sylow(J, p)**

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)

UseGenerators	BOOL	<i>Default</i> : false
Generators	SETENUM	<i>Default</i> :

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 **UseGenerators** is set then the group structure computation is achieved by extracting relations from the user-supplied set of generators in **Generators**.

HasAdditionAlgorithm(J)

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.

125.12 Jacobians over Number Fields or \mathbf{Q}

Some functions in this section work for general number fields (notably `TwoSelmerGroup`), while many are only implemented over \mathbf{Q} .

125.12.1 Searching For Points

`Points(J)`

`RationalPoints(J)`

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.

125.12.2 Torsion

`TwoTorsionSubgroup(J)`

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)`

Given the Jacobian J of a hyperelliptic curve defined over the rationals, this function returns a bound on the size of the rational torsion subgroup of the Jacobian. The bound is obtained by examining the group $J(\mathbf{F}_p)$ for the first n good primes p .

`TorsionSubgroup(J)`

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 H125E26

For the curve

$$C : y^2 = (x + 3)(x + 2)(x + 1)x(x - 1)(x - 2),$$

the only \mathbf{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 \mathbf{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
> 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

```

125.12.3 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 \mathbf{Q} .

In the case of genus two curves defined over \mathbf{Q} , the canonical height is computed using the algorithm of Flynn and Smart[FS97] with improvements by Stoll[Sto99]. This algorithm computes the canonical height using local error functions on the associated Kummer surface.

In all other cases the algorithm described in chapter 5 of [Mül10a] 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 125.10); 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 114.12.2) 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 125.18.

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 $n \in \mathbf{Z}$, 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}^{2g-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}^{2g-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)

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 $J \rightarrow K \rightarrow \mathbf{P}^3$.

Height(P: parameters)

CanonicalHeight(P: parameters)

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 \mathbf{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)

Effort	RNGINTELT	Default : 0
Factor	BOOLELT	Default : false

Given the Jacobian J of a genus 2 curve over \mathbf{Q} of the form $y^2 = f(x)$ with integral coefficients, this computes a real number c such that $h(P) \leq \hat{h}(P) + c$ for all P in $J(\mathbf{Q})$, where h is the naive height and \hat{h} is the canonical height.

The parameter **Effort** (which can be 0, 1 or 2) indicates how much effort should be put into finding a good bound. The second value returned is a bound for μ_∞ , the contribution from the infinite place. If the parameter **Factor** is **true**, then the discriminant will be factored, and its prime divisors will be considered individually, usually resulting in an improvement of the bound.

HeightPairing(P, Q: parameters)

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 \mathbf{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)

`Precision` `RNGINTELT` *Default : 0*

Given a sequence $[P_1, \dots, P_n]$ 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)

`Precision` `RNGINTELT` *Default : 0*

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)

`Precision` `RNGINTELT` *Default : 0*

Given a sequence of points on the Jacobian J of a hyperelliptic curve defined over a number field k , this function returns an LLL- reduced basis for the subgroup of $J(k)/J_{tors}(k)$ generated by the given points (that is, the function reduces the real lattice formed by the points, under the positive definite quadratic form given by the canonical height pairing). The height pairing matrix of the sequence is returned as a second value.

Example H125E27

This example illustrates some basic properties of heights, and proves that a certain point in $J(\mathbf{Q})$ is not a nontrivial multiple of any other point in $J(\mathbf{Q})$. 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 ];
[ 0.479839797450405152023279542502, 0.000000000000000000000000000000,
0.479839797450405152023279542491, 0.479839797450405152023279542491,
0.000000000000000000000000000000 ]
```

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 $J(\mathbf{Q})/J_{tors}(\mathbf{Q})$. The next command verifies this.

```
> ReducedBasis(ptsJ);
[ (x + 1, x^3 + 3, 2) ]
[0.479839797450405152023279542491]
> P := ptsJ[3];
> P;
(x + 1, x^3 + 3, 2)
```

So this point (which is the third 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[1], 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);
529.0000000000000000000000000000
```

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, Factor);
> HC;
3.73654623288305720113473940376
```

In particular, all torsion points should have naive height less than this.

```
> for T in Jtors do
>   NaiveHeight(maptoJ(T));
> end for;
0.000000000000000000000000000000
```

```

0.693147180559945309417232121458
0.693147180559945309417232121458
0.693147180559945309417232121458
0.693147180559945309417232121458
0.693147180559945309417232121458
0.693147180559945309417232121458
0.693147180559945309417232121458
0.693147180559945309417232121458
> // Does the inequality hold for 23*P?
> NaiveHeight(23*P) - Height(23*P) le HeightConstant(J);
true

```

Finally, we show that P is not a nontrivial multiple of another point in $J(\mathbf{Q})/J_{tors}(\mathbf{Q})$ (implying that P generates $J(\mathbf{Q})/J_{tors}(\mathbf{Q})$ if this has rank 1). For suppose that $P = nQ + T$ for some $Q \in J(\mathbf{Q}), T \in J_{tors}(\mathbf{Q})$. 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 ]);
[ (x^2 + 1/2*x + 1/2, 1/4*x - 5/4, 2) ]
[0.479839797450405152023279542491]

```

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 H125E35, where we prove that $J(\mathbf{Q})$ has rank 1, and then use the fact that P generates $J(\mathbf{Q})/J_{tors}(\mathbf{Q})$ to find all rational points on C .

Example H125E28

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
[0.2797317933688278287667056839      -5.005347463630776922E-17]
[      -5.005347463630776922E-17  0.9524462128097307697800077959]
> Determinant(M);
0.2664294871966142247655164984

```

Since $\text{Det}(M)$ is nonzero, the two generators in B are independent.

125.12.4 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.

Starting with MAGMA 2.13, a simpler user interface for computing 2-Selmer groups was introduced. The two implementations that were previously available still exist internally. In MAGMA 2.18-4 a new implementation has been included. When `TwoSelmerGroup` is called, it chooses one of them (or the user may choose).

<code>BadPrimes(C)</code>

<code>BadPrimes(J)</code>

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.

<code>HasSquareSha(J)</code>

<code>IsEven(J)</code>

Given the Jacobian of a hyperelliptic curve over \mathbf{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)

For a genus 2 curve defined over \mathbf{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 \mathbf{Q}_p (when p is a prime) or over \mathbf{R} (when p is 0). Equivalently, C is deficient if there is no \mathbf{Q}_p -rational divisor of odd degree on C .

HasIndexOne(C, p)

HasIndexOne(C, p)

Given a hyperelliptic curve C over \mathbf{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)

Returns true if and only if C has index one over all completions of its base field (including real completions).

TwoSelmerGroup(J)

Al	MONSTGELT	<i>Default :</i>
Fields	SETENUM	<i>Default :</i>
ReturnFakeSelmerData	BOOLELT	<i>Default : false</i>
ReturnRawData	BOOLELT	<i>Default : false</i>
Verbose	Selmer	<i>Maximum : 3</i>

The 2-Selmer group of the Jacobian of a hyperelliptic curve defined over \mathbf{Q} or a number field. If the curve is not defined over \mathbf{Q} 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 $A^*/(A^*)^2\mathbf{Q}^*$ (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 37.6.1). 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 $\mathbf{Q}^*/(\mathbf{Q}^*)^2$. The fake Selmer group S is then determined by the exact sequence $0 \rightarrow S_3 \rightarrow S_2 \rightarrow S_1 \rightarrow S \rightarrow 0$. 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 \mathbf{Z}^F , where F is the number of elements in the `factorbase`. The image in $A^*/(A^*)^2$ of the i th Selmer group generator is then the sum over j of $f_j^{e_j}$ where (e_j) is the i th exponent vector.

<code>RankBound(J)</code>

<code>RankBounds(J)</code>

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 \mathbf{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 print a warning if it is not.

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 $\text{Sha}[2]$. 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 $\text{Sha}[2]$, 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 $J(\mathbf{Q})/2J(\mathbf{Q})$.

Example H125E29

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 + 33018689414366470785600*x^2
+ 45012299099933971943424000000*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;
```

Abelian Group isomorphic to $\mathbf{Z}/2 + \mathbf{Z}/2 + \mathbf{Z}/2 + \mathbf{Z}/4$

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 $\mathbf{Z}/4$ 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 $J(\mathbf{Q})[2]$ has order 16, and that the rank of $J(\mathbf{Q})$ 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 $J(\mathbf{Q})$ has rank 2, even though the other generator of $J(\mathbf{Q})$ 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 H125E30

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 $a = 1 \pmod{3}$, 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.)
>     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 $a = 1 \pmod{3}$, 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 $J(\mathbf{Q})$, 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 $J(\mathbf{Q})$ has rank 0. This rank bound is unconditional. In fact, the computation proves that there is a subgroup $\mathbf{Z}/2$ 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 H125E31

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 $\text{Jac}(C_d)(\mathbf{Q})$ is at least four. We note that $\text{rankJac}(C)(\mathbf{Q}(\sqrt{d})) = \text{rankJac}(C)(\mathbf{Q}) + \text{rankJac}(C_d)(\mathbf{Q})$, so if we can find an upper bound on the Mordell-Weil rank of $\text{Jac}(C)$ over $\mathbf{Q}(\sqrt{d})$ then we also find an upper bound on $\text{rankJac}(C_d)(\mathbf{Q})$. We first try this with unproven class group data.

```
> SetClassGroupBounds(1000);
```

```
> K:=QuadraticField(d);
```

```
> CK:=BaseChange(C,K);
```

```
> JCK:=Jacobian(CK);
```

```
> NumberOfGenerators(TwoSelmerGroup(JCK));
```

```
5
```

We find that $\text{rankJac}(C)(\mathbf{Q}(\sqrt{d})) \leq 5$, which means that $\text{Jac}(C)$ must indeed have Mordell-Weil rank 1 over \mathbf{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 H125E32

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 `A1 := "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 H125E33

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);
WARNING: upper bound is only an upper bound for the rank of Pic^0(X).
2
```

The warning appears 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 \mathbf{Q} -rational divisor class is represented by a \mathbf{Q} -rational divisor. In particular $Pic^0(C) = J(\mathbf{Q})$. So we can disregard the warning.

```
> HasIndexOneEverywhereLocally(C);
false 0
> Roots(f,RealField(50));
[]
> Evaluate(f,0) %t 0;
true
> &and[ HasIndexOne(C,p) : p in BadPrimes(C) ];
true
```

125.13 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 $\pi_\delta : D_\delta \rightarrow C$ 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)		
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 H125E34

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 \mathbf{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) @}
]

```

125.14 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 \mathbf{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 require a generator of the Mordell-Weil group of the Jacobian (which must have rank 1) to be provided.

In addition a routine `Chabauty0` is provided to handle the trivial case where the Jacobian has rank 0.

How Chabauty's method works: Under the assumption that the rank of $J = Jac(C)$ is strictly less than the genus of C , the closure V of $J(\mathbf{Q})$ in $J(\mathbf{Q}_p)$ (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 $J(\mathbf{Q})$. The image of C in J (under a natural embedding) will intersect V in a finite set, and this set must contain $C(\mathbf{Q})$. By examining the power series that define this finite intersection, one obtains upper bounds for the number of points on $C(\mathbf{Q})$ in each congruence class modulo p (or any power of p).

Chabauty0(J)

Given the Jacobian of a hyperelliptic curve C over \mathbf{Q} , the function finds all rational points on C , under the assumption that $J(\mathbf{Q})$ has rank zero.

The assumption is not checked. When it does not hold, the function computes a subset of $C(\mathbf{Q})$ corresponding (in some sense) to the torsion in $J(\mathbf{Q})$.

Chabauty(P : ptC)**ptC****PT***Default :*

For a curve C of genus 2 over \mathbf{Q} , this returns the full set of rational points. The algorithm involves Chabauty's method combined with a Mordell-Weil sieve. The argument P should be a rational point on the Jacobian of C , and P is assumed to generate the Mordell-Weil group (which must have rank 1 in order for Chabauty's method to be applicable).

The algorithm requires knowledge of one rational point on the curve. (In particular, it cannot be used to show that a curve has no rational points!) Such a point may be supplied as the optional argument **ptC**; otherwise, one is found by searching.

Chabauty(P, p: Precision)**Precision****RNGINTELT***Default : 5*

Given a point on the Jacobian of a hyperelliptic curve C over \mathbf{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 $J(\mathbf{Q})/J_{tors}(\mathbf{Q})$. 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 H125E35

In this example, which continues Example H125E27, 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), (-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 $J(\mathbf{Q})$. In Example H125E27 we proved that $PJ2$ is not divisible in $J(\mathbf{Q})$. Since we now know $J(\mathbf{Q})$ has rank 1, we can conclude that $PJ2$ is a generator of $J(\mathbf{Q})/J(\mathbf{Q})_{tors}$. Therefore we can use it for Chabauty's method.

```
> all_pts := Chabauty(PJ2); all_pts;
{ (1 : -2 : 1), (-1 : -2 : 1), (1 : 2 : 1),
  (1 : -1 : 0), (1 : 1 : 0), (-1 : 2 : 1) }
```

So we find that we already had the full set of rational points. 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>, <80, 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 H125E36

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 $J(\mathbf{Q})$ has rank 1, and PJ has infinite order. Now we check that PJ generates $J(\mathbf{Q})/J_{tors}(\mathbf{Q})$. (For more explanation, see the similar calculation in Example H125E27).

```
> 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) ]
[0.326617338771488428260076768590]
> Height(PJ);
0.326617338771488428260076768590

```

This shows there are no points in $J(\mathbf{Q})$ with canonical height smaller than that of PJ , so PJ is a generator modulo torsion and we may use it for Chabauty's method.

```

> 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) @}

```

Example H125E37

In Example H125E29 (in the section on 2-Selmer groups) we found a curve of rank 2. Although we cannot apply Chabauty's method to get information about the full set of rational points on C , we can still apply the routines to 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 we find all the points $P \in C(\mathbf{Q})$ for which the image $P - P_0 \in J(\mathbf{Q})$ lies in the subgroup generated by PJ , where P_0 denotes $\text{ptsC}[1]$.

```

> pts := Chabauty(PJ : ptC:=ptsC[1] ); pts;
{ (-1806336 : 0 : 1), (0 : 0 : 1), (43264 : 44828581639628800 : 1),
  (1 : 0 : 0), (-29224836 : 0 : 1), (-7290000 : 0 : 1),
  (-116964225 : 0 : 1), (43264 : -44828581639628800 : 1) }

```

Example H125E38

We find all rational points on $y^2 = x^6 + 1$, whose Jacobian J has rank 0, ie the group $J(\mathbf{Q})$ is finite.

```

> _<x> := PolynomialRing(Rationals());
> C := HyperellipticCurve(x^6+1);
> J := Jacobian(C);
> RankBound(J);
0

```

```
// So we may use Chabauty0.
> Chabauty0(J);
{@ (1 : -1 : 0), (1 : 1 : 0), (0 : 1 : 1), (0 : -1 : 1) @}
```

This Selmer computation required class/unit computations in the fields given by roots of $x^6 + 1$. In this example, there's clearly an approach requiring less work: to use the self-evident map from C to the elliptic curve $y^2 = x^3 + 1$, which has rank 0.

To define the map, we need to refer to the space where C lives, which is a weighted projective space in which 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 (otherwise MAGMA would have already complained).

```
> CtoE;
Mapping from: CrvHyp: C to CrvEll: E
with equations :
X^2*Z
Y
Z^3
```

Now 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. Of course, we could instead 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) @}
{@ @}
{@ @}
```

125.15 Cyclic Covers of \mathbf{P}^1

A q -cyclic cover of the projective line is a curve C which admits a degree q morphism to \mathbf{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.

125.15.1 Points

A point on a cyclic cover $C : y^q = f_d x^d + f_{d-1} x^{d-1} + \dots + f_0$ 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 $\mathbf{P}^2(1 : d/q : 1)$. In this case a point is represented by a triple $[X, Y, Z]$ satisfying

$$Y^q = f_d X^d + f_{d-1} X^{d-1} Z + \dots + f_0 Z^d.$$

When $d = \text{Degree}(f(x))$ is not divisible by q , we consider points in the weighted projective plane $\mathbf{P}^2(q : d : 1)$. In this case a point is represented by a triple $[X, Y, Z]$ satisfying

$$Y^q = f_d X^d + f_{d-1} X^{d-1} Z^q + \dots + f_0 Z^{dq}.$$

RationalPoints(f,q)

Bound	RNGINTELT	<i>Default : 0</i>
DenominatorBound	RNGINTELT	<i>Default : 0</i>
NPrimes	RNGINTELT	<i>Default : 0</i>

For a polynomial f defined over a number field (or \mathbf{Q}) this Searches for rational points on the curve $y^q = f(x)$ in a box bounded by **Bound** (which must be specified). 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 $\text{LCM}(q, \text{Degree}(f(x))) / \text{Degree}(f(x))$.

The search is by a sieve method described in Appendix A of [Bru02]. The parameter **NPrimes** controls the number of primes which are used and **DenominatorBound** the size of the denominators used.

HasPoint(f,q,v)

HasPoint(f,q,v)

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 $\text{LCM}(q, \text{Degree}(f(x)))/\text{Degree}(f(x))$.

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)

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 .

125.15.2 Descent

Let C be a curve with Jacobian J and suppose $\psi : A \rightarrow J$ is an isogeny of abelian varieties. We say that a morphism of curves $\pi : D \rightarrow C$ 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 $\mathbf{Z}[\zeta]$, 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)

<code>PrimeBound</code>	<code>RNGINTELT</code>	<i>Default : 0</i>
<code>PrimeCutoff</code>	<code>RNGINTELT</code>	<i>Default : 0</i>
<code>KnownPoints</code>	<code>SET</code>	<i>Default :</i>
<code>Verbose</code>	<code>CycCov</code>	<i>Maximum : 3</i>

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 \mathbf{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 H125E39

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;
{}
```

125.15.3 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 [PS97]. 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 [PS97], or consider the example below.

PhiSelmerGroup(f, q)

ReturnRawData	BOOLELT	<i>Default : false</i>
Verbose	Selmer	<i>Maximum : 3</i>

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.

RankBound(f, q)**RankBounds(f, q)**

ReturnGenerators	BOOLELT	<i>Default : false</i>
Verbose	Selmer	<i>Maximum : 3</i>

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 \mathbf{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 H125E40

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 \mathbf{Q} (see [PS97]). 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 $J(\mathbf{Q})$ 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;
[*
  T^3 - T^2 + 4*T + 4
*]

```

This polynomial defines a divisor on \mathbf{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 $P \in \mathbf{P}^1$ under the degree 3 map $C \rightarrow \mathbf{P}^1$. Then $D - D_0$ is a degree 0 defined over the rationals. According to `RankBounds` its class has infinite order in $J(\mathbf{Q})$. This was determined by considering its image under the composition, $J(\mathbf{Q}) \rightarrow J(k) \rightarrow \text{Sel}^\phi(J/k)$. 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 \mathbf{Q} has been computed unconditionally it follows that the ϕ -torsion subgroup of Sha is isomorphic to $(\mathbf{Z}/3)^2$, and since it contains an element not divisible by ϕ in Sha one can deduce that the 3-primary part of Sha is also isomorphic to $(\mathbf{Z}/3)^2$.

125.15.4 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 $(h_1(x), \dots, h_n(x))$ 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 $J[\phi]$.

<code>qCoverPartialDescent(f, factors, q)</code>		
--	--	--

<code>PrimeBound</code>	<code>RNGINTELT</code>	<i>Default : 0</i>
<code>PrimeCutoff</code>	<code>RNGINTELT</code>	<i>Default : 0</i>
<code>KnownPoints</code>	<code>SET</code>	<i>Default :</i>
<code>Verbose</code>	<code>CycCov</code>	<i>Maximum : 3</i>

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 étale 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 H125E41

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

$$\begin{aligned}\lambda^2 y_1^3 &= x^2 + 1, \\ 9\lambda^2 y_2^3 &= x^2 + 17, \\ (2\theta + 2)\lambda y_3^3 &= x - \theta,\end{aligned}$$

where θ is a square root of 17. This covering of D also evidently covers the genus one curve defined over $\mathbf{Q}(\theta)$,

$$C : (2\theta + 2)v^3 = (x^2 + 1)(x - \theta).$$

One can now use the method of Elliptic Curve Chabauty to find all points of $Y_\delta(K)$ which map to points of $D(\mathbf{Q})$. Namely any such point gives rise to a point of $C(K)$ with x -coordinate in \mathbf{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];
> coverttoE:=coverstoE[1];
> pts:={@pt@coverttoE : 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 $E(K) \simeq \mathbf{Z}$, 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 $E \rightarrow \mathbf{P}^1$ such that the composition $C \rightarrow E \rightarrow \mathbf{P}^1$ 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 $(-1, -12) \in D(\mathbf{Q})$. A similar argument proves that the only rational point coming from the other element of the partial Selmer set is $(1, -12)$, proving $D(\mathbf{Q}) = \{(1, 12), (-1, -12)\}$.

125.16 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 [Mül10b] which extends earlier work by Flynn, described in chapter 3 of [CF96].

125.16.1 Creation of a Kummer Surface

`KummerSurface(J)`

The Kummer surface of the Jacobian J of a genus 2 curve.

125.16.2 Structure Operations

`DefiningPolynomial(K)`

The defining polynomial of the Kummer surface K .

125.16.3 Base Ring

`BaseField(K)`

`BaseRing(K)`

`CoefficientRing(K)`

The base field of the Kummer surface K .

125.16.4 Changing the Base Ring

BaseChange(K , F)

BaseExtend(K , F)

Extends the base field of the Kummer surface K to the field F .

BaseChange(K , j)

BaseExtend(K , j)

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)

BaseExtend(K , n)

Extends the finite base field of the Kummer surface K over a finite field to the degree n extension.

125.17 Points on the Kummer Surface

Points are given by their projective coordinates, normalized depending on the base field.

125.17.1 Creation of Points

$K ! 0$

Returns the image of the identity element on the Kummer surface K , which is normalized to be the origin $(0 : 0 : 0 : 1)$.

$K ! [x_1, x_2, x_3, x_4]$

Returns the point on the Kummer surface K defined by the projective coordinates x_1, x_2, x_3 , and x_4 .

$K ! P$

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)

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 , $[x_1, x_2, x_3]$)

Returns the indexed set of points on the Kummer surface K with first three coordinates given by the sequence $[x_1, x_2, x_3]$.

125.17.2 Access Operations`P[i]`

Returns the i -th coordinate of the point P , for $1 \leq i \leq 4$.

`Eltseq(P)``ElementToSequence(P)`

Given a point P on a Kummer surface, the function returns the coordinates of P as a sequence.

125.17.3 Predicates on Points`P eq Q`

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`

Given two points on the same Kummer surface, this returns `false` if and only if the points P and Q are equal.

125.17.4 Arithmetic of Points`-P`

Returns the negation of the point P on the Kummer surface, equal to P itself.

`n * P``P * n`

Returns the n -th multiple of the point P on the Kummer surface K .

`Double(P)`

Returns the double $2 * P$ of the point P .

`PseudoAdd(P1, P2, P3)`

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)`

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 $P + n * Q$.

125.17.5 Rational Points on the Kummer Surface

<code>RationalPoints(K, Q)</code>

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 H125E42

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
  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) @}
```

125.17.6 Pullback to the Jacobian

<code>Points(J, P)</code>

<code>RationalPoints(J, P)</code>

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 .

125.18 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, $C(\mathbf{C})$, 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 $\phi_i, i = 1, 2, \dots, g$ be a basis for this vector space and set $\bar{\phi} = {}^t(\phi_1, \dots, \phi_g)$, 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 $\text{Int} : C \rightarrow \mathbf{C}^g$ by

$$P \mapsto \int_P^\infty \bar{\phi}.$$

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 $C, H_1(C, \mathbf{Z})$, to \mathbf{C}^g which sends a closed path, $\sigma \in H_1(C, \mathbf{Z})$, on the Riemann surface to $\int_\sigma \bar{\phi}$. Then Λ turns out to be a lattice in \mathbf{C}^g and so we get the complex torus \mathbf{C}^g/Λ . We then see that Int is a well-defined mapping from C into \mathbf{C}^g/Λ . 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 \mathbf{C}^g/Λ and the points of the (algebraic) Jacobian.

So far the analytic Jacobian \mathbf{C}^g/Λ is just a torus, but it can be made into an abelian manifold by choosing a polarization. Let $A_1, \dots, A_g, B_1, \dots, B_g$ 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 \mathbf{Z} -basis for the lattice $\Lambda = P\mathbf{Z}^{2g}$. 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

$$E(Px, Py) = {}^t x J y,$$

for any $x, y \in \mathbf{R}^{2g}$, then E is a Riemann form for the torus \mathbf{C}^g/Λ 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).

125.18.1 Creation and Access Functions

`AnalyticJacobian(f)`

Given $f \in \mathbf{C}[x]$ where \mathbf{C} is a complex field (see Chapter 25), 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)`

Returns the polynomial defining the hyperelliptic curve whose Jacobian is A .

`SmallPeriodMatrix(A)`

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)`

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 \mathbf{C}^g/Λ , where Λ is the \mathbf{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`.

`HomologyBasis(A)`

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 $i = 1, 2, \dots$. These closed loops form a

homology basis for the curve but probably not a symplectic basis. The third return value, \mathbf{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 $a = \mathbf{A}'\text{InfiniteRoot}$ 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)`

`Genus(A)`

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)`

`BaseRing(A)`

`CoefficientRing(A)`

The base field of the Jacobian A .

125.18.2 Maps between Jacobians

`ToAnalyticJacobian(x, y, A)`

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 $a = \mathbf{A}'\text{InfiniteRoot}$ 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 \mathbf{C}^g/Λ where Λ is the \mathbf{Z} -lattice generated by the columns of the `BigPeriodMatrix`.

`FromAnalyticJacobian(z, A)`

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 \mathbf{C}^g/Λ where Λ is the \mathbf{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 $\sum_{i=1}^g P_i - g\infty$.


```

> xpol;
x^2 - 5/4*x + 169/64
> P1+P2;
(x^2 - 5/4*x + 169/64, 125/16*x - 403/128, 2)

```

125.18.2.1 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 $A_1 = \mathbf{C}^g/\Lambda_1$ and $A_2 = \mathbf{C}^g/\Lambda_2$ with a morphism $\phi : A_1 \mapsto A_2$. This map lifts to a map $\mathbf{C}^g \mapsto \mathbf{C}^g$, 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)

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)

Given two small period matrices t_1 and t_2 , this function returns a basis for the \mathbf{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)

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)

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)

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)

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)

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)

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)

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 H125E44

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 \mathbf{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 \mathbf{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]
[ 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];
9.49857267318279326916448048991143479789137754919824276557860348643797948105949
8481543257426864218501E-101 + -1.4142135623730950488016887242096980785696718753
76948073176679737990732478462107038850387534327641573*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 \mathbf{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:A1:="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 \mathbf{Q} . In this particular case we get lucky and `alst[3]` itself is already rational:

```

> alpha := alst[3];

```



```

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) @}

```

125.18.3 From Period Matrix to Curve

RosenhainInvariants(t)

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 $y^2 = x(x - 1)(x - \lambda_1) \dots (x - \lambda_{2g-1})$ is said to be in Rosenhain normal form.

Example H125E45

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 $\mathbf{Q}(\sqrt{-2 + \sqrt{2}})$.

```

> 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, $\mathbf{C}^2/\Phi(\mathcal{O})$ is a torus with Complex Multiplication by the maximal order of our field $\mathbf{Q}(\sqrt{-2 + \sqrt{2}})$. 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*0.2; // a chosen generator of D^-1
> xi*0 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(0), a in Basis(0)]);
> D, C := FrobeniusFormAlternating(E); D;
[ 0 0 1 0]
[ 0 0 0 1]
[-1 0 0 0]
[ 0 -1 0 0]
> newb := ElementToSequence(Matrix(0,C)*Matrix(0,4,1,Basis(0)));
> SetKantPrecision(0,100);
> Abs(Re(Conjugate(xi,2))) lt 10^-10 and Im(Conjugate(xi,2)) gt 0;
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 τ 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  $y^2 = -x^5 - 3x^4 + 2x^3 + 6x^2 - 3x - 1$ 
over Rational Field

```

Let's check that this curve's analytic Jacobian is isomorphic to the one corresponding to τ .

```

> 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
[ 1 -2  0  0]
[ 0  0  2  1]
[ 1 -2  5  2]
[ 2 -5  4  2]

```

125.18.4 Voronoi Cells

This section describes two functions that are concerned with the computation of Voronoi cells.

Delaunay(sites)

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 i th sequence contains the list of indices of the sites to which site i should be joined to form the triangulation.

Voronoi(sites)

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, *sitedges*, *dualsites* and *cells* are returned. The sequence *sitedges* 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 i th sequence is a list $[i1, i2, i3, \dots, im]$ 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].

125.19 Bibliography

- [BD09] Nils Bruin and Kevin Doerksen. The arithmetic of genus two curves with (4,4)-split Jacobians. ArXiv preprint. URL:<http://arxiv.org/abs/0902.3480>, 2009.
- [Bos00] Wieb Bosma, editor. *ANTS IV*, volume 1838 of *LNCS*. Springer-Verlag, 2000.
- [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. Dissertation, University of Leiden, Leiden, 1999.
- [BS09] Nils Bruin and Michael Stoll. Two-cover descent on hyperelliptic curves. *Math. Comp.*, 78:2347–2370, 2009.
- [Car03] G. Cardona. On the number of curves of genus 2 over a finite field. *Finite Fields and Their Applications*, 9(4):505–526, 2003.
- [CF96] J.W.S. Cassels and E.V. Flynn. *Prolegomena to a middlebrow arithmetic of curves of genus 2*. Cambridge University Press, Cambridge, 1996.
- [CNP05] G. Cardona, E. Nart, and J. Pujolas. Curves of genus two over fields of even characteristic. *Mathematische Zeitschrift*, 250:177–201, 2005.
- [CQ05] 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.
- [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.
- [FS97] E.V. Flynn and N.P. Smart. Canonical heights on the Jacobians of curves of genus 2 and the infinite descent. *Acta Arith.*, 79:333 – 352, 1997.
- [GH00] P. Gaudry and R. Harley. Counting Points on Hyperelliptic Curves over Finite Fields. In Bosma [Bos00], pages 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 URL:<http://arxiv.org/abs/1004.4503>, 2006.

- [Hub06] Hendrik Hubrechts. Point counting in families of hyperelliptic curves. Preprint URL:<http://arxiv.org/abs/math.NT/0601438>, 2006.
- [Ked01] Kiran S. Kedlaya. Counting Points on Hyperelliptic Curves using Monsky-Washnitzer Cohomology. *J. Ramanujan Math. Soc.*, 16:323 – 338, 2001.
- [LL] R. Lercier and D. Lubicz. A Quasi-Quadratic Time Algorithm for Hyperelliptic Curve Point Counting. *to appear*.
- [LR12] R. Lercier and C. Ritzenthaler. Hyperelliptic curves and their invariants: geometric, arithmetic and algorithmic aspects. *Journal of Algebra*, 2012. URL:<http://dx.doi.org/10.1016/j.jalgebra.2012.07.054>.
- [Mae90] T. Maeda. On the invariant fields of binary octavics. *Hiroshima Math. J.*, 20:619–632, 1990.
- [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.*, pages 313–334. Birkhäuser, 1991. Proc. Congress in Livorno, Italy, April 17–21, 1990.
- [Mül10a] Jan Steffen Müller. *Computing canonical heights on Jacobians*. PhD Thesis, Universität Bayreuth, 2010. available at URL:<http://www.math.uni-hamburg.de/home/js.mueller/>.
- [Mül10b] Jan Steffen Müller. Explicit Kummer surface formulas for arbitrary characteristic. *LMS J. Comput. Math.*, 4:47–64, 2010.
- [Mum84] David Mumford. *Tata Lectures on Theta II*, volume 43 of *Progress in Mathematics*. Birkhäuser, 1984.
- [PS97] 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.
- [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. URL:<http://hdl.handle.net/2123/1066>.
- [Sto99] M. Stoll. On the height constant for curves of genus two. *Acta Arith.*, 90(2):183 – 201, 1999.
- [Sto01] Michael Stoll. Implementing 2-descent for Jacobians of hyperelliptic curves. *Acta Arith.*, 98(3):245–277, 2001.
- [SV01] Tony Shaska and Völklein. *Elliptic subfields and automorphisms of genus 2 function fields*. Springer, 2001. URL:<http://au.arxiv.org/abs/math.AG/0107142>.
- [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.
- [**Wam01**] P. Van Wamelen. Computing with the Jacobian of a Genus 2 Curve. URL:<http://www.math.lsu.edu/~wamelen/genus2.html>, 2001.
- [**Wet97**] J. L. Wetherell. *Bounding the number of rational points on certain curves of high rank*. PhD thesis, U.C. Berkeley, 1997.

126 HYPERGEOMETRIC MOTIVES

126.1 Introduction	4217	<code>IsPrimitive(H)</code>	4221
126.2 Functionality	4219	<i>126.2.3 Functionality with L-series and Euler Factors</i>	<i>4221</i>
<i>126.2.1 Creation Functions</i>	<i>4219</i>	<code>EulerFactor(H, t, p)</code>	4221
<code>HypergeometricData(A, B)</code>	4219	<code>LSeries(H, t)</code>	4222
<code>HypergeometricData(F, G)</code>	4219	<code>ArtinRepresentation(H, t)</code>	4223
<code>HypergeometricData(G)</code>	4219	<code>EllipticCurve(H)</code>	4223
<code>HypergeometricData(G)</code>	4219	<code>EllipticCurve(H, t)</code>	4223
<code>HypergeometricData(F, G)</code>	4219	<code>HyperellipticCurve(H)</code>	4223
<code>HypergeometricData(E)</code>	4219	<code>HyperellipticCurve(H, t)</code>	4223
<code>Twist(H)</code>	4219	<code>Identify(H, t)</code>	4223
<code>PrimitiveData(H)</code>	4220	<i>126.2.4 Associated Schemes and Curves .</i>	<i>4224</i>
<code>PossibleHypergeometricData(d)</code>	4220	<code>CanonicalScheme(H)</code>	4224
<i>126.2.2 Access Functions</i>	<i>4220</i>	<code>CanonicalScheme(H, t)</code>	4224
<code>Weight(H)</code>	4220	<code>CanonicalCurve(H)</code>	4224
<code>Degree(H)</code>	4220	<code>CanonicalCurve(H, t)</code>	4224
<code>DefiningPolynomials(H)</code>	4220	<i>126.2.5 Utility Functions</i>	<i>4224</i>
<code>CyclotomicData(H)</code>	4220	<code>HypergeometricMotiveSaveLimit(n)</code>	4224
<code>AlphaBetaData(H)</code>	4220	<code>HypergeometricMotiveClearTable()</code>	4224
<code>MValue(H)</code>	4221	126.3 Examples	4225
<code>GammaArray(H)</code>	4221	126.4 Bibliography	4233
<code>GammaList(H)</code>	4221		
<code>eq</code>	4221		
<code>ne</code>	4221		

Chapter 126

HYPERGEOMETRIC MOTIVES

126.1 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

$$z(\theta + \alpha_1) \cdots (\theta + \alpha_n)F(z) = (\theta + \beta_1 - 1) \cdots (\theta + \beta_n - 1)F(z), \quad \theta = z \frac{d}{dz},$$

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

$$z^{1-\beta_i} {}_nF_{n-1} \left(\frac{\alpha_1 - \beta_i + 1, \dots, \alpha_n - \beta_i + 1}{\beta_1 - \beta_i + 1, \dots, \beta_n - \beta_i + 1} \middle| z \right)$$

for $i = 1 \dots n$, and the i th term $\beta_i - \beta_i + 1$ is suppressed. The generalised hypergeometric function ${}_nF_{n-1}$ is given by

$${}_nF_{n-1} \left(\frac{a_1, \dots, a_n}{b_1, \dots, b_{n-1}} \middle| z \right) = \sum_{k=0}^{\infty} \frac{(a_1)_k \cdots (a_n)_k}{(b_1)_k \cdots (b_{n-1})_k} \frac{z^k}{k!},$$

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 $z \rightarrow 1/z$ 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 $M : G \rightarrow GL_n(V_{\vec{\alpha}, \vec{\beta}})$. 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 (by convention, we take β to contain the smallest such rational). 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. The Euler factor is given by the standard recipe $E_p(T) = \exp(-\sum_n U_{p^n}(t)T^n/n)$, and this is a polynomial that satisfies a local functional equation.

126.2 Functionality

126.2.1 Creation Functions

`HypergeometricData(A, B)`

`HypergeometricData(F, G)`

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. *NOTE:* the routine will always switch A and B (or f and g) so that the latter has the smallest rational in $[0, 1)$ in it.

`HypergeometricData(G)`

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. Again α and β will be reversed if necessary (which corresponds to negating all the entries of G).

`HypergeometricData(G)`

This is a fourth way to specify hypergeometric data, by giving a *list* G of nonzero integers (with repetition possible) corresponding to the sequence G of the previous intrinsic, with negative integers for those where $G[v]$ is negative. The sum of the members of the list must be 0.

`HypergeometricData(F, G)`

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)`

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)`

This intrinsic takes hypergeometric data H , and adds $1/2$ to every element in α and β , returning new hypergeometric data. Note that the new α and β may be switched due to this twisting.

PrimitiveData(H)

Given hypergeometric data H , return its primitive associated data. This is most easily described in terms of `GammaList`, where one divides all the elements by the gcd.

PossibleHypergeometricData(d)

<code>Weight</code>	RNGINTELT	<i>Default : false</i>
<code>TwistMinimal</code>	BOOLELT	<i>Default : false</i>
<code>CyclotomicData</code>	BOOLELT	<i>Default : false</i>
<code>Primitive</code>	RNGINTELT	<i>Default : 0</i>

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.

126.2.2 Access Functions**Weight(H)**

The weight of the given hypergeometric data H .

Degree(H)

The degree of the given hypergeometric data H .

DefiningPolynomials(H)

The (products of cyclotomic) polynomials corresponding to α and β corresponding to hypergeometric data H .

CyclotomicData(H)

This 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)

This returns two arrays of rationals, giving the α and β of the hypergeometric data H .

MValue(H)

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 n th 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)

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)

This returns a *list* of integers corresponding to γ_v , where $\text{sign}(\gamma_v) \cdot v$ appears in the list $|\gamma_v|$ times.

H1 eq H2**H1 ne H2**

Two instances of hypergeometric data $H1$ and $H2$ are equal if they have the same α and β .

IsPrimitive(H)

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**.

126.2.3 Functionality with L -series and Euler Factors

EulerFactor(H, t, p)

Precision	RNGINTELT	<i>Default</i> : 0
Check	BOOLELT	<i>Default</i> : false
Fake	BOOLELT	<i>Default</i> : 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 p th Euler factor of the hypergeometric motive at t . This uses p -adic Γ -functions, as indicated by Cohen. The **Precision** 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 **Precision**, 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 pseudoreflexion). 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 H126E7), 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 f th power (due to repetitions in the above extraction), and the f th 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.

<code>LSeries(H, t)</code>

<code>BadPrimes</code>	SEQENUM	<i>Default</i> : []
<code>GAMMA</code>	SEQENUM	<i>Default</i> : []
<code>Identify</code>	BOOLELT	<i>Default</i> : true

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 via `BadPrimes` and `GAMMA`). 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.

126.2.3.1 Identification of Hypergeometric Data as Other Objects

`ArtinRepresentation(H, t)`

Check

BOOLEAN

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 such H of degree 3 or less, and for some of degree 4 and higher. The condition needed is that `GammaList(H)` have cardinality 3. When **Check** is true, good primes up to 100 have their Euler factors checked for correctness.

`EllipticCurve(H)`

`EllipticCurve(H, t)`

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)`

`HyperellipticCurve(H, t)`

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)`

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).

126.2.4 Associated Schemes and Curves

`CanonicalScheme(H)`

`CanonicalScheme(H, t)`

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)`

`CanonicalCurve(H, t)`

Given suitable hypergeometric data 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`.

126.2.5 Utility Functions

`HypergeometricMotiveSaveLimit(n)`

`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 q th table entry will have $(q - 1)$ elements in it.

126.3 Examples

Example H126E1

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 [ 1/6, 1/2, 5/6 ] and [ 0, 0, 0 ]
> 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]); 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]); 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);
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 H126E2

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 [ 1/6, 5/6 ] and [ 0, 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 [ 1/3, 2/3 ] and [ 0, 1/2 ]
> A := ArtinRepresentation(T,-4/5); // can be 1/t sometimes
> 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),1/3); // 1/t here
> 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),1/t);
> Q := QuadraticTwist(T,5*t); // get right parameter
> assert IsIsomorphic(Q,S);

```

Example H126E3

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);
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 H126E4

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 H126E5

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 H127E28.

```

> H := HypergeometricData([1/2,1/2,1/2,1/2],[0,0,0,0]);
> L := LSeries(H,-1 : BadPrimes:=[<2,9,1>]); // guessed
> CheckFunctionalEquation(L);
-5.91645678915758854058796423962E-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, 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, 0, -252, 0, 0, 0 ]

```

Example H126E6

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(CheckFunctionalEquation(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(CheckFunctionalEquation(L)) lt 10^(-28);

```

Example H126E7

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 H34E24. 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); 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);
12775
> CheckFunctionalEquation(L); // takes about 40s
-3.8147E-6

```

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
> CheckFunctionalEquation(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>]);
> Conductor(L); // 2*31*331, Magma computes Euler factors
20552
> LSetPrecision(L,9); // about 1300 terms
> CheckFunctionalEquation(L);
4.65661287E-10

```

Example H126E8

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
```

```

> 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 H126E9

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 ]
[ 51, 1001, 1234, 3247, 894, 1450, 204, 363 ]
> D4w1 := PossibleHypergeometricData(4 : Weight:=1);
> D := [HypergeometricData(x) : x in D4w1];
> #[x : x in D | Twist(x) eq x]; // 12 are self-twists
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

```

126.4 Bibliography

- [**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.

127 L-FUNCTIONS

127.1 Overview	4237	<code>GammaFactors(L)</code>	4261
127.2 Built-in <i>L</i>-series	4238	<code>LSeriesData(L)</code>	4261
<code>RiemannZeta()</code>	4238	<code>Factorization(L)</code>	4262
<code>LSeries(K)</code>	4238	127.7 Precision	4263
<code>LSeries(A)</code>	4241	<code>LSetPrecision(L, precision)</code>	4263
<code>LSeries(E)</code>	4242	<i>127.7.1 L-series with Unusual Coefficient Growth</i>	<i>4264</i>
<code>LSeries(E, K)</code>	4243	<i>127.7.2 Computing $L(s)$ when $Im(s)$ is Large (ImS Parameter)</i>	<i>4264</i>
<code>LSeries(E, A)</code>	4244	<i>127.7.3 Implementation of L-series Computations (Asymptotics Parameter)</i>	<i>4264</i>
<code>LSeries(C)</code>	4245	127.8 Verbose Printing	4265
<code>LSeries(Chi)</code>	4245	127.9 Advanced Examples	4265
<code>LSeries(hmf)</code>	4246	<i>127.9.1 Handmade L-series of an Elliptic Curve</i>	<i>4265</i>
<code>LSeries(psi)</code>	4246	<i>127.9.2 Self-made Dedekind Zeta Function</i>	<i>4266</i>
<code>LSeries(psi)</code>	4246	<i>127.9.3 L-series of a Genus 2 Hyperelliptic Curve</i>	<i>4266</i>
<code>LSeries(f)</code>	4247	<i>127.9.4 Experimental Mathematics for Small Conductor</i>	<i>4268</i>
127.3 Computing <i>L</i>-values	4249	<i>127.9.5 Tensor Product of L-series Coming from <i>l</i>-adic Representations</i>	<i>4269</i>
<code>Evaluate(L, s0)</code>	4249	<i>127.9.6 Non-abelian Twist of an Elliptic Curve</i>	<i>4270</i>
<code>CentralValue(L)</code>	4249	<i>127.9.7 Other Tensor Products</i>	<i>4271</i>
<code>LStar(L, s0)</code>	4249	<i>127.9.8 Symmetric Powers</i>	<i>4273</i>
<code>LTaylor(L, s0, n)</code>	4249	<code>SymmetricPower(L, m)</code>	4274
127.4 Arithmetic with <i>L</i>-series	4251	127.10 Weil Polynomials	4276
<code>*</code>	4251	<code>SetVerbose("WeilPolynomials", v)</code>	4276
<code>/</code>	4251	<code>HasAllRootsOnUnitCircle(f)</code>	4276
<code>TensorProduct(L1, L2, ExcFactors)</code>	4251	<code>FrobeniusTracesToWeil</code>	
<code>TensorProduct(L1, L2)</code>	4251	<code>Polynomials(tr, q, i, deg)</code>	4276
<code>TensorProduct(L1, L2, ExcFactors, K)</code>	4251	<code>WeilPolynomialToRankBound(f, q)</code>	4276
<code>TensorProduct(L1, L2, K)</code>	4251	<code>ArtinTateFormula(f, q, h20)</code>	4276
127.5 General <i>L</i>-series	4252	<code>WeilPolynomialOverField</code>	
<i>127.5.1 Terminology</i>	<i>4253</i>	<code>Extension(f, deg)</code>	4276
<i>127.5.2 Constructing a General L-Series</i>	<i>4254</i>	<code>CheckWeilPolynomial(f, q, h20)</code>	4277
<code>LSeries(weight, gamma, conductor, cfun)</code>	4254	127.11 Bibliography	4279
<code>CheckFunctionalEquation(L)</code>	4256		
<i>127.5.3 Setting the Coefficients</i>	<i>4258</i>		
<code>LSetCoefficients(L, cfun)</code>	4258		
<i>127.5.4 Specifying the Coefficients Later</i>	<i>4258</i>		
<i>127.5.5 Generating the Coefficients from Local Factors</i>	<i>4260</i>		
127.6 Accessing the Invariants	4260		
<code>LCfRequired(L)</code>	4260		
<code>LGetCoefficients(L, N)</code>	4260		
<code>EulerFactor(L, p)</code>	4261		
<code>Conductor(L)</code>	4261		
<code>Sign(L)</code>	4261		

Chapter 127

L-FUNCTIONS

127.1 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.

127.2 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 $\chi : (\mathbf{Z}/m\mathbf{Z})^* \rightarrow \mathbf{C}^*$ 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 (apart from the Riemann zeta function that is not quite associated with anything) 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 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()`

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 H127E1

Check that $\zeta(2)$ agrees numerically with $\pi^2/6$.

```
> L := RiemannZeta( : Precision:=40);
> Evaluate(L,2);
1.644934066848226436472415166646025189219
> Pi(RealField(40))^2/6;
1.644934066848226436472415166646025189219
```

`LSeries(K)`

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 $\sum_I \text{Norm}_{K/\mathbf{Q}}(I)^{-s}$, where the sum is taken over the non-zero ideals I of the maximal order of K . For $K = \mathbf{Q}$, 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 H127E2

This code computes the value of $\zeta(\mathbf{Q}(i), s)$ at $s = 2$.

```
> P<x> := PolynomialRing(Integers());
> K := NumberField(x^2+1);
> L := LSeries(K);
> Evaluate(L, 2);
1.50670300992298503088656504818
```

This particular example could have been expressed somewhat more succinctly by replacing the first two lines by either `K:=CyclotomicField(4)` or `K:=QuadraticField(-1)`.

Example H127E3

The code computes $\zeta(F, 2)$ for $F = \mathbf{Q}(\sqrt[12]{3})$.

```
> R<x> := PolynomialRing(Rationals());
> F := NumberField(x^12-3);
> L := LSeries(F: Method:="Direct");
Time: 0.078
> 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: 5.234

```

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, 576, 243, 15552, 15552 ]
> time Evaluate(L, 2);
1.63925427193646882835990708818
Time: 3.250

```

Example H127E4

This code follows an example of Serre and Armitage (see [Ser71, Arm71, Fri76]) 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);
2739
> CheckFunctionalEquation(L);
1.57772181044202361082345713057E-30
> Evaluate(L, 1/2); // zero as expected
-9.98707556173617338749102627597E-62

```

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 of Number Field A with
    character ( 1, 1, 1, -1, -1 ) and conductor 41, 1>,
  <L-series of Artin representation of Number Field A with
    character ( 1, 1, -1, -1, 1 ) and conductor 5, 1>,
  <L-series of Artin representation of Number Field A with
    character ( 1, 1, -1, 1, -1 ) and conductor 205, 1>,
  <L-series of Artin representation of Number Field A with

```



```
> L := LSeries(E, CyclotomicField(11));
> time Evaluate(L, 1);
-1.03578452039312258255988860081E-28
Time: 4.797
```

LSeries(E, A)

Precision

RNGINTELT

Default :

Twisted L -series of an elliptic curve E/\mathbf{Q} by an Artin representation A . Currently does not allow E and A to be simultaneously wildly ramified at either 2 or 3.

Example H127E9

We take the elliptic curve 11A3 and twist it by the characters of $\mathbf{Q}(\zeta_5)/\mathbf{Q}$:

```
> E := EllipticCurve(CremonaDatabase(), "11A3");
> K := CyclotomicField(5);
> art := ArtinRepresentations(K);
> for A in art do Evaluate(LSeries(E,A),1); end for;
0.253841860855910684337758923351
0.685976714588516438169889514223 + 1.10993363969520543571381847366*$.1
2.83803828204429619496466743334
0.685976714588516438169889514223 - 1.10993363969520543571381847366*$.1
```

All the L -values are non-zero, so according to the Birch-Swinnerton-Dyer conjecture E has rank 0 over $\mathbf{Q}(\zeta_5)$. Indeed:

```
> #TwoSelmerGroup(BaseChange(E,K));
1
```

Example H127E10

As a higher-dimensional example, we twist $E = X_1(11)/\mathbf{Q}$ 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 of Number Field with defining polynomial
x^8 - 12*x^6 + 36*x^4 - 36*x^2 + 9 over the Rational Field with
character ( 2, -2, 0, 0, 0 )
> L:=LSeries(E,a: Precision:=10);
> LCfRequired(L);
208818
> time Evaluate(L,1);
1.678012769
```

```
Time: 23.688
> Sign(L);
1.000000002
```

LSeries(C)

Precision	RNGINTELT	<i>Default :</i>
ExcFactors	SEQENUM	<i>Default :</i> []

L -series of a hyperelliptic curve C defined over the rationals.

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 sequence of tuples $\langle \text{prime}, \text{conductor exponent}, \text{local factor} \rangle$, e.g. `ExcFactors:=⟨2, 11, 1 - x⟩`.

Example H127E11

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)
1.03140710417331775
> Sign(L); // sign in the functional equation
1.000000000000000000
```

The L -value is non-zero, indicating that the Jacobian should have rank 0. In fact, it does:

```
> RankBound(Jacobian(C));
0
```

LSeries(Chi)

Precision	RNGINTELT	<i>Default :</i>
------------------	-----------	------------------

Given a primitive dirichlet character $\chi : (\mathbf{Z}/m\mathbf{Z})^* \rightarrow \mathbf{C}^*$, 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 19.8.

Example H127E12

We define a primitive character $\chi : (\mathbf{Z}/37\mathbf{Z})^* \rightarrow \mathbf{C}^*$ 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 - 0.551607898922910805875537715934*$.1
```

LSeries(hmf)

Given a cuspidal newform in a space of Hilbert modular forms, this creates the associated L -series.

Example H127E13

```
> K := QuadraticField(5);
> H := HilbertCuspForms(K, 7*Integers(K), [2,2]);
> f := NewformDecomposition(NewSubspace(H))[1];
> L := LSeries(Eigenform(f));
> LSetPrecision(L,9);
> LCfRequired(L);
198
> time CheckFunctionalEquation(L);
-2.32830644E-10
Time: 16.590
```

LSeries(psi)

LSeries(psi)

Precision

RNGINTELT

Default :

Given a primitive Hecke (Grössen)character on ideals, construct the associated L -series.

For more information on these see Section 34.9.


```
0.359306437003505684066327207778 - 0.0714704939991172686588458066910*$.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);
```

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 far from 1, wrong functional equation?
```

```
0.736718188651826073550560964422
```

```
> CheckFunctionalEquation(L);
```

```
0.00814338037134482026061721221244
```

The function `CheckFunctionalEquation` 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 127.5.1. Here are some examples of modular forms that do.

```
> CheckFunctionalEquation(LSeries(f1^2*f2));
```

```
1.57772181044202361082345713057E-30
```

```
> f3 := ModularForm(EllipticCurve([0, -1, 1, 0, 0]));
```

```
> CheckFunctionalEquation(LSeries(f3));
```

```
0.00000000000000000000000000000000
```

```
> M := Newforms("37k2");
```

```
> f4 := M[1,1]; f5 := M[2,1]; f6 := M[3,1];
```

```
> CheckFunctionalEquation(LSeries((f5+2*f6)*f4));
```

```
5.91645678915758854058796423962E-31
```

127.3 Computing L -values

Once an L -series $L(s)$ has been constructed using either a standard zeta- or L -function (Section 127.2), a user defined L -function (Section 127.5.2) or constructed from other L -functions (Section 44.4), 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)

Derivative	RNGINTELT	<i>Default : 0</i>
Leading	BOOLELT	<i>Default : false</i>

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,

$$L(s_0) = L'(s_0) = \dots = L^{(D-1)}(s_0) = 0,$$

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)

Given an L -function of even weight $2k$ (in the MAGMA sense), the value of L is computed at $s = k$.

LStar(L, s0)

Derivative	RNGINTELT	<i>Default : 0</i>
-------------------	-----------	--------------------

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 127.5.1)

$$L^*(s) = \text{sign} \cdot \bar{L}^*(\text{weight} - s)$$

(cf. Section 127.5.1).

LTaylor(L, s0, n)

ZeroBelow	RNGINTELT	<i>Default : 0</i>
------------------	-----------	--------------------

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:

$$L(s_0) + L'(s_0)x + L''(s_0)x^2/2! + \dots + L^{(n)}(s_0)x^n/n! + O(x^{n+1}).$$

If the first few terms $L(s_0), \dots, L^{(k)}(s_0)$ of this expansion are known to be zero, the computation time can be reduced by setting **ZeroBelow:=k + 1**.

Example H127E15

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);
0.0000000000000000
> Evaluate(L, 1 : Derivative:=1, Leading:=true);
-1.69522909186553E-17
> Evaluate(L, 1 : Derivative:=2, Leading:=true);
6.27623031179552E-17
> 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 rational elliptic curve 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 + 2.93970849657696*$.1^5
Time: 11.070
> time LTaylor(L, 1, 5);
-1.69522909186553E-17*$.1 + 3.13811515589776E-17*$.1^2 +
1.73184990011930*$.1^3 - 3.20590558844390*$.1^4 + 2.93970849657696*$.1^5
Time: 20.320
```

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
```

127.4 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.

L1 * L2

Poles	SEQENUM	<i>Default</i> : []
Residues	SEQENUM	<i>Default</i> : []
Precision	RNGINTELT	<i>Default</i> :

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 127.5.1 for the terminology and Section 127.5.2 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

Poles	SEQENUM	<i>Default</i> : []
Residues	SEQENUM	<i>Default</i> : []
Precision	RNGINTELT	<i>Default</i> :

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 127.5.1 for the terminology and Section 127.5.2 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 that of the default real field.

TensorProduct(L1, L2, ExcFactors)

TensorProduct(L1, L2)

TensorProduct(L1, L2, ExcFactors, K)

TensorProduct(L1, L2, K)

Precision	RNGINTELT	<i>Default :</i>
Sign	FLDCOMELT	<i>Default :</i>

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 H127E24, H127E25).

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** 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 that of the default real field.

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.

See Examples H127E18 and H127E24, H127E25, and Section 127.9.7.

127.5 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.

127.5.1 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 $\operatorname{Re}(s)$ 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*

$$\gamma(s) = \Gamma\left(\frac{s+\lambda_1}{2}\right) \cdots \Gamma\left(\frac{s+\lambda_d}{2}\right)$$

of *dimension* $d \geq 1$ and rational γ -*shifts* $\lambda_1, \dots, \lambda_d$, such that

$$L^*(s) = \left(\frac{\text{conductor}}{\pi^d}\right)^{s/2} \gamma(s) L(s)$$

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 $L(s) = \sum_{n=1}^{\infty} \bar{a}_n/n^s$. 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.

127.5.2 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).

<code>LSeries(weight, gamma, conductor, cffun)</code>		
---	--	--

<code>Sign</code>	<code>FLDCOMELT</code>	<i>Default</i> : 0
<code>Poles</code>	<code>SEQENUM</code>	<i>Default</i> : []
<code>Residues</code>	<code>SEQENUM</code>	<i>Default</i> : []
<code>Parent</code>	<code>ANY</code>	<i>Default</i> :
<code>CoefficientGrowth</code>	<code>USERPROGRAM</code>	<i>Default</i> :
<code>Precision</code>	<code>RNGINTELT</code>	<i>Default</i> :
<code>ImS</code>	<code>FLDREELT</code>	<i>Default</i> : 0
<code>Asymptotics</code>	<code>BOOLELT</code>	<i>Default</i> : 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 127.5.1 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 $s \rightarrow w - s$ 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 \mathbf{Q} and $[0, \dots, 0, 1, \dots, 1]$ 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,

$|\text{Discriminant}(K)|$ for the Dedekind zeta of a number field K and the conductor of the Jacobian for an algebraic curve over the rationals.

cf fun — specifies the coefficients a_n . The following possibilities are allowed:

- A finite sequence $[a_1, \dots, a_n]$;
- 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 127.5.3 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 $\text{Re } z \geq \text{weight}/2$ 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 $s = \text{weight}/2$ 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 127.7.

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]);
> CheckFunctionalEquation(Z);
0.00000000000000000000000000000000
```

For more examples, see the “Advanced Examples” Section 127.9.

It is strongly advised that the user apply the function **CheckFunctionalEquation** whenever a generic L -series is defined. 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)

t

FLDREELT

Default : 1.2

Given an L -series L , this function tests the functional equation numerically and should ideally return 0 (to the current precision), meaning that the test was passed. 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, **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 127.5.1. 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 $|\text{Sign}| - 1$ 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 **CheckFunctionalEquation**. The parameter

t must be a real number satisfying $1.05 < t < 1.2$ and for every such number the function should return 0, provided that the functional equation is indeed correct.

Example H127E16

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);
> CheckFunctionEquation(L);
1.152455615560373697496605481547
```

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);
> CheckFunctionEquation(L);
1.063726235879220898712968226243
```

This still does not look right. We gave the wrong sign in the functional equation.

```
> L := LSeries(1, [1], 3, [1,-1,0,1,-1,0] : Sign:=1);
> CheckFunctionEquation(L);
3.328171077588057787282583863548E-15
```

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);
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);
> CheckFunctionEquation(L);
9.860761315262647567646607066035E-32
```

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);
> CheckFunctionEquation(L);
9.860761315262647567646607066035E-32
```

This allows MAGMA to calculate as many a_n as it deems necessary using the provided function.

127.5.3 Setting the Coefficients

LSetCoefficients(L, cffun)

This function defines the coefficients a_n of the L -series L . The argument $cffun$ can be one of the following:

- A sequence $[a_1, \dots, a_n]$;
- 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` parameter 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`.

127.5.4 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 44.3).

Example H127E17

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 $[a_1, \dots, a_N]$.

```
> 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 `CheckFunctionalEquation` to get an indication of the resulting accuracy

```
> LSetCoefficients(L, [1,1]);
> CheckFunctionalEquation(L);
4.948762810534411372665820496508E-9
> Evaluate(L, 2);
1.64493406684811250127872978182
> $1 - Pi(RealField(28))^2/6;
-1.139351936853848646746774340E-13
```

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.

127.5.5 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+0(x^(d+1))> :
    Poles:=[1], Residues:=[-1], Sign:=1);
```

127.6 Accessing the Invariants

`LCfRequired(L)`

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 127.5.4.

`LGetCoefficients(L, N)`

Compute the vector of first N coefficients `[* a1, ..., aN *]` of the L -series given by L .

EulerFactor(L, p)

Given an L -series and a prime p , this computes the p th 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.

Conductor(L)

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)

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 `CheckFunctionalEquation(L)` or any evaluation function sets the sign.)

GammaFactors(L)

A sequence of Gamma factors $\lambda_1, \dots, \lambda_d$ for $L(s)$. Each one represents a factor $\Gamma((s + \lambda_i)/2)$ entering the functional equation of the L -function.

LSeriesData(L)

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 127.5.2).

Example H127E18

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 + 0(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, ... >
```

The conductor of this L -series (second entry) is very large 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); // a lot!
280632
```

Decreasing the precision will help somewhat.

```
> LSetPrecision(LEK,9);
> LCfRequired(LEK);
17508
```

And realizing that our L -series has a piece that can be factored out will certainly help (see the Arithmetic section).

```
> Q := LEK/LE;
> LSetPrecision(Q,9);
> LCfRequired(Q);
3780
> time Evaluate(Q,2) * Evaluate(LE,2); // = L(E/K,2)
0.892165947
Time: 8.020
```

Factorization(L)

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 $[\dots \langle L_i, n_i \rangle \dots]$.

Example H127E19

```
> 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 of Number Field with
  defining polynomial x^6 + 108 over the Rational Field
  with character ( 1, -1, 1 ) and conductor 3, 1>,
<L-series of Artin representation of Number Field with
  defining polynomial x^6 + 108 over the Rational Field
  with character ( 2, 0, -1 ) and conductor 108, 2>
]

```

127.7 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);
1.28519095548414940291751179869957460396917839702892124395133

```

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
> Rank(E);
3

```

This example checks numerically that $L'(E, 1) = 0$ for the elliptic curve E of conductor 5077 from Example H127E15.

The precision may be changed at a later time using `LSetPrecision`.

<code>LSetPrecision(L,precision)</code>

<code>CoefficientGrowth</code>	<code>USERPROGRAM</code>	<i>Default :</i>
<code>ImS</code>	<code>FLDREELT</code>	<i>Default : 0</i>
<code>Asymptotics</code>	<code>BOOLELT</code>	<i>Default : true</i>

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 127.7.1.

127.7.1 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.

127.7.2 Computing $L(s)$ when $\text{Im}(s)$ 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 $\text{Im}(s)$ 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
0.793044952561928671982128851045 - 1.04127461465106502007452195086*i
> LSetPrecision(L, 30: ImS:=40);
> Evaluate(L, 1/2+40*i);           // right
0.793044952561928671964892588898 - 1.04127461465106502005189059539*i
```

127.7.3 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.

127.8 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_coefs and L'vprint_series of a variable L of type LSeries and may be changed at the user's desire.)

127.9 Advanced Examples

127.9.1 Handmade L -series of an Elliptic Curve

Example H127E20

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);
0.977431866894500508679039127647
> Evaluate(LSeries(E),2);
0.977431866894500508679039127647
```

127.9.2 Self-made Dedekind Zeta Function

Example H127E21

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 $1 - x^{f_i}$ where the f_i are the residue degrees of primes above p . Note that this local factor has (maximal) degree $[K : \mathbf{Q}]$ 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);
L*(s) has a pole at s = 1
> L := Z/RiemannZeta();
> Evaluate(L,1);
0.339837278240523535464278781159
```

127.9.3 L -series of a Genus 2 Hyperelliptic Curve

Example H127E22

We compute 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 $H^1(C)$ 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 $\text{EulerFactor}(J, \text{GF}(p, 1))$. 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.)

```
> CheckFunctionalEquation(L);
-2.958228394578794270293982119810E-31
> Evaluate(L,1);
0.0904903908324296291135897572580
> Evaluate(L,2);
0.364286342944364068154291450139
```

127.9.4 Experimental Mathematics for Small Conductor

Example H127E23

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 `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 $[a_1, a_2, 0, 0, 0, 0, \dots]$ 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, CheckFunctionalEquation(L);
> end for;
-2 0.008448098707478090187579588380635
-1 0.07557498328782515608604001309880
0 0.1427018678681722219845004378169
1 0.2098287524485192878829608625350
2 0.2769556370288663537814212872532
```

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, \dots, 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(CheckFunctionalEquation(L));
> end function;
> for p := 2 to 20 do // try a_p in Hasse-Weil range and find best one
>   if IsPrime(p) then
>     hasse := Floor(2*Sqrt(p));
>     _,V[p] := Min([Try(V,p, a_p): a_p in [-hasse..hasse]]);
```

```

> V[p] -= hasse+1;
> end if;
> 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*]
> CheckFunctionalEquation(L);
4.186579790674123374418985668816E-16

```

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 + 0(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...10 and sign = 1 to sign = ± 1) one can show that for $N < 11$ there are no modular elliptic curves of conductor N .

127.9.5 Tensor Product of L -series Coming from l -adic Representations

Example H127E24

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;
> CheckFunctionalEquation(L);
|Sign| is far from 1, wrong functional equation?

```

0.0234062571006075114301535636401

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>
> loc, model:=LocalInformation(EK,p);loc,model;
<Principal Prime Ideal
Generator:
  [0, -1, 0], 0, 0, 1, IO>
Elliptic Curve defined by  $y^2 - y = x^3$  over  $K$ 
```

We see that E has acquired good reduction at $p|2$. This means that the inertia invariants at 2 of the l -adic representation associated to L , namely $\rho_E \otimes (\rho_K - \rho_{\mathbf{Q}})$ 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 $E/K \bmod p$, 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;
> CheckFunctionalEquation(L);
3.15544362088404722164691426113E-30
```

In fact, our L -series is the same as that constructed by `LSeries(E,K)/LE`.

127.9.6 Non-abelian Twist of an Elliptic Curve

Example H127E25

In this example we illustrate how to use `LSeries(E,K)` to construct non-abelian twists of elliptic curves.

```
> 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;
```


Example H127E26

A tensor product of two elliptic curves over \mathbf{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, [ 0, 1, -1, 0 ], 34969, ... >
> CheckFunctionalEquation(L);
-2.83989925879564249948222283502E-29
```

Example H127E27

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, []);
> LSeriesData(L); // weight is (12-1)+(26-1)+1 -- motivic weight is 11+25
<37, [ 0, 1, -11, -10 ], 1, ... >
> CheckFunctionalEquation(L);
-5.75014913311889483177106362349E-25
> Pi(RealField(30))^2 * Evaluate(L,24) / Evaluate(L,25);
9.87142857142857142857142857136
> 691/70.; // Ramanujan congruence
9.87142857142857142857142857142
```

Example H127E28

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)
> time CheckFunctionalEquation(L);
```

```
-3.15544362088404722164691426113E-30
Time: 2.090
```

Example H127E29

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);
1642
> CheckFunctionalEquation(TP);
4.65661287E-10
```

127.9.8 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 k th symmetric power is simply the L -function associated to the k th power of the underlying character, though we must be careful to ensure primitivity (this disregards the bad primes). Explicitly, we have

$$L(\mathrm{Sym}^k \psi, s) = \prod_{\mathfrak{p}} \left(1 - \psi(\mathfrak{p})^k / N\mathfrak{p}^s\right)^{-1},$$

(again ignoring bad primes) and so the eigenvalues are just the k th 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 k th symmetric power is an L -function of degree $(k + 1)$ over the field of definition, given by

$$L(\mathrm{Sym}^k A, s) = \prod_{\mathfrak{p}} \prod_{i=0}^k \left(1 - \alpha_1(\mathfrak{p})^{k-i} \alpha_2(\mathfrak{p})^i / N\mathfrak{p}^s\right)^{-1},$$

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 k th 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.

SymmetricPower(L, m)

BadEulerFactors

SEQENUM

Default : []

Return the L -series corresponding to the m th symmetric power of L . The BadEulerFactors 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 H127E30

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; // Random now gives an imprimitive character
> L := LSeries(chi);
> LS3 := SymmetricPower(L, 3);
> Lc3 := LSeries(chi^3);
> Evaluate(LS3, 1);
0.843964498053507794372984784470 + 0.199232992137116783033645803753*i
> Evaluate(Lc3, 1);
0.843964498053507794372984784470 + 0.199232992137116783033645803753*i
```

Example H127E31

```
> K := QuadraticField(-23);
> I := Factorization(23 * IntegerRing(K))[1][1];
> G := HeckeCharacterGroup(I);
> psi := G.1^14; // Random now gives an imprimitive character
> L := LSeries(psi);
> LS5 := SymmetricPower(L, 5);
> Lp5 := LSeries(psi^5);
> Evaluate(LS5, 1);
0.870801884824381647583631149814 + 0.622589291563954831229726530813*i
> Evaluate(Lp5, 1);
0.870801884824381647583631149814 + 0.622589291563954831229726530813*i
```

Example H127E32

In this example we take symmetric powers of the L -function of a Grossencharacter.

```
> GR := Grossencharacter(psi, [[1,0]]);
> L := LSeries(GR);
> LS4 := SymmetricPower(L, 4);
> Lp4 := LSeries(GR^4);
> Evaluate(LS4, 3);
0.354651275716915313430268098042 - 0.147519877277453173656613056548*i
> Evaluate(Lp4, 3);
0.354651275716915313430268098042 - 0.147519877277453173656613056548*i
```

Example H127E33

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 ], 389^2, function(p, d) ... end function, 1, [], []>
> LSetPrecision(L2, 9);
> Evaluate(L2, 2);
3.17231145
> ($1 * Conductor(E)) / (2 * Pi(RealField())) * FundamentalVolume(E);
40.0000000
> ModularDegree(E);
40
```

Example H127E34

This is an example where a higher symmetric power has a vanishing central value, here a zero of (presumably) order 4. It comes from work of Buhler, Schoen, and Top [BST97].

```
> E := EllipticCurve("73a");
> L := LSeries(E);
> L3 := SymmetricPower(L, 3);
> LSeriesData(L3); // Magma knows the Sign is +1
<4, [ 0, -1, 1, 0 ], 73^3, function(p, d) ... end function, 1, [], []>
> LSetPrecision(L3, 9);
> CentralValue(L3);
-2.16367048E-12
```

127.10 Weil Polynomials

The characteristic polynomial of the Frobenius on the étale cohomology is called a Weil polynomial. This section contains some auxiliary routines for simpler handling of these polynomials. All routines make frequently use of `PowerSumToCoefficients`, `CoefficientsToElementarySymmetric` and `ElementarySymmetricToPowerSums`.

`SetVerbose("WeilPolynomials", v)`

Set the verbose printing level for the analysis of weil polynomials. Maximal value 2.

`HasAllRootsOnUnitCircle(f)`

Given a polynomial f with rational coefficients this routine checks that the complex roots of the polynomial are all of absolute value 1. The algorithm does not use floating point approximations.

`FrobeniusTracesToWeilPolynomials(tr, q, i, deg)`

<code>KnownFactor</code>	<code>RNGUPOLELT</code>	<i>Default : 1</i>
<code>Verbose</code>	<code>WeilPolynomials</code>	<i>Maximum : 2</i>

Given the sequence of Frobenius traces on the i -th étale cohomology this function returns a list of possible characteristic polynomials. Here q is the order of the basefield and deg is the degree of the characteristic polynomial. In the case of even i the sign in the functional equation has to be determined. This is done by checking the absolute values of the roots of the hypothetical polynomials. In the case that this does not work both candidates are returned. In the case of contradictory data an empty sequence is returned.

If a factor of the characteristic polynomial is known it can be given as optional argument.

`WeilPolynomialToRankBound(f, q)`

Given a polynomial f this routine counts the zeros (with multiplicity) of f that are q times a root of unity. This is an upper bound for the Picard rank of the corresponding algebraic surface; according to the Tate conjecture, it is precisely the Picard rank.

`ArtinTateFormula(f, q, h20)`

Given a Weil polynomial corresponding to H^2 of an algebraic surface with Hodge number $h^{2,0}$ this routine evaluates the Artin-Tate formula. If the Tate conjecture holds, the two values returned are the arithmetic Picard rank, and the absolute value of the discriminant of the Picard group times the order of the Brauer group of the surface.

`WeilPolynomialOverFieldExtension(f, deg)`

Given the characteristic polynomial f of the Frobenius on an étale cohomology group. This routine returns the characteristic polynomial of the deg times iterated Frobenius.

CheckWeilPolynomial(f, q, h20)

SurfDeg

RNGINTELT

Default : -1

Given a polynomial f this routine checks several conditions that must be satisfied by the characteristic polynomial of the Frobenius on H^2 of an algebraic surface. Here q is the size of the basefield, $h20$ is the Hodge number $h^{2,0}$ and **SurfDeg** is the degree of the surface (-1 means degree unknown). For example, this routine can be used to determine the sign in the functional equation.

The routine checks the valuation of the roots at all places (including p and ∞). It also checks the functional equation and the Artin-Tate conditions (see [EJ10] for details).

Example H127E35

```
> q1<t> := PolynomialRing(RationalField());
> z4<x,y,z,w> := PolynomialRing(IntegerRing(),4);
> f := x^4 + x^3*z + x^2*y^2 + x^2*y*w + x^2*z^2 + x^2*z*w +
>     x*y^3 + x*y*z*w + x*y*w^2 + x*z^2*w + y^3*w + y^2*z^2 +
>     y^2*z*w + y^2*w^2 + y*z^2*w + z^4 + z*w^3;
> Tr := [];
> for i := 1 to 10 do
>   P3 := ProjectiveSpace(GF(2^i),3);
>   S := Scheme(P3,f);
>   time
>   Tr[i] := #Points(S) - 1 - 2^(2*i);
> end for;
> cpl := FrobeniusTracesToWeilPolynomials(Tr, 2, 2, 22: KnownFactor := t-2);
> cpl;
[
  t^22 - t^21 - 2*t^20 - 4*t^19 + 16*t^17 + 32*t^16 - 64*t^15
    + 16384*t^7 - 32768*t^6 - 65536*t^5 + 262144*t^3
    + 524288*t^2 + 1048576*t - 4194304,
  t^22 - t^21 - 2*t^20 - 4*t^19 + 16*t^17 + 32*t^16 - 64*t^15
    - 16384*t^7 + 32768*t^6 + 65536*t^5 - 262144*t^3
    - 524288*t^2 - 1048576*t + 4194304
]
```

Here the sign in the functional equation can not be determined by the absolute values of roots test. Further point counting would be very slow, as the coefficients of t^{11}, t^{10}, t^9, t^8 are zero, thus point counting over $\mathbf{F}_{2^{15}}$ would be necessary to determine the sign in this way. Instead, we can try to use more invariants of the surface.

```
> cpl_2 := [wp : wp in cpl | CheckWeilPolynomial(wp,2,1: SurfDeg := 4)];
> cpl_2;
[
  t^22 - t^21 - 2*t^20 - 4*t^19 + 16*t^17 + 32*t^16 - 64*t^15
    + 16384*t^7 - 32768*t^6 - 65536*t^5 + 262144*t^3
    + 524288*t^2 + 1048576*t - 4194304
]
```

```

]
> WeilPolynomialToRankBound(cpl_2[1],2);
2
> ArtinTateFormula(cpl_2[1],2,1);
1 4
> ArtinTateFormula(WeilPolynomialOverFieldExtension(cpl_2[1],2),2^2,1);
2 68

```

Thus the minus sign in the functional equation is correct. The surface has arithmetic Picard rank 1 and relative to the Tate conjecture the geometric Picard group has rank 2 and discriminant either -17 or -68 (depending on the order of the Brauer group).

Advanced example: Bounding Picard rank with less point counting:

The computation is based on the same equation as the example above. Now we do not use the most costly Frobenius trace, but we assume additional cyclotomic factors.

```

> Tr := [ 1, 5, 19, 33, 11, -55, 435, -31, -197]; /* First 9 Frobenius traces */
> cycl := [ Evaluate(CyclotomicPolynomial(i),t/2)*2^EulerPhi(i)
>           : i in [1..100] | EulerPhi(i) lt 23];
> cycl[1] := cycl[1]^2;
> cycl[2] := cycl[2]^2; /* linear factors need multiplicity two */
> hyp := &cat [FrobeniusTracesToWeilPolynomials(Tr, 2, 2, 22 :
>           KnownFactor := (t-2)*add) : add in cycl];
> hyp := SetToSequence(Set(hyp));
> hyp;
[
  t^22 - t^21 - 2*t^20 - 4*t^19 + 16*t^17 + 32*t^16 - 64*t^15
    - 16384*t^7 + 32768*t^6 + 65536*t^5 - 262144*t^3
    - 524288*t^2 - 1048576*t + 4194304
]

```

Using only the number of point over $\mathbf{F}_2, \dots, \mathbf{F}_{2^9}$ we can prove that either the Picard rank is bounded by 2 or the correct Weil polynomial is in the list.

```

> CheckWeilPolynomial(hyp[1],2,1:SurfDeg := 4);
false

```

The invariants of the surface and the hypothetical Weil polynomial are contradictory. Thus the Picard rank is at most 2.

127.11 Bibliography

- [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.
- [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.
- [Dok02] Tim Dokchitser. ComputeL, pari package to compute motivic L -functions. URL:<http://www.maths.dur.ac.uk/~dma0td/computel/>, 2002.
- [Dok04] Tim Dokchitser. Computing special values of motivic L -functions. *Experiment. Math.*, 13(2):137–149, 2004.
- [EJ10] Andreas-Stephan Elsenhans and Jörg Jahnel. Weil polynomials of K3 surfaces. In *Algorithmic number theory*, volume 6197 of *Lecture Notes in Computer Science*, pages 126–141, Berlin, 2010. Springer.
- [Fri76] J. B. Friedlander. On the class numbers of certain quadratic extensions. *Acta Arith.*, 28(4):391–393, 1975/76.
- [HPP06] F. Hess, S. Pauli, and M. Pohst, editors. *ANTS VII*, volume 4076 of *LNCS*. Springer-Verlag, 2006.
- [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.
- [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 Hess et al. [HPP06], pages 377–392.
- [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.
- [Ser71] J.-P. Serre. Conducteurs d’Artin des caractères réels. *Invent. Math.*, 14(3): 173–183, 1971.
- [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.