Elliptic curves

Release 10.4.rc1

The Sage Development Team

Jun 27, 2024

CONTENTS

1 Elliptic curve constructor 1

2 Construct elliptic curves as Jacobians 17

3 Points on elliptic curves 21

4 Elliptic curves over a general ring 59

5 Elliptic curves over a general field 91

6 Elliptic curves over finite fields 121

7 Formal groups of elliptic curves 153

8 Elliptic-curve morphisms 159

9 Composite morphisms of elliptic curves 173

10 Sums of morphisms of elliptic curves 181

11 Isomorphisms between Weierstrass models of elliptic curves 187

12 Isogenies 193

13 Square-root Vélu algorithm for elliptic-curve isogenies 215

14 Scalar-multiplication morphisms of elliptic curves 223

15 Frobenius isogenies of elliptic curves 229

16 Isogenies of small prime degree 235

17 Modular polynomials for elliptic curves 267

18 Elliptic curves over number fields 269

19 To be sorted 627

20 Hyperelliptic curves 641

21 Indices and Tables 743

Python Module Index 745

Index 747

CHAPTER

ONE

ELLIPTIC CURVE CONSTRUCTOR

AUTHORS:

- William Stein (2005): Initial version
- John Cremona (2008-01): EllipticCurve(j) fixed for all cases

class sage.schemes.elliptic_curves.constructor.**EllipticCurveFactory**

Bases: UniqueFactory

Construct an elliptic curve.

In Sage, an elliptic curve is always specified by (the coefficients of) a long Weierstrass equation

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

INPUT:

There are several ways to construct an elliptic curve:

- `EllipticCurve([a1,a2,a3,a4,a6])`: Elliptic curve with given a-invariants. The invariants are coerced into a common parent. If all are integers, they are coerced into the rational numbers.
- `EllipticCurve([a4,a6])`: Same as above, but $a_1 = a_2 = a_3 = 0$.
- `EllipticCurve(label)`: Returns the elliptic curve over $\mathbf{Q}$ from the Cremona database with the given label. The label is a string, such as `"11a"` or `"37b2"`. The letters in the label *must* be lower case (Cremona's new labeling).
- `EllipticCurve(R, [a1,a2,a3,a4,a6])`: Create the elliptic curve over R with given a-invariants. Here R can be an arbitrary commutative ring, although most functionality is only implemented over fields.
- `EllipticCurve(j=j0)` or `EllipticCurve_from_j(j0)`: Return an elliptic curve with j-invariant `j0`.
- `EllipticCurve(polynomial)`: Read off the a-invariants from the polynomial coefficients, see *EllipticCurve_from_Weierstrass_polynomial()*.
- `EllipticCurve(cubic, point)`: The elliptic curve defined by a plane cubic (homogeneous polynomial in three variables), with a rational point.

Instead of giving the coefficients as a *list* of length 2 or 5, one can also give a *tuple*.

EXAMPLES:

We illustrate creating elliptic curves:

```
sage: EllipticCurve([0,0,1,-1,0])
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
```

We create a curve from a Cremona label:

```
sage: EllipticCurve('37b2')
Elliptic Curve defined by y^2 + y = x^3 + x^2 - 1873*x - 31833 over Rational Field
sage: EllipticCurve('5077a')
Elliptic Curve defined by y^2 + y = x^3 - 7*x + 6 over Rational Field
sage: EllipticCurve('389a')
Elliptic Curve defined by y^2 + y = x^3 + x^2 - 2*x over Rational Field
```

Old Cremona labels are allowed:

```
sage: EllipticCurve('2400FF')
Elliptic Curve defined by y^2 = x^3 + x^2 + 2*x + 8 over Rational Field
```

Unicode labels are allowed:

```
sage: EllipticCurve(u'389a')
Elliptic Curve defined by y^2 + y = x^3 + x^2 - 2*x over Rational Field
```

We create curves over a finite field as follows:

```
sage: EllipticCurve([GF(5)(0),0,1,-1,0])
Elliptic Curve defined by y^2 + y = x^3 + 4*x over Finite Field of size 5
sage: EllipticCurve(GF(5), [0, 0,1,-1,0])
Elliptic Curve defined by y^2 + y = x^3 + 4*x over Finite Field of size 5
```

Elliptic curves over $\mathbf{Z}/N\mathbf{Z}$ with N prime are of type "elliptic curve over a finite field":

```
sage: F = Zmod(101)
sage: EllipticCurve(F, [2, 3])
Elliptic Curve defined by y^2 = x^3 + 2*x + 3 over Ring of integers modulo 101
sage: E = EllipticCurve([F(2), F(3)])
sage: type(E)
<class 'sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field_
↪with_category'>
sage: E.category()
Category of abelian varieties over Ring of integers modulo 101
```

In contrast, elliptic curves over $\mathbf{Z}/N\mathbf{Z}$ with N composite are of type "generic elliptic curve":

```
sage: F = Zmod(95)
sage: EllipticCurve(F, [2, 3])
Elliptic Curve defined by y^2 = x^3 + 2*x + 3 over Ring of integers modulo 95
sage: E = EllipticCurve([F(2), F(3)])
sage: type(E)
<class 'sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic_with_
↪category'>
sage: E.category()
Category of schemes over Ring of integers modulo 95
```

The following is a curve over the complex numbers:

```
sage: E = EllipticCurve(CC, [0,0,1,-1,0])
sage: E
Elliptic Curve defined by y^2 + 1.00000000000000*y = x^3 + (-1.00000000000000)*x
 over Complex Field with 53 bits of precision
sage: E.j_invariant()
2988.97297297297
```

We can also create elliptic curves by giving the Weierstrass equation:

```
sage: R2.<x,y> = PolynomialRing(QQ,2)
sage: EllipticCurve(y^2 + y - ( x^3 + x - 9 ))
Elliptic Curve defined by y^2 + y = x^3 + x - 9 over Rational Field

sage: R.<x,y> = GF(5)[]
sage: EllipticCurve(x^3 + x^2 + 2 - y^2 - y*x)
Elliptic Curve defined by y^2 + x*y  = x^3 + x^2 + 2 over Finite Field of size 5
```

We can also create elliptic curves by giving a smooth plane cubic with a rational point:

```
sage: R3.<x,y,z> = PolynomialRing(QQ,3)
sage: F = x^3 + y^3 + 30*z^3
sage: P = [1,-1,0]
sage: EllipticCurve(F,P)
Elliptic Curve defined by y^2 - 270*y = x^3 - 24300 over Rational Field
```

We can explicitly specify the j-invariant:

```
sage: E = EllipticCurve(j=1728); E; E.j_invariant(); E.label()
Elliptic Curve defined by y^2 = x^3 - x over Rational Field
1728
'32a2'

sage: E = EllipticCurve(j=GF(5)(2)); E; E.j_invariant()
Elliptic Curve defined by y^2 = x^3 + x + 1 over Finite Field of size 5
2
```

See Issue #6657

```
sage: EllipticCurve(GF(144169), j=1728)                                    #␣
↪needs sage.rings.finite_rings
Elliptic Curve defined by y^2 = x^3 + x over Finite Field of size 144169
```

Elliptic curves over the same ring with the same Weierstrass coefficients are identical, even when they are constructed in different ways (see Issue #11474):

```
sage: EllipticCurve('11a3') is EllipticCurve(QQ, [0, -1, 1, 0, 0])
True
```

By default, when a rational value of j is given, the constructed curve is a minimal twist (minimal conductor for curves with that j-invariant). This can be changed by setting the optional parameter `minimal_twist`, which is True by default, to False:

```
sage: EllipticCurve(j=100)
Elliptic Curve defined by y^2 = x^3 + x^2 + 3392*x + 307888 over Rational Field
sage: E =EllipticCurve(j=100); E
Elliptic Curve defined by y^2 = x^3 + x^2 + 3392*x + 307888 over Rational Field
sage: E.conductor()
33129800
sage: E.j_invariant()
100
sage: E =EllipticCurve(j=100, minimal_twist=False); E
Elliptic Curve defined by y^2 = x^3 + 488400*x - 530076800 over Rational Field
sage: E.conductor()
298168200
```

(continues on next page)

(continued from previous page)

```
sage: E.j_invariant()
100
```

Without this option, constructing the curve could take a long time since both j and $j - 1728$ have to be factored to compute the minimal twist (see Issue #13100):

```
sage: E = EllipticCurve_from_j(2^256+1, minimal_twist=False)
sage: E.j_invariant() == 2^256+1
True
```

create_key_and_extra_args (*x=None*, *y=None*, *j=None*, *minimal_twist=True*, ***kwds*)

Return a `UniqueFactory` key and possibly extra parameters.

INPUT: See the documentation for *EllipticCurveFactory*.

OUTPUT:

A pair `(key, extra_args)`:

- `key` has the form $(R, (a_1, a_2, a_3, a_4, a_6))$, representing a ring and the Weierstrass coefficients of an elliptic curve over that ring;
- `extra_args` is a dictionary containing additional data to be inserted into the elliptic curve structure.

EXAMPLES:

```
sage: EllipticCurve.create_key_and_extra_args(j=8000)
((Rational Field, (0, 1, 0, -3, 1)), {})
```

When constructing a curve over **Q** from a Cremona or LMFDB label, the invariants from the database are returned as `extra_args`:

```
sage: key, data = EllipticCurve.create_key_and_extra_args('389.a1')
sage: key
(Rational Field, (0, 1, 1, -2, 0))
sage: data['conductor']
389
sage: data['cremona_label']
'389a1'
sage: data['lmfdb_label']
'389.a1'
sage: data['rank']
2
sage: data['torsion_order']
1
```

User-specified keywords are also included in `extra_args`:

```
sage: key, data = EllipticCurve.create_key_and_extra_args((0, 0, 1, -23737,␣
↪960366), rank=4)
sage: data['rank']
4
```

Furthermore, keywords takes precedence over data from the database, which can be used to specify an alternative set of generators for the Mordell-Weil group:

```
sage: key, data = EllipticCurve.create_key_and_extra_args('5077a1', gens=[[1,␣
↪-1], [-2, 3], [4, -7]])
```

(continues on next page)

(continued from previous page)

```
sage: data['gens']
[[1, -1], [-2, 3], [4, -7]]
sage: E = EllipticCurve.create_object(0, key, **data)
sage: E.gens()
[(-2 : 3 : 1), (1 : -1 : 1), (4 : -7 : 1)]
```

Note that elliptic curves are equal if and only they have the same base ring and Weierstrass equation; the data in `extra_args` do not influence comparison of elliptic curves. A consequence of this is that passing keyword arguments only works when constructing an elliptic curve the first time:

```
sage: E = EllipticCurve('433a1', gens=[[-1, 1], [3, 4]])
sage: E.gens()
[(-1 : 1 : 1), (3 : 4 : 1)]
sage: E = EllipticCurve('433a1', gens=[[-1, 0], [0, 1]])
sage: E.gens()
[(-1 : 1 : 1), (3 : 4 : 1)]
```

Warning: Manually specifying extra data is almost never necessary and is not guaranteed to have any effect, as the above example shows. Almost no checking is done, so specifying incorrect data may lead to wrong results of computations instead of errors or warnings.

create_object(*version*, *key*, ***kwds*)

Create an object from a `UniqueFactory` key.

EXAMPLES:

```
sage: E = EllipticCurve.create_object(0, (GF(3), (1, 2, 0, 1, 2)))
sage: type(E)
<class 'sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_
↪field_with_category'>
```

Note: Keyword arguments are currently only passed to the constructor for elliptic curves over **Q**; elliptic curves over other fields do not support them.

sage.schemes.elliptic_curves.constructor.**EllipticCurve_from_Weierstrass_polynomial**(*f*)

Return the elliptic curve defined by a cubic in (long) Weierstrass form.

INPUT:

- `f` – a inhomogeneous cubic polynomial in long Weierstrass form.

OUTPUT: The elliptic curve defined by it.

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: f = y^2 + 1*x*y + 3*y - (x^3 + 2*x^2 + 4*x + 6)
sage: EllipticCurve(f)
Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 6 over Rational␣
↪Field
sage: EllipticCurve(f).a_invariants()
(1, 2, 3, 4, 6)
```

The polynomial ring may have extra variables as long as they do not occur in the polynomial itself:

```
sage: R.<x,y,z,w> = QQ[]
sage: EllipticCurve(-y^2 + x^3 + 1)
Elliptic Curve defined by y^2 = x^3 + 1 over Rational Field
sage: EllipticCurve(-x^2 + y^3 + 1)
Elliptic Curve defined by y^2 = x^3 + 1 over Rational Field
sage: EllipticCurve(-w^2 + z^3 + 1)
Elliptic Curve defined by y^2 = x^3 + 1 over Rational Field
```

sage.schemes.elliptic_curves.constructor.**EllipticCurve_from_c4c6**(*c4*, *c6*)

Return an elliptic curve with given c_4 and c_6 invariants.

EXAMPLES:

```
sage: E = EllipticCurve_from_c4c6(17, -2005)
sage: E
Elliptic Curve defined by y^2  = x^3 - 17/48*x + 2005/864 over Rational Field
sage: E.c_invariants()
(17, -2005)
```

sage.schemes.elliptic_curves.constructor.**EllipticCurve_from_cubic**(*F*, *P=None*, *morphism=True*)

Construct an elliptic curve from a ternary cubic with a rational point.

If you just want the Weierstrass form and are not interested in the morphism then it is easier to use the function *Jacobian()* instead. If there is a rational point on the given cubic, this function will construct the same elliptic curve but you do not have to supply the point `P`.

INPUT:

- `F` – a homogeneous cubic in three variables with rational coefficients, as a polynomial ring element, defining a smooth plane cubic curve C.
- `P` – a 3-tuple (x, y, z) defining a projective point on C, or `None`. If `None` then a rational flex will be used as a base point if one exists, otherwise an error will be raised.
- `morphism` – boolean (default: `True`). If `True` returns a birational isomorphism from C to a Weierstrass elliptic curve E, otherwise just returns E.

OUTPUT:

Either (when `morphism``=``False`) an elliptic curve E in long Weierstrass form isomorphic to the plane cubic curve C defined by the equation $F = 0$.

Or (when `morphism=True`), a birational isomorphism from C to the elliptic curve E. If the given point is a flex, this is a linear isomorphism.

Note: The function *Jacobian()* may be used instead. It constructs the same elliptic curve (which is in all cases the Jacobian of $(F = 0)$) and needs no base point to be provided, but also returns no isomorphism since in general there is none: the plane cubic is only isomorphic to its Jacobian when it has a rational point.

Note: When `morphism=True`, a birational isomorphism between the curve $F = 0$ and the Weierstrass curve is returned. If the point happens to be a flex, then this is a linear isomorphism. The morphism does not necessarily take the given point P to the point at infinity on E, since we always use a rational flex on C as base-point when one exists.

EXAMPLES:

First we find that the Fermat cubic is isomorphic to the curve with Cremona label 27a1:

```
sage: R.<x,y,z> = QQ[]
sage: cubic = x^3 + y^3 + z^3
sage: P = [1,-1,0]
sage: E = EllipticCurve_from_cubic(cubic, P, morphism=False); E
Elliptic Curve defined by y^2 - 9*y = x^3 - 27 over Rational Field
sage: E.cremona_label()
'27a1'
sage: EllipticCurve_from_cubic(cubic, [0,1,-1], morphism=False).cremona_label()
'27a1'
sage: EllipticCurve_from_cubic(cubic, [1,0,-1], morphism=False).cremona_label()
'27a1'
```

Next we find the minimal model and conductor of the Jacobian of the Selmer curve:

```
sage: R.<a,b,c> = QQ[]
sage: cubic = a^3 + b^3 + 60*c^3
sage: P = [1,-1,0]
sage: E = EllipticCurve_from_cubic(cubic, P, morphism=False);  E
Elliptic Curve defined by y^2 - 540*y = x^3 - 97200 over Rational Field
sage: E.minimal_model()
Elliptic Curve defined by y^2 = x^3 - 24300 over Rational Field
sage: E.conductor()
24300
```

We can also get the birational isomorphism to and from the Weierstrass form. We start with an example where `P` is a flex and the equivalence is a linear isomorphism:

```
sage: f = EllipticCurve_from_cubic(cubic, P, morphism=True)
sage: f
Scheme morphism:
  From: Projective Plane Curve over Rational Field defined by a^3 + b^3 + 60*c^3
  To:   Elliptic Curve defined by y^2 - 540*y = x^3 - 97200 over Rational Field
  Defn: Defined on coordinates by sending (a : b : c) to
        (-c : 3*a : 1/180*a + 1/180*b)

sage: finv = f.inverse();  finv
Scheme morphism:
  From: Elliptic Curve defined by y^2 - 540*y = x^3 - 97200 over Rational Field
  To:   Projective Plane Curve over Rational Field defined by a^3 + b^3 + 60*c^3
  Defn: Defined on coordinates by sending (x : y : z) to
        (1/3*y : -1/3*y + 180*z : -x)

Scheme morphism:
  From: Elliptic Curve defined by y^2 + 2*x*y + 20*y = x^3 - x^2 - 20*x - 400/3
        over Rational Field
  To:   Closed subscheme of Projective Space of dimension 2 over Rational Field
        defined by: a^3 + b^3 + 60*c^3
  Defn: Defined on coordinates by sending (x : y : z) to
        (x + y + 20*z : -x - y : -x)
```

We verify that f maps the chosen point $P = (1, -1, 0)$ on the cubic to the origin of the elliptic curve:

```
sage: f([1,-1,0])
(0 : 1 : 0)
sage: finv([0,1,0])
(-1 : 1 : 0)
```

To verify the output, we plug in the polynomials to check that this indeed transforms the cubic into Weierstrass form:

```
sage: cubic(finv.defining_polynomials()) * finv.post_rescaling()
-x^3 + y^2*z - 540*y*z^2 + 97200*z^3

sage: E.defining_polynomial()(f.defining_polynomials()) * f.post_rescaling()
a^3 + b^3 + 60*c^3
```

If the given point is not a flex and the cubic has no rational flexes, then the cubic can not be transformed to a Weierstrass equation by a linear transformation. The general birational transformation is still a birational isomorphism, but is quadratic:

```
sage: R.<x,y,z> = QQ[]
sage: cubic = x^2*y + 4*x*y^2 + x^2*z + 8*x*y*z + 4*y^2*z + 9*x*z^2 + 9*y*z^2
sage: f = EllipticCurve_from_cubic(cubic, [1,-1,1], morphism=True); f
Scheme morphism:
  From: Projective Plane Curve over Rational Field defined
        by x^2*y + 4*x*y^2 + x^2*z + 8*x*y*z + 4*y^2*z + 9*x*z^2 + 9*y*z^2
  To:   Elliptic Curve defined
        by y^2 + 7560/19*x*y + 552960000000/2352637*y = x^3 - 3445200/133*x^2
        over Rational Field
  Defn: Defined on coordinates by sending (x : y : z) to
        (2527/17280*x^2 + 133/2160*x*y + 133/108000*y^2 + 133/2880*x*z
           + 931/18000*y*z - 3857/48000*z^2
         : -6859/288*x^2 + 323/36*x*y + 359/1800*y^2 + 551/48*x*z
           + 2813/300*y*z + 24389/800*z^2
         : -2352637/99532800000*x^2 - 2352637/124416000000*x*y
           - 2352637/622080000000*y^2 + 2352637/82944000000*x*z
           + 2352637/207360000000*y*z - 2352637/276480000000*z^2)
```

Note that the morphism returned cannot be evaluated directly at the given point `P=(1:-1:1)` since the polynomials defining it all vanish there:

```
sage: f([1,-1,1])
Traceback (most recent call last):
...
ValueError: [0, 0, 0] does not define a valid projective point since all entries␣
↪are zero
```

Using the group law on the codomain elliptic curve, which has rank 1 and full 2-torsion, and the inverse morphism, we can find many points on the cubic. First we find the preimages of multiples of the generator:

```
sage: E = f.codomain()
sage: E.label()
'720e2'
sage: E.rank()
1
sage: R = E.gens()[0]; R
(-17280000/2527 : 9331200000/6859 : 1)
sage: finv = f.inverse()
sage: [finv(k*R) for k in range(1,10)]
[(-4 : 1 : 0),
(-1 : 4 : 1),
(-20 : -55/76 : 1),
(319/399 : -11339/7539 : 1),
(159919/14360 : -4078139/1327840 : 1),
(-27809119/63578639 : 1856146436/3425378659 : 1),
```

(continues on next page)

(continued from previous page)

```
(-510646582340/56909753439 : 424000923715/30153806197284 : 1),
(-56686114363679/4050436059492161 : -24330348169777728281/1072927821085503881 : 1),
(650589589099815846721/72056273157352822480 : -347376189546061993109881/
↪19412738349594402675232 0 : 1)]
```

The elliptic curve also has torsion, which we can map back:

```
sage: E.torsion_points()
[(0 : 1 : 0),
 (-144000000/17689 : 3533760000000/2352637 : 1),
 (-92160000/17689 : 2162073600000/2352637 : 1),
 (-5760000/17689 : -124070400000/2352637 : 1)]
sage: [finv(Q) for Q in E.torsion_points() if Q]
[(9 : -9/4 : 1), (-9 : 0 : 1), (0 : 1 : 0)]
```

In this example, the given point `P` is not a flex but the cubic does have a rational flex, `(-4:0:1)`. We return a linear isomorphism which maps this flex to the point at infinity on the Weierstrass model:

```
sage: R.<a,b,c> = QQ[]
sage: cubic =  a^3 + 7*b^3 + 64*c^3
sage: P = [2,2,-1]
sage: f = EllipticCurve_from_cubic(cubic, P, morphism=True)
sage: E = f.codomain();  E
Elliptic Curve defined by y^2 - 258048*y = x^3 - 22196256768 over Rational Field
sage: E.minimal_model()
Elliptic Curve defined by y^2 + y = x^3 - 331 over Rational Field

sage: f
Scheme morphism:
  From: Projective Plane Curve over Rational Field defined by a^3 + 7*b^3 + 64*c^3
  To:   Elliptic Curve defined by y^2 - 258048*y = x^3 - 22196256768 over␣
↪Rational Field
  Defn: Defined on coordinates by sending (a : b : c) to
        (b : -48*a : -1/5376*a - 1/1344*c)

sage: finv = f.inverse();  finv
Scheme morphism:
  From: Elliptic Curve defined by y^2 - 258048*y = x^3 - 22196256768 over␣
↪Rational Field
  To:   Projective Plane Curve over Rational Field defined by a^3 + 7*b^3 + 64*c^3
  Defn: Defined on coordinates by sending (x : y : z) to
        (-1/48*y : x : 1/192*y - 1344*z)

sage: cubic(finv.defining_polynomials()) * finv.post_rescaling()
-x^3 + y^2*z - 258048*y*z^2 + 22196256768*z^3

sage: E.defining_polynomial()(f.defining_polynomials()) * f.post_rescaling()
a^3 + 7*b^3 + 64*c^3

sage: f(P)
(5376 : -258048 : 1)
sage: f([-4,0,1])
(0 : 1 : 0)
```

It is possible to not provide a base point `P` provided that the cubic has a rational flex. In this case the flexes will be found and one will be used as a base point:

```
sage: R.<x,y,z> = QQ[]
sage: cubic = x^3 + y^3 + z^3
sage: f = EllipticCurve_from_cubic(cubic, morphism=True)
sage: f
Scheme morphism:
  From: Projective Plane Curve over Rational Field defined by x^3 + y^3 + z^3
  To:   Elliptic Curve defined by y^2 - 9*y = x^3 - 27 over Rational Field
  Defn: Defined on coordinates by sending (x : y : z) to
        (y : -3*x : -1/3*x - 1/3*z)
```

An error will be raised if no point is given and there are no rational flexes:

```
sage: R.<x,y,z> = QQ[]
sage: cubic = 3*x^3 + 4*y^3 + 5*z^3
sage: EllipticCurve_from_cubic(cubic)
Traceback (most recent call last):
...
ValueError: A point must be given when the cubic has no rational flexes
```

An example over a finite field, using a flex:

```
sage: K = GF(17)
sage: R.<x,y,z> = K[]
sage: cubic = 2*x^3 + 3*y^3 + 4*z^3
sage: EllipticCurve_from_cubic(cubic, [0,3,1])
Scheme morphism:
  From: Projective Plane Curve over Finite Field of size 17
        defined by 2*x^3 + 3*y^3 + 4*z^3
  To:   Elliptic Curve defined by y^2 + 16*y = x^3 + 11 over Finite Field of size␣
↪17
  Defn: Defined on coordinates by sending (x : y : z) to
        (-x : 4*y : 4*y + 5*z)
```

An example in characteristic 3:

```
sage: K = GF(3)
sage: R.<x,y,z> = K[]
sage: cubic = x^3 + y^3 + z^3 + x*y*z
sage: EllipticCurve_from_cubic(cubic, [0,1,-1])
Scheme morphism:
  From: Projective Plane Curve over Finite Field of size 3
        defined by x^3 + y^3 + x*y*z + z^3
  To:   Elliptic Curve defined by y^2 + x*y = x^3 + 1 over Finite Field of size 3
  Defn: Defined on coordinates by sending (x : y : z) to
        (y + z : -y : x)
```

An example over a number field, using a non-flex and where there are no rational flexes:

```
sage: # needs sage.rings.number_field
sage: K.<a> = QuadraticField(-3)
sage: R.<x,y,z> = K[]
sage: cubic = 2*x^3 + 3*y^3 + 5*z^3
sage: EllipticCurve_from_cubic(cubic, [1,1,-1])
Scheme morphism:
  From: Projective Plane Curve over Number Field in a
        with defining polynomial x^2 + 3 with a = 1.732050807568878?*I
        defined by 2*x^3 + 3*y^3 + 5*z^3
```

(continues on next page)

(continued from previous page)

```
  To:   Elliptic Curve defined by
        y^2 + 1754460/2053*x*y + 5226454388736000/8653002877*y
        = x^3 + (-652253285700/4214809)*x^2
        over Number Field in a with defining polynomial x^2 + 3
        with a = 1.732050807568878?*I
  Defn: Defined on coordinates by sending (x : y : z) to
        (-16424/127575*x^2 - 231989/680400*x*y - 14371/64800*y^2 - 26689/
↪81648*x*z - 10265/27216*y*z - 2053/163296*z^2
         : 24496/315*x^2 + 119243/840*x*y + 4837/80*y^2 + 67259/504*x*z + 25507/
↪168*y*z + 5135/1008*z^2
         : 8653002877/2099914709760000*x^2 + 8653002877/699971569920000*x*y +␣
↪8653002877/933295426560000*y^2 + 8653002877/419982941952000*x*x*z + 8653002877/
↪279988627968000*y*z + 8653002877/335986353561600*z^2)
```

An example over a function field, using a non-flex:

```
sage: K.<t> = FunctionField(QQ)
sage: R.<x,y,z> = K[]
sage: cubic = x^3 + t*y^3 + (1+t)*z^3
sage: EllipticCurve_from_cubic(cubic, [1,1,-1], morphism=False)                #␣
↪needs sage.libs.singular
Elliptic Curve defined by y^2 + ((162*t^6+486*t^5+810*t^4+810*t^3+486*t^2+162*t)/
↪(t^6+12*t^5-3*t^4-20*t^3-3*t^2+12*t+1))*x*y + ((314928*t^14+4094064*t^
↪13+23462136*t^12+78102144*t^11+167561379*t^10+243026001*t^9+243026001*t^
↪8+167561379*t^7+78102144*t^6+23462136*t^5+4094064*t^4+314928*t^3)/(t^14+40*t^
↪13+577*t^12+3524*t^11+8075*t^10+5288*t^9-8661*t^8-17688*t^7-8661*t^6+5288*t^
↪5+8075*t^4+3524*t^3+577*t^2+40*t+1))*y = x^3 + ((2187*t^12+13122*t^11-17496*t^
↪10-207765*t^9-516132*t^8-673596*t^7-516132*t^6-207765*t^5-17496*t^4+13122*t^
↪3+2187*t^2)/(t^12+24*t^11+138*t^10-112*t^9-477*t^8+72*t^7+708*t^6+72*t^5-477*t^
↪4-112*t^3+138*t^2+24*t+1))*x^2
 over Rational function field in t over Rational Field
```

sage.schemes.elliptic_curves.constructor.**EllipticCurve_from_j**(*j*, *minimal_twist=True*)

Return an elliptic curve with given j-invariant.

INPUT:

- `j` – an element of some field.
- `minimal_twist` (boolean, default: `True`) – If True and `j` is in **Q**, the curve returned is a minimal twist, i.e. has minimal conductor; when there is more than one curve with minimal conductor, the curve returned is the one whose label comes first if the curves are in the CremonaDatabase, otherwise the one whose minimal a-invariants are first lexicographically. If j is not in **Q** this parameter is ignored.

OUTPUT:

An elliptic curve with j-invariant j.

EXAMPLES:

```
sage: E = EllipticCurve_from_j(0); E; E.j_invariant(); E.label()
Elliptic Curve defined by y^2 + y = x^3 over Rational Field
0
'27a3'

sage: E = EllipticCurve_from_j(1728); E; E.j_invariant(); E.label()
Elliptic Curve defined by y^2 = x^3 - x over Rational Field
1728
```

(continues on next page)

(continued from previous page)

```
'32a2'

sage: E = EllipticCurve_from_j(1); E; E.j_invariant()
Elliptic Curve defined by y^2 + x*y = x^3 + 36*x + 3455 over Rational Field
1
```

The `minimal_twist` parameter (ignored except over **Q** and True by default) controls whether or not a minimal twist is computed:

```
sage: EllipticCurve_from_j(100)
Elliptic Curve defined by y^2 = x^3 + x^2 + 3392*x + 307888 over Rational Field
sage: _.conductor()
33129800
sage: EllipticCurve_from_j(100, minimal_twist=False)
Elliptic Curve defined by y^2 = x^3 + 488400*x - 530076800 over Rational Field
sage: _.conductor()
298168200
```

Since computing the minimal twist requires factoring both j and $j - 1728$ the following example would take a long time without setting `minimal_twist` to False:

```
sage: E = EllipticCurve_from_j(2^256+1, minimal_twist=False)
sage: E.j_invariant() == 2^256+1
True
```

sage.schemes.elliptic_curves.constructor.**EllipticCurves_with_good_reduction_outside_S**(*S=[]*, *proof=* *verbose=1*

Return a sorted list of all elliptic curves defined over **Q** with good reduction outside the set S of primes.

INPUT:

- `S` – list of primes (default: empty list)
- `proof` – boolean (default `True`): the MW basis for auxiliary curves will be computed with this proof flag
- `verbose` – boolean (default `False`): if `True`, some details of the computation will be output

---

Note: Proof flag: The algorithm used requires determining all S-integral points on several auxiliary curves, which in turn requires the computation of their generators. This is not always possible (even in theory) using current knowledge.

The value of this flag is passed to the function which computes generators of various auxiliary elliptic curves, in order to find their S-integral points. Set to `False` if the default (`True`) causes warning messages, but note that you can then not rely on the set of curves returned being complete.

EXAMPLES:

```
sage: EllipticCurves_with_good_reduction_outside_S([])
[]
sage: elist = EllipticCurves_with_good_reduction_outside_S([2])
sage: elist
[Elliptic Curve defined by y^2 = x^3 + 4*x over Rational Field,
 Elliptic Curve defined by y^2 = x^3 - x over Rational Field,
 ...
```

(continues on next page)

(continued from previous page)

```
 Elliptic Curve defined by y^2 = x^3 - x^2 - 13*x + 21 over Rational Field]
sage: len(elist)
24
sage: ', '.join(e.label() for e in elist)
'32a1, 32a2, 32a3, 32a4, 64a1, 64a2, 64a3, 64a4, 128a1, 128a2, 128b1, 128b2,␣
↪128c1, 128c2, 128d1, 128d2, 256a1, 256a2, 256b1, 256b2, 256c1, 256c2, 256d1,␣
↪256d2'
```

Without `Proof=False`, this example gives two warnings:

```
sage: elist = EllipticCurves_with_good_reduction_outside_S([11], proof=False)    #␣
↪long time (14s on sage.math, 2011)
sage: len(elist)                                                                  #␣
↪long time
12
sage: ', '.join(e.label() for e in elist)                                         #␣
↪long time
'11a1, 11a2, 11a3, 121a1, 121a2, 121b1, 121b2, 121c1, 121c2, 121d1, 121d2, 121d3'

sage: # long time
sage: elist = EllipticCurves_with_good_reduction_outside_S([2,3])                 #␣
↪long time (26s on sage.math, 2011)
sage: len(elist)
752
sage: conds = sorted(set([e.conductor() for e in elist]))
sage: max(conds)
62208
sage: [N.factor() for N in conds]
[2^3 * 3,
 3^3,
 2^5,
 2^2 * 3^2,
 2^4 * 3,
 2 * 3^3,
 2^6,
 2^3 * 3^2,
 2^5 * 3,
 2^2 * 3^3,
 2^7,
 2^4 * 3^2,
 2 * 3^4,
 2^6 * 3,
 2^3 * 3^3,
 3^5,
 2^8,
 2^5 * 3^2,
 2^2 * 3^4,
 2^7 * 3,
 2^4 * 3^3,
 2 * 3^5,
 2^6 * 3^2,
 2^3 * 3^4,
 2^8 * 3,
 2^5 * 3^3,
 2^2 * 3^5,
 2^7 * 3^2,
 2^4 * 3^4,
```

(continues on next page)

(continued from previous page)

```
 2^6 * 3^3,
 2^3 * 3^5,
 2^8 * 3^2,
 2^5 * 3^4,
 2^7 * 3^3,
 2^4 * 3^5,
 2^6 * 3^4,
 2^8 * 3^3,
 2^5 * 3^5,
 2^7 * 3^4,
 2^6 * 3^5,
 2^8 * 3^4,
 2^7 * 3^5,
 2^8 * 3^5]
```

sage.schemes.elliptic_curves.constructor.**are_projectively_equivalent**(*P*, *Q*, *base_ring*)

Test whether `P` and `Q` are projectively equivalent.

INPUT:

- `P`, `Q` – list/tuple of projective coordinates.
- `base_ring` – the base ring.

OUTPUT: A boolean.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.constructor import are_projectively_
↪equivalent
sage: are_projectively_equivalent([0,1,2,3], [0,1,2,2], base_ring=QQ)
False
sage: are_projectively_equivalent([0,1,2,3], [0,2,4,6], base_ring=QQ)
True
```

sage.schemes.elliptic_curves.constructor.**chord_and_tangent**(*F*, *P*)

Return the third point of intersection of a cubic with the tangent at one point.

INPUT:

- `F` – a homogeneous cubic in three variables with rational coefficients, as a polynomial ring element, defining a smooth plane cubic curve.
- `P` – a 3-tuple (x, y, z) defining a projective point on the curve $F = 0$.

OUTPUT:

A point `Q` such that `F(Q)=0`, namely the third point of intersection of the tangent at `P` with the curve `F=0`, so `Q=P` if and only if `P` is a flex.

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: from sage.schemes.elliptic_curves.constructor import chord_and_tangent
sage: F = x^3 + y^3 + 60*z^3
sage: chord_and_tangent(F, [1,-1,0])
(-1 : 1 : 0)
```

(continues on next page)

(continued from previous page)

```
sage: F = x^3 + 7*y^3 + 64*z^3
sage: p0 = [2,2,-1]
sage: p1 = chord_and_tangent(F, p0);  p1
(5 : -3 : 1)
sage: p2 = chord_and_tangent(F, p1);  p2
(-1265/314 : 183/314 : 1)
```

sage.schemes.elliptic_curves.constructor.**coefficients_from_Weierstrass_polynomial**(*f*)

Return the coefficients $[a_1, a_2, a_3, a_4, a_6]$ of a cubic in Weierstrass form.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.constructor import coefficients_from_
↪Weierstrass_polynomial
sage: R.<w,z> = QQ[]
sage: coefficients_from_Weierstrass_polynomial(-w^2 + z^3 + 1)
[0, 0, 0, 0, 1]
```

sage.schemes.elliptic_curves.constructor.**coefficients_from_j**(*j*, *minimal_twist=True*)

Return Weierstrass coefficients $(a_1, a_2, a_3, a_4, a_6)$ for an elliptic curve with given j-invariant.

INPUT: See *EllipticCurve_from_j()*.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.constructor import coefficients_from_j
sage: coefficients_from_j(0)
[0, 0, 1, 0, 0]
sage: coefficients_from_j(1728)
[0, 0, 0, -1, 0]
sage: coefficients_from_j(1)
[1, 0, 0, 36, 3455]
```

The `minimal_twist` parameter (ignored except over **Q** and True by default) controls whether or not a minimal twist is computed:

```
sage: coefficients_from_j(100)
[0, 1, 0, 3392, 307888]
sage: coefficients_from_j(100, minimal_twist=False)
[0, 0, 0, 488400, -530076800]
```

sage.schemes.elliptic_curves.constructor.**projective_point**(*p*)

Return equivalent point with denominators removed

INPUT:

- `P`, `Q` – list/tuple of projective coordinates.

OUTPUT:

List of projective coordinates.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.constructor import projective_point
sage: projective_point([4/5, 6/5, 8/5])
[2, 3, 4]
sage: F = GF(11)
```

(continues on next page)

(continued from previous page)

```
sage: projective_point([F(4), F(8), F(2)])
[4, 8, 2]
```

sage.schemes.elliptic_curves.constructor.**tangent_at_smooth_point**(C, P)

Return the tangent at the smooth point P of projective curve C.

INPUT:

- C – a projective plane curve.
- P – a 3-tuple (x, y, z) defining a projective point on C.

OUTPUT:

The linear form defining the tangent at P to C.

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: from sage.schemes.elliptic_curves.constructor import tangent_at_smooth_point
sage: C = Curve(x^3 + y^3 + 60*z^3)
sage: tangent_at_smooth_point(C, [1,-1,0])
x + y

sage: K.<t> = FunctionField(QQ)
sage: R.<x,y,z> = K[]
sage: C = Curve(x^3 + 2*y^3 + 3*z^3)
sage: from sage.schemes.elliptic_curves.constructor import tangent_at_smooth_point
sage: tangent_at_smooth_point(C,[1,1,-1])
3*x + 6*y + 9*z
```

CHAPTER

TWO

CONSTRUCT ELLIPTIC CURVES AS JACOBIANS

An elliptic curve is a genus one curve with a designated point. The Jacobian of a genus-one curve can be defined as the set of line bundles on the curve, and is isomorphic to the original genus-one curve. It is also an elliptic curve with the trivial line bundle as designated point. The utility of this construction is that we can construct elliptic curves without having to specify which point we take as the origin.

EXAMPLES:

```
sage: R.<u,v,w> = QQ[]
sage: Jacobian(u^3 + v^3 + w^3)
Elliptic Curve defined by y^2 = x^3 - 27/4 over Rational Field
sage: Jacobian(u^4 + v^4 + w^2)
Elliptic Curve defined by y^2 = x^3 - 4*x over Rational Field

sage: C = Curve(u^3 + v^3 + w^3)
sage: Jacobian(C)
Elliptic Curve defined by y^2 = x^3 - 27/4 over Rational Field

sage: P2.<u,v,w> = ProjectiveSpace(2, QQ)
sage: C = P2.subscheme(u^3 + v^3 + w^3)
sage: Jacobian(C)
Elliptic Curve defined by y^2 = x^3 - 27/4 over Rational Field
```

One can also define Jacobians of varieties that are not genus-one curves. These are not implemented in this module, but we call the relevant functionality:

```
sage: R.<x> = PolynomialRing(QQ)
sage: f = x**5 + 1184*x**3 + 1846*x**2 + 956*x + 560
sage: C = HyperellipticCurve(f)
sage: Jacobian(C)
Jacobian of Hyperelliptic Curve over Rational Field defined
 by y^2 = x^5 + 1184*x^3 + 1846*x^2 + 956*x + 560
```

REFERENCES:

- Wikipedia article Jacobian_variety

sage.schemes.elliptic_curves.jacobian.**Jacobian**(*X*, ***kwds*)

Return the Jacobian.

INPUT:

- `X` – polynomial, algebraic variety, or anything else that has a Jacobian elliptic curve.
- `kwds` – optional keyword arguments.

The input `X` can be one of the following:

- A polynomial, see *Jacobian_of_equation()* for details.
- A curve, see *Jacobian_of_curve()* for details.

EXAMPLES:

```
sage: R.<u,v,w> = QQ[]
sage: Jacobian(u^3 + v^3 + w^3)
Elliptic Curve defined by y^2 = x^3 - 27/4 over Rational Field

sage: C = Curve(u^3 + v^3 + w^3)
sage: Jacobian(C)
Elliptic Curve defined by y^2 = x^3 - 27/4 over Rational Field

sage: P2.<u,v,w> = ProjectiveSpace(2, QQ)
sage: C = P2.subscheme(u^3 + v^3 + w^3)
sage: Jacobian(C)
Elliptic Curve defined by y^2 = x^3 - 27/4 over Rational Field

sage: Jacobian(C, morphism=True)
Scheme morphism:
  From: Closed subscheme of Projective Space of dimension 2 over Rational Field␣
↪defined by:
  u^3 + v^3 + w^3
  To:   Elliptic Curve defined by y^2 = x^3 - 27/4 over Rational Field
  Defn: Defined on coordinates by sending (u : v : w) to
        (-u^4*v^4*w - u^4*v*w^4 - u*v^4*w^4 :
         1/2*u^6*v^3 - 1/2*u^3*v^6 - 1/2*u^6*w^3 + 1/2*v^6*w^3 + 1/2*u^3*w^6 - 1/
↪2*v^3*w^6 :
         u^3*v^3*w^3)
```

sage.schemes.elliptic_curves.jacobian.**Jacobian_of_curve**(*curve*, *morphism=False*)

Return the Jacobian of a genus-one curve

INPUT:

- `curve` – a one-dimensional algebraic variety of genus one.

OUTPUT: Its Jacobian elliptic curve.

EXAMPLES:

```
sage: R.<u,v,w> = QQ[]
sage: C = Curve(u^3 + v^3 + w^3)
sage: Jacobian(C)
Elliptic Curve defined by y^2 = x^3 - 27/4 over Rational Field
```

sage.schemes.elliptic_curves.jacobian.**Jacobian_of_equation**(*polynomial*, *variables=None*, *curve=None*)

Construct the Jacobian of a genus-one curve given by a polynomial.

INPUT:

- `F` – a polynomial defining a plane curve of genus one. May be homogeneous or inhomogeneous.
- `variables` – list of two or three variables or `None` (default). The inhomogeneous or homogeneous coordinates. By default, all variables in the polynomial are used.
- `curve` – the genus-one curve defined by `polynomial` or # `None` (default). If specified, suitable morphism from the jacobian elliptic curve to the curve is returned.

OUTPUT:

An elliptic curve in short Weierstrass form isomorphic to the curve `polynomial=0`. If the optional argument `curve` is specified, a rational multicover from the Jacobian elliptic curve to the genus-one curve is returned.

EXAMPLES:

```
sage: R.<a,b,c> = QQ[]
sage: f = a^3 + b^3 + 60*c^3
sage: Jacobian(f)
Elliptic Curve defined by y^2 = x^3 - 24300 over Rational Field
sage: Jacobian(f.subs(c=1))
Elliptic Curve defined by y^2 = x^3 - 24300 over Rational Field
```

If we specify the domain curve, the birational covering is returned:

```
sage: h = Jacobian(f, curve=Curve(f));  h
Scheme morphism:
  From: Projective Plane Curve over Rational Field defined by a^3 + b^3 + 60*c^3
  To:   Elliptic Curve defined by y^2 = x^3 - 24300 over Rational Field
  Defn: Defined on coordinates by sending (a : b : c) to
        (-216000*a^4*b^4*c - 12960000*a^4*b*c^4 - 12960000*a*b^4*c^4
         : 108000*a^6*b^3 - 108000*a^3*b^6 - 6480000*a^6*c^3 + 6480000*b^6*c^3
           + 388800000*a^3*c^6 - 388800000*b^3*c^6
         : 216000*a^3*b^3*c^3)

sage: h([1,-1,0])
(0 : 1 : 0)
```

Plugging in the polynomials defining h allows us to verify that it is indeed a rational morphism to the elliptic curve:

```
sage: E = h.codomain()
sage: E.defining_polynomial()(h.defining_polynomials()).factor()
(2519424000000000) * c^3 * b^3 * a^3 * (a^3 + b^3 + 60*c^3)
* (a^9*b^6 + a^6*b^9 - 120*a^9*b^3*c^3 + 900*a^6*b^6*c^3 - 120*a^3*b^9*c^3
   + 3600*a^9*c^6 + 54000*a^6*b^3*c^6 + 54000*a^3*b^6*c^6 + 3600*b^9*c^6
   + 216000*a^6*c^9 - 432000*a^3*b^3*c^9 + 216000*b^6*c^9)
```

By specifying the variables, we can also construct an elliptic curve over a polynomial ring:

```
sage: R.<u,v,t> = QQ[]
sage: Jacobian(u^3 + v^3 + t, variables=[u,v])
Elliptic Curve defined by y^2 = x^3 + (-27/4*t^2) over
 Multivariate Polynomial Ring in u, v, t over Rational Field
```

CHAPTER

THREE

POINTS ON ELLIPTIC CURVES

The base class *EllipticCurvePoint* currently provides little functionality of its own. Its derived class *EllipticCurvePoint_field* provides support for points on elliptic curves over general fields. The derived classes *EllipticCurvePoint_number_field* and *EllipticCurvePoint_finite_field* provide further support for points on curves over number fields (including the rational field **Q**) and over finite fields.

EXAMPLES:

An example over **Q**:

```
sage: E = EllipticCurve('389a1')
sage: P = E(-1,1); P
(-1 : 1 : 1)
sage: Q = E(0,-1); Q
(0 : -1 : 1)
sage: P+Q
(4 : 8 : 1)
sage: P-Q
(1 : 0 : 1)
sage: 3*P-5*Q
(328/361 : -2800/6859 : 1)
```

An example over a number field:

```
sage: # needs sage.rings.number_field
sage: K.<i> = QuadraticField(-1)
sage: E = EllipticCurve(K, [1,0,0,0,-1])
sage: P = E(0,i); P
(0 : i : 1)
sage: P.order()
+Infinity
sage: 101*P - 100*P == P
True
```

An example over a finite field:

```
sage: # needs sage.rings.finite_rings
sage: K.<a> = GF((101,3))
sage: E = EllipticCurve(K, [1,0,0,0,-1])
sage: P = E(40*a^2 + 69*a + 84 , 58*a^2 + 73*a + 45)
sage: P.order()
1032210
sage: E.cardinality()
1032210
```

Arithmetic with a point over an extension of a finite field:

```
sage: # needs sage.rings.finite_rings
sage: k.<a> = GF((5,2))
sage: E = EllipticCurve(k,[1,0]); E
Elliptic Curve defined by y^2 = x^3 + x over Finite Field in a of size 5^2
sage: P = E([a,2*a+4])
sage: 5*P
(2*a + 3 : 2*a : 1)
sage: P*5
(2*a + 3 : 2*a : 1)
sage: P + P + P + P + P
(2*a + 3 : 2*a : 1)
```

```
sage: F = Zmod(3)
sage: E = EllipticCurve(F, [1,0]);
sage: P = E([2,1])
sage: import sys
sage: n = sys.maxsize
sage: P*(n+1)-P*n == P
True
```

Arithmetic over $\mathbf{Z}/N\mathbf{Z}$ with composite N is supported. When an operation tries to invert a non-invertible element, a `ZeroDivisionError` is raised and a factorization of the modulus appears in the error message:

```
sage: N = 1715761513
sage: E = EllipticCurve(Integers(N), [3,-13])
sage: P = E(2,1)
sage: LCM([2..60])*P
Traceback (most recent call last):
...
ZeroDivisionError: Inverse of 26927 does not exist
(characteristic = 1715761513 = 26927*63719)
```

AUTHORS:

- William Stein (2005) – Initial version
- Robert Bradshaw et al….
- John Cremona (Feb 2008) – Point counting and group structure for non-prime fields, Frobenius endomorphism and order, elliptic logs
- John Cremona (Aug 2008) – Introduced `EllipticCurvePoint_number_field` class
- Tobias Nagel, Michael Mardaus, John Cremona (Dec 2008) – p-adic elliptic logarithm over $\mathbf{Q}$
- David Hansen (Jan 2009) – Added `weil_pairing` function to `EllipticCurvePoint_finite_field` class
- Mariah Lenox (March 2011) – Added `tate_pairing` and `ate_pairing` functions to `EllipticCurvePoint_finite_field` class

class sage.schemes.elliptic_curves.ell_point.**EllipticCurvePoint**(*X*, *v*, *check=True*)

Bases: `AdditiveGroupElement`, `SchemeMorphism_point_projective_ring`

A point on an elliptic curve.

curve()

Return the curve that this point is on.

This is a synonym for `scheme()`.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: P = E([-1, 1])
sage: P.curve()
Elliptic Curve defined by y^2 + y = x^3 + x^2 - 2*x over Rational Field

sage: E = EllipticCurve(QQ, [1, 1])
sage: P = E(0, 1)
sage: P.scheme()
Elliptic Curve defined by y^2 = x^3 + x + 1 over Rational Field
sage: P.scheme() == P.curve()
True
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^2 - 3,'a')                                    #␣
↪needs sage.rings.number_field
sage: P = E.base_extend(K)(1, a)                                          #␣
↪needs sage.rings.number_field
sage: P.scheme()                                                          #␣
↪needs sage.rings.number_field
Elliptic Curve defined by y^2 = x^3 + x + 1 over Number Field in a with␣
↪defining polynomial x^2 - 3
```

class sage.schemes.elliptic_curves.ell_point.**EllipticCurvePoint_field**(*curve*, *v*, *check=True*)

Bases: *EllipticCurvePoint*, SchemeMorphism_point_abelian_variety_field

A point on an elliptic curve over a field. The point has coordinates in the base field.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E([0,0])
(0 : 0 : 1)
sage: E(0,0)               # brackets are optional
(0 : 0 : 1)
sage: E([GF(5)(0), 0])     # entries are coerced
(0 : 0 : 1)

sage: E(0.000, 0)
(0 : 0 : 1)

sage: E(1,0,0)
Traceback (most recent call last):
...
TypeError: Coordinates [1, 0, 0] do not define a point on
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
```

```
sage: E = EllipticCurve([0,0,1,-1,0])
sage: S = E(QQ); S
Abelian group of points on
 Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field

sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<i> = NumberField(x^2 + 1)
sage: E = EllipticCurve(K, [0,1,0,-160,308])
sage: P = E(26, -120)
```

(continues on next page)

(continued from previous page)

```
sage: Q = E(2+12*i, -36+48*i)
sage: P.order() == Q.order() == 4        # long time
True
sage: 2*P == 2*Q
False
```

```
sage: K.<t> = FractionField(PolynomialRing(QQ,'t'))
sage: E = EllipticCurve([0,0,0,0,t^2])
sage: P = E(0,t)
sage: P, 2*P, 3*P
((0 : t : 1), (0 : -t : 1), (0 : 1 : 0))
```

additive_order()

Return the order of this point on the elliptic curve.

If the point is zero, returns 1, otherwise raise a `NotImplementedError`.

For curves over number fields and finite fields, see below.

Note: *additive_order()* is a synonym for *order()*

EXAMPLES:

```
sage: K.<t> = FractionField(PolynomialRing(QQ,'t'))
sage: E = EllipticCurve([0, 0, 0, -t^2, 0])
sage: P = E(t,0)
sage: P.order()
Traceback (most recent call last):
...
NotImplementedError: Computation of order of a point not implemented
over general fields.
sage: E(0).additive_order()
1
sage: E(0).order() == 1
True
```

ate_pairing(*Q*, *n*, *k*, *t*, *q=None*)

Return ate pairing of n-torsion points P (`=self`) and Q.

Also known as the n-th modified ate pairing. P is $GF(q)$-rational, and Q must be an element of $Ker(\pi - p)$, where π is the q-frobenius map (and hence Q is $GF(q^k)$-rational).

INPUT:

- P (`=self`) – a point of order n, in $ker(\pi - 1)$, where π is the q-Frobenius map (e.g., P is q-rational).
- `Q` – a point of order n in $ker(\pi - q)$
- `n` – the order of P and Q.
- `k` – the embedding degree.
- `t` – the trace of Frobenius of the curve over $GF(q)$.
- `q` – (default: None) the size of base field (the "big" field is $GF(q^k)$). q needs to be set only if its value cannot be deduced.

OUTPUT:

FiniteFieldElement in $GF(q^k)$ – the ate pairing of P and Q.

EXAMPLES:

An example with embedding degree 6:

```
sage: # needs sage.rings.finite_rings
sage: p = 7549; A = 0; B = 1; n = 157; k = 6; t = 14
sage: F = GF(p); E = EllipticCurve(F, [A, B])
sage: R.<x> = F[]; K.<a> = GF((p,k), modulus=x^k+2)
sage: EK = E.base_extend(K)
sage: P = EK(3050, 5371); Q = EK(6908*a^4, 3231*a^3)
sage: P.ate_pairing(Q, n, k, t)
6708*a^5 + 4230*a^4 + 4350*a^3 + 2064*a^2 + 4022*a + 6733
sage: s = Integer(randrange(1, n))
sage: (s*P).ate_pairing(Q, n, k, t) == P.ate_pairing(s*Q, n, k, t)
True
sage: P.ate_pairing(s*Q, n, k, t) == P.ate_pairing(Q, n, k, t)^s
True
```

Another example with embedding degree 7 and positive trace:

```
sage: # needs sage.rings.finite_rings
sage: p = 2213; A = 1; B = 49; n = 1093; k = 7; t = 28
sage: F = GF(p); E = EllipticCurve(F, [A, B])
sage: R.<x> = F[]; K.<a> = GF((p,k), modulus=x^k+2)
sage: EK = E.base_extend(K)
sage: P = EK(1583, 1734)
sage: Qx = 1729*a^6+1767*a^5+245*a^4+980*a^3+1592*a^2+1883*a+722
sage: Qy = 1299*a^6+1877*a^5+1030*a^4+1513*a^3+1457*a^2+309*a+1636
sage: Q = EK(Qx, Qy)
sage: P.ate_pairing(Q, n, k, t)
1665*a^6 + 1538*a^5 + 1979*a^4 + 239*a^3 + 2134*a^2 + 2151*a + 654
sage: s = Integer(randrange(1, n))
sage: (s*P).ate_pairing(Q, n, k, t) == P.ate_pairing(s*Q, n, k, t)
True
sage: P.ate_pairing(s*Q, n, k, t) == P.ate_pairing(Q, n, k, t)^s
True
```

Another example with embedding degree 7 and negative trace:

```
sage: # needs sage.rings.finite_rings
sage: p = 2017; A = 1; B = 30; n = 29; k = 7; t = -70
sage: F = GF(p); E = EllipticCurve(F, [A, B])
sage: R.<x> = F[]; K.<a> = GF((p,k), modulus=x^k+2)
sage: EK = E.base_extend(K)
sage: P = EK(369, 716)
sage: Qx = 1226*a^6+1778*a^5+660*a^4+1791*a^3+1750*a^2+867*a+770
sage: Qy = 1764*a^6+198*a^5+1206*a^4+406*a^3+1200*a^2+273*a+1712
sage: Q = EK(Qx, Qy)
sage: P.ate_pairing(Q, n, k, t)
1794*a^6 + 1161*a^5 + 576*a^4 + 488*a^3 + 1950*a^2 + 1905*a + 1315
sage: s = Integer(randrange(1, n))
sage: (s*P).ate_pairing(Q, n, k, t) == P.ate_pairing(s*Q, n, k, t)
True
sage: P.ate_pairing(s*Q, n, k, t) == P.ate_pairing(Q, n, k, t)^s
True
```

Using the same data, we show that the ate pairing is a power of the Tate pairing (see [HSV2006] end of

section 3.1):

```
sage: # needs sage.rings.finite_rings
sage: c = (k*p^(k-1)).mod(n); T = t - 1
sage: N = gcd(T^k - 1, p^k - 1)
sage: s = Integer(N/n)
sage: L = Integer((T^k - 1)/N)
sage: M = (L*s*c.inverse_mod(n)).mod(n)
sage: P.ate_pairing(Q, n, k, t) == Q.tate_pairing(P, n, k)^M
True
```

An example where we have to pass the base field size (and we again have agreement with the Tate pairing). Note that though Px is not F-rational, (it is the homomorphic image of an F-rational point) it is nonetheless in $ker(\pi - 1)$, and so is a legitimate input:

```
sage: # needs sage.rings.finite_rings
sage: q = 2^5; F.<a> = GF(q)
sage: n = 41; k = 4; t = -8
sage: E = EllipticCurve(F,[0,0,1,1,1])
sage: P = E(a^4 + 1, a^3)
sage: Fx.<b> = GF(q^k)
sage: Ex = EllipticCurve(Fx, [0,0,1,1,1])
sage: phi = Hom(F, Fx)(F.gen().minpoly().roots(Fx)[0][0])
sage: Px = Ex(phi(P.x()), phi(P.y()))
sage: Qx = Ex(b^19+b^18+b^16+b^12+b^10+b^9+b^8+b^5+b^3+1,
....:         b^18+b^13+b^10+b^8+b^5+b^4+b^3+b)
sage: Qx = Ex(Qx[0]^q, Qx[1]^q) - Qx  # ensure Qx is in ker(pi - q)
sage: Px.ate_pairing(Qx, n, k, t)
Traceback (most recent call last):
...
ValueError: Unexpected field degree: set keyword argument q equal to
the size of the base field (big field is GF(q^4)).
sage: Px.ate_pairing(Qx, n, k, t, q)
b^19 + b^18 + b^17 + b^16 + b^15 + b^14 + b^13 + b^12
+ b^11 + b^9 + b^8 + b^5 + b^4 + b^2 + b + 1
sage: s = Integer(randrange(1, n))
sage: (s*Px).ate_pairing(Qx, n, k, t, q) == Px.ate_pairing(s*Qx, n, k, t, q)
True
sage: Px.ate_pairing(s*Qx, n, k, t, q) == Px.ate_pairing(Qx, n, k, t, q)^s
True
sage: c = (k*q^(k-1)).mod(n); T = t - 1
sage: N = gcd(T^k - 1, q^k - 1)
sage: s = Integer(N/n)
sage: L = Integer((T^k - 1)/N)
sage: M = (L*s*c.inverse_mod(n)).mod(n)
sage: Px.ate_pairing(Qx, n, k, t, q) == Qx.tate_pairing(Px, n, k, q)^M
True
```

It is an error if Q is not in the kernel of $\pi - p$, where π is the Frobenius automorphism:

```
sage: # needs sage.rings.finite_rings
sage: p = 29; A = 1; B = 0; n = 5; k = 2; t = 10
sage: F = GF(p); R.<x> = F[]
sage: E = EllipticCurve(F, [A, B]);
sage: K.<a> = GF((p,k), modulus=x^k+2); EK = E.base_extend(K)
sage: P = EK(13, 8); Q = EK(13, 21)
sage: P.ate_pairing(Q, n, k, t)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: Point (13 : 21 : 1) not in Ker(pi - q)
```

It is also an error if P is not in the kernel os $\pi - 1$:

```
sage: # needs sage.rings.finite_rings
sage: p = 29; A = 1; B = 0; n = 5; k = 2; t = 10
sage: F = GF(p); R.<x> = F[]
sage: E = EllipticCurve(F, [A, B]);
sage: K.<a> = GF((p,k), modulus=x^k+2); EK = E.base_extend(K)
sage: P = EK(14, 10*a); Q = EK(13, 21)
sage: P.ate_pairing(Q, n, k, t)
Traceback (most recent call last):
...
ValueError: This point (14 : 10*a : 1) is not in Ker(pi - 1)
```

Note: First defined in the paper of [HSV2006], the ate pairing can be computationally effective in those cases when the trace of the curve over the base field is significantly smaller than the expected value. This implementation is simply Miller's algorithm followed by a naive exponentiation, and makes no claims towards efficiency.

AUTHORS:

- Mariah Lenox (2011-03-08)

division_points(*m*, *poly_only=False*)

Return a list of all points Q such that $mQ = P$ where P = `self`.

Only points on the elliptic curve containing self and defined over the base field are included.

INPUT:

- `m` – a positive integer
- `poly_only` – bool (default: `False`); if True return polynomial whose roots give all possible x-coordinates of m-th roots of `self`.

OUTPUT:

(list) – a (possibly empty) list of solutions Q to $mQ = P$, where P = self.

EXAMPLES:

We find the five 5-torsion points on an elliptic curve:

```
sage: E = EllipticCurve('11a'); E
Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field
sage: P = E(0); P
(0 : 1 : 0)
sage: P.division_points(5)
[(0 : 1 : 0), (5 : -6 : 1), (5 : 5 : 1), (16 : -61 : 1), (16 : 60 : 1)]
```

Note above that 0 is included since [5]*0 = 0.

We create a curve of rank 1 with no torsion and do a consistency check:

```
sage: E = EllipticCurve('11a').quadratic_twist(-7)
sage: Q = E([44,-270])
sage: (4*Q).division_points(4)
[(44 : -270 : 1)]
```

We create a curve over a non-prime finite field with group of order 18:

```
sage: # needs sage.rings.finite_rings
sage: k.<a> = GF((5,2))
sage: E = EllipticCurve(k, [1,2+a,3,4*a,2])
sage: P = E([3, 3*a+4])
sage: factor(E.order())
2 * 3^2
sage: P.order()
9
```

We find the 1-division points as a consistency check – there is just one, of course:

```
sage: P.division_points(1)                                                  #␣
↪needs sage.rings.finite_rings
[(3 : 3*a + 4 : 1)]
```

The point P has order coprime to 2 but divisible by 3, so:

```
sage: P.division_points(2)                                                  #␣
↪needs sage.rings.finite_rings
[(2*a + 1 : 3*a + 4 : 1), (3*a + 1 : a : 1)]
```

We check that each of the 2-division points works as claimed:

```
sage: [2*Q for Q in P.division_points(2)]                                   #␣
↪needs sage.rings.finite_rings
[(3 : 3*a + 4 : 1), (3 : 3*a + 4 : 1)]
```

Some other checks:

```
sage: P.division_points(3)                                                  #␣
↪needs sage.rings.finite_rings
[]
sage: P.division_points(4)                                                  #␣
↪needs sage.rings.finite_rings
[(0 : 3*a + 2 : 1), (1 : 0 : 1)]
sage: P.division_points(5)                                                  #␣
↪needs sage.rings.finite_rings
[(1 : 1 : 1)]
```

An example over a number field (see Issue #3383):

```
sage: # needs sage.rings.number_field
sage: E = EllipticCurve('19a1')
sage: x = polygen(ZZ, 'x')
sage: K.<t> = NumberField(x^9 - 3*x^8 - 4*x^7 + 16*x^6 - 3*x^5
....:                     - 21*x^4 + 5*x^3 + 7*x^2 - 7*x + 1)
sage: EK = E.base_extend(K)
sage: E(0).division_points(3)
[(0 : 1 : 0), (5 : -10 : 1), (5 : 9 : 1)]
sage: EK(0).division_points(3)
```

(continues on next page)

(continued from previous page)

```
[(0 : 1 : 0), (5 : 9 : 1), (5 : -10 : 1)]
sage: E(0).division_points(9)
[(0 : 1 : 0), (5 : -10 : 1), (5 : 9 : 1)]
sage: EK(0).division_points(9)
[(0 : 1 : 0), (5 : 9 : 1), (5 : -10 : 1), (-150/121*t^8 + 414/121*t^7 + 1481/
↪242*t^6 - 2382/121*t^5 - 103/242*t^4 + 629/22*t^3 - 367/242*t^2 - 1307/
↪121*t + 625/121 : 35/484*t^8 - 133/242*t^7 + 445/242*t^6 - 799/242*t^5 +␣
↪373/484*t^4 + 113/22*t^3 - 2355/484*t^2 - 753/242*t + 1165/484 : 1), (-150/
↪121*t^8 + 414/121*t^7 + 1481/242*t^6 - 2382/121*t^5 - 103/242*t^4 + 629/
↪22*t^3 - 367/242*t^2 - 1307/121*t + 625/121 : -35/484*t^8 + 133/242*t^7 -␣
↪445/242*t^6 + 799/242*t^5 - 373/484*t^4 - 113/22*t^3 + 2355/484*t^2 + 753/
↪242*t - 1649/484 : 1), (-1383/484*t^8 + 970/121*t^7 + 3159/242*t^6 - 5211/
↪121*t^5 + 37/484*t^4 + 654/11*t^3 - 909/484*t^2 - 4831/242*t + 6791/484 :␣
↪927/121*t^8 - 5209/242*t^7 - 8187/242*t^6 + 27975/242*t^5 - 1147/242*t^4 -␣
↪1729/11*t^3 + 1566/121*t^2 + 12873/242*t - 10871/242 : 1), (-1383/484*t^8 +␣
↪970/121*t^7 + 3159/242*t^6 - 5211/121*t^5 + 37/484*t^4 + 654/11*t^3 - 909/
↪484*t^2 - 4831/242*t + 6791/484 : -927/121*t^8 + 5209/242*t^7 + 8187/242*t^
↪6 - 27975/242*t^5 + 1147/242*t^4 + 1729/11*t^3 - 1566/121*t^2 - 12873/242*t␣
↪+ 10629/242 : 1), (-4793/484*t^8 + 6791/242*t^7 + 10727/242*t^6 - 18301/
↪121*t^5 + 2347/484*t^4 + 2293/11*t^3 - 7311/484*t^2 - 17239/242*t + 26767/
↪484 : 30847/484*t^8 - 21789/121*t^7 - 34605/121*t^6 + 117164/121*t^5 -␣
↪10633/484*t^4 - 29437/22*t^3 + 39725/484*t^2 + 55428/121*t - 176909/484 :␣
↪1), (-4793/484*t^8 + 6791/242*t^7 + 10727/242*t^6 - 18301/121*t^5 + 2347/
↪484*t^4 + 2293/11*t^3 - 7311/484*t^2 - 17239/242*t + 26767/484 : -30847/
↪484*t^8 + 21789/121*t^7 + 34605/121*t^6 - 117164/121*t^5 + 10633/484*t^4 +␣
↪29437/22*t^3 - 39725/484*t^2 - 55428/121*t + 176425/484 : 1)]
```

has_finite_order()

Return `True` if this point has finite additive order as an element of the group of points on this curve.

For fields other than number fields and finite fields, this is NotImplemented unless self.is_zero().

EXAMPLES:

```
sage: K.<t> = FractionField(PolynomialRing(QQ,'t'))
sage: E = EllipticCurve([0, 0, 0, -t^2, 0])
sage: P = E(0)
sage: P.has_finite_order()
True
sage: P = E(t,0)
sage: P.has_finite_order()
Traceback (most recent call last):
...
NotImplementedError: Computation of order of a point not implemented
over general fields.
sage: (2*P).is_zero()
True
```

has_infinite_order()

Return True if this point has infinite additive order as an element of the group of points on this curve.

For fields other than number fields and finite fields, this is NotImplemented unless self.is_zero().

EXAMPLES:

```
sage: K.<t> = FractionField(PolynomialRing(QQ,'t'))
sage: E = EllipticCurve([0, 0, 0, -t^2, 0])
```

(continues on next page)

(continued from previous page)

```
sage: P = E(0)
sage: P.has_infinite_order()
False
sage: P = E(t,0)
sage: P.has_infinite_order()
Traceback (most recent call last):
...
NotImplementedError: Computation of order of a point not implemented over␣
↪general fields.
sage: (2*P).is_zero()
True
```

has_order(*n*)

Test if this point has order exactly n.

INPUT:

- n – integer, or its `Factorization`

ALGORITHM:

Compare a cached order if available, otherwise use `sage.groups.generic.has_order()`.

EXAMPLES:

```
sage: E = EllipticCurve('26b1')
sage: P = E(1, 0)
sage: P.has_order(7)
True
sage: P._order
7
sage: P.has_order(7)
True
```

It also works with a `Factorization` object:

```
sage: E = EllipticCurve(GF(419), [1,0])
sage: P = E(-33, 8)
sage: P.has_order(factor(21))
True
sage: P._order
21
sage: P.has_order(factor(21))
True
```

This method can be much faster than computing the order and comparing:

```
sage: # not tested -- timings are different each time
sage: p = 4 * prod(primes(3,377)) * 587 - 1
sage: E = EllipticCurve(GF(p), [1,0])
sage: %timeit P = E.random_point(); P.set_order(multiple=p+1)
72.4 ms ± 773 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
sage: %timeit P = E.random_point(); P.has_order(p+1)
32.8 ms ± 3.12 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
sage: fac = factor(p+1)
sage: %timeit P = E.random_point(); P.has_order(fac)
30.6 ms ± 3.48 ms per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

The order is cached once it has been confirmed once, and the cache is shared with *order()*:

```
sage: # not tested -- timings are different each time
sage: P, = E.gens()
sage: delattr(P, '_order')
sage: %time P.has_order(p+1)
CPU times: user 83.6 ms, sys: 30 µs, total: 83.6 ms
Wall time: 83.8 ms
True
sage: %time P.has_order(p+1)
CPU times: user 31 µs, sys: 2 µs, total: 33 µs
Wall time: 37.9 µs
True
sage: %time P.order()
CPU times: user 11 µs, sys: 0 ns, total: 11 µs
Wall time: 16 µs
5326738796327623094747867617954605554069371494832722337612446642054009560026576537626892113
sage: delattr(P, '_order')
sage: %time P.has_order(fac)
CPU times: user 68.6 ms, sys: 17 µs, total: 68.7 ms
Wall time: 68.7 ms
True
sage: %time P.has_order(fac)
CPU times: user 92 µs, sys: 0 ns, total: 92 µs
Wall time: 97.5 µs
True
sage: %time P.order()
CPU times: user 10 µs, sys: 1e+03 ns, total: 11 µs
Wall time: 14.5 µs
5326738796327623094747867617954605554069371494832722337612446642054009560026576537626892113
```

is_divisible_by(*m*)

Return True if there exists a point Q defined over the same field as self such that mQ == self.

INPUT:

- m – a positive integer.

OUTPUT:

(bool) – True if there is a solution, else False.

Warning: This function usually triggers the computation of the m-th division polynomial of the associated elliptic curve, which will be expensive if m is large, though it will be cached for subsequent calls with the same m.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: Q = 5*E(0,0); Q
(-2739/1444 : -77033/54872 : 1)
sage: Q.is_divisible_by(4)
False
sage: Q.is_divisible_by(5)
True
```

A finite field example:

```
sage: E = EllipticCurve(GF(101), [23,34])
sage: E.cardinality().factor()
2 * 53
sage: Set([T.order() for T in E.points()])
{1, 106, 2, 53}
sage: len([T for T in E.points() if T.is_divisible_by(2)])
53
sage: len([T for T in E.points() if T.is_divisible_by(3)])
106
```

is_finite_order()

Return True if this point has finite additive order as an element of the group of points on this curve.

For fields other than number fields and finite fields, this is NotImplemented unless self.is_zero().

EXAMPLES:

```
sage: K.<t> = FractionField(PolynomialRing(QQ,'t'))
sage: E = EllipticCurve([0, 0, 0, -t^2, 0])
sage: P = E(0)
sage: P.has_finite_order()
True
sage: P = E(t,0)
sage: P.has_finite_order()
Traceback (most recent call last):
...
NotImplementedError: Computation of order of a point not implemented
over general fields.
sage: (2*P).is_zero()
True
```

order()

Return the order of this point on the elliptic curve.

If the point is zero, returns 1, otherwise raise a NotImplementedError.

For curves over number fields and finite fields, see below.

Note: *additive_order()* is a synonym for *order()*

EXAMPLES:

```
sage: K.<t> = FractionField(PolynomialRing(QQ,'t'))
sage: E = EllipticCurve([0, 0, 0, -t^2, 0])
sage: P = E(t,0)
sage: P.order()
Traceback (most recent call last):
...
NotImplementedError: Computation of order of a point not implemented
over general fields.
sage: E(0).additive_order()
1
sage: E(0).order() == 1
True
```

plot(***args*)

Plot this point on an elliptic curve.

INPUT:

- `**args` – all arguments get passed directly onto the point plotting function.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: P = E([-1,1])
sage: P.plot(pointsize=30, rgbcolor=(1,0,0))                                #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

point_of_jacobian_of_curve()

Return the point in the Jacobian of the curve.

The Jacobian is the one attached to the projective curve associated with this elliptic curve.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: k.<a> = GF((5,2))
sage: E = EllipticCurve(k,[1,0]); E
Elliptic Curve defined by y^2 = x^3 + x over Finite Field in a of size 5^2
sage: E.order()
32
sage: P = E([a, 2*a + 4])
sage: P
(a : 2*a + 4 : 1)
sage: P.order()
8
sage: p = P.point_of_jacobian_of_curve()
sage: p
[Place (x + 4*a, y + 3*a + 1)]
sage: p.order()
8
sage: Q = 3*P
sage: q = Q.point_of_jacobian_of_curve()
sage: q == 3*p
True
sage: G = p.parent()
sage: G.order()
32
sage: G
Group of rational points of Jacobian over Finite Field in a of size 5^2 (Hess␣
↪model)
sage: J = G.parent(); J
Jacobian of Projective Plane Curve over Finite Field in a of size 5^2
 defined by x^2*y + y^3 - x*z^2 (Hess model)
sage: J.curve() == E.affine_patch(2).projective_closure()
True
```

set_order(*value*, *multiple*, *check=None*)

Set the cached order of this point (i.e., the value of `self._order`) to the given `value`.

Alternatively, when `multiple` is given, this method will first run `order_from_multiple()` to determine the exact order from the given multiple of the point order, then cache the result.

Use this when you know a priori the order of this point, or a multiple of the order, to avoid a potentially expensive order calculation.

INPUT:

- `value` – positive integer
- `multiple` – positive integer; mutually exclusive with `value`

OUTPUT: `None`

EXAMPLES:

This example illustrates basic usage.

```
sage: E = EllipticCurve(GF(7), [0, 1])  # This curve has order 12
sage: G = E(5, 0)
sage: G.set_order(2)
sage: 2*G
(0 : 1 : 0)
sage: G = E(0, 6)
sage: G.set_order(multiple=12)
sage: G._order
3
```

We now give a more interesting case, the NIST-P521 curve. Its order is too big to calculate with Sage, and takes a long time using other packages, so it is very useful here.

```
sage: # needs sage.rings.finite_rings
sage: p = 2^521 - 1
sage: prev_proof_state = proof.arithmetic()
sage: proof.arithmetic(False)  # turn off primality checking
sage: F = GF(p)
sage: A = p - 3
sage: B =␣
↪1093849038073734274511112390766805569936207598951683748994586394495953116150735016013708737
sage: q =␣
↪6864797660130609714981900799081393217269435300143305409394463459185543183397655394245057746
sage: E = EllipticCurve([F(A), F(B)])
sage: G = E.random_point()
sage: G.set_order(q)
sage: G.order() * G  # This takes practically no time.
(0 : 1 : 0)
sage: proof.arithmetic(prev_proof_state) # restore state
```

Using `.set_order()` with a `multiple=` argument can be used to compute a point's order *significantly* faster than calling *order()* if the point is already known to be m-torsion:

```
sage: F.<a> = GF((10007, 23))
sage: E = EllipticCurve(F, [9,9])
sage: n = E.order()
sage: m = 5 * 47 * 139 * 1427 * 2027 * 4831 * 275449 * 29523031
sage: assert m.divides(n)
sage: P = n/m * E.lift_x(6747+a)
sage: assert m * P == 0
sage: P.set_order(multiple=m)    # compute exact order
sage: factor(m // P.order())     # order is now cached
47 * 139
```

The algorithm used internally for this functionality is `order_from_multiple()`. Indeed, simply calling *order()* on `P` would take much longer since factoring `n` is fairly expensive:

```
sage: n == m * 6670822796985115651 * 441770032618665681677 *␣
↪928997347828563460611492 7
True
```

It is an error to pass a `value` equal to 0:

```
sage: # needs sage.rings.finite_rings
sage: E = EllipticCurve(GF(7), [0, 1])  # This curve has order 12
sage: G = E.random_point()
sage: G.set_order(0)
Traceback (most recent call last):
...
ValueError: Value 0 illegal for point order
sage: G.set_order(1000)
Traceback (most recent call last):
...
ValueError: Value 1000 illegal: outside max Hasse bound
```

It is also very likely an error to pass a value which is not the actual order of this point. How unlikely is determined by the factorization of the actual order, and the actual group structure:

```
sage: E = EllipticCurve(GF(7), [0, 1])  # This curve has order 12
sage: G = E(5, 0)   # G has order 2
sage: G.set_order(11)
Traceback (most recent call last):
...
ValueError: Value 11 illegal: 11 * (5 : 0 : 1) is not the identity
```

However, `set_order` can be fooled. For instance, the order can be set to a multiple the actual order:

```
sage: E = EllipticCurve(GF(7), [0, 1])  # This curve has order 12
sage: G = E(5, 0)   # G has order 2
sage: G.set_order(8)
sage: G.order()
8
```

AUTHORS:

- Mariah Lenox (2011-02-16)
- Lorenz Panny (2022): add `multiple=` option

tate_pairing(*Q*, *n*, *k*, *q=None*)

Return Tate pairing of n-torsion point $P = self$ and point Q.

The value returned is $f_{n,P}(Q)^e$ where $f_{n,P}$ is a function with divisor $n[P] - n[O]$.. This is also known as the "modified Tate pairing". It is a well-defined bilinear map.

INPUT:

- `P=self` – Elliptic curve point having order n
- `Q` – Elliptic curve point on same curve as P (can be any order)
- `n` – positive integer: order of P
- `k` – positive integer: embedding degree
- `q` – positive integer: size of base field (the "big" field is $GF(q^k)$. q needs to be set only if its value cannot be deduced.)

OUTPUT:

An n'th root of unity in the base field `self.curve().base_field()`

EXAMPLES:

A simple example, pairing a point with itself, and pairing a point with another rational point:

```
sage: p = 103; A = 1; B = 18; E = EllipticCurve(GF(p), [A, B])
sage: P = E(33, 91); n = P.order(); n
19
sage: k = GF(n)(p).multiplicative_order(); k
6
sage: P.tate_pairing(P, n, k)
1
sage: Q = E(87, 51)
sage: P.tate_pairing(Q, n, k)
1
sage: set_random_seed(35)
sage: P.tate_pairing(P, n, k)
1
```

We now let Q be a point on the same curve as above, but defined over the pairing extension field, and we also demonstrate the bilinearity of the pairing:

```
sage: # needs sage.rings.finite_rings
sage: K.<a> = GF((p,k))
sage: EK = E.base_extend(K); P = EK(P)
sage: Qx = 69*a^5 + 96*a^4 + 22*a^3 + 86*a^2 + 6*a + 35
sage: Qy = 34*a^5 + 24*a^4 + 16*a^3 + 41*a^2 + 4*a + 40
sage: Q = EK(Qx, Qy);
```

Multiply by cofactor so Q has order n:

```
sage: # needs sage.rings.finite_rings
sage: h = 551269674; Q = h*Q
sage: P = EK(P); P.tate_pairing(Q, n, k)
24*a^5 + 34*a^4 + 3*a^3 + 69*a^2 + 86*a + 45
sage: s = Integer(randrange(1,n))
sage: ans1 = (s*P).tate_pairing(Q, n, k)
sage: ans2 = P.tate_pairing(s*Q, n, k)
sage: ans3 = P.tate_pairing(Q, n, k)^s
sage: ans1 == ans2 == ans3
True
sage: (ans1 != 1) and (ans1^n == 1)
True
```

Here is an example of using the Tate pairing to compute the Weil pairing (using the same data as above):

```
sage: # needs sage.rings.finite_rings
sage: e = Integer((p^k-1)/n); e
62844857712
sage: P.weil_pairing(Q, n)^e
94*a^5 + 99*a^4 + 29*a^3 + 45*a^2 + 57*a + 34
sage: P.tate_pairing(Q, n, k) == P._miller_(Q, n)^e
True
sage: Q.tate_pairing(P, n, k) == Q._miller_(P, n)^e
True
sage: P.tate_pairing(Q, n, k)/Q.tate_pairing(P, n, k)
94*a^5 + 99*a^4 + 29*a^3 + 45*a^2 + 57*a + 34
```

An example where we have to pass the base field size (and we again have agreement with the Weil pairing):

```
sage: # needs sage.rings.finite_rings
sage: F.<a> = GF((2,5))
sage: E = EllipticCurve(F, [0,0,1,1,1])
sage: P = E(a^4 + 1, a^3)
sage: Fx.<b> = GF((2,4*5))
sage: Ex = EllipticCurve(Fx,[0,0,1,1,1])
sage: phi = Hom(F, Fx)(F.gen().minpoly().roots(Fx)[0][0])
sage: Px = Ex(phi(P.x()), phi(P.y()))
sage: Qx = Ex(b^19 + b^18 + b^16 + b^12 + b^10 + b^9 + b^8 + b^5 + b^3 + 1,
....:         b^18 + b^13 + b^10 + b^8 + b^5 + b^4 + b^3 + b)
sage: Px.tate_pairing(Qx, n=41, k=4)
Traceback (most recent call last):
...
ValueError: Unexpected field degree: set keyword argument q equal to
the size of the base field (big field is GF(q^4)).
sage: num = Px.tate_pairing(Qx, n=41, k=4, q=32); num
b^19 + b^14 + b^13 + b^12 + b^6 + b^4 + b^3
sage: den = Qx.tate_pairing(Px, n=41, k=4, q=32); den
b^19 + b^17 + b^16 + b^15 + b^14 + b^10 + b^6 + b^2 + 1
sage: e = Integer((32^4-1)/41); e
25575
sage: Px.weil_pairing(Qx, 41)^e == num/den
True
```

An example over a large base field:

```
sage: F = GF(65537^2, modulus=[3,46810,1], name='z2')
sage: F.inject_variables()
Defining z2
sage: E = EllipticCurve(F, [0,1])
sage: P = E(22, 28891)
sage: Q = E(-93, 40438*z2 + 31573)
sage: P.tate_pairing(Q, 7282, 2)
34585*z2 + 4063
```

ALGORITHM:

- pari:elltatepairing computes the non-reduced tate pairing and the exponentiation is handled by Sage using user input for k (and optionally q).

AUTHORS:

- Mariah Lenox (2011-03-07)
- Giacomo Pope (2024): Use of PARI for the non-reduced Tate pairing

weil_pairing(*Q*, *n*, *algorithm=None*)

Compute the Weil pairing of this point with another point Q on the same curve.

INPUT:

- `Q` – another point on the same curve as `self`.
- `n` – an integer n such that $nP = nQ = (0 : 1 : 0)$, where P is `self`.
- `algorithm` (default: `None`) – choices are `pari` and `sage`. PARI is usually significantly faster, but it only works over finite fields. When `None` is given, a suitable algorithm is chosen automatically.

OUTPUT:

An n'th root of unity in the base field of the curve.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: F.<a> = GF((2,5))
sage: E = EllipticCurve(F, [0,0,1,1,1])
sage: P = E(a^4 + 1, a^3)
sage: Fx.<b> = GF((2, 4*5))
sage: Ex = EllipticCurve(Fx, [0,0,1,1,1])
sage: phi = Hom(F, Fx)(F.gen().minpoly().roots(Fx)[0][0])
sage: Px = Ex(phi(P.x()), phi(P.y()))
sage: O = Ex(0)
sage: Qx = Ex(b^19 + b^18 + b^16 + b^12 + b^10 + b^9 + b^8 + b^5 + b^3 + 1,
....:         b^18 + b^13 + b^10 + b^8 + b^5 + b^4 + b^3 + b)
sage: Px.weil_pairing(Qx, 41) == b^19 + b^15 + b^9 + b^8 + b^6 + b^4 + b^3 +␣
↪b^2 + 1
True
sage: Px.weil_pairing(17*Px, 41) == Fx(1)
True
sage: Px.weil_pairing(O, 41) == Fx(1)
True
```

An error is raised if either point is not n-torsion:

```
sage: Px.weil_pairing(O, 40)                                                #␣
↪needs sage.rings.finite_rings
Traceback (most recent call last):
...
ValueError: points must both be n-torsion
```

A larger example (see Issue #4964):

```
sage: # needs sage.rings.finite_rings
sage: P, Q = EllipticCurve(GF((19,4),'a'), [-1,0]).gens()
sage: P.order(), Q.order()
(360, 360)
sage: z = P.weil_pairing(Q, 360)
sage: z.multiplicative_order()
360
```

Another larger example:

```
sage: F = GF(65537^2, modulus=[3,-1,1], name='a')
sage: F.inject_variables()
Defining a
sage: E = EllipticCurve(F, [0,1])
sage: P = E(22, 28891)
sage: Q = E(-93, 2728*a + 64173)
sage: P.weil_pairing(Q, 7282, algorithm='sage')
53278*a + 36700
```

An example over a number field:

```
sage: # needs sage.rings.number_field
sage: E = EllipticCurve('11a1').change_ring(CyclotomicField(5))
sage: P, Q = E.torsion_subgroup().gens()
sage: P, Q = (P.element(), Q.element())
sage: (P.order(), Q.order())
(5, 5)
```

(continues on next page)

(continued from previous page)

```
sage: P.weil_pairing(Q, 5)
zeta5^2
sage: Q.weil_pairing(P, 5)
zeta5^3
```

ALGORITHM:

- For `algorithm='pari'`: pari:ellweilpairing.
- For `algorithm='sage'`: Implemented using Proposition 8 in [Mil2004]. The value 1 is returned for linearly dependent input points. This condition is caught via a `ZeroDivisionError`, since the use of a discrete logarithm test for linear dependence is much too slow for large n.

AUTHORS:

- David Hansen (2009-01-25)
- Lorenz Panny (2022): `algorithm='pari'`

x()

Return the x coordinate of this point, as an element of the base field. If this is the point at infinity, a `ZeroDivisionError` is raised.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: P = E([-1,1])
sage: P.x()
-1
sage: Q = E(0); Q
(0 : 1 : 0)
sage: Q.x()
Traceback (most recent call last):
...
ZeroDivisionError: rational division by zero
```

xy()

Return the x and y coordinates of this point, as a 2-tuple. If this is the point at infinity, a `ZeroDivisionError` is raised.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: P = E([-1,1])
sage: P.xy()
(-1, 1)
sage: Q = E(0); Q
(0 : 1 : 0)
sage: Q.xy()
Traceback (most recent call last):
...
ZeroDivisionError: rational division by zero
```

y()

Return the y coordinate of this point, as an element of the base field. If this is the point at infinity, a `ZeroDivisionError` is raised.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: P = E([-1,1])
sage: P.y()
1
sage: Q = E(0); Q
(0 : 1 : 0)
sage: Q.y()
Traceback (most recent call last):
...
ZeroDivisionError: rational division by zero
```

class sage.schemes.elliptic_curves.ell_point.**EllipticCurvePoint_finite_field**(*curve*, *v*, *check=True*)

Bases: *EllipticCurvePoint_field*

Class for elliptic curve points over finite fields.

additive_order()

Return the order of this point on the elliptic curve.

ALGORITHM: Use PARI function pari:ellorder.

Note: *additive_order()* is a synonym for *order()*

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: k.<a> = GF((5,5))
sage: E = EllipticCurve(k,[2,4]); E
Elliptic Curve defined by y^2 = x^3 + 2*x + 4 over Finite Field in a of size␣
↪5^5
sage: P = E(3*a^4 + 3*a, 2*a + 1)
sage: P.order()
3227
sage: Q = E(0,2)
sage: Q.order()
7
sage: Q.additive_order()
7
```

```
sage: # needs sage.rings.finite_rings
sage: p = next_prime(2^150)
sage: E = EllipticCurve(GF(p), [1,1])
sage: P = E(831623307675610677632782670796608848711856078,
....:       422957860428733667065732925335886382172329 64)
sage: P.order()
1427247692705959881058262545272474300628281448
sage: P.order() == E.cardinality()
True
```

The next example has $j(E) = 0$:

```
sage: # needs sage.rings.finite_rings
sage: p = 33554501
sage: F.<u> = GF((p,2))
```

(continues on next page)

(continued from previous page)

```
sage: E = EllipticCurve(F, [0,1])
sage: E.j_invariant()
0
sage: P = E.random_point()
sage: P.order()  # random
16777251
```

Similarly when $j(E) = 1728$:

```
sage: # needs sage.rings.finite_rings
sage: p = 33554473
sage: F.<u> = GF((p,2))
sage: E = EllipticCurve(F, [1,0])
sage: E.j_invariant()
1728
sage: P = E.random_point()
sage: P.order()  # random
46912611635760
```

discrete_log(Q)

Legacy version of *log()* with its arguments swapped.

Note that this method uses the opposite argument ordering of all other logarithm methods in Sage; see Issue #37150.

EXAMPLES:

```
sage: E = EllipticCurve(j=GF(101)(5))
sage: P, = E.gens()
sage: (2*P).log(P)
2
sage: (2*P).discrete_log(P)
doctest:warning ...
DeprecationWarning: The syntax P.discrete_log(Q) ... Please update your code.␣
↪...
45
sage: P.discrete_log(2*P)
2
```

has_finite_order()

Return `True` if this point has finite additive order as an element of the group of points on this curve.

Since the base field is finite, the answer will always be `True`.

EXAMPLES:

```
sage: E = EllipticCurve(GF(7), [1,3])
sage: P = E.points()[3]
sage: P.has_finite_order()
True
```

log(*base*)

Return the discrete logarithm of this point to the given `base`. In other words, return an integer x such that $xP = Q$ where P is `base` and Q is this point.

A `ValueError` is raised if there is no solution.

ALGORITHM:

To compute the actual logarithm, pari:elllog is called.

However, `elllog()` does not guarantee termination if Q is not a multiple of P, so we first need to check subgroup membership. This is done as follows:

- Let n denote the order of P. First check that nQ equals the point at infinity (and hence the order of Q divides n).
- If the curve order $\#E$ has been cached, check whether $\gcd(n^2, \#E) = n$. If this holds, the curve has cyclic n-torsion, hence all points whose order divides n must be multiples of P and we are done.
- Otherwise (if this test is inconclusive), check that the Weil pairing of P and Q is trivial.

For anomalous curves with $\#E = p$, the *padic_elliptic_logarithm()* function is called.

INPUT:

- `base` (point) – another point on the same curve as `self`.

OUTPUT:

(integer) – The discrete logarithm of Q with respect to P, which is an integer x with $0 \leq x < \mathrm{ord}(P)$ such that $xP = Q$, if one exists.

AUTHORS:

- John Cremona. Adapted to use generic functions 2008-04-05.
- Lorenz Panny (2022): switch to PARI.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: F = GF((3,6),'a')
sage: a = F.gen()
sage: E = EllipticCurve([0,1,1,a,a])
sage: E.cardinality()
762
sage: P = E.gens()[0]
sage: Q = 400*P
sage: Q.log(P)
400
```

order()

Return the order of this point on the elliptic curve.

ALGORITHM: Use PARI function pari:ellorder.

Note: *additive_order()* is a synonym for *order()*

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: k.<a> = GF((5,5))
sage: E = EllipticCurve(k,[2,4]); E
Elliptic Curve defined by y^2 = x^3 + 2*x + 4 over Finite Field in a of size␣
↪5^5
sage: P = E(3*a^4 + 3*a, 2*a + 1)
sage: P.order()
3227
sage: Q = E(0,2)
```

(continues on next page)

(continued from previous page)

```
sage: Q.order()
7
sage: Q.additive_order()
7
```

```
sage: # needs sage.rings.finite_rings
sage: p = next_prime(2^150)
sage: E = EllipticCurve(GF(p), [1,1])
sage: P = E(831623307675610677632782670796608848711856078,
....:       422957860428733667065732925335886382172329 64)
sage: P.order()
1427247692705959881058262545272474300628281448
sage: P.order() == E.cardinality()
True
```

The next example has $j(E) = 0$:

```
sage: # needs sage.rings.finite_rings
sage: p = 33554501
sage: F.<u> = GF((p,2))
sage: E = EllipticCurve(F, [0,1])
sage: E.j_invariant()
0
sage: P = E.random_point()
sage: P.order()  # random
16777251
```

Similarly when $j(E) = 1728$:

```
sage: # needs sage.rings.finite_rings
sage: p = 33554473
sage: F.<u> = GF((p,2))
sage: E = EllipticCurve(F, [1,0])
sage: E.j_invariant()
1728
sage: P = E.random_point()
sage: P.order()  # random
46912611635760
```

padic_elliptic_logarithm(Q, p)

Return the discrete logarithm of Q to base P = `self`, that is, an integer x such that $xP = Q$ only for anomalous curves.

ALGORITHM:

Discrete logarithm computed as in [Sma1999] with a loop to avoid the canonical lift.

INPUT:

- `Q` (point) – another point on the same curve as `self`.
- `p` (integer) – a prime equal to the order of the curve.

OUTPUT:

(integer) – The discrete logarithm of Q with respect to P, which is an integer x with $0 \leq x < \text{ord}(P)$ such that $xP = Q$.

AUTHORS:

- Sylvain Pelissier (2022) based on Samuel Neves code.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: p = 235322474717419
sage: b = 8856682
sage: E = EllipticCurve(GF(p), [0, b])
sage: P = E(200673830421813, 57025307876612)
sage: Q = E(40345734829479, 211738132651297)
sage: x = P.padic_elliptic_logarithm(Q, p)                                  #␣
↪needs sage.rings.padics
sage: x * P == Q                                                            #␣
↪needs sage.rings.padics
True
```

class sage.schemes.elliptic_curves.ell_point.**EllipticCurvePoint_number_field**(*curve*, *v*, *check=True*)

Bases: *EllipticCurvePoint_field*

A point on an elliptic curve over a number field.

Most of the functionality is derived from the parent class *EllipticCurvePoint_field*. In addition we have support for orders, heights, reduction modulo primes, and elliptic logarithms.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E([0,0])
(0 : 0 : 1)
sage: E(0,0)               # brackets are optional
(0 : 0 : 1)
sage: E([GF(5)(0), 0])     # entries are coerced
(0 : 0 : 1)

sage: E(0.000, 0)
(0 : 0 : 1)

sage: E(1,0,0)
Traceback (most recent call last):
...
TypeError: Coordinates [1, 0, 0] do not define a point on
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
```

```
sage: E = EllipticCurve([0,0,1,-1,0])
sage: S = E(QQ); S
Abelian group of points on
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
```

additive_order()

Return the order of this point on the elliptic curve.

If the point has infinite order, returns +Infinity. For curves defined over $\mathbf{Q}$, we call PARI; over other number fields we implement the function here.

Note: *additive_order()* is a synonym for *order()*

EXAMPLES:

```
sage: E = EllipticCurve([0,0,1,-1,0])
sage: P = E([0,0]); P
(0 : 0 : 1)
sage: P.order()
+Infinity
```

```
sage: E = EllipticCurve([0,1])
sage: P = E([-1,0])
sage: P.order()
2
sage: P.additive_order()
2
```

archimedean_local_height(*v=None*, *prec=None*, *weighted=False*)

Compute the local height of self at the archimedean place v.

INPUT:

- `self` – a point on an elliptic curve over a number field K.
- `v` – a real or complex embedding of K, or None (default). If v is a real or complex embedding, return the local height of self at v. If v is None, return the total archimedean contribution to the global height.
- `prec` – integer, or None (default). The precision of the computation. If None, the precision is deduced from v.
- `weighted` – boolean. If False (default), the height is normalised to be invariant under extension of K. If True, return this normalised height multiplied by the local degree if v is a single place, or by the degree of K if v is None.

OUTPUT:

A real number. The normalisation is twice that in Silverman's paper [Sil1988]. Note that this local height depends on the model of the curve.

ALGORITHM:

See [Sil1988], Section 4.

EXAMPLES:

Examples 1, 2, and 3 from [Sil1988]:

```
sage: # needs sage.rings.number_field
sage: K.<a> = QuadraticField(-2)
sage: E = EllipticCurve(K, [0,-1,1,0,0]); E
Elliptic Curve defined by y^2 + y = x^3 + (-1)*x^2 over Number Field
 in a with defining polynomial x^2 + 2 with a = 1.414213562373095?*I
sage: P = E.lift_x(2 + a); P
(a + 2 : -2*a - 2 : 1)
sage: P.archimedean_local_height(K.places(prec=170)[0]) / 2
0.45754773287523276736211210741423654346576029814695

sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<i> = NumberField(x^2 + 1)
sage: E = EllipticCurve(K, [0,0,4,6*i,0]); E
Elliptic Curve defined by y^2 + 4*y = x^3 + 6*i*x
 over Number Field in i with defining polynomial x^2 + 1
```

(continues on next page)

(continued from previous page)

```
sage: P = E((0,0))
sage: P.archimedean_local_height(K.places()[0]) / 2
0.510184995162373

sage: Q = E.lift_x(-9/4); Q                                                    #␣
↪needs sage.rings.number_field
(-9/4 : 27/8*i - 4 : 1)
sage: Q.archimedean_local_height(K.places()[0]) / 2                            #␣
↪needs sage.rings.number_field
0.654445619529600
```

An example over the rational numbers:

```
sage: E = EllipticCurve([0, 0, 0, -36, 0])
sage: P = E([-3, 9])
sage: P.archimedean_local_height()
1.98723816350773
```

Local heights of torsion points can be non-zero (unlike the global height):

```
sage: # needs sage.rings.number_field
sage: K.<i> = QuadraticField(-1)
sage: E = EllipticCurve([0, 0, 0, K(1), 0])
sage: P = E(i, 0)
sage: P.archimedean_local_height()
0.346573590279973
```

elliptic_logarithm(*embedding=None*, *precision=100*, *algorithm='pari'*)

Return the elliptic logarithm of this elliptic curve point.

An embedding of the base field into **R** or **C** (with arbitrary precision) may be given; otherwise the first real embedding is used (with the specified precision) if any, else the first complex embedding.

INPUT:

- `embedding`: an embedding of the base field into **R** or **C**
- `precision`: a positive integer (default 100) setting the number of bits of precision for the computation
- `algorithm`: either `'pari'` (default for real embeddings) to use PARI's pari:ellpointtoz, or `'sage'` for a native implementation. Ignored for complex embeddings.

ALGORITHM:

See [Coh1993] for the case of real embeddings, and Cremona, J.E. and Thongjunthug, T. 2010 for the complex case.

AUTHORS:

- Michael Mardaus (2008-07),
- Tobias Nagel (2008-07) – original version from [Coh1993].
- John Cremona (2008-07) – revision following eclib code.
- John Cremona (2010-03) – implementation for complex embeddings.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: E.discriminant() > 0
True
sage: P = E([-1,1])
sage: P.is_on_identity_component ()
False
sage: P.elliptic_logarithm (precision=96)
0.47934825019021931612953301 01 + 0.985868850775824102211203849...*I
sage: Q = E([3,5])
sage: Q.is_on_identity_component()
True
sage: Q.elliptic_logarithm (precision=96)
1.9311282715425594424885852 20
```

An example with negative discriminant, and a torsion point:

```
sage: E = EllipticCurve('11a1')
sage: E.discriminant() < 0
True
sage: P = E([16,-61])
sage: P.elliptic_logarithm(precision=70)
0.25384186085591068434
sage: E.period_lattice().real_period(prec=70) / P.elliptic_
↪logarithm(precision=70)
5.0000000000000000000
```

A larger example. The default algorithm uses PARI and makes sure the result has the requested precision:

```
sage: E = EllipticCurve([1, 0, 1, -85357462, 303528987048]) #18074g1
sage: P = E([4458713781401/835903744, -64466909836503771/24167649046528, 1])
sage: P.elliptic_logarithm()  # 100 bits
0.27656204014107061464076203097
```

The native algorithm `'sage'` used to have trouble with precision in this example, but no longer:

```
sage: P.elliptic_logarithm(algorithm='sage')  # 100 bits
0.27656204014107061464076203097
```

This shows that the bug reported at Issue #4901 has been fixed:

```
sage: E = EllipticCurve("4390c2")
sage: P = E(683762969925/44944,-565388972095220019/9528128)
sage: P.elliptic_logarithm()
0.00025638725886520225353198932529
sage: P.elliptic_logarithm(precision=64)
0.0002563872588652022 54
sage: P.elliptic_logarithm(precision=65)
0.00025638725886520225 35
sage: P.elliptic_logarithm(precision=128)
0.00025638725886520225353198932528666427412
sage: P.elliptic_logarithm(precision=129)
0.00025638725886520225353198932528666427412
sage: P.elliptic_logarithm(precision=256)
0.
↪0002563872588652022535319893252866642741168388008346370015005142128009610936373
sage: P.elliptic_logarithm(precision=257)
0.
↪00025638725886520225353198932528666427411683880083463700150051421280096109363730
```

Examples over number fields:

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)
sage: embs = K.embeddings(CC)
sage: E = EllipticCurve([0,1,0,a,a])
sage: Ls = [E.period_lattice(e) for e in embs]
sage: [L.real_flag for L in Ls]
[0, 0, -1]
sage: P = E(-1,0)  # order 2
sage: [L.elliptic_logarithm(P) for L in Ls]
[-1.73964256006716 - 1.07861534489191*I,
 -0.363756518406398 - 1.50699412135253*I, 1.90726488608927]

sage: # needs sage.rings.number_field
sage: E = EllipticCurve([-a^2 - a - 1, a^2 + a])
sage: Ls = [E.period_lattice(e) for e in embs]
sage: pts = [E(2*a^2 - a - 1 , -2*a^2 - 2*a + 6 ),
....:        E(-2/3*a^2 - 1/3 , -4/3*a - 2/3 ),
....:        E(5/4*a^2 - 1/2*a , -a^2 - 1/4*a + 9/4 ),
....:        E(2*a^2 + 3*a + 4 , -7*a^2 - 10*a - 12 )]
sage: [[L.elliptic_logarithm(P) for P in pts] for L in Ls]
[[0.250819591818930 - 0.411963479992219*I, -0.290994550611374 - 1.
↪37239400324105*I,
  -0.693473752205595 - 2.45028458830342*I, -0.151659609775291 - 1.
↪48985406505459*I],
 [1.33444787667954 - 1.50889756650544*I, 0.792633734249234 - 0.
↪548467043256610*I,
  0.390154532655013 + 0.529423541805758*I, 0.931968675085317 - 0.
↪431006981443071*I],
 [1.14758249500109 + 0.853389664016075*I, 2.59823462472518 + 0.
↪853389664016075*I,
  1.75372176444709, 0.303069634723001]]
```

```
sage: # needs sage.rings.number_field
sage: K.<i> = QuadraticField(-1)
sage: E = EllipticCurve([0,0,0,9*i-10,21-i])
sage: emb = K.embeddings(CC)[1]
sage: L = E.period_lattice(emb)
sage: P = E(2-i, 4+2*i)
sage: L.elliptic_logarithm(P, prec=100)
0.70448375537782208460499649302 - 0.79246725643650979858266018068*I
```

has_finite_order()

Return `True` iff this point has finite order on the elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([0,0,1,-1,0])
sage: P = E([0,0]); P
(0 : 0 : 1)
sage: P.has_finite_order()
False
```

```
sage: E = EllipticCurve([0,1])
sage: P = E([-1,0])
```

(continues on next page)

(continued from previous page)

```
sage: P.has_finite_order()
True
```

has_good_reduction(*P=None*)

Returns True iff this point has good reduction modulo a prime.

INPUT:

- `P` – a prime of the base_field of the point's curve, or `None` (default)

OUTPUT:

(bool) If a prime P of the base field is specified, returns True iff the point has good reduction at P; otherwise, return true if the point has god reduction at all primes in the support of the discriminant of this model.

EXAMPLES:

```
sage: E = EllipticCurve('990e1')
sage: P = E.gen(0); P
(15 : 51 : 1)
sage: [E.has_good_reduction(p) for p in [2,3,5,7]]
[False, False, False, True]
sage: [P.has_good_reduction(p) for p in [2,3,5,7]]
[True, False, True, True]
sage: [E.tamagawa_exponent(p) for p in [2,3,5,7]]
[2, 2, 1, 1]
sage: [(2*P).has_good_reduction(p) for p in [2,3,5,7]]
[True, True, True, True]
sage: P.has_good_reduction()
False
sage: (2*P).has_good_reduction()
True
sage: (3*P).has_good_reduction()
False
```

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<i> = NumberField(x^2 + 1)
sage: E = EllipticCurve(K, [0,1,0,-160,308])
sage: P = E(26, -120)
sage: E.discriminant().support()
[Fractional ideal (i + 1),
 Fractional ideal (-i - 2),
 Fractional ideal (2*i + 1),
 Fractional ideal (3)]
sage: [E.tamagawa_exponent(p) for p in E.discriminant().support()]
[1, 4, 4, 4]
sage: P.has_good_reduction()
False
sage: (2*P).has_good_reduction()
False
sage: (4*P).has_good_reduction()
True
```

has_infinite_order()

Return True iff this point has infinite order on the elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([0,0,1,-1,0])
sage: P = E([0,0]); P
(0 : 0 : 1)
sage: P.has_infinite_order()
True
```

```
sage: E = EllipticCurve([0,1])
sage: P = E([-1,0])
sage: P.has_infinite_order()
False
```

height (*precision=None*, *normalised=True*, *algorithm='pari'*)

Return the Néron-Tate canonical height of the point.

INPUT:

- `self` – a point on an elliptic curve over a number field K.
- `precision` – positive integer, or None (default). The precision in bits of the result. If None, the default real precision is used.
- `normalised` – boolean. If True (default), the height is normalised to be invariant under extension of K. If False, return this normalised height multiplied by the degree of K.
- `algorithm` – string: either `'pari'` (default) or `'sage'`. If `'pari'` and the base field is **Q**, use the PARI library function; otherwise use the Sage implementation.

OUTPUT:

The rational number 0, or a non-negative real number.

There are two normalisations used in the literature, one of which is double the other. We use the larger of the two, which is the one appropriate for the BSD conjecture. This is consistent with [Cre1997] and double that of [Sil2009].

See Wikipedia article Néron-Tate height.

Note: The correct height to use for the regulator in the BSD formula is the non-normalised height.

EXAMPLES:

```
sage: E = EllipticCurve('11a'); E
Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field
sage: P = E([5,5]); P
(5 : 5 : 1)
sage: P.height()
0
sage: Q = 5*P
sage: Q.height()
0
```

```
sage: E = EllipticCurve('37a'); E
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
sage: P = E([0,0])
sage: P.height()
0.0511114082399688
sage: P.order()
```

(continues on next page)

(continued from previous page)

```
+Infinity
sage: E.regulator()
0.0511114082399688...

sage: def naive_height(P):
....:     return log(RR(max(abs(P[0].numerator()), abs(P[0].denominator()))))
sage: for n in [1..10]:
....:     print(naive_height(2^n*P)/4^n)
0.000000000000000
0.0433216987849966
0.0502949347635656
0.0511006335618645
0.0511007834799612
0.0511013666152466
0.0511034199907743
0.0511106492906471
0.0511114081541082
0.0511114081541180
```

```
sage: E = EllipticCurve('4602a1'); E
Elliptic Curve defined by y^2 + x*y  = x^3 + x^2 - 37746035*x - 89296920339
over Rational Field
sage: x = 77985922458974949246858229195945103471590
sage: y = 19575260230015313702261379022151675961965157108920263594545223
sage: d = 2254020761884782243
sage: E([ x / d^2,  y / d^3 ]).height()
86.7406561381275
```

```
sage: E = EllipticCurve([17, -60, -120, 0, 0]); E
Elliptic Curve defined by y^2 + 17*x*y - 120*y = x^3 - 60*x^2 over Rational␣
↪Field
sage: E([30, -90]).height()
0

sage: E = EllipticCurve('389a1'); E
Elliptic Curve defined by y^2 + y = x^3 + x^2 - 2*x over Rational Field
sage: P, Q = E(-1,1), E(0,-1)
sage: P.height(precision=100)
0.68666708330558658572355210295
sage: (3*Q).height(precision=100)/Q.height(precision=100)
9.0000000000000000000000000000
sage: _.parent()
Real Field with 100 bits of precision
```

Canonical heights over number fields are implemented as well:

```
sage: R.<x> = QQ[]
sage: K.<a> = NumberField(x^3 - 2)                                          #␣
↪needs sage.rings.number_field
sage: E = EllipticCurve([a, 4]); E                                          #␣
↪needs sage.rings.number_field
Elliptic Curve defined by y^2 = x^3 + a*x + 4
 over Number Field in a with defining polynomial x^3 - 2
sage: P = E((0,2))                                                          #␣
↪needs sage.rings.number_field
sage: P.height()                                                            #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.rings.number_field
0.810463096585925
sage: P.height(precision=100)                                                  #␣
↪needs sage.rings.number_field
0.81046309658592536863991810577
sage: P.height(precision=200)                                                  #␣
↪needs sage.rings.number_field
0.810463096585925368639918105768651588961302864171558323780 86
sage: (2*P).height() / P.height()                                              #␣
↪needs sage.rings.number_field
4.00000000000000
sage: (100*P).height() / P.height()                                            #␣
↪needs sage.rings.number_field
10000.0000000000
```

Setting `normalised=False` multiplies the height by the degree of K:

```
sage: E = EllipticCurve('37a')
sage: P = E([0,0])
sage: P.height()
0.0511114082399688
sage: P.height(normalised=False)
0.0511114082399688
sage: K.<z> = CyclotomicField(5)                                               #␣
↪needs sage.rings.number_field
sage: EK = E.change_ring(K)                                                    #␣
↪needs sage.rings.number_field
sage: PK = EK([0,0])                                                           #␣
↪needs sage.rings.number_field
sage: PK.height()                                                              #␣
↪needs sage.rings.number_field
0.0511114082399688
sage: PK.height(normalised=False)                                              #␣
↪needs sage.rings.number_field
0.204445632959875
```

Some consistency checks:

```
sage: E = EllipticCurve('5077a1')
sage: P = E([-2,3,1])
sage: P.height()
1.36857250535393

sage: EK = E.change_ring(QuadraticField(-3,'a'))                               #␣
↪needs sage.rings.number_field
sage: PK = EK([-2,3,1])                                                        #␣
↪needs sage.rings.number_field
sage: PK.height()                                                              #␣
↪needs sage.rings.number_field
1.36857250535393

sage: # needs sage.rings.number_field
sage: K.<i> = NumberField(x^2 + 1)
sage: E = EllipticCurve(K, [0,0,4,6*i,0])
sage: Q = E.lift_x(-9/4); Q
(-9/4 : 27/8*i - 4 : 1)
sage: Q.height()
```

(continues on next page)

(continued from previous page)

```
2.69518560017909
sage: (15*Q).height() / Q.height()
225.000000000000

sage: E = EllipticCurve('37a')
sage: P = E([0,-1])
sage: P.height()
0.0511114082399688
sage: K.<a> = QuadraticField(-7)                                                #␣
↪needs sage.rings.number_field
sage: ED = E.quadratic_twist(-7)                                                #␣
↪needs sage.rings.number_field
sage: Q = E.isomorphism_to(ED.change_ring(K))(P); Q                             #␣
↪needs sage.rings.number_field
(0 : -7/2*a - 1/2 : 1)
sage: Q.height()                                                                #␣
↪needs sage.rings.number_field
0.0511114082399688
sage: Q.height(precision=100)                                                   #␣
↪needs sage.rings.number_field
0.051111408239968840235886099757
```

An example to show that the bug at Issue #5252 is fixed:

```
sage: E = EllipticCurve([1, -1, 1, -2063758701246626370773726978,␣
↪32838647793306133075103747085833809114881])
sage: P = E([-30987785091199, 258909576181697016447])
sage: P.height()
25.8603170675462
sage: P.height(precision=100)
25.860317067546190743868840741
sage: P.height(precision=250)
25.860317067546190743868840740735110323098872903844416215577171041783572513
sage: P.height(precision=500)
25.
↪860317067546190743868840740735110323098872903844416215577171041783572512955113057088981328

sage: P.height(precision=100) == P.non_archimedean_local_height(prec=100)+P.
↪archimedean_local_height(prec=100)
True
```

An example to show that the bug at Issue #8319 is fixed (correct height when the curve is not minimal):

```
sage: E = EllipticCurve([-55804723294461149528055058045934980800000,-
↪15733973378536811038297368990353605478770049722330636800000000])
sage: xP =␣
↪204885147732879546487576840131729064308289385547094673627174585676211859152978311600/
↪23625501907057948132262217188983681204856907657753178415430361
sage: P = E.lift_x(xP)
sage: P.height()
157.432598516754
sage: Q = 2*P
sage: Q.height() # long time (4s)
629.730394067016
sage: Q.height()-4*P.height() # long time
0.000000000000000
```

An example to show that the bug at Issue #12509 is fixed (precision issues):

```
sage: # needs sage.rings.number_field
sage: x = polygen(QQ)
sage: K.<a> = NumberField(x^2 - x - 1)
sage: v = [0, a + 1, 1, 28665*a - 46382, 2797026*a - 4525688]
sage: E = EllipticCurve(v)
sage: P = E([72*a - 509/5,  -682/25*a - 434/25])
sage: P.height()
1.38877711688727
sage: (2*P).height()/P.height()
4.00000000000000
sage: (2*P).height(precision=100)/P.height(precision=100)
4.0000000000000000000000000000
sage: (2*P).height(precision=1000)/P.height(precision=1000)
4.
↪0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

This shows that the bug reported at Issue #13951 has been fixed:

```
sage: E = EllipticCurve([0,17])
sage: P1 = E(2,5)
sage: P1.height()
1.06248137652528
sage: F = E.change_ring(QuadraticField(-3, 'a'))                        #␣
↪needs sage.rings.number_field
sage: P2 = F([2,5])                                                     #␣
↪needs sage.rings.number_field
sage: P2.height()                                                       #␣
↪needs sage.rings.number_field
1.06248137652528
```

is_on_identity_component (*embedding=None*)

Return True iff this point is on the identity component of its curve with respect to a given (real or complex) embedding.

INPUT:

- `self` – a point on a curve over any ordered field (e.g. **Q**)
- `embedding` – an embedding from the base_field of the point's curve into **R** or **C**; if `None` (the default) it uses the first embedding of the base_field into **R** if any, else the first embedding into **C**.

OUTPUT:

(bool) – `True` iff the point is on the identity component of the curve. (If the point is zero then the result is True.)

EXAMPLES:

For $K = \mathbf{Q}$ there is no need to specify an embedding:

```
sage: E = EllipticCurve('5077a1')
sage: [E.lift_x(x).is_on_identity_component() for x in srange(-3,5)]
[False, False, False, False, False, True, True, True]
```

An example over a field with two real embeddings:

```
sage: # needs sage.rings.number_field
sage: L.<a> = QuadraticField(2)
```

(continues on next page)

(continued from previous page)

```
sage: E = EllipticCurve(L, [0,1,0,a,a])
sage: P = E(-1,0)
sage: [P.is_on_identity_component(e) for e in L.embeddings(RR)]
[False, True]
```

We can check this as follows:

```
sage: # needs sage.rings.number_field
sage: [e(E.discriminant()) > 0 for e in L.embeddings(RR)]
[True, False]
sage: e = L.embeddings(RR)[0]
sage: E1 = EllipticCurve(RR, [e(ai) for ai in E.ainvs()])
sage: e1, e2, e3 = E1.two_division_polynomial().roots(RR,
....:                                                 multiplicities=False)
sage: e1 < e2 < e3 and e(P[0]) < e3
True
```

non_archimedean_local_height(*v=None*, *prec=None*, *weighted=False*, *is_minimal=None*)

Compute the local height of `self` at the non-archimedean place v.

INPUT:

- `self` – a point on an elliptic curve over a number field K.
- `v` – a non-archimedean place of K, or `None` (default). If v is a non-archimedean place, return the local height of self at v. If v is `None`, return the total non-archimedean contribution to the global height.
- `prec` – integer, or `None` (default). The precision of the computation. If `None`, the height is returned symbolically.
- `weighted` – boolean. If `False` (default), the height is normalised to be invariant under extension of K. If `True`, return this normalised height multiplied by the local degree if v is a single place, or by the degree of K if v is None.

OUTPUT:

A real number. The normalisation is twice that in Silverman's paper [Sil1988]. Note that this local height depends on the model of the curve.

ALGORITHM:

See [Sil1988], Section 5.

EXAMPLES:

Examples 2 and 3 from [Sil1988]:

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<i> = NumberField(x^2 + 1)
sage: E = EllipticCurve(K, [0,0,4,6*i,0]); E
Elliptic Curve defined by y^2 + 4*y = x^3 + 6*i*x
 over Number Field in i with defining polynomial x^2 + 1
sage: P = E((0,0))
sage: P.non_archimedean_local_height(K.ideal(i+1))
-1/2*log(2)
sage: P.non_archimedean_local_height(K.ideal(3))
0
sage: P.non_archimedean_local_height(K.ideal(1-2*i))
0
```

(continues on next page)

(continued from previous page)

```
sage: # needs sage.rings.number_field
sage: Q = E.lift_x(-9/4); Q
(-9/4 : 27/8*i - 4 : 1)
sage: Q.non_archimedean_local_height(K.ideal(1+i))
2*log(2)
sage: Q.non_archimedean_local_height(K.ideal(3))
0
sage: Q.non_archimedean_local_height(K.ideal(1-2*i))
0
sage: Q.non_archimedean_local_height()
2*log(2)
```

An example over the rational numbers:

```
sage: E = EllipticCurve([0, 0, 0, -36, 0])
sage: P = E([-3, 9])
sage: P.non_archimedean_local_height()
-log(3)
```

Local heights of torsion points can be non-zero (unlike the global height):

```
sage: # needs sage.rings.number_field
sage: K.<i> = QuadraticField(-1)
sage: E = EllipticCurve([0, 0, 0, K(1), 0])
sage: P = E(i, 0)
sage: P.non_archimedean_local_height()
-1/2*log(2)
```

order()

Return the order of this point on the elliptic curve.

If the point has infinite order, returns +Infinity. For curves defined over **Q**, we call PARI; over other number fields we implement the function here.

Note: *additive_order()* is a synonym for *order()*

EXAMPLES:

```
sage: E = EllipticCurve([0,0,1,-1,0])
sage: P = E([0,0]); P
(0 : 0 : 1)
sage: P.order()
+Infinity
```

```
sage: E = EllipticCurve([0,1])
sage: P = E([-1,0])
sage: P.order()
2
sage: P.additive_order()
2
```

padic_elliptic_logarithm(*p*, *absprec=20*)

Computes the p-adic elliptic logarithm of this point.

INPUT:

- p – integer: a prime absprec – integer (default: 20): the initial p-adic absolute precision of the computation

OUTPUT:

The p-adic elliptic logarithm of self, with precision absprec.

AUTHORS:

- Tobias Nagel
- Michael Mardaus
- John Cremona

ALGORITHM:

For points in the formal group (i.e. not integral at p) we take the log() function from the formal groups module and evaluate it at $-x/y$. Otherwise we first multiply the point to get into the formal group, and divide the result afterwards.

Todo: See comments at Issue #4805. Currently the absolute precision of the result may be less than the given value of absprec, and error-handling is imperfect.

EXAMPLES:

```
sage: E = EllipticCurve([0,1,1,-2,0])
sage: E(0).padic_elliptic_logarithm(3)                                          #␣
↪needs sage.rings.padics
0
sage: P = E(0, 0)                                                               #␣
↪needs sage.rings.padics
sage: P.padic_elliptic_logarithm(3)                                             #␣
↪needs sage.rings.padics
2 + 2*3 + 3^3 + 2*3^7 + 3^8 + 3^9 + 3^11 + 3^15 + 2*3^17 + 3^18 + O(3^19)
sage: P.padic_elliptic_logarithm(3).lift()                                      #␣
↪needs sage.rings.padics
660257522
sage: P = E(-11/9, 28/27)                                                       #␣
↪needs sage.rings.padics
sage: [(2*P).padic_elliptic_logarithm(p)/P.padic_elliptic_logarithm(p) for p␣
↪in prime_range(20)]  # long time, needs sage.rings.padics
[2 + O(2^19), 2 + O(3^20), 2 + O(5^19), 2 + O(7^19), 2 + O(11^19), 2 + O(13^
↪19), 2 + O(17^19), 2 + O(19^19)]
sage: [(3*P).padic_elliptic_logarithm(p)/P.padic_elliptic_logarithm(p) for p␣
↪in prime_range(12)]  # long time, needs sage.rings.padics
[1 + 2 + O(2^19), 3 + 3^20 + O(3^21), 3 + O(5^19), 3 + O(7^19), 3 + O(11^19)]
sage: [(5*P).padic_elliptic_logarithm(p)/P.padic_elliptic_logarithm(p) for p␣
↪in prime_range(12)]  # long time, needs sage.rings.padics
[1 + 2^2 + O(2^19), 2 + 3 + O(3^20), 5 + O(5^19), 5 + O(7^19), 5 + O(11^19)]
```

An example which arose during reviewing Issue #4741:

```
sage: E = EllipticCurve('794a1')
sage: P = E(-1,2)
sage: P.padic_elliptic_logarithm(2)  # default precision=20                     #␣
↪needs sage.rings.padics
2^4 + 2^5 + 2^6 + 2^8 + 2^9 + 2^13 + 2^14 + 2^15 + O(2^16)
sage: P.padic_elliptic_logarithm(2, absprec=30)                                 #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.rings.padics
2^4 + 2^5 + 2^6 + 2^8 + 2^9 + 2^13 + 2^14 + 2^15 + 2^22 + 2^23 + 2^24 + O(2^
↪26)
sage: P.padic_elliptic_logarithm(2, absprec=40)                              #␣
↪needs sage.rings.padics
2^4 + 2^5 + 2^6 + 2^8 + 2^9 + 2^13 + 2^14 + 2^15 + 2^22 + 2^23 + 2^24
+ 2^28 + 2^29 + 2^31 + 2^34 + O(2^35)
```

reduction(p)

This finds the reduction of a point P on the elliptic curve modulo the prime p.

INPUT:

- `self` – A point on an elliptic curve.
- `p` – a prime number

OUTPUT:

The point reduced to be a point on the elliptic curve modulo p.

EXAMPLES:

```
sage: E = EllipticCurve([1,2,3,4,0])
sage: P = E(0,0)
sage: P.reduction(5)
(0 : 0 : 1)
sage: Q = E(98,931)
sage: Q.reduction(5)
(3 : 1 : 1)
sage: Q.reduction(5).curve() == E.reduction(5)
True
```

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: F.<a> = NumberField(x^2 + 5)
sage: E = EllipticCurve(F, [1,2,3,4,0])
sage: Q = E(98, 931)
sage: Q.reduction(a)
(3 : 1 : 1)
sage: Q.reduction(11)
(10 : 7 : 1)
```

```
sage: # needs sage.rings.number_field
sage: F.<a> = NumberField(x^3 + x^2 + 1)
sage: E = EllipticCurve(F, [a,2])
sage: P = E(a, 1)
sage: P.reduction(F.ideal(5))
(abar : 1 : 1)
sage: P.reduction(F.ideal(a^2 - 4*a - 2))
(abar : 1 : 1)
```

CHAPTER

FOUR

ELLIPTIC CURVES OVER A GENERAL RING

Sage defines an elliptic curve over a ring R as a *Weierstrass Model* with five coefficients $[a_1, a_2, a_3, a_4, a_6]$ in R given by

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

Note that the (usual) scheme-theoretic definition of an elliptic curve over R would require the discriminant to be a unit in R; Sage only imposes that the discriminant is non-zero. Also note that in Magma, "Weierstrass Model" refers to a model with $a_1 = a_2 = a_3 = 0$, which is called *Short Weierstrass Model* in Sage; these do not always exist in characteristics 2 and 3.

EXAMPLES:

We construct an elliptic curve over an elaborate base ring:

```
sage: p, a, b = 97, 1, 3
sage: R.<u> = GF(p)[]
sage: S.<v> = R[]
sage: T = S.fraction_field()
sage: E = EllipticCurve(T, [a, b]); E
Elliptic Curve defined by y^2  = x^3 + x + 3 over Fraction Field of Univariate
Polynomial Ring in v over Univariate Polynomial Ring in u over Finite Field of size 97
sage: latex(E)
y^2  = x^{3} + x + 3
```

AUTHORS:

- William Stein (2005): Initial version
- Robert Bradshaw et al….
- John Cremona (2008-01): isomorphisms, automorphisms and twists in all characteristics
- Julian Rueth (2014-04-11): improved caching
- Lorenz Panny (2022-04-14): added `.montgomery_model()`

class sage.schemes.elliptic_curves.ell_generic.**EllipticCurve_generic**(*K*, *ainvs*, *category=None*)

Bases: `WithEqualityById`, `ProjectivePlaneCurve`

Elliptic curve over a generic base ring.

EXAMPLES:

```
sage: E = EllipticCurve([1,2,3/4,7,19]); E
Elliptic Curve defined by y^2 + x*y + 3/4*y = x^3 + 2*x^2 + 7*x + 19 over␣
↪Rational Field
sage: loads(E.dumps()) == E
```

(continues on next page)

(continued from previous page)

```
True
sage: E = EllipticCurve([1,3])
sage: P = E([-1,1,1])
sage: -5*P
(179051/80089 : -91814227/22665187 : 1)
```

a1()

Return the a_1 invariant of this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([1,2,3,4,6])
sage: E.a1()
1
```

a2()

Return the a_2 invariant of this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([1,2,3,4,6])
sage: E.a2()
2
```

a3()

Return the a_3 invariant of this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([1,2,3,4,6])
sage: E.a3()
3
```

a4()

Return the a_4 invariant of this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([1,2,3,4,6])
sage: E.a4()
4
```

a6()

Return the a_6 invariant of this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([1,2,3,4,6])
sage: E.a6()
6
```

a_invariants()

The a-invariants of this elliptic curve, as a tuple.

OUTPUT:

(tuple) - a 5-tuple of the a-invariants of this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([1,2,3,4,5])
sage: E.a_invariants()
(1, 2, 3, 4, 5)

sage: E = EllipticCurve([0,1]); E
Elliptic Curve defined by y^2 = x^3 + 1 over Rational Field
sage: E.a_invariants()
(0, 0, 0, 0, 1)

sage: E = EllipticCurve([GF(7)(3),5])
sage: E.a_invariants()
(0, 0, 0, 3, 5)
```

ainvs()

The a-invariants of this elliptic curve, as a tuple.

OUTPUT:

(tuple) - a 5-tuple of the a-invariants of this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([1,2,3,4,5])
sage: E.a_invariants()
(1, 2, 3, 4, 5)

sage: E = EllipticCurve([0,1]); E
Elliptic Curve defined by y^2 = x^3 + 1 over Rational Field
sage: E.a_invariants()
(0, 0, 0, 0, 1)

sage: E = EllipticCurve([GF(7)(3),5])
sage: E.a_invariants()
(0, 0, 0, 3, 5)
```

automorphisms(*field=None*)

Return the set of isomorphisms from `self` to itself (as a list).

The identity and negation morphisms are guaranteed to appear as the first and second entry of the returned list.

INPUT:

- `field` (default `None`) – a field into which the coefficients of the curve may be coerced (by default, uses the base field of the curve).

OUTPUT:

(list) A list of `WeierstrassIsomorphism` objects consisting of all the isomorphisms from the curve `self` to itself defined over `field`.

EXAMPLES:

```
sage: E = EllipticCurve_from_j(QQ(0))  # a curve with j=0 over QQ
sage: E.automorphisms()
[Elliptic-curve endomorphism of Elliptic Curve defined by y^2 + y = x^3
  over Rational Field
   Via:  (u,r,s,t) = (1, 0, 0, 0),
```

(continues on next page)

(continued from previous page)

```
 Elliptic-curve endomorphism of Elliptic Curve defined by y^2 + y = x^3
  over Rational Field
   Via:  (u,r,s,t) = (-1, 0, 0, -1)]
```

We can also find automorphisms defined over extension fields:

```
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^2 + 3)  # adjoin roots of unity                #␣
↪needs sage.rings.number_field
sage: E.automorphisms(K)                                                    #␣
↪needs sage.rings.number_field
[Elliptic-curve endomorphism of Elliptic Curve defined by y^2 + y = x^3
  over Number Field in a with defining polynomial x^2 + 3
   Via:  (u,r,s,t) = (1, 0, 0, 0),
 Elliptic-curve endomorphism of Elliptic Curve defined by y^2 + y = x^3
  over Number Field in a with defining polynomial x^2 + 3
   Via:  (u,r,s,t) = (-1, 0, 0, -1),
 Elliptic-curve endomorphism of Elliptic Curve defined by y^2 + y = x^3
  over Number Field in a with defining polynomial x^2 + 3
   Via:  (u,r,s,t) = (-1/2*a - 1/2, 0, 0, 0),
 Elliptic-curve endomorphism of Elliptic Curve defined by y^2 + y = x^3
  over Number Field in a with defining polynomial x^2 + 3
   Via:  (u,r,s,t) = (1/2*a + 1/2, 0, 0, -1),
 Elliptic-curve endomorphism of Elliptic Curve defined by y^2 + y = x^3
  over Number Field in a with defining polynomial x^2 + 3
   Via:  (u,r,s,t) = (1/2*a - 1/2, 0, 0, 0),
 Elliptic-curve endomorphism of Elliptic Curve defined by y^2 + y = x^3
  over Number Field in a with defining polynomial x^2 + 3
   Via:  (u,r,s,t) = (-1/2*a + 1/2, 0, 0, -1)]
```

```
sage: [len(EllipticCurve_from_j(GF(q,'a')(0)).automorphisms())              #␣
↪needs sage.rings.finite_rings
....:  for q in [2,4,3,9,5,25,7,49]]
[2, 24, 2, 12, 2, 6, 6, 6]
```

b2()

Return the b_2 invariant of this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([1,2,3,4,5])
sage: E.b2()
9
```

b4()

Return the b_4 invariant of this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([1,2,3,4,5])
sage: E.b4()
11
```

b6()

Return the b_6 invariant of this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([1,2,3,4,5])
sage: E.b6()
29
```

b8()

Return the b_8 invariant of this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([1,2,3,4,5])
sage: E.b8()
35
```

b_invariants()

Return the b-invariants of this elliptic curve, as a tuple.

OUTPUT:

(tuple) - a 4-tuple of the b-invariants of this elliptic curve.

This method is cached.

EXAMPLES:

```
sage: E = EllipticCurve([0, -1, 1, -10, -20])
sage: E.b_invariants()
(-4, -20, -79, -21)

sage: E = EllipticCurve([-4,0])
sage: E.b_invariants()
(0, -8, 0, -16)

sage: E = EllipticCurve([1,2,3,4,5])
sage: E.b_invariants()
(9, 11, 29, 35)
sage: E.b2()
9
sage: E.b4()
11
sage: E.b6()
29
sage: E.b8()
35
```

ALGORITHM:

These are simple functions of the a-invariants.

AUTHORS:

- William Stein (2005-04-25)

base_extend(R)

Return the base extension of `self` to R.

INPUT:

- `R` – either a ring into which the a-invariants of `self` may be converted, or a morphism which may be applied to them.

OUTPUT:

An elliptic curve over the new ring whose a-invariants are the images of the a-invariants of `self`.

EXAMPLES:

```
sage: E = EllipticCurve(GF(5), [1,1]); E
Elliptic Curve defined by y^2 = x^3 + x + 1 over Finite Field of size 5
sage: E1 = E.base_extend(GF(125,'a')); E1                                   #␣
↪needs sage.rings.finite_rings
Elliptic Curve defined by y^2 = x^3 + x + 1 over Finite Field in a of size 5^3
```

base_ring()

Return the base ring of the elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve(GF(49, 'a'), [3,5])                                 #␣
↪needs sage.rings.finite_rings
sage: E.base_ring()                                                         #␣
↪needs sage.rings.finite_rings
Finite Field in a of size 7^2
```

```
sage: E = EllipticCurve([1,1])
sage: E.base_ring()
Rational Field
```

```
sage: E = EllipticCurve(ZZ, [3,5])
sage: E.base_ring()
Integer Ring
```

c4()

Return the c_4 invariant of this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([0, -1, 1, -10, -20])
sage: E.c4()
496
```

c6()

Return the c_6 invariant of this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([0, -1, 1, -10, -20])
sage: E.c6()
20008
```

c_invariants()

Return the c-invariants of this elliptic curve, as a tuple.

This method is cached.

OUTPUT:

(tuple) - a 2-tuple of the c-invariants of the elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([0, -1, 1, -10, -20])
sage: E.c_invariants()
(496, 20008)

sage: E = EllipticCurve([-4,0])
sage: E.c_invariants()
(192, 0)
```

ALGORITHM:

These are simple functions of the a-invariants.

AUTHORS:

- William Stein (2005-04-25)

change_ring(*R*)

Return the base change of `self` to R.

This has the same effect as `self.base_extend(R)`.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: F2 = GF(5^2,'a'); a = F2.gen()
sage: F4 = GF(5^4,'b'); b = F4.gen()
sage: roots = a.charpoly().roots(ring=F4, multiplicities=False)
sage: h = F2.hom([roots[0]], F4)
sage: E = EllipticCurve(F2, [1,a]); E
Elliptic Curve defined by y^2 = x^3 + x + a
over Finite Field in a of size 5^2
sage: E.change_ring(h)
Elliptic Curve defined by y^2 = x^3 + x + (4*b^3+4*b^2+4*b+3)
over Finite Field in b of size 5^4
```

change_weierstrass_model(**urst*)

Return a new Weierstrass model of `self` under the standard transformation (u, r, s, t)

$$(x, y) \mapsto (x', y') = (u^2x + r, u^3y + su^2x + t).$$

EXAMPLES:

```
sage: E = EllipticCurve('15a')
sage: F1 = E.change_weierstrass_model([1/2,0,0,0]); F1
Elliptic Curve defined by y^2 + 2*x*y + 8*y = x^3 + 4*x^2 - 160*x - 640
over Rational Field
sage: F2 = E.change_weierstrass_model([7,2,1/3,5]); F2
Elliptic Curve defined by
y^2 + 5/21*x*y + 13/343*y = x^3 + 59/441*x^2 - 10/7203*x - 58/117649
over Rational Field
sage: F1.is_isomorphic(F2)
True
```

discriminant()

Return the discriminant of this elliptic curve.

This method is cached.

EXAMPLES:

```
sage: E = EllipticCurve([0,0,1,-1,0])
sage: E.discriminant()
37

sage: E = EllipticCurve([0, -1, 1, -10, -20])
sage: E.discriminant()
-161051

sage: E = EllipticCurve([GF(7)(2),1])
sage: E.discriminant()
1
```

division_polynomial(*m*, *x=None*, *two_torsion_multiplicity=2*, *force_evaluate=None*)

Return the m^{th} division polynomial of this elliptic curve evaluated at x.

The division polynomial is cached if x is `None`.

INPUT:

- `m` – positive integer.
- `x` – optional ring element to use as the x variable. If x is `None` (omitted), then a new polynomial ring will be constructed over the base ring of the elliptic curve, and its generator will be used as x. Note that x does not need to be a generator of a polynomial ring; any ring element works. This permits fast calculation of the torsion polynomial *evaluated* on any element of a ring.
- `two_torsion_multiplicity` – 0, 1, or 2

  If 0: For even m when x is `None`, a univariate polynomial over the base ring of the curve is returned, which omits factors whose roots are the x-coordinates of the 2-torsion points. When x is not `None`, the evaluation of such a polynomial at x is returned.

  If 2: For even m when x is `None`, a univariate polynomial over the base ring of the curve is returned, which includes a factor of degree 3 whose roots are the x-coordinates of the 2-torsion points. Similarly, when x is not `None`, the evaluation of such a polynomial at x is returned.

  If 1: For even m when x is `None`, a bivariate polynomial over the base ring of the curve is returned, which includes a factor $2y + a_1x + a_3$ having simple zeros at the 2-torsion points. When x is not `None`, it should be a tuple of length 2, and the evaluation of such a polynomial at x is returned.

- `force_evaluate` (optional) – 0, 1, or 2

  By default, this method makes use of previously cached generic division polynomials to compute the value of the polynomial at a given element x whenever it appears beneficial to do so. Explicitly setting this flag overrides the default behavior.

  Note that the complexity of evaluating a generic division polynomial scales much worse than that of computing the value at a point directly (using the recursive formulas), hence setting this flag can be detrimental to performance.

  If 0: Do not use cached generic division polynomials.

  If 1: If the generic division polynomial for this m has been cached before, evaluate it at x to compute the result.

  If 2: Compute the value at x by evaluating the generic division polynomial. If the generic m-division polynomial has not yet been cached, compute and cache it first.

EXAMPLES:

```
sage: E = EllipticCurve([0,0,1,-1,0])
sage: E.division_polynomial(1)
1
sage: E.division_polynomial(2, two_torsion_multiplicity=0)
1
sage: E.division_polynomial(2, two_torsion_multiplicity=1)
2*y + 1
sage: E.division_polynomial(2, two_torsion_multiplicity=2)
4*x^3 - 4*x + 1
sage: E.division_polynomial(2)
4*x^3 - 4*x + 1
sage: [E.division_polynomial(3, two_torsion_multiplicity=i) for i in range(3)]
[3*x^4 - 6*x^2 + 3*x - 1, 3*x^4 - 6*x^2 + 3*x - 1, 3*x^4 - 6*x^2 + 3*x - 1]
sage: [type(E.division_polynomial(3, two_torsion_multiplicity=i)) for i in␣
↪range(3)]
[<... 'sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_
↪flint'>,
 <... 'sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_
↪libsingular'>,
 <... 'sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_
↪flint'>]
```

```
sage: E = EllipticCurve([0, -1, 1, -10, -20])
sage: R.<z> = PolynomialRing(QQ)
sage: E.division_polynomial(4, z, 0)
2*z^6 - 4*z^5 - 100*z^4 - 790*z^3 - 210*z^2 - 1496*z - 5821
sage: E.division_polynomial(4, z)
8*z^9 - 24*z^8 - 464*z^7 - 2758*z^6 + 6636*z^5 + 34356*z^4
 + 53510*z^3 + 99714*z^2 + 351024*z + 459859
```

This does not work, since when two_torsion_multiplicity is 1, we compute a bivariate polynomial, and must evaluate at a tuple of length 2:

```
sage: E.division_polynomial(4,z,1)
Traceback (most recent call last):
...
ValueError: x should be a tuple of length 2 (or None)
when two_torsion_multiplicity is 1
sage: R.<z,w> = PolynomialRing(QQ, 2)
sage: E.division_polynomial(4, (z,w), 1).factor()
(2*w + 1) * (2*z^6 - 4*z^5 - 100*z^4 - 790*z^3 - 210*z^2 - 1496*z - 5821)
```

We can also evaluate this bivariate polynomial at a point:

```
sage: P = E(5,5)
sage: E.division_polynomial(4,P,two_torsion_multiplicity=1)
-1771561
```

division_polynomial_0(*n*, *x=None*)

Return the n^{th} torsion (division) polynomial, without the 2-torsion factor if n is even, as a polynomial in x.

These are the polynomials g_n defined in [MT1991], but with the sign flipped for even n, so that the leading coefficient is always positive.

Note: This function is intended for internal use; users should use *division_polynomial()*.

See also:

- *division_polynomial()*
- _multiple_x_numerator()
- _multiple_x_denominator()

INPUT:

- n – positive integer, or the special values −1 and −2 which mean $B_6 = (2y + a_1x + a_3)^2$ and B_6^2 respectively (in the notation of [MT1991]); or a list of integers.
- x – a ring element to use as the "x" variable or None (default: None). If None, then a new polynomial ring will be constructed over the base ring of the elliptic curve, and its generator will be used as x. Note that x does not need to be a generator of a polynomial ring; any ring element is ok. This permits fast calculation of the torsion polynomial *evaluated* on any element of a ring.

ALGORITHM:

Recursion described in [MT1991]. The recursive formulae are evaluated $O(\log^2 n)$ times.

AUTHORS:

- David Harvey (2006-09-24): initial version
- John Cremona (2008-08-26): unified division polynomial code

EXAMPLES:

```
sage: E = EllipticCurve("37a")
sage: E.division_polynomial_0(1)
1
sage: E.division_polynomial_0(2)
1
sage: E.division_polynomial_0(3)
3*x^4 - 6*x^2 + 3*x - 1
sage: E.division_polynomial_0(4)
2*x^6 - 10*x^4 + 10*x^3 - 10*x^2 + 2*x + 1
sage: E.division_polynomial_0(5)
5*x^12 - 62*x^10 + 95*x^9 - 105*x^8 - 60*x^7 + 285*x^6 - 174*x^5 - 5*x^4 -␣
↪5*x^3 + 35*x^2 - 15*x + 2
sage: E.division_polynomial_0(6)
3*x^16 - 72*x^14 + 168*x^13 - 364*x^12 + 1120*x^10 - 1144*x^9 + 300*x^8 -␣
↪540*x^7 + 1120*x^6 - 588*x^5 - 133*x^4 + 252*x^3 - 114*x^2 + 22*x - 1
sage: E.division_polynomial_0(7)
7*x^24 - 308*x^22 + 986*x^21 - 2954*x^20 + 28*x^19 + 17171*x^18 - 23142*x^17␣
↪+ 511*x^16 - 5012*x^15 + 43804*x^14 - 7140*x^13 - 96950*x^12 + 111356*x^11 -
↪ 19516*x^10 - 49707*x^9 + 40054*x^8 - 124*x^7 - 18382*x^6 + 13342*x^5 -␣
↪4816*x^4 + 1099*x^3 - 210*x^2 + 35*x - 3
sage: E.division_polynomial_0(8)
4*x^30 - 292*x^28 + 1252*x^27 - 5436*x^26 + 2340*x^25 + 39834*x^24 - 79560*x^
↪23 + 51432*x^22 - 142896*x^21 + 451596*x^20 - 212040*x^19 - 1005316*x^18 +␣
↪1726416*x^17 - 671160*x^16 - 954924*x^15 + 1119552*x^14 + 313308*x^13 -␣
↪1502818*x^12 + 1189908*x^11 - 160152*x^10 - 399176*x^9 + 386142*x^8 -␣
↪220128*x^7 + 99558*x^6 - 33528*x^5 + 6042*x^4 + 310*x^3 - 406*x^2 + 78*x - 5
```

```
sage: E.division_polynomial_0(18) % E.division_polynomial_0(6) == 0
True
```

An example to illustrate the relationship with torsion points:

```
sage: F = GF(11)
sage: E = EllipticCurve(F, [0, 2]); E
Elliptic Curve defined by y^2  = x^3 + 2 over Finite Field of size 11
sage: f = E.division_polynomial_0(5); f
5*x^12 + x^9 + 8*x^6 + 4*x^3 + 7
sage: f.factor()
(5) * (x^2 + 5) * (x^2 + 2*x + 5) * (x^2 + 5*x + 7)
 * (x^2 + 7*x + 7) * (x^2 + 9*x + 5) * (x^2 + 10*x + 7)
```

This indicates that the x-coordinates of all the 5-torsion points of E are in $\mathbf{F}_{11^2}$, and therefore the y-coordinates are in $\mathbf{F}_{11^4}$:

```
sage: # needs sage.rings.finite_rings
sage: K = GF(11^4, 'a')
sage: X = E.change_ring(K)
sage: f = X.division_polynomial_0(5)
sage: x_coords = f.roots(multiplicities=False); x_coords
[10*a^3 + 4*a^2 + 5*a + 6,
 9*a^3 + 8*a^2 + 10*a + 8,
 8*a^3 + a^2 + 4*a + 10,
 8*a^3 + a^2 + 4*a + 8,
 8*a^3 + a^2 + 4*a + 4,
 6*a^3 + 9*a^2 + 3*a + 4,
 5*a^3 + 2*a^2 + 8*a + 7,
 3*a^3 + 10*a^2 + 7*a + 8,
 3*a^3 + 10*a^2 + 7*a + 3,
 3*a^3 + 10*a^2 + 7*a + 1,
 2*a^3 + 3*a^2 + a + 7,
 a^3 + 7*a^2 + 6*a]
```

Now we check that these are exactly the x-coordinates of the 5-torsion points of E:

```
sage: for x in x_coords:                                                    #␣
↪needs sage.rings.finite_rings
....:     assert X.lift_x(x).order() == 5
```

The roots of the polynomial are the x-coordinates of the points P such that $mP = 0$ but $2P \neq 0$:

```
sage: E = EllipticCurve('14a1')
sage: T = E.torsion_subgroup()
sage: [n*T.0 for n in range(6)]
[(0 : 1 : 0),
(9 : 23 : 1),
(2 : 2 : 1),
(1 : -1 : 1),
(2 : -5 : 1),
(9 : -33 : 1)]
sage: pol = E.division_polynomial_0(6)
sage: xlist = pol.roots(multiplicities=False); xlist
[9, 2, -1/3, -5]
sage: [E.lift_x(x, all=True) for x in xlist]
[[(9 : -33 : 1), (9 : 23 : 1)], [(2 : -5 : 1), (2 : 2 : 1)], [], []]
```

Note: The point of order 2 and the identity do not appear. The points with $x = -1/3$ and $x = -5$ are not rational.

formal()

Return the formal group associated to this elliptic curve.

This method is cached.

EXAMPLES:

```
sage: E = EllipticCurve("37a")
sage: E.formal_group()
Formal Group associated to the Elliptic Curve
defined by y^2 + y = x^3 - x over Rational Field
```

formal_group()

Return the formal group associated to this elliptic curve.

This method is cached.

EXAMPLES:

```
sage: E = EllipticCurve("37a")
sage: E.formal_group()
Formal Group associated to the Elliptic Curve
defined by y^2 + y = x^3 - x over Rational Field
```

frobenius_isogeny(*n=1*)

Return the n-power Frobenius isogeny from this curve to its Galois conjugate.

The Frobenius *endo*morphism is the special case where n is divisible by the degree of the base ring of the curve.

See also:

frobenius_endomorphism()

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: z3, = GF(13^3).gens()
sage: E = EllipticCurve([z3, z3^2])
sage: E.frobenius_isogeny()
Frobenius isogeny of degree 13:
  From: Elliptic Curve defined by y^2 = x^3 + z3*x + z3^2
        over Finite Field in z3 of size 13^3
  To:   Elliptic Curve defined by y^2 = x^3 + (5*z3^2+7*z3+11)*x + (5*z3^
↪2+12*z3+1)
        over Finite Field in z3 of size 13^3
sage: E.frobenius_isogeny(3)
Frobenius endomorphism of degree 2197 = 13^3:
  From: Elliptic Curve defined by y^2 = x^3 + z3*x + z3^2
        over Finite Field in z3 of size 13^3
  To:   Elliptic Curve defined by y^2 = x^3 + z3*x + z3^2
        over Finite Field in z3 of size 13^3
```

gen(*i*)

Function returning the i'th generator of this elliptic curve.

Note: Relies on gens() being implemented.

EXAMPLES:

```
sage: R.<a1,a2,a3,a4,a6> = QQ[]
sage: E = EllipticCurve([a1,a2,a3,a4,a6])
sage: E.gen(0)
Traceback (most recent call last):
...
NotImplementedError: not implemented.
```

gens()

Placeholder function to return generators of an elliptic curve.

Note: This functionality is implemented in certain derived classes, such as EllipticCurve_rational_field.

EXAMPLES:

```
sage: R.<a1,a2,a3,a4,a6> = QQ[]
sage: E = EllipticCurve([a1,a2,a3,a4,a6])
sage: E.gens()
Traceback (most recent call last):
...
NotImplementedError: not implemented.
sage: E = EllipticCurve(QQ, [1,1])
sage: E.gens()
[(0 : 1 : 1)]
```

hyperelliptic_polynomials()

Return a pair of polynomials $g(x)$, $h(x)$ such that this elliptic curve can be defined by the standard hyperelliptic equation

$$y^2 + h(x)y = g(x).$$

EXAMPLES:

```
sage: R.<a1,a2,a3,a4,a6>=QQ[]
sage: E = EllipticCurve([a1,a2,a3,a4,a6])
sage: E.hyperelliptic_polynomials()
(x^3 + a2*x^2 + a4*x + a6, a1*x + a3)
```

identity_morphism()

Return the identity endomorphism of this elliptic curve as an `EllipticCurveHom` object.

EXAMPLES:

```
sage: E = EllipticCurve(j=42)
sage: E.identity_morphism()
Elliptic-curve endomorphism of Elliptic Curve defined by y^2 = x^3 + 5901*x +␣
↪1105454 over Rational Field
  Via:  (u,r,s,t) = (1, 0, 0, 0)
sage: E.identity_morphism() == E.scalar_multiplication(1)
True
```

is_isomorphic(*other*, *field=None*)

Return whether or not `self` is isomorphic to `other`.

INPUT:

- `other` – another elliptic curve.

- `field` (default None) – a field into which the coefficients of the curves may be coerced (by default, uses the base field of the curves).

OUTPUT:

(bool) True if there is an isomorphism from curve `self` to curve `other` defined over `field`.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: F = E.change_weierstrass_model([2,3,4,5]); F
Elliptic Curve defined by y^2 + 4*x*y + 11/8*y = x^3 - 3/2*x^2 - 13/16*x
over Rational Field
sage: E.is_isomorphic(F)
True
sage: E.is_isomorphic(F.change_ring(CC))
False
```

is_on_curve(x, y)

Return True if (x, y) is an affine point on this curve.

INPUT:

- `x`, `y` – elements of the base ring of the curve.

EXAMPLES:

```
sage: E = EllipticCurve(QQ,[1,1])
sage: E.is_on_curve(0,1)
True
sage: E.is_on_curve(1,1)
False
```

is_x_coord(x)

Return True if `x` is the x-coordinate of a point on this curve.

Note: See also *lift_x()* to find the point(s) with a given x-coordinate. This function may be useful in cases where testing an element of the base field for being a square is faster than finding its square root.

EXAMPLES:

```
sage: E = EllipticCurve('37a'); E
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
sage: E.is_x_coord(1)
True
sage: E.is_x_coord(2)
True
```

There are no rational points with x-coordinate 3:

```
sage: E.is_x_coord(3)
False
```

However, there are such points in $E(\mathbf{R})$:

```
sage: E.change_ring(RR).is_x_coord(3)
True
```

And of course it always works in $E(\mathbf{C})$:

```
sage: E.change_ring(RR).is_x_coord(-3)
False
sage: E.change_ring(CC).is_x_coord(-3)
True
```

AUTHORS:

- John Cremona (2008-08-07): adapted from *lift_x()*

isomorphism(*u*, *r*, *s=0*, *t=0*, *is_codomain=0*)

Given four values u, r, s, t in the base ring of this curve, return the `WeierstrassIsomorphism` defined by u, r, s, t with this curve as its codomain. (The value u must be a unit; the values r, s, t default to zero.)

Optionally, if the keyword argument `is_codomain` is set to `True`, return the isomorphism defined by u, r, s, t with this curve as its **co**domain.

EXAMPLES:

```
sage: E = EllipticCurve([1, 2, 3, 4, 5])
sage: iso = E.isomorphism(6); iso
Elliptic-curve morphism:
  From: Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5␣
↪over Rational Field
  To:   Elliptic Curve defined by y^2 + 1/6*x*y + 1/72*y = x^3 + 1/18*x^2 + 1/
↪324*x + 5/46656 over Rational Field
  Via:  (u,r,s,t) = (6, 0, 0, 0)
sage: iso.domain() == E
True
sage: iso.codomain() == E.scale_curve(1 / 6)
True

sage: iso = E.isomorphism(1, 7, 8, 9); iso
Elliptic-curve morphism:
  From: Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5␣
↪over Rational Field
  To:   Elliptic Curve defined by y^2 + 17*x*y + 28*y = x^3 - 49*x^2 - 54*x +␣
↪303 over Rational Field
  Via:  (u,r,s,t) = (1, 7, 8, 9)
sage: iso.domain() == E
True
sage: iso.codomain() == E.rst_transform(7, 8, 9)
True

sage: iso = E.isomorphism(6, 7, 8, 9); iso
Elliptic-curve morphism:
  From: Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5␣
↪over Rational Field
  To:   Elliptic Curve defined by y^2 + 17/6*x*y + 7/54*y = x^3 - 49/36*x^2 -␣
↪1/24*x + 101/15552 over Rational Field
  Via:  (u,r,s,t) = (6, 7, 8, 9)
sage: iso.domain() == E
True
sage: iso.codomain() == E.rst_transform(7, 8, 9).scale_curve(1 / 6)
True
```

The `is_codomain` argument reverses the role of domain and codomain:

```
sage: E = EllipticCurve([1, 2, 3, 4, 5])
sage: iso = E.isomorphism(6, is_codomain=True); iso
Elliptic-curve morphism:
  From: Elliptic Curve defined by y^2 + 6*x*y + 648*y = x^3 + 72*x^2 + 5184*x␣
↪+ 233280 over Rational Field
  To:   Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5␣
↪over Rational Field
  Via:  (u,r,s,t) = (6, 0, 0, 0)
sage: iso.domain() == E.scale_curve(6)
True
sage: iso.codomain() == E
True

sage: iso = E.isomorphism(1, 7, 8, 9, is_codomain=True); iso
Elliptic-curve morphism:
  From: Elliptic Curve defined by y^2 - 15*x*y + 90*y = x^3 - 75*x^2 + 796*x -
↪ 2289 over Rational Field
  To:   Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5␣
↪over Rational Field
  Via:  (u,r,s,t) = (1, 7, 8, 9)
sage: iso.domain().rst_transform(7, 8, 9) == E
True
sage: iso.codomain() == E
True

sage: iso = E.isomorphism(6, 7, 8, 9, is_codomain=True); iso
Elliptic-curve morphism:
  From: Elliptic Curve defined by y^2 - 10*x*y + 700*y = x^3 + 35*x^2 +␣
↪9641*x + 169486 over Rational Field
  To:   Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5␣
↪over Rational Field
  Via:  (u,r,s,t) = (6, 7, 8, 9)
sage: iso.domain().rst_transform(7, 8, 9) == E.scale_curve(6)
True
sage: iso.codomain() == E
True
```

See also:

- *WeierstrassIsomorphism*
- *rst_transform()*
- *scale_curve()*

isomorphism_to(*other*)

Given another weierstrass model `other` of `self`, return an isomorphism from `self` to `other`.

INPUT:

- `other` – an elliptic curve isomorphic to `self`.

OUTPUT:

(Weierstrassmorphism) An isomorphism from `self` to `other`.

Note: If the curves in question are not isomorphic, a `ValueError` is raised.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: F = E.short_weierstrass_model()
sage: w = E.isomorphism_to(F); w
Elliptic-curve morphism:
  From: Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
  To:   Elliptic Curve defined by y^2  = x^3 - 16*x + 16 over Rational Field
  Via:  (u,r,s,t) = (1/2, 0, 0, -1/2)
sage: P = E(0,-1,1)
sage: w(P)
(0 : -4 : 1)
sage: w(5*P)
(1 : 1 : 1)
sage: 5*w(P)
(1 : 1 : 1)
sage: 120*w(P) == w(120*P)
True
```

We can also handle injections to different base rings:

```
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 7)                                              #␣
↪needs sage.rings.number_field
sage: E.isomorphism_to(E.change_ring(K))                                        #␣
↪needs sage.rings.number_field
Elliptic-curve morphism:
  From: Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
  To:   Elliptic Curve defined by y^2 + y = x^3 + (-1)*x
        over Number Field in a with defining polynomial x^3 - 7
  Via:  (u,r,s,t) = (1, 0, 0, 0)
```

isomorphisms(*other*, *field=None*)

Return the set of isomorphisms from `self` to `other` (as a list).

INPUT:

- `other` – another elliptic curve.
- `field` (default `None`) – a field into which the coefficients of the curves may be coerced (by default, uses the base field of the curves).

OUTPUT:

(list) A list of `WeierstrassIsomorphism` objects consisting of all the isomorphisms from the curve `self` to the curve `other` defined over `field`.

EXAMPLES:

```
sage: E = EllipticCurve_from_j(QQ(0)) # a curve with j=0 over QQ
sage: F = EllipticCurve('27a3') # should be the same one
sage: E.isomorphisms(F)
[Elliptic-curve endomorphism of Elliptic Curve defined by y^2 + y = x^3
  over Rational Field
   Via:  (u,r,s,t) = (1, 0, 0, 0),
 Elliptic-curve endomorphism of Elliptic Curve defined by y^2 + y = x^3
  over Rational Field
   Via:  (u,r,s,t) = (-1, 0, 0, -1)]
```

We can also find isomorphisms defined over extension fields:

```
sage: # needs sage.rings.finite_rings
sage: E = EllipticCurve(GF(7), [0,0,0,1,1])
sage: F = EllipticCurve(GF(7), [0,0,0,1,-1])
sage: E.isomorphisms(F)
[]
sage: E.isomorphisms(F, GF(49,'a'))
[Elliptic-curve morphism:
   From: Elliptic Curve defined by y^2 = x^3 + x + 1
         over Finite Field in a of size 7^2
   To:   Elliptic Curve defined by y^2 = x^3 + x + 6
         over Finite Field in a of size 7^2
   Via:  (u,r,s,t) = (a + 3, 0, 0, 0),
 Elliptic-curve morphism:
   From: Elliptic Curve defined by y^2 = x^3 + x + 1
         over Finite Field in a of size 7^2
   To:   Elliptic Curve defined by y^2 = x^3 + x + 6
         over Finite Field in a of size 7^2
   Via:  (u,r,s,t) = (6*a + 4, 0, 0, 0)]
```

j_invariant()

Return the j-invariant of this elliptic curve.

This method is cached.

EXAMPLES:

```
sage: E = EllipticCurve([0,0,1,-1,0])
sage: E.j_invariant()
110592/37

sage: E = EllipticCurve([0, -1, 1, -10, -20])
sage: E.j_invariant()
-122023936/161051

sage: E = EllipticCurve([-4,0])
sage: E.j_invariant()
1728

sage: E = EllipticCurve([GF(7)(2),1])
sage: E.j_invariant()
1
```

lift_x(*x*, *all=False*, *extend=False*)

Return one or all points with given x-coordinate.

This method is deterministic: It returns the same data each time when called again with the same x.

INPUT:

- `x` – an element of the base ring of the curve, or of an extension.
- `all` (bool, default: `False`) – if True, return a (possibly empty) list of all points; if False, return just one point, or raise a `ValueError` if there are none.
- `extend` (bool, default: `False`) –
  - if `False`, extend the base if necessary and possible to include x, and only return point(s) defined over this ring, or raise an error when there are none with this x-coordinate;

– If `True`, the base ring will be extended if necessary to contain the y-coordinates of the point(s) with this x-coordinate, in addition to a possible base change to include x.

OUTPUT:

A point or list of up to 2 points on this curve, or a base-change of this curve to a larger ring.

See also:

is_x_coord()

EXAMPLES:

```
sage: E = EllipticCurve('37a'); E
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
sage: E.lift_x(1)
(1 : -1 : 1)
sage: E.lift_x(2)
(2 : -3 : 1)
sage: E.lift_x(1/4, all=True)
[(1/4 : -5/8 : 1), (1/4 : -3/8 : 1)]
```

There are no rational points with x-coordinate 3:

```
sage: E.lift_x(3)
Traceback (most recent call last):
...
ValueError: No point with x-coordinate 3
on Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
```

We can use the `extend` parameter to make the necessary quadratic extension. Note that in such cases the returned point is a point on a new curve object, the result of changing the base ring to the parent of x:

```
sage: P = E.lift_x(3, extend=True); P                                       #␣
↪needs sage.rings.number_field
(3 : -y - 1 : 1)
sage: P.curve()                                                             #␣
↪needs sage.rings.number_field
Elliptic Curve defined by y^2 + y = x^3 + (-1)*x
over Number Field in y with defining polynomial y^2 + y - 24
```

Or we can extend scalars. There are two such points in $E(\mathbf{R})$:

```
sage: E.change_ring(RR).lift_x(3, all=True)
[(3.00000000000000 : -5.42442890089805 : 1.00000000000000),
 (3.00000000000000 : 4.42442890089805 : 1.00000000000000)]
```

And of course it always works in $E(\mathbf{C})$:

```
sage: E.change_ring(RR).lift_x(.5, all=True)
[]
sage: E.change_ring(CC).lift_x(.5)
(0.500000000000000 : -0.500000000000000 - 0.353553390593274*I : 1.
↪00000000000000)
```

In this example we start with a curve defined over $\mathbf{Q}$ which has no rational points with $x = 0$, but using `extend = True` we can construct such a point over a quadratic field:

```
sage: E = EllipticCurve([0,0,0,0,2]); E
Elliptic Curve defined by y^2 = x^3 + 2 over Rational Field
sage: P = E.lift_x(0, extend=True); P                                          #␣
↪needs sage.rings.number_field
(0 : -y : 1)
sage: P.curve()                                                                #␣
↪needs sage.rings.number_field
Elliptic Curve defined by y^2 = x^3 + 2
over Number Field in y with defining polynomial y^2 - 2
```

We can perform these operations over finite fields too:

```
sage: E = EllipticCurve('37a').change_ring(GF(17)); E
Elliptic Curve defined by y^2 + y = x^3 + 16*x over Finite Field of size 17
sage: E.lift_x(7)
(7 : 5 : 1)
sage: E.lift_x(3)
Traceback (most recent call last):
...
ValueError: No point with x-coordinate 3 on
Elliptic Curve defined by y^2 + y = x^3 + 16*x over Finite Field of size 17
```

Note that there is only one lift with x-coordinate 10 in $E(\mathbf{F}_{17})$:

```
sage: E.lift_x(10, all=True)
[(10 : 8 : 1)]
```

We can lift over more exotic rings too. If the supplied x value is in an extension of the base, note that the point returned is on the base-extended curve:

```
sage: E = EllipticCurve('37a')
sage: P = E.lift_x(pAdicField(17, 5)(6)); P                                    #␣
↪needs sage.rings.padics
(6 + O(17^5) : 14 + O(17^5) : 1 + O(17^5))
sage: P.curve()                                                                #␣
↪needs sage.rings.padics
Elliptic Curve defined by
y^2 + (1+O(17^5))*y = x^3 + (16+16*17+16*17^2+16*17^3+16*17^4+O(17^5))*x
over 17-adic Field with capped relative precision 5
sage: K.<t> = PowerSeriesRing(QQ, 't', 5)
sage: P = E.lift_x(1 + t); P
(1 + t : -1 - 2*t + t^2 - 5*t^3 + 21*t^4 + O(t^5) : 1)
sage: K.<a> = GF(16)                                                           #␣
↪needs sage.rings.finite_rings
sage: P = E.change_ring(K).lift_x(a^3); P                                      #␣
↪needs sage.rings.finite_rings
(a^3 : a^3 + a : 1)
sage: P.curve()                                                                #␣
↪needs sage.rings.finite_rings
Elliptic Curve defined by y^2 + y = x^3 + x over Finite Field in a of size 2^4
```

We can extend the base field to include the associated y value(s):

```
sage: E = EllipticCurve([0,0,0,0,2]); E
Elliptic Curve defined by y^2 = x^3 + 2 over Rational Field
sage: x = polygen(QQ)
sage: P = E.lift_x(x, extend=True); P
(x : -y : 1)
```

This point is a generic point on E:

```
sage: P.curve()
Elliptic Curve defined by y^2 = x^3 + 2
over Univariate Quotient Polynomial Ring in y
over Fraction Field of Univariate Polynomial Ring in x over Rational Field
with modulus y^2 - x^3 - 2
sage: -P
(x : y : 1)
sage: 2*P
((1/4*x^4 - 4*x)/(x^3 + 2) : ((-1/8*x^6 - 5*x^3 + 4)/(x^6 + 4*x^3 + 4))*y : 1)
```

Check that Issue #30297 is fixed:

```
sage: K = Qp(5)                                                             #␣
↪needs sage.rings.padics
sage: E = EllipticCurve([K(0), K(1)])                                       #␣
↪needs sage.rings.padics
sage: E.lift_x(1, extend=True)                                              #␣
↪needs sage.rings.padics
(1 + O(5^20) : y + O(5^20) : 1 + O(5^20))
```

AUTHORS:

- Robert Bradshaw (2007-04-24)
- John Cremona (2017-11-10)

montgomery_model (*twisted=False*, *morphism=False*)

Return a (twisted or untwisted) Montgomery model for this elliptic curve, if possible.

A Montgomery curve is a smooth projective curve of the form

$$BY^2 = X^3 + AX^2 + X.$$

The Montgomery curve is called *untwisted* if $B = 1$.

INPUT:

- `twisted` – boolean (default: `False`); allow $B \neq 1$
- `morphism` – boolean (default: `False`); also return an isomorphism from this curve to the computed Montgomery model

OUTPUT:

If `twisted` is `False` (the default), an *EllipticCurve_generic* object encapsulating an untwisted Montgomery curve. Otherwise, a `ProjectivePlaneCurve` object encapsulating a (potentially twisted) Montgomery curve.

If `morphism` is `True`, this method returns a tuple consisting of such a curve together with an isomorphism of suitable type (either *WeierstrassIsomorphism* or *WeierstrassTransformationWithInverse*) from this curve to the Montgomery model.

EXAMPLES:

```
sage: E = EllipticCurve(QQbar, '11a1')                                      #␣
↪needs sage.rings.number_field
sage: E.montgomery_model()                                                  #␣
↪needs sage.rings.number_field
Elliptic Curve defined by y^2 = x^3 + (-1.953522420987248?)*x^2 + x
over Algebraic Field
```

```
sage: E = EllipticCurve(GF(431^2), [7,7])                                    #␣
↪needs sage.rings.finite_rings
sage: E.montgomery_model()                                                   #␣
↪needs sage.rings.finite_rings
Elliptic Curve defined by y^2 = x^3 + (51*z2+190)*x^2 + x
over Finite Field in z2 of size 431^2
```

An isomorphism between the Montgomery and Weierstrass form can be obtained using the `morphism` parameter:

```
sage: E.montgomery_model(morphism=True)                                      #␣
↪needs sage.rings.finite_rings
(Elliptic Curve defined by y^2 = x^3 + (51*z2+190)*x^2 + x
  over Finite Field in z2 of size 431^2,
 Elliptic-curve morphism:
   From: Elliptic Curve defined by y^2 = x^3 + 7*x + 7
         over Finite Field in z2 of size 431^2
   To:   Elliptic Curve defined by y^2 = x^3 + (51*z2+190)*x^2 + x
         over Finite Field in z2 of size 431^2
   Via:  (u,r,s,t) = (64*z2 + 407, 159, 0, 0))
```

Not all elliptic curves have a Montgomery model over their field of definition:

```
sage: E = EllipticCurve(GF(257), [1,1])
sage: E.montgomery_model()
Traceback (most recent call last):
...
ValueError: Elliptic Curve defined by y^2 = x^3 + x + 1
over Finite Field of size 257 has no Montgomery model
```

```
sage: E = EllipticCurve(GF(257), [10,10])
sage: E.montgomery_model()
Traceback (most recent call last):
...
ValueError: Elliptic Curve defined by y^2 = x^3 + 10*x + 10
over Finite Field of size 257 has no untwisted Montgomery model
```

However, as hinted by the error message, the latter curve does admit a *twisted* Montgomery model, which can be computed by passing `twisted=True`:

```
sage: E.montgomery_model(twisted=True)
Projective Plane Curve over Finite Field of size 257
defined by -x^3 + 8*x^2*z - 127*y^2*z - x*z^2
```

Since Sage internally represents elliptic curves as (long) Weierstrass curves, which do not feature the Montgomery B coefficient, the returned curve in this case is merely a `ProjectivePlaneCurve` rather than the usual *`EllipticCurve_generic`*.

Arithmetic on curves of this type is not implemented natively, but can easily be emulated by mapping back and forth to the corresponding Weierstrass curve:

```
sage: C, f = E.montgomery_model(twisted=True, morphism=True)
sage: f
Scheme morphism:
  From: Elliptic Curve defined by y^2 = x^3 + 10*x + 10
        over Finite Field of size 257
  To:   Projective Plane Curve over Finite Field of size 257
```

(continues on next page)

(continued from previous page)

```
        defined by -x^3 + 8*x^2*z - 127*y^2*z - x*z^2
  Defn: Defined on coordinates by sending (x : y : z) to
        (x + 116*z : -y : -85*z)
sage: g = f.inverse(); g
Scheme morphism:
  From: Projective Plane Curve over Finite Field of size 257
        defined by -x^3 + 8*x^2*z - 127*y^2*z - x*z^2
  To:   Elliptic Curve defined by y^2 = x^3 + 10*x + 10
        over Finite Field of size 257
  Defn: Defined on coordinates by sending (x : y : z) to
        (-85*x - 116*z : 85*y : z)
sage: P = C(70, 8)
sage: Q = C(17, 17)
sage: P + Q              # this doesn't work...
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +: ...
sage: f(g(P) + g(Q))     # ...but this does
(107 : 168 : 1)
```

Using the fact that the Weil pairing satisfies $e(\psi(P), \psi(Q)) = e(P, Q)^{\deg \psi}$, even pairings can be emulated in this way (note that isomorphisms have degree 1):

```
sage: # needs sage.rings.finite_rings
sage: F.<z2> = GF(257^2)
sage: C_ = C.change_ring(F)
sage: g_ = g.change_ring(F)
sage: g_(P).order()
12
sage: T = C_(-7 * z2 - 57, 31 * z2 - 52, 1)
sage: g_(T).order()
12
sage: g_(P).weil_pairing(g_(T), 12)
15*z2 + 204
```

Another alternative is to simply extend the base field enough for the curve to have an untwisted Montgomery model:

```
sage: C_ = E.change_ring(F).montgomery_model(); C_                              #␣
↪needs sage.rings.finite_rings
Elliptic Curve defined by y^2 = x^3 + 249*x^2 + x
over Finite Field in z2 of size 257^2
sage: h = C.defining_polynomial().change_ring(F); h                             #␣
↪needs sage.rings.finite_rings
-x^3 + 8*x^2*z - 127*y^2*z - x*z^2
sage: C_.is_isomorphic(EllipticCurve_from_cubic(h).codomain())                  #␣
↪needs sage.rings.finite_rings
True
```

See also:

The inverse conversion — computing a Weierstrass model for a given Montgomery curve — can be performed using *EllipticCurve_from_cubic()*.

ALGORITHM: [CS2018], §2.4

REFERENCES:

- Original publication: [Mont1987], §10.3.1

- More recent survey article: [CS2018]

multiplication_by_m(*m*, *x_only=False*)

Return the multiplication-by-m map from `self` to `self`

The result is a pair of rational functions in two variables x, y (or a rational function in one variable x if `x_only` is `True`).

INPUT:

- `m` – a nonzero integer
- `x_only` – boolean (default: `False`) if `True`, return only the x-coordinate of the map (as a rational function in one variable).

OUTPUT:

- a pair $(f(x), g(x,y))$, where f and g are rational functions with the degree of y in $g(x,y)$ exactly 1,
- or just $f(x)$ if `x_only` is `True`

Note:

- The result is not cached.
- `m` is allowed to be negative (but not 0).

EXAMPLES:

```
sage: E = EllipticCurve([-1,3])
```

We verify that multiplication by 1 is just the identity:

```
sage: E.multiplication_by_m(1)
(x, y)
```

Multiplication by 2 is more complicated:

```
sage: f = E.multiplication_by_m(2)
sage: f
((x^4 + 2*x^2 - 24*x + 1)/(4*x^3 - 4*x + 12),
 (8*x^6*y - 40*x^4*y + 480*x^3*y - 40*x^2*y + 96*x*y - 568*y)/(64*x^6 - 128*x^
↪4 + 384*x^3 + 64*x^2 - 384*x + 576))
```

Grab only the x-coordinate (less work):

```
sage: mx = E.multiplication_by_m(2, x_only=True); mx
(1/4*x^4 + 1/2*x^2 - 6*x + 1/4)/(x^3 - x + 3)
sage: mx.parent()
Fraction Field of Univariate Polynomial Ring in x over Rational Field
```

We check that it works on a point:

```
sage: P = E([2,3])
sage: eval = lambda f,P: [fi(P[0],P[1]) for fi in f]
sage: assert E(eval(f,P)) == 2*P
```

We do the same but with multiplication by 3:

```
sage: f = E.multiplication_by_m(3)
sage: assert E(eval(f,P)) == 3*P
```

And the same with multiplication by 4:

```
sage: f = E.multiplication_by_m(4)
sage: assert E(eval(f,P)) == 4*P
```

And the same with multiplication by -1,-2,-3,-4:

```
sage: for m in [-1,-2,-3,-4]:
....:     f = E.multiplication_by_m(m)
....:     assert E(eval(f,P)) == m*P
```

multiplication_by_m_isogeny(*m*)

Return the `EllipticCurveIsogeny` object associated to the multiplication-by-m map on this elliptic curve.

The resulting isogeny will have the associated rational maps (i.e., those returned by *multiplication_by_m()*) already computed.

NOTE: This function is currently *much* slower than the result of `self.multiplication_by_m()`, because constructing an isogeny precomputes a significant amount of information. See Issue #7368 and Issue #8014 for the status of improving this situation.

INPUT:

- `m` – a nonzero integer

OUTPUT:

- An `EllipticCurveIsogeny` object associated to the multiplication-by-m map on this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: E.multiplication_by_m_isogeny(7)
doctest:warning ... DeprecationWarning: ...
Isogeny of degree 49
 from Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20
       over Rational Field
 to   Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20
       over Rational Field
```

pari_curve()

Return the PARI curve corresponding to this elliptic curve.

The result is cached.

EXAMPLES:

```
sage: # needs sage.libs.pari
sage: E = EllipticCurve([RR(0), RR(0), RR(1), RR(-1), RR(0)])
sage: e = E.pari_curve()
sage: type(e)
<... 'cypari2.gen.Gen'>
sage: e.type()
't_VEC'
```

(continues on next page)

(continued from previous page)

```
sage: e.disc()
37.0000000000000
```

Over a finite field:

```
sage: EllipticCurve(GF(41), [2,5]).pari_curve()                                #␣
↪needs sage.libs.pari
[Mod(0, 41), Mod(0, 41), Mod(0, 41), Mod(2, 41), Mod(5, 41),
 Mod(0, 41), Mod(4, 41), Mod(20, 41), Mod(37, 41), Mod(27, 41),
 Mod(26, 41), Mod(4, 41), Mod(11, 41),
 Vecsmall([3]),
 [41, [9, 31, [6, 0, 0, 0]]], [0, 0, 0, 0]]
```

Over a p-adic field:

```
sage: # needs sage.libs.pari sage.rings.padics
sage: Qp = pAdicField(5, prec=3)
sage: E = EllipticCurve(Qp, [3, 4])
sage: E.pari_curve()
[0, 0, 0, 3, 4, 0, 6, 16, -9, -144, -3456, -8640, 1728/5,
 Vecsmall([2]), [O(5^3)], [0, 0]]
sage: E.j_invariant()
3*5^-1 + O(5)
```

Over a number field:

```
sage: K.<a> = QuadraticField(2)                                                #␣
↪needs sage.libs.pari sage.rings.number_field
sage: E = EllipticCurve([1,a])                                                 #␣
↪needs sage.libs.pari sage.rings.number_field
sage: E.pari_curve()                                                           #␣
↪needs sage.libs.pari sage.rings.number_field
[0, 0, 0, Mod(1, y^2 - 2),
 Mod(y, y^2 - 2), 0, Mod(2, y^2 - 2), Mod(4*y, y^2 - 2),
 Mod(-1, y^2 - 2), Mod(-48, y^2 - 2), Mod(-864*y, y^2 - 2),
 Mod(-928, y^2 - 2), Mod(3456/29, y^2 - 2),
 Vecsmall([5]),
 [[y^2 - 2, [2, 0], 8, 1, [[1, -1.41421356237310; 1, 1.41421356237310],
 [1, -1.41421356237310; 1, 1.41421356237310],
 [16, -23; 16, 23], [2, 0; 0, 4], [4, 0; 0, 2], [2, 0; 0, 1],
 [2, [0, 2; 1, 0]], [2]], [-1.41421356237310, 1.41421356237310],
 [1, y], [1, 0; 0, 1], [1, 0, 0, 2; 0, 1, 1, 0]]], [0, 0, 0, 0, 0]]
```

PARI no longer requires that the j-invariant has negative p-adic valuation:

```
sage: E = EllipticCurve(Qp,[1, 1])                                             #␣
↪needs sage.libs.pari sage.rings.padics
sage: E.j_invariant()  # the j-invariant is a p-adic integer                   #␣
↪needs sage.libs.pari sage.rings.padics
2 + 4*5^2 + O(5^3)
sage: E.pari_curve()                                                           #␣
↪needs sage.libs.pari sage.rings.padics
[0, 0, 0, 1, 1, 0, 2, 4, -1, -48, -864, -496, 6912/31,
 Vecsmall([2]), [O(5^3)], [0, 0]]
```

plot(*xmin=None*, *xmax=None*, *components='both'*, ***args*)

Draw a graph of this elliptic curve.

The plot method is only implemented when there is a natural coercion from the base ring of `self` to `RR`. In this case, `self` is plotted as if it was defined over `RR`.

INPUT:

- `xmin`, `xmax` – (optional) points will be computed at least within this range, but possibly farther.
- `components` – a string, one of the following:
  - `both` – (default), scale so that both bounded and unbounded components appear
  - `bounded` – scale the plot to show the bounded component. Raises an error if there is only one real component.
  - `unbounded` – scale the plot to show the unbounded component, including the two flex points.
- `plot_points` – passed to `sage.plot.generate_plot_points()`
- `adaptive_tolerance` – passed to `sage.plot.generate_plot_points()`
- `adaptive_recursion` – passed to `sage.plot.generate_plot_points()`
- `randomize` – passed to `sage.plot.generate_plot_points()`
- `**args` – all other options are passed to `sage.plot.line.Line`

EXAMPLES:

```
sage: E = EllipticCurve([0, -1])
sage: plot(E, rgbcolor=hue(0.7))                                                #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
sage: E = EllipticCurve('37a')
sage: plot(E)                                                                   #␣
↪needs sage.plot
Graphics object consisting of 2 graphics primitives
sage: plot(E, xmin=25, xmax=26)                                                 #␣
↪needs sage.plot
Graphics object consisting of 2 graphics primitives
```

With Issue #12766 we added the components keyword:

```
sage: E.real_components()
2
sage: E.plot(components='bounded')                                              #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
sage: E.plot(components='unbounded')                                            #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

If there is only one component then specifying components='bounded' raises a ValueError:

```
sage: E = EllipticCurve('9990be2')
sage: E.plot(components='bounded')                                              #␣
↪needs sage.plot
Traceback (most recent call last):
...
ValueError: no bounded component for this curve
```

An elliptic curve defined over the Complex Field can not be plotted:

```
sage: E = EllipticCurve(CC, [0,0,1,-1,0])
sage: E.plot()                                                              #␣
↪needs sage.plot
Traceback (most recent call last):
...
NotImplementedError: plotting of curves over Complex Field
with 53 bits of precision is not implemented yet
```

rst_transform(r, s, t)

Return the transform of the curve by (r, s, t) (with $u = 1$).

INPUT:

- r, s, t – three elements of the base ring.

OUTPUT:

The elliptic curve obtained from self by the standard Weierstrass transformation (u, r, s, t) with $u = 1$.

Note: This is just a special case of *change_weierstrass_model()*, with $u = 1$.

EXAMPLES:

```
sage: R.<r,s,t> = QQ[]
sage: E = EllipticCurve([1,2,3,4,5])
sage: E.rst_transform(r, s, t)
Elliptic Curve defined by y^2 + (2*s+1)*x*y + (r+2*t+3)*y
= x^3 + (-s^2+3*r-s+2)*x^2 + (3*r^2-r*s-2*s*t+4*r-3*s-t+4)*x
  + (r^3+2*r^2-r*t-t^2+4*r-3*t+5)
over Multivariate Polynomial Ring in r, s, t over Rational Field
```

scalar_multiplication(m)

Return the scalar-multiplication map $[m]$ on this elliptic curve as a *sage.schemes.elliptic_curves.hom_scalar.EllipticCurveHom_scalar* object.

EXAMPLES:

```
sage: E = EllipticCurve('77a1')
sage: m = E.scalar_multiplication(-7); m
Scalar-multiplication endomorphism [-7]
of Elliptic Curve defined by y^2 + y = x^3 + 2*x over Rational Field
sage: m.degree()
49
sage: P = E(2,3)
sage: m(P)
(-26/225 : -2132/3375 : 1)
sage: m.rational_maps() == E.multiplication_by_m(-7)
True
```

scale_curve(u)

Return the transform of the curve by scale factor u.

INPUT:

- u – an invertible element of the base ring.

OUTPUT:

The elliptic curve obtained from `self` by the standard Weierstrass transformation (u, r, s, t) with $r = s = t = 0$.

Note: This is just a special case of *change_weierstrass_model()*, with $r = s = t = 0$.

EXAMPLES:

```
sage: K = Frac(PolynomialRing(QQ, 'u'))
sage: u = K.gen()
sage: E = EllipticCurve([1,2,3,4,5])
sage: E.scale_curve(u)
Elliptic Curve defined by
y^2 + u*x*y + 3*u^3*y = x^3 + 2*u^2*x^2 + 4*u^4*x + 5*u^6
over Fraction Field of Univariate Polynomial Ring in u over Rational Field
```

short_weierstrass_model (*complete_cube=True*)

Return a short Weierstrass model for self.

INPUT:

- `complete_cube` – boolean (default: `True`); for meaning, see below.

OUTPUT:

An elliptic curve.

If `complete_cube=True`: Return a model of the form $y^2 = x^3 + a * x + b$ for this curve. The characteristic must not be 2; in characteristic 3, it is only possible if $b_2 = 0$.

If `complete_cube=False`: Return a model of the form $y^2 = x^3 + ax^2 + bx + c$ for this curve. The characteristic must not be 2.

EXAMPLES:

```
sage: E = EllipticCurve([1,2,3,4,5])
sage: E
Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5 over␣
↪Rational Field
sage: F = E.short_weierstrass_model()
sage: F
Elliptic Curve defined by y^2  = x^3 + 4941*x + 185166 over Rational Field
sage: E.is_isomorphic(F)
True
sage: F = E.short_weierstrass_model(complete_cube=False)
sage: F
Elliptic Curve defined by y^2  = x^3 + 9*x^2 + 88*x + 464 over Rational Field
sage: E.is_isomorphic(F)
True
```

```
sage: E = EllipticCurve(GF(3), [1,2,3,4,5])
sage: E.short_weierstrass_model(complete_cube=False)
Elliptic Curve defined by y^2 = x^3 + x + 2 over Finite Field of size 3
```

This used to be different see Issue #3973:

```
sage: E.short_weierstrass_model()
Elliptic Curve defined by y^2 = x^3 + x + 2 over Finite Field of size 3
```

More tests in characteristic 3:

```
sage: E = EllipticCurve(GF(3), [0,2,1,2,1])
sage: E.short_weierstrass_model()
Traceback (most recent call last):
...
ValueError: short_weierstrass_model(): no short model for Elliptic Curve
defined by y^2 + y = x^3 + 2*x^2 + 2*x + 1 over Finite Field of size 3
(characteristic is 3)
sage: E.short_weierstrass_model(complete_cube=False)
Elliptic Curve defined by y^2 = x^3 + 2*x^2 + 2*x + 2
over Finite Field of size 3
sage: E.short_weierstrass_model(complete_cube=False).is_isomorphic(E)
True
```

torsion_polynomial(*m*, *x=None*, *two_torsion_multiplicity=2*, *force_evaluate=None*)

Return the m^{th} division polynomial of this elliptic curve evaluated at x.

The division polynomial is cached if x is `None`.

INPUT:

- `m` – positive integer.
- `x` – optional ring element to use as the x variable. If x is `None` (omitted), then a new polynomial ring will be constructed over the base ring of the elliptic curve, and its generator will be used as x. Note that x does not need to be a generator of a polynomial ring; any ring element works. This permits fast calculation of the torsion polynomial *evaluated* on any element of a ring.
- `two_torsion_multiplicity` – 0, 1, or 2

  If 0: For even m when x is `None`, a univariate polynomial over the base ring of the curve is returned, which omits factors whose roots are the x-coordinates of the 2-torsion points. When x is not `None`, the evaluation of such a polynomial at x is returned.

  If 2: For even m when x is `None`, a univariate polynomial over the base ring of the curve is returned, which includes a factor of degree 3 whose roots are the x-coordinates of the 2-torsion points. Similarly, when x is not `None`, the evaluation of such a polynomial at x is returned.

  If 1: For even m when x is `None`, a bivariate polynomial over the base ring of the curve is returned, which includes a factor $2y + a_1x + a_3$ having simple zeros at the 2-torsion points. When x is not `None`, it should be a tuple of length 2, and the evaluation of such a polynomial at x is returned.

- `force_evaluate` (optional) – 0, 1, or 2

  By default, this method makes use of previously cached generic division polynomials to compute the value of the polynomial at a given element x whenever it appears beneficial to do so. Explicitly setting this flag overrides the default behavior.

  Note that the complexity of evaluating a generic division polynomial scales much worse than that of computing the value at a point directly (using the recursive formulas), hence setting this flag can be detrimental to performance.

  If 0: Do not use cached generic division polynomials.

  If 1: If the generic division polynomial for this m has been cached before, evaluate it at x to compute the result.

  If 2: Compute the value at x by evaluating the generic division polynomial. If the generic m-division polynomial has not yet been cached, compute and cache it first.

EXAMPLES:

```
sage: E = EllipticCurve([0,0,1,-1,0])
sage: E.division_polynomial(1)
1
sage: E.division_polynomial(2, two_torsion_multiplicity=0)
1
sage: E.division_polynomial(2, two_torsion_multiplicity=1)
2*y + 1
sage: E.division_polynomial(2, two_torsion_multiplicity=2)
4*x^3 - 4*x + 1
sage: E.division_polynomial(2)
4*x^3 - 4*x + 1
sage: [E.division_polynomial(3, two_torsion_multiplicity=i) for i in range(3)]
[3*x^4 - 6*x^2 + 3*x - 1, 3*x^4 - 6*x^2 + 3*x - 1, 3*x^4 - 6*x^2 + 3*x - 1]
sage: [type(E.division_polynomial(3, two_torsion_multiplicity=i)) for i in␣
↪range(3)]
[<... 'sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_
↪flint'>,
 <... 'sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_
↪libsingular'>,
 <... 'sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_
↪flint'>]
```

```
sage: E = EllipticCurve([0, -1, 1, -10, -20])
sage: R.<z> = PolynomialRing(QQ)
sage: E.division_polynomial(4, z, 0)
2*z^6 - 4*z^5 - 100*z^4 - 790*z^3 - 210*z^2 - 1496*z - 5821
sage: E.division_polynomial(4, z)
8*z^9 - 24*z^8 - 464*z^7 - 2758*z^6 + 6636*z^5 + 34356*z^4
 + 53510*z^3 + 99714*z^2 + 351024*z + 459859
```

This does not work, since when two_torsion_multiplicity is 1, we compute a bivariate polynomial, and must evaluate at a tuple of length 2:

```
sage: E.division_polynomial(4,z,1)
Traceback (most recent call last):
...
ValueError: x should be a tuple of length 2 (or None)
when two_torsion_multiplicity is 1
sage: R.<z,w> = PolynomialRing(QQ, 2)
sage: E.division_polynomial(4, (z,w), 1).factor()
(2*w + 1) * (2*z^6 - 4*z^5 - 100*z^4 - 790*z^3 - 210*z^2 - 1496*z - 5821)
```

We can also evaluate this bivariate polynomial at a point:

```
sage: P = E(5,5)
sage: E.division_polynomial(4,P,two_torsion_multiplicity=1)
-1771561
```

two_division_polynomial (*x=None*)

Return the 2-division polynomial of this elliptic curve evaluated at `x`.

INPUT:

- `x` – optional ring element to use as the x variable. If `x` is `None`, then a new polynomial ring will be constructed over the base ring of the elliptic curve, and its generator will be used as `x`. Note that `x` does not need to be a generator of a polynomial ring; any ring element is acceptable. This permits fast calculation of the torsion polynomial *evaluated* on any element of a ring.

EXAMPLES:

```
sage: E = EllipticCurve('5077a1')
sage: E.two_division_polynomial()
4*x^3 - 28*x + 25
sage: E = EllipticCurve(GF(3^2,'a'), [1,1,1,1,1])                          #␣
↪needs sage.rings.finite_rings
sage: E.two_division_polynomial()                                           #␣
↪needs sage.rings.finite_rings
x^3 + 2*x^2 + 2
sage: E.two_division_polynomial().roots()                                   #␣
↪needs sage.rings.finite_rings
[(2, 1), (2*a, 1), (a + 2, 1)]
```

sage.schemes.elliptic_curves.ell_generic.**is_EllipticCurve**(*x*)

Utility function to test if `x` is an instance of an Elliptic Curve class.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_generic import is_EllipticCurve
sage: E = EllipticCurve([1,2,3/4,7,19])
sage: is_EllipticCurve(E)
doctest:warning...
DeprecationWarning: The function is_EllipticCurve is deprecated; use 'isinstance(.
↪.., EllipticCurve_generic)' instead.
See https://github.com/sagemath/sage/issues/38022 for details.
True
sage: is_EllipticCurve(0)
False
```

CHAPTER

FIVE

ELLIPTIC CURVES OVER A GENERAL FIELD

This module defines the class *EllipticCurve_field*, based on EllipticCurve_generic, for elliptic curves over general fields.

class sage.schemes.elliptic_curves.ell_field.**EllipticCurve_field**(*R*, *data*, *category=None*)

Bases: *EllipticCurve_generic*, ProjectivePlaneCurve_field

Constructor for elliptic curves over fields.

Identical to the constructor for elliptic curves over general rings, except for setting the default category to AbelianVarieties.

EXAMPLES:

```
sage: E = EllipticCurve(QQ, [1,1])
sage: E.category()
Category of abelian varieties over Rational Field
sage: E = EllipticCurve(GF(101), [1,1])
sage: E.category()
Category of abelian varieties over Finite Field of size 101
```

base_field()

Return the base ring of the elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve(GF(49, 'a'), [3,5])                                    #␣
↪needs sage.rings.finite_rings
sage: E.base_ring()                                                            #␣
↪needs sage.rings.finite_rings
Finite Field in a of size 7^2
```

```
sage: E = EllipticCurve([1,1])
sage: E.base_ring()
Rational Field
```

```
sage: E = EllipticCurve(ZZ, [3,5])
sage: E.base_ring()
Integer Ring
```

descend_to(*K*, *f=None*)

Given an elliptic curve self defined over a field L and a subfield K of L, return all elliptic curves over K which are isomorphic over L to self.

INPUT:

- K – a field which embeds into the base field L of self.
- f (optional) – an embedding of K into L. Ignored if K is $\mathbf{Q}$.

OUTPUT:

A list (possibly empty) of elliptic curves defined over K which are isomorphic to self over L, up to isomorphism over K.

Note: Currently only implemented over number fields. To extend to other fields of characteristic not 2 or 3, what is needed is a method giving the preimages in $K^*/(K^*)^m$ of an element of the base field, for $m = 2, 4, 6$.

EXAMPLES:

```
sage: E = EllipticCurve([1,2,3,4,5])
sage: E.descend_to(ZZ)
Traceback (most recent call last):
...
TypeError: Input must be a field.
```

```
sage: # needs sage.rings.number_field
sage: F.<b> = QuadraticField(23)
sage: x = polygen(ZZ, 'x')
sage: G.<a> = F.extension(x^3 + 5)
sage: E = EllipticCurve(j=1728*b).change_ring(G)
sage: EF = E.descend_to(F); EF
[Elliptic Curve defined by y^2 = x^3 + (27*b-621)*x + (-1296*b+2484)
  over Number Field in b with defining polynomial x^2 - 23
  with b = 4.795831523312720?]
sage: all(Ei.change_ring(G).is_isomorphic(E) for Ei in EF)
True
```

```
sage: # needs sage.rings.number_field
sage: L.<a> = NumberField(x^4 - 7)
sage: K.<b> = NumberField(x^2 - 7, embedding=a^2)
sage: E = EllipticCurve([a^6, 0])
sage: EK = E.descend_to(K); EK
[Elliptic Curve defined by y^2 = x^3 + b*x over Number Field in b
  with defining polynomial x^2 - 7 with b = a^2,
 Elliptic Curve defined by y^2 = x^3 + 7*b*x over Number Field in b
  with defining polynomial x^2 - 7 with b = a^2]
sage: all(Ei.change_ring(L).is_isomorphic(E) for Ei in EK)
True
```

```
sage: K.<a> = QuadraticField(17)                                    #␣
↪needs sage.rings.number_field
sage: E = EllipticCurve(j=2*a)                                      #␣
↪needs sage.rings.number_field
sage: E.descend_to(QQ)                                              #␣
↪needs sage.rings.number_field
[]
```

division_field(*n*, *names='t'*, *map=False*, ***kwds*)

Given an elliptic curve over a number field or finite field F and a positive integer n, construct the n-division field $F(E[n])$.

The n-division field is the smallest extension of F over which all n-torsion points of E are defined.

INPUT:

- n – a positive integer
- `names` – (default: `'t'`) a variable name for the division field
- `map` – (default: `False`) also return an embedding of the *base_field()* into the resulting field
- `kwds` – additional keyword arguments passed to `splitting_field()`

OUTPUT:

If `map` is `False`, the division field K as an absolute number field or a finite field. If `map` is `True`, a tuple (K, ϕ) where ϕ is an embedding of the base field in the division field K.

Warning: This can take a very long time when the degree of the division field is large (e.g. when n is large or when the Galois representation is surjective). The `simplify` flag also has a big influence on the running time over number fields: sometimes `simplify=False` is faster, sometimes the default `simplify=True` is faster.

EXAMPLES:

The 2-division field is the same as the splitting field of the 2-division polynomial (therefore, it has degree 1, 2, 3 or 6):

```
sage: # needs sage.rings.number_field
sage: E = EllipticCurve('15a1')
sage: K.<b> = E.division_field(2); K
Number Field in b with defining polynomial x
sage: E = EllipticCurve('14a1')
sage: K.<b> = E.division_field(2); K
Number Field in b with defining polynomial x^2 + 5*x + 92
sage: E = EllipticCurve('196b1')
sage: K.<b> = E.division_field(2); K
Number Field in b with defining polynomial x^3 + x^2 - 114*x - 127
sage: E = EllipticCurve('19a1')
sage: K.<b> = E.division_field(2); K
Number Field in b with defining polynomial
 x^6 + 10*x^5 + 24*x^4 - 212*x^3 + 1364*x^2 + 24072*x + 104292
```

For odd primes n, the division field is either the splitting field of the n-division polynomial, or a quadratic extension of it.

```
sage: # needs sage.rings.number_field
sage: E = EllipticCurve('50a1')
sage: F.<a> = E.division_polynomial(3).splitting_field(simplify_all=True); F
Number Field in a
 with defining polynomial x^6 - 3*x^5 + 4*x^4 - 3*x^3 - 2*x^2 + 3*x + 3
sage: K.<b> = E.division_field(3, simplify_all=True); K
Number Field in b
 with defining polynomial x^6 - 3*x^5 + 4*x^4 - 3*x^3 - 2*x^2 + 3*x + 3
```

If we take any quadratic twist, the splitting field of the 3-division polynomial remains the same, but the 3-division field becomes a quadratic extension:

```
sage: # needs sage.rings.number_field
sage: E = E.quadratic_twist(5)  # 50b3
sage: F.<a> = E.division_polynomial(3).splitting_field(simplify_all=True); F
Number Field in a
 with defining polynomial x^6 - 3*x^5 + 4*x^4 - 3*x^3 - 2*x^2 + 3*x + 3
sage: K.<b> = E.division_field(3, simplify_all=True); K
Number Field in b with defining polynomial x^12 - 3*x^11 + 8*x^10 - 15*x^9
 + 30*x^8 - 63*x^7 + 109*x^6 - 144*x^5 + 150*x^4 - 120*x^3 + 68*x^2 - 24*x + 4
```

Try another quadratic twist, this time over a subfield of F:

```
sage: # needs sage.rings.number_field
sage: G.<c>,_,_ = F.subfields(3)[0]
sage: E = E.base_extend(G).quadratic_twist(c); E
Elliptic Curve defined
 by y^2 = x^3 + 5*a0*x^2 + (-200*a0^2)*x + (-42000*a0^2+42000*a0+126000)
 over Number Field in a0 with defining polynomial x^3 - 3*x^2 + 3*x + 9
sage: K.<b> = E.division_field(3, simplify_all=True); K
Number Field in b with defining polynomial x^12 - 25*x^10 + 130*x^8 + 645*x^6␣
↪+ 1050*x^4 + 675*x^2 + 225
```

Some higher-degree examples:

```
sage: # needs sage.rings.number_field
sage: E = EllipticCurve('11a1')
sage: K.<b> = E.division_field(2); K
Number Field in b with defining polynomial
 x^6 + 2*x^5 - 48*x^4 - 436*x^3 + 1668*x^2 + 28792*x + 73844
sage: K.<b> = E.division_field(3); K        # long time
Number Field in b with defining polynomial x^48 ...
sage: K.<b> = E.division_field(5); K
Number Field in b with defining polynomial x^4 - x^3 + x^2 - x + 1
sage: E.division_field(5, 'b', simplify=False)
Number Field in b with defining polynomial x^4 + x^3 + 11*x^2 + 41*x + 101
sage: E.base_extend(K).torsion_subgroup()   # long time
Torsion Subgroup isomorphic to Z/5 + Z/5 associated to the Elliptic Curve
 defined by y^2 + y = x^3 + (-1)*x^2 + (-10)*x + (-20)
 over Number Field in b with defining polynomial x^4 - x^3 + x^2 - x + 1

sage: # needs sage.rings.number_field
sage: E = EllipticCurve('27a1')
sage: K.<b> = E.division_field(3); K
Number Field in b with defining polynomial x^2 + 3*x + 9
sage: K.<b> = E.division_field(2); K
Number Field in b with defining polynomial
 x^6 + 6*x^5 + 24*x^4 - 52*x^3 - 228*x^2 + 744*x + 3844
sage: K.<b> = E.division_field(2, simplify_all=True); K
Number Field in b with defining polynomial x^6 - 3*x^5 + 5*x^3 - 3*x + 1
sage: K.<b> = E.division_field(5); K        # long time
Number Field in b with defining polynomial x^48 ...
sage: K.<b> = E.division_field(7); K        # long time
Number Field in b with defining polynomial x^72 ...
```

Over a number field:

```
sage: # needs sage.rings.number_field
sage: R.<x> = PolynomialRing(QQ)
```

(continues on next page)

(continued from previous page)

```
sage: K.<i> = NumberField(x^2 + 1)
sage: E = EllipticCurve([0,0,0,0,i])
sage: L.<b> = E.division_field(2); L
Number Field in b with defining polynomial x^4 - x^2 + 1
sage: L.<b>, phi = E.division_field(2, map=True); phi
Ring morphism:
  From: Number Field in i with defining polynomial x^2 + 1
  To:   Number Field in b with defining polynomial x^4 - x^2 + 1
  Defn: i |--> -b^3
sage: L.<b>, phi = E.division_field(3, map=True)
sage: L
Number Field in b with defining polynomial x^24 - 6*x^22 - 12*x^21
 - 21*x^20 + 216*x^19 + 48*x^18 + 804*x^17 + 1194*x^16 - 13488*x^15
 + 21222*x^14 + 44196*x^13 - 47977*x^12 - 102888*x^11 + 173424*x^10
 - 172308*x^9 + 302046*x^8 + 252864*x^7 - 931182*x^6 + 180300*x^5
 + 879567*x^4 - 415896*x^3 + 1941012*x^2 + 650220*x + 443089
sage: phi
Ring morphism:
  From: Number Field in i with defining polynomial x^2 + 1
  To:   Number Field in b with defining polynomial x^24 ...
  Defn: i |--> -215621657062634529/183360797284413355040732*b^23 ...
```

Over a finite field:

```
sage: E = EllipticCurve(GF(431^2), [1,0])                                    #␣
↪needs sage.rings.finite_rings
sage: E.division_field(5, map=True)                                          #␣
↪needs sage.rings.finite_rings
(Finite Field in t of size 431^4,
 Ring morphism:
   From: Finite Field in z2 of size 431^2
   To:   Finite Field in t of size 431^4
   Defn: z2 |--> 52*t^3 + 222*t^2 + 78*t + 105)
```

```
sage: E = EllipticCurve(GF(433^2), [1,0])                                    #␣
↪needs sage.rings.finite_rings
sage: K.<v> = E.division_field(7); K                                         #␣
↪needs sage.rings.finite_rings
Finite Field in v of size 433^16
```

It also works for composite orders:

```
sage: E = EllipticCurve(GF(11), [5,5])
sage: E.change_ring(E.division_field(8)).abelian_group().torsion_subgroup(8).
↪invariants()
(8, 8)
sage: E.change_ring(E.division_field(9)).abelian_group().torsion_subgroup(9).
↪invariants()
(9, 9)
sage: E.change_ring(E.division_field(10)).abelian_group().torsion_
↪subgroup(10).invariants()
(10, 10)
sage: E.change_ring(E.division_field(36)).abelian_group().torsion_
↪subgroup(36).invariants()
(36, 36)
sage: E.change_ring(E.division_field(11)).abelian_group().torsion_
```

(continues on next page)

(continued from previous page)

```
↪subgroup(11).invariants()
(11,)
sage: E.change_ring(E.division_field(66)).abelian_group().torsion_
↪subgroup(66).invariants()
(6, 66)
```

…also over number fields:

```
sage: R.<x> = PolynomialRing(QQ)
sage: K.<i> = NumberField(x^2 + 1)
sage: E = EllipticCurve([0,0,0,0,i])
sage: L,emb = E.division_field(6, names='b', map=True); L
Number Field in b with defining polynomial x^24 + 12*x^23 + ...
sage: E.change_ring(emb).torsion_subgroup().invariants()
(6, 6)
```

See also:

To compute a basis of the n-torsion once the base field has been extended, you may use *sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field.torsion_subgroup()* or *sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field.torsion_basis()*.

AUTHORS:

- Jeroen Demeyer (2014-01-06): Issue #11905, use `splitting_field` method, moved from `gal_reps.py`, make it work over number fields.
- Lorenz Panny (2022): extend to finite fields
- Lorenz Panny (2023): extend to composite n.

endomorphism_ring_is_commutative()

Check whether the endomorphism ring of this elliptic curve *over its base field* is commutative.

ALGORITHM: The endomorphism ring is always commutative in characteristic zero. Over finite fields, it is commutative if and only if the Frobenius endomorphism is not in **Z**. All elliptic curves with non-commutative endomorphism ring are supersingular. (The converse holds over the algebraic closure, but here we consider endomorphisms *over the field of definition*.)

EXAMPLES:

```
sage: EllipticCurve(QQ, [1,1]).endomorphism_ring_is_commutative()
True
sage: EllipticCurve(QQ, [1,0]).endomorphism_ring_is_commutative()
True
sage: EllipticCurve(GF(19), [1,1]).endomorphism_ring_is_commutative()
True
sage: EllipticCurve(GF(19^2), [1,1]).endomorphism_ring_is_commutative()
True
sage: EllipticCurve(GF(19), [1,0]).endomorphism_ring_is_commutative()
True
sage: EllipticCurve(GF(19^2), [1,0]).endomorphism_ring_is_commutative()
False
sage: EllipticCurve(GF(19^3), [1,0]).endomorphism_ring_is_commutative()
True
```

genus()

Return 1 for elliptic curves.

EXAMPLES:

```
sage: E = EllipticCurve(GF(3), [0, -1, 0, -346, 2652])
sage: E.genus()
1

sage: R = FractionField(QQ['z'])
sage: E = EllipticCurve(R, [0, -1, 0, -346, 2652])
sage: E.genus()
1
```

hasse_invariant()

Return the Hasse invariant of this elliptic curve.

OUTPUT:

The Hasse invariant of this elliptic curve, as an element of the base field. This is only defined over fields of positive characteristic, and is an element of the field which is zero if and only if the curve is supersingular. Over a field of characteristic zero, where the Hasse invariant is undefined, a `ValueError` is raised.

EXAMPLES:

```
sage: E = EllipticCurve([Mod(1,2), Mod(1,2), 0, 0, Mod(1,2)])
sage: E.hasse_invariant()
1
sage: E = EllipticCurve([0, 0, Mod(1,3), Mod(1,3), Mod(1,3)])
sage: E.hasse_invariant()
0
sage: E = EllipticCurve([0, 0, Mod(1,5), 0, Mod(2,5)])
sage: E.hasse_invariant()
0
sage: E = EllipticCurve([0, 0, Mod(1,5), Mod(1,5), Mod(2,5)])
sage: E.hasse_invariant()
2
```

Some examples over larger fields:

```
sage: # needs sage.rings.finite_rings
sage: EllipticCurve(GF(101), [0,0,0,0,1]).hasse_invariant()
0
sage: EllipticCurve(GF(101), [0,0,0,1,1]).hasse_invariant()
98
sage: EllipticCurve(GF(103), [0,0,0,0,1]).hasse_invariant()
20
sage: EllipticCurve(GF(103), [0,0,0,1,1]).hasse_invariant()
17
sage: F.<a> = GF(107^2)
sage: EllipticCurve(F, [0,0,0,a,1]).hasse_invariant()
62*a + 75
sage: EllipticCurve(F, [0,0,0,0,a]).hasse_invariant()
0
```

Over fields of characteristic zero, the Hasse invariant is undefined:

```
sage: E = EllipticCurve([0,0,0,0,1])
sage: E.hasse_invariant()
Traceback (most recent call last):
...
ValueError: Hasse invariant only defined in positive characteristic
```

is_isogenous(*other*, *field=None*)

Return whether or not self is isogenous to other.

INPUT:

- `other` – another elliptic curve.
- `field` (default None) – Currently not implemented. A field containing the base fields of the two elliptic curves onto which the two curves may be extended to test if they are isogenous over this field. By default `is_isogenous` will not try to find this field unless one of the curves can be be extended into the base field of the other, in which case it will test over the larger base field.

OUTPUT:

(bool) True if there is an isogeny from curve `self` to curve `other` defined over `field`.

METHOD:

Over general fields this is only implemented in trivial cases.

EXAMPLES:

```
sage: E1 = EllipticCurve(CC, [1,18]); E1
Elliptic Curve defined by y^2 = x^3 + 1.00000000000000*x + 18.0000000000000
 over Complex Field with 53 bits of precision
sage: E2 = EllipticCurve(CC, [2,7]); E2
Elliptic Curve defined by y^2 = x^3 + 2.00000000000000*x + 7.00000000000000
 over Complex Field with 53 bits of precision
sage: E1.is_isogenous(E2)
Traceback (most recent call last):
...
NotImplementedError: Only implemented for isomorphic curves over general␣
↪fields.

sage: E1 = EllipticCurve(Frac(PolynomialRing(ZZ,'t')), [2,19]); E1
Elliptic Curve defined by y^2 = x^3 + 2*x + 19
 over Fraction Field of Univariate Polynomial Ring in t over Integer Ring
sage: E2 = EllipticCurve(CC, [23,4]); E2
Elliptic Curve defined by y^2 = x^3 + 23.0000000000000*x + 4.00000000000000
 over Complex Field with 53 bits of precision
sage: E1.is_isogenous(E2)
Traceback (most recent call last):
...
NotImplementedError: Only implemented for isomorphic curves over general␣
↪fields.
```

is_quadratic_twist(*other*)

Determine whether this curve is a quadratic twist of another.

INPUT:

- `other` – an elliptic curve with the same base field as `self`.

OUTPUT:

Either 0, if the curves are not quadratic twists, or D if `other` is `self.quadratic_twist(D)` (up to isomorphism). If `self` and `other` are isomorphic, returns 1.

If the curves are defined over $\mathbf{Q}$, the output D is a squarefree integer.

Note: Not fully implemented in characteristic 2, or in characteristic 3 when both j-invariants are 0.

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: Et = E.quadratic_twist(-24)
sage: E.is_quadratic_twist(Et)
-6

sage: E1 = EllipticCurve([0,0,1,0,0])
sage: E1.j_invariant()
0
sage: E2 = EllipticCurve([0,0,0,0,2])
sage: E1.is_quadratic_twist(E2)
2
sage: E1.is_quadratic_twist(E1)
1
sage: type(E1.is_quadratic_twist(E1)) == type(E1.is_quadratic_twist(E2))    #␣
↪Issue #6574
True
```

```
sage: E1 = EllipticCurve([0,0,0,1,0])
sage: E1.j_invariant()
1728
sage: E2 = EllipticCurve([0,0,0,2,0])
sage: E1.is_quadratic_twist(E2)
0
sage: E2 = EllipticCurve([0,0,0,25,0])
sage: E1.is_quadratic_twist(E2)
5
```

```
sage: # needs sage.rings.finite_rings
sage: F = GF(101)
sage: E1 = EllipticCurve(F, [4,7])
sage: E2 = E1.quadratic_twist()
sage: D = E1.is_quadratic_twist(E2); D != 0
True
sage: F = GF(101)
sage: E1 = EllipticCurve(F, [4,7])
sage: E2 = E1.quadratic_twist()
sage: D = E1.is_quadratic_twist(E2)
sage: E1.quadratic_twist(D).is_isomorphic(E2)
True
sage: E1.is_isomorphic(E2)
False
sage: F2 = GF(101^2,'a')
sage: E1.change_ring(F2).is_isomorphic(E2.change_ring(F2))
True
```

A characteristic 3 example:

```
sage: # needs sage.rings.finite_rings
sage: F = GF(3^5,'a')
sage: E1 = EllipticCurve_from_j(F(1))
sage: E2 = E1.quadratic_twist(-1)
sage: D = E1.is_quadratic_twist(E2); D != 0
True
sage: E1.quadratic_twist(D).is_isomorphic(E2)
True
```

```
sage: # needs sage.rings.finite_rings
sage: E1 = EllipticCurve_from_j(F(0))
sage: E2 = E1.quadratic_twist()
sage: D = E1.is_quadratic_twist(E2); D
1
sage: E1.is_isomorphic(E2)
True
```

is_quartic_twist(*other*)

Determine whether this curve is a quartic twist of another.

INPUT:

- `other` – an elliptic curves with the same base field as self.

OUTPUT:

Either 0, if the curves are not quartic twists, or D if `other` is `self.quartic_twist(D)` (up to isomorphism). If `self` and `other` are isomorphic, returns 1.

Note: Not fully implemented in characteristics 2 or 3.

EXAMPLES:

```
sage: E = EllipticCurve_from_j(GF(13)(1728))
sage: E1 = E.quartic_twist(2)
sage: D = E.is_quartic_twist(E1); D!=0
True
sage: E.quartic_twist(D).is_isomorphic(E1)
True
```

```
sage: E = EllipticCurve_from_j(1728)
sage: E1 = E.quartic_twist(12345)
sage: D = E.is_quartic_twist(E1); D
15999120
sage: (D/12345).is_perfect_power(4)
True
```

is_sextic_twist(*other*)

Determine whether this curve is a sextic twist of another.

INPUT:

- `other` – an elliptic curves with the same base field as self.

OUTPUT:

Either 0, if the curves are not sextic twists, or D if `other` is `self.sextic_twist(D)` (up to isomorphism). If `self` and `other` are isomorphic, returns 1.

Note: Not fully implemented in characteristics 2 or 3.

EXAMPLES:

```
sage: E = EllipticCurve_from_j(GF(13)(0))
sage: E1 = E.sextic_twist(2)
```

(continues on next page)

(continued from previous page)

```
sage: D = E.is_sextic_twist(E1); D != 0
True
sage: E.sextic_twist(D).is_isomorphic(E1)
True
```

```
sage: E = EllipticCurve_from_j(0)
sage: E1 = E.sextic_twist(12345)
sage: D = E.is_sextic_twist(E1); D
575968320
sage: (D/12345).is_perfect_power(6)
True
```

isogenies_prime_degree(*l=None*, *max_l=31*)

Return a list of all separable isogenies of given prime degree(s) with domain equal to `self`, which are defined over the base field of `self`.

INPUT:

- `l` – a prime or a list of primes.
- `max_l` – (default: 31) a bound on the primes to be tested. This is only used if `l` is None.

OUTPUT:

(list) All separable l-isogenies for the given l with domain self.

ALGORITHM:

Calls the generic function *isogenies_prime_degree()*. This is generic code, valid for all fields. It requires that certain operations have been implemented over the base field, such as root-finding for univariate polynomials.

EXAMPLES:

Examples over finite fields:

```
sage: # needs sage.libs.pari
sage: E = EllipticCurve(GF(next_prime(1000000)), [7,8])
sage: E.isogenies_prime_degree(2)
[Isogeny of degree 2
  from Elliptic Curve defined by y^2 = x^3 + 7*x + 8 over Finite Field of␣
↪size 1000003
    to Elliptic Curve defined by y^2 = x^3 + 970389*x + 794257 over Finite␣
↪Field of size 1000003,
 Isogeny of degree 2
  from Elliptic Curve defined by y^2 = x^3 + 7*x + 8 over Finite Field of␣
↪size 1000003
    to Elliptic Curve defined by y^2 = x^3 + 29783*x + 206196 over Finite␣
↪Field of size 1000003,
 Isogeny of degree 2
  from Elliptic Curve defined by y^2 = x^3 + 7*x + 8 over Finite Field of␣
↪size 1000003
    to Elliptic Curve defined by y^2 = x^3 + 999960*x + 78 over Finite Field␣
↪of size 1000003]
sage: E.isogenies_prime_degree(3)
[]
sage: E.isogenies_prime_degree(5)
[]
sage: E.isogenies_prime_degree(7)
```

(continues on next page)

(continued from previous page)

```
[]
sage: E.isogenies_prime_degree(11)
[]
sage: E.isogenies_prime_degree(13)
[Isogeny of degree 13
  from Elliptic Curve defined by y^2 = x^3 + 7*x + 8 over Finite Field of␣
↪size 1000003
    to Elliptic Curve defined by y^2 = x^3 + 878063*x + 845666 over Finite␣
↪Field of size 1000003,
 Isogeny of degree 13
  from Elliptic Curve defined by y^2 = x^3 + 7*x + 8 over Finite Field of␣
↪size 1000003
    to Elliptic Curve defined by y^2 = x^3 + 375648*x + 342776 over Finite␣
↪Field of size 1000003]
sage: E.isogenies_prime_degree(max_l=13)
[Isogeny of degree 2
  from Elliptic Curve defined by y^2 = x^3 + 7*x + 8 over Finite Field of␣
↪size 1000003
    to Elliptic Curve defined by y^2 = x^3 + 970389*x + 794257 over Finite␣
↪Field of size 1000003,
 Isogeny of degree 2
  from Elliptic Curve defined by y^2 = x^3 + 7*x + 8 over Finite Field of␣
↪size 1000003
    to Elliptic Curve defined by y^2 = x^3 + 29783*x + 206196 over Finite␣
↪Field of size 1000003,
 Isogeny of degree 2
  from Elliptic Curve defined by y^2 = x^3 + 7*x + 8 over Finite Field of␣
↪size 1000003
    to Elliptic Curve defined by y^2 = x^3 + 999960*x + 78 over Finite Field␣
↪of size 1000003,
 Isogeny of degree 13
  from Elliptic Curve defined by y^2 = x^3 + 7*x + 8 over Finite Field of␣
↪size 1000003
    to Elliptic Curve defined by y^2 = x^3 + 878063*x + 845666 over Finite␣
↪Field of size 1000003,
 Isogeny of degree 13
  from Elliptic Curve defined by y^2 = x^3 + 7*x + 8 over Finite Field of␣
↪size 1000003
    to Elliptic Curve defined by y^2 = x^3 + 375648*x + 342776 over Finite␣
↪Field of size 1000003]
sage: E.isogenies_prime_degree()  # Default limit of 31
[Isogeny of degree 2
  from Elliptic Curve defined by y^2 = x^3 + 7*x + 8 over Finite Field of␣
↪size 1000003
    to Elliptic Curve defined by y^2 = x^3 + 970389*x + 794257 over Finite␣
↪Field of size 1000003,
 Isogeny of degree 2
  from Elliptic Curve defined by y^2 = x^3 + 7*x + 8 over Finite Field of␣
↪size 1000003
    to Elliptic Curve defined by y^2 = x^3 + 29783*x + 206196 over Finite␣
↪Field of size 1000003,
 Isogeny of degree 2
  from Elliptic Curve defined by y^2 = x^3 + 7*x + 8 over Finite Field of␣
↪size 1000003
    to Elliptic Curve defined by y^2 = x^3 + 999960*x + 78 over Finite Field␣
↪of size 1000003,
 Isogeny of degree 13
```

(continues on next page)

(continued from previous page)

```
  from Elliptic Curve defined by y^2 = x^3 + 7*x + 8 over Finite Field of␣
↪size 1000003
    to Elliptic Curve defined by y^2 = x^3 + 878063*x + 845666 over Finite␣
↪Field of size 1000003,
 Isogeny of degree 13
  from Elliptic Curve defined by y^2 = x^3 + 7*x + 8 over Finite Field of␣
↪size 1000003
    to Elliptic Curve defined by y^2 = x^3 + 375648*x + 342776 over Finite␣
↪Field of size 1000003,
 Isogeny of degree 17
  from Elliptic Curve defined by y^2 = x^3 + 7*x + 8 over Finite Field of␣
↪size 1000003
    to Elliptic Curve defined by y^2 = x^3 + 347438*x + 594729 over Finite␣
↪Field of size 1000003,
 Isogeny of degree 17
  from Elliptic Curve defined by y^2 = x^3 + 7*x + 8 over Finite Field of␣
↪size 1000003
    to Elliptic Curve defined by y^2 = x^3 + 674846*x + 7392 over Finite␣
↪Field of size 1000003,
 Isogeny of degree 23
  from Elliptic Curve defined by y^2 = x^3 + 7*x + 8 over Finite Field of␣
↪size 1000003
    to Elliptic Curve defined by y^2 = x^3 + 390065*x + 605596 over Finite␣
↪Field of size 1000003]

sage: E = EllipticCurve(GF(17), [2,0])
sage: E.isogenies_prime_degree(3)
[]
sage: E.isogenies_prime_degree(2)
[Isogeny of degree 2
  from Elliptic Curve defined by y^2 = x^3 + 2*x over Finite Field of size 17
  to Elliptic Curve defined by y^2 = x^3 + 9*x over Finite Field of size 17,
 Isogeny of degree 2
  from Elliptic Curve defined by y^2 = x^3 + 2*x over Finite Field of size 17
  to Elliptic Curve defined by y^2 = x^3 + 5*x + 9 over Finite Field of size␣
↪17,
 Isogeny of degree 2
  from Elliptic Curve defined by y^2 = x^3 + 2*x over Finite Field of size 17
  to Elliptic Curve defined by y^2 = x^3 + 5*x + 8 over Finite Field of size␣
↪17]
```

The base field matters, over a field extension we find more isogenies:

```
sage: E = EllipticCurve(GF(13), [2,8])
sage: E.isogenies_prime_degree(max_l=3)
[Isogeny of degree 2
  from Elliptic Curve defined by y^2 = x^3 + 2*x + 8 over Finite Field of␣
↪size 13
    to Elliptic Curve defined by y^2 = x^3 + 7*x + 4 over Finite Field of␣
↪size 13,
 Isogeny of degree 3
  from Elliptic Curve defined by y^2 = x^3 + 2*x + 8 over Finite Field of␣
↪size 13
    to Elliptic Curve defined by y^2 = x^3 + 9*x + 11 over Finite Field of␣
↪size 13]

sage: # needs sage.rings.finite_rings
```

(continues on next page)

(continued from previous page)

```
sage: E = EllipticCurve(GF(13^6), [2,8])
sage: E.isogenies_prime_degree(max_l=3)
[Isogeny of degree 2
  from Elliptic Curve defined by y^2 = x^3 + 2*x + 8 over Finite Field in z6␣
↪of size 13^6
    to Elliptic Curve defined by y^2 = x^3 + 7*x + 4 over Finite Field in z6␣
↪of size 13^6,
 Isogeny of degree 2
  from Elliptic Curve defined by y^2 = x^3 + 2*x + 8 over Finite Field in z6␣
↪of size 13^6
    to Elliptic Curve defined by y^2 = x^3 + (2*z6^5+6*z6^4+9*z6^3+8*z6+7)*x␣
↪+ (3*z6^5+9*z6^4+7*z6^3+12*z6+7) over Finite Field in z6 of size 13^6,
 Isogeny of degree 2
  from Elliptic Curve defined by y^2 = x^3 + 2*x + 8 over Finite Field in z6␣
↪of size 13^6
    to Elliptic Curve defined by y^2 = x^3 + (11*z6^5+7*z6^4+4*z6^3+5*z6+9)*x␣
↪+ (10*z6^5+4*z6^4+6*z6^3+z6+10) over Finite Field in z6 of size 13^6,
 Isogeny of degree 3
  from Elliptic Curve defined by y^2 = x^3 + 2*x + 8 over Finite Field in z6␣
↪of size 13^6
    to Elliptic Curve defined by y^2 = x^3 + 9*x + 11 over Finite Field in z6␣
↪of size 13^6,
 Isogeny of degree 3
  from Elliptic Curve defined by y^2 = x^3 + 2*x + 8 over Finite Field in z6␣
↪of size 13^6
    to Elliptic Curve defined by y^2 = x^3 + (3*z6^5+5*z6^4+8*z6^3+11*z6^
↪2+5*z6+12)*x + (12*z6^5+6*z6^4+8*z6^3+4*z6^2+7*z6+6) over Finite Field in␣
↪z6 of size 13^6,
 Isogeny of degree 3
  from Elliptic Curve defined by y^2 = x^3 + 2*x + 8 over Finite Field in z6␣
↪of size 13^6
    to Elliptic Curve defined by y^2 = x^3 + (7*z6^4+12*z6^3+7*z6^2+4)*x +␣
↪(6*z6^5+10*z6^3+12*z6^2+10*z6+8) over Finite Field in z6 of size 13^6,
 Isogeny of degree 3
  from Elliptic Curve defined by y^2 = x^3 + 2*x + 8 over Finite Field in z6␣
↪of size 13^6
    to Elliptic Curve defined by y^2 = x^3 + (10*z6^5+z6^4+6*z6^3+8*z6^
↪2+8*z6)*x + (8*z6^5+7*z6^4+8*z6^3+10*z6^2+9*z6+7) over Finite Field in z6␣
↪of size 13^6]
```

If the degree equals the characteristic, we find only separable isogenies:

```
sage: E = EllipticCurve(GF(13), [2,8])
sage: E.isogenies_prime_degree(13)
[Isogeny of degree 13
  from Elliptic Curve defined by y^2 = x^3 + 2*x + 8 over Finite Field of␣
↪size 13
    to Elliptic Curve defined by y^2 = x^3 + 6*x + 5 over Finite Field of␣
↪size 13]
sage: E = EllipticCurve(GF(5), [1,1])
sage: E.isogenies_prime_degree(5)
[Isogeny of degree 5
  from Elliptic Curve defined by y^2 = x^3 + x + 1 over Finite Field of size 5
    to Elliptic Curve defined by y^2 = x^3 + x + 4 over Finite Field of size␣
↪5]

sage: # needs sage.rings.finite_rings
```

(continues on next page)

(continued from previous page)

```
sage: k.<a> = GF(3^4)
sage: E = EllipticCurve(k, [0,1,0,0,a])
sage: E.isogenies_prime_degree(3)
[Isogeny of degree 3
  from Elliptic Curve defined by y^2 = x^3 + x^2 + a
       over Finite Field in a of size 3^4
    to Elliptic Curve defined by y^2 = x^3 + x^2 + (2*a^3+a^2+2)*x + (a^2+2)
       over Finite Field in a of size 3^4]
```

In the supersingular case, there are no separable isogenies of degree equal to the characteristic:

```
sage: E = EllipticCurve(GF(5), [0,1])
sage: E.isogenies_prime_degree(5)
[]
```

An example over a rational function field:

```
sage: R.<t> = GF(5)[]
sage: K = R.fraction_field()
sage: E = EllipticCurve(K, [1, t^5])
sage: E.isogenies_prime_degree(5)
[Isogeny of degree 5
  from Elliptic Curve defined by y^2 = x^3 + x + t^5 over Fraction Field
       of Univariate Polynomial Ring in t over Finite Field of size 5
    to Elliptic Curve defined by y^2 = x^3 + x + 4*t over Fraction Field
       of Univariate Polynomial Ring in t over Finite Field of size 5]
```

Examples over number fields (other than QQ):

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: QQroot2.<e> = NumberField(x^2 - 2)
sage: E = EllipticCurve(QQroot2, j=8000)
sage: E.isogenies_prime_degree()
[Isogeny of degree 2
  from Elliptic Curve defined by y^2 = x^3 + (-150528000)*x + (-629407744000)
       over Number Field in e with defining polynomial x^2 - 2
    to Elliptic Curve defined by y^2 = x^3 + (-36750)*x + 2401000
       over Number Field in e with defining polynomial x^2 - 2,
Isogeny of degree 2
  from Elliptic Curve defined by y^2 = x^3 + (-150528000)*x + (-629407744000)
       over Number Field in e with defining polynomial x^2 - 2
    to Elliptic Curve defined by y^2 = x^3 + (220500*e-257250)*x +␣
↪(54022500*e-88837000)
       over Number Field in e with defining polynomial x^2 - 2,
Isogeny of degree 2
  from Elliptic Curve defined by y^2 = x^3 + (-150528000)*x + (-629407744000)
       over Number Field in e with defining polynomial x^2 - 2
    to Elliptic Curve defined by y^2 = x^3 + (-220500*e-257250)*x + (-
↪54022500*e-88837000)
       over Number Field in e with defining polynomial x^2 - 2]
sage: E = EllipticCurve(QQroot2, [1,0,1,4, -6]); E
Elliptic Curve defined by y^2 + x*y + y = x^3 + 4*x + (-6)
 over Number Field in e with defining polynomial x^2 - 2
sage: E.isogenies_prime_degree(2)
[Isogeny of degree 2
  from Elliptic Curve defined by y^2 + x*y + y = x^3 + 4*x + (-6)
```

(continues on next page)

(continued from previous page)

```
       over Number Field in e with defining polynomial x^2 - 2
    to Elliptic Curve defined by y^2 + x*y + y = x^3 + (-36)*x + (-70)
       over Number Field in e with defining polynomial x^2 - 2]
sage: E.isogenies_prime_degree(3)
[Isogeny of degree 3
  from Elliptic Curve defined by y^2 + x*y + y = x^3 + 4*x + (-6)
       over Number Field in e with defining polynomial x^2 - 2
    to Elliptic Curve defined by y^2 + x*y + y = x^3 + (-1)*x
       over Number Field in e with defining polynomial x^2 - 2,
 Isogeny of degree 3
  from Elliptic Curve defined by y^2 + x*y + y = x^3 + 4*x + (-6)
       over Number Field in e with defining polynomial x^2 - 2
    to Elliptic Curve defined by y^2 + x*y + y = x^3 + (-171)*x + (-874)
       over Number Field in e with defining polynomial x^2 - 2]
```

These are not implemented yet:

```
sage: E = EllipticCurve(QQbar, [1,18]); E                                   #␣
↪needs sage.rings.number_field
Elliptic Curve defined by y^2 = x^3 + x + 18 over Algebraic Field
sage: E.isogenies_prime_degree()                                            #␣
↪needs sage.rings.number_field
Traceback (most recent call last):
...
NotImplementedError: This code could be implemented for QQbar, but has not␣
↪been yet.

sage: E = EllipticCurve(CC, [1,18]); E
Elliptic Curve defined by y^2 = x^3 + 1.00000000000000*x + 18.0000000000000
over Complex Field with 53 bits of precision
sage: E.isogenies_prime_degree(11)
Traceback (most recent call last):
...
NotImplementedError: This code could be implemented for general complex␣
↪fields,
but has not been yet.
```

isogeny(*kernel*, *codomain=None*, *degree=None*, *model=None*, *check=True*, *algorithm=None*, *velu_sqrt_bound=None*)

Return an elliptic-curve isogeny from this elliptic curve.

The isogeny can be specified in two ways, by passing either a polynomial or a set of torsion points. The methods used are:

- Factored Isogenies (see *hom_composite*): Given a point, or a list of points which generate a composite-order subgroup, decomposes the isogeny into prime-degree steps. This can be used to construct isogenies of extremely large, smooth degree. When applicable, this algorithm is selected as default (see below). After factoring the degree single isogenies are computed using the other methods. This algorithm is selected using `algorithm="factored"`.
- Vélu's Formulas: Vélu's original formulas for computing isogenies. This algorithm is selected by giving as the `kernel` parameter a single point generating a finite subgroup.
- Kohel's Formulas: Kohel's original formulas for computing isogenies. This algorithm is selected by giving as the `kernel` parameter a monic polynomial (or a coefficient list in little endian) which will define the kernel of the isogeny. Kohel's algorithm is currently only implemented for cyclic isogenies, with the exception of [2].

- √élu Algorithm (see *hom_velusqrt*): A variant of Vélu's formulas with essentially square-root instead of linear complexity (in the degree). Currently only available over finite fields. The input must be a single kernel point of odd order ≥ 5. This algorithm is selected using `algorithm="velusqrt"`.

INPUT:

- `kernel` – a kernel: either a point on this curve, a list of points on this curve, a monic kernel polynomial, or `None`. If initializing from a codomain, this must be `None`.
- `codomain` – an elliptic curve (default: `None`).
  - If `kernel` is `None`, then `degree` must be given as well and the given `codomain` must be the codomain of a cyclic, separable, normalized isogeny of the given degree.
  - If `kernel` is not `None`, then this must be isomorphic to the codomain of the separable isogeny defined by `kernel`; in this case, the isogeny is post-composed with an isomorphism so that the codomain equals the given curve.
- `degree` – an integer (default: `None`).
  - If `kernel` is `None`, then this is the degree of the isogeny from this curve to `codomain`.
  - If `kernel` is not `None`, then this is used to determine whether or not to skip a gcd of the given kernel polynomial with the two-torsion polynomial of this curve.
- `model` – a string (default: `None`). Supported values (cf. *compute_model()*):
  - `"minimal"`: If `self` is a curve over the rationals or over a number field, then the codomain is a global minimal model where this exists.
  - `"short_weierstrass"`: The codomain is a short Weierstrass curve, assuming one exists.
  - `"montgomery"`: The codomain is an (untwisted) Montgomery curve, assuming one exists over this field.
- `check` (default: `True`) – check whether the input is valid. Setting this to `False` can lead to significant speedups.
- `algorithm` – string (optional). The possible choices are:
  - `"velusqrt"`: Use *EllipticCurveHom_velusqrt*.
  - `"factored"`: Use *EllipticCurveHom_composite* to decompose the isogeny into prime-degree steps.
  - `"traditional"`: Use *EllipticCurveIsogeny*.

  When `algorithm` is not specified, and `kernel` is not `None`, an algorithm is selected using the following criteria:
  - if `kernel` is a list of multiple points, `"factored"` is selected.
  - If `kernel` is a single point, or a list containing a single point:
    - if the order of the point is unknown, `"traditional"` is selected.
    - If the order is known and composite, `"factored"` is selected.
    - If the order is known and prime, a choice between `"velusqrt"` and `"traditional"` is done according to the `velu_sqrt_bound` parameter (see below).

  If none of the previous apply, `"traditional"` is selected.
- `velu_sqrt_bound` – an integer (default: `None`). Establish the highest (prime) degree for which the `"traditional"` algorithm should be selected instead of `"velusqrt"`. If `None`, the default value from `_VeluBoundObj` is used. This value is initially set to 1000, but can be modified by the user. If

an integer is supplied and the isogeny computation goes through the "factored" algorithm, the same integer is supplied to each factor.

The degree parameter is not supported when an algorithm is specified.

OUTPUT:

An isogeny between elliptic curves. This is a morphism of curves. (In all cases, the returned object will be an instance of *EllipticCurveHom*.)

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: F = GF(2^5, 'alpha'); alpha = F.gen()
sage: E = EllipticCurve(F, [1,0,1,1,1])
sage: R.<x> = F[]
sage: phi = E.isogeny(x + 1)
sage: phi.rational_maps()
((x^2 + x + 1)/(x + 1), (x^2*y + x)/(x^2 + 1))
```

```
sage: E = EllipticCurve('11a1')
sage: P = E.torsion_points()[1]
sage: E.isogeny(P)
Isogeny of degree 5
 from Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20
      over Rational Field
   to Elliptic Curve defined by y^2 + y = x^3 - x^2 - 7820*x - 263580
      over Rational Field
```

```
sage: E = EllipticCurve(GF(19),[1,1])
sage: P = E(15,3); Q = E(2,12)
sage: (P.order(), Q.order())
(7, 3)
sage: phi = E.isogeny([P,Q]); phi
Composite morphism of degree 21 = 7*3:
  From: Elliptic Curve defined by y^2 = x^3 + x + 1 over Finite Field of size␣
↪19
  To:   Elliptic Curve defined by y^2 = x^3 + x + 1 over Finite Field of size␣
↪19
sage: phi(E.random_point())  # all points defined over GF(19) are in the␣
↪kernel
(0 : 1 : 0)
```

```
sage: E = EllipticCurve(GF(2^32 - 5), [170246996, 2036646110])              #␣
↪needs sage.rings.finite_rings
sage: P = E.lift_x(2)                                                       #␣
↪needs sage.rings.finite_rings
sage: E.isogeny(P, algorithm="factored")                                    #␣
↪needs sage.rings.finite_rings
Composite morphism of degree 1073721825 = 3^4*5^2*11*19*43*59:
  From: Elliptic Curve defined by y^2 = x^3 + 170246996*x + 2036646110
        over Finite Field of size 4294967291
  To:   Elliptic Curve defined by y^2 = x^3 + 272790262*x + 1903695400
        over Finite Field of size 4294967291
```

Not all polynomials define a finite subgroup (Issue #6384):

```
sage: E = EllipticCurve(GF(31), [1,0,0,1,2])
sage: phi = E.isogeny([14,27,4,1])
Traceback (most recent call last):
...
ValueError: the polynomial x^3 + 4*x^2 + 27*x + 14 does not define a finite
subgroup of Elliptic Curve defined by y^2 + x*y = x^3 + x + 2
over Finite Field of size 31
```

Order of the point known and composite:

```
sage: E = EllipticCurve(GF(31), [1,0,0,1,2])
sage: P = E(26, 4)
sage: assert P.order() == 12
sage: print(P._order)
12
sage: E.isogeny(P)
Composite morphism of degree 12 = 2^2*3:
  From: Elliptic Curve defined by y^2 + x*y = x^3 + x + 2 over Finite Field␣
↪of size 31
  To:   Elliptic Curve defined by y^2 + x*y = x^3 + 26*x + 8 over Finite␣
↪Field of size 31
```

`kernel` is a list of points:

```
sage: E = EllipticCurve(GF(31), [1,0,0,1,2])
sage: P = E(21,2)
sage: Q = E(7, 12)
sage: print(P.order())
6
sage: print(Q.order())
2
sage: E.isogeny([P, Q])
Composite morphism of degree 12 = 2*3*2:
  From: Elliptic Curve defined by y^2 + x*y = x^3 + x + 2 over Finite Field␣
↪of size 31
  To:   Elliptic Curve defined by y^2 + x*y = x^3 + 2*x + 26 over Finite␣
↪Field of size 31
```

Multiple ways to set the $velu_sqrt_bound$:

```
sage: E = EllipticCurve_from_j(GF(97)(42))
sage: P = E.gens()[0]*4
sage: print(P.order())
23
sage: E.isogeny(P)
Isogeny of degree 23 from Elliptic Curve defined by y^2 = x^3 + 6*x + 46 over␣
↪Finite Field of size 97 to Elliptic Curve defined by y^2 = x^3 + 72*x + 29␣
↪over Finite Field of size 97
sage: E.isogeny(P, velu_sqrt_bound=10)
Elliptic-curve isogeny (using square-root Vélu) of degree 23:
  From: Elliptic Curve defined by y^2 = x^3 + 6*x + 46 over Finite Field of␣
↪size 97
  To:   Elliptic Curve defined by y^2 = x^3 + 95*x + 68 over Finite Field of␣
↪size 97
sage: from sage.schemes.elliptic_curves.hom_velusqrt import _velu_sqrt_bound
sage: _velu_sqrt_bound.set(10)
sage: E.isogeny(P)
```

(continues on next page)

(continued from previous page)

```
Elliptic-curve isogeny (using square-root Vélu) of degree 23:
  From: Elliptic Curve defined by y^2 = x^3 + 6*x + 46 over Finite Field of␣
↪size 97
  To:   Elliptic Curve defined by y^2 = x^3 + 95*x + 68 over Finite Field of␣
↪size 97
sage: _velu_sqrt_bound.set(1000) # Reset bound
```

If the order of the point is unknown, fall back to `"traditional"`:

```
sage: E = EllipticCurve_from_j(GF(97)(42))
sage: P = E(2, 39)
sage: from sage.schemes.elliptic_curves.hom_velusqrt import _velu_sqrt_bound
sage: _velu_sqrt_bound.set(1)
sage: E.isogeny(P)
Isogeny of degree 46 from Elliptic Curve defined by y^2 = x^3 + 6*x + 46 over␣
↪Finite Field of size 97 to Elliptic Curve defined by y^2 = x^3 + 87*x + 47␣
↪over Finite Field of size 97
sage: _velu_sqrt_bound.set(1000) # Reset bound
```

See also:

- *EllipticCurveHom*
- *EllipticCurveIsogeny*
- *EllipticCurveHom_composite*

isogeny_codomain(*kernel*)

Return the codomain of the isogeny from `self` with given kernel.

INPUT:

- `kernel` – Either a list of points in the kernel of the isogeny, or a kernel polynomial (specified as either a univariate polynomial or a coefficient list.)

OUTPUT:

An elliptic curve, the codomain of the separable normalized isogeny defined by this kernel.

EXAMPLES:

```
sage: E = EllipticCurve('17a1')
sage: R.<x> = QQ[]
sage: E2 = E.isogeny_codomain(x - 11/4); E2
Elliptic Curve defined by y^2 + x*y + y = x^3 - x^2 - 1461/16*x - 19681/64
 over Rational Field
```

isogeny_ell_graph(*l*, *directed=True*, *label_by_j=False*)

Return a graph representing the `l`-degree `K`-isogenies between `K`-isomorphism classes of elliptic curves for `K = self.base_field()`.

INPUT:

- `l` – prime degree of isogenies
- `directed` – boolean (default: `True`); whether to return a directed or undirected graph. In the undirected case, the in-degrees and out-degrees of the vertices must be balanced and therefore the number of out-edges from the vertices corresponding to j-invariants 0 and 1728 (if they are part of the graph) are reduced to match the number of in-edges.

- `label_by_j` – boolean (default: `False`); whether to label graph vertices by the j-invariant corresponding to the isomorphism class of curves. If the j-invariant is not unique in the isogeny class, append `*` to it to indicate a twist. Otherwise, if `False` label vertices by the equation of a representative curve.

OUTPUT: A `Graph` or `DiGraph`.

EXAMPLES:

Ordinary curve over finite extension field of degree 2:

```
sage: # needs sage.graphs sage.rings.finite_rings
sage: x = polygen(ZZ, 'x')
sage: E = EllipticCurve(GF(59^2, "i", x^2 + 1), j=5)
sage: G = E.isogeny_ell_graph(5, directed=False, label_by_j=True); G
Graph on 20 vertices
sage: G.vertices(sort=True)
['1',
 '12',
 ...
 'i + 55']
sage: G.edges(sort=True)
[('1', '28*i + 11', None),
 ('1', '31*i + 11', None),
 ...
 ('8', 'i + 1', None)]
```

Supersingular curve over prime field:

```
sage: # needs sage.graphs sage.rings.finite_rings
sage: E = EllipticCurve(GF(419), j=1728)
sage: G3 = E.isogeny_ell_graph(3, directed=False, label_by_j=True); G3
Graph on 27 vertices
sage: G3.vertices(sort=True)
['0',
 '0*',
 ...
 '98*']
sage: G3.edges(sort=True)
[('0', '0*', None),
 ('0', '13', None),
 ...
 ('48*', '98*', None)]
 sage: G5 = E.isogeny_ell_graph(5, directed=False, label_by_j=True); G5
 Graph on 9 vertices
 sage: G5.vertices(sort=True)
 ['13', '13*', '407', '407*', '52', '62', '62*', '98', '98*']
 sage: G5.edges(sort=True)
 [('13', '52', None),
  ('13', '98', None),
  ...
  ('62*', '98*', None)]
```

Supersingular curve over finite extension field of degree 2:

```
sage: # needs sage.graphs sage.rings.finite_rings
sage: K = GF(431^2, "i", x^2 + 1)
sage: E = EllipticCurve(K, j=0)
sage: E.is_supersingular()
True
```

(continues on next page)

(continued from previous page)

```
sage: G = E.isogeny_ell_graph(2, directed=True, label_by_j=True); G
Looped multi-digraph on 37 vertices
sage: G.vertices(sort=True)
['0',
 '102',
 ...
 '87*i + 190']
sage: G.edges(sort=True)
[('0', '125', None),
 ('0', '125', None),
 ...
 '81*i + 65', None)]
sage: H = E.isogeny_ell_graph(2, directed=False, label_by_j=True); H
Looped multi-graph on 37 vertices
sage: H.vertices(sort=True)
['0',
 '102',
 ...
 '87*i + 190']
sage: H.edges(sort=True)
[('0', '125', None),
 ('102', '125', None),
 ...
 ('81*i + 65', '87*i + 190', None)]
```

Curve over a quadratic number field:

```
sage: # needs sage.graphs sage.rings.finite_rings sage.rings.number_field
sage: K.<e> = NumberField(x^2 - 2)
sage: E = EllipticCurve(K, [1, 0, 1, 4, -6])
sage: G2 = E.isogeny_ell_graph(2, directed=False)
sage: G2.vertices(sort=True)
['y^2 + x*y + y = x^3 + (-130*e-356)*x + (-2000*e-2038)',
 'y^2 + x*y + y = x^3 + (-36)*x + (-70)',
 'y^2 + x*y + y = x^3 + (130*e-356)*x + (2000*e-2038)',
 'y^2 + x*y + y = x^3 + 4*x + (-6)']
sage: G2.edges(sort=True)
[('y^2 + x*y + y = x^3 + (-130*e-356)*x + (-2000*e-2038)',
  'y^2 + x*y + y = x^3 + (-36)*x + (-70)', None),
 ('y^2 + x*y + y = x^3 + (-36)*x + (-70)',
  'y^2 + x*y + y = x^3 + (130*e-356)*x + (2000*e-2038)', None),
 ('y^2 + x*y + y = x^3 + (-36)*x + (-70)',
  'y^2 + x*y + y = x^3 + 4*x + (-6)', None)]
sage: G3 = E.isogeny_ell_graph(3, directed=False)
sage: G3.vertices(sort=True)
['y^2 + x*y + y = x^3 + (-1)*x',
 'y^2 + x*y + y = x^3 + (-171)*x + (-874)',
 'y^2 + x*y + y = x^3 + 4*x + (-6)']
sage: G3.edges(sort=True)
[('y^2 + x*y + y = x^3 + (-1)*x',
  'y^2 + x*y + y = x^3 + 4*x + (-6)', None),
 ('y^2 + x*y + y = x^3 + (-171)*x + (-874)',
  'y^2 + x*y + y = x^3 + 4*x + (-6)', None)]
```

kernel_polynomial_from_divisor(*f*, *l*, *check*)

Given an irreducible divisor f of the l-division polynomial on this curve, return the kernel polynomial defining the subgroup defined by f.

If the given polynomial does not define a rational subgroup, a `ValueError` is raised.

This method is currently only implemented for prime l.

EXAMPLES:

```
sage: E = EllipticCurve(GF(101^2), [0,1])
sage: f,_ = E.division_polynomial(5).factor()[0]
sage: ker = E.kernel_polynomial_from_divisor(f, 5); ker
x^2 + (49*z2 + 10)*x + 30*z2 + 80
sage: E.isogeny(ker)
Isogeny of degree 5
 from Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field in z2 of size␣
↪101^2
 to Elliptic Curve defined by y^2 = x^3 + (6*z2+16)*x + 18 over Finite Field␣
↪in z2 of size 101^2
```

The method detects invalid inputs:

```
sage: E = EllipticCurve(GF(101), [0,1])
sage: f,_ = E.division_polynomial(5).factor()[-1]
sage: E.kernel_polynomial_from_divisor(f, 5)
Traceback (most recent call last):
...
ValueError: given polynomial does not define a rational 5-isogeny
```

```
sage: E = EllipticCurve(GF(101), [1,1])
sage: f,_ = E.division_polynomial(7).factor()[-1]
sage: E.kernel_polynomial_from_divisor(f, 7)
Traceback (most recent call last):
...
ValueError: given polynomial does not define a rational 7-isogeny
```

```
sage: x = polygen(QQ)
sage: K.<t> = NumberField(x^12 - 2*x^10 + 3*x^8 + 228/13*x^6 + 235/13*x^4 +␣
↪22/13*x^2 + 1/13)
sage: E = EllipticCurve(K, [1,0])
sage: ker = E.kernel_polynomial_from_divisor(x - t, 13); ker
x^6 + (-169/64*t^10 + 169/32*t^8 - 247/32*t^6 - 377/8*t^4 - 2977/64*t^2 - 105/
↪32)*x^4 + (-169/32*t^10 + 169/16*t^8 - 247/16*t^6 - 377/4*t^4 - 2977/32*t^2␣
↪- 89/16)*x^2 - 13/64*t^10 + 13/32*t^8 - 19/32*t^6 - 29/8*t^4 - 229/64*t^2 -␣
↪13/32
sage: phi = E.isogeny(ker, check=True); phi
Isogeny of degree 13
 from Elliptic Curve defined by y^2 = x^3 + x
  over Number Field in t with defining polynomial x^12 - 2*x^10 + 3*x^8 + 228/
↪13*x^6 + 235/13*x^4 + 22/13*x^2 + 1/13
 to Elliptic Curve defined by y^2 = x^3 + (-2535/16*t^10+2535/8*t^8-3705/8*t^
↪6-5655/2*t^4-44655/16*t^2-2047/8)*x
  over Number Field in t with defining polynomial x^12 - 2*x^10 + 3*x^8 + 228/
↪13*x^6 + 235/13*x^4 + 22/13*x^2 + 1/13
```

ALGORITHM: [EPSV2023], Algorithm 3 (`KernelPolynomialFromDivisor`).

kernel_polynomial_from_point(*P*, *algorithm*)

Given a point P on this curve which generates a rational subgroup, return the kernel polynomial of that subgroup as a polynomial over the base field of the curve. (The point P itself may be defined over an extension.)

EXAMPLES:

```
sage: E = EllipticCurve(GF(101), [1,1])
sage: F = GF(101^3)
sage: EE = E.change_ring(F)
sage: xK = F([77, 28, 8]); xK
8*z3^2 + 28*z3 + 77
sage: K = EE.lift_x(xK); K.order()
43
sage: E.kernel_polynomial_from_point(K)
x^21 + 7*x^20 + 22*x^19 + 4*x^18 + 7*x^17 + 81*x^16 + 41*x^15 + 68*x^14 +␣
↪18*x^13 + 58*x^12 + 31*x^11 + 26*x^10 + 62*x^9 + 20*x^8 + 73*x^7 + 23*x^6 +␣
↪66*x^5 + 79*x^4 + 12*x^3 + 40*x^2 + 50*x + 93
```

The "minpoly" algorithm is often much faster than the "basic" algorithm:

```
sage: from sage.schemes.elliptic_curves.ell_field import EllipticCurve_field,␣
↪point_of_order
sage: p = 2^127 - 1
sage: E = EllipticCurve(GF(p), [1,0])
sage: P = point_of_order(E, 31)
sage: %timeit E.kernel_polynomial_from_point(P, algorithm='basic')    # not␣
↪tested
4.38 ms ± 13.7 µs per loop (mean ± std. dev. of 7 runs, 100 loops each)
sage: %timeit E.kernel_polynomial_from_point(P, algorithm='minpoly')  # not␣
↪tested
854 µs ± 1.56 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Example of finding all the rational isogenies using this method:

```
sage: E = EllipticCurve(GF(71), [1,2,3,4,5])
sage: F = E.division_field(11)
sage: EE = E.change_ring(F)
sage: fs = set()
sage: for K in EE(0).division_points(11):
....:     if not K:
....:         continue
....:     Kp = EE.frobenius_isogeny()(K)
....:     if Kp.weil_pairing(K, 11) == 1:
....:         fs.add(E.kernel_polynomial_from_point(K))
sage: fs = sorted(fs); fs
[x^5 + 10*x^4 + 18*x^3 + 10*x^2 + 43*x + 46,
 x^5 + 65*x^4 + 39*x^2 + 20*x + 63]
sage: from sage.schemes.elliptic_curves.isogeny_small_degree import is_kernel_
↪polynomial
sage: {is_kernel_polynomial(E, 11, f) for f in fs}
{True}
sage: isogs = [E.isogeny(f) for f in fs]
sage: isogs[0]
Isogeny of degree 11 from Elliptic Curve defined by y^2 + x*y + 3*y = x^3 +␣
↪2*x^2 + 4*x + 5 over Finite Field of size 71 to Elliptic Curve defined by y^
↪2 + x*y + 3*y = x^3 + 2*x^2 + 34*x + 42 over Finite Field of size 71
sage: isogs[1]
Isogeny of degree 11 from Elliptic Curve defined by y^2 + x*y + 3*y = x^3 +␣
↪2*x^2 + 4*x + 5 over Finite Field of size 71 to Elliptic Curve defined by y^
↪2 + x*y + 3*y = x^3 + 2*x^2 + 12*x + 40 over Finite Field of size 71
sage: set(isogs) == set(E.isogenies_prime_degree(11))
True
```

ALGORITHM:

- The "basic" algorithm is to multiply together all the linear factors $(X - x([i]P))$ of the kernel polynomial using a product tree, then converting the result to the base field of the curve. Its complexity is $\widetilde{O}(\ell k)$ where k is the extension degree.
- The "minpoly" algorithm is [EPSV2023], Algorithm 4 (KernelPolynomialFromIrrationalX). Over finite fields, its complexity is $O(\ell k) + \widetilde{O}(\ell)$ where k is the extension degree.

quadratic_twist(*D=None*)

Return the quadratic twist of this curve by D.

INPUT:

- D (default None) the twisting parameter (see below).

In characteristics other than 2, D must be nonzero, and the twist is isomorphic to self after adjoining $\sqrt{(}D)$ to the base.

In characteristic 2, D is arbitrary, and the twist is isomorphic to self after adjoining a root of $x^2 + x + D$ to the base.

In characteristic 2 when $j = 0$, this is not implemented.

If the base field F is finite, D need not be specified, and the curve returned is the unique curve (up to isomorphism) defined over F isomorphic to the original curve over the quadratic extension of F but not over F itself. Over infinite fields, an error is raised if D is not given.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: E = EllipticCurve([GF(1103)(1), 0, 0, 107, 340]); E
Elliptic Curve defined by y^2 + x*y  = x^3 + 107*x + 340
 over Finite Field of size 1103
sage: F = E.quadratic_twist(-1); F
Elliptic Curve defined by y^2  = x^3 + 1102*x^2 + 609*x + 300
 over Finite Field of size 1103
sage: E.is_isomorphic(F)
False
sage: E.is_isomorphic(F, GF(1103^2,'a'))
True
```

A characteristic 2 example:

```
sage: E = EllipticCurve(GF(2), [1,0,1,1,1])
sage: E1 = E.quadratic_twist(1)
sage: E.is_isomorphic(E1)
False
sage: E.is_isomorphic(E1, GF(4,'a'))
True
```

Over finite fields, the twisting parameter may be omitted:

```
sage: # needs sage.rings.finite_rings
sage: k.<a> = GF(2^10)
sage: E = EllipticCurve(k, [a^2,a,1,a+1,1])
sage: Et = E.quadratic_twist()
sage: Et  # random (only determined up to isomorphism)
Elliptic Curve defined
 by y^2 + x*y  = x^3 + (a^7+a^4+a^3+a^2+a+1)*x^2 + (a^8+a^6+a^4+1)
 over Finite Field in a of size 2^10
sage: E.is_isomorphic(Et)
```

(continues on next page)

(continued from previous page)

```
False
sage: E.j_invariant() == Et.j_invariant()
True

sage: # needs sage.rings.finite_rings
sage: p = next_prime(10^10)
sage: k = GF(p)
sage: E = EllipticCurve(k, [1,2,3,4,5])
sage: Et = E.quadratic_twist()
sage: Et  # random (only determined up to isomorphism)
Elliptic Curve defined
 by y^2  = x^3 + 7860088097*x^2 + 9495240877*x + 3048660957
 over Finite Field of size 10000000019
sage: E.is_isomorphic(Et)
False
sage: k2 = GF(p^2,'a')
sage: E.change_ring(k2).is_isomorphic(Et.change_ring(k2))
True
```

quartic_twist(D)

Return the quartic twist of this curve by D.

INPUT:

- D (must be nonzero) – the twisting parameter

Note: The characteristic must not be 2 or 3, and the j-invariant must be 1728.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: E = EllipticCurve_from_j(GF(13)(1728)); E
Elliptic Curve defined by y^2 = x^3 + x over Finite Field of size 13
sage: E1 = E.quartic_twist(2); E1
Elliptic Curve defined by y^2 = x^3 + 5*x over Finite Field of size 13
sage: E.is_isomorphic(E1)
False
sage: E.is_isomorphic(E1, GF(13^2,'a'))
False
sage: E.is_isomorphic(E1, GF(13^4,'a'))
True
```

sextic_twist(D)

Return the sextic twist of this curve by D.

INPUT:

- D (must be nonzero) – the twisting parameter

Note: The characteristic must not be 2 or 3, and the j-invariant must be 0.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: E = EllipticCurve_from_j(GF(13)(0)); E
Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field of size 13
sage: E1 = E.sextic_twist(2); E1
Elliptic Curve defined by y^2 = x^3 + 11 over Finite Field of size 13
sage: E.is_isomorphic(E1)
False
sage: E.is_isomorphic(E1, GF(13^2,'a'))
False
sage: E.is_isomorphic(E1, GF(13^4,'a'))
False
sage: E.is_isomorphic(E1, GF(13^6,'a'))
True
```

two_torsion_rank()

Return the dimension of the 2-torsion subgroup of $E(K)$.

This will be 0, 1 or 2.

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: E.two_torsion_rank()
0
sage: K.<alpha> = QQ.extension(E.division_polynomial(2).monic())            #␣
↪needs sage.rings.number_field
sage: E.base_extend(K).two_torsion_rank()                                   #␣
↪needs sage.rings.number_field
1
sage: E.reduction(53).two_torsion_rank()
2
```

```
sage: E = EllipticCurve('14a1')
sage: E.two_torsion_rank()
1
sage: f = E.division_polynomial(2).monic().factor()[1][0]
sage: K.<alpha> = QQ.extension(f)                                           #␣
↪needs sage.rings.number_field
sage: E.base_extend(K).two_torsion_rank()                                   #␣
↪needs sage.rings.number_field
2
```

```
sage: EllipticCurve('15a1').two_torsion_rank()
2
```

weierstrass_p(*prec=20*, *algorithm=None*)

Compute the Weierstrass $\wp$-function of this elliptic curve.

ALGORITHM: *sage.schemes.elliptic_curves.ell_wp.weierstrass_p()*

INPUT:

- `prec` – precision
- `algorithm` – string or `None` (default: `None`): a choice of algorithm among `"pari"`, `"fast"`, `"quadratic"`; or `None` to let this function determine the best algorithm to use.

OUTPUT:

A Laurent series in one variable z with coefficients in the base field k of E.

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: E.weierstrass_p(prec=10)
z^-2 + 31/15*z^2 + 2501/756*z^4 + 961/675*z^6 + 77531/41580*z^8 + O(z^10)
sage: E.weierstrass_p(prec=8)
z^-2 + 31/15*z^2 + 2501/756*z^4 + 961/675*z^6 + O(z^8)
sage: Esh = E.short_weierstrass_model()
sage: Esh.weierstrass_p(prec=8)
z^-2 + 13392/5*z^2 + 1080432/7*z^4 + 59781888/25*z^6 + O(z^8)
sage: E.weierstrass_p(prec=20, algorithm='fast')
z^-2 + 31/15*z^2 + 2501/756*z^4 + 961/675*z^6 + 77531/41580*z^8
+ 1202285717/928746000*z^10 + 2403461/2806650*z^12 + 30211462703/
↪43418875500*z^14
+ 3539374016033/7723451736000*z^16 + 413306031683977/1289540602350000*z^18 +␣
↪O(z^20)
sage: E.weierstrass_p(prec=20, algorithm='pari')
z^-2 + 31/15*z^2 + 2501/756*z^4 + 961/675*z^6 + 77531/41580*z^8
+ 1202285717/928746000*z^10 + 2403461/2806650*z^12 + 30211462703/
↪43418875500*z^14
+ 3539374016033/7723451736000*z^16 + 413306031683977/1289540602350000*z^18 +␣
↪O(z^20)
sage: E.weierstrass_p(prec=20, algorithm='quadratic')
z^-2 + 31/15*z^2 + 2501/756*z^4 + 961/675*z^6 + 77531/41580*z^8
+ 1202285717/928746000*z^10 + 2403461/2806650*z^12 + 30211462703/
↪43418875500*z^14
+ 3539374016033/7723451736000*z^16 + 413306031683977/1289540602350000*z^18 +␣
↪O(z^20)
```

sage.schemes.elliptic_curves.ell_field.**compute_model**(*E*, *name*)

Return a model of an elliptic curve `E` of the type specified in the `name` parameter.

Used as a helper function in *EllipticCurveIsogeny*.

INPUT:

- `E` (elliptic curve)
- `name` (string) – current options:
  - `"minimal"`: Return a global minimal model of `E` if it exists, and a semi-global minimal model otherwise. For this choice, `E` must be defined over a number field. See *global_minimal_model()*.
  - `"short_weierstrass"`: Return a short Weierstrass model of `E` assuming one exists. See *short_weierstrass_model()*.
  - `"montgomery"`: Return an (untwisted) Montgomery model of `E` assuming one exists over this field. See *montgomery_model()*.

OUTPUT:

An elliptic curve of the specified type isomorphic to E.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_field import compute_model
sage: E = EllipticCurve([12/7, 405/49, 0, -81/8, 135/64])
sage: compute_model(E, 'minimal')
Elliptic Curve defined by y^2 = x^3 - x^2 - 7*x + 10 over Rational Field
```

(continues on next page)

(continued from previous page)

```
sage: compute_model(E, 'short_weierstrass')
Elliptic Curve defined by y^2 = x^3 - 48114*x + 4035015 over Rational Field
sage: compute_model(E, 'montgomery')
Elliptic Curve defined by y^2 = x^3 + 5*x^2 + x over Rational Field
```

sage.schemes.elliptic_curves.ell_field.**point_of_order**(E, n)

Given an elliptic curve E over a finite field or a number field and an integer $n \geq 1$, construct a point of order n on E, possibly defined over an extension of the base field of E.

Currently only prime powers n are supported.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_field import point_of_order
sage: E = EllipticCurve(GF(101), [1,2,3,4,5])
sage: P = point_of_order(E, 5); P  # random
(50*Y^5 + 48*Y^4 + 26*Y^3 + 37*Y^2 + 48*Y + 15 : 25*Y^5 + 31*Y^4 + 79*Y^3 + 39*Y^
↪2 + 3*Y + 20 : 1)
sage: P.base_ring()
Finite Field in Y of size 101^6
sage: P.order()
5
sage: P.curve().a_invariants()
(1, 2, 3, 4, 5)
```

```
sage: Q = point_of_order(E, 8); Q  # random
(69*x^5 + 24*x^4 + 100*x^3 + 65*x^2 + 88*x + 97 : 65*x^5 + 28*x^4 + 5*x^3 + 45*x^
↪2 + 42*x + 18 : 1)
sage: 8*Q == 0 and 4*Q != 0
True
```

```
sage: from sage.schemes.elliptic_curves.ell_field import point_of_order
sage: E = EllipticCurve(QQ, [7,7])
sage: P = point_of_order(E, 3); P  # random
(x : -Y : 1)
sage: P.base_ring()
Number Field in Y with defining polynomial Y^2 - x^3 - 7*x - 7 over its base field
sage: P.base_ring().base_field()
Number Field in x with defining polynomial x^4 + 14*x^2 + 28*x - 49/3
sage: P.order()
3
sage: P.curve().a_invariants()
(0, 0, 0, 7, 7)
```

```
sage: Q = point_of_order(E, 4); Q  # random
(x : Y : 1)
sage: Q.base_ring()
Number Field in Y with defining polynomial Y^2 - x^3 - 7*x - 7 over its base field
sage: Q.base_ring().base_field()
Number Field in x with defining polynomial x^6 + 35*x^4 + 140*x^3 - 245*x^2 -␣
↪196*x - 735
sage: Q.order()
4
```

CHAPTER

SIX

ELLIPTIC CURVES OVER FINITE FIELDS

AUTHORS:

- William Stein (2005): Initial version
- Robert Bradshaw et al….
- John Cremona (2008-02): Point counting and group structure for non-prime fields, Frobenius endomorphism and order, elliptic logs
- Mariah Lenox (2011-03): Added `set_order` method
- Lorenz Panny, John Cremona (2023-02): `.twists()`
- Lorenz Panny (2023): `special_supersingular_curve()`

class sage.schemes.elliptic_curves.ell_finite_field.**EllipticCurve_finite_field**(*R*, *data*, *category=None*)

Bases: *EllipticCurve_field*

Elliptic curve over a finite field.

EXAMPLES:

```
sage: EllipticCurve(GF(101),[2,3])
Elliptic Curve defined by y^2  = x^3 + 2*x + 3 over Finite Field of size 101

sage: # needs sage.rings.finite_rings
sage: F = GF(101^2, 'a')
sage: EllipticCurve([F(2),F(3)])
Elliptic Curve defined by y^2  = x^3 + 2*x + 3 over Finite Field in a of size 101^
↪2
```

Elliptic curves over $\mathbf{Z}/N\mathbf{Z}$ with N prime are of type "elliptic curve over a finite field":

```
sage: F = Zmod(101)
sage: EllipticCurve(F, [2, 3])
Elliptic Curve defined by y^2 = x^3 + 2*x + 3 over Ring of integers modulo 101
sage: E = EllipticCurve([F(2), F(3)])
sage: type(E)
<class 'sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field_
↪with_category'>
sage: E.category()
Category of abelian varieties over Ring of integers modulo 101
```

Elliptic curves over $\mathbf{Z}/N\mathbf{Z}$ with N composite are of type "generic elliptic curve":

```
sage: F = Zmod(95)
sage: EllipticCurve(F, [2, 3])
Elliptic Curve defined by y^2 = x^3 + 2*x + 3 over Ring of integers modulo 95
sage: E = EllipticCurve([F(2), F(3)])
sage: type(E)
<class 'sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic_with_
↪category'>
sage: E.category()
Category of schemes over Ring of integers modulo 95
sage: TestSuite(E).run(skip=["_test_elements"])
```

abelian_group()

Return the abelian group structure of the group of points on this elliptic curve.

See also:

If you do not need the complete abelian group structure but only generators of the group, use *gens()* which can be much faster in some cases.

This method relies on *gens()*, which uses random points on the curve and hence the generators are likely to differ from one run to another. However, the group is cached, so the generators will not change in any one run of Sage.

OUTPUT:

- an AdditiveAbelianGroupWrapper object encapsulating the abelian group of rational points on this elliptic curve

ALGORITHM:

We first call *gens()* to obtain a generating set (P, Q). Letting P denote the point of larger order n_1, we extend P to a basis (P, Q') by computing a scalar x such that $Q' = Q - [x]P$ has order $n_2 = \#E/n_1$. Finding x involves a (typically easy) discrete-logarithm computation.

The complexity of the algorithm is the cost of factoring the group order, plus $\Theta(\sqrt{\ell})$ for each prime ℓ such that the rational ℓ^∞-torsion of `self` is isomorphic to $\mathbf{Z}/\ell^r \times \mathbf{Z}/\ell^s$ with $r > s > 0$, times a polynomial in the logarithm of the base-field size.

AUTHORS:

- John Cremona: original implementation
- Lorenz Panny (2021): current implementation

See also:

AdditiveAbelianGroupWrapper.from_generators()

EXAMPLES:

```
sage: E = EllipticCurve(GF(11),[2,5])
sage: E.abelian_group()
Additive abelian group isomorphic to Z/10 embedded in
 Abelian group of points on Elliptic Curve defined by y^2 = x^3 + 2*x + 5
  over Finite Field of size 11
```

```
sage: E = EllipticCurve(GF(41),[2,5])
sage: E.abelian_group()
Additive abelian group isomorphic to Z/22 + Z/2 ...
```

```
sage: # needs sage.rings.finite_rings
sage: F.<a> = GF(3^6,'a')
sage: E = EllipticCurve([a^4 + a^3 + 2*a^2 + 2*a, 2*a^5 + 2*a^3 + 2*a^2 + 1])
sage: E.abelian_group()
Additive abelian group isomorphic to Z/26 + Z/26 ...
```

```
sage: # needs sage.rings.finite_rings
sage: F.<a> = GF(101^3,'a')
sage: E = EllipticCurve([2*a^2 + 48*a + 27, 89*a^2 + 76*a + 24])
sage: E.abelian_group()
Additive abelian group isomorphic to Z/1031352 ...
```

The group can be trivial:

```
sage: E = EllipticCurve(GF(2), [0,0,1,1,1])
sage: E.abelian_group()
Trivial group embedded in Abelian group of points on
 Elliptic Curve defined by y^2 + y = x^3 + x + 1 over Finite Field of size 2
```

Of course, there are plenty of points if we extend the field:

```
sage: E.cardinality(extension_degree=100)
1267650600228231653296516890625
```

This tests the patch for Issue #3111, using 10 primes randomly selected:

```
sage: E = EllipticCurve('389a')
sage: for p in [5927, 2297, 1571, 1709, 3851, 127, 3253, 5783, 3499, 4817]:
....:     G = E.change_ring(GF(p)).abelian_group()
sage: for p in prime_range(10000):  # long time (19s on sage.math, 2011)
....:     if p != 389:
....:         G = E.change_ring(GF(p)).abelian_group()
```

This tests that the bug reported in Issue #3926 has been fixed:

```
sage: # needs sage.rings.number_field
sage: K.<i> = QuadraticField(-1)
sage: OK = K.ring_of_integers()
sage: P = K.factor(10007)[0][0]
sage: OKmodP = OK.residue_field(P)
sage: E = EllipticCurve([0, 0, 0, i, i + 3])
sage: Emod = E.change_ring(OKmodP); Emod
Elliptic Curve defined by y^2 = x^3 + ibar*x + (ibar+3)
 over Residue field in ibar of Fractional ideal (10007)
sage: Emod.abelian_group() #random generators
(Multiplicative Abelian group isomorphic to C50067594 x C2,
 ((3152*ibar + 7679 : 7330*ibar + 7913 : 1), (8466*ibar + 1770 : 0 : 1)))
```

cardinality(*algorithm=None*, *extension_degree=1*)

Return the number of points on this elliptic curve.

INPUT:

- `algorithm` – (optional) string:
  - `'pari'` – use the PARI C-library function `ellcard`.
  - **`'bsgs'` – use the baby-step giant-step method as** implemented in Sage, with the Cremona-Sutherland version of Mestre's trick.

  - `'exhaustive'` – naive point counting.
  - `'subfield'` – reduce to a smaller field, provided that the j-invariant lies in a subfield.
  - `'all'` – compute cardinality with both `'pari'` and `'bsgs'`; return result if they agree or raise a `AssertionError` if they do not

- `extension_degree` – an integer d (default: 1): if the base field is $\mathbf{F}_q$, return the cardinality of `self` over the extension $\mathbf{F}_{q^d}$ of degree d.

OUTPUT:

The order of the group of rational points of `self` over its base field, or over an extension field of degree d as above. The result is cached.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: EllipticCurve(GF(4, 'a'), [1,2,3,4,5]).cardinality()
8
sage: k.<a> = GF(3^3)
sage: l = [a^2 + 1, 2*a^2 + 2*a + 1, a^2 + a + 1, 2, 2*a]
sage: EllipticCurve(k,l).cardinality()
29
```

```
sage: # needs sage.rings.finite_rings
sage: l = [1, 1, 0, 2, 0]
sage: EllipticCurve(k, l).cardinality()
38
```

An even bigger extension (which we check against Magma):

```
sage: # needs sage.rings.finite_rings
sage: EllipticCurve(GF(3^100, 'a'), [1,2,3,4,5]).cardinality()
515377520732011331036459693969645888996929981504
sage: magma.eval("Order(EllipticCurve([GF(3^100)|1,2,3,4,5]))")     # optional␣
↪- magma
'515377520732011331036459693969645888996929981504'
```

```
sage: EllipticCurve(GF(10007), [1,2,3,4,5]).cardinality()
10076
sage: EllipticCurve(GF(10007), [1,2,3,4,5]).cardinality(algorithm='pari')
10076
sage: EllipticCurve(GF(next_prime(10**20)), [1,2,3,4,5]).cardinality()
100000000011093199520
```

The cardinality is cached:

```
sage: # needs sage.rings.finite_rings
sage: E = EllipticCurve(GF(3^100, 'a'), [1,2,3,4,5])
sage: E.cardinality() is E.cardinality()
True
```

The following is very fast since the curve is actually defined over the prime field:

```
sage: # needs sage.rings.finite_rings
sage: k.<a> = GF(11^100)
sage: E1 = EllipticCurve(k, [3,3])
sage: N1 = E1.cardinality(algorithm="subfield"); N1
```

(continues on next page)

(continued from previous page)

```
13780612339822270184118337172089636776264331200038467184683526694179151034106556517649784650
sage: E1.cardinality_pari() == N1
True
sage: E2 = E1.quadratic_twist()
sage: N2 = E2.cardinality(algorithm="subfield"); N2
13780612339822270184118337172089636776264331200038465681609428410130819384998058836230447249
sage: E2.cardinality_pari() == N2
True
sage: N1 + N2 == 2*(k.cardinality() + 1)
True
```

We can count points over curves defined as a reduction:

```
sage: # needs sage.rings.number_field
sage: x = polygen(QQ)
sage: K.<w> = NumberField(x^2 + x + 1)
sage: EK = EllipticCurve(K, [0, 0, w, 2, 1])
sage: E = EK.base_extend(K.residue_field(2))
sage: E
Elliptic Curve defined by y^2 + wbar*y = x^3 + 1
 over Residue field in wbar of Fractional ideal (2)
sage: E.cardinality()
7
sage: E = EK.base_extend(K.residue_field(w - 1))
sage: E.abelian_group()
Trivial group embedded in Abelian group of points on Elliptic Curve defined
 by y^2 + y = x^3 + 2*x + 1 over Residue field of Fractional ideal (w - 1)
```

```
sage: R.<x> = GF(17)[]
sage: pol = R.irreducible_element(5)
sage: k.<a> = R.residue_field(pol)
sage: E = EllipticCurve(R, [1, x]).base_extend(k)
sage: E
Elliptic Curve defined by y^2 = x^3 + x + a
 over Residue field in a of Principal ideal (x^5 + x + 14)
  of Univariate Polynomial Ring in x over Finite Field of size 17
sage: E.cardinality()
1421004
```

cardinality_bsgs(*verbose=False*)

Return the cardinality of `self` over the base field.

ALGORITHM: A variant of "Mestre's trick" extended to all finite fields by Cremona and Sutherland, 2008.

Note:

1. The Mestre-Schoof-Cremona-Sutherland algorithm may fail for a small finite number of curves over F_q for q at most 49, so for $q < 50$ we use an exhaustive count.
2. Quadratic twists are not implemented in characteristic 2 when $j = 0 (= 1728)$; but this case is treated separately.

EXAMPLES:

```
sage: p = next_prime(10^3)
sage: E = EllipticCurve(GF(p), [3,4])
sage: E.cardinality_bsgs()
1020
sage: E = EllipticCurve(GF(3^4,'a'), [1,1])
sage: E.cardinality_bsgs()
64
sage: F.<a> = GF(101^3,'a')
sage: E = EllipticCurve([2*a^2 + 48*a + 27, 89*a^2 + 76*a + 24])
sage: E.cardinality_bsgs()
1031352
```

cardinality_exhaustive()

Return the cardinality of `self` over the base field.

This simply adds up the number of points with each x-coordinate: only used for small field sizes!

EXAMPLES:

```
sage: p = next_prime(10^3)
sage: E = EllipticCurve(GF(p), [3,4])
sage: E.cardinality_exhaustive()
1020
sage: E = EllipticCurve(GF(3^4,'a'), [1,1])
sage: E.cardinality_exhaustive()
64
```

cardinality_pari()

Return the cardinality of `self` using PARI.

This uses pari:ellcard.

EXAMPLES:

```
sage: p = next_prime(10^3)
sage: E = EllipticCurve(GF(p),[3,4])
sage: E.cardinality_pari()
1020
sage: K = GF(next_prime(10^6))
sage: E = EllipticCurve(K,[1,0,0,1,1])
sage: E.cardinality_pari()
999945
```

Since Issue #16931, this now works over finite fields which are not prime fields:

```
sage: # needs sage.rings.finite_rings
sage: k.<a> = GF(7^3)
sage: E = EllipticCurve_from_j(a)
sage: E.cardinality_pari()
318
sage: K.<a> = GF(3^20)
sage: E = EllipticCurve(K,[1,0,0,1,a])
sage: E.cardinality_pari()
3486794310
```

count_points(*n=1*)

Return the cardinality of this elliptic curve over the base field or extensions.

INPUT:

- n (int) – a positive integer

OUTPUT:

If $n = 1$, returns the cardinality of the curve over its base field.

If $n > 1$, returns a list $[c_1, c_2, ..., c_n]$ where c_d is the cardinality of the curve over the extension of degree d of its base field.

EXAMPLES:

```
sage: p = 101
sage: F = GF(p)
sage: E = EllipticCurve(F, [2,3])
sage: E.count_points(1)
96
sage: E.count_points(5)
[96, 10368, 1031904, 104053248, 10509895776]
```

```
sage: # needs sage.rings.finite_rings
sage: F.<a> = GF(p^2)
sage: E = EllipticCurve(F, [a,a])
sage: E.cardinality()
10295
sage: E.count_points()
10295
sage: E.count_points(1)
10295
sage: E.count_points(5)
[10295, 104072155, 1061518108880, 10828567126268595, 1104622125554391923 75]
```

endomorphism_discriminant_from_class_number(h)

Return the endomorphism order discriminant of this ordinary elliptic curve, given its class number h.

INPUT:

- h – a positive integer

OUTPUT:

(integer) The discriminant of the endomorphism ring End(E), if this has class number h. If End(E) does not have class number h, a ValueError is raised.

ALGORITHM:

Compute the trace of Frobenius and hence the discriminant D_0 and class number h_0 of the maximal order containing the endomorphism order. From the given value of h, which must be a multiple of h_0, compute the possible conductors, using *height_above_floor()* for each prime ℓ dividing the quotient h/h_0. If exactly one conductor f remains, return $f^2 D_0$, otherwise raise a ValueError; this can onlyhappen when the input value of h was incorrect.

Note: Adapted from [RouSuthZur2022]. The application for which one knows the class number in advance is in the recognition of Hilbert Class Polynomials: see *sage.schemes.elliptic_curves.cm.is_HCP()*.

EXAMPLES:

```
sage: F = GF(312401)
sage: E = EllipticCurve(F,(0, 0, 0, 309381, 93465))
sage: E.endomorphism_discriminant_from_class_number(30)
-671
```

We check that this is the correct discriminant, and the input value of h was correct:

```
sage: H = hilbert_class_polynomial(-671)
sage: H(E.j_invariant()) == 0 and H.degree()==30
True
```

frobenius()

Return the frobenius of `self` as an element of a quadratic order.

Note: This computes the curve cardinality, which may be time-consuming.

Frobenius is only determined up to conjugacy.

EXAMPLES:

```
sage: E = EllipticCurve(GF(11),[3,3])
sage: E.frobenius()
phi
sage: E.frobenius().minpoly()
x^2 - 4*x + 11
```

For some supersingular curves, Frobenius is in Z:

```
sage: # needs sage.rings.finite_rings
sage: E = EllipticCurve(GF(25,'a'),[0,0,0,0,1])
sage: E.frobenius()
-5
```

frobenius_discriminant()

Return the discriminant of the ring $\mathbf{Z}[\pi_E]$ where π_E is the Frobenius endomorphism.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: F.<t> = GF(11^4)
sage: E = EllipticCurve([t,t])
sage: E.frobenius_discriminant()
-57339
```

frobenius_endomorphism()

Return the q-power Frobenius endomorphism of this elliptic curve, where q is the cardinality of the (finite) base field.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: F.<t> = GF(11^4)
sage: E = EllipticCurve([t,t])
sage: E.frobenius_endomorphism()
Frobenius endomorphism of degree 14641 = 11^4:
  From: Elliptic Curve defined by y^2 = x^3 + t*x + t over Finite Field in t␣
```

(continues on next page)

(continued from previous page)

```
↪of size 11^4
  To:   Elliptic Curve defined by y^2 = x^3 + t*x + t over Finite Field in t␣
↪of size 11^4
sage: E.frobenius_endomorphism() == E.frobenius_isogeny(4)
True
```

See also:

frobenius_isogeny()

frobenius_order()

Return the quadratic order Z[phi] where phi is the Frobenius endomorphism of the elliptic curve.

Note: This computes the curve cardinality, which may be time-consuming.

EXAMPLES:

```
sage: E = EllipticCurve(GF(11),[3,3])
sage: E.frobenius_order()
Order of conductor 2 generated by phi
 in Number Field in phi with defining polynomial x^2 - 4*x + 11
```

For some supersingular curves, Frobenius is in Z and the Frobenius order is Z:

```
sage: # needs sage.rings.finite_rings
sage: E = EllipticCurve(GF(25,'a'),[0,0,0,0,1])
sage: R = E.frobenius_order()
sage: R
Order generated by []
 in Number Field in phi with defining polynomial x + 5
sage: R.degree()
1
```

frobenius_polynomial()

Return the characteristic polynomial of Frobenius.

The Frobenius endomorphism of the elliptic curve has quadratic characteristic polynomial. In most cases this is irreducible and defines an imaginary quadratic order; for some supersingular curves, Frobenius is an integer a and the polynomial is $(x - a)^2$.

Note: This computes the curve cardinality, which may be time-consuming.

EXAMPLES:

```
sage: E = EllipticCurve(GF(11),[3,3])
sage: E.frobenius_polynomial()
x^2 - 4*x + 11
```

For some supersingular curves, Frobenius is in Z and the polynomial is a square:

```
sage: # needs sage.rings.finite_rings
sage: E = EllipticCurve(GF(25,'a'),[0,0,0,0,1])
sage: E.frobenius_polynomial().factor()
(x + 5)^2
```

gens()

Return points which generate the abelian group of points on this elliptic curve.

The algorithm involves factoring the group order of `self`, but is otherwise (randomized) polynomial-time.

(The points returned by this function are not guaranteed to be the same each time, although they should remain fixed within a single run of Sage unless *abelian_group()* is called.)

OUTPUT: a tuple of points on the curve.

- if the group is trivial: an empty tuple.
- if the group is cyclic: a tuple with 1 point, a generator.
- if the group is not cyclic: a tuple with 2 points, where the order of the first point equals the exponent of the group.

Warning: In the case of 2 generators P and Q, it is not guaranteed that the group is the cartesian product of the 2 cyclic groups $\langle P\rangle$ and $\langle Q\rangle$. In other words, the order of Q is not as small as possible. If you really need a basis (rather than just a generating set) of the group, use *abelian_group()*.

EXAMPLES:

```
sage: E = EllipticCurve(GF(11),[2,5])
sage: P = E.gens()[0]; P # random
(0 : 7 : 1)
sage: E.cardinality(), P.order()
(10, 10)
sage: E = EllipticCurve(GF(41),[2,5])
sage: E.gens()  # random
((20 : 38 : 1), (25 : 31 : 1))
sage: E.cardinality()
44
```

If the abelian group has been computed, return those generators instead:

```
sage: E.abelian_group()
Additive abelian group isomorphic to Z/22 + Z/2
 embedded in Abelian group of points on Elliptic Curve
 defined by y^2 = x^3 + 2*x + 5 over Finite Field of size 41
sage: ab_gens = E.abelian_group().gens()
sage: ab_gens == E.gens()
True
sage: E.gens()[0].order()
22
sage: E.gens()[1].order()
2
```

Examples with 1 and 0 generators:

```
sage: # needs sage.rings.finite_rings
sage: F.<a> = GF(3^6)
sage: E = EllipticCurve([a, a+1])
sage: pts = E.gens()
sage: len(pts)
1
sage: pts[0].order() == E.cardinality()
True
```

(continues on next page)

(continued from previous page)

```
sage: E = EllipticCurve(GF(2), [0,0,1,1,1])
sage: E.gens()
()
```

This works over larger finite fields where *abelian_group()* may be too expensive:

```
sage: # needs sage.rings.finite_rings
sage: k.<a> = GF(5^60)
sage: E = EllipticCurve([a, a])
sage: len(E.gens())
2
sage: E.cardinality()
867361737988403547206134229616487867594472
sage: a = E.gens()[0].order(); a # random
433680868994201773603067114808243933797236
sage: b = E.gens()[1].order(); b # random
309772049281572695430762224863031381283 74
sage: lcm(a,b)
433680868994201773603067114808243933797236
```

height_above_floor(*ell*, *e*)

Return the height of the j-invariant of this ordinary elliptic curve on its ℓ-volcano.

INPUT:

- `ell` – a prime number
- `e` – a non-negative integer, the ell-adic valuation of the conductor the Frobenius order

---

Note: For an ordinary $E/\mathbf{F}_q$, and a prime ℓ, the height e of the ℓ-volcano containing $j(E)$ is the ℓ-adic valuation of the conductor of the order generated by the Frobenius π_E; the height of $j(E)$ on its ell-volcano is the ℓ-adic valuation of the conductor of the order End(E).

ALGORITHM:

See [RouSuthZur2022].

EXAMPLES:

```
sage: F = GF(312401)
sage: E = EllipticCurve(F,(0, 0, 0, 309381, 93465))
sage: D = E.frobenius_discriminant(); D
-687104
sage: D.factor()
-1 * 2^10 * 11 * 61
sage: E.height_above_floor(2,8)
5
```

is_isogenous(*other*, *field=None*, *proof=True*)

Return whether or not self is isogenous to other.

INPUT:

- `other` – another elliptic curve.
- `field` (default None) – a field containing the base fields of the two elliptic curves into which the two curves may be extended to test if they are isogenous over this field. By default is_isogenous will not try

to find this field unless one of the curves can be extended into the base field of the other, in which case it will test over the larger base field.

- `proof` (default: `True`) – this parameter is here only to be consistent with versions for other types of elliptic curves.

OUTPUT:

(bool) True if there is an isogeny from curve `self` to curve `other` defined over `field`.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: E1 = EllipticCurve(GF(11^2,'a'),[2,7]); E1
Elliptic Curve defined by y^2 = x^3 + 2*x + 7 over Finite Field in a of size␣
↪11^2
sage: E1.is_isogenous(5)
Traceback (most recent call last):
...
ValueError: Second argument is not an Elliptic Curve.
sage: E1.is_isogenous(E1)
True

sage: # needs sage.rings.finite_rings
sage: E2 = EllipticCurve(GF(7^3,'b'),[3,1]); E2
Elliptic Curve defined by y^2 = x^3 + 3*x + 1 over Finite Field in b of size␣
↪7^3
sage: E1.is_isogenous(E2)
Traceback (most recent call last):
...
ValueError: The base fields must have the same characteristic.

sage: # needs sage.rings.finite_rings
sage: E3 = EllipticCurve(GF(11^2,'c'),[4,3]); E3
Elliptic Curve defined by y^2 = x^3 + 4*x + 3 over Finite Field in c of size␣
↪11^2
sage: E1.is_isogenous(E3)
False

sage: # needs sage.rings.finite_rings
sage: E4 = EllipticCurve(GF(11^6,'d'),[6,5]); E4
Elliptic Curve defined by y^2 = x^3 + 6*x + 5 over Finite Field in d of size␣
↪11^6
sage: E1.is_isogenous(E4)
True

sage: # needs sage.rings.finite_rings
sage: E5 = EllipticCurve(GF(11^7,'e'),[4,2]); E5
Elliptic Curve defined by y^2 = x^3 + 4*x + 2 over Finite Field in e of size␣
↪11^7
sage: E1.is_isogenous(E5)
Traceback (most recent call last):
...
ValueError: Curves have different base fields: use the field parameter.
```

When the field is given:

```
sage: # needs sage.rings.finite_rings
sage: E1 = EllipticCurve(GF(13^2,'a'),[2,7]); E1
```

(continues on next page)

(continued from previous page)

```
Elliptic Curve defined by y^2 = x^3 + 2*x + 7 over Finite Field in a of size␣
↪13^2
sage: E1.is_isogenous(5,GF(13^6,'f'))
Traceback (most recent call last):
...
ValueError: Second argument is not an Elliptic Curve.
sage: E6 = EllipticCurve(GF(11^3,'g'),[9,3]); E6
Elliptic Curve defined by y^2 = x^3 + 9*x + 3 over Finite Field in g of size␣
↪11^3
sage: E1.is_isogenous(E6,QQ)
Traceback (most recent call last):
...
ValueError: The base fields must have the same characteristic.
sage: E7 = EllipticCurve(GF(13^5,'h'),[2,9]); E7
Elliptic Curve defined by y^2 = x^3 + 2*x + 9 over Finite Field in h of size␣
↪13^5
sage: E1.is_isogenous(E7,GF(13^4,'i'))
Traceback (most recent call last):
...
ValueError: Field must be an extension of the base fields of both curves
sage: E1.is_isogenous(E7,GF(13^10,'j'))
False
sage: E1.is_isogenous(E7,GF(13^30,'j'))
False
```

is_ordinary (*proof=True*)

Return True if this elliptic curve is ordinary, else False.

INPUT:

- `proof` (boolean, default: `True`) – If True, returns a proved result. If False, then a return value of True is certain but a return value of False may be based on a probabilistic test. See the documentation of the function *is_j_supersingular()* for more details.

EXAMPLES:

```
sage: F = GF(101)
sage: EllipticCurve(j=F(0)).is_ordinary()
False
sage: EllipticCurve(j=F(1728)).is_ordinary()
True
sage: EllipticCurve(j=F(66)).is_ordinary()
False
sage: EllipticCurve(j=F(99)).is_ordinary()
True
```

is_supersingular (*proof=True*)

Return True if this elliptic curve is supersingular, else False.

INPUT:

- `proof` (boolean, default: `True`) – If True, returns a proved result. If False, then a return value of False is certain but a return value of True may be based on a probabilistic test. See the documentation of the function *is_j_supersingular()* for more details.

EXAMPLES:

```
sage: F = GF(101)
sage: EllipticCurve(j=F(0)).is_supersingular()
True
sage: EllipticCurve(j=F(1728)).is_supersingular()
False
sage: EllipticCurve(j=F(66)).is_supersingular()
True
sage: EllipticCurve(j=F(99)).is_supersingular()
False
```

multiplication_by_p_isogeny()

Return the multiplication-by-p isogeny.

EXAMPLES:

```
sage: p = 23
sage: K.<a> = GF(p^3)
sage: E = EllipticCurve(j=K.random_element())
sage: phi = E.multiplication_by_p_isogeny()
sage: assert phi.degree() == p**2
sage: P = E.random_element()
sage: assert phi(P) == P * p
```

order(*algorithm=None*, *extension_degree=1*)

Return the number of points on this elliptic curve.

INPUT:

- `algorithm` – (optional) string:
  - `'pari'` – use the PARI C-library function `ellcard`.
  - **`'bsgs'` – use the baby-step giant-step method as** implemented in Sage, with the Cremona-Sutherland version of Mestre's trick.
  - `'exhaustive'` – naive point counting.
  - `'subfield'` – reduce to a smaller field, provided that the j-invariant lies in a subfield.
  - `'all'` – compute cardinality with both `'pari'` and `'bsgs'`; return result if they agree or raise a `AssertionError` if they do not
- `extension_degree` – an integer d (default: 1): if the base field is $\mathbf{F}_q$, return the cardinality of `self` over the extension $\mathbf{F}_{q^d}$ of degree d.

OUTPUT:

The order of the group of rational points of `self` over its base field, or over an extension field of degree d as above. The result is cached.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: EllipticCurve(GF(4, 'a'), [1,2,3,4,5]).cardinality()
8
sage: k.<a> = GF(3^3)
sage: l = [a^2 + 1, 2*a^2 + 2*a + 1, a^2 + a + 1, 2, 2*a]
sage: EllipticCurve(k,l).cardinality()
29
```

```
sage: # needs sage.rings.finite_rings
sage: l = [1, 1, 0, 2, 0]
sage: EllipticCurve(k, l).cardinality()
38
```

An even bigger extension (which we check against Magma):

```
sage: # needs sage.rings.finite_rings
sage: EllipticCurve(GF(3^100, 'a'), [1,2,3,4,5]).cardinality()
515377520732011331036459693969645888996929981504
sage: magma.eval("Order(EllipticCurve([GF(3^100)|1,2,3,4,5]))")     # optional␣
↪- magma
'515377520732011331036459693969645888996929981504'
```

```
sage: EllipticCurve(GF(10007), [1,2,3,4,5]).cardinality()
10076
sage: EllipticCurve(GF(10007), [1,2,3,4,5]).cardinality(algorithm='pari')
10076
sage: EllipticCurve(GF(next_prime(10**20)), [1,2,3,4,5]).cardinality()
100000000011093199520
```

The cardinality is cached:

```
sage: # needs sage.rings.finite_rings
sage: E = EllipticCurve(GF(3^100, 'a'), [1,2,3,4,5])
sage: E.cardinality() is E.cardinality()
True
```

The following is very fast since the curve is actually defined over the prime field:

```
sage: # needs sage.rings.finite_rings
sage: k.<a> = GF(11^100)
sage: E1 = EllipticCurve(k, [3,3])
sage: N1 = E1.cardinality(algorithm="subfield"); N1
137806123398222701841183371720896367762643312000384671846835266941791510341065565176497846500
sage: E1.cardinality_pari() == N1
True
sage: E2 = E1.quadratic_twist()
sage: N2 = E2.cardinality(algorithm="subfield"); N2
137806123398222701841183371720896367762643312000384656816094284101308193849980588362304472490
sage: E2.cardinality_pari() == N2
True
sage: N1 + N2 == 2*(k.cardinality() + 1)
True
```

We can count points over curves defined as a reduction:

```
sage: # needs sage.rings.number_field
sage: x = polygen(QQ)
sage: K.<w> = NumberField(x^2 + x + 1)
sage: EK = EllipticCurve(K, [0, 0, w, 2, 1])
sage: E = EK.base_extend(K.residue_field(2))
sage: E
Elliptic Curve defined by y^2 + wbar*y = x^3 + 1
 over Residue field in wbar of Fractional ideal (2)
sage: E.cardinality()
7
```

(continues on next page)

(continued from previous page)

```
sage: E = EK.base_extend(K.residue_field(w - 1))
sage: E.abelian_group()
Trivial group embedded in Abelian group of points on Elliptic Curve defined
 by y^2 + y = x^3 + 2*x + 1 over Residue field of Fractional ideal (w - 1)
```

```
sage: R.<x> = GF(17)[]
sage: pol = R.irreducible_element(5)
sage: k.<a> = R.residue_field(pol)
sage: E = EllipticCurve(R, [1, x]).base_extend(k)
sage: E
Elliptic Curve defined by y^2 = x^3 + x + a
 over Residue field in a of Principal ideal (x^5 + x + 14)
  of Univariate Polynomial Ring in x over Finite Field of size 17
sage: E.cardinality()
1421004
```

plot(**args*, ***kwds*)

Draw a graph of this elliptic curve over a prime finite field.

INPUT:

- `*args`, `**kwds` – all other options are passed to the circle graphing primitive.

EXAMPLES:

```
sage: E = EllipticCurve(FiniteField(17), [0,1])
sage: P = plot(E, rgbcolor=(0,0,1))                                    #␣
↪needs sage.plot
```

points()

Return all rational points on this elliptic curve. The list of points is cached so subsequent calls are free.

EXAMPLES:

```
sage: p = 5
sage: F = GF(p)
sage: E = EllipticCurve(F, [1, 3])
sage: len(E.points())
4
sage: E.order()
4
sage: E.points()
[(0 : 1 : 0), (1 : 0 : 1), (4 : 1 : 1), (4 : 4 : 1)]
```

```
sage: K = GF((p, 2), 'a')
sage: E = E.change_ring(K)
sage: len(E.points())
32
sage: E.order()
32
sage: w = E.points(); w
[(0 : 1 : 0), (0 : 2*a + 4 : 1), (0 : 3*a + 1 : 1), (1 : 0 : 1), (2 : 2*a + 4␣
↪: 1), (2 : 3*a + 1 : 1), (3 : 2*a + 4 : 1), (3 : 3*a + 1 : 1), (4 : 1 : 1),␣
↪(4 : 4 : 1), (a : 1 : 1), (a : 4 : 1), (a + 2 : a + 1 : 1), (a + 2 : 4*a +␣
↪4 : 1), (a + 3 : a : 1), (a + 3 : 4*a : 1), (a + 4 : 0 : 1), (2*a : 2*a :␣
↪1), (2*a : 3*a : 1), (2*a + 4 : a + 1 : 1), (2*a + 4 : 4*a + 4 : 1), (3*a +␣
↪1 : a + 3 : 1), (3*a + 1 : 4*a + 2 : 1), (3*a + 2 : 2*a + 3 : 1), (3*a + 2␣
```

(continues on next page)

(continued from previous page)

```
↪: 3*a + 2 : 1), (4*a : 0 : 1), (4*a + 1 : 1 : 1), (4*a + 1 : 4 : 1), (4*a +␣
↪3 : a + 3 : 1), (4*a + 3 : 4*a + 2 : 1), (4*a + 4 : a + 4 : 1), (4*a + 4 :␣
↪4*a + 1 : 1)]
```

Note that the returned list is an immutable sorted Sequence:

```
sage: w[0] = 9
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

random_element()

Return a random point on this elliptic curve, uniformly chosen among all rational points.

ALGORITHM:

Choose the point at infinity with probability $1/(2q+1)$. Otherwise, take a random element from the field as x-coordinate and compute the possible y-coordinates. Return the i'th possible y-coordinate, where i is randomly chosen to be 0 or 1. If the i'th y-coordinate does not exist (either there is no point with the given x-coordinate or we hit a 2-torsion point with i == 1), try again.

This gives a uniform distribution because you can imagine $2q+1$ buckets, one for the point at infinity and 2 for each element of the field (representing the x-coordinates). This gives a 1-to-1 map of elliptic curve points into buckets. At every iteration, we simply choose a random bucket until we find a bucket containing a point.

AUTHORS:

- Jeroen Demeyer (2014-09-09): choose points uniformly random, see Issue #16951.

EXAMPLES:

```
sage: k = GF(next_prime(7^5))
sage: E = EllipticCurve(k,[2,4])
sage: P = E.random_element(); P  # random
(16740 : 12486 : 1)
sage: type(P)
<class 'sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_finite_field
↪'>
sage: P in E
True
```

```
sage: # needs sage.rings.finite_rings
sage: k.<a> = GF(7^5)
sage: E = EllipticCurve(k,[2,4])
sage: P = E.random_element(); P  # random
(5*a^4 + 3*a^3 + 2*a^2 + a + 4 : 2*a^4 + 3*a^3 + 4*a^2 + a + 5 : 1)
sage: type(P)
<class 'sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_finite_field
↪'>
sage: P in E
True
```

```
sage: # needs sage.rings.finite_rings
sage: k.<a> = GF(2^5)
sage: E = EllipticCurve(k,[a^2,a,1,a+1,1])
sage: P = E.random_element(); P  # random
(a^4 + a : a^4 + a^3 + a^2 : 1)
```

(continues on next page)

(continued from previous page)

```
sage: type(P)
<class 'sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_finite_field
↪'>
sage: P in E
True
```

Ensure that the entire point set is reachable:

```
sage: E = EllipticCurve(GF(11), [2,1])
sage: S = set()
sage: while len(S) < E.cardinality():
....:     S.add(E.random_element())
```

random_point()

Return a random point on this elliptic curve, uniformly chosen among all rational points.

ALGORITHM:

Choose the point at infinity with probability $1/(2q+1)$. Otherwise, take a random element from the field as x-coordinate and compute the possible y-coordinates. Return the i'th possible y-coordinate, where i is randomly chosen to be 0 or 1. If the i'th y-coordinate does not exist (either there is no point with the given x-coordinate or we hit a 2-torsion point with i == 1), try again.

This gives a uniform distribution because you can imagine $2q+1$ buckets, one for the point at infinity and 2 for each element of the field (representing the x-coordinates). This gives a 1-to-1 map of elliptic curve points into buckets. At every iteration, we simply choose a random bucket until we find a bucket containing a point.

AUTHORS:

- Jeroen Demeyer (2014-09-09): choose points uniformly random, see Issue #16951.

EXAMPLES:

```
sage: k = GF(next_prime(7^5))
sage: E = EllipticCurve(k,[2,4])
sage: P = E.random_element(); P  # random
(16740 : 12486 : 1)
sage: type(P)
<class 'sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_finite_field
↪'>
sage: P in E
True
```

```
sage: # needs sage.rings.finite_rings
sage: k.<a> = GF(7^5)
sage: E = EllipticCurve(k,[2,4])
sage: P = E.random_element(); P  # random
(5*a^4 + 3*a^3 + 2*a^2 + a + 4 : 2*a^4 + 3*a^3 + 4*a^2 + a + 5 : 1)
sage: type(P)
<class 'sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_finite_field
↪'>
sage: P in E
True
```

```
sage: # needs sage.rings.finite_rings
sage: k.<a> = GF(2^5)
sage: E = EllipticCurve(k,[a^2,a,1,a+1,1])
```

(continues on next page)

(continued from previous page)

```
sage: P = E.random_element(); P  # random
(a^4 + a : a^4 + a^3 + a^2 : 1)
sage: type(P)
<class 'sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_finite_field
↪'>
sage: P in E
True
```

Ensure that the entire point set is reachable:

```
sage: E = EllipticCurve(GF(11), [2,1])
sage: S = set()
sage: while len(S) < E.cardinality():
....:     S.add(E.random_element())
```

rational_points()

Return all rational points on this elliptic curve. The list of points is cached so subsequent calls are free.

EXAMPLES:

```
sage: p = 5
sage: F = GF(p)
sage: E = EllipticCurve(F, [1, 3])
sage: len(E.points())
4
sage: E.order()
4
sage: E.points()
[(0 : 1 : 0), (1 : 0 : 1), (4 : 1 : 1), (4 : 4 : 1)]
```

```
sage: K = GF((p, 2), 'a')
sage: E = E.change_ring(K)
sage: len(E.points())
32
sage: E.order()
32
sage: w = E.points(); w
[(0 : 1 : 0), (0 : 2*a + 4 : 1), (0 : 3*a + 1 : 1), (1 : 0 : 1), (2 : 2*a + 4␣
↪: 1), (2 : 3*a + 1 : 1), (3 : 2*a + 4 : 1), (3 : 3*a + 1 : 1), (4 : 1 : 1),␣
↪(4 : 4 : 1), (a : 1 : 1), (a : 4 : 1), (a + 2 : a + 1 : 1), (a + 2 : 4*a +␣
↪4 : 1), (a + 3 : a : 1), (a + 3 : 4*a : 1), (a + 4 : 0 : 1), (2*a : 2*a :␣
↪1), (2*a : 3*a : 1), (2*a + 4 : a + 1 : 1), (2*a + 4 : 4*a + 4 : 1), (3*a +␣
↪1 : a + 3 : 1), (3*a + 1 : 4*a + 2 : 1), (3*a + 2 : 2*a + 3 : 1), (3*a + 2␣
↪: 3*a + 2 : 1), (4*a : 0 : 1), (4*a + 1 : 1 : 1), (4*a + 1 : 4 : 1), (4*a +␣
↪3 : a + 3 : 1), (4*a + 3 : 4*a + 2 : 1), (4*a + 4 : a + 4 : 1), (4*a + 4 :␣
↪4*a + 1 : 1)]
```

Note that the returned list is an immutable sorted Sequence:

```
sage: w[0] = 9
Traceback (most recent call last):
...
ValueError: object is immutable; please change a copy instead.
```

set_order(*value*, *check*, *num_checks*)

Set the value of `self._order` to `value`.

Use this when you know a priori the order of the curve to avoid a potentially expensive order calculation.

INPUT:

- `value` – integer in the Hasse-Weil range for this curve.
- `check` (boolean, default: `True`) – whether or not to run sanity checks on the input.
- `num_checks` (integer, default: 8) – if `check` is `True`, the number of times to check whether `value` times a random point on this curve equals the identity.

OUTPUT:

None

EXAMPLES:

This example illustrates basic usage:

```
sage: E = EllipticCurve(GF(7), [0, 1]) # This curve has order 12
sage: E.set_order(12)
sage: E.order()
12
sage: E.order() * E.random_point()
(0 : 1 : 0)
```

We now give a more interesting case, the NIST-P521 curve. Its order is too big to calculate with Sage, and takes a long time using other packages, so it is very useful here:

```
sage: p = 2^521 - 1
sage: prev_proof_state = proof.arithmetic()
sage: proof.arithmetic(False) # turn off primality checking
sage: F = GF(p)
sage: A = p - 3
sage: B =␣
↪1093849038073734274511112390766805569936207598951683748994586394495953116150735016013708737
sage: q =␣
↪6864797660130609714981900799081393217269435300143305409394463459185543183397655394245057746
sage: E = EllipticCurve([F(A), F(B)])
sage: E.set_order(q)
sage: G = E.random_point()
sage: G.order() * G  # This takes practically no time.
(0 : 1 : 0)
sage: proof.arithmetic(prev_proof_state) # restore state
```

It is an error to pass a value which is not an integer in the Hasse-Weil range:

```
sage: E = EllipticCurve(GF(7), [0, 1]) # This curve has order 12
sage: E.set_order("hi")
Traceback (most recent call last):
...
TypeError: unable to convert 'hi' to an integer
sage: E.set_order(0)
Traceback (most recent call last):
...
ValueError: Value 0 illegal (not an integer in the Hasse range)
sage: E.set_order(1000)
Traceback (most recent call last):
...
ValueError: Value 1000 illegal (not an integer in the Hasse range)
```

It is also very likely an error to pass a value which is not the actual order of this curve. How unlikely is determined by `num_checks`, the factorization of the actual order, and the actual group structure:

```
sage: E = EllipticCurve(GF(1009), [0, 1]) # This curve has order 948
sage: E.set_order(947)
Traceback (most recent call last):
...
ValueError: Value 947 illegal (multiple of random point not the identity)
```

For curves over small finite fields, the order is cheap to compute, so it is computed directly and compared:

```
sage: E = EllipticCurve(GF(7), [0, 1]) # This curve has order 12
sage: E.set_order(11)
Traceback (most recent call last):
...
ValueError: Value 11 illegal (correct order is 12)
```

Todo: Add provable correctness check by computing the abelian group structure and comparing.

AUTHORS:

- Mariah Lenox (2011-02-16): Initial implementation
- Gareth Ma (2024-01-21): Fix bug for small curves

torsion_basis(n)

Return a basis of the n-torsion subgroup of this elliptic curve, assuming it is fully rational.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: E = EllipticCurve(GF(62207^2), [1,0])
sage: E.abelian_group()
Additive abelian group isomorphic to Z/62208 + Z/62208 embedded in
 Abelian group of points on Elliptic Curve defined by y^2 = x^3 + x
  over Finite Field in z2 of size 62207^2
sage: PA,QA = E.torsion_basis(2^8)
sage: PA.weil_pairing(QA, 2^8).multiplicative_order()
256
sage: PB,QB = E.torsion_basis(3^5)
sage: PB.weil_pairing(QB, 3^5).multiplicative_order()
243
```

```
sage: E = EllipticCurve(GF(101), [4,4])
sage: E.torsion_basis(23)
Traceback (most recent call last):
...
ValueError: curve does not have full rational 23-torsion
sage: F = E.division_field(23); F
Finite Field in t of size 101^11
sage: EE = E.change_ring(F)
sage: P, Q = EE.torsion_basis(23)
sage: P  # random
(89*z11^10 + 51*z11^9 + 96*z11^8 + 8*z11^7 + 67*z11^6
 + 31*z11^5 + 55*z11^4 + 59*z11^3 + 28*z11^2 + 8*z11 + 88
 : 40*z11^10 + 33*z11^9 + 80*z11^8 + 87*z11^7 + 97*z11^6
 + 69*z11^5 + 56*z11^4 + 17*z11^3 + 26*z11^2 + 69*z11 + 11
```

(continues on next page)

(continued from previous page)

```
 : 1)
sage: Q  # random
(25*z11^10 + 61*z11^9 + 49*z11^8 + 17*z11^7 + 80*z11^6
 + 20*z11^5 + 49*z11^4 + 52*z11^3 + 61*z11^2 + 27*z11 + 61
 : 60*z11^10 + 91*z11^9 + 89*z11^8 + 7*z11^7 + 63*z11^6
 + 55*z11^5 + 23*z11^4 + 17*z11^3 + 90*z11^2 + 91*z11 + 68
 : 1)
```

See also:

Use *division_field()* to determine a field extension containing the full ℓ-torsion subgroup.

ALGORITHM:

This method currently uses *abelian_group()* and AdditiveAbelianGroupWrapper.torsion_subgroup().

trace_of_frobenius()

Return the trace of Frobenius acting on this elliptic curve.

Note: This computes the curve cardinality, which may be time-consuming.

EXAMPLES:

```
sage: E = EllipticCurve(GF(101),[2,3])
sage: E.trace_of_frobenius()
6
sage: E = EllipticCurve(GF(11^5,'a'),[2,5])                                #␣
↪needs sage.rings.finite_rings
sage: E.trace_of_frobenius()                                               #␣
↪needs sage.rings.finite_rings
802
```

The following shows that the issue from Issue #2849 is fixed:

```
sage: E = EllipticCurve(GF(3^5,'a'),[-1,-1])                               #␣
↪needs sage.rings.finite_rings
sage: E.trace_of_frobenius()                                               #␣
↪needs sage.rings.finite_rings
-27
```

twists()

Return a list of k-isomorphism representatives of all twists of this elliptic curve, where k is the base field.

The input curve appears as the first entry of the result.

Note: A *twist* of E/k is an elliptic curve E' defined over k that is isomorphic to E over the algebraic closure $\bar{k}$.

Most elliptic curves over a finite field only admit a single nontrivial twist (the quadratic twist); the only exceptions are curves with j-invariant 0 or 1728.

In all cases the sum over all the twists E' of $1/|Aut(E')|$ is 1.

See also:

- *quadratic_twist()*
- *quartic_twist()*
- *sextic_twist()*

EXAMPLES:

```
sage: E = EllipticCurve(GF(97), [1,1])
sage: E.j_invariant()
54
sage: E.twists()
[Elliptic Curve defined by y^2 = x^3 + x + 1 over Finite Field of size 97,
 Elliptic Curve defined by y^2 = x^3 + ... over Finite Field of size 97]
```

```
sage: E = EllipticCurve(GF(97), [1,0])
sage: E.j_invariant()
79
sage: E.twists()
[Elliptic Curve defined by y^2 = x^3 + x over Finite Field of size 97,
 Elliptic Curve defined by y^2 = x^3 + ... over Finite Field of size 97,
 Elliptic Curve defined by y^2 = x^3 + ... over Finite Field of size 97,
 Elliptic Curve defined by y^2 = x^3 + ... over Finite Field of size 97]
```

```
sage: E = EllipticCurve(GF(97), [0,1])
sage: E.j_invariant()
0
sage: E.twists()
[Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field of size 97,
 Elliptic Curve defined by y^2 = x^3 + ... over Finite Field of size 97,
 Elliptic Curve defined by y^2 = x^3 + ... over Finite Field of size 97,
 Elliptic Curve defined by y^2 = x^3 + ... over Finite Field of size 97,
 Elliptic Curve defined by y^2 = x^3 + ... over Finite Field of size 97,
 Elliptic Curve defined by y^2 = x^3 + ... over Finite Field of size 97]
```

This can be useful to quickly compute a list of all elliptic curves over a finite field k up to k-isomorphism:

```
sage: Es = [E for j in GF(13) for E in EllipticCurve(j=j).twists()]
sage: len(Es)
32
sage: Es
[Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field of size 13,
 ...
 Elliptic Curve defined by y^2 = x^3 + ... over Finite Field of size 13]
```

In characteristic 3, the number of twists is 2 except for $j = 0 = 1728$, when there are either 4 or 6 depending on whether the field has odd or even degree over $\mathbf{F}_3$:

```
sage: # needs sage.rings.finite_rings
sage: K = GF(3**5)
sage: [E.ainvs() for E in EllipticCurve(j=K(1)).twists()]
[(0, 1, 0, 0, 2), (0, z5, 0, 0, 2*z5^3)]

sage: # needs sage.rings.finite_rings
sage: K = GF(3**5)
sage: [E.ainvs() for E in EllipticCurve(j=K(0)).twists()] # random
[(0, 0, 0, 1, 0),
 (0, 0, 0, 2, 0),
```

(continues on next page)

(continued from previous page)

```
 (0, 0, 0, 2, z5^4 + z5^3 + z5^2),
 (0, 0, 0, 2, 2*z5^4 + 2*z5^3 + 2*z5^2)]

sage: # needs sage.rings.finite_rings
sage: K = GF(3**4)
sage: [E.ainvs() for E in EllipticCurve(j=K(1)).twists()]
[(0, 1, 0, 0, 2), (0, z4, 0, 0, 2*z4^3)]

sage: # needs sage.rings.finite_rings
sage: K = GF(3**4)
sage: [E.ainvs() for E in EllipticCurve(j=K(0)).twists()] # random
[(0, 0, 0, 1, 0),
 (0, 0, 0, 2, 2*z4^3 + 2*z4^2 + 2*z4 + 2),
 (0, 0, 0, 1, 0),
 (0, 0, 0, 1, 2*z4^3 + 2*z4^2 + 2*z4 + 2),
 (0, 0, 0, z4, 0),
 (0, 0, 0, z4^3, 0)]
```

In characteristic 2, the number of twists is 2 except for $j = 0 = 1728$, when there are either 3 or 7 depending on whether the field has odd or even degree over $\mathbf{F}_2$:

```
sage: # needs sage.rings.finite_rings
sage: K = GF(2**7)
sage: [E.ainvs() for E in EllipticCurve(j=K(1)).twists()]
[(1, 0, 0, 0, 1), (1, 1, 0, 0, 1)]

sage: # needs sage.rings.finite_rings
sage: K = GF(2**7)
sage: [E.ainvs() for E in EllipticCurve(j=K(0)).twists()]
[(0, 0, 1, 0, 0), (0, 0, 1, 1, 0), (0, 0, 1, 1, 1)]

sage: # needs sage.rings.finite_rings
sage: K = GF(2**8)
sage: [E.ainvs() for E in EllipticCurve(j=K(1)).twists()] # random
[(1, 0, 0, 0, 1), (1, z8^7 + z8^6 + z8^5 + z8^4 + z8^2 + z8, 0, 0, 1)]

sage: # needs sage.rings.finite_rings
sage: K = GF(2**8)
sage: [E.ainvs() for E in EllipticCurve(j=K(0)).twists()] # random
[(0, 0, 1, 0, 0),
 (0, 0, 1, 0, z8^5 + z8^4 + z8^3),
 (0, 0, 1, z8^6 + z8^5 + z8^2 + 1, 0),
 (0, 0, z8^4 + z8^3 + z8^2 + 1, 0, 0),
 (0, 0, z8^4 + z8^3 + z8^2 + 1, 0, z8^3 + z8^2 + 1),
 (0, 0, z8^6 + z8^3 + z8^2, 0, 0),
 (0, 0, z8^6 + z8^3 + z8^2, 0, z8^3 + z8^2)]
```

sage.schemes.elliptic_curves.ell_finite_field.**EllipticCurve_with_order**(*m*, *D*)

Return an iterator for elliptic curves over finite fields with the given order. The curves are computed using the Complex Multiplication (CM) method.

A : *sage* : ` *sage.structure.factorization.Factorization* can be passed for `m`, in which case the algorithm is more efficient.

If `D` is specified, it is used as the discriminant.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_finite_field import EllipticCurve_
↪with_order
sage: E = next(EllipticCurve_with_order(1234)); E  # random
Elliptic Curve defined by y^2 = x^3 + 1142*x + 1209 over Finite Field of size 1237
sage: E.order() == 1234
True
```

When `iter` is set, the function returns an iterator of all elliptic curves with the given order:

```
sage: from sage.schemes.elliptic_curves.ell_finite_field import EllipticCurve_
↪with_order
sage: it = EllipticCurve_with_order(21); it
<generator object EllipticCurve_with_order at 0x...>
sage: E = next(it); E  # random
Elliptic Curve defined by y^2 = x^3 + 6*x + 14 over Finite Field of size 23
sage: E.order() == 21
True
sage: Es = [E] + list(it); Es  # random
[Elliptic Curve defined by y^2 = x^3 + 6*x + 14 over Finite Field of size 23,
 Elliptic Curve defined by y^2 = x^3 + 12*x + 4 over Finite Field of size 23,
 Elliptic Curve defined by y^2 = x^3 + 5*x + 2 over Finite Field of size 23,
 Elliptic Curve defined by y^2 = x^3 + (z2+3) over Finite Field in z2 of size 5^2,
 Elliptic Curve defined by y^2 = x^3 + (2*z2+2) over Finite Field in z2 of size 5^
↪2,
 Elliptic Curve defined by y^2 = x^3 + 7*x + 1 over Finite Field of size 19,
 Elliptic Curve defined by y^2 = x^3 + 17*x + 10 over Finite Field of size 19,
 Elliptic Curve defined by y^2 = x^3 + 5*x + 12 over Finite Field of size 17,
 Elliptic Curve defined by y^2 = x^3 + 9*x + 1 over Finite Field of size 17,
 Elliptic Curve defined by y^2 = x^3 + 7*x + 6 over Finite Field of size 17,
 Elliptic Curve defined by y^2 = x^3 + z3^2*x^2 + (2*z3^2+z3) over Finite Field␣
↪in z3 of size 3^3,
 Elliptic Curve defined by y^2 = x^3 + (z3^2+2*z3+1)*x^2 + (2*z3^2+2*z3) over␣
↪Finite Field in z3 of size 3^3,
 Elliptic Curve defined by y^2 = x^3 + (z3^2+z3+1)*x^2 + (2*z3^2+1) over Finite␣
↪Field in z3 of size 3^3,
 Elliptic Curve defined by y^2 + (z4^2+z4+1)*y = x^3 over Finite Field in z4 of␣
↪size 2^4,
 Elliptic Curve defined by y^2 + (z4^2+z4)*y = x^3 over Finite Field in z4 of␣
↪size 2^4,
 Elliptic Curve defined by y^2 = x^3 + 18*x + 26 over Finite Field of size 29,
 Elliptic Curve defined by y^2 = x^3 + 11*x + 19 over Finite Field of size 29,
 Elliptic Curve defined by y^2 = x^3 + 4 over Finite Field of size 19,
 Elliptic Curve defined by y^2 = x^3 + 19 over Finite Field of size 31,
 Elliptic Curve defined by y^2 = x^3 + 4 over Finite Field of size 13]
sage: all(E.order() == 21 for E in Es)
True
```

Indeed, we can verify that this is correct. Hasse's bounds tell us that $p \leq 50$ (approximately), and the rest can be checked via bruteforce:

```
sage: for p in prime_range(50):
....:     for j in range(p):
....:         E0 = EllipticCurve(GF(p), j=j)
....:         for Et in E0.twists():
....:             if Et.order() == 21:
....:                 assert any(Et.is_isomorphic(E) for E in Es)
```

Note: The output curves are not deterministic, as *EllipticCurve_finite_field.twists()* is not deterministic. However, the order of the j-invariants and base fields is fixed.

AUTHORS:

- Gareth Ma and Giacomo Pope (Sage Days 123): initial version

sage.schemes.elliptic_curves.ell_finite_field.**curves_with_j_0**(K)

Return a complete list of pairwise nonisomorphic elliptic curves with j-invariant 0 over the finite field K.

Note: In characteristics 2 and 3 this function simply calls `curves_with_j_0_char2` or `curves_with_j_0_char3`. Otherwise there are either 2 or 6 curves, parametrised by $K^*/(K^*)^6$.

Examples:

For $K = \mathbf{F}_q$ where $q \equiv 1 \mod 6$ there are six curves, the sextic twists of $y^2 = x^3 + 1$:

```
sage: # needs sage.rings.finite_rings
sage: from sage.schemes.elliptic_curves.ell_finite_field import curves_with_j_0
sage: sorted(curves_with_j_0(GF(7)), key = lambda E: E.a_invariants())
[Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field of size 7,
 Elliptic Curve defined by y^2 = x^3 + 2 over Finite Field of size 7,
 Elliptic Curve defined by y^2 = x^3 + 3 over Finite Field of size 7,
 Elliptic Curve defined by y^2 = x^3 + 4 over Finite Field of size 7,
 Elliptic Curve defined by y^2 = x^3 + 5 over Finite Field of size 7,
 Elliptic Curve defined by y^2 = x^3 + 6 over Finite Field of size 7]
sage: curves = curves_with_j_0(GF(25)); len(curves)
6
sage: all(not curves[i].is_isomorphic(curves[j]) for i in range(6) for j in␣
↪range(i + 1, 6))
True
sage: set(E.j_invariant() for E in curves)
{0}
```

For $K = \mathbf{F}_q$ where $q \equiv 5 \mod 6$ there are two curves, quadratic twists of each other by -3: $y^2 = x^3 + 1$ and $y^2 = x^3 - 27$:

```
sage: from sage.schemes.elliptic_curves.ell_finite_field import curves_with_j_0
sage: curves_with_j_0(GF(5))
[Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field of size 5,
 Elliptic Curve defined by y^2 = x^3 + 3 over Finite Field of size 5]
sage: curves_with_j_0(GF(11))
[Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field of size 11,
 Elliptic Curve defined by y^2 = x^3 + 6 over Finite Field of size 11]
```

sage.schemes.elliptic_curves.ell_finite_field.**curves_with_j_0_char2**(K)

Return a complete list of pairwise nonisomorphic elliptic curves with j-invariant 0 over the finite field K of characteristic 2.

Note: The number of twists is either 3 or 7 depending on whether the field has odd or even degree over $\mathbf{F}_2$. See [Connell1999], pages 429-431.

Examples:

In odd degree, there are three isomorphism classes all with representatives defined over $\mathbf{F}_2$:

```
sage: from sage.schemes.elliptic_curves.ell_finite_field import curves_with_j_0_
↪char2
sage: # needs sage.rings.finite_rings
sage: K = GF(2**7)
sage: curves = curves_with_j_0_char2(K)
sage: len(curves)
3
sage: [E.ainvs() for E in curves]
[(0, 0, 1, 0, 0), (0, 0, 1, 1, 0), (0, 0, 1, 1, 1)]
```

Check that the curves are mutually non-isomorphic:

```
sage: all((e1 == e2 or not e1.is_isomorphic(e2))                                #␣
↪needs sage.rings.finite_rings
....:     for e1 in curves for e2 in curves)
True
```

Check that the weight formula holds:

```
sage: sum(1/len(E.automorphisms()) for E in curves) == 1                        #␣
↪needs sage.rings.finite_rings
True
```

In even degree there are seven isomorphism classes:

```
sage: from sage.schemes.elliptic_curves.ell_finite_field import curves_with_j_0_
↪char2
sage: # needs sage.rings.finite_rings
sage: K = GF(2**8)
sage: curves = EllipticCurve(j=K(0)).twists()
sage: len(curves)
7
sage: [E.ainvs() for E in curves] # random
[(0, 0, 1, 0, 0),
 (0, 0, 1, 0, z8^5 + z8^4 + z8^3),
 (0, 0, 1, z8^6 + z8^5 + z8^2 + 1, 0),
 (0, 0, z8^4 + z8^3 + z8^2 + 1, 0, 0),
 (0, 0, z8^4 + z8^3 + z8^2 + 1, 0, z8^3 + z8^2 + 1),
 (0, 0, z8^6 + z8^3 + z8^2, 0, 0),
 (0, 0, z8^6 + z8^3 + z8^2, 0, z8^3 + z8^2)]
```

Check that the twists are mutually non-isomorphic:

```
sage: all((e1 == e2 or not e1.is_isomorphic(e2))                                #␣
↪needs sage.rings.finite_rings
....:     for e1 in curves for e2 in curves)
True
```

Check that the weight formula holds:

```
sage: sum(1/len(E.automorphisms()) for E in curves) == 1                        #␣
↪needs sage.rings.finite_rings
True
```

sage.schemes.elliptic_curves.ell_finite_field.**curves_with_j_0_char3**(K)

Return a complete list of pairwise nonisomorphic elliptic curves with j-invariant 0 over the finite field K of characteristic 3.

Note: The number of twists is either 4 or 6 depending on whether the field has odd or even degree over $\mathbf{F}_3$. See [Connell1999], pages 429-431.

Examples:

In odd degree, there are four isomorphism classes:

```
sage: from sage.schemes.elliptic_curves.ell_finite_field import curves_with_j_0_
↪char3
sage: # needs sage.rings.finite_rings
sage: K = GF(3**5)
sage: curves = curves_with_j_0_char3(K)
sage: len(curves)
4
sage: [E.ainvs() for E in curves] # random
[(0, 0, 0, 1, 0),
 (0, 0, 0, 2, 0),
 (0, 0, 0, 2, z5^4 + z5^3 + z5^2),
 (0, 0, 0, 2, 2*z5^4 + 2*z5^3 + 2*z5^2)]
```

Check that the twists are mutually non-isomorphic:

```
sage: all((e1 == e2 or not e1.is_isomorphic(e2))                                #␣
↪needs sage.rings.finite_rings
....:     for e1 in curves for e2 in curves)
True
```

Check that the weight formula holds:

```
sage: sum(1/len(E.automorphisms()) for E in curves) == 1                        #␣
↪needs sage.rings.finite_rings
True
```

In even degree, there are six isomorphism classes:

```
sage: from sage.schemes.elliptic_curves.ell_finite_field import curves_with_j_0_
↪char3
sage: # needs sage.rings.finite_rings
sage: K = GF(3**4)
sage: curves = EllipticCurve(j=K(0)).twists()
sage: len(curves)
6
sage: [E.ainvs() for E in curves] # random
[(0, 0, 0, 1, 0),
 (0, 0, 0, 2, 2*z4^3 + 2*z4^2 + 2*z4 + 2),
 (0, 0, 0, 1, 0),
 (0, 0, 0, 1, 2*z4^3 + 2*z4^2 + 2*z4 + 2),
 (0, 0, 0, z4, 0),
 (0, 0, 0, z4^3, 0)]
```

Check that the twists are mutually non-isomorphic:

```
sage: all((e1 == e2 or not e1.is_isomorphic(e2))                                #␣
↪needs sage.rings.finite_rings
```

(continues on next page)

(continued from previous page)

```
....:     for e1 in curves for e2 in curves)
True
```

Check that the weight formula holds:

```
sage: sum(1/len(E.automorphisms()) for E in curves) == 1                    #␣
↪needs sage.rings.finite_rings
True
```

sage.schemes.elliptic_curves.ell_finite_field.**curves_with_j_1728**(K)

Return a complete list of pairwise nonisomorphic elliptic curves with j-invariant 1728 over the finite field K.

Note: In characteristics 2 and 3 (so 0=1728) this function simply calls `curves_with_j_0_char2` or `curves_with_j_0_char3`. Otherwise there are either 2 or 4 curves, parametrised by $K^*/(K^*)^4$.

EXAMPLES:

For $K = \mathbf{F}_q$ where $q \equiv 1 \mod 4$, there are four curves, the quartic twists of $y^2 = x^3 + x$:

```
sage: from sage.schemes.elliptic_curves.ell_finite_field import curves_with_j_1728
sage: sorted(curves_with_j_1728(GF(5)), key = lambda E: E.a_invariants())
[Elliptic Curve defined by y^2 = x^3 + x over Finite Field of size 5,
 Elliptic Curve defined by y^2 = x^3 + 2*x over Finite Field of size 5,
 Elliptic Curve defined by y^2 = x^3 + 3*x over Finite Field of size 5,
 Elliptic Curve defined by y^2 = x^3 + 4*x over Finite Field of size 5]
sage: curves_with_j_1728(GF(49))  # random                                  #␣
↪needs sage.rings.finite_rings
[Elliptic Curve defined by y^2 = x^3 + x over Finite Field in z2 of size 7^2,
 Elliptic Curve defined by y^2 = x^3 + z2*x over Finite Field in z2 of size 7^2,
 Elliptic Curve defined by y^2 = x^3 + (z2+4)*x over Finite Field in z2 of size 7^
↪2,
 Elliptic Curve defined by y^2 = x^3 + (5*z2+4)*x over Finite Field in z2 of size␣
↪7^2]
```

For $K = \mathbf{F}_q$ where $q \equiv 3 \mod 4$, there are two curves, quadratic twists of each other by -1: $y^2 = x^3 + x$ and $y^2 = x^3 - x$:

```
sage: from sage.schemes.elliptic_curves.ell_finite_field import curves_with_j_1728
sage: curves_with_j_1728(GF(7))
[Elliptic Curve defined by y^2 = x^3 + x over Finite Field of size 7,
 Elliptic Curve defined by y^2 = x^3 + 6*x over Finite Field of size 7]
sage: curves_with_j_1728(GF(11))
[Elliptic Curve defined by y^2 = x^3 + x over Finite Field of size 11,
 Elliptic Curve defined by y^2 = x^3 + 10*x over Finite Field of size 11]
```

sage.schemes.elliptic_curves.ell_finite_field.**fill_ss_j_dict**()

Fill the global cache of supersingular j-_polynomials.

This function does nothing except the first time it is called, when it fills `supersingular_j_polynomials` with precomputed values for $p < 300$. Setting the values this way avoids start-up costs.

sage.schemes.elliptic_curves.ell_finite_field.**is_j_supersingular**(j, *proof*=*True*)

Return True if j is a supersingular j-invariant.

INPUT:

- `j` (finite field element) – an element of a finite field
- `proof` (boolean, default: `True`) – If True, returns a proved result. If False, then a return value of False is certain but a return value of True may be based on a probabilistic test. See the ALGORITHM section below for more details.

OUTPUT:

(boolean) True if j is supersingular, else False.

ALGORITHM:

For small characteristics p we check whether the j-invariant is in a precomputed list of supersingular values. Otherwise we next check the j-invariant. If $j = 0$, the curve is supersingular if and only if $p = 2$ or $p \equiv 3 \pmod 4$; if $j = 1728$, the curve is supersingular if and only if $p = 3$ or $p \equiv 2 \pmod 3$. Next, if the base field is the prime field GF(p), we check that $(p+1)P = 0$ for several random points P, returning False if any fail: supersingular curves over GF(p) have cardinality $p + 1$. If Proof is false we now return True. Otherwise we compute the cardinality and return True if and only if it is divisible by p.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_finite_field import is_j_
↪supersingular, supersingular_j_polynomials
sage: [(p,[j for j in GF(p) if is_j_supersingular(j)]) for p in prime_range(30)]
[(2, [0]), (3, [0]), (5, [0]), (7, [6]), (11, [0, 1]), (13, [5]),
 (17, [0, 8]), (19, [7, 18]), (23, [0, 3, 19]), (29, [0, 2, 25])]

sage: [j for j in GF(109) if is_j_supersingular(j)]
[17, 41, 43]
sage: PolynomialRing(GF(109),'j')(supersingular_j_polynomials[109]).roots()
[(43, 1), (41, 1), (17, 1)]

sage: [p for p in prime_range(100) if is_j_supersingular(GF(p)(0))]
[2, 3, 5, 11, 17, 23, 29, 41, 47, 53, 59, 71, 83, 89]
sage: [p for p in prime_range(100) if is_j_supersingular(GF(p)(1728))]
[2, 3, 7, 11, 19, 23, 31, 43, 47, 59, 67, 71, 79, 83]
sage: [p for p in prime_range(100) if is_j_supersingular(GF(p)(123456))]
[2, 3, 59, 89]
```

`sage.schemes.elliptic_curves.ell_finite_field.`**`special_supersingular_curve`**(*F*, *endomorphism*)

Given a finite field `F`, construct a "special" supersingular elliptic curve E defined over `F`.

Such a curve

- has coefficients in $\mathbb{F}_p$;
- has group structure $E(\mathbb{F}_p) \cong \mathbf{Z}/(p+1)$ and $E(\mathbb{F}_{p^2}) \cong \mathbf{Z}/(p+1) \times \mathbf{Z}/(p+1)$;
- has an endomorphism ϑ of small degree q that anticommutes with the $\mathbb{F}_p$-Frobenius on E.

(The significance of ϑ is that any such endomorphism, together with the $\mathbb{F}_p$-Frobenius, generates the endomorphism algebra $\mathrm{End}(E) \otimes \mathbf{Q}$.)

INPUT:

- `F` – finite field $\mathbb{F}_{p^r}$;
- `endomorphism` – boolean (default: `False`): When set to `True`, it is required that $2 \mid r$, and the function then additionally returns ϑ.

EXAMPLES:

```
sage: special_supersingular_curve(GF(1013^2), endomorphism=True)
(Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field in z2 of size 1013^2,
 Isogeny of degree 3 from Elliptic Curve defined by y^2 = x^3 + 1 over Finite␣
↪Field in z2 of size 1013^2 to Elliptic Curve defined by y^2 = x^3 + 1 over␣
↪Finite Field in z2 of size 1013^2)

sage: special_supersingular_curve(GF(1019^2), endomorphism=True)
(Elliptic Curve defined by y^2 = x^3 + x over Finite Field in z2 of size 1019^2,
 Elliptic-curve endomorphism of Elliptic Curve defined by y^2 = x^3 + x over␣
↪Finite Field in z2 of size 1019^2
   Via:  (u,r,s,t) = (389*z2 + 241, 0, 0, 0))

sage: special_supersingular_curve(GF(1021^2), endomorphism=True)
(Elliptic Curve defined by y^2 = x^3 + 785*x + 794 over Finite Field in z2 of␣
↪size 1021^2,
 Isogeny of degree 2 from Elliptic Curve defined by y^2 = x^3 + 785*x + 794 over␣
↪Finite Field in z2 of size 1021^2 to Elliptic Curve defined by y^2 = x^3 +␣
↪785*x + 794 over Finite Field in z2 of size 1021^2)

sage: special_supersingular_curve(GF(1031^2), endomorphism=True)
(Elliptic Curve defined by y^2 = x^3 + x over Finite Field in z2 of size 1031^2,
 Elliptic-curve endomorphism of Elliptic Curve defined by y^2 = x^3 + x over␣
↪Finite Field in z2 of size 1031^2
   Via:  (u,r,s,t) = (747*z2 + 284, 0, 0, 0))

sage: special_supersingular_curve(GF(1033^2), endomorphism=True)
(Elliptic Curve defined by y^2 = x^3 + 53*x + 980 over Finite Field in z2 of size␣
↪1033^2,
 Isogeny of degree 11 from Elliptic Curve defined by y^2 = x^3 + 53*x + 980 over␣
↪Finite Field in z2 of size 1033^2 to Elliptic Curve defined by y^2 = x^3 + 53*x␣
↪+ 980 over Finite Field in z2 of size 1033^2)

sage: special_supersingular_curve(GF(1039^2), endomorphism=True)
(Elliptic Curve defined by y^2 = x^3 + x over Finite Field in z2 of size 1039^2,
 Elliptic-curve endomorphism of Elliptic Curve defined by y^2 = x^3 + x over␣
↪Finite Field in z2 of size 1039^2
   Via:  (u,r,s,t) = (626*z2 + 200, 0, 0, 0))

sage: special_supersingular_curve(GF(1049^2), endomorphism=True)
(Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field in z2 of size 1049^2,
 Isogeny of degree 3 from Elliptic Curve defined by y^2 = x^3 + 1 over Finite␣
↪Field in z2 of size 1049^2 to Elliptic Curve defined by y^2 = x^3 + 1 over␣
↪Finite Field in z2 of size 1049^2)

sage: special_supersingular_curve(GF(1051^2), endomorphism=True)
(Elliptic Curve defined by y^2 = x^3 + x over Finite Field in z2 of size 1051^2,
 Elliptic-curve endomorphism of Elliptic Curve defined by y^2 = x^3 + x over␣
↪Finite Field in z2 of size 1051^2
   Via:  (u,r,s,t) = (922*z2 + 129, 0, 0, 0))
```

Note: This function makes no guarantees about the distribution of the output. The current implementation is deterministic in many cases.

ALGORITHM: [Bro2009], Algorithm 2.4

sage.schemes.elliptic_curves.ell_finite_field.**supersingular_j_polynomial**(*p*, *use_cache=True*)

Return a polynomial whose roots are the supersingular j-invariants in characteristic p, other than 0, 1728.

INPUT:

- p (integer) – a prime number.
- `use_cache` (boolean, default `True`) – use cached coefficients if they exist

ALGORITHM:

First compute H(X) whose roots are the Legendre λ-invariants of supersingular curves (Silverman V.4.1(b)) in characteristic p. Then, using a resultant computation with the polynomial relating λ and j (Silverman III.1.7(b)), we recover the polynomial (in variable `j`) whose roots are the j-invariants. Factors of j and $j - 1728$ are removed if present.

Note: The only point of the use_cache parameter is to allow checking the precomputed coefficients.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_finite_field import supersingular_j_
↪polynomial
sage: f = supersingular_j_polynomial(67); f
j^5 + 53*j^4 + 4*j^3 + 47*j^2 + 36*j + 8
sage: f.factor()
(j + 1) * (j^2 + 8*j + 45) * (j^2 + 44*j + 24)
```

```
sage: [supersingular_j_polynomial(p) for p in prime_range(30)]
[1, 1, 1, 1, 1, j + 8, j + 9, j + 12, j + 4, j^2 + 2*j + 21]
```

CHAPTER

SEVEN

FORMAL GROUPS OF ELLIPTIC CURVES

AUTHORS:

- William Stein: original implementations
- David Harvey: improved asymptotics of some methods
- Nick Alexander: separation from ell_generic.py, bugfixes and docstrings

class sage.schemes.elliptic_curves.formal_group.**EllipticCurveFormalGroup**(E)

Bases: SageObject

The formal group associated to an elliptic curve.

curve()

Return the elliptic curve this formal group is associated to.

EXAMPLES:

```
sage: E = EllipticCurve("37a")
sage: F = E.formal_group()
sage: F.curve()
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
```

differential(*prec=20*)

Return the power series $f(t) = 1 + \cdots$ such that $f(t)dt$ is the usual invariant differential $dx/(2y + a_1x + a_3)$.

INPUT:

- `prec` – nonnegative integer (default: 20), answer will be returned $O(t^{\text{prec}})$

OUTPUT: a power series with given precision

Return the formal series

$$f(t) = 1 + a_1 t + ({a_1}^2 + a_2)t^2 + \cdots$$

to precision $O(t^{prec})$ of page 113 of [Sil2009].

The result is cached, and a cached version is returned if possible.

Warning: The resulting series will have precision prec, but its parent PowerSeriesRing will have default precision 20 (or whatever the default default is).

EXAMPLES:

```
sage: EllipticCurve([-1, 1/4]).formal_group().differential(15)
 1 - 2*t^4 + 3/4*t^6 + 6*t^8 - 5*t^10 - 305/16*t^12 + 105/4*t^14 + O(t^15)
sage: EllipticCurve(Integers(53), [-1, 1/4]).formal_group().differential(15)
 1 + 51*t^4 + 14*t^6 + 6*t^8 + 48*t^10 + 24*t^12 + 13*t^14 + O(t^15)
```

AUTHORS:

- David Harvey (2006-09-10): factored out of log

group_law (*prec=10*)

Return the formal group law.

INPUT:

- `prec` – integer (default: 10)

OUTPUT: a power series with given precision in $R[[t_1, t_2]]$, where the curve is defined over R.

Return the formal power series

$$F(t_1, t_2) = t_1 + t_2 - a_1 t_1 t_2 - \cdots$$

to precision $O(t_1, t_2)^{prec}$ of page 115 of [Sil2009].

The result is cached, and a cached version is returned if possible.

AUTHORS:

- Nick Alexander: minor fixes, docstring
- Francis Clarke (2012-08): modified to use two-variable power series ring

EXAMPLES:

```
sage: e = EllipticCurve([1, 2])
sage: e.formal_group().group_law(6)
t1 + t2 - 2*t1^4*t2 - 4*t1^3*t2^2 - 4*t1^2*t2^3 - 2*t1*t2^4 + O(t1, t2)^6

sage: e = EllipticCurve('14a1')
sage: ehat = e.formal()
sage: ehat.group_law(3)
t1 + t2 - t1*t2 + O(t1, t2)^3
sage: ehat.group_law(5)
t1 + t2 - t1*t2 - 2*t1^3*t2 - 3*t1^2*t2^2 - 2*t1*t2^3 + O(t1, t2)^5

sage: e = EllipticCurve(GF(7), [3, 4])
sage: ehat = e.formal()
sage: ehat.group_law(3)
t1 + t2 + O(t1, t2)^3
sage: F = ehat.group_law(7); F
t1 + t2 + t1^4*t2 + 2*t1^3*t2^2 + 2*t1^2*t2^3 + t1*t2^4 + O(t1, t2)^7
```

inverse (*prec=20*)

Return the formal group inverse law $i(t)$, which satisfies $F(t, i(t)) = 0$.

INPUT:

- `prec` – integer (default: 20)

OUTPUT: a power series with given precision

Return the formal power series

$$i(t) = -t + a_1 t^2 + \cdots$$

to precision $O(t^{prec})$ of page 114 of [Sil2009].

The result is cached, and a cached version is returned if possible.

Warning: The resulting power series will have precision prec, but its parent PowerSeriesRing will have default precision 20 (or whatever the default default is).

EXAMPLES:

```
sage: P.<a1, a2, a3, a4, a6> = ZZ[]
sage: E = EllipticCurve(list(P.gens()))
sage: i = E.formal_group().inverse(6); i
-t - a1*t^2 - a1^2*t^3 + (-a1^3 - a3)*t^4 + (-a1^4 - 3*a1*a3)*t^5 + O(t^6)
sage: F = E.formal_group().group_law(6)
sage: F(i.parent().gen(), i)
O(t^6)
```

log (*prec=20*)

Return the power series $f(t) = t + \cdots$ which is an isomorphism to the additive formal group.

Generally this only makes sense in characteristic zero, although the terms before t^p may work in characteristic p.

INPUT:

- `prec` – nonnegative integer (default: 20)

OUTPUT: a power series with given precision

EXAMPLES:

```
sage: EllipticCurve([-1, 1/4]).formal_group().log(15)
 t - 2/5*t^5 + 3/28*t^7 + 2/3*t^9 - 5/11*t^11 - 305/208*t^13 + O(t^15)
```

AUTHORS:

- David Harvey (2006-09-10): rewrote to use differential

mult_by_n (*n*, *prec=10*)

Return the formal 'multiplication by n' endomorphism $[n]$.

INPUT:

- `prec` – integer (default: 10)

OUTPUT: a power series with given precision

Return the formal power series

$$[n](t) = nt + \cdots$$

to precision $O(t^{prec})$ of Proposition 2.3 of [Sil2009].

Warning: The resulting power series will have precision prec, but its parent PowerSeriesRing will have default precision 20 (or whatever the default default is).

AUTHORS:

- Nick Alexander: minor fixes, docstring
- David Harvey (2007-03): faster algorithm for char 0 field case
- Hamish Ivey-Law (2009-06): double-and-add algorithm for non char 0 field case.
- Tom Boothby (2009-06): slight improvement to double-and-add
- Francis Clarke (2012-08): adjustments and simplifications using group_law code as modified to yield a two-variable power series.

EXAMPLES:

```
sage: e = EllipticCurve([1, 2, 3, 4, 6])
sage: e.formal_group().mult_by_n(0, 5)
 O(t^5)
sage: e.formal_group().mult_by_n(1, 5)
 t + O(t^5)
```

We verify an identity of low degree:

```
sage: none = e.formal_group().mult_by_n(-1, 5)
sage: two = e.formal_group().mult_by_n(2, 5)
sage: ntwo = e.formal_group().mult_by_n(-2, 5)
sage: ntwo - none(two)
 O(t^5)
sage: ntwo - two(none)
 O(t^5)
```

It's quite fast:

```
sage: E = EllipticCurve("37a"); F = E.formal_group()
sage: F.mult_by_n(100, 20)
100*t - 49999950*t^4 + 3999999960*t^5 + 14285614285800*t^7 -␣
↪2999989920000150*t^8 + 133333325333333400*t^9 - 357137857167499980 0*t^10 +␣
↪1402585362624965454000*t^11 - 14666605706671284799950 0*t^12 +␣
↪5336978000014213190385000*t^13 - 519472790950932256570002000*t^14 +␣
↪9385192768368356727039200280 0*t^15 - 667378721156381236863073032 5175*t^16 +␣
↪3201290603350508750091915249930 00*t^17 -␣
↪4567028886978347847287283321498 6000*t^18 +␣
↪5302464956134111125466184947310391600*t^19 + O(t^20)
```

sigma (*prec=10*)

Return the Weierstrass sigma function as a formal power series solution to the differential equation

$$\frac{d^2 \log \sigma}{dz^2} = -\wp(z)$$

with initial conditions $\sigma(O) = 0$ and $\sigma'(O) = 1$, expressed in the variable $t = \log_E(z)$ of the formal group.

INPUT:

- `prec` – integer (default: 10)

OUTPUT: a power series with given precision

Other solutions can be obtained by multiplication with a function of the form $\exp(cz^2)$. If the curve has good ordinary reduction at a prime p then there is a canonical choice of c that produces the canonical p-adic sigma function. To obtain that, please use `E.padic_sigma(p)` instead. See *padic_sigma()*

EXAMPLES:

```
sage: E = EllipticCurve('14a')
sage: F = E.formal_group()
sage: F.sigma(5)
t + 1/2*t^2 + 1/3*t^3 + 3/4*t^4 + O(t^5)
```

w (*prec=20*)

Return the formal group power series w.

INPUT:

- `prec` – integer (default: 20)

OUTPUT: a power series with given precision

Return the formal power series

$$w(t) = t^3 + a_1 t^4 + (a_2 + a_1^2)t^5 + \cdots$$

to precision $O(t^{prec})$ of Proposition IV.1.1 of [Sil2009]. This is the formal expansion of $w = -1/y$ about the formal parameter $t = -x/y$ at ∞.

The result is cached, and a cached version is returned if possible.

Warning: The resulting power series will have precision prec, but its parent PowerSeriesRing will have default precision 20 (or whatever the default default is).

ALGORITHM: Uses Newton's method to solve the elliptic curve equation at the origin. Complexity is roughly $O(M(n))$ where n is the precision and $M(n)$ is the time required to multiply polynomials of length n over the coefficient ring of E.

AUTHORS:

- David Harvey (2006-09-09): modified to use Newton's method instead of a recurrence formula.

EXAMPLES:

```
sage: e = EllipticCurve([0, 0, 1, -1, 0])
sage: e.formal_group().w(10)
 t^3 + t^6 - t^7 + 2*t^9 + O(t^10)
```

Check that caching works:

```
sage: e = EllipticCurve([3, 2, -4, -2, 5])
sage: e.formal_group().w(20)
 t^3 + 3*t^4 + 11*t^5 + 35*t^6 + 101*t^7 + 237*t^8 + 312*t^9 - 949*t^10 -␣
↪10389*t^11 - 57087*t^12 - 244092*t^13 - 865333*t^14 - 2455206*t^15 -␣
↪4366196*t^16 + 6136610*t^17 + 109938783*t^18 + 688672497*t^19 + O(t^20)
sage: e.formal_group().w(7)
 t^3 + 3*t^4 + 11*t^5 + 35*t^6 + O(t^7)
sage: e.formal_group().w(35)
 t^3 + 3*t^4 + 11*t^5 + 35*t^6 + 101*t^7 + 237*t^8 + 312*t^9 - 949*t^10 -␣
↪10389*t^11 - 57087*t^12 - 244092*t^13 - 865333*t^14 - 2455206*t^15 -␣
↪4366196*t^16 + 6136610*t^17 + 109938783*t^18 + 688672497*t^19 +␣
↪3219525807*t^20 + 12337076504*t^21 + 38106669615*t^22 + 79452618700*t^23 -␣
↪33430470002*t^24 - 1522228110356*t^25 - 10561222329021*t^26 -␣
↪52449326572178*t^27 - 211701726058446*t^28 - 693522772940043*t^29 -␣
↪1613471639599050*t^30 - 4218179064213 78*t^31 + 23651687753515182*t^32 +␣
↪181817896829144595*t^33 + 950887648021211163*t^34 + O(t^35)
```

x (*prec=20*)

Return the formal series $x(t) = t/w(t)$ in terms of the local parameter $t = -x/y$ at infinity.

INPUT:

- `prec` – integer (default: 20)

OUTPUT: a Laurent series with given precision

Return the formal series

$$x(t) = t^{-2} - a_1 t^{-1} - a_2 - a_3 t - \cdots$$

to precision $O(t^{prec})$ of page 113 of [Sil2009].

Warning: The resulting series will have precision prec, but its parent PowerSeriesRing will have default precision 20 (or whatever the default default is).

EXAMPLES:

```
sage: EllipticCurve([0, 0, 1, -1, 0]).formal_group().x(10)
 t^-2 - t + t^2 - t^4 + 2*t^5 - t^6 - 2*t^7 + 6*t^8 - 6*t^9 + O(t^10)
```

y (*prec=20*)

Return the formal series $y(t) = -1/w(t)$ in terms of the local parameter $t = -x/y$ at infinity.

INPUT:

- `prec` – integer (default: 20)

OUTPUT: a Laurent series with given precision

Return the formal series

$$y(t) = -t^{-3} + a_1 t^{-2} + a_2 t + a_3 + \cdots$$

to precision $O(t^{prec})$ of page 113 of [Sil2009].

The result is cached, and a cached version is returned if possible.

Warning: The resulting series will have precision prec, but its parent PowerSeriesRing will have default precision 20 (or whatever the default default is).

EXAMPLES:

```
sage: EllipticCurve([0, 0, 1, -1, 0]).formal_group().y(10)
 -t^-3 + 1 - t + t^3 - 2*t^4 + t^5 + 2*t^6 - 6*t^7 + 6*t^8 + 3*t^9 + O(t^10)
```

Maps between them

CHAPTER

EIGHT

ELLIPTIC-CURVE MORPHISMS

This class serves as a common parent for various specializations of morphisms between elliptic curves, with the aim of providing a common interface regardless of implementation details.

Current implementations of elliptic-curve morphisms (child classes):

- *EllipticCurveIsogeny*
- *WeierstrassIsomorphism*
- *EllipticCurveHom_composite*
- EllipticCurveHom_sum
- *EllipticCurveHom_scalar*
- *EllipticCurveHom_frobenius*
- *EllipticCurveHom_velusqrt*

AUTHORS:

- See authors of *EllipticCurveIsogeny*. Some of the code in this class was lifted from there.
- Lorenz Panny (2021): Refactor isogenies and isomorphisms into the common *EllipticCurveHom* interface.
- Lorenz Panny (2022): *matrix_on_subgroup()*
- Lorenz Panny (2023): *trace()*, *characteristic_polynomial()*

class sage.schemes.elliptic_curves.hom.**EllipticCurveHom**(**args*, ***kwds*)

Bases: Morphism

Base class for elliptic-curve morphisms.

as_morphism()

Return self as a morphism of projective schemes.

EXAMPLES:

```
sage: k = GF(11)
sage: E = EllipticCurve(k, [1,1])
sage: Q = E(6,5)
sage: phi = E.isogeny(Q)
sage: mor = phi.as_morphism()
sage: mor.domain() == E
True
sage: mor.codomain() == phi.codomain()
True
sage: mor(Q) == phi(Q)
True
```

characteristic_polynomial()

Return the characteristic polynomial of this elliptic-curve morphism, which must be an endomorphism.

See also:

- *degree()*
- *trace()*

EXAMPLES:

```
sage: E = EllipticCurve(QQ, [42, 42])
sage: m5 = E.scalar_multiplication(5)
sage: m5.characteristic_polynomial()
x^2 - 10*x + 25
```

```
sage: E = EllipticCurve(GF(71), [42, 42])
sage: pi = E.frobenius_endomorphism()
sage: pi.characteristic_polynomial()
x^2 - 8*x + 71
sage: E.frobenius().charpoly()
x^2 - 8*x + 71
```

degree()

Return the degree of this elliptic-curve morphism.

EXAMPLES:

```
sage: E = EllipticCurve(QQ, [0,0,0,1,0])
sage: phi = EllipticCurveIsogeny(E, E((0,0)))
sage: phi.degree()
2
sage: phi = EllipticCurveIsogeny(E, [0,1,0,1])
sage: phi.degree()
4

sage: E = EllipticCurve(GF(31), [1,0,0,1,2])
sage: phi = EllipticCurveIsogeny(E, [17, 1])
sage: phi.degree()
3
```

Degrees are multiplicative, so the degree of a composite isogeny is the product of the degrees of the individual factors:

```
sage: from sage.schemes.elliptic_curves.hom_composite import EllipticCurveHom_
↪composite
sage: E = EllipticCurve(GF(419), [1,0])
sage: P, = E.gens()
sage: phi = EllipticCurveHom_composite(E, P+P)
sage: phi.degree()
210
sage: phi.degree() == prod(f.degree() for f in phi.factors())
True
```

Isomorphisms always have degree 1 by definition:

```
sage: E1 = EllipticCurve([1,2,3,4,5])
sage: E2 = EllipticCurve_from_j(E1.j_invariant())
sage: E1.isomorphism_to(E2).degree()
1
```

dual()

Return the dual of this elliptic-curve morphism.

Implemented by child classes. For examples, see:

- *EllipticCurveIsogeny.dual()*
- *sage.schemes.elliptic_curves.weierstrass_morphism.WeierstrassIsomorphism.dual()*
- *sage.schemes.elliptic_curves.hom_composite.EllipticCurveHom_composite.dual()*
- *sage.schemes.elliptic_curves.hom_sum.EllipticCurveHom_sum.dual()*
- *sage.schemes.elliptic_curves.hom_scalar.EllipticCurveHom_scalar.dual()*
- *sage.schemes.elliptic_curves.hom_frobenius.EllipticCurveHom_frobenius.dual()*

formal(*prec=20*)

Return the formal isogeny associated to this elliptic-curve morphism as a power series in the variable $t = -x/y$ on the domain curve.

INPUT:

- `prec` – (default: 20), the precision with which the computations in the formal group are carried out.

EXAMPLES:

```
sage: E = EllipticCurve(GF(13),[1,7])
sage: phi = E.isogeny(E(10,4))
sage: phi.formal()
t + 12*t^13 + 2*t^17 + 8*t^19 + 2*t^21 + O(t^23)
```

```
sage: E = EllipticCurve([0,1])
sage: phi = E.isogeny(E(2,3))
sage: phi.formal(prec=10)
t + 54*t^5 + 255*t^7 + 2430*t^9 + 19278*t^11 + O(t^13)
```

```
sage: E = EllipticCurve('11a2')
sage: R.<x> = QQ[]
sage: phi = E.isogeny(x^2 + 101*x + 12751/5)
sage: phi.formal(prec=7)
t - 2724/5*t^5 + 209046/5*t^7 - 4767/5*t^8 + 29200946/5*t^9 + O(t^10)
```

inseparable_degree()

Return the inseparable degree of this isogeny.

Implemented by child classes. For examples, see:

- *EllipticCurveIsogeny.inseparable_degree()*
- *sage.schemes.elliptic_curves.weierstrass_morphism.WeierstrassIsomorphism.inseparable_degree()*

- *sage.schemes.elliptic_curves.hom_composite.EllipticCurveHom_composite.inseparable_degree()*
- *sage.schemes.elliptic_curves.hom_sum.EllipticCurveHom_sum.inseparable_degree()*
- *sage.schemes.elliptic_curves.hom_scalar.EllipticCurveHom_scalar.inseparable_degree()*
- *sage.schemes.elliptic_curves.hom_frobenius.EllipticCurveHom_frobenius.inseparable_degree()*

is_injective()

Determine whether or not this morphism has trivial kernel.

The kernel is trivial if and only if this morphism is a purely inseparable isogeny.

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: R.<x> = QQ[]
sage: f = x^2 + x - 29/5
sage: phi = EllipticCurveIsogeny(E, f)
sage: phi.is_injective()
False
sage: phi = EllipticCurveIsogeny(E, R(1))
sage: phi.is_injective()
True
```

```
sage: F = GF(7)
sage: E = EllipticCurve(j=F(0))
sage: phi = EllipticCurveIsogeny(E, [ E((0,-1)), E((0,1))])
sage: phi.is_injective()
False
sage: phi = EllipticCurveIsogeny(E, E(0))
sage: phi.is_injective()
True
```

```
sage: from sage.schemes.elliptic_curves.hom_composite import EllipticCurveHom_
↪composite
sage: E = EllipticCurve([1,0])
sage: phi = EllipticCurveHom_composite(E, E(0,0))
sage: phi.is_injective()
False
sage: E = EllipticCurve_from_j(GF(3).algebraic_closure()(0))
sage: nu = EllipticCurveHom_composite.from_factors(E.automorphisms())
sage: nu
Composite morphism of degree 1 = 1^12:
  From: Elliptic Curve defined by y^2 = x^3 + x
        over Algebraic closure of Finite Field of size 3
  To:   Elliptic Curve defined by y^2 = x^3 + x
        over Algebraic closure of Finite Field of size 3
sage: nu.is_injective()
True
```

```
sage: E = EllipticCurve(GF(23), [1,0])
sage: E.scalar_multiplication(4).is_injective()
False
```

(continues on next page)

(continued from previous page)

```
sage: E.scalar_multiplication(5).is_injective()
False
sage: E.scalar_multiplication(1).is_injective()
True
sage: E.scalar_multiplication(-1).is_injective()
True
sage: E.scalar_multiplication(23).is_injective()
True
sage: E.scalar_multiplication(-23).is_injective()
True
sage: E.scalar_multiplication(0).is_injective()
False
```

```
sage: from sage.schemes.elliptic_curves.hom_frobenius import EllipticCurveHom_
↪frobenius
sage: E = EllipticCurve(GF(11), [1,1])
sage: pi = EllipticCurveHom_frobenius(E, 5)
sage: pi.is_injective()
True
```

is_normalized()

Determine whether this morphism is a normalized isogeny.

Note: An isogeny $\varphi\colon E_1 \to E_2$ between two given Weierstrass equations is said to be *normalized* if the $\varphi^*(\omega_2) = \omega_1$, where ω_1 and ω_2 are the invariant differentials on E_1 and E_2 corresponding to the given equation.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.weierstrass_morphism import␣
↪WeierstrassIsomorphism
sage: E = EllipticCurve(GF(7), [0,0,0,1,0])
sage: R.<x> = GF(7)[]
sage: phi = EllipticCurveIsogeny(E, x)
sage: phi.is_normalized()
True
sage: isom = WeierstrassIsomorphism(phi.codomain(), (3, 0, 0, 0))
sage: phi = isom * phi
sage: phi.is_normalized()
False
sage: isom = WeierstrassIsomorphism(phi.codomain(), (5, 0, 0, 0))
sage: phi = isom * phi
sage: phi.is_normalized()
True
sage: isom = WeierstrassIsomorphism(phi.codomain(), (1, 1, 1, 1))
sage: phi = isom * phi
sage: phi.is_normalized()
True
```

```
sage: F = GF(2^5, 'alpha'); alpha = F.gen()
sage: E = EllipticCurve(F, [1,0,1,1,1])
sage: R.<x> = F[]
sage: phi = EllipticCurveIsogeny(E, x+1)
sage: isom = WeierstrassIsomorphism(phi.codomain(), (alpha, 0, 0, 0))
```

(continues on next page)

(continued from previous page)

```
sage: phi.is_normalized()
True
sage: phi = isom * phi
sage: phi.is_normalized()
False
sage: isom = WeierstrassIsomorphism(phi.codomain(), (1/alpha, 0, 0, 0))
sage: phi = isom * phi
sage: phi.is_normalized()
True
sage: isom = WeierstrassIsomorphism(phi.codomain(), (1, 1, 1, 1))
sage: phi = isom * phi
sage: phi.is_normalized()
True
```

```
sage: E = EllipticCurve('11a1')
sage: R.<x> = QQ[]
sage: f = x^3 - x^2 - 10*x - 79/4
sage: phi = EllipticCurveIsogeny(E, f)
sage: isom = WeierstrassIsomorphism(phi.codomain(), (2, 0, 0, 0))
sage: phi.is_normalized()
True
sage: phi = isom * phi
sage: phi.is_normalized()
False
sage: isom = WeierstrassIsomorphism(phi.codomain(), (1/2, 0, 0, 0))
sage: phi = isom * phi
sage: phi.is_normalized()
True
sage: isom = WeierstrassIsomorphism(phi.codomain(), (1, 1, 1, 1))
sage: phi = isom * phi
sage: phi.is_normalized()
True
```

ALGORITHM: We check if *scaling_factor()* returns 1.

is_separable()

Determine whether or not this morphism is a separable isogeny.

EXAMPLES:

```
sage: E = EllipticCurve(GF(17), [0,0,0,3,0])
sage: phi = EllipticCurveIsogeny(E,  E((0,0)))
sage: phi.is_separable()
True
```

```
sage: E = EllipticCurve('11a1')
sage: phi = EllipticCurveIsogeny(E, E.torsion_points())
sage: phi.is_separable()
True
```

```
sage: E = EllipticCurve(GF(31337), [0,1])                                    #␣
↪needs sage.rings.finite_rings
sage: {f.is_separable() for f in E.automorphisms()}                          #␣
↪needs sage.rings.finite_rings
{True}
```

```
sage: # needs sage.rings.finite_rings
sage: from sage.schemes.elliptic_curves.hom_composite import EllipticCurveHom_
↪composite
sage: E = EllipticCurve(GF(7^2), [3,2])
sage: P = E.lift_x(1)
sage: phi = EllipticCurveHom_composite(E, P); phi
Composite morphism of degree 7:
  From: Elliptic Curve defined by y^2 = x^3 + 3*x + 2 over Finite Field in z2␣
↪of size 7^2
  To:   Elliptic Curve defined by y^2 = x^3 + 3*x + 2 over Finite Field in z2␣
↪of size 7^2
sage: phi.is_separable()
True
```

```
sage: E = EllipticCurve(GF(11), [4,4])
sage: E.scalar_multiplication(11).is_separable()
False
sage: E.scalar_multiplication(-11).is_separable()
False
sage: E.scalar_multiplication(777).is_separable()
True
sage: E.scalar_multiplication(-1).is_separable()
True
sage: E.scalar_multiplication(77).is_separable()
False
sage: E.scalar_multiplication(121).is_separable()
False
```

```
sage: from sage.schemes.elliptic_curves.hom_frobenius import EllipticCurveHom_
↪frobenius
sage: E = EllipticCurve(GF(11), [1,1])
sage: pi = EllipticCurveHom_frobenius(E)
sage: pi.degree()
11
sage: pi.is_separable()
False
sage: pi = EllipticCurveHom_frobenius(E, 0)
sage: pi.degree()
1
sage: pi.is_separable()
True
```

```
sage: E = EllipticCurve(GF(17), [0,0,0,3,0])
sage: phi = E.isogeny(E((1,2)), algorithm='velusqrt')
sage: phi.is_separable()
True
```

is_surjective()

Determine whether or not this morphism is surjective.

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: R.<x> = QQ[]
sage: f = x^2 + x - 29/5
sage: phi = EllipticCurveIsogeny(E, f)
```

(continues on next page)

(continued from previous page)

```
sage: phi.is_surjective()
True
```

```
sage: E = EllipticCurve(GF(7), [0,0,0,1,0])
sage: phi = EllipticCurveIsogeny(E,  E((0,0)))
sage: phi.is_surjective()
True
```

```
sage: F = GF(2^5, 'omega')
sage: E = EllipticCurve(j=F(0))
sage: R.<x> = F[]
sage: phi = EllipticCurveIsogeny(E, x)
sage: phi.is_surjective()
True
```

is_zero()

Check whether this elliptic-curve morphism is the zero map.

EXAMPLES:

```
sage: E = EllipticCurve(j=GF(7)(0))
sage: phi = EllipticCurveIsogeny(E, [E(0,1), E(0,-1)])
sage: phi.is_zero()
False
```

kernel_polynomial()

Return the kernel polynomial of this elliptic-curve morphism.

Implemented by child classes. For examples, see:

- *EllipticCurveIsogeny.kernel_polynomial()*
- *sage.schemes.elliptic_curves.weierstrass_morphism.WeierstrassIsomorphism.kernel_polynomial()*
- *sage.schemes.elliptic_curves.hom_composite.EllipticCurveHom_composite.kernel_polynomial()*
- *sage.schemes.elliptic_curves.hom_sum.EllipticCurveHom_sum.kernel_polynomial()*
- *sage.schemes.elliptic_curves.hom_scalar.EllipticCurveHom_scalar.kernel_polynomial()*
- *sage.schemes.elliptic_curves.hom_frobenius.EllipticCurveHom_frobenius.kernel_polynomial()*

matrix_on_subgroup(*domain_gens*, *codomain_gens=None*)

Return the matrix by which this isogeny acts on the n-torsion subgroup with respect to the given bases.

INPUT:

- `domain_gens` – basis (P, Q) of some n-torsion subgroup on the domain of this elliptic-curve morphism
- `codomain_gens` – basis (R, S) of the n-torsion on the codomain of this morphism, or (default) `None` if `self` is an endomorphism

OUTPUT:

A 2×2 matrix M over $\mathbf{Z}/n$, such that the image of any point $[a]P + [b]Q$ under this morphism equals $[c]R + [d]S$ where $(c\ d)^T = (a\ b)M$.

EXAMPLES:

```
sage: F.<i> = GF(419^2, modulus=[1,0,1])
sage: E = EllipticCurve(F, [1,0])
sage: P = E(3, 176*i)
sage: Q = E(i+7, 67*i+48)
sage: P.weil_pairing(Q, 420).multiplicative_order()
420
sage: iota = E.automorphisms()[2]; iota
Elliptic-curve endomorphism of Elliptic Curve defined by y^2 = x^3 + x over␣
↪Finite Field in i of size 419^2
  Via:  (u,r,s,t) = (i, 0, 0, 0)
sage: iota^2 == E.scalar_multiplication(-1)
True
sage: mat = iota.matrix_on_subgroup((P,Q)); mat
[301 386]
[ 83 119]
sage: mat.parent()
Full MatrixSpace of 2 by 2 dense matrices over Ring of integers modulo 420
sage: iota(P) == 301*P + 386*Q
True
sage: iota(Q) == 83*P + 119*Q
True
sage: a,b = 123, 456
sage: c,d = vector((a,b)) * mat; (c,d)
(111, 102)
sage: iota(a*P + b*Q) == c*P + d*Q
True
```

One important application of this is to compute generators of the kernel subgroup of an isogeny, when the n-torsion subgroup containing the kernel is accessible:

```
sage: K = E(83*i-16, 9*i-147)
sage: K.order()
7
sage: phi = E.isogeny(K)
sage: R,S = phi.codomain().gens()
sage: mat = phi.matrix_on_subgroup((P,Q), (R,S))
sage: mat  # random -- depends on R,S
[124 263]
[115 141]
sage: kermat = mat.left_kernel_matrix(); kermat
[300  60]
sage: ker = [ZZ(v[0])*P + ZZ(v[1])*Q for v in kermat]
sage: {phi(T) for T in ker}
{(0 : 1 : 0)}
sage: phi == E.isogeny(ker)
True
```

We can also compute the matrix of a Frobenius endomorphism (*EllipticCurveHom_frobenius*) on a large enough subgroup to verify point-counting results:

```
sage: F.<a> = GF((101, 36))
```

(continues on next page)

(continued from previous page)

```
sage: E = EllipticCurve(GF(101), [1,1])
sage: EE = E.change_ring(F)
sage: P,Q = EE.torsion_basis(37)
sage: pi = EE.frobenius_isogeny()
sage: M = pi.matrix_on_subgroup((P,Q))
sage: M.parent()
Full MatrixSpace of 2 by 2 dense matrices over Ring of integers modulo 37
sage: M.trace()
34
sage: E.trace_of_frobenius()
-3
```

See also:

To compute a basis of the n-torsion, you may use *torsion_basis()*.

rational_maps()

Return the pair of explicit rational maps defining this elliptic-curve morphism as fractions of bivariate polynomials in x and y.

Implemented by child classes. For examples, see:

- *EllipticCurveIsogeny.rational_maps()*
- *sage.schemes.elliptic_curves.weierstrass_morphism.WeierstrassIsomorphism.rational_maps()*
- *sage.schemes.elliptic_curves.hom_composite.EllipticCurveHom_composite.rational_maps()*
- *sage.schemes.elliptic_curves.hom_sum.EllipticCurveHom_sum.rational_maps()*
- *sage.schemes.elliptic_curves.hom_scalar.EllipticCurveHom_scalar.rational_maps()*
- *sage.schemes.elliptic_curves.hom_frobenius.EllipticCurveHom_frobenius.rational_maps()*

scaling_factor()

Return the Weierstrass scaling factor associated to this elliptic-curve morphism.

The scaling factor is the constant u (in the base field) such that $\varphi^*\omega_2 = u\omega_1$, where $\varphi : E_1 \to E_2$ is this morphism and ω_i are the standard Weierstrass differentials on E_i defined by $\mathrm{d}x/(2y + a_1x + a_3)$.

Implemented by child classes. For examples, see:

- *EllipticCurveIsogeny.scaling_factor()*
- *sage.schemes.elliptic_curves.weierstrass_morphism.WeierstrassIsomorphism.scaling_factor()*
- *sage.schemes.elliptic_curves.hom_composite.EllipticCurveHom_composite.scaling_factor()*
- *sage.schemes.elliptic_curves.hom_sum.EllipticCurveHom_sum.scaling_factor()*
- *sage.schemes.elliptic_curves.hom_scalar.EllipticCurveHom_scalar.scaling_factor()*

separable_degree()

Return the separable degree of this isogeny.

The separable degree is the result of dividing the *degree()* by the *inseparable_degree()*.

EXAMPLES:

```
sage: E = EllipticCurve(GF(11), [5,5])
sage: E.is_supersingular()
False
sage: E.scalar_multiplication(-77).separable_degree()
539
sage: E = EllipticCurve(GF(11), [5,0])
sage: E.is_supersingular()
True
sage: E.scalar_multiplication(-77).separable_degree()
49
```

trace()

Return the trace of this elliptic-curve morphism, which must be an endomorphism.

ALGORITHM: *compute_trace_generic()*

EXAMPLES:

```
sage: E = EllipticCurve(QQ, [42, 42])
sage: m5 = E.scalar_multiplication(5)
sage: m5.trace()
10
```

```
sage: E = EllipticCurve(GF(71^2), [45, 45])
sage: P = E.lift_x(27)
sage: P.order()
71
sage: tau = E.isogeny(P, codomain=E)
sage: tau.trace()
-1
```

x_rational_map()

Return the x-coordinate rational map of this elliptic-curve morphism as a univariate rational expression in x.

Implemented by child classes. For examples, see:

- *EllipticCurveIsogeny.x_rational_map()*
- *sage.schemes.elliptic_curves.weierstrass_morphism.WeierstrassIsomorphism.x_rational_map()*
- *sage.schemes.elliptic_curves.hom_composite.EllipticCurveHom_composite.x_rational_map()*
- *sage.schemes.elliptic_curves.hom_sum.EllipticCurveHom_sum.x_rational_map()*
- *sage.schemes.elliptic_curves.hom_scalar.EllipticCurveHom_scalar.x_rational_map()*
- *sage.schemes.elliptic_curves.hom_frobenius.EllipticCurveHom_frobenius.x_rational_map()*

sage.schemes.elliptic_curves.hom.**compare_via_evaluation**(*left*, *right*)

Test if two elliptic-curve morphisms are equal by evaluating them at enough points.

INPUT:

- left, right – *EllipticCurveHom* objects

ALGORITHM:

We use the fact that two isogenies of equal degree d must be the same if and only if they behave identically on more than $4d$ points. (It suffices to check this on a few points that generate a large enough subgroup.)

If the domain curve does not have sufficiently many rational points, the base field is extended first: Taking an extension of degree $O(\log(d))$ suffices.

EXAMPLES:

```
sage: E = EllipticCurve(GF(83), [1,0])
sage: phi = E.isogeny(12*E.0, model='montgomery'); phi
Isogeny of degree 7 from Elliptic Curve defined by y^2 = x^3 + x over Finite␣
↪Field of size 83 to Elliptic Curve defined by y^2 = x^3 + 70*x^2 + x over␣
↪Finite Field of size 83
sage: psi = phi.dual(); psi
Isogeny of degree 7 from Elliptic Curve defined by y^2 = x^3 + 70*x^2 + x over␣
↪Finite Field of size 83 to Elliptic Curve defined by y^2 = x^3 + x over Finite␣
↪Field of size 83
sage: from sage.schemes.elliptic_curves.hom_composite import EllipticCurveHom_
↪composite
sage: mu = EllipticCurveHom_composite.from_factors([phi, psi])
sage: from sage.schemes.elliptic_curves.hom import compare_via_evaluation
sage: compare_via_evaluation(mu, E.scalar_multiplication(7))
True
```

See also:

- sage.schemes.elliptic_curves.hom_composite.EllipticCurveHom_composite._richcmp_()

sage.schemes.elliptic_curves.hom.**compute_trace_generic**(*phi*)

Compute the trace of the given elliptic-curve endomorphism.

ALGORITHM: Simple variant of Schoof's algorithm. For enough small primes ℓ, we find an order-ℓ point P on E and use a discrete-logarithm calculation to find the unique scalar $t_\ell \in \{0, ..., \ell-1\}$ such that $\varphi^2(P)+[\deg(\varphi)]P = [t_\ell]\varphi(P)$. Then t_ℓ equals the trace of φ modulo ℓ, which can therefore be recovered using the Chinese remainder theorem.

EXAMPLES:

It works over finite fields:

```
sage: from sage.schemes.elliptic_curves.hom import compute_trace_generic
sage: E = EllipticCurve(GF(31337), [1,1])
sage: compute_trace_generic(E.frobenius_endomorphism())
314
```

It works over **Q**:

```
sage: from sage.schemes.elliptic_curves.hom import compute_trace_generic
sage: E = EllipticCurve(QQ, [1,2,3,4,5])
sage: dbl = E.scalar_multiplication(2)
```

(continues on next page)

(continued from previous page)

```
sage: compute_trace_generic(dbl)
4
```

It works over number fields (for a CM curve):

```
sage: from sage.schemes.elliptic_curves.hom import compute_trace_generic
sage: x = polygen(QQ)
sage: K.<t> = NumberField(5*x^2 - 2*x + 1)
sage: E = EllipticCurve(K, [1,0])
sage: phi = E.isogeny([t,0,1], codomain=E)  # phi = 2 + i
sage: compute_trace_generic(phi)
4
```

sage.schemes.elliptic_curves.hom.**find_post_isomorphism**(*phi*, *psi*)

Given two isogenies $\phi : E \to E'$ and $\psi : E \to E''$ which are equal up to post-isomorphism defined over the same field, find that isomorphism.

In other words, this function computes an isomorphism $\alpha : E' \to E''$ such that $\alpha \circ \phi = \psi$.

ALGORITHM:

Start with a list of all isomorphisms $E' \to E''$. Then repeatedly evaluate ϕ and ψ at random points P to filter the list for isomorphisms α with $\alpha(\phi(P)) = \psi(P)$. Once only one candidate is left, return it. Periodically extend the base field to avoid getting stuck (say, if all candidate isomorphisms act the same on all rational points).

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.hom import find_post_isomorphism
sage: E = EllipticCurve(GF(7^2), [1,0])
sage: f = E.scalar_multiplication(1)
sage: g = choice(E.automorphisms())
sage: find_post_isomorphism(f, g) == g
True
```

```
sage: from sage.schemes.elliptic_curves.weierstrass_morphism import␣
↪WeierstrassIsomorphism
sage: from sage.schemes.elliptic_curves.hom_composite import EllipticCurveHom_
↪composite
sage: x = polygen(ZZ, 'x')
sage: F.<i> = GF(883^2, modulus=x^2+1)
sage: E = EllipticCurve(F, [1,0])
sage: P = E.lift_x(117)
sage: Q = E.lift_x(774)
sage: w = WeierstrassIsomorphism(E, [i,0,0,0])
sage: phi = EllipticCurveHom_composite(E, [P,w(Q)]) * w
sage: psi = EllipticCurveHom_composite(E, [Q,w(P)])
sage: phi.kernel_polynomial() == psi.kernel_polynomial()
True
sage: find_post_isomorphism(phi, psi)
Elliptic-curve morphism:
  From: Elliptic Curve defined by y^2 = x^3 + 320*x + 482 over Finite Field in i␣
↪of size 883^2
  To:   Elliptic Curve defined by y^2 = x^3 + 320*x + 401 over Finite Field in i␣
↪of size 883^2
  Via:  (u,r,s,t) = (882*i, 0, 0, 0)
```

CHAPTER

NINE

COMPOSITE MORPHISMS OF ELLIPTIC CURVES

It is often computationally convenient (for example, in cryptography) to factor an isogeny of large degree into a composition of isogenies of smaller (prime) degree. This class implements such a decomposition while exposing (close to) the same interface as "normal", unfactored elliptic-curve isogenies.

EXAMPLES:

The following example would take quite literally forever with the straightforward *EllipticCurveIsogeny* implementation, but decomposing into prime steps is exponentially faster:

```
sage: # needs sage.rings.finite_rings
sage: from sage.schemes.elliptic_curves.hom_composite import EllipticCurveHom_
↪composite
sage: p = 3 * 2^143 - 1
sage: GF(p^2).inject_variables()
Defining z2
sage: E = EllipticCurve(GF(p^2), [1,0])
sage: P = E.lift_x(31415926535897932384626433832795028841971 - z2)
sage: P.order().factor()
2^143
sage: EllipticCurveHom_composite(E, P)
Composite morphism of degree 11150372599265311570767859136324180752990208 = 2^143:
  From: Elliptic Curve defined by y^2 = x^3 + x
        over Finite Field in z2 of size 33451117797795934712303577408972542258970623^2
  To:   Elliptic Curve defined by y^2 = x^3 +␣
↪(18676616716352953484576727486205473216172067*z2+32690199585974925193292786311814241821808308)*x
        +␣
↪(33697024363513674039100788775919463002019 03*z2+152275586156990412418519786050027046266897 22)
        over Finite Field in z2 of size 33451117797795934712303577408972542258970623^2
```

Yet, the interface provided by *EllipticCurveHom_composite* is identical to *EllipticCurveIsogeny* and other instantiations of *EllipticCurveHom*:

```
sage: # needs sage.rings.finite_rings
sage: E = EllipticCurve(GF(419), [0,1])
sage: P = E.lift_x(33); P.order()
35
sage: psi = EllipticCurveHom_composite(E, P); psi
Composite morphism of degree 35 = 5*7:
  From: Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field of size 419
  To:   Elliptic Curve defined by y^2 = x^3 + 101*x + 285 over Finite Field of size␣
↪419
sage: psi(E.lift_x(11))
(352 : 346 : 1)
sage: psi.rational_maps()
```

(continues on next page)

(continued from previous page)

```
((x^35 + 162*x^34 + 186*x^33 + 92*x^32 - ... + 44*x^3 + 190*x^2 + 80*x
  - 72)/(x^34 + 162*x^33 - 129*x^32 + 41*x^31 + ... + 66*x^3 - 191*x^2 + 119*x + 21),
 (x^51*y - 176*x^50*y + 115*x^49*y - 120*x^48*y + ... + 72*x^3*y + 129*x^2*y + 163*x*y
  + 178*y)/(x^51 - 176*x^50 + 11*x^49 + 26*x^48 - ... - 77*x^3 + 185*x^2 + 169*x -␣
↪128))
sage: psi.kernel_polynomial()
x^17 + 81*x^16 + 7*x^15 + 82*x^14 + 49*x^13 + 68*x^12 + 109*x^11 + 326*x^10
 + 117*x^9 + 136*x^8 + 111*x^7 + 292*x^6 + 55*x^5 + 389*x^4 + 175*x^3 + 43*x^2 +␣
↪149*x + 373
sage: psi.dual()
Composite morphism of degree 35 = 7*5:
  From: Elliptic Curve defined by y^2 = x^3 + 101*x + 285 over Finite Field of size␣
↪419
  To:   Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field of size 419
sage: psi.formal()
t + 211*t^5 + 417*t^7 + 159*t^9 + 360*t^11 + 259*t^13 + 224*t^15 + 296*t^17 + 139*t^
↪19 + 222*t^21 + O(t^23)
```

Equality is decided correctly (and, in some cases, much faster than comparing *EllipticCurveHom.rational_maps()*) even when distinct factorizations of the same isogeny are compared:

```
sage: psi == EllipticCurveIsogeny(E, P)                                    #␣
↪needs sage.rings.finite_rings
True
```

We can easily obtain the individual factors of the composite map:

```
sage: psi.factors()                                                        #␣
↪needs sage.rings.finite_rings
(Isogeny of degree 5
  from Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field of size 419
    to Elliptic Curve defined by y^2 = x^3 + 140*x + 214 over Finite Field of size␣
↪419,
 Isogeny of degree 7
  from Elliptic Curve defined by y^2 = x^3 + 140*x + 214 over Finite Field of size 419
    to Elliptic Curve defined by y^2 = x^3 + 101*x + 285 over Finite Field of size␣
↪419)
```

AUTHORS:

- Lukas Zobernig (2020): initial proof-of-concept version
- Lorenz Panny (2021): *EllipticCurveHom* interface, documentation and tests, equality testing

class sage.schemes.elliptic_curves.hom_composite.**EllipticCurveHom_composite**(*E*, *kernel*, *codomain=None*, *model=None*, *velu_sqrt_bound=Non*

Bases: *EllipticCurveHom*

Construct a composite isogeny with given kernel (and optionally, prescribed codomain curve). The isogeny is decomposed into steps of prime degree.

The `codomain` and `model` parameters have the same meaning as for *EllipticCurveIsogeny*.

The optional parameter `velu_sqrt_bound` prescribes the point in which the computation of a single isogeny

should be performed using square root Velu instead of simple Velu. If not provided, the system default is used (see `EllipticCurve_field.isogeny` for a more detailed discussion.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.hom_composite import EllipticCurveHom_
↪composite
sage: E = EllipticCurve(GF(419), [1,0])                                    #␣
↪needs sage.rings.finite_rings
sage: EllipticCurveHom_composite(E, E.lift_x(23))                          #␣
↪needs sage.rings.finite_rings
Composite morphism of degree 105 = 3*5*7:
  From: Elliptic Curve defined by y^2 = x^3 + x
        over Finite Field of size 419
  To:   Elliptic Curve defined by y^2 = x^3 + 373*x + 126
        over Finite Field of size 419
```

The given kernel generators need not be independent:

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^2 - x - 5)
sage: E = EllipticCurve('210.b6').change_ring(K)
sage: E.torsion_subgroup()
Torsion Subgroup isomorphic to Z/12 + Z/2 associated to the Elliptic Curve
 defined by y^2 + x*y + y = x^3 + (-578)*x + 2756
  over Number Field in a with defining polynomial x^2 - x - 5
sage: EllipticCurveHom_composite(E, E.torsion_points())
Composite morphism of degree 24 = 2^3*3:
  From: Elliptic Curve defined by y^2 + x*y + y = x^3 + (-578)*x + 2756
        over Number Field in a with defining polynomial x^2 - x - 5
  To:   Elliptic Curve defined by
        y^2 + x*y + y = x^3 + (-89915533/16)*x + (-328200928141/64)
        over Number Field in a with defining polynomial x^2 - x - 5
```

dual()

Return the dual of this composite isogeny.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: from sage.schemes.elliptic_curves.hom_composite import EllipticCurveHom_
↪composite
sage: E = EllipticCurve(GF(65537), [1,2,3,4,5])
sage: P = E.lift_x(7321)
sage: phi = EllipticCurveHom_composite(E, P); phi
Composite morphism of degree 9 = 3^2:
  From: Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5
        over Finite Field of size 65537
  To:   Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 28339*x +␣
↪59518
        over Finite Field of size 65537
sage: psi = phi.dual(); psi
Composite morphism of degree 9 = 3^2:
  From: Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 28339*x +␣
↪59518
        over Finite Field of size 65537
  To:   Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5
```

(continues on next page)

(continued from previous page)

```
        over Finite Field of size 65537
sage: psi * phi == phi.domain().scalar_multiplication(phi.degree())
True
sage: phi * psi == psi.domain().scalar_multiplication(psi.degree())
True
```

factors()

Return the factors of this composite isogeny as a tuple.

The isogenies are returned in left-to-right order, i.e., the returned tuple $(f_1, ..., f_n)$ corresponds to the map $f_n \circ \cdots \circ f_1$.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.hom_composite import EllipticCurveHom_
↪composite
sage: E = EllipticCurve(GF(43), [1,0])
sage: P, = E.gens()
sage: phi = EllipticCurveHom_composite(E, P)
sage: phi.factors()
(Isogeny of degree 2
  from Elliptic Curve defined by y^2 = x^3 + x over Finite Field of size 43
    to Elliptic Curve defined by y^2 = x^3 + 39*x over Finite Field of size␣
↪43,
 Isogeny of degree 2
  from Elliptic Curve defined by y^2 = x^3 + 39*x over Finite Field of size 43
    to Elliptic Curve defined by y^2 = x^3 + 42*x + 26 over Finite Field of␣
↪size 43,
 Isogeny of degree 11
  from Elliptic Curve defined by y^2 = x^3 + 42*x + 26 over Finite Field of␣
↪size 43
    to Elliptic Curve defined by y^2 = x^3 + x over Finite Field of size 43)
```

formal(*prec=20*)

Return the formal isogeny corresponding to this composite isogeny as a power series in the variable $t = -x/y$ on the domain curve.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: from sage.schemes.elliptic_curves.hom_composite import EllipticCurveHom_
↪composite
sage: E = EllipticCurve(GF(65537), [1,2,3,4,5])
sage: P = E.lift_x(7321)
sage: phi = EllipticCurveHom_composite(E, P)
sage: phi.formal()
t + 54203*t^5 + 48536*t^6 + 40698*t^7 + 37808*t^8 + 21111*t^9 + 42381*t^10
 + 46688*t^11 + 657*t^12 + 38916*t^13 + 62261*t^14 + 59707*t^15
 + 30767*t^16 + 7248*t^17 + 60287*t^18 + 50451*t^19 + 38305*t^20
 + 12312*t^21 + 31329*t^22 + O(t^23)
sage: (phi.dual() * phi).formal(prec=5)
9*t + 65501*t^2 + 65141*t^3 + 59183*t^4 + 21491*t^5 + 8957*t^6
 + 999*t^7 + O(t^8)
```

classmethod from_factors(*maps*, *E=None*, *strict=True*)

This method constructs a *EllipticCurveHom_composite* object encapsulating a given sequence of compatible isogenies.

The isogenies are composed in left-to-right order, i.e., the resulting composite map equals $f_{n-1} \circ \cdots \circ f_0$ where f_i denotes `maps[i]`.

INPUT:

- `maps` – sequence of *EllipticCurveHom* objects
- `E` (optional) – the domain elliptic curve
- `strict` (default: `True`) – if `True`, always return an *EllipticCurveHom_composite* object; else may return another *EllipticCurveHom* type

OUTPUT: the composite of `maps`

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.hom_composite import EllipticCurveHom_
↪composite
sage: E = EllipticCurve(GF(43), [1,0])
sage: P, = E.gens()
sage: phi = EllipticCurveHom_composite(E, P)
sage: psi = EllipticCurveHom_composite.from_factors(phi.factors())
sage: psi == phi
True
```

inseparable_degree()

Return the inseparable degree of this morphism.

Like the degree, the inseparable degree is multiplicative under composition, so this method returns the product of the inseparable degrees of the factors.

EXAMPLES:

```
sage: E = EllipticCurve(j=GF(11^5).random_element())
sage: phi = E.frobenius_isogeny(2) * E.scalar_multiplication(77)
sage: type(phi)
<class 'sage.schemes.elliptic_curves.hom_composite.EllipticCurveHom_composite
↪'>
sage: phi.inseparable_degree()
1331
```

kernel_polynomial()

Return the kernel polynomial of this composite isogeny.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: from sage.schemes.elliptic_curves.hom_composite import EllipticCurveHom_
↪composite
sage: E = EllipticCurve(GF(65537), [1,2,3,4,5])
sage: P = E.lift_x(7321)
sage: phi = EllipticCurveHom_composite(E, P); phi
Composite morphism of degree 9 = 3^2:
  From: Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5
        over Finite Field of size 65537
  To:   Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 28339*x +␣
↪59518
        over Finite Field of size 65537
sage: phi.kernel_polynomial()
x^4 + 46500*x^3 + 19556*x^2 + 7643*x + 15952
```

rational_maps()

Return the pair of explicit rational maps defining this composite isogeny.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: from sage.schemes.elliptic_curves.hom_composite import EllipticCurveHom_
↪composite
sage: E = EllipticCurve(GF(65537), [1,2,3,4,5])
sage: P = E.lift_x(7321)
sage: phi = EllipticCurveHom_composite(E, P)
sage: phi.rational_maps()
((x^9 + 27463*x^8 + 21204*x^7 - 5750*x^6 + 1610*x^5 + 14440*x^4
  + 26605*x^3 - 15569*x^2 - 3341*x + 1267)/(x^8 + 27463*x^7 + 26871*x^6
  + 5999*x^5 - 20194*x^4 - 6310*x^3 + 24366*x^2 - 20905*x - 13867),
 (x^12*y + 8426*x^11*y + 5667*x^11 + 27612*x^10*y + 26124*x^10 + 9688*x^9*y
  - 22715*x^9 + 19864*x^8*y + 498*x^8 + 22466*x^7*y - 14036*x^7 + 8070*x^6*y
  + 19955*x^6 - 20765*x^5*y - 12481*x^5 + 12672*x^4*y + 24142*x^4 - 23695*x^
↪3*y
  + 26667*x^3 + 23780*x^2*y + 17864*x^2 + 15053*x*y - 30118*x + 17539*y
  - 23609)/(x^12 + 8426*x^11 + 21945*x^10 - 22587*x^9 + 22094*x^8 + 14603*x^7
  - 26255*x^6 + 11171*x^5 - 16508*x^4 - 14435*x^3 - 2170*x^2 + 29081*x -␣
↪19009))
```

scaling_factor()

Return the Weierstrass scaling factor associated to this composite morphism.

The scaling factor is the constant u (in the base field) such that $\varphi^*\omega_2 = u\omega_1$, where $\varphi : E_1 \to E_2$ is this morphism and ω_i are the standard Weierstrass differentials on E_i defined by $\mathrm{d}x/(2y + a_1x + a_3)$.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: from sage.schemes.elliptic_curves.hom_composite import EllipticCurveHom_
↪composite
sage: from sage.schemes.elliptic_curves.weierstrass_morphism import␣
↪WeierstrassIsomorphism
sage: E = EllipticCurve(GF(65537), [1,2,3,4,5])
sage: P = E.lift_x(7321)
sage: phi = EllipticCurveHom_composite(E, P)
sage: phi = WeierstrassIsomorphism(phi.codomain(), [7,8,9,10]) * phi
sage: phi.formal()
7*t + 65474*t^2 + 511*t^3 + 61316*t^4 + 20548*t^5 + 45511*t^6 + 37285*t^7
 + 48414*t^8 + 9022*t^9 + 24025*t^10 + 35986*t^11 + 55397*t^12 + 25199*t^13
 + 18744*t^14 + 46142*t^15 + 9078*t^16 + 18030*t^17 + 47599*t^18
 + 12158*t^19 + 50630*t^20 + 56449*t^21 + 43320*t^22 + O(t^23)
sage: phi.scaling_factor()
7
```

ALGORITHM: The scaling factor is multiplicative under composition, so we return the product of the individual scaling factors associated to each factor.

x_rational_map()

Return the x-coordinate rational map of this composite isogeny.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: from sage.schemes.elliptic_curves.hom_composite import EllipticCurveHom_
```

(continues on next page)

(continued from previous page)

```
↪composite
sage: E = EllipticCurve(GF(65537), [1,2,3,4,5])
sage: P = E.lift_x(7321)
sage: phi = EllipticCurveHom_composite(E, P)
sage: phi.x_rational_map() == phi.rational_maps()[0]
True
```

CHAPTER

TEN

SUMS OF MORPHISMS OF ELLIPTIC CURVES

The set $\mathrm{Hom}(E, E')$ of morphisms between two elliptic curves forms an abelian group under pointwise addition. An important special case is the endomorphism ring $\mathrm{End}(E) = \mathrm{Hom}(E, E)$. However, it is not immediately obvious how to compute some properties of the sum $\varphi + \psi$ of two isogenies, even when both are given explicitly. This class provides functionality for representing sums of elliptic-curve morphisms (in particular, isogenies and endomorphisms) formally, and explicitly computing important properties (such as the degree or the kernel polynomial) from the formal representation.

EXAMPLES:

```
sage: E = EllipticCurve(GF(101), [5,5])
sage: phi = E.isogenies_prime_degree(7)[0]
sage: phi + phi
Sum morphism:
  From: Elliptic Curve defined by y^2 = x^3 + 5*x + 5 over Finite Field of size 101
  To:   Elliptic Curve defined by y^2 = x^3 + 12*x + 98 over Finite Field of size 101
  Via:  (Isogeny of degree 7 from Elliptic Curve defined by y^2 = x^3 + 5*x + 5 over␣
↪Finite Field of size 101 to Elliptic Curve defined by y^2 = x^3 + 12*x + 98 over␣
↪Finite Field of size 101, Isogeny of degree 7 from Elliptic Curve defined by y^2 =␣
↪x^3 + 5*x + 5 over Finite Field of size 101 to Elliptic Curve defined by y^2 = x^3␣
↪+ 12*x + 98 over Finite Field of size 101)
sage: phi + phi == phi * E.scalar_multiplication(2)
True
sage: phi + phi + phi == phi * E.scalar_multiplication(3)
True
```

An example of computing with a supersingular endomorphism ring:

```
sage: E = EllipticCurve(GF(419^2), [1,0])
sage: i = E.automorphisms()[-1]
sage: j = E.frobenius_isogeny()
sage: i * j == - j * i                          # i,j anticommute
True
sage: (i + j) * i == i^2 - i*j                  # distributive law
True
sage: (j - E.scalar_multiplication(1)).degree() # point counting!
420
```

AUTHORS:

- Lorenz Panny (2023)

class sage.schemes.elliptic_curves.hom_sum.**EllipticCurveHom_sum**(*phis*, *domain=None*, *codomain=None*)

Bases: *EllipticCurveHom*

Construct a sum morphism of elliptic curves from its summands. (For empty sums, the domain and codomain curves must be given.)

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.hom_sum import EllipticCurveHom_sum
sage: E = EllipticCurve(GF(101), [5,5])
sage: phi = E.isogenies_prime_degree(7)[0]
sage: EllipticCurveHom_sum([phi, phi])
Sum morphism:
  From: Elliptic Curve defined by y^2 = x^3 + 5*x + 5 over Finite Field of size␣
↪101
  To:   Elliptic Curve defined by y^2 = x^3 + 12*x + 98 over Finite Field of size␣
↪101
  Via:  (Isogeny of degree 7 from Elliptic Curve defined by y^2 = x^3 + 5*x + 5␣
↪over Finite Field of size 101 to Elliptic Curve defined by y^2 = x^3 + 12*x +␣
↪98 over Finite Field of size 101, Isogeny of degree 7 from Elliptic Curve␣
↪defined by y^2 = x^3 + 5*x + 5 over Finite Field of size 101 to Elliptic Curve␣
↪defined by y^2 = x^3 + 12*x + 98 over Finite Field of size 101)
```

The zero morphism can be constructed even between non-isogenous curves:

```
sage: E1 = EllipticCurve(GF(101), [5,5])
sage: E2 = EllipticCurve(GF(101), [7,7])
sage: E1.is_isogenous(E2)
False
sage: EllipticCurveHom_sum([], E1, E2)
Sum morphism:
  From: Elliptic Curve defined by y^2 = x^3 + 5*x + 5 over Finite Field of size␣
↪101
  To:   Elliptic Curve defined by y^2 = x^3 + 7*x + 7 over Finite Field of size␣
↪101
  Via:  ()
```

degree()

Return the degree of this sum morphism.

EXAMPLES:

```
sage: E = EllipticCurve(GF(101), [5,5])
sage: phi = E.isogenies_prime_degree(7)[0]
sage: (phi + phi).degree()
28
```

This method yields a simple toy point-counting algorithm:

```
sage: E = EllipticCurve(GF(101), [5,5])
sage: m1 = E.scalar_multiplication(1)
sage: pi = E.frobenius_endomorphism()
sage: (pi - m1).degree()
119
sage: E.count_points()
119
```

ALGORITHM: Essentially Schoof's algorithm; see `_compute_degree()`.

dual()

Return the dual of this sum morphism.

EXAMPLES:

```
sage: E = EllipticCurve(GF(101), [5,5])
sage: phi = E.isogenies_prime_degree(7)[0]
sage: (phi + phi).dual()
Sum morphism:
  From: Elliptic Curve defined by y^2 = x^3 + 12*x + 98 over Finite Field of␣
↪size 101
  To:   Elliptic Curve defined by y^2 = x^3 + 5*x + 5 over Finite Field of␣
↪size 101
  Via:  (Isogeny of degree 7 from Elliptic Curve defined by y^2 = x^3 + 12*x␣
↪+ 98 over Finite Field of size 101 to Elliptic Curve defined by y^2 = x^3 +␣
↪5*x + 5 over Finite Field of size 101, Isogeny of degree 7 from Elliptic␣
↪Curve defined by y^2 = x^3 + 12*x + 98 over Finite Field of size 101 to␣
↪Elliptic Curve defined by y^2 = x^3 + 5*x + 5 over Finite Field of size 101)
sage: (phi + phi).dual() == phi.dual() + phi.dual()
True
```

```
sage: E = EllipticCurve(GF(431^2), [1,0])
sage: iota = E.automorphisms()[2]
sage: m2 = E.scalar_multiplication(2)
sage: endo = m2 + iota
sage: endo.dual()
Sum morphism:
  From: Elliptic Curve defined by y^2 = x^3 + x over Finite Field in z2 of␣
↪size 431^2
  To:   Elliptic Curve defined by y^2 = x^3 + x over Finite Field in z2 of␣
↪size 431^2
  Via:  (Scalar-multiplication endomorphism [2] of Elliptic Curve defined by␣
↪y^2 = x^3 + x over Finite Field in z2 of size 431^2, Elliptic-curve␣
↪endomorphism of Elliptic Curve defined by y^2 = x^3 + x over Finite Field␣
↪in z2 of size 431^2
  Via:  (u,r,s,t) = (8*z2 + 427, 0, 0, 0))
sage: endo.dual() == (m2 - iota)
True
```

ALGORITHM: Taking the dual distributes over addition.

inseparable_degree()

Compute the inseparable degree of this sum morphism.

EXAMPLES:

```
sage: E = EllipticCurve(GF(7), [0,1])
sage: m3 = E.scalar_multiplication(3)
sage: m3.inseparable_degree()
1
sage: m4 = E.scalar_multiplication(4)
sage: m7 = m3 + m4; m7
Sum morphism:
  From: Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field of size 7
  To:   Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field of size 7
  Via:  (Scalar-multiplication endomorphism [3] of Elliptic Curve defined by␣
↪y^2 = x^3 + 1 over Finite Field of size 7, Scalar-multiplication␣
↪endomorphism [4] of Elliptic Curve defined by y^2 = x^3 + 1 over Finite␣
↪Field of size 7)
sage: m7.degree()
49
sage: m7.inseparable_degree()
7
```

A supersingular example:

```
sage: E = EllipticCurve(GF(7), [1,0])
sage: m3 = E.scalar_multiplication(3)
sage: m3.inseparable_degree()
1
sage: m4 = E.scalar_multiplication(4)
sage: m7 = m3 + m4; m7
Sum morphism:
  From: Elliptic Curve defined by y^2 = x^3 + x over Finite Field of size 7
  To:   Elliptic Curve defined by y^2 = x^3 + x over Finite Field of size 7
  Via:  (Scalar-multiplication endomorphism [3] of Elliptic Curve defined by␣
↪y^2 = x^3 + x over Finite Field of size 7, Scalar-multiplication␣
↪endomorphism [4] of Elliptic Curve defined by y^2 = x^3 + x over Finite␣
↪Field of size 7)
sage: m7.inseparable_degree()
49
```

kernel_polynomial()

Return the kernel polynomial of this sum morphism.

EXAMPLES:

```
sage: E = EllipticCurve(GF(101), [5,5])
sage: phi = E.isogenies_prime_degree(7)[0]
sage: (phi + phi).kernel_polynomial()
x^15 + 75*x^14 + 16*x^13 + 59*x^12 + 28*x^11 + 60*x^10 + 69*x^9 + 79*x^8 +␣
↪79*x^7 + 52*x^6 + 35*x^5 + 11*x^4 + 37*x^3 + 69*x^2 + 66*x + 63
```

```
sage: E = EllipticCurve(GF(11), [5,5])
sage: pi = E.frobenius_endomorphism()
sage: m1 = E.scalar_multiplication(1)
sage: (pi - m1).kernel_polynomial()
x^9 + 7*x^8 + 2*x^7 + 4*x^6 + 10*x^4 + 4*x^3 + 9*x^2 + 7*x
```

ALGORITHM: *to_isogeny_chain()*.

rational_maps()

Return the rational maps of this sum morphism.

EXAMPLES:

```
sage: E = EllipticCurve(GF(101), [5,5])
sage: phi = E.isogenies_prime_degree(7)[0]
sage: (phi + phi).rational_maps()
((5*x^28 + 43*x^27 + 26*x^26 - ... + 7*x^2 - 23*x + 38)/(23*x^27 + 16*x^26 +␣
↪9*x^25 + ... - 43*x^2 - 22*x + 37),
 (42*x^42*y - 44*x^41*y - 22*x^40*y + ... - 26*x^2*y - 50*x*y - 18*y)/(-24*x^
↪42 - 47*x^41 - 12*x^40 + ... + 18*x^2 - 48*x + 18))
```

ALGORITHM: *to_isogeny_chain()*.

scaling_factor()

Return the Weierstrass scaling factor associated to this sum morphism.

The scaling factor is the constant u (in the base field) such that $\varphi^*\omega_2 = u\omega_1$, where $\varphi : E_1 \to E_2$ is this morphism and ω_i are the standard Weierstrass differentials on E_i defined by $\mathrm{d}x/(2y + a_1x + a_3)$.

EXAMPLES:

```
sage: E = EllipticCurve(GF(101), [5,5])
sage: phi = E.isogenies_prime_degree(7)[0]
sage: phi.scaling_factor()
84
sage: (phi + phi).scaling_factor()
67
```

ALGORITHM: The scaling factor is additive under addition of elliptic-curve morphisms, so we simply add together the scaling factors of the *summands()*.

summands()

Return the individual summands making up this sum morphism.

EXAMPLES:

```
sage: E = EllipticCurve(j=5)
sage: m2 = E.scalar_multiplication(2)
sage: m3 = E.scalar_multiplication(3)
sage: m2 + m3
Sum morphism:
  From: Elliptic Curve defined by y^2 + x*y = x^3 + x^2 + 180*x + 17255 over␣
↪Rational Field
  To:   Elliptic Curve defined by y^2 + x*y = x^3 + x^2 + 180*x + 17255 over␣
↪Rational Field
  Via:  (Scalar-multiplication endomorphism [2] of Elliptic Curve defined by␣
↪y^2 + x*y = x^3 + x^2 + 180*x + 17255 over Rational Field, Scalar-
↪multiplication endomorphism [3] of Elliptic Curve defined by y^2 + x*y = x^
↪3 + x^2 + 180*x + 17255 over Rational Field)
```

to_isogeny_chain()

Convert this formal sum of elliptic-curve morphisms into a *EllipticCurveHom_composite* object representing the same morphism.

EXAMPLES:

```
sage: E = EllipticCurve(GF(101), [5,5])
sage: phi = E.isogenies_prime_degree(7)[0]
sage: (phi + phi).to_isogeny_chain()
Composite morphism of degree 28 = 4*1*7:
  From: Elliptic Curve defined by y^2 = x^3 + 5*x + 5 over Finite Field of␣
↪size 101
  To:   Elliptic Curve defined by y^2 = x^3 + 12*x + 98 over Finite Field of␣
↪size 101
```

```
sage: p = 419
sage: E = EllipticCurve(GF(p^2), [1,0])
sage: iota = E.automorphisms()[2]    # sqrt(-1)
sage: pi = E.frobenius_isogeny()     # sqrt(-p)
sage: endo = iota + pi
sage: endo.degree()
420
sage: endo.to_isogeny_chain()
Composite morphism of degree 420 = 4*1*3*5*7:
  From: Elliptic Curve defined by y^2 = x^3 + x over Finite Field in z2 of␣
↪size 419^2
  To:   Elliptic Curve defined by y^2 = x^3 + x over Finite Field in z2 of␣
↪size 419^2
```

The decomposition is impossible for the constant zero map:

```
sage: endo = iota*pi + pi*iota
sage: endo.degree()
0
sage: endo.to_isogeny_chain()
Traceback (most recent call last):
...
ValueError: zero morphism cannot be written as a composition of isogenies
```

Isomorphisms are supported as well:

```
sage: E = EllipticCurve(j=5); E
Elliptic Curve defined by y^2 + x*y = x^3 + x^2 + 180*x + 17255 over Rational␣
↪Field
sage: m2 = E.scalar_multiplication(2)
sage: m3 = E.scalar_multiplication(3)
sage: (m2 - m3).to_isogeny_chain()
Composite morphism of degree 1 = 1^2:
  From: Elliptic Curve defined by y^2 + x*y = x^3 + x^2 + 180*x + 17255 over␣
↪Rational Field
  To:   Elliptic Curve defined by y^2 + x*y = x^3 + x^2 + 180*x + 17255 over␣
↪Rational Field
sage: (m2 - m3).rational_maps()
(x, -x - y)
```

x_rational_map()

Return the x-coordinate rational map of this sum morphism.

EXAMPLES:

```
sage: E = EllipticCurve(GF(101), [5,5])
sage: phi = E.isogenies_prime_degree(7)[0]
sage: (phi + phi).x_rational_map()
(9*x^28 + 37*x^27 + 67*x^26 + ... + 53*x^2 + 100*x + 28)/(x^27 + 49*x^26 +␣
↪97*x^25 + ... + 64*x^2 + 21*x + 6)
```

ALGORITHM: *to_isogeny_chain()*.

CHAPTER

ELEVEN

ISOMORPHISMS BETWEEN WEIERSTRASS MODELS OF ELLIPTIC CURVES

AUTHORS:

- Robert Bradshaw (2007): initial version
- John Cremona (Jan 2008): isomorphisms, automorphisms and twists in all characteristics
- Lorenz Panny (2021): *EllipticCurveHom* interface

class sage.schemes.elliptic_curves.weierstrass_morphism.**WeierstrassIsomorphism**(*E=None*, *urst=None*, *F=None*)

Bases: *EllipticCurveHom*, *baseWI*

Class representing a Weierstrass isomorphism between two elliptic curves.

INPUT:

- `E` – an `EllipticCurve`, or `None` (see below).
- `urst` – a 4-tuple (u, r, s, t), a *baseWI* object, or `None` (see below).
- `F` – an `EllipticCurve`, or `None` (see below).

Given two Elliptic Curves `E` and `F` (represented by Weierstrass models as usual), and a transformation `urst` from `E` to `F`, construct an isomorphism from `E` to `F`. An exception is raised if `urst(E) != F`. At most one of `E`, `F`, `urst` can be `None`. In this case, the missing input is constructed from the others in such a way that `urst(E) == F` holds, and an exception is raised if this is impossible (typically because `E` and `F` are not isomorphic).

Users will not usually need to use this class directly, but instead use methods such as *isomorphism_to()* or *isomorphisms()*.

Explicitly, the isomorphism defined by (u, r, s, t) maps a point (x, y) to the point

$$((x - r)/u^2,\ (y - s(x - r) - t)/u^3).$$

If the domain E has Weierstrass coefficients $[a_1, a_2, a_3, a_4, a_6]$, the codomain F is given by

$$\begin{aligned}
a'_1 &= (a_1 + 2s)/u \\
a'_2 &= (a_2 - a_1 s + 3r - s^2)/u^2 \\
a'_3 &= (a_3 + a_1 r + 2t)/u^3 \\
a'_4 &= (a_4 + 2a_2 r - a_1(rs + t) - a_3 s + 3r^2 - 2st)/u^4 \\
a'_6 &= (a_6 - a_1 rt + a_2 r^2 - a_3 t + a_4 r + r^3 - t^2)/u^6.
\end{aligned}$$

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.weierstrass_morphism import *
sage: WeierstrassIsomorphism(EllipticCurve([0,1,2,3,4]), (-1,2,3,4))
Elliptic-curve morphism:
  From: Elliptic Curve defined by y^2 + 2*y = x^3 + x^2 + 3*x + 4 over Rational␣
↪Field
  To:   Elliptic Curve defined by y^2 - 6*x*y - 10*y = x^3 - 2*x^2 - 11*x - 2␣
↪over Rational Field
  Via:  (u,r,s,t) = (-1, 2, 3, 4)
sage: E = EllipticCurve([0,1,2,3,4])
sage: F = EllipticCurve(E.cremona_label())
sage: WeierstrassIsomorphism(E, None, F)
Elliptic-curve morphism:
  From: Elliptic Curve defined by y^2 + 2*y = x^3 + x^2 + 3*x + 4 over Rational␣
↪Field
  To:   Elliptic Curve defined by y^2  = x^3 + x^2 + 3*x + 5 over Rational Field
  Via:  (u,r,s,t) = (1, 0, 0, -1)
sage: w = WeierstrassIsomorphism(None, (1,0,0,-1), F)
sage: w._domain == E
True
```

dual()

Return the dual isogeny of this isomorphism.

For isomorphisms, the dual is just the inverse.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.weierstrass_morphism import␣
↪WeierstrassIsomorphism
sage: E = EllipticCurve(QuadraticField(-3), [0,1])                              #␣
↪needs sage.rings.number_field
sage: w = WeierstrassIsomorphism(E, (CyclotomicField(3).gen(),0,0,0))           #␣
↪needs sage.rings.number_field
sage: (w.dual() * w).rational_maps()                                            #␣
↪needs sage.rings.number_field
(x, y)
```

```
sage: E1 = EllipticCurve([11,22,33,44,55])
sage: E2 = E1.short_weierstrass_model()
sage: iso = E1.isomorphism_to(E2)
sage: iso.dual() == ~iso
True
```

inseparable_degree()

Return the inseparable degree of this Weierstrass isomorphism.

For isomorphisms, this method always returns one.

is_identity()

Check if this Weierstrass isomorphism is the identity.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.weierstrass_morphism import␣
↪WeierstrassIsomorphism
sage: p = 97
sage: Fp = GF(p)
sage: E = EllipticCurve(Fp, [1, 28])
```

(continues on next page)

(continued from previous page)

```
sage: ws = WeierstrassIsomorphism(E, None, E)
sage: ws.is_identity()
False
```

```
sage: from sage.schemes.elliptic_curves.weierstrass_morphism import␣
↪WeierstrassIsomorphism
sage: p = 97
sage: Fp = GF(p)
sage: E = EllipticCurve(Fp, [1, 28])
sage: ws = WeierstrassIsomorphism(E, (1, 0, 0, 0), None)
sage: ws.is_identity()
True
```

kernel_polynomial()

Return the kernel polynomial of this isomorphism.

Isomorphisms have trivial kernel by definition, hence this method always returns 1.

EXAMPLES:

```
sage: E1 = EllipticCurve([11,22,33,44,55])
sage: E2 = EllipticCurve_from_j(E1.j_invariant())
sage: iso = E1.isomorphism_to(E2)
sage: iso.kernel_polynomial()
1
sage: psi = E1.isogeny(iso.kernel_polynomial(), codomain=E2); psi
Isogeny of degree 1
 from Elliptic Curve defined by y^2 + 11*x*y + 33*y = x^3 + 22*x^2 + 44*x + 55
      over Rational Field
   to Elliptic Curve defined by y^2 + x*y = x^3 + x^2 - 684*x + 6681
      over Rational Field
sage: psi in {iso, -iso}
True
```

order()

Compute the order of this Weierstrass isomorphism if it is an automorphism.

A `ValueError` is raised if the domain is not equal to the codomain.

A `NotImplementedError` is raised if the order of the automorphism is not 1, 2, 3, 4 or 6.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.weierstrass_morphism import *
sage: p = 97
sage: Fp = GF(p)
sage: E = EllipticCurve(Fp, [1, 28])
sage: ws = WeierstrassIsomorphism(E, None, E)
sage: ws.order()
2
```

rational_maps()

Return the pair of rational maps defining this isomorphism.

EXAMPLES:

```
sage: E1 = EllipticCurve([11,22,33,44,55])
sage: E2 = EllipticCurve_from_j(E1.j_invariant())
sage: iso = E1.isomorphism_to(E2); iso
Elliptic-curve morphism:
  From: Elliptic Curve defined by y^2 + 11*x*y + 33*y = x^3 + 22*x^2 + 44*x +␣
↪55
        over Rational Field
  To:   Elliptic Curve defined by y^2 + x*y = x^3 + x^2 - 684*x + 6681
        over Rational Field
  Via:  (u,r,s,t) = (1, -17, -5, 77)
sage: iso.rational_maps()
(x + 17, 5*x + y + 8)
sage: f = E2.defining_polynomial()(*iso.rational_maps(), 1)
sage: I = E1.defining_ideal()
sage: x,y,z = I.ring().gens()
sage: f in I + Ideal(z-1)
True
```

```
sage: # needs sage.rings.finite_rings
sage: E = EllipticCurve(GF(65537), [1,1,1,1,1])
sage: w = E.isomorphism_to(E.short_weierstrass_model())
sage: f,g = w.rational_maps()
sage: P = E.random_point()
sage: w(P).xy() == (f(P.xy()), g(P.xy()))
True
```

scaling_factor()

Return the Weierstrass scaling factor associated to this Weierstrass isomorphism.

The scaling factor is the constant u (in the base field) such that $\varphi^*\omega_2 = u\omega_1$, where $\varphi : E_1 \to E_2$ is this isomorphism and ω_i are the standard Weierstrass differentials on E_i defined by $\mathrm{d}x/(2y + a_1x + a_3)$.

EXAMPLES:

```
sage: E = EllipticCurve(QQbar, [0,1])                                          #␣
↪needs sage.rings.number_field
sage: all(f.scaling_factor() == f.formal()[1] for f in E.automorphisms())      #␣
↪needs sage.rings.number_field
True
```

ALGORITHM: The scaling factor equals the u component of the tuple (u, r, s, t) defining the isomorphism.

x_rational_map()

Return the x-coordinate rational map of this isomorphism.

EXAMPLES:

```
sage: E1 = EllipticCurve([11,22,33,44,55])
sage: E2 = EllipticCurve_from_j(E1.j_invariant())
sage: iso = E1.isomorphism_to(E2); iso
Elliptic-curve morphism:
  From: Elliptic Curve defined by y^2 + 11*x*y + 33*y = x^3 + 22*x^2 + 44*x +␣
↪55
        over Rational Field
  To:   Elliptic Curve defined by y^2 + x*y = x^3 + x^2 - 684*x + 6681
        over Rational Field
  Via:  (u,r,s,t) = (1, -17, -5, 77)
sage: iso.x_rational_map()
```

(continues on next page)

(continued from previous page)

```
x + 17
sage: iso.x_rational_map() == iso.rational_maps()[0]
True
```

class sage.schemes.elliptic_curves.weierstrass_morphism.**baseWI**(*u=1*, *r=0*, *s=0*, *t=0*)

Bases: object

This class implements the basic arithmetic of isomorphisms between Weierstrass models of elliptic curves.

These are specified by lists of the form $[u, r, s, t]$ (with $u \neq 0$) which specifies a transformation $(x, y) \mapsto (x', y')$ where

$$(x, y) = (u^2x' + r, u^3y' + su^2x' + t).$$

INPUT:

- u, r, s, t – (default: 1, 0, 0, 0); standard parameters of an isomorphism between Weierstrass models

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.weierstrass_morphism import *
sage: baseWI()
(1, 0, 0, 0)
sage: baseWI(2,3,4,5)
(2, 3, 4, 5)
sage: R.<u,r,s,t> = QQ[]
sage: baseWI(u,r,s,t)
(u, r, s, t)
```

is_identity()

Return True if this is the identity isomorphism.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.weierstrass_morphism import *
sage: w = baseWI(); w.is_identity()
True
sage: w = baseWI(2,3,4,5); w.is_identity()
False
```

tuple()

Return the parameters u, r, s, t as a tuple.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.weierstrass_morphism import *
sage: w = baseWI(2,3,4,5)
sage: w.tuple()
(2, 3, 4, 5)
```

sage.schemes.elliptic_curves.weierstrass_morphism.**identity_morphism**(*E*)

Given an elliptic curve E, return the identity morphism on E as a *WeierstrassIsomorphism*.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.weierstrass_morphism import identity_
↪morphism
sage: E = EllipticCurve([5,6,7,8,9])
```

(continues on next page)

(continued from previous page)

```
sage: id_ = identity_morphism(E)
sage: id_.rational_maps()
(x, y)
```

sage.schemes.elliptic_curves.weierstrass_morphism.**negation_morphism**(*E*)

Given an elliptic curve E, return the negation endomorphism $[-1]$ of E as a *WeierstrassIsomorphism*.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.weierstrass_morphism import negation_
↪morphism
sage: E = EllipticCurve([5,6,7,8,9])
sage: neg = negation_morphism(E)
sage: neg.rational_maps()
(x, -5*x - y - 7)
```

CHAPTER

TWELVE

ISOGENIES

An isogeny $\varphi : E_1 \to E_2$ between two elliptic curves E_1 and E_2 is a morphism of curves that sends the origin of E_1 to the origin of E_2. Such a morphism is automatically a morphism of group schemes and the kernel is a finite subgroup scheme of E_1. Such a subscheme can either be given by a list of generators, which have to be torsion points, or by a polynomial in the coordinate x of the Weierstrass equation of E_1.

The usual way to create and work with isogenies is illustrated with the following example:

```
sage: k = GF(11)
sage: E = EllipticCurve(k, [1,1])
sage: Q = E(6,5)
sage: phi = E.isogeny(Q)
sage: phi
Isogeny of degree 7
 from Elliptic Curve defined by y^2 = x^3 + x + 1 over Finite Field of size 11
   to Elliptic Curve defined by y^2 = x^3 + 7*x + 8
over Finite Field of size 11
sage: P = E(4,5)
sage: phi(P)
(10 : 0 : 1)
sage: phi.codomain()
Elliptic Curve defined by y^2 = x^3 + 7*x + 8 over Finite Field of size 11
sage: phi.rational_maps()
((x^7 + 4*x^6 - 3*x^5 - 2*x^4
   - 3*x^3 + 3*x^2 + x - 2)/(x^6 + 4*x^5 - 4*x^4 - 5*x^3 + 5*x^2),
 (x^9*y - 5*x^8*y - x^7*y + x^5*y - x^4*y
   - 5*x^3*y - 5*x^2*y - 2*x*y - 5*y)/(x^9 - 5*x^8 + 4*x^6 - 3*x^4 + 2*x^3))
```

The methods directly accessible from an elliptic curve `E` over a field are *isogeny()* and *isogeny_codomain()*.

The most useful methods that apply to isogenies are:

- `.domain()`
- `.codomain()`
- *degree()*
- *dual()*
- *rational_maps()*
- *kernel_polynomial()*

Warning: This class only implements separable isogenies. When using Kohel's algorithm, only cyclic isogenies can be computed (except for [2]).

Working with other kinds of isogenies may be possible using other child classes of *EllipticCurveHom*.

Some algorithms may need the isogeny to be normalized.

AUTHORS:

- Daniel Shumow <shumow@gmail.com>: 2009-04-19: initial version
- Chris Wuthrich: 7/09: add check of input, not the full list is needed. 10/09: eliminating some bugs.
- John Cremona 2014-08-08: tidying of code and docstrings, systematic use of univariate vs. bivariate polynomials and rational functions.
- Lorenz Panny (2022-04): major cleanup of code and documentation
- Lorenz Panny (2022): inseparable duals
- Rémy Oudompheng (2023): implementation of the BMSS algorithm

class sage.schemes.elliptic_curves.ell_curve_isogeny.**EllipticCurveIsogeny**(*E*, *kernel*, *codomain=None*, *degree=None*, *model=None*, *check=True*)

Bases: *EllipticCurveHom*

This class implements separable isogenies of elliptic curves.

Several different algorithms for computing isogenies are available. These include:

- Vélu's Formulas: Vélu's original formulas for computing isogenies. This algorithm is selected by giving as the `kernel` parameter a single point, or a list of points, generating a finite subgroup.
- Kohel's Formulas: Kohel's original formulas for computing isogenies. This algorithm is selected by giving as the `kernel` parameter a monic polynomial (or a coefficient list) which will define the kernel of the isogeny. Kohel's algorithm is currently only implemented for cyclic isogenies, with the exception of $[2]$.

INPUT:

- `E` – an elliptic curve, the domain of the isogeny to initialize.
- `kernel` – a kernel: either a point on `E`, a list of points on `E`, a monic kernel polynomial, or `None`. If initializing from a domain/codomain, this must be `None`.
- `codomain` – an elliptic curve (default: `None`).
  - If `kernel` is `None`, then `degree` must be given as well and the given `codomain` must be the codomain of a cyclic, separable, normalized isogeny of the given degree.
  - If `kernel` is not `None`, then this must be isomorphic to the codomain of the separable isogeny defined by `kernel`; in this case, the isogeny is post-composed with an isomorphism so that the codomain equals the given curve.
- `degree` – an integer (default: `None`).
  - If `kernel` is `None`, then this is the degree of the isogeny from `E` to `codomain`.
  - If `kernel` is not `None`, then this is used to determine whether or not to skip a gcd of the given kernel polynomial with the two-torsion polynomial of `E`.

- `model` – a string (default: `None`). Supported values (cf. *`compute_model()`*):
  - `"minimal"`: If `E` is a curve over the rationals or over a number field, then the codomain is a global minimal model where this exists.
  - `"short_weierstrass"`: The codomain is a short Weierstrass curve, assuming one exists.
  - `"montgomery"`: The codomain is an (untwisted) Montgomery curve, assuming one exists over this field.
- `check` (default: `True`) – check whether the input is valid. Setting this to `False` can lead to significant speedups.

EXAMPLES:

A simple example of creating an isogeny of a field of small characteristic:

```
sage: E = EllipticCurve(GF(7), [0,0,0,1,0])
sage: phi = EllipticCurveIsogeny(E, E((0,0)) ); phi
Isogeny of degree 2
 from Elliptic Curve defined by y^2 = x^3 + x over Finite Field of size 7
   to Elliptic Curve defined by y^2 = x^3 + 3*x over Finite Field of size 7
sage: phi.degree() == 2
True
sage: phi.kernel_polynomial()
x
sage: phi.rational_maps()
((x^2 + 1)/x, (x^2*y - y)/x^2)
sage: phi == loads(dumps(phi))           # known bug
True
```

A more complicated example of a characteristic-2 field:

```
sage: # needs sage.rings.finite_rings
sage: E = EllipticCurve(GF(2^4,'alpha'), [0,0,1,0,1])
sage: P = E((1,1))
sage: phi_v = EllipticCurveIsogeny(E, P); phi_v
Isogeny of degree 3
 from Elliptic Curve defined by y^2 + y = x^3 + 1
      over Finite Field in alpha of size 2^4
   to Elliptic Curve defined by y^2 + y = x^3
      over Finite Field in alpha of size 2^4
sage: phi_ker_poly = phi_v.kernel_polynomial()
sage: phi_ker_poly
x + 1
sage: phi_k = EllipticCurveIsogeny(E, phi_ker_poly)
sage: phi_k == phi_v
True
sage: phi_k.rational_maps()
((x^3 + x + 1)/(x^2 + 1), (x^3*y + x^2*y + x*y + x + y)/(x^3 + x^2 + x + 1))
sage: phi_v.rational_maps()
((x^3 + x + 1)/(x^2 + 1), (x^3*y + x^2*y + x*y + x + y)/(x^3 + x^2 + x + 1))
sage: phi_k.degree() == phi_v.degree() == 3
True
sage: phi_k.is_separable()
True
sage: phi_v(E(0))
(0 : 1 : 0)
sage: alpha = E.base_field().gen()
sage: Q = E((0, alpha*(alpha + 1)))
```

(continues on next page)

(continued from previous page)

```
sage: phi_v(Q)
(1 : alpha^2 + alpha : 1)
sage: phi_v(P) == phi_k(P)
True
sage: phi_k(P) == phi_v.codomain()(0)
True
```

We can create an isogeny whose kernel equals the full 2-torsion:

```
sage: # needs sage.rings.finite_rings
sage: E = EllipticCurve(GF((3,2)), [0,0,0,1,1])
sage: ker_poly = E.division_polynomial(2)
sage: phi = EllipticCurveIsogeny(E, ker_poly); phi
Isogeny of degree 4
 from Elliptic Curve defined by y^2 = x^3 + x + 1
      over Finite Field in z2 of size 3^2
   to Elliptic Curve defined by y^2 = x^3 + x + 1
      over Finite Field in z2 of size 3^2
sage: P1,P2,P3 = filter(bool, E(0).division_points(2))
sage: phi(P1)
(0 : 1 : 0)
sage: phi(P2)
(0 : 1 : 0)
sage: phi(P3)
(0 : 1 : 0)
sage: phi.degree()
4
```

We can also create trivial isogenies with the trivial kernel:

```
sage: E = EllipticCurve(GF(17), [11, 11, 4, 12, 10])
sage: phi_v = EllipticCurveIsogeny(E, E(0))
sage: phi_v.degree()
1
sage: phi_v.rational_maps()
(x, y)
sage: E == phi_v.codomain()
True
sage: P = E.random_point()
sage: phi_v(P) == P
True

sage: E = EllipticCurve(GF(31), [23, 1, 22, 7, 18])
sage: phi_k = EllipticCurveIsogeny(E, [1]); phi_k
Isogeny of degree 1
 from Elliptic Curve defined by y^2 + 23*x*y + 22*y = x^3 + x^2 + 7*x + 18
      over Finite Field of size 31
   to Elliptic Curve defined by y^2 + 23*x*y + 22*y = x^3 + x^2 + 7*x + 18
      over Finite Field of size 31
sage: phi_k.degree()
1
sage: phi_k.rational_maps()
(x, y)
sage: phi_k.codomain() == E
True
sage: phi_k.kernel_polynomial()
1
```

(continues on next page)

(continued from previous page)

```
sage: P = E.random_point(); P == phi_k(P)
True
```

Vélu and Kohel also work in characteristic 0:

```
sage: E = EllipticCurve(QQ, [0,0,0,3,4])
sage: P_list = E.torsion_points()
sage: phi = EllipticCurveIsogeny(E, P_list); phi
Isogeny of degree 2
 from Elliptic Curve defined by y^2 = x^3 + 3*x + 4 over Rational Field
   to Elliptic Curve defined by y^2 = x^3 - 27*x + 46 over Rational Field
sage: P = E((0,2))
sage: phi(P)
(6 : -10 : 1)
sage: phi_ker_poly = phi.kernel_polynomial()
sage: phi_ker_poly
x + 1
sage: phi_k = EllipticCurveIsogeny(E, phi_ker_poly); phi_k
Isogeny of degree 2
 from Elliptic Curve defined by y^2 = x^3 + 3*x + 4 over Rational Field
   to Elliptic Curve defined by y^2 = x^3 - 27*x + 46 over Rational Field
sage: phi_k(P) == phi(P)
True
sage: phi_k == phi
True
sage: phi_k.degree()
2
sage: phi_k.is_separable()
True
```

A more complicated example over the rationals (of odd degree):

```
sage: E = EllipticCurve('11a1')
sage: P_list = E.torsion_points()
sage: phi_v = EllipticCurveIsogeny(E, P_list); phi_v
Isogeny of degree 5
 from Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational␣
↪Field
   to Elliptic Curve defined by y^2 + y = x^3 - x^2 - 7820*x - 263580 over␣
↪Rational Field
sage: P = E((16,-61))
sage: phi_v(P)
(0 : 1 : 0)
sage: ker_poly = phi_v.kernel_polynomial(); ker_poly
x^2 - 21*x + 80
sage: phi_k = EllipticCurveIsogeny(E, ker_poly); phi_k
Isogeny of degree 5
 from Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational␣
↪Field
   to Elliptic Curve defined by y^2 + y = x^3 - x^2 - 7820*x - 263580 over␣
↪Rational Field
sage: phi_k == phi_v
True
sage: phi_v(P) == phi_k(P)
True
sage: phi_k.is_separable()
True
```

We can also do this same example over the number field defined by the irreducible two-torsion polynomial of E:

```
sage: # needs sage.rings.number_field
sage: E = EllipticCurve('11a1')
sage: P_list = E.torsion_points()
sage: x = polygen(ZZ, 'x')
sage: K.<alpha> = NumberField(x^3 - 2* x^2 - 40*x - 158)
sage: EK = E.change_ring(K)
sage: P_list = [EK(P) for P in P_list]
sage: phi_v = EllipticCurveIsogeny(EK, P_list); phi_v
Isogeny of degree 5
 from Elliptic Curve defined by y^2 + y = x^3 + (-1)*x^2 + (-10)*x + (-20)
      over Number Field in alpha with defining polynomial x^3 - 2*x^2 - 40*x - 158
   to Elliptic Curve defined by y^2 + y = x^3 + (-1)*x^2 + (-7820)*x + (-263580)
      over Number Field in alpha with defining polynomial x^3 - 2*x^2 - 40*x - 158
sage: P = EK((alpha/2,-1/2))
sage: phi_v(P)
(122/121*alpha^2 + 1633/242*alpha - 3920/121 : -1/2 : 1)
sage: ker_poly = phi_v.kernel_polynomial()
sage: ker_poly
x^2 - 21*x + 80
sage: phi_k = EllipticCurveIsogeny(EK, ker_poly); phi_k
Isogeny of degree 5
 from Elliptic Curve defined by y^2 + y = x^3 + (-1)*x^2 + (-10)*x + (-20)
      over Number Field in alpha with defining polynomial x^3 - 2*x^2 - 40*x - 158
   to Elliptic Curve defined by y^2 + y = x^3 + (-1)*x^2 + (-7820)*x + (-263580)
      over Number Field in alpha with defining polynomial x^3 - 2*x^2 - 40*x - 158
sage: phi_v == phi_k
True
sage: phi_k(P) == phi_v(P)
True
sage: phi_k == phi_v
True
sage: phi_k.degree()
5
sage: phi_v.is_separable()
True
```

The following example shows how to specify an isogeny from domain and codomain:

```
sage: E = EllipticCurve('11a1')
sage: R.<x> = QQ[]
sage: f = x^2 - 21*x + 80
sage: phi = E.isogeny(f)
sage: E2 = phi.codomain()
sage: phi_s = EllipticCurveIsogeny(E, None, E2, 5); phi_s
Isogeny of degree 5
 from Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational␣
↪Field
   to Elliptic Curve defined by y^2 + y = x^3 - x^2 - 7820*x - 263580 over␣
↪Rational Field
sage: phi_s == phi
True
sage: phi_s.rational_maps() == phi.rational_maps()
True
```

However, only cyclic normalized isogenies can be constructed this way. The non-cyclic multiplication-by-3 isogeny won't be found:

```
sage: E.isogeny(None, codomain=E, degree=9)
Traceback (most recent call last):
...
ValueError: the two curves are not linked by a cyclic normalized isogeny of␣
↪degree 9
```

Non-normalized isogeny also won't be found:

```
sage: E2.isogeny(None, codomain=E, degree=5)
Traceback (most recent call last):
...
ValueError: the two curves are not linked by a cyclic normalized isogeny of␣
↪degree 5
sage: phihat = phi.dual(); phihat
Isogeny of degree 5
 from Elliptic Curve defined by y^2 + y = x^3 - x^2 - 7820*x - 263580
      over Rational Field
   to Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational␣
↪Field
sage: phihat.is_normalized()
False
```

Here an example of a construction of a endomorphisms with cyclic kernel on a CM-curve:

```
sage: # needs sage.rings.number_field
sage: K.<i> = NumberField(x^2 + 1)
sage: E = EllipticCurve(K, [1,0])
sage: RK.<X> = K[]
sage: f = X^2 - 2/5*i + 1/5
sage: phi= E.isogeny(f)
sage: isom = phi.codomain().isomorphism_to(E)
sage: phi = isom * phi
sage: phi.codomain() == phi.domain()
True
sage: phi.rational_maps()
(((4/25*i + 3/25)*x^5 + (4/5*i - 2/5)*x^3 - x)/(x^4 + (-4/5*i + 2/5)*x^2 + (-4/
↪25*i - 3/25)),
 ((11/125*i + 2/125)*x^6*y + (-23/125*i + 64/125)*x^4*y + (141/125*i + 162/125)*x^
↪2*y + (3/25*i - 4/25)*y)/(x^6 + (-6/5*i + 3/5)*x^4 + (-12/25*i - 9/25)*x^2 + (2/
↪125*i - 11/125)))
```

dual()

Return the isogeny dual to this isogeny.

Note: If $\varphi\colon E \to E'$ is the given isogeny and n is its degree, then the dual is by definition the unique isogeny $\hat{\varphi}\colon E' \to E$ such that the compositions $\hat{\varphi} \circ \varphi$ and $\varphi \circ \hat{\varphi}$ are the multiplication-by-n maps on E and E', respectively.

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: R.<x> = QQ[]
sage: f = x^2 - 21*x + 80
sage: phi = EllipticCurveIsogeny(E, f)
sage: phi_hat = phi.dual()
```

(continues on next page)

(continued from previous page)

```
sage: phi_hat.domain() == phi.codomain()
True
sage: phi_hat.codomain() == phi.domain()
True
sage: (X, Y) = phi.rational_maps()
sage: (Xhat, Yhat) = phi_hat.rational_maps()
sage: Xm = Xhat.subs(x=X, y=Y)
sage: Ym = Yhat.subs(x=X, y=Y)
sage: (Xm, Ym) == E.multiplication_by_m(5)
True

sage: E = EllipticCurve(GF(37), [0,0,0,1,8])
sage: R.<x> = GF(37)[]
sage: f = x^3 + x^2 + 28*x + 33
sage: phi = EllipticCurveIsogeny(E, f)
sage: phi_hat = phi.dual()
sage: phi_hat.codomain() == phi.domain()
True
sage: phi_hat.domain() == phi.codomain()
True
sage: (X, Y) = phi.rational_maps()
sage: (Xhat, Yhat) = phi_hat.rational_maps()
sage: Xm = Xhat.subs(x=X, y=Y)
sage: Ym = Yhat.subs(x=X, y=Y)
sage: (Xm, Ym) == E.multiplication_by_m(7)
True

sage: E = EllipticCurve(GF(31), [0,0,0,1,8])
sage: R.<x> = GF(31)[]
sage: f = x^2 + 17*x + 29
sage: phi = EllipticCurveIsogeny(E, f)
sage: phi_hat = phi.dual()
sage: phi_hat.codomain() == phi.domain()
True
sage: phi_hat.domain() == phi.codomain()
True
sage: (X, Y) = phi.rational_maps()
sage: (Xhat, Yhat) = phi_hat.rational_maps()
sage: Xm = Xhat.subs(x=X, y=Y)
sage: Ym = Yhat.subs(x=X, y=Y)
sage: (Xm, Ym) == E.multiplication_by_m(5)
True
```

Inseparable duals should be computed correctly:

```
sage: # needs sage.rings.finite_rings
sage: z2 = GF(71^2).gen()
sage: E = EllipticCurve(j=57*z2+51)
sage: E.isogeny(3*E.lift_x(0)).dual()
Composite morphism of degree 71 = 71*1^2:
  From: Elliptic Curve defined by y^2 = x^3 + (32*z2+67)*x + (24*z2+37)
        over Finite Field in z2 of size 71^2
  To:   Elliptic Curve defined by y^2 = x^3 + (41*z2+56)*x + (18*z2+42)
        over Finite Field in z2 of size 71^2
sage: E.isogeny(E.lift_x(0)).dual()
Composite morphism of degree 213 = 71*3:
  From: Elliptic Curve defined by y^2 = x^3 + (58*z2+31)*x + (34*z2+58)
```

(continues on next page)

(continued from previous page)

```
        over Finite Field in z2 of size 71^2
  To:   Elliptic Curve defined by y^2 = x^3 + (41*z2+56)*x + (18*z2+42)
        over Finite Field in z2 of size 71^2
```

…even if pre- or post-isomorphisms are present:

```
sage: # needs sage.rings.finite_rings
sage: from sage.schemes.elliptic_curves.weierstrass_morphism import␣
↪WeierstrassIsomorphism
sage: phi = E.isogeny(E.lift_x(0))
sage: pre = ~WeierstrassIsomorphism(phi.domain(), (z2,2,3,4))
sage: post = WeierstrassIsomorphism(phi.codomain(), (5,6,7,8))
sage: phi = post * phi * pre
sage: phi.dual()
Composite morphism of degree 213 = 71*3:
  From: Elliptic Curve defined
        by y^2 + 17*x*y + 45*y = x^3 + 30*x^2 + (6*z2+64)*x + (48*z2+65)
        over Finite Field in z2 of size 71^2
  To:   Elliptic Curve defined
        by y^2 + (60*z2+22)*x*y + (69*z2+37)*y = x^3 + (32*z2+48)*x^2
           + (19*z2+58)*x + (56*z2+22)
        over Finite Field in z2 of size 71^2
```

inseparable_degree()

Return the inseparable degree of this isogeny.

Since this class only implements separable isogenies, this method always returns one.

kernel_polynomial()

Return the kernel polynomial of this isogeny.

EXAMPLES:

```
sage: E = EllipticCurve(QQ, [0,0,0,2,0])
sage: phi = EllipticCurveIsogeny(E, E((0,0)))
sage: phi.kernel_polynomial()
x

sage: E = EllipticCurve('11a1')
sage: phi = EllipticCurveIsogeny(E, E.torsion_points())
sage: phi.kernel_polynomial()
x^2 - 21*x + 80

sage: E = EllipticCurve(GF(17), [1,-1,1,-1,1])
sage: phi = EllipticCurveIsogeny(E, [1])
sage: phi.kernel_polynomial()
1

sage: E = EllipticCurve(GF(31), [0,0,0,3,0])
sage: phi = EllipticCurveIsogeny(E, [0,3,0,1])
sage: phi.kernel_polynomial()
x^3 + 3*x
```

rational_maps()

Return the pair of rational maps defining this isogeny.

Note: Both components are returned as elements of the function field $F(x, y)$ in two variables over the base

field F, though the first only involves x. To obtain the x-coordinate function as a rational function in $F(x)$, use *x_rational_map()*.

EXAMPLES:

```
sage: E = EllipticCurve(QQ, [0,2,0,1,-1])
sage: phi = EllipticCurveIsogeny(E, [1])
sage: phi.rational_maps()
(x, y)

sage: E = EllipticCurve(GF(17), [0,0,0,3,0])
sage: phi = EllipticCurveIsogeny(E, E((0,0)))
sage: phi.rational_maps()
((x^2 + 3)/x, (x^2*y - 3*y)/x^2)
```

scaling_factor()

Return the Weierstrass scaling factor associated to this elliptic-curve isogeny.

The scaling factor is the constant u (in the base field) such that $\varphi^*\omega_2 = u\omega_1$, where $\varphi : E_1 \to E_2$ is this isogeny and ω_i are the standard Weierstrass differentials on E_i defined by $\mathrm{d}x/(2y + a_1x + a_3)$.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: E = EllipticCurve(GF(257^2), [0,1])
sage: phi = E.isogeny(E.lift_x(240))
sage: phi.degree()
43
sage: phi.scaling_factor()
1
sage: phi.dual().scaling_factor()
43
```

ALGORITHM: The "inner" isogeny is normalized by construction, so we only need to account for the scaling factors of a pre- and post-isomorphism.

x_rational_map()

Return the rational map giving the x-coordinate of this isogeny.

Note: This function returns the x-coordinate component of the isogeny as a rational function in $F(x)$, where F is the base field. To obtain both coordinate functions as elements of the function field $F(x, y)$ in two variables, use *rational_maps()*.

EXAMPLES:

```
sage: E = EllipticCurve(QQ, [0,2,0,1,-1])
sage: phi = EllipticCurveIsogeny(E, [1])
sage: phi.x_rational_map()
x

sage: E = EllipticCurve(GF(17), [0,0,0,3,0])
sage: phi = EllipticCurveIsogeny(E, E((0,0)))
sage: phi.x_rational_map()
(x^2 + 3)/x
```

sage.schemes.elliptic_curves.ell_curve_isogeny.**compute_codomain_formula**(*E*, *v*, *w*)

Compute the codomain curve given parameters v and w (as in Vélu/Kohel/etc. formulas).

INPUT:

- `E` – an elliptic curve
- `v`, `w` – elements of the base field of `E`

OUTPUT:

The elliptic curve with invariants $[a_1, a_2, a_3, a_4 - 5v, a_6 - (a_1^2 + 4a_2)v - 7w]$ where $E = [a_1, a_2, a_3, a_4, a_6]$.

EXAMPLES:

This formula is used by every invocation of the *EllipticCurveIsogeny* constructor:

```
sage: E = EllipticCurve(GF(19), [1,2,3,4,5])
sage: phi = EllipticCurveIsogeny(E, E((1,2)) )
sage: phi.codomain()
Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 9*x + 13
 over Finite Field of size 19
sage: from sage.schemes.elliptic_curves.ell_curve_isogeny import compute_codomain_
↪formula
sage: v = phi._EllipticCurveIsogeny__v
sage: w = phi._EllipticCurveIsogeny__w
sage: compute_codomain_formula(E, v, w) == phi.codomain()
True
```

sage.schemes.elliptic_curves.ell_curve_isogeny.**compute_codomain_kohel**(*E*, *kernel*)

Compute the codomain from the kernel polynomial using Kohel's formulas.

INPUT:

- `E` – domain elliptic curve
- `kernel` (polynomial or list) – the kernel polynomial, or a list of its coefficients

OUTPUT:

(elliptic curve) The codomain elliptic curve of the isogeny defined by `kernel`.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_curve_isogeny import compute_codomain_
↪kohel
sage: E = EllipticCurve(GF(19), [1,2,3,4,5])
sage: phi = EllipticCurveIsogeny(E, [9,1])
sage: phi.codomain() == isogeny_codomain_from_kernel(E, [9,1])
True
sage: compute_codomain_kohel(E, [9,1])
Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 9*x + 8
 over Finite Field of size 19
sage: R.<x> = GF(19)[]
sage: E = EllipticCurve(GF(19), [18,17,16,15,14])
sage: phi = EllipticCurveIsogeny(E, x^3 + 14*x^2 + 3*x + 11)
sage: phi.codomain() == isogeny_codomain_from_kernel(E, x^3 + 14*x^2 + 3*x + 11)
True
sage: compute_codomain_kohel(E, x^3 + 14*x^2 + 3*x + 11)
Elliptic Curve defined by y^2 + 18*x*y + 16*y = x^3 + 17*x^2 + 18*x + 18
 over Finite Field of size 19
sage: E = EllipticCurve(GF(19), [1,2,3,4,5])
```

(continues on next page)

(continued from previous page)

```
sage: phi = EllipticCurveIsogeny(E, x^3 + 7*x^2 + 15*x + 12)
sage: isogeny_codomain_from_kernel(E, x^3 + 7*x^2 + 15*x + 12) == phi.codomain()
True
sage: compute_codomain_kohel(E, x^3 + 7*x^2 + 15*x + 12)
Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 3*x + 15
 over Finite Field of size 19
```

ALGORITHM:

This function uses the formulas of Section 2.4 of [Koh1996].

sage.schemes.elliptic_curves.ell_curve_isogeny.**compute_intermediate_curves**(*E1*, *E2*)

Return intermediate curves and isomorphisms.

Note: This is used to compute $\wp$ functions from the short Weierstrass model more easily.

Warning: The base field must be of characteristic not equal to 2 or 3.

INPUT:

- `E1`, `E2` – elliptic curves

OUTPUT:

A tuple (`pre_isomorphism`, `post_isomorphism`, `intermediate_domain`, `intermediate_codomain`) where:

- `intermediate_domain` is a short Weierstrass curve isomorphic to `E1`;
- `intermediate_codomain` is a short Weierstrass curve isomorphic to `E2`;
- `pre_isomorphism` is a normalized isomorphism from `E1` to `intermediate_domain`;
- `post_isomorphism` is a normalized isomorphism from `intermediate_codomain` to `E2`.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_curve_isogeny import compute_
↪intermediate_curves
sage: E = EllipticCurve(GF(83), [1,0,1,1,0])
sage: R.<x> = GF(83)[]; f = x + 24
sage: phi = EllipticCurveIsogeny(E, f)
sage: E2 = phi.codomain()
sage: compute_intermediate_curves(E, E2)
(Elliptic Curve defined by y^2 = x^3 + 62*x + 74 over Finite Field of size 83,
 Elliptic Curve defined by y^2 = x^3 + 65*x + 69 over Finite Field of size 83,
 Elliptic-curve morphism:
  From: Elliptic Curve defined by y^2 + x*y + y = x^3 + x
        over Finite Field of size 83
  To:   Elliptic Curve defined by y^2 = x^3 + 62*x + 74
        over Finite Field of size 83
  Via:  (u,r,s,t) = (1, 76, 41, 3),
 Elliptic-curve morphism:
  From: Elliptic Curve defined by y^2 = x^3 + 65*x + 69
        over Finite Field of size 83
```

(continues on next page)

(continued from previous page)

```
  To:   Elliptic Curve defined by y^2 + x*y + y = x^3 + 4*x + 16
        over Finite Field of size 83
  Via:  (u,r,s,t) = (1, 7, 42, 42))

sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: K.<i> = NumberField(x^2 + 1)
sage: E = EllipticCurve(K, [0,0,0,1,0])
sage: E2 = EllipticCurve(K, [0,0,0,16,0])
sage: compute_intermediate_curves(E, E2)
(Elliptic Curve defined by y^2 = x^3 + x
  over Number Field in i with defining polynomial x^2 + 1,
 Elliptic Curve defined by y^2 = x^3 + 16*x
  over Number Field in i with defining polynomial x^2 + 1,
 Elliptic-curve endomorphism of Elliptic Curve defined by y^2 = x^3 + x
  over Number Field in i with defining polynomial x^2 + 1
  Via:  (u,r,s,t) = (1, 0, 0, 0),
 Elliptic-curve endomorphism of Elliptic Curve defined by y^2 = x^3 + 16*x
  over Number Field in i with defining polynomial x^2 + 1
  Via:  (u,r,s,t) = (1, 0, 0, 0))
```

sage.schemes.elliptic_curves.ell_curve_isogeny.**compute_isogeny_bmss**(*E1*, *E2*, *l*)

Compute the kernel polynomial of the unique normalized isogeny of degree `l` between `E1` and `E2`.

Both curves must be given in short Weierstrass form, and the characteristic must be either 0 or no smaller than $4l + 4$.

ALGORITHM: [BMSS2006], algorithm *fastElkies'*.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_curve_isogeny import compute_isogeny_
↪bmss
sage: E1 = EllipticCurve(GF(167), [153, 112])
sage: E2 = EllipticCurve(GF(167), [56, 40])
sage: compute_isogeny_bmss(E1, E2, 13)
x^6 + 139*x^5 + 73*x^4 + 139*x^3 + 120*x^2 + 88*x
```

sage.schemes.elliptic_curves.ell_curve_isogeny.**compute_isogeny_kernel_polynomial**(*E1*, *E2*, *ell*, *algorithm=None*)

Return the kernel polynomial of a cyclic, separable, normalized isogeny of degree `ell` from `E1` to `E2`.

INPUT:

- `E1` – domain elliptic curve in short Weierstrass form
- `E2` – codomain elliptic curve in short Weierstrass form
- `ell` – the degree of an isogeny from `E1` to `E2`
- **`algorithm` – `None` (default, choose automatically) or** `"bmss"` (*compute_isogeny_bmss()*) or `"stark"` (*compute_isogeny_stark()*)

OUTPUT:

The kernel polynomial of an isogeny from `E1` to `E2`.

Note: If there is no degree-`ell`, cyclic, separable, normalized isogeny from `E1` to `E2`, a `ValueError` will be raised.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_curve_isogeny import compute_isogeny_
↪kernel_polynomial

sage: E = EllipticCurve(GF(37), [0,0,0,1,8])
sage: R.<x> = GF(37)[]
sage: f = (x + 14) * (x + 30)
sage: phi = EllipticCurveIsogeny(E, f)
sage: E2 = phi.codomain()
sage: compute_isogeny_kernel_polynomial(E, E2, 5)
x^2 + 7*x + 13
sage: f
x^2 + 7*x + 13

sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: K.<i> = NumberField(x^2 + 1)
sage: E = EllipticCurve(K, [0,0,0,1,0])
sage: E2 = EllipticCurve(K, [0,0,0,16,0])
sage: compute_isogeny_kernel_polynomial(E, E2, 4)
x^3 + x
```

sage.schemes.elliptic_curves.ell_curve_isogeny.**compute_isogeny_stark**(*E1*, *E2*, *ell*)

Return the kernel polynomial of an isogeny of degree `ell` from `E1` to `E2`.

INPUT:

- `E1` – domain elliptic curve in short Weierstrass form
- `E2` – codomain elliptic curve in short Weierstrass form
- `ell` – the degree of an isogeny from `E1` to `E2`

OUTPUT:

The kernel polynomial of an isogeny from `E1` to `E2`.

Note: If there is no degree-`ell`, cyclic, separable, normalized isogeny from `E1` to `E2`, a `ValueError` will be raised.

ALGORITHM:

This function uses Stark's algorithm as presented in Section 6.2 of [BMSS2006].

Note: As published in [BMSS2006], the algorithm is incorrect, and a correct version (with slightly different notation) can be found in [Mo2009]. The algorithm originates in [Sta1973].

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_curve_isogeny import compute_isogeny_
↪stark, compute_sequence_of_maps
```

(continues on next page)

(continued from previous page)

```
sage: E = EllipticCurve(GF(97), [1,0,1,1,0])
sage: R.<x> = GF(97)[]; f = x^5 + 27*x^4 + 61*x^3 + 58*x^2 + 28*x + 21
sage: phi = EllipticCurveIsogeny(E, f)
sage: E2 = phi.codomain()
sage: isom1, isom2, E1pr, E2pr, ker_poly = compute_sequence_of_maps(E, E2, 11)
sage: compute_isogeny_stark(E1pr, E2pr, 11)
x^10 + 37*x^9 + 53*x^8 + 66*x^7 + 66*x^6 + 17*x^5 + 57*x^4 + 6*x^3 + 89*x^2 +␣
↪53*x + 8

sage: E = EllipticCurve(GF(37), [0,0,0,1,8])
sage: R.<x> = GF(37)[]
sage: f = (x + 14) * (x + 30)
sage: phi = EllipticCurveIsogeny(E, f)
sage: E2 = phi.codomain()
sage: compute_isogeny_stark(E, E2, 5)
x^4 + 14*x^3 + x^2 + 34*x + 21
sage: f**2
x^4 + 14*x^3 + x^2 + 34*x + 21

sage: E = EllipticCurve(QQ, [0,0,0,1,0])
sage: R.<x> = QQ[]
sage: f = x
sage: phi = EllipticCurveIsogeny(E, f)
sage: E2 = phi.codomain()
sage: compute_isogeny_stark(E, E2, 2)
x
```

sage.schemes.elliptic_curves.ell_curve_isogeny.**compute_sequence_of_maps**(*E1*, *E2*, *ell*)

Return intermediate curves, isomorphisms and kernel polynomial.

INPUT:

- `E1`, `E2` – elliptic curves
- `ell` – a prime such that there is a degree-`ell` separable normalized isogeny from `E1` to `E2`

OUTPUT:

A tuple (`pre_isom`, `post_isom`, `E1pr`, `E2pr`, `ker_poly`) where:

- `E1pr` is an elliptic curve in short Weierstrass form isomorphic to `E1`;
- `E2pr` is an elliptic curve in short Weierstrass form isomorphic to `E2`;
- `pre_isom` is a normalized isomorphism from `E1` to `E1pr`;
- `post_isom` is a normalized isomorphism from `E2pr` to `E2`;
- `ker_poly` is the kernel polynomial of an `ell`-isogeny from `E1pr` to `E2pr`.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_curve_isogeny import compute_sequence_
↪of_maps
sage: E = EllipticCurve('11a1')
sage: R.<x> = QQ[]; f = x^2 - 21*x + 80
sage: phi = EllipticCurveIsogeny(E, f)
sage: E2 = phi.codomain()
```

(continues on next page)

(continued from previous page)

```
sage: compute_sequence_of_maps(E, E2, 5)
(Elliptic-curve morphism:
  From: Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20
        over Rational Field
  To:   Elliptic Curve defined by y^2 = x^3 - 31/3*x - 2501/108
        over Rational Field
  Via:  (u,r,s,t) = (1, 1/3, 0, -1/2),
 Elliptic-curve morphism:
  From: Elliptic Curve defined by y^2 = x^3 - 23461/3*x - 28748141/108
        over Rational Field
  To:   Elliptic Curve defined by y^2 + y = x^3 - x^2 - 7820*x - 263580
        over Rational Field
  Via:  (u,r,s,t) = (1, -1/3, 0, 1/2),
 Elliptic Curve defined by y^2 = x^3 - 31/3*x - 2501/108 over Rational Field,
 Elliptic Curve defined by y^2 = x^3 - 23461/3*x - 28748141/108 over Rational␣
↪Field,
 x^2 - 61/3*x + 658/9)

sage: # needs sage.rings.number_field
sage: K.<i> = NumberField(x^2 + 1)
sage: E = EllipticCurve(K, [0,0,0,1,0])
sage: E2 = EllipticCurve(K, [0,0,0,16,0])
sage: compute_sequence_of_maps(E, E2, 4)
(Elliptic-curve endomorphism of Elliptic Curve defined by y^2 = x^3 + x
  over Number Field in i with defining polynomial x^2 + 1
  Via:  (u,r,s,t) = (1, 0, 0, 0),
 Elliptic-curve endomorphism of Elliptic Curve defined by y^2 = x^3 + 16*x
  over Number Field in i with defining polynomial x^2 + 1
  Via:  (u,r,s,t) = (1, 0, 0, 0),
 Elliptic Curve defined by y^2 = x^3 + x
  over Number Field in i with defining polynomial x^2 + 1,
 Elliptic Curve defined by y^2 = x^3 + 16*x
  over Number Field in i with defining polynomial x^2 + 1,
 x^3 + x)

sage: E = EllipticCurve(GF(97), [1,0,1,1,0])
sage: R.<x> = GF(97)[]; f = x^5 + 27*x^4 + 61*x^3 + 58*x^2 + 28*x + 21
sage: phi = EllipticCurveIsogeny(E, f)
sage: E2 = phi.codomain()
sage: compute_sequence_of_maps(E, E2, 11)
(Elliptic-curve morphism:
  From: Elliptic Curve defined by y^2 + x*y + y = x^3 + x
        over Finite Field of size 97
  To:   Elliptic Curve defined by y^2 = x^3 + 52*x + 31
        over Finite Field of size 97
  Via:  (u,r,s,t) = (1, 8, 48, 44),
 Elliptic-curve morphism:
  From: Elliptic Curve defined by y^2 = x^3 + 41*x + 66
        over Finite Field of size 97
  To:   Elliptic Curve defined by y^2 + x*y + y = x^3 + 87*x + 26
        over Finite Field of size 97
  Via:  (u,r,s,t) = (1, 89, 49, 49),
 Elliptic Curve defined by y^2 = x^3 + 52*x + 31 over Finite Field of size 97,
 Elliptic Curve defined by y^2 = x^3 + 41*x + 66 over Finite Field of size 97,
 x^5 + 67*x^4 + 13*x^3 + 35*x^2 + 77*x + 69)
```

sage.schemes.elliptic_curves.ell_curve_isogeny.**compute_vw_kohel_even_deg1**(*x0*, *y0*, *a1*, *a2*, *a4*)

Compute Vélu's (v, w) using Kohel's formulas for isogenies of degree exactly divisible by 2.

INPUT:

- `x0`, `y0` – coordinates of a 2-torsion point on an elliptic curve E
- `a1`, `a2`, `a4` – invariants of E

OUTPUT:

(tuple) Vélu's isogeny parameters (v, w).

EXAMPLES:

This function will be implicitly called by the following example:

```
sage: E = EllipticCurve(GF(19), [1,2,3,4,5])
sage: phi = EllipticCurveIsogeny(E, [9,1]); phi
Isogeny of degree 2
 from Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5
      over Finite Field of size 19
   to Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 9*x + 8
      over Finite Field of size 19
sage: from sage.schemes.elliptic_curves.ell_curve_isogeny import compute_vw_kohel_
↪even_deg1
sage: a1,a2,a3,a4,a6 = E.a_invariants()
sage: x0 = -9
sage: y0 = -(a1*x0 + a3)/2
sage: compute_vw_kohel_even_deg1(x0, y0, a1, a2, a4)
(18, 9)
```

sage.schemes.elliptic_curves.ell_curve_isogeny.**compute_vw_kohel_even_deg3**(*b2*, *b4*, *s1*, *s2*, *s3*)

Compute Vélu's (v, w) using Kohel's formulas for isogenies of degree divisible by 4.

INPUT:

- `b2`, `b4` – invariants of an elliptic curve E
- `s1`, `s2`, `s3` – signed coefficients of the 2-division polynomial of E

OUTPUT:

(tuple) Vélu's isogeny parameters (v, w).

EXAMPLES:

This function will be implicitly called by the following example:

```
sage: E = EllipticCurve(GF(19), [1,2,3,4,5])
sage: R.<x> = GF(19)[]
sage: phi = EllipticCurveIsogeny(E, x^3 + 7*x^2 + 15*x + 12); phi
Isogeny of degree 4
 from Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5
      over Finite Field of size 19
```

(continues on next page)

(continued from previous page)

```
   to Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 3*x + 15
      over Finite Field of size 19
sage: from sage.schemes.elliptic_curves.ell_curve_isogeny import compute_vw_kohel_
↪even_deg3
sage: b2,b4 = E.b2(), E.b4()
sage: s1, s2, s3 = -7, 15, -12
sage: compute_vw_kohel_even_deg3(b2, b4, s1, s2, s3)
(4, 7)
```

sage.schemes.elliptic_curves.ell_curve_isogeny.**compute_vw_kohel_odd**(*b2*, *b4*, *b6*, *s1*, *s2*, *s3*, *n*)

Compute Vélu's (v, w) using Kohel's formulas for isogenies of odd degree.

INPUT:

- `b2`, `b4`, `b6` – invariants of an elliptic curve E
- `s1`, `s2`, `s3` – signed coefficients of lowest powers of x in the kernel polynomial
- `n` (integer) – the degree

OUTPUT:

(tuple) Vélu's isogeny parameters (v, w).

EXAMPLES:

This function will be implicitly called by the following example:

```
sage: E = EllipticCurve(GF(19), [18,17,16,15,14])
sage: R.<x> = GF(19)[]
sage: phi = EllipticCurveIsogeny(E, x^3 + 14*x^2 + 3*x + 11); phi
Isogeny of degree 7
 from Elliptic Curve defined by y^2 + 18*x*y + 16*y = x^3 + 17*x^2 + 15*x + 14
      over Finite Field of size 19
   to Elliptic Curve defined by y^2 + 18*x*y + 16*y = x^3 + 17*x^2 + 18*x + 18
      over Finite Field of size 19
sage: from sage.schemes.elliptic_curves.ell_curve_isogeny import compute_vw_kohel_
↪odd
sage: b2,b4,b6 = E.b2(), E.b4(), E.b6()
sage: s1,s2,s3 = -14,3,-11
sage: compute_vw_kohel_odd(b2,b4,b6,s1,s2,s3,3)
(7, 1)
```

sage.schemes.elliptic_curves.ell_curve_isogeny.**fill_isogeny_matrix**(*M*)

Return a filled isogeny matrix giving all degrees from one giving only prime degrees.

INPUT:

- `M` – a square symmetric matrix whose off-diagonal i, j entry is either a prime l if the i'th and j'th curves have an l-isogeny between them, otherwise 0

OUTPUT:

(matrix) A square matrix with entries 1 on the diagonal, and in general the i, j entry is $d > 0$ if d is the minimal degree of an isogeny from the i'th to the j'th curve.

EXAMPLES:

```
sage: M = Matrix([[0, 2, 3, 3, 0, 0], [2, 0, 0, 0, 3, 3], [3, 0, 0, 0, 2, 0],
....:             [3, 0, 0, 0, 0, 2], [0, 3, 2, 0, 0, 0], [0, 3, 0, 2, 0, 0]]); M
[0 2 3 3 0 0]
[2 0 0 0 3 3]
[3 0 0 0 2 0]
[3 0 0 0 0 2]
[0 3 2 0 0 0]
[0 3 0 2 0 0]
sage: from sage.schemes.elliptic_curves.ell_curve_isogeny import fill_isogeny_
↪matrix
sage: fill_isogeny_matrix(M)
[ 1  2  3  3  6  6]
[ 2  1  6  6  3  3]
[ 3  6  1  9  2 18]
[ 3  6  9  1 18  2]
[ 6  3  2 18  1  9]
[ 6  3 18  2  9  1]
```

sage.schemes.elliptic_curves.ell_curve_isogeny.**isogeny_codomain_from_kernel**(*E*, *kernel*)

Compute the isogeny codomain given a kernel.

INPUT:

- `E` – domain elliptic curve
- **`kernel` – either a list of points in the kernel of the isogeny,**
  or a kernel polynomial (specified as either a univariate polynomial or a coefficient list)

OUTPUT:

(elliptic curve) The codomain of the separable normalized isogeny defined by this kernel.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_curve_isogeny import isogeny_codomain_
↪from_kernel
sage: E = EllipticCurve(GF(7), [1,0,1,0,1])
sage: R.<x> = GF(7)[]
sage: isogeny_codomain_from_kernel(E, [4,1])
Elliptic Curve defined by y^2 + x*y + y = x^3 + 4*x + 6
 over Finite Field of size 7
sage: (EllipticCurveIsogeny(E, [4,1]).codomain()
....:  == isogeny_codomain_from_kernel(E, [4,1]))
True
sage: isogeny_codomain_from_kernel(E, x^3 + x^2 + 4*x + 3)
Elliptic Curve defined by y^2 + x*y + y = x^3 + 4*x + 6
 over Finite Field of size 7
sage: isogeny_codomain_from_kernel(E, x^3 + 2*x^2 + 4*x + 3)
Elliptic Curve defined by y^2 + x*y + y = x^3 + 5*x + 2
 over Finite Field of size 7

sage: # needs sage.rings.finite_rings
sage: E = EllipticCurve(GF(19), [1,2,3,4,5])
sage: kernel_list = [E((15,10)), E((10,3)), E((6,5))]
sage: isogeny_codomain_from_kernel(E, kernel_list)
Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 3*x + 15
 over Finite Field of size 19
```

sage.schemes.elliptic_curves.ell_curve_isogeny.**two_torsion_part**(*E*, *psi*)

Return the greatest common divisor of `psi` and the 2-torsion polynomial of E.

INPUT:

- `E` – an elliptic curve
- `psi` – a univariate polynomial over the base field of `E`

OUTPUT:

(polynomial) The gcd of `psi` and the 2-torsion polynomial of `E`.

EXAMPLES:

Every function that computes the kernel polynomial via Kohel's formulas will call this function:

```
sage: E = EllipticCurve(GF(19), [1,2,3,4,5])
sage: R.<x> = GF(19)[]
sage: phi = EllipticCurveIsogeny(E, x + 13)
sage: isogeny_codomain_from_kernel(E, x + 13) == phi.codomain()
True
sage: from sage.schemes.elliptic_curves.ell_curve_isogeny import two_torsion_part
sage: two_torsion_part(E, x + 13)
x + 13
```

sage.schemes.elliptic_curves.ell_curve_isogeny.**unfill_isogeny_matrix**(*M*)

Reverses the action of `fill_isogeny_matrix`.

INPUT:

- `M` – a square symmetric matrix of integers

OUTPUT:

(matrix) A square symmetric matrix obtained from `M` by replacing non-prime entries with 0.

EXAMPLES:

```
sage: M = Matrix([[0, 2, 3, 3, 0, 0], [2, 0, 0, 0, 3, 3], [3, 0, 0, 0, 2, 0],
....:             [3, 0, 0, 0, 0, 2], [0, 3, 2, 0, 0, 0], [0, 3, 0, 2, 0, 0]]); M
[0 2 3 3 0 0]
[2 0 0 0 3 3]
[3 0 0 0 2 0]
[3 0 0 0 0 2]
[0 3 2 0 0 0]
[0 3 0 2 0 0]
sage: from sage.schemes.elliptic_curves.ell_curve_isogeny import fill_isogeny_
↪matrix, unfill_isogeny_matrix
sage: M1 = fill_isogeny_matrix(M); M1
[ 1  2  3  3  6  6]
[ 2  1  6  6  3  3]
[ 3  6  1  9  2 18]
[ 3  6  9  1 18  2]
[ 6  3  2 18  1  9]
[ 6  3 18  2  9  1]
sage: unfill_isogeny_matrix(M1)
[0 2 3 3 0 0]
[2 0 0 0 3 3]
[3 0 0 0 2 0]
[3 0 0 0 0 2]
[0 3 2 0 0 0]
```

(continues on next page)

(continued from previous page)

```
[0 3 0 2 0 0]
sage: unfill_isogeny_matrix(M1) == M
True
```

CHAPTER

THIRTEEN

SQUARE-ROOT VÉLU ALGORITHM FOR ELLIPTIC-CURVE ISOGENIES

The square-root Vélu algorithm, also called the √élu algorithm, computes isogenies of elliptic curves in time $\tilde{O}(\sqrt{\ell})$ rather than naïvely $O(\ell)$, where ℓ is the degree.

The core idea is to reindex the points in the kernel subgroup in a baby-step-giant-step manner, then use fast resultant computations to evaluate "elliptic polynomials" (see *FastEllipticPolynomial*) in essentially square-root time.

Based on experiments with Sage version 9.7, the isogeny degree where *EllipticCurveHom_velusqrt* begins to outperform *EllipticCurveIsogeny* can be as low as ≈ 100, but is typically closer to ≈ 1000, depending on the exact situation.

REFERENCES: [BDLS2020]

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.hom_velusqrt import EllipticCurveHom_velusqrt
sage: E = EllipticCurve(GF(6666679), [5,5])
sage: K = E(9970, 1003793, 1)
sage: K.order()
10009
sage: phi = EllipticCurveHom_velusqrt(E, K)
sage: phi
Elliptic-curve isogeny (using square-root Vélu) of degree 10009:
  From: Elliptic Curve defined by y^2 = x^3 + 5*x + 5 over Finite Field of size␣
↪6666679
  To:   Elliptic Curve defined by y^2 = x^3 + 227975*x + 3596133 over Finite Field of␣
↪size 6666679
sage: phi.codomain()
Elliptic Curve defined by y^2 = x^3 + 227975*x + 3596133 over Finite Field of size␣
↪6666679
```

Note that the isogeny is usually not identical to the one computed by *EllipticCurveIsogeny*:

```
sage: psi = EllipticCurveIsogeny(E, K)
sage: psi
Isogeny of degree 10009
    from Elliptic Curve defined by y^2 = x^3 + 5*x + 5 over Finite Field of size␣
↪6666679
    to Elliptic Curve defined by y^2 = x^3 + 5344836*x + 3950273 over Finite Field of␣
↪size 6666679
```

However, they are certainly separable isogenies with the same kernel and must therefore be equal *up to post-isomorphism*:

```
sage: isos = psi.codomain().isomorphisms(phi.codomain())
sage: sum(iso * psi == phi for iso in isos)
1
```

Just like *EllipticCurveIsogeny*, the constructor supports a `model` keyword argument:

```
sage: E = EllipticCurve(GF(6666679), [1,1])
sage: K = E(9091, 517864)
sage: phi = EllipticCurveHom_velusqrt(E, K, model='montgomery')
sage: phi
Elliptic-curve isogeny (using square-root Vélu) of degree 2999:
  From: Elliptic Curve defined by y^2 = x^3 + x + 1 over Finite Field of size 6666679
  To:   Elliptic Curve defined by y^2 = x^3 + 1559358*x^2 + x over Finite Field of␣
↪size 6666679
```

Internally, *EllipticCurveHom_velusqrt* works on short Weierstraß curves, but it performs the conversion automatically:

```
sage: E = EllipticCurve(GF(101), [1,2,3,4,5])
sage: K = E(1, 2)
sage: K.order()
37
sage: EllipticCurveHom_velusqrt(E, K)
Elliptic-curve isogeny (using square-root Vélu) of degree 37:
  From: Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5 over Finite␣
↪Field of size 101
  To:   Elliptic Curve defined by y^2 = x^3 + 66*x + 86 over Finite Field of size 101
```

However, this does imply not all elliptic curves are supported. Curves without a short Weierstraß model exist in characteristics 2 and 3:

```
sage: F.<t> = GF(3^3)
sage: E = EllipticCurve(F, [1,1,1,1,1])
sage: P = E(t^2+2, 1)
sage: P.order()
19
sage: EllipticCurveHom_velusqrt(E, P)
Traceback (most recent call last):
...
NotImplementedError: only implemented for curves having a short Weierstrass model
```

Furthermore, the implementation is restricted to finite fields, since this appears to be the most relevant application for the square-root Vélu algorithm:

```
sage: E = EllipticCurve('26b1')
sage: P = E(1,0)
sage: P.order()
7
sage: EllipticCurveHom_velusqrt(E, P)
Traceback (most recent call last):
...
NotImplementedError: only implemented for elliptic curves over finite fields
```

Note: Some of the methods inherited from *EllipticCurveHom* compute data whose size is linear in the degree; this includes kernel polynomial and rational maps. In consequence, those methods cannot possibly run in the otherwise

advertised square-root complexity, as merely storing the result already takes linear time.

AUTHORS:

- Lorenz Panny (2022)

class sage.schemes.elliptic_curves.hom_velusqrt.**EllipticCurveHom_velusqrt**(*E*, *P*, *, *codomain=None*, *model=None*, *Q=None*)

Bases: *EllipticCurveHom*

This class implements separable odd-degree isogenies of elliptic curves over finite fields using the square-root Vélu algorithm.

The complexity is $\tilde{O}(\sqrt{\ell})$ base-field operations, where ℓ is the degree.

REFERENCES: [BDLS2020]

INPUT:

- `E` – an elliptic curve over a finite field
- `P` – a point on E of odd order ≥ 9
- `codomain` – codomain elliptic curve (optional)
- `model` – string (optional); input to *compute_model()*
- `Q` – a point on E outside $\langle P \rangle$, or `None`

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.hom_velusqrt import EllipticCurveHom_
↪velusqrt
sage: F.<t> = GF(10009^3)
sage: E = EllipticCurve(F, [t,t])
sage: K = E(2154*t^2 + 5711*t + 2899, 7340*t^2 + 4653*t + 6935)
sage: phi = EllipticCurveHom_velusqrt(E, K); phi
Elliptic-curve isogeny (using square-root Vélu) of degree 601:
  From: Elliptic Curve defined by y^2 = x^3 + t*x + t over Finite Field in t of␣
↪size 10009^3
  To:   Elliptic Curve defined by y^2 = x^3 + (263*t^2+3173*t+4759)*x + (3898*t^
↪2+6111*t+9443) over Finite Field in t of size 10009^3
sage: phi(K)
(0 : 1 : 0)
sage: P = E(2, 3163*t^2 + 7293*t + 5999)
sage: phi(P)
(6085*t^2 + 855*t + 8720 : 8078*t^2 + 9889*t + 6030 : 1)
sage: Q = E(6, 5575*t^2 + 6607*t + 9991)
sage: phi(Q)
(626*t^2 + 9749*t + 1291 : 5931*t^2 + 8549*t + 3111 : 1)
sage: phi(P + Q)
(983*t^2 + 4894*t + 4072 : 5047*t^2 + 9325*t + 336 : 1)
sage: phi(P) + phi(Q)
(983*t^2 + 4894*t + 4072 : 5047*t^2 + 9325*t + 336 : 1)
```

See also:

EllipticCurveIsogeny

dual()

Return the dual of this square-root Vélu isogeny as an *EllipticCurveHom*.

Note: The dual is computed by *EllipticCurveIsogeny*, hence it does not benefit from the square-root Vélu speedup.

EXAMPLES:

```
sage: E = EllipticCurve(GF(101^2), [1, 1, 1, 1, 1])
sage: K = E.cardinality() // 11 * E.gens()[0]
sage: phi = E.isogeny(K, algorithm='velusqrt'); phi
Elliptic-curve isogeny (using square-root Vélu) of degree 11:
  From: Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 + x + 1 over␣
↪Finite Field in z2 of size 101^2
  To:   Elliptic Curve defined by y^2 = x^3 + 39*x + 40 over Finite Field in␣
↪z2 of size 101^2
sage: phi.dual()
Isogeny of degree 11 from Elliptic Curve defined by y^2 = x^3 + 39*x + 40␣
↪over Finite Field in z2 of size 101^2 to Elliptic Curve defined by y^2 +␣
↪x*y + y = x^3 + x^2 + x + 1 over Finite Field in z2 of size 101^2
sage: phi.dual() * phi == phi.domain().scalar_multiplication(11)
True
sage: phi * phi.dual() == phi.codomain().scalar_multiplication(11)
True
```

inseparable_degree()

Return the inseparable degree of this square-root Vélu isogeny.

Since *EllipticCurveHom_velusqrt* only implements separable isogenies, this method always returns one.

kernel_polynomial()

Return the kernel polynomial of this square-root Vélu isogeny.

Note: The data returned by this method has size linear in the degree.

EXAMPLES:

```
sage: E = EllipticCurve(GF(65537^2,'a'), [5,5])
sage: K = E.cardinality()//31 * E.gens()[0]
sage: phi = E.isogeny(K, algorithm='velusqrt')
sage: h = phi.kernel_polynomial(); h
x^15 + 21562*x^14 + 8571*x^13 + 20029*x^12 + 1775*x^11 + 60402*x^10 + 17481*x^
↪9 + 46543*x^8 + 46519*x^7 + 18590*x^6 + 36554*x^5 + 36499*x^4 + 48857*x^3 +␣
↪3066*x^2 + 23264*x + 53937
sage: h == E.isogeny(K).kernel_polynomial()
True
sage: h(K.x())
0
```

rational_maps()

Return the pair of explicit rational maps of this square-root Vélu isogeny as fractions of bivariate polynomials in x and y.

Note: The data returned by this method has size linear in the degree.

EXAMPLES:

```
sage: E = EllipticCurve(GF(101^2), [1, 1, 1, 1, 1])
sage: K = (E.cardinality() // 11) * E.gens()[0]
sage: phi = E.isogeny(K, algorithm='velusqrt'); phi
Elliptic-curve isogeny (using square-root Vélu) of degree 11:
  From: Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 + x + 1 over␣
↪Finite Field in z2 of size 101^2
  To:   Elliptic Curve defined by y^2 = x^3 + 39*x + 40 over Finite Field in␣
↪z2 of size 101^2
sage: phi.rational_maps()
((-17*x^11 - 34*x^10 - 36*x^9 + ... - 29*x^2 - 25*x - 25)/(x^10 + 10*x^9 +␣
↪19*x^8 - ... + x^2 + 47*x + 24),
 (-3*x^16 - 6*x^15*y - 48*x^15 + ... - 49*x - 9*y + 46)/(x^15 + 15*x^14 -␣
↪35*x^13 - ... + 3*x^2 - 45*x + 47))
```

scaling_factor()

Return the Weierstrass scaling factor associated to this square-root Vélu isogeny.

The scaling factor is the constant u (in the base field) such that $\varphi^*\omega_2 = u\omega_1$, where $\varphi : E_1 \to E_2$ is this isogeny and ω_i are the standard Weierstrass differentials on E_i defined by $\mathrm{d}x/(2y + a_1x + a_3)$.

EXAMPLES:

```
sage: E = EllipticCurve(GF(101^2), [1, 1, 1, 1, 1])
sage: K = (E.cardinality() // 11) * E.gens()[0]
sage: phi = E.isogeny(K, algorithm='velusqrt', model='montgomery'); phi
Elliptic-curve isogeny (using square-root Vélu) of degree 11:
  From: Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 + x + 1 over␣
↪Finite Field in z2 of size 101^2
  To:   Elliptic Curve defined by y^2 = x^3 + 61*x^2 + x over Finite Field in␣
↪z2 of size 101^2
sage: phi.scaling_factor()
55
sage: phi.scaling_factor() == phi.formal()[1]
True
```

x_rational_map()

Return the x-coordinate rational map of this square-root Vélu isogeny as a univariate rational function in x.

Note: The data returned by this method has size linear in the degree.

EXAMPLES:

```
sage: E = EllipticCurve(GF(101^2), [1, 1, 1, 1, 1])
sage: K = (E.cardinality() // 11) * E.gens()[0]
sage: phi = E.isogeny(K, algorithm='velusqrt'); phi
Elliptic-curve isogeny (using square-root Vélu) of degree 11:
  From: Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 + x + 1 over␣
↪Finite Field in z2 of size 101^2
  To:   Elliptic Curve defined by y^2 = x^3 + 39*x + 40 over Finite Field in␣
↪z2 of size 101^2
sage: phi.x_rational_map()
```

(continues on next page)

(continued from previous page)

```
(84*x^11 + 67*x^10 + 65*x^9 + ... + 72*x^2 + 76*x + 76)/(x^10 + 10*x^9 + 19*x^
↪8 + ... + x^2 + 47*x + 24)
sage: phi.x_rational_map() == phi.rational_maps()[0]
True
```

class sage.schemes.elliptic_curves.hom_velusqrt.**FastEllipticPolynomial**(*E*, *n*, *P*, *Q=None*)

Bases: `object`

A class to represent and evaluate an *elliptic polynomial*, and optionally its derivative, in essentially square-root time.

The elliptic polynomials computed by this class are of the form

$$h_S(Z) = \prod_{i \in S} (Z - x(Q + [i]P))$$

where P is a point of odd order $n \geq 5$ and Q is either `None`, in which case it is assumed to be ∞, or an arbitrary point which is not a multiple of P.

The index set S is chosen as follows:

- If Q is given, then $S = \{0, 1, 2, 3, ..., n-1\}$.
- If Q is omitted, then $S = \{1, 3, 5, ..., n-2\}$. Note that in this case, $h_{\{1,2,3,...,n-1\}}$ can be computed as h_S^2 since n is odd.

INPUT:

- `E` – an elliptic curve in short Weierstraß form
- `n` – an odd integer ≥ 5
- `P` – a point on E
- `Q` – a point on E, or `None`

ALGORITHM: [BDLS2020], Algorithm 2

Note: Currently only implemented for short Weierstraß curves.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.hom_velusqrt import FastEllipticPolynomial
sage: E = EllipticCurve(GF(71), [5,5])
sage: P = E(4, 35)
sage: hP = FastEllipticPolynomial(E, P.order(), P); hP
Fast elliptic polynomial prod(Z - x(i*P) for i in range(1,n,2)) with n = 19, P =␣
↪(4 : 35 : 1)
sage: hP(7)
19
sage: prod(7 - (i*P).x() for i in range(1,P.order(),2))
19
```

Passing Q changes the index set:

```
sage: Q = E(0, 17)
sage: hPQ = FastEllipticPolynomial(E, P.order(), P, Q)
sage: hPQ(7)
58
```

(continues on next page)

(continued from previous page)

```
sage: prod(7 - (Q+i*P).x() for i in range(P.order()))
58
```

The call syntax has an optional keyword argument `derivative`, which will make the function return the pair $(h_S(\alpha), h'_S(\alpha))$ instead of just $h_S(\alpha)$:

```
sage: hP(7, derivative=True)
(19, 15)
sage: R.<Z> = E.base_field()[]
sage: HP = prod(Z - (i*P).x() for i in range(1,P.order(),2))
sage: HP
Z^9 + 16*Z^8 + 57*Z^7 + 6*Z^6 + 45*Z^5 + 31*Z^4 + 46*Z^3 + 10*Z^2 + 28*Z + 41
sage: HP(7)
19
sage: HP.derivative()(7)
15
```

```
sage: hPQ(7, derivative=True)
(58, 62)
sage: R.<Z> = E.base_field()[]
sage: HPQ = prod(Z - (Q+i*P).x() for i in range(P.order()))
sage: HPQ
Z^19 + 53*Z^18 + 67*Z^17 + 39*Z^16 + 56*Z^15 + 32*Z^14 + 44*Z^13 + 6*Z^12 + 27*Z^
↪11 + 29*Z^10 + 38*Z^9 + 48*Z^8 + 38*Z^7 + 43*Z^6 + 21*Z^5 + 25*Z^4 + 33*Z^3 +␣
↪49*Z^2 + 60*Z
sage: HPQ(7)
58
sage: HPQ.derivative()(7)
62
```

The input can be an element of any algebra over the base ring:

```
sage: R.<T> = GF(71)[]
sage: S.<t> = R.quotient(T^2)
sage: hP(7 + t)
15*t + 19
```

CHAPTER

FOURTEEN

SCALAR-MULTIPLICATION MORPHISMS OF ELLIPTIC CURVES

This class provides an *EllipticCurveHom* instantiation for multiplication-by-m maps on elliptic curves.

EXAMPLES:

We can construct and evaluate scalar multiplications:

```
sage: from sage.schemes.elliptic_curves.hom_scalar import EllipticCurveHom_scalar
sage: E = EllipticCurve('77a1')
sage: phi = E.scalar_multiplication(5); phi
Scalar-multiplication endomorphism [5] of Elliptic Curve defined by y^2 + y = x^3 +␣
↪2*x over Rational Field
sage: P = E(2,3)
sage: phi(P)
(30 : 164 : 1)
```

The usual *EllipticCurveHom* methods are supported:

```
sage: phi.degree()
25
sage: phi.kernel_polynomial()
x^12 + 124/5*x^10 + 19*x^9 - 84*x^8 + 24*x^7 - 483*x^6 - 696/5*x^5 - 448*x^4 - 37*x^3␣
↪- 332*x^2 - 84*x + 47/5
sage: phi.rational_maps()
((x^25 - 200*x^23 - 520*x^22 + 9000*x^21 + ... + 1377010*x^3 + 20360*x^2 - 39480*x +␣
↪2209),
 (10*x^36*y - 620*x^36 + 3240*x^34*y - 44880*x^34 + ... + 424927560*x*y + 226380480*x␣
↪+ 42986410*y + 20974090)/(1250*x^36 + 93000*x^34 + 71250*x^33 + 1991400*x^32 + ...␣
↪+ 1212964050*x^3 + 138715800*x^2 - 27833400*x + 1038230))
sage: phi.dual()
Scalar-multiplication endomorphism [5] of Elliptic Curve defined by y^2 + y = x^3 +␣
↪2*x over Rational Field
sage: phi.dual() is phi
True
sage: phi.formal()
5*t - 310*t^4 - 2496*t^5 + 10540*t^7 + ... - 38140146674516*t^20 - 46800256902400*t^
↪21 + 522178541079910*t^22 + O(t^23)
sage: phi.is_normalized()
False
sage: phi.is_separable()
True
sage: phi.is_injective()
False
sage: phi.is_surjective()
True
```

Contrary to constructing an *EllipticCurveIsogeny* from the division polynomial, *EllipticCurveHom_scalar* can deal with huge scalars very quickly:

```
sage: E = EllipticCurve(GF(2^127-1), [1,2,3,4,5])
sage: phi = E.scalar_multiplication(9^99); phi
Scalar-multiplication endomorphism␣
↪[295126654306527521487534802261977363143592725170438328860638846376769434334780203327094110048891␣
↪of Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5 over Finite␣
↪Field of size 170141183460469231731687303715884105727
sage: phi(E(1,2))
(82124533143060719620799539030695848450 : 170160220386248146557226821340214023 79 : 1)
```

Composition of scalar multiplications results in another scalar multiplication:

```
sage: E = EllipticCurve(GF(19), [4,4])
sage: phi = E.scalar_multiplication(-3); phi
Scalar-multiplication endomorphism [-3] of Elliptic Curve defined by y^2 = x^3 + 4*x␣
↪+ 4 over Finite Field of size 19
sage: psi = E.scalar_multiplication(7); psi
Scalar-multiplication endomorphism [7] of Elliptic Curve defined by y^2 = x^3 + 4*x +␣
↪4 over Finite Field of size 19
sage: phi * psi
Scalar-multiplication endomorphism [-21] of Elliptic Curve defined by y^2 = x^3 + 4*x␣
↪+ 4 over Finite Field of size 19
sage: psi * phi
Scalar-multiplication endomorphism [-21] of Elliptic Curve defined by y^2 = x^3 + 4*x␣
↪+ 4 over Finite Field of size 19
sage: phi * psi == psi * phi
True
sage: -phi == E.scalar_multiplication(-1) * phi
True
```

The zero endomorphism $[0]$ is supported:

```
sage: E = EllipticCurve(GF(71), [1,1])
sage: zero = E.scalar_multiplication(0); zero
Scalar-multiplication endomorphism [0] of Elliptic Curve defined by y^2 = x^3 + x + 1␣
↪over Finite Field of size 71
sage: zero.is_zero()
True
sage: zero.is_injective()
False
sage: zero.is_surjective()
False
sage: zero(E.random_point())
(0 : 1 : 0)
```

Retrieving multiplication-by-m maps when m is divisible by the characteristic also works (since Issue #37096):

```
sage: E = EllipticCurve(GF(7), [1,0])
sage: phi = E.scalar_multiplication(7); phi
Scalar-multiplication endomorphism [7] of Elliptic Curve defined by y^2 = x^3 + x␣
↪over Finite Field of size 7
sage: phi.rational_maps()
(x^49, -y^49)
sage: phi.x_rational_map()
x^49
sage: psi = E.scalar_multiplication(-2); psi
```

(continues on next page)

(continued from previous page)

```
Scalar-multiplication endomorphism [-2] of Elliptic Curve defined by y^2 = x^3 + x␣
↪over Finite Field of size 7
sage: chi = E.scalar_multiplication(-14); chi
Scalar-multiplication endomorphism [-14] of Elliptic Curve defined by y^2 = x^3 + x␣
↪over Finite Field of size 7
sage: chi == psi * phi
True
sage: chi.rational_maps()
((x^196 - 2*x^98 + 1)/(-3*x^147 - 3*x^49),
 (-x^294*y^49 + 2*x^196*y^49 - 2*x^98*y^49 + y^49)/(-x^294 - 2*x^196 - x^98))
sage: chi.x_rational_map()
(2*x^196 + 3*x^98 + 2)/(x^147 + x^49)
sage: chi.rational_maps() == tuple(f(*phi.rational_maps()) for f in psi.rational_
↪maps())
True
sage: chi.x_rational_map() == psi.x_rational_map()(phi.x_rational_map())
True
```

```
sage: E = EllipticCurve(GF(7), [0,1])
sage: phi = E.scalar_multiplication(7); phi
Scalar-multiplication endomorphism [7] of Elliptic Curve defined by y^2 = x^3 + 1␣
↪over Finite Field of size 7
sage: phi.rational_maps()   # known bug -- #6413
((-3*x^49 - x^28 - x^7)/(x^42 - x^21 + 2),
 (-x^72*y - 3*x^69*y - 3*x^66*y - x^63*y + 3*x^51*y + 2*x^48*y + 2*x^45*y + 3*x^42*y -
↪ x^9*y - 3*x^6*y - 3*x^3*y - y)/(x^63 + 2*x^42 - x^21 - 1))
sage: phi.x_rational_map()
(4*x^49 + 6*x^28 + 6*x^7)/(x^42 + 6*x^21 + 2)
```

AUTHORS:

- Lorenz Panny (2021): implement *EllipticCurveHom_scalar*

class sage.schemes.elliptic_curves.hom_scalar.**EllipticCurveHom_scalar**(E, m)

Bases: *EllipticCurveHom*

Construct a scalar-multiplication map on an elliptic curve.

degree()

Return the degree of this scalar-multiplication morphism.

The map $[m]$ has degree m^2.

EXAMPLES:

```
sage: E = EllipticCurve(GF(23), [0,1])
sage: phi = E.scalar_multiplication(1111111)
sage: phi.degree()
1234567654321
```

dual()

Return the dual isogeny of this scalar-multiplication map.

This method simply returns `self` as scalars are self-dual.

EXAMPLES:

```
sage: E = EllipticCurve([5,5])
sage: phi = E.scalar_multiplication(5)
sage: phi.dual() is phi
True
```

inseparable_degree()

Return the inseparable degree of this scalar-multiplication map.

EXAMPLES:

```
sage: E = EllipticCurve(GF(7), [0,1])
sage: E.is_supersingular()
False
sage: E.scalar_multiplication(4).inseparable_degree()
1
sage: E.scalar_multiplication(-7).inseparable_degree()
7
```

```
sage: E = EllipticCurve(GF(7), [1,0])
sage: E.is_supersingular()
True
sage: E.scalar_multiplication(4).inseparable_degree()
1
sage: E.scalar_multiplication(-7).inseparable_degree()
49
```

kernel_polynomial()

Return the kernel polynomial of this scalar-multiplication map. (When $m = 0$, return 0.)

EXAMPLES:

```
sage: E = EllipticCurve(GF(997), [7,7,7,7,7])
sage: phi = E.scalar_multiplication(5)
sage: phi.kernel_polynomial()
x^12 + 77*x^11 + 380*x^10 + 198*x^9 + 840*x^8 + 376*x^7 + 946*x^6 + 848*x^5 +␣
↪246*x^4 + 778*x^3 + 77*x^2 + 518*x + 28
```

```
sage: E = EllipticCurve(GF(997), [5,6,7,8,9])
sage: phi = E.scalar_multiplication(11)
sage: phi.kernel_polynomial()
x^60 + 245*x^59 + 353*x^58 + 693*x^57 + 499*x^56 + 462*x^55 + 820*x^54 +␣
↪962*x^53 + ... + 736*x^7 + 939*x^6 + 429*x^5 + 267*x^4 + 116*x^3 + 770*x^2␣
↪+ 491*x + 519
```

rational_maps()

Return the pair of explicit rational maps defining this scalar multiplication.

ALGORITHM: *EllipticCurve_generic.multiplication_by_m()*

EXAMPLES:

```
sage: E = EllipticCurve('77a1')
sage: phi = E.scalar_multiplication(5)
sage: phi.rational_maps()
((x^25 - 200*x^23 - 520*x^22 + ... + 368660*x^2 + 163195*x + 16456)/(25*x^24␣
↪+ 1240*x^22 + 950*x^21 + ... + 20360*x^2 - 39480*x + 2209),
 (10*x^36*y - 620*x^36 + 3240*x^34*y - ... + 226380480*x + 42986410*y +␣
```

(continues on next page)

(continued from previous page)

```
↪20974090)/(1250*x^36 + 93000*x^34 + 71250*x^33 + ... + 138715800*x^2 -␣
↪27833400*x + 1038230))
sage: P = (2,3)
sage: Q = tuple(r(P) for r in phi.rational_maps()); Q
(30, 164)
sage: E(Q) == 5*E(P)
True
```

scaling_factor()

Return the Weierstrass scaling factor associated to this scalar multiplication.

The scaling factor is the constant u (in the base field) such that $\varphi^*\omega_2 = u\omega_1$, where $\varphi : E_1 \to E_2$ is this morphism and ω_i are the standard Weierstrass differentials on E_i defined by $\mathrm{d}x/(2y + a_1x + a_3)$.

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: phi = E.scalar_multiplication(5)
sage: u = phi.scaling_factor()
sage: u == phi.formal()[1]
True
sage: u == E.multiplication_by_m_isogeny(5).scaling_factor()
doctest:warning ... DeprecationWarning: ...
True
```

The scaling factor lives in the base ring:

```
sage: E = EllipticCurve(GF(101^2), [5,5])
sage: phi = E.scalar_multiplication(123)
sage: phi.scaling_factor()
22
sage: phi.scaling_factor().parent()
Finite Field in z2 of size 101^2
```

ALGORITHM: The scaling factor equals the scalar that is being multiplied by.

x_rational_map()

Return the x-coordinate rational map of this scalar multiplication.

ALGORITHM: *EllipticCurve_generic.multiplication_by_m()*

EXAMPLES:

```
sage: E = EllipticCurve(GF(65537), [1,2,3,4,5])
sage: phi = E.scalar_multiplication(7)
sage: phi.x_rational_map() == phi.rational_maps()[0]
True
```

CHAPTER

FIFTEEN

FROBENIUS ISOGENIES OF ELLIPTIC CURVES

Frobenius isogenies only exist in positive characteristic p. They are given by $\pi_n : (x, y) \mapsto (x^{p^n}, y^{p^n})$.

This class implements π_n for $n \geq 0$. Together with existing tools for composing isogenies (see *EllipticCurveHom_composite*), we can therefore represent arbitrary inseparable isogenies in Sage.

EXAMPLES:

Constructing a Frobenius isogeny is straightforward:

```
sage: from sage.schemes.elliptic_curves.hom_frobenius import EllipticCurveHom_
↪frobenius
sage: z5, = GF(17^5).gens()
sage: E = EllipticCurve([z5,1])
sage: pi = EllipticCurveHom_frobenius(E); pi
Frobenius isogeny of degree 17:
  From: Elliptic Curve defined by y^2 = x^3 + z5*x + 1
        over Finite Field in z5 of size 17^5
  To:   Elliptic Curve defined by y^2 = x^3 + (9*z5^4+7*z5^3+10*z5^2+z5+14)*x + 1
        over Finite Field in z5 of size 17^5
```

By passing n, we can also construct higher-power Frobenius maps, such as the Frobenius *endo*morphism:

```
sage: z5, = GF(7^5).gens()
sage: E = EllipticCurve([z5,1])
sage: pi = EllipticCurveHom_frobenius(E, 5); pi
Frobenius endomorphism of degree 16807 = 7^5:
  From: Elliptic Curve defined by y^2 = x^3 + z5*x + 1
        over Finite Field in z5 of size 7^5
  To:   Elliptic Curve defined by y^2 = x^3 + z5*x + 1
        over Finite Field in z5 of size 7^5
```

The usual *EllipticCurveHom* methods are supported:

```
sage: z5, = GF(7^5).gens()
sage: E = EllipticCurve([z5,1])
sage: pi = EllipticCurveHom_frobenius(E,5)
sage: pi.degree()
16807
sage: pi.rational_maps()
(x^16807, y^16807)
sage: pi.formal()                    # known bug
...
sage: pi.is_normalized()             # known bug
...
sage: pi.is_separable()
```

(continues on next page)

(continued from previous page)

```
False
sage: pi.is_injective()
True
sage: pi.is_surjective()
True
```

Computing the dual of Frobenius is supported as well:

```
sage: E = EllipticCurve([GF(17^6).gen(), 0])
sage: pi = EllipticCurveHom_frobenius(E)
sage: pihat = pi.dual(); pihat
Isogeny of degree 17
 from Elliptic Curve defined by y^2 = x^3 + (15*z6^5+5*z6^4+8*z6^3+12*z6^2+11*z6+7)*x
      over Finite Field in z6 of size 17^6
   to Elliptic Curve defined by y^2 = x^3 + z6*x
      over Finite Field in z6 of size 17^6
sage: pihat.is_separable()
True
sage: pihat * pi == EllipticCurveHom_scalar(E,17)    # known bug -- #6413
True
```

A supersingular example (with purely inseparable dual):

```
sage: E = EllipticCurve([0, GF(17^6).gen()])
sage: E.is_supersingular()
True
sage: pi1 = EllipticCurveHom_frobenius(E)
sage: pi1hat = pi1.dual(); pi1hat
Composite morphism of degree 17 = 17*1:
  From: Elliptic Curve defined by y^2 = x^3 + (15*z6^5+5*z6^4+8*z6^3+12*z6^2+11*z6+7)
        over Finite Field in z6 of size 17^6
  To:   Elliptic Curve defined by y^2 = x^3 + z6
        over Finite Field in z6 of size 17^6
sage: pi6 = EllipticCurveHom_frobenius(E,6)
sage: pi6hat = pi6.dual(); pi6hat
Composite morphism of degree 24137569 = 24137569*1:
  From: Elliptic Curve defined by y^2 = x^3 + z6
        over Finite Field in z6 of size 17^6
  To:   Elliptic Curve defined by y^2 = x^3 + z6
        over Finite Field in z6 of size 17^6
sage: pi6hat.factors()
(Frobenius endomorphism of degree 24137569 = 17^6:
   From: Elliptic Curve defined by y^2 = x^3 + z6
         over Finite Field in z6 of size 17^6
   To:   Elliptic Curve defined by y^2 = x^3 + z6
         over Finite Field in z6 of size 17^6,
 Elliptic-curve endomorphism of
  Elliptic Curve defined by y^2 = x^3 + z6 over Finite Field in z6 of size 17^6
   Via:  (u,r,s,t) = (2*z6^5 + 10*z6^3 + z6^2 + 8, 0, 0, 0))
```

AUTHORS:

- Lorenz Panny (2021): implement *EllipticCurveHom_frobenius*
- Mickaël Montessinos (2021): computing the dual of a Frobenius isogeny

class sage.schemes.elliptic_curves.hom_frobenius.**EllipticCurveHom_frobenius**(*E*, *power=1*)

Bases: *EllipticCurveHom*

Construct a Frobenius isogeny on a given curve with a given power of the base-ring characteristic.

Writing n for the parameter `power` (default: 1), the isogeny is defined by $(x, y) \to (x^{p^n}, y^{p^n})$ where p is the characteristic of the base ring.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.hom_frobenius import EllipticCurveHom_
↪frobenius
sage: E = EllipticCurve(j=GF(11^2).gen())
sage: EllipticCurveHom_frobenius(E)
Frobenius isogeny of degree 11:
  From: Elliptic Curve defined by y^2 = x^3 + (2*z2+6)*x + (8*z2+8) over Finite␣
↪Field in z2 of size 11^2
  To:   Elliptic Curve defined by y^2 = x^3 + (9*z2+3)*x + (3*z2+7) over Finite␣
↪Field in z2 of size 11^2
sage: EllipticCurveHom_frobenius(E, 2)
Frobenius endomorphism of degree 121 = 11^2:
  From: Elliptic Curve defined by y^2 = x^3 + (2*z2+6)*x + (8*z2+8) over Finite␣
↪Field in z2 of size 11^2
  To:   Elliptic Curve defined by y^2 = x^3 + (2*z2+6)*x + (8*z2+8) over Finite␣
↪Field in z2 of size 11^2
```

dual()

Compute the dual of this Frobenius isogeny.

This method returns an *EllipticCurveHom* object.

EXAMPLES:

An ordinary example:

```
sage: from sage.schemes.elliptic_curves.hom_scalar import EllipticCurveHom_
↪scalar
sage: from sage.schemes.elliptic_curves.hom_frobenius import EllipticCurveHom_
↪frobenius
sage: E = EllipticCurve(GF(31), [0,1])
sage: f = EllipticCurveHom_frobenius(E)
sage: f.dual() * f == EllipticCurveHom_scalar(f.domain(), 31)
True
sage: f * f.dual() == EllipticCurveHom_scalar(f.codomain(), 31)
True
```

A supersingular example:

```
sage: E = EllipticCurve(GF(31), [1,0])
sage: f = EllipticCurveHom_frobenius(E)
sage: f.dual() * f == EllipticCurveHom_scalar(f.domain(), 31)
True
sage: f * f.dual() == EllipticCurveHom_scalar(f.codomain(), 31)
True
```

ALGORITHM:

- For supersingular curves, the dual of Frobenius is again purely inseparable, so we start out with a Frobenius isogeny of equal degree in the opposite direction.

- For ordinary curves, we immediately reduce to the case of prime degree. The kernel of the dual is the unique subgroup of size p, which we compute from the p-division polynomial.

In both cases, we then search for the correct post-isomorphism using *find_post_isomorphism()*.

inseparable_degree()

Return the inseparable degree of this Frobenius isogeny.

Since this class implements only purely inseparable isogenies, the inseparable degree equals the degree.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.hom_frobenius import EllipticCurveHom_
↪frobenius
sage: E = EllipticCurve(GF(11), [1,1])
sage: pi = EllipticCurveHom_frobenius(E, 4)
sage: pi.inseparable_degree()
14641
sage: pi.inseparable_degree() == pi.degree()
True
```

kernel_polynomial()

Return the kernel polynomial of this Frobenius isogeny as a polynomial in x. This method always returns 1.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.hom_frobenius import EllipticCurveHom_
↪frobenius
sage: E = EllipticCurve(GF(11), [1,1])
sage: pi = EllipticCurveHom_frobenius(E, 5)
sage: pi.kernel_polynomial()
1
```

rational_maps()

Return the explicit rational maps defining this Frobenius isogeny as (sparse) bivariate rational maps in x and y.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.hom_frobenius import EllipticCurveHom_
↪frobenius
sage: E = EllipticCurve(GF(11), [1,1])
sage: pi = EllipticCurveHom_frobenius(E, 4)
sage: pi.rational_maps()
(x^14641, y^14641)
```

scaling_factor()

Return the Weierstrass scaling factor associated to this Frobenius morphism.

The scaling factor is the constant u (in the base field) such that $\varphi^*\omega_2 = u\omega_1$, where $\varphi : E_1 \to E_2$ is this morphism and ω_i are the standard Weierstrass differentials on E_i defined by $\mathrm{d}x/(2y + a_1x + a_3)$.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.hom_frobenius import EllipticCurveHom_
↪frobenius
sage: E = EllipticCurve(GF(11), [1,1])
sage: pi = EllipticCurveHom_frobenius(E)
sage: pi.formal()
```

(continues on next page)

(continued from previous page)

```
t^11 + O(t^33)
sage: pi.scaling_factor()
0
sage: pi = EllipticCurveHom_frobenius(E, 3)
sage: pi.formal()
t^1331 + O(t^1353)
sage: pi.scaling_factor()
0
sage: pi = EllipticCurveHom_frobenius(E, 0)
sage: pi == E.scalar_multiplication(1)
True
sage: pi.scaling_factor()
1
```

The scaling factor lives in the base ring:

```
sage: pi.scaling_factor().parent()
Finite Field of size 11
```

ALGORITHM: Inseparable isogenies of degree > 1 have scaling factor 0.

x_rational_map()

Return the x-coordinate rational map of this Frobenius isogeny as a (sparse) univariate rational map in x.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.hom_frobenius import EllipticCurveHom_
↪frobenius
sage: E = EllipticCurve(GF(11), [1,1])
sage: pi = EllipticCurveHom_frobenius(E, 4)
sage: pi.x_rational_map()
x^14641
```

CHAPTER

SIXTEEN

ISOGENIES OF SMALL PRIME DEGREE

Functions for the computation of isogenies of small primes degree. First: l = 2, 3, 5, 7, or 13, where the modular curve $X_0(l)$ has genus 0. Second: l = 11, 17, 19, 23, 29, 31, 41, 47, 59, or 71, where $X_0^+(l)$ has genus 0 and $X_0(l)$ is elliptic or hyperelliptic. Also: l = 11, 17, 19, 37, 43, 67 or 163 over $\mathbf{Q}$ (the sporadic cases with only finitely many j-invariants each). All the above only require factorization of a polynomial of degree $l + 1$. Finally, a generic function which works for arbitrary odd primes l (including the characteristic), but requires factorization of the l-division polynomial, of degree $(l^2 - 1)/2$.

AUTHORS:

- John Cremona and Jenny Cooley: 2009-07..11: the genus 0 cases the sporadic cases over $\mathbf{Q}$.
- Kimi Tsukazaki and John Cremona: 2013-07: The 10 (hyper)-elliptic cases and the generic algorithm. See [KT2013].

sage.schemes.elliptic_curves.isogeny_small_degree.**Fricke_module**()

Fricke module for `l` =2,3,5,7,13.

For these primes (and these only) the modular curve $X_0(l)$ has genus zero, and its field is generated by a single modular function called the Fricke module (or Hauptmodul), t. There is a classical choice of such a generator t in each case, and the j-function is a rational function of t of degree $l + 1$ of the form $P(t)/t$ where P is a polynomial of degree $l + 1$. Up to scaling, t is determined by the condition that the ramification points above $j = \infty$ are $t = 0$ (with ramification degree 1) and $t = \infty$ (with degree l). The ramification above $j = 0$ and $j = 1728$ may be seen in the factorizations of $j(t)$ and $k(t)$ where $k = j - 1728$.

OUTPUT:

The rational function $P(t)/t$.

sage.schemes.elliptic_curves.isogeny_small_degree.**Fricke_polynomial**()

Fricke polynomial for `l` =2,3,5,7,13.

For these primes (and these only) the modular curve $X_0(l)$ has genus zero, and its field is generated by a single modular function called the Fricke module (or Hauptmodul), t. There is a classical choice of such a generator t in each case, and the j-function is a rational function of t of degree $l + 1$ of the form $P(t)/t$ where P is a polynomial of degree $l + 1$. Up to scaling, t is determined by the condition that the ramification points above $j = \infty$ are $t = 0$ (with ramification degree 1) and $t = \infty$ (with degree l). The ramification above $j = 0$ and $j = 1728$ may be seen in the factorizations of $j(t)$ and $k(t)$ where $k = j - 1728$.

OUTPUT:

The polynomial $P(t)$ as an element of $\mathbf{Z}[t]$.

sage.schemes.elliptic_curves.isogeny_small_degree.**Psi**(*use_stored=True*)

Generic kernel polynomial for genus zero primes.

For each of the primes l for which $X_0(l)$ has genus zero (namely $l = 2, 3, 5, 7, 13$), we may define an elliptic curve E_t over $\mathbf{Q}(t)$, with coefficients in $\mathbf{Z}[t]$, which has good reduction except at $t = 0$ and $t = \infty$ (which lie above

$j = \infty$) and at certain other values of t above $j = 0$ when $l = 3$ (one value) or $l \equiv 1 \pmod 3$ (two values) and above $j = 1728$ when $l = 2$ (one value) or $l \equiv 1 \pmod 4$ (two values). (These exceptional values correspond to endomorphisms of E_t of degree l.) The l-division polynomial of E_t has a unique factor of degree $(l-1)/2$ (or 1 when $l = 2$), with coefficients in $\mathbf{Z}[t]$, which we call the Generic Kernel Polynomial for l. These are used, by specialising t, in the function *isogenies_prime_degree_genus_0()*, which also has to take into account the twisting factor between E_t for a specific value of t and the short Weierstrass form of an elliptic curve with j-invariant $j(t)$. This enables the computation of the kernel polynomials of isogenies without having to compute and factor division polynomials.

All of this data is quickly computed from the Fricke modules, except that for $l = 13$ the factorization of the Generic Division Polynomial takes a long time, so the value have been precomputed and cached; by default the cached values are used, but the code here will recompute them when `use_stored` is `False`, as in the doctests.

INPUT:

- `l` – either 2, 3, 5, 7, or 13.
- `use_stored` (boolean, default: `True`) – If True, use precomputed values, otherwise compute them on the fly.

Note: This computation takes a negligible time for $l = 2, 3, 5, 7$ but more than 100s for $l = 13$. The reason for allowing dynamic computation here instead of just using precomputed values is for testing.

`sage.schemes.elliptic_curves.isogeny_small_degree.`**`Psi2`**`()`

Return the generic kernel polynomial for hyperelliptic l-isogenies.

INPUT:

- `l` – either 11, 17, 19, 23, 29, 31, 41, 47, 59, or 71.

OUTPUT:

The generic l-kernel polynomial.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.isogeny_small_degree import Psi2
sage: Psi2(11)
x^5 - 55*x^4*u + 994*x^3*u^2 - 8774*x^2*u^3 + 41453*x*u^4 - 928945/11*u^5
+ 33*x^4 + 276*x^3*u - 7794*x^2*u^2 + 4452*x*u^3 + 1319331/11*u^4 + 216*x^3*v
- 4536*x^2*u*v + 31752*x*u^2*v - 842616/11*u^3*v + 162*x^3 + 38718*x^2*u
- 610578*x*u^2 + 33434694/11*u^3 - 4536*x^2*v + 73872*x*u*v - 2745576/11*u^2*v
- 16470*x^2 + 580068*x*u - 67821354/11*u^2 - 185976*x*v + 14143896/11*u*v
+ 7533*x - 20437029/11*u - 12389112/11*v + 19964151/11
sage: p = Psi2(71)                          # long time
sage: (x,u,v) = p.variables()               # long time
sage: p.coefficient({x: 0, u: 210, v: 0}) # long time
-22093807117225051795062587395152885841161472373938152664680764365 21/71
sage: p.coefficient({x: 0, u: 0, v: 0})   # long time
-14790739586438315394567393301990769678157425619440464678252277649/71
```

`sage.schemes.elliptic_curves.isogeny_small_degree.`**`is_kernel_polynomial`**`(`E, m, f`)`

Test whether `E` has a cyclic isogeny of degree `m` with kernel polynomial `f`.

INPUT:

- `E` – an elliptic curve.
- `m` – a positive integer.

- `f` – a polynomial over the base field of `E`.

OUTPUT:

(bool) `True` if `E` has a cyclic isogeny of degree `m` with kernel polynomial `f`, else `False`.

ALGORITHM:

f must have degree $(m-1)/2$ (if m is odd) or degree $m/2$ (if m is even), and have the property that for each root x of f, $\mu(x)$ is also a root where μ is the multiplication-by-m map on E and m runs over a set of generators of $(\mathbf{Z}/m\mathbf{Z})^*/\{1,-1\}$.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.isogeny_small_degree import is_kernel_
↪polynomial
sage: E = EllipticCurve([0, -1, 1, -10, -20])
sage: x = polygen(QQ)
sage: is_kernel_polynomial(E, 5, x^2 + x - 29/5)
True
sage: is_kernel_polynomial(E, 5, (x - 16) * (x - 5))
True
```

An example from [KT2013], where the 13-division polynomial splits into 14 factors each of degree 6, but only two of these is a kernel polynomial for a 13-isogeny:

```
sage: F = GF(3)
sage: E = EllipticCurve(F, [0,0,0,-1,0])
sage: f13 = E.division_polynomial(13)
sage: factors = [f for f, e in f13.factor()]
sage: all(f.degree() == 6 for f in factors)
True
sage: [is_kernel_polynomial(E, 13, f) for f in factors]
[True,
True,
False,
False,
False,
False,
False,
False,
False,
False,
False,
False,
False,
False]
```

See Issue #22232:

```
sage: # needs sage.rings.finite_rings
sage: K = GF(47^2)
sage: E = EllipticCurve([0, K.gen()])
sage: psi7 = E.division_polynomial(7)
sage: f = psi7.factor()[4][0]
sage: f
x^3 + (7*z2 + 11)*x^2 + (25*z2 + 33)*x + 25*z2
sage: f.divides(psi7)
True
```

(continues on next page)

(continued from previous page)

```
sage: is_kernel_polynomial(E, 7, f)
False
```

sage.schemes.elliptic_curves.isogeny_small_degree.**isogenies_13_0**(*E*, *minimal_models=True*)

Return list of all 13-isogenies from E when the j-invariant is 0.

INPUT:

- `E` – an elliptic curve with j-invariant 0.
- `minimal_models` (bool, default `True`) – if `True`, all curves computed will be minimal or semi-minimal models. Over fields of larger degree it can be expensive to compute these so set to `False`.

OUTPUT:

(list) 13-isogenies with codomain E. In general these are normalised; but if -3 is a square then there are two endomorphisms of degree 13, for which the codomain is the same as the domain.

Note: This implementation requires that the characteristic is not 2, 3 or 13.

Note: This function would normally be invoked indirectly via `E.isogenies_prime_degree(13)`.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.isogeny_small_degree import isogenies_13_0
```

Endomorphisms of degree 13 will exist when -3 is a square:

```
sage: # needs sage.rings.number_field
sage: K.<r> = QuadraticField(-3)
sage: E = EllipticCurve(K, [0, r]); E
Elliptic Curve defined by y^2 = x^3 + r over Number Field in r
 with defining polynomial x^2 + 3 with r = 1.732050807568878?*I
sage: isogenies_13_0(E)
[Isogeny of degree 13
  from Elliptic Curve defined by y^2 = x^3 + r over Number Field in r
       with defining polynomial x^2 + 3 with r = 1.732050807568878?*I
    to Elliptic Curve defined by y^2 = x^3 + r over Number Field in r
       with defining polynomial x^2 + 3 with r = 1.732050807568878?*I,
 Isogeny of degree 13
  from Elliptic Curve defined by y^2 = x^3 + r over Number Field in r
       with defining polynomial x^2 + 3 with r = 1.732050807568878?*I
    to Elliptic Curve defined by y^2 = x^3 + r over Number Field in r
       with defining polynomial x^2 + 3 with r = 1.732050807568878?*I]
sage: isogenies_13_0(E)[0].rational_maps()
(((7/338*r + 23/338)*x^13 + (-164/13*r - 420/13)*x^10
  + (720/13*r + 3168/13)*x^7 + (3840/13*r - 576/13)*x^4
  + (4608/13*r + 2304/13)*x)/(x^12 + (4*r + 36)*x^9 + (1080/13*r + 3816/13)*x^6
                              + (2112/13*r - 5184/13)*x^3 + (-17280/169*r - 1152/
↪169)),
 ((18/2197*r + 35/2197)*x^18*y + (23142/2197*r + 35478/2197)*x^15*y
  + (-1127520/2197*r - 1559664/2197)*x^12*y + (-87744/2197*r + 5992704/2197)*x^9*y
  + (-6625152/2197*r - 9085824/2197)*x^6*y + (-28919808/2197*r - 2239488/2197)*x^
```

(continues on next page)

(continued from previous page)

```
↪3*y
  + (-1990656/2197*r - 3870720/2197)*y)/(x^18 + (6*r + 54)*x^15
        + (3024/13*r + 11808/13)*x^12 + (31296/13*r + 51840/13)*x^9
        + (487296/169*r - 2070144/169)*x^6 + (-940032/169*r + 248832/169)*x^3
        + (1990656/2197*r + 3870720/2197)))
```

An example of endomorphisms over a finite field:

```
sage: # needs sage.rings.finite_rings
sage: K = GF(19^2,'a')
sage: E = EllipticCurve(j=K(0)); E
Elliptic Curve defined by y^2 = x^3 + 1
 over Finite Field in a of size 19^2
sage: isogenies_13_0(E)
[Isogeny of degree 13
  from Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field in a of size 19^2
    to Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field in a of size 19^
↪2,
 Isogeny of degree 13
  from Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field in a of size 19^2
    to Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field in a of size 19^
↪2]
sage: isogenies_13_0(E)[0].rational_maps()
((6*x^13 - 6*x^10 - 3*x^7 + 6*x^4 + x)/(x^12 - 5*x^9 - 9*x^6 - 7*x^3 + 5),
 (-8*x^18*y - 9*x^15*y + 9*x^12*y - 5*x^9*y
  + 5*x^6*y - 7*x^3*y + 7*y)/(x^18 + 2*x^15 + 3*x^12 - x^9 + 8*x^6 - 9*x^3 + 7))
```

A previous implementation did not work in some characteristics:

```
sage: K = GF(29)
sage: E = EllipticCurve(j=K(0))
sage: isogenies_13_0(E)
[Isogeny of degree 13
  from Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field of size 29
    to Elliptic Curve defined by y^2 = x^3 + 26*x + 12 over Finite Field of size␣
↪29,
 Isogeny of degree 13
  from Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field of size 29
    to Elliptic Curve defined by y^2 = x^3 + 16*x + 28 over Finite Field of size␣
↪29]
```

```
sage: K = GF(101)
sage: E = EllipticCurve(j=K(0)); E.ainvs()
(0, 0, 0, 0, 1)
sage: [phi.codomain().ainvs() for phi in isogenies_13_0(E)]
[(0, 0, 0, 64, 36), (0, 0, 0, 42, 66)]
```

```
sage: x = polygen(QQ)
sage: f = x^12 + 78624*x^9 - 130308048*x^6 + 2270840832*x^3 - 54500179968
sage: K.<a> = NumberField(f)                                                #␣
↪needs sage.rings.number_field
sage: E = EllipticCurve(j=K(0)); E.ainvs()                                  #␣
↪needs sage.rings.number_field
(0, 0, 0, 0, 1)
sage: len([phi.codomain().ainvs()       # long time                         #␣
↪needs sage.rings.number_field
```

(continues on next page)

(continued from previous page)

```
....:       for phi in isogenies_13_0(E)])
2
```

sage.schemes.elliptic_curves.isogeny_small_degree.**isogenies_13_1728**(*E*, *minimal_models=True*)

Return list of all 13-isogenies from E when the j-invariant is 1728.

INPUT:

- `E` – an elliptic curve with j-invariant 1728.
- `minimal_models` (bool, default `True`) – if `True`, all curves computed will be minimal or semi-minimal models. Over fields of larger degree it can be expensive to compute these so set to `False`.

OUTPUT:

(list) 13-isogenies with codomain E. In general these are normalised; but if -1 is a square then there are two endomorphisms of degree 13, for which the codomain is the same as the domain; and over $\mathbf{Q}$ or a number field, the codomain is a global minimal model where possible.

Note: This implementation requires that the characteristic is not 2, 3 or 13.

Note: This function would normally be invoked indirectly via `E.isogenies_prime_degree(13)`.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.isogeny_small_degree import isogenies_13_
↪1728

sage: K.<i> = QuadraticField(-1)                                                #␣
↪needs sage.rings.number_field
sage: E = EllipticCurve([0,0,0,i,0]); E.ainvs()                                 #␣
↪needs sage.rings.number_field
(0, 0, 0, i, 0)
sage: isogenies_13_1728(E)                                                      #␣
↪needs sage.rings.number_field
[Isogeny of degree 13
  from Elliptic Curve defined by y^2 = x^3 + i*x over Number Field in i
       with defining polynomial x^2 + 1 with i = 1*I
    to Elliptic Curve defined by y^2 = x^3 + i*x over Number Field in i
       with defining polynomial x^2 + 1 with i = 1*I,
 Isogeny of degree 13
  from Elliptic Curve defined by y^2 = x^3 + i*x over Number Field in i
       with defining polynomial x^2 + 1 with i = 1*I
    to Elliptic Curve defined by y^2 = x^3 + i*x over Number Field in i
       with defining polynomial x^2 + 1 with i = 1*I]
```

```
sage: K = GF(83)
sage: E = EllipticCurve(K, [0,0,0,5,0]); E.ainvs()
(0, 0, 0, 5, 0)
sage: isogenies_13_1728(E)
[]
sage: K = GF(89)
```

(continues on next page)

(continued from previous page)

```
sage: E = EllipticCurve(K, [0,0,0,5,0]); E.ainvs()
(0, 0, 0, 5, 0)
sage: isogenies_13_1728(E)
[Isogeny of degree 13
  from Elliptic Curve defined by y^2 = x^3 + 5*x over Finite Field of size 89
    to Elliptic Curve defined by y^2 = x^3 + 5*x over Finite Field of size 89,
 Isogeny of degree 13
  from Elliptic Curve defined by y^2 = x^3 + 5*x over Finite Field of size 89
    to Elliptic Curve defined by y^2 = x^3 + 5*x over Finite Field of size 89]
```

```
sage: K = GF(23)
sage: E = EllipticCurve(K, [1,0])
sage: isogenies_13_1728(E)
[Isogeny of degree 13
  from Elliptic Curve defined by y^2 = x^3 + x over Finite Field of size 23
    to Elliptic Curve defined by y^2 = x^3 + 16 over Finite Field of size 23,
 Isogeny of degree 13
  from Elliptic Curve defined by y^2 = x^3 + x over Finite Field of size 23
    to Elliptic Curve defined by y^2 = x^3 + 7 over Finite Field of size 23]
```

```
sage: x = polygen(QQ)
sage: f = (x^12 + 1092*x^10 - 432432*x^8 + 6641024*x^6
....:      - 282896640*x^4 - 149879808*x^2 - 349360128)
sage: K.<a> = NumberField(f)                                                #␣
↪needs sage.rings.number_field
sage: E = EllipticCurve(K, [1,0])                                           #␣
↪needs sage.rings.number_field
sage: [phi.codomain().ainvs()           # long time                         #␣
↪needs sage.rings.number_field
....:  for phi in isogenies_13_1728(E)]
[(0,
  0,
  0,
  -4225010072113/3063768069807341568*a^10 - 24841071989413/15957125363579904*a^8
    + 11179537789374271/21276167151439872*a^6 - 407474562289492049/
↪47871376090739712*a^4
    + 1608052769560747/4522994717568*a^2 + 7786720245212809/36937790193472,
  -363594277511/5744565513088876544*a^11 - 7213386922793/2991961005671232*a^9
    - 2810970361185589/1329760446964992*a^7 + 281503836888046601/
↪8975883017013696*a^5
    - 1287313166530075/848061509544*a^3 + 9768837984886039/6925835661276*a),
 (0,
  0,
  0,
  -4225010072113/3063768069807341568*a^10 - 24841071989413/15957125363579904*a^8
    + 11179537789374271/21276167151439872*a^6 - 407474562289492049/
↪47871376090739712*a^4
    + 1608052769560747/4522994717568*a^2 + 7786720245212809/36937790193472,
  363594277511/5744565513088876544*a^11 + 7213386922793/2991961005671232*a^9
    + 2810970361185589/1329760446964992*a^7 - 281503836888046601/
↪8975883017013696*a^5
    + 1287313166530075/848061509544*a^3 - 9768837984886039/6925835661276*a)]
```

sage.schemes.elliptic_curves.isogeny_small_degree.**isogenies_2**(*E*, *minimal_models=True*)

Return a list of all 2-isogenies with domain `E`.

INPUT:

- `E` – an elliptic curve.
- `minimal_models` (bool, default `True`) – if `True`, all curves computed will be minimal or semi-minimal models. Over fields of larger degree it can be expensive to compute these so set to `False`.

OUTPUT:

(list) 2-isogenies with domain `E`. In general these are normalised, but over **Q** and other number fields, the codomain is a minimal model where possible.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.isogeny_small_degree import isogenies_2
sage: E = EllipticCurve('14a1'); E
Elliptic Curve defined by y^2 + x*y + y = x^3 + 4*x - 6 over Rational Field
sage: [phi.codomain().ainvs() for phi in isogenies_2(E)]
[(1, 0, 1, -36, -70)]

sage: E = EllipticCurve([1,2,3,4,5]); E
Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5 over Rational␣
↪Field
sage: [phi.codomain().ainvs() for phi in isogenies_2(E)]
[]
sage: E = EllipticCurve(QQbar, [9,8]); E                                    #␣
↪needs sage.rings.number_field
Elliptic Curve defined by y^2 = x^3 + 9*x + 8 over Algebraic Field
sage: isogenies_2(E)                    # not implemented                   #␣
↪needs sage.rings.number_field
```

sage.schemes.elliptic_curves.isogeny_small_degree.**isogenies_3**(*E*, *minimal_models=True*)

Return a list of all 3-isogenies with domain `E`.

INPUT:

- `E` – an elliptic curve.
- `minimal_models` (bool, default `True`) – if `True`, all curves computed will be minimal or semi-minimal models. Over fields of larger degree it can be expensive to compute these so set to `False`.

OUTPUT:

(list) 3-isogenies with domain `E`. In general these are normalised, but over **Q** or a number field, the codomain is a global minimal model where possible.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.isogeny_small_degree import isogenies_3
sage: E = EllipticCurve(GF(17), [1,1])
sage: [phi.codomain().ainvs() for phi in isogenies_3(E)]
[(0, 0, 0, 9, 7), (0, 0, 0, 0, 1)]

sage: E = EllipticCurve(GF(17^2,'a'), [1,1])                                #␣
↪needs sage.rings.finite_rings
sage: [phi.codomain().ainvs() for phi in isogenies_3(E)]                    #␣
↪needs sage.rings.finite_rings
[(0, 0, 0, 9, 7), (0, 0, 0, 0, 1), (0, 0, 0, 5*a + 1, a + 13), (0, 0, 0, 12*a + 6,
↪ 16*a + 14)]
```

(continues on next page)

(continued from previous page)

```
sage: E = EllipticCurve('19a1')
sage: [phi.codomain().ainvs() for phi in isogenies_3(E)]
[(0, 1, 1, 1, 0), (0, 1, 1, -769, -8470)]

sage: E = EllipticCurve([1,1])
sage: [phi.codomain().ainvs() for phi in isogenies_3(E)]
[]
```

sage.schemes.elliptic_curves.isogeny_small_degree.**isogenies_5_0**(*E*, *minimal_models=True*)

Return a list of all the 5-isogenies with domain `E` when the j-invariant is 0.

INPUT:

- `E` – an elliptic curve with j-invariant 0.
- `minimal_models` (bool, default `True`) – if `True`, all curves computed will be minimal or semi-minimal models. Over fields of larger degree it can be expensive to compute these so set to `False`.

OUTPUT:

(list) 5-isogenies with codomain E. In general these are normalised, but over **Q** or a number field, the codomain is a global minimal model where possible.

Note: This implementation requires that the characteristic is not 2, 3 or 5.

Note: This function would normally be invoked indirectly via `E.isogenies_prime_degree(5)`.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.isogeny_small_degree import isogenies_5_0
sage: E = EllipticCurve([0,12])
sage: isogenies_5_0(E)
[]

sage: E = EllipticCurve(GF(13^2,'a'), [0,-3])                                  #␣
↪needs sage.rings.finite_rings
sage: isogenies_5_0(E)                                                         #␣
↪needs sage.rings.finite_rings
[Isogeny of degree 5
  from Elliptic Curve defined by y^2 = x^3 + 10 over Finite Field in a of size 13^
↪2
    to Elliptic Curve defined by y^2 = x^3 + (4*a+6)*x + (2*a+10)
       over Finite Field in a of size 13^2,
 Isogeny of degree 5
  from Elliptic Curve defined by y^2 = x^3 + 10 over Finite Field in a of size 13^
↪2
    to Elliptic Curve defined by y^2 = x^3 + (12*a+5)*x + (2*a+10)
       over Finite Field in a of size 13^2,
 Isogeny of degree 5
  from Elliptic Curve defined by y^2 = x^3 + 10 over Finite Field in a of size 13^
↪2
    to Elliptic Curve defined by y^2 = x^3 + (10*a+2)*x + (2*a+10)
       over Finite Field in a of size 13^2,
```

(continues on next page)

(continued from previous page)

```
 Isogeny of degree 5
  from Elliptic Curve defined by y^2 = x^3 + 10 over Finite Field in a of size 13^
↪2
    to Elliptic Curve defined by y^2 = x^3 + (3*a+12)*x + (11*a+12)
       over Finite Field in a of size 13^2,
 Isogeny of degree 5
  from Elliptic Curve defined by y^2 = x^3 + 10 over Finite Field in a of size 13^
↪2
    to Elliptic Curve defined by y^2 = x^3 + (a+4)*x + (11*a+12)
       over Finite Field in a of size 13^2,
 Isogeny of degree 5
  from Elliptic Curve defined by y^2 = x^3 + 10 over Finite Field in a of size 13^
↪2
    to Elliptic Curve defined by y^2 = x^3 + (9*a+10)*x + (11*a+12)
       over Finite Field in a of size 13^2]

sage: x = polygen(QQ, 'x')
sage: K.<a> = NumberField(x**6 - 320*x**3 - 320)                                #␣
↪needs sage.rings.number_field
sage: E = EllipticCurve(K, [0,0,1,0,0])                                         #␣
↪needs sage.rings.number_field
sage: isogenies_5_0(E)                                                          #␣
↪needs sage.rings.number_field
[Isogeny of degree 5
  from Elliptic Curve defined by y^2 + y = x^3
       over Number Field in a with defining polynomial x^6 - 320*x^3 - 320
    to Elliptic Curve defined by
       y^2 + y = x^3 + (241565/32*a^5-362149/48*a^4+180281/24*a^3-9693307/4*a^
↪2+14524871/6*a-7254985/3)*x
                  + (1660391123/192*a^5-829315373/96*a^4+77680504/9*a^3-
↪66622345345/24*a^2+33276655441/12*a-24931615912/9)
       over Number Field in a with defining polynomial x^6 - 320*x^3 - 320,
 Isogeny of degree 5
  from Elliptic Curve defined by y^2 + y = x^3
       over Number Field in a with defining polynomial x^6 - 320*x^3 - 320
    to Elliptic Curve defined by
       y^2 + y = x^3 + (47519/32*a^5-72103/48*a^4+32939/24*a^3-1909753/4*a^
↪2+2861549/6*a-1429675/3)*x
                  + (-131678717/192*a^5+65520419/96*a^4-12594215/18*a^
↪3+5280985135/24*a^2-2637787519/12*a+1976130088/9)
       over Number Field in a with defining polynomial x^6 - 320*x^3 - 320]
```

sage.schemes.elliptic_curves.isogeny_small_degree.**isogenies_5_1728**(*E*, *minimal_models=True*)

Return a list of 5-isogenies with domain `E` when the j-invariant is 1728.

INPUT:

- `E` – an elliptic curve with j-invariant 1728.
- `minimal_models` (bool, default `True`) – if `True`, all curves computed will be minimal or semi-minimal models. Over fields of larger degree it can be expensive to compute these so set to `False`.

OUTPUT:

(list) 5-isogenies with codomain E. In general these are normalised; but if -1 is a square then there are two endomorphisms of degree 5, for which the codomain is the same as the domain curve; and over $\mathbf{Q}$ or a number field,

the codomain is a global minimal model where possible.

Note: This implementation requires that the characteristic is not 2, 3 or 5.

Note: This function would normally be invoked indirectly via `E.isogenies_prime_degree(5)`.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.isogeny_small_degree import isogenies_5_
↪1728
sage: E = EllipticCurve([7,0])
sage: isogenies_5_1728(E)
[]

sage: E = EllipticCurve(GF(13), [11,0])
sage: isogenies_5_1728(E)
[Isogeny of degree 5
  from Elliptic Curve defined by y^2 = x^3 + 11*x over Finite Field of size 13
    to Elliptic Curve defined by y^2 = x^3 + 11*x over Finite Field of size 13,
 Isogeny of degree 5
  from Elliptic Curve defined by y^2 = x^3 + 11*x over Finite Field of size 13
    to Elliptic Curve defined by y^2 = x^3 + 11*x over Finite Field of size 13]
```

An example of endomorphisms of degree 5:

```
sage: # needs sage.rings.number_field
sage: K.<i> = QuadraticField(-1)
sage: E = EllipticCurve(K, [0,0,0,1,0])
sage: isogenies_5_1728(E)
[Isogeny of degree 5
  from Elliptic Curve defined by y^2 = x^3 + x over Number Field in i
       with defining polynomial x^2 + 1 with i = 1*I
    to Elliptic Curve defined by y^2 = x^3 + x over Number Field in i
       with defining polynomial x^2 + 1 with i = 1*I,
 Isogeny of degree 5
  from Elliptic Curve defined by y^2 = x^3 + x over Number Field in i
       with defining polynomial x^2 + 1 with i = 1*I
    to Elliptic Curve defined by y^2 = x^3 + x over Number Field in i
       with defining polynomial x^2 + 1 with i = 1*I]
sage: _[0].rational_maps()
(((4/25*i + 3/25)*x^5
  + (4/5*i - 2/5)*x^3 - x)/(x^4 + (-4/5*i + 2/5)*x^2 + (-4/25*i - 3/25)),
 ((11/125*i + 2/125)*x^6*y + (-23/125*i + 64/125)*x^4*y
  + (141/125*i + 162/125)*x^2*y
  + (3/25*i - 4/25)*y)/(x^6 + (-6/5*i + 3/5)*x^4
                        + (-12/25*i - 9/25)*x^2 + (2/125*i - 11/125)))
```

An example of 5-isogenies over a number field:

```
sage: # needs sage.rings.number_field
sage: x = polygen(QQ, 'x')
sage: K.<a> = NumberField(x**4 + 20*x**2 - 80)
sage: K(5).is_square()  # necessary but not sufficient!
True
sage: E = EllipticCurve(K, [0,0,0,1,0])
```

(continues on next page)

(continued from previous page)

```
sage: isogenies_5_1728(E)
[Isogeny of degree 5
  from Elliptic Curve defined by y^2 = x^3 + x
       over Number Field in a with defining polynomial x^4 + 20*x^2 - 80
    to Elliptic Curve defined by y^2 = x^3 + (-753/4*a^2-4399)*x + (2779*a^
↪3+65072*a)
       over Number Field in a with defining polynomial x^4 + 20*x^2 - 80,
 Isogeny of degree 5
  from Elliptic Curve defined by y^2 = x^3 + x
       over Number Field in a with defining polynomial x^4 + 20*x^2 - 80
    to Elliptic Curve defined by y^2 = x^3 + (-753/4*a^2-4399)*x + (-2779*a^3-
↪65072*a)
       over Number Field in a with defining polynomial x^4 + 20*x^2 - 80]
```

See Issue #19840:

```
sage: # needs sage.rings.number_field
sage: K.<a> = NumberField(x^4 - 5*x^2 + 5)
sage: E = EllipticCurve([a^2 + a + 1, a^3 + a^2 + a + 1, a^2 + a,
....:                    17*a^3 + 34*a^2 - 16*a - 37,
....:                    54*a^3 + 105*a^2 - 66*a - 135])
sage: len(E.isogenies_prime_degree(5))
2
sage: from sage.schemes.elliptic_curves.isogeny_small_degree import isogenies_5_
↪1728
sage: [phi.codomain().j_invariant() for phi in isogenies_5_1728(E)]
[19691491018752*a^2 - 27212977933632, 19691491018752*a^2 - 27212977933632]
```

sage.schemes.elliptic_curves.isogeny_small_degree.**isogenies_7_0**(*E*, *minimal_models=True*)

Return list of all 7-isogenies from E when the j-invariant is 0.

INPUT:

- `E` – an elliptic curve with j-invariant 0.
- `minimal_models` (bool, default `True`) – if `True`, all curves computed will be minimal or semi-minimal models. Over fields of larger degree it can be expensive to compute these so set to `False`.

OUTPUT:

(list) 7-isogenies with codomain E. In general these are normalised; but if -3 is a square then there are two endomorphisms of degree 7, for which the codomain is the same as the domain; and over $\mathbf{Q}$ or a number field, the codomain is a global minimal model where possible.

Note: This implementation requires that the characteristic is not 2, 3 or 7.

Note: This function would normally be invoked indirectly via `E.isogenies_prime_degree(7)`.

EXAMPLES:

First some examples of endomorphisms:

```
sage: from sage.schemes.elliptic_curves.isogeny_small_degree import isogenies_7_0
sage: K.<r> = QuadraticField(-3)                                              #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.rings.number_field
sage: E = EllipticCurve(K, [0,1])                                              #␣
↪needs sage.rings.number_field
sage: isogenies_7_0(E)                                                         #␣
↪needs sage.rings.number_field
[Isogeny of degree 7
  from Elliptic Curve defined by y^2 = x^3 + 1 over Number Field in r
       with defining polynomial x^2 + 3 with r = 1.7320508075688?*I
    to Elliptic Curve defined by y^2 = x^3 + 1 over Number Field in r
       with defining polynomial x^2 + 3 with r = 1.7320508075688?*I,
 Isogeny of degree 7
  from Elliptic Curve defined by y^2 = x^3 + 1 over Number Field in r
       with defining polynomial x^2 + 3 with r = 1.7320508075688?*I
    to Elliptic Curve defined by y^2 = x^3 + 1 over Number Field in r
       with defining polynomial x^2 + 3 with r = 1.7320508075688?*I]

sage: E = EllipticCurve(GF(13^2,'a'), [0,-3])                                  #␣
↪needs sage.rings.finite_rings
sage: isogenies_7_0(E)                                                         #␣
↪needs sage.rings.finite_rings
[Isogeny of degree 7
  from Elliptic Curve defined by y^2 = x^3 + 10 over Finite Field in a of size 13^
↪2
    to Elliptic Curve defined by y^2 = x^3 + 10 over Finite Field in a of size 13^
↪2,
 Isogeny of degree 7
  from Elliptic Curve defined by y^2 = x^3 + 10 over Finite Field in a of size 13^
↪2
    to Elliptic Curve defined by y^2 = x^3 + 10 over Finite Field in a of size 13^
↪2]
```

Now some examples of 7-isogenies which are not endomorphisms:

```
sage: K = GF(101)
sage: E = EllipticCurve(K, [0,1])
sage: isogenies_7_0(E)
[Isogeny of degree 7
  from Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field of size 101
    to Elliptic Curve defined by y^2 = x^3 + 55*x + 100 over Finite Field of size␣
↪101,
 Isogeny of degree 7
  from Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field of size 101
  to Elliptic Curve defined by y^2 = x^3 + 83*x + 26 over Finite Field of size␣
↪101]
```

Examples over a number field:

```
sage: from sage.schemes.elliptic_curves.isogeny_small_degree import isogenies_7_0
sage: E = EllipticCurve('27a1').change_ring(QuadraticField(-3,'r'))            #␣
↪needs sage.rings.number_field
sage: isogenies_7_0(E)                                                         #␣
↪needs sage.rings.number_field
[Isogeny of degree 7
  from Elliptic Curve defined by y^2 + y = x^3 + (-7) over Number Field in r
       with defining polynomial x^2 + 3 with r = 1.7320508075688?*I
    to Elliptic Curve defined by y^2 + y = x^3 + (-7) over Number Field in r
       with defining polynomial x^2 + 3 with r = 1.7320508075688?*I,
```

(continues on next page)

(continued from previous page)

```
 Isogeny of degree 7
  from Elliptic Curve defined by y^2 + y = x^3 + (-7) over Number Field in r
       with defining polynomial x^2 + 3 with r = 1.732050807568878?*I
    to Elliptic Curve defined by y^2 + y = x^3 + (-7) over Number Field in r
       with defining polynomial x^2 + 3 with r = 1.732050807568878?*I]

sage: # needs sage.rings.number_field
sage: x = polygen(QQ, 'x')
sage: K.<a> = NumberField(x^6 + 1512*x^3 - 21168)
sage: E = EllipticCurve(K, [0,1])
sage: isogs = isogenies_7_0(E)
sage: [phi.codomain().a_invariants() for phi in isogs]
[(0,
  0,
  0,
  -415/98*a^5 - 675/14*a^4 + 2255/7*a^3 - 74700/7*a^2 - 25110*a - 66420,
  -141163/56*a^5 + 1443453/112*a^4 - 374275/2*a^3
                 - 3500211/2*a^2 - 17871975/4*a - 7710065),
 (0,
  0,
  0,
  -24485/392*a^5 - 1080/7*a^4 - 2255/7*a^3 - 1340865/14*a^2 - 230040*a - 553500,
  1753037/56*a^5 + 8345733/112*a^4 + 374275/2*a^3
                 + 95377029/2*a^2 + 458385345/4*a + 275241835)]
sage: [phi.codomain().j_invariant() for phi in isogs]
[158428486656000/7*a^3 - 313976217600000,
-158428486656000/7*a^3 - 34534529335296000]
```

sage.schemes.elliptic_curves.isogeny_small_degree.**isogenies_7_1728**(*E*, *minimal_models=True*)

Return list of all 7-isogenies from E when the j-invariant is 1728.

INPUT:

- `E` – an elliptic curve with j-invariant 1728.
- `minimal_models` (bool, default `True`) – if `True`, all curves computed will be minimal or semi-minimal models. Over fields of larger degree it can be expensive to compute these so set to `False`.

OUTPUT:

(list) 7-isogenies with codomain E. In general these are normalised; but over **Q** or a number field, the codomain is a global minimal model where possible.

Note: This implementation requires that the characteristic is not 2, 3, or 7.

Note: This function would normally be invoked indirectly via `E.isogenies_prime_degree(7)`.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.isogeny_small_degree import isogenies_7_
↪1728
sage: E = EllipticCurve(GF(47), [1, 0])
```

(continues on next page)

(continued from previous page)

```
sage: isogenies_7_1728(E)
[Isogeny of degree 7
  from Elliptic Curve defined by y^2 = x^3 + x over Finite Field of size 47
    to Elliptic Curve defined by y^2 = x^3 + 26 over Finite Field of size 47,
 Isogeny of degree 7
  from Elliptic Curve defined by y^2 = x^3 + x over Finite Field of size 47
    to Elliptic Curve defined by y^2 = x^3 + 21 over Finite Field of size 47]
```

An example in characteristic 53 (for which an earlier implementation did not work):

```
sage: # needs sage.rings.finite_rings
sage: from sage.schemes.elliptic_curves.isogeny_small_degree import isogenies_7_
↪1728
sage: E = EllipticCurve(GF(53), [1, 0])
sage: isogenies_7_1728(E)
[]
sage: E = EllipticCurve(GF(53^2,'a'), [1, 0])
sage: [iso.codomain().ainvs() for iso in isogenies_7_1728(E)]
[(0, 0, 0, 36, 19*a + 15), (0, 0, 0, 36, 34*a + 38), (0, 0, 0, 33, 39*a + 28),
 (0, 0, 0, 33, 14*a + 25), (0, 0, 0, 19, 45*a + 16), (0, 0, 0, 19, 8*a + 37),
 (0, 0, 0, 3, 45*a + 16), (0, 0, 0, 3, 8*a + 37)]
```

```
sage: # needs sage.rings.number_field
sage: x = polygen(QQ, 'x')
sage: K.<a> = NumberField(x^8 + 84*x^6 - 1890*x^4 + 644*x^2 - 567)
sage: E = EllipticCurve(K, [1, 0])
sage: isogs = isogenies_7_1728(E)
sage: [phi.codomain().j_invariant() for phi in isogs]
[-526110256146528/53*a^6 + 183649373229024*a^4
                         - 3333881559996576/53*a^2 + 2910267397643616/53,
 -526110256146528/53*a^6 + 183649373229024*a^4
                         - 3333881559996576/53*a^2 + 2910267397643616/53]
sage: E1 = isogs[0].codomain()
sage: E2 = isogs[1].codomain()
sage: E1.is_isomorphic(E2)
False
sage: E1.is_quadratic_twist(E2)
-1
```

sage.schemes.elliptic_curves.isogeny_small_degree.**isogenies_prime_degree**(*E*, *l*, *minimal_models=True*)

Return all separable `l`-isogenies with domain `E`.

INPUT:

- `E` – an elliptic curve.
- `l` – a prime.
- `minimal_models` (bool, default `True`) – if `True`, all curves computed will be minimal or semi-minimal models. Over fields of larger degree it can be expensive to compute these so set to `False`. Ignored except over number fields other than QQ.

OUTPUT:

A list of all separable isogenies of degree l with domain `E`.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.isogeny_small_degree import isogenies_
↪prime_degree
sage: E = EllipticCurve_from_j(GF(2^6,'a')(1))                                #␣
↪needs sage.rings.finite_rings
sage: isogenies_prime_degree(E, 7)                                            #␣
↪needs sage.rings.finite_rings
[Isogeny of degree 7
  from Elliptic Curve defined by y^2 + x*y = x^3 + 1
       over Finite Field in a of size 2^6
    to Elliptic Curve defined by y^2 + x*y = x^3 + x
       over Finite Field in a of size 2^6]
sage: E = EllipticCurve_from_j(GF(3^12,'a')(2))                               #␣
↪needs sage.rings.finite_rings
sage: isogenies_prime_degree(E, 17)                                           #␣
↪needs sage.rings.finite_rings
[Isogeny of degree 17
  from Elliptic Curve defined by y^2 = x^3 + 2*x^2 + 2
       over Finite Field in a of size 3^12
    to Elliptic Curve defined by y^2 = x^3 + 2*x^2 + x + 2
       over Finite Field in a of size 3^12,
 Isogeny of degree 17
  from Elliptic Curve defined by y^2 = x^3 + 2*x^2 + 2
       over Finite Field in a of size 3^12
    to Elliptic Curve defined by y^2 = x^3 + 2*x^2 + 2*x
       over Finite Field in a of size 3^12]
sage: E = EllipticCurve('50a1')
sage: isogenies_prime_degree(E, 3)
[Isogeny of degree 3
  from Elliptic Curve defined by y^2 + x*y + y = x^3 - x - 2 over Rational Field
    to Elliptic Curve defined by y^2 + x*y + y = x^3 - 126*x - 552 over Rational␣
↪Field]
sage: isogenies_prime_degree(E, 5)
[Isogeny of degree 5
  from Elliptic Curve defined by y^2 + x*y + y = x^3 - x - 2 over Rational Field
    to Elliptic Curve defined by y^2 + x*y + y = x^3 - 76*x + 298 over Rational␣
↪Field]
sage: E = EllipticCurve([0, 0, 1, -1862, -30956])
sage: isogenies_prime_degree(E, 19)
[Isogeny of degree 19
  from Elliptic Curve defined by y^2 + y = x^3 - 1862*x - 30956
       over Rational Field
    to Elliptic Curve defined by y^2 + y = x^3 - 672182*x + 212325489
       over Rational Field]
sage: E = EllipticCurve([0, -1, 0, -6288, 211072])
sage: isogenies_prime_degree(E, 37)
[Isogeny of degree 37
  from Elliptic Curve defined by y^2 = x^3 - x^2 - 6288*x + 211072
       over Rational Field
    to Elliptic Curve defined by y^2 = x^3 - x^2 - 163137088*x - 801950801728
       over Rational Field]
```

Isogenies of degree equal to the characteristic are computed (but only the separable isogeny). In the following example we consider an elliptic curve which is supersingular in characteristic 2 only:

```
sage: from sage.schemes.elliptic_curves.isogeny_small_degree import isogenies_
↪prime_degree
```

(continues on next page)

(continued from previous page)

```
sage: ainvs = (0,1,1,-1,-1)
sage: for l in prime_range(50):
....:     E = EllipticCurve(GF(l), ainvs)
....:     isogenies_prime_degree(E, l)
[]
[Isogeny of degree 3
  from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 2*x + 2 over Finite Field␣
↪of size 3
    to Elliptic Curve defined by y^2 + y = x^3 + x^2 + x over Finite Field of␣
↪size 3]
[Isogeny of degree 5
  from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 4*x + 4 over Finite Field␣
↪of size 5
    to Elliptic Curve defined by y^2 + y = x^3 + x^2 + 4*x + 4 over Finite Field␣
↪of size 5]
[Isogeny of degree 7
  from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 6*x + 6 over Finite Field␣
↪of size 7
    to Elliptic Curve defined by y^2 + y = x^3 + x^2 + 4 over Finite Field of␣
↪size 7]
[Isogeny of degree 11
  from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 10*x + 10 over Finite␣
↪Field of size 11
    to Elliptic Curve defined by y^2 + y = x^3 + x^2 + x + 1 over Finite Field of␣
↪size 11]
[Isogeny of degree 13
  from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 12*x + 12 over Finite␣
↪Field of size 13
    to Elliptic Curve defined by y^2 + y = x^3 + x^2 + 12*x + 12 over Finite␣
↪Field of size 13]
[Isogeny of degree 17
  from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 16*x + 16 over Finite␣
↪Field of size 17
    to Elliptic Curve defined by y^2 + y = x^3 + x^2 + 15 over Finite Field of␣
↪size 17]
[Isogeny of degree 19
  from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 18*x + 18 over Finite␣
↪Field of size 19
    to Elliptic Curve defined by y^2 + y = x^3 + x^2 + 3*x + 12 over Finite Field␣
↪of size 19]
[Isogeny of degree 23
  from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 22*x + 22 over Finite␣
↪Field of size 23
    to Elliptic Curve defined by y^2 + y = x^3 + x^2 + 22*x + 22 over Finite␣
↪Field of size 23]
[Isogeny of degree 29
  from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 28*x + 28 over Finite␣
↪Field of size 29
    to Elliptic Curve defined by y^2 + y = x^3 + x^2 + 7*x + 27 over Finite Field␣
↪of size 29]
[Isogeny of degree 31
  from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 30*x + 30 over Finite␣
↪Field of size 31
    to Elliptic Curve defined by y^2 + y = x^3 + x^2 + 15*x + 16 over Finite␣
↪Field of size 31]
[Isogeny of degree 37
  from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 36*x + 36 over Finite␣
```

(continues on next page)

(continued from previous page)

```
↪Field of size 37
   to Elliptic Curve defined by y^2 + y = x^3 + x^2 + 16*x + 17 over Finite␣
↪Field of size 37]
[Isogeny of degree 41
  from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 40*x + 40 over Finite␣
↪Field of size 41
   to Elliptic Curve defined by y^2 + y = x^3 + x^2 + 33*x + 16 over Finite␣
↪Field of size 41]
[Isogeny of degree 43
  from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 42*x + 42 over Finite␣
↪Field of size 43
   to Elliptic Curve defined by y^2 + y = x^3 + x^2 + 36 over Finite Field of␣
↪size 43]
[Isogeny of degree 47
  from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 46*x + 46 over Finite␣
↪Field of size 47
   to Elliptic Curve defined by y^2 + y = x^3 + x^2 + 42*x + 34 over Finite␣
↪Field of size 47]
```

Note that the computation is faster for degrees equal to one of the genus 0 primes (2, 3, 5, 7, 13) or one of the hyperelliptic primes (11, 17, 19, 23, 29, 31, 41, 47, 59, 71) than when the generic code must be used:

```
sage: E = EllipticCurve(GF(101), [-3440, 77658])
sage: E.isogenies_prime_degree(71)  # fast
[]
sage: E.isogenies_prime_degree(73)      # long time
[]
```

Test that Issue #32269 is fixed:

```
sage: K = QuadraticField(-11)                                                  #␣
↪needs sage.rings.number_field
sage: E = EllipticCurve(K, [0,1,0,-117,-541])                                  #␣
↪needs sage.rings.number_field
sage: E.isogenies_prime_degree(37)      # long time                            #␣
↪needs sage.rings.number_field
[Isogeny of degree 37
  from Elliptic Curve defined by y^2 = x^3 + x^2 + (-117)*x + (-541)
       over Number Field in a with defining polynomial x^2 + 11
       with a = 3.316624790355400?*I
    to Elliptic Curve defined by
       y^2 = x^3 + x^2 + (-30800*a+123963)*x + (-3931312*a-21805005)
       over Number Field in a with defining polynomial x^2 + 11
       with a = 3.316624790355400?*I,
 Isogeny of degree 37
  from Elliptic Curve defined by y^2 = x^3 + x^2 + (-117)*x + (-541)
       over Number Field in a with defining polynomial x^2 + 11
       with a = 3.316624790355400?*I
    to Elliptic Curve defined by
       y^2 = x^3 + x^2 + (30800*a+123963)*x + (3931312*a-21805005)
       over Number Field in a with defining polynomial x^2 + 11
       with a = 3.316624790355400?*I]
```

sage.schemes.elliptic_curves.isogeny_small_degree.**isogenies_prime_degree_general**(*E*, *l*, *minimal_models=True*)

Return all separable `l`-isogenies with domain `E`.

INPUT:

- `E` – an elliptic curve.
- `l` – a prime.
- `minimal_models` (bool, default `True`) – if `True`, all curves computed will be minimal or semi-minimal models. Over fields of larger degree it can be expensive to compute these so set to `False`.

OUTPUT:

A list of all separable isogenies of degree l with domain `E` (up to post-isomorphism).

ALGORITHM:

This algorithm factors the `l`-division polynomial, then combines its factors to obtain kernels. Originally this was done using [KT2013], Chapter 3, but nowadays the recombination step is instead delegated to *kernel_polynomial_from_divisor()*.

Note: This function works for any prime l. Normally one should use the function *isogenies_prime_degree()* which uses special functions for certain small primes.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.isogeny_small_degree import isogenies_
↪prime_degree_general
sage: E = EllipticCurve_from_j(GF(2^6,'a')(1))                                #␣
↪needs sage.rings.finite_rings
sage: isogenies_prime_degree_general(E, 7)                                    #␣
↪needs sage.rings.finite_rings
[Isogeny of degree 7
  from Elliptic Curve defined by y^2 + x*y = x^3 + 1
       over Finite Field in a of size 2^6
    to Elliptic Curve defined by y^2 + x*y = x^3 + x
       over Finite Field in a of size 2^6]
sage: E = EllipticCurve_from_j(GF(3^12,'a')(2))                               #␣
↪needs sage.rings.finite_rings
sage: isogenies_prime_degree_general(E, 17)                                   #␣
↪needs sage.rings.finite_rings
[Isogeny of degree 17
  from Elliptic Curve defined by y^2 = x^3 + 2*x^2 + 2
       over Finite Field in a of size 3^12
    to Elliptic Curve defined by y^2 = x^3 + 2*x^2 + x + 2
       over Finite Field in a of size 3^12,
 Isogeny of degree 17
  from Elliptic Curve defined by y^2 = x^3 + 2*x^2 + 2
       over Finite Field in a of size 3^12
    to Elliptic Curve defined by y^2 = x^3 + 2*x^2 + 2*x
       over Finite Field in a of size 3^12]
sage: E = EllipticCurve('50a1')
```

(continues on next page)

(continued from previous page)

```
sage: isogenies_prime_degree_general(E, 3)
[Isogeny of degree 3
  from Elliptic Curve defined by y^2 + x*y + y = x^3 - x - 2 over Rational Field
    to Elliptic Curve defined by y^2 + x*y + y = x^3 - 126*x - 552
       over Rational Field]
sage: isogenies_prime_degree_general(E, 5)
[Isogeny of degree 5
  from Elliptic Curve defined by y^2 + x*y + y = x^3 - x - 2 over Rational Field
    to Elliptic Curve defined by y^2 + x*y + y = x^3 - 76*x + 298
       over Rational Field]
sage: E = EllipticCurve([0, 0, 1, -1862, -30956])
sage: isogenies_prime_degree_general(E, 19)
[Isogeny of degree 19
  from Elliptic Curve defined by y^2 + y = x^3 - 1862*x - 30956
       over Rational Field
    to Elliptic Curve defined by y^2 + y = x^3 - 672182*x + 212325489
       over Rational Field]
sage: E = EllipticCurve([0, -1, 0, -6288, 211072])
sage: isogenies_prime_degree_general(E, 37)  # long time (2s)
[Isogeny of degree 37
  from Elliptic Curve defined by y^2 = x^3 - x^2 - 6288*x + 211072
       over Rational Field
    to Elliptic Curve defined by y^2 = x^3 - x^2 - 163137088*x - 801950801728
       over Rational Field]

sage: E = EllipticCurve([-3440, 77658])
sage: isogenies_prime_degree_general(E, 43)  # long time (2s)
[Isogeny of degree 43
  from Elliptic Curve defined by y^2 = x^3 - 3440*x + 77658 over Rational Field
    to Elliptic Curve defined by y^2 = x^3 - 6360560*x - 6174354606
       over Rational Field]
```

Isogenies of degree equal to the characteristic are computed (but only the separable isogeny). In the following example we consider an elliptic curve which is supersingular in characteristic 2 only:

```
sage: from sage.schemes.elliptic_curves.isogeny_small_degree import isogenies_
↪prime_degree_general
sage: ainvs = (0,1,1,-1,-1)
sage: for l in prime_range(50):
....:     E = EllipticCurve(GF(l),ainvs)
....:     isogenies_prime_degree_general(E,l)
[]
[Isogeny of degree 3
  from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 2*x + 2
       over Finite Field of size 3
    to Elliptic Curve defined by y^2 + y = x^3 + x^2 + x over Finite Field of␣
↪size 3]
[Isogeny of degree 5
  from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 4*x + 4
       over Finite Field of size 5
    to Elliptic Curve defined by y^2 + y = x^3 + x^2 + 4*x + 4
       over Finite Field of size 5]
[Isogeny of degree 7
  from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 6*x + 6
       over Finite Field of size 7
    to Elliptic Curve defined by y^2 + y = x^3 + x^2 + 4 over Finite Field of␣
↪size 7]
```

(continues on next page)

(continued from previous page)

```
[Isogeny of degree 11
  from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 10*x + 10
       over Finite Field of size 11
    to Elliptic Curve defined by y^2 + y = x^3 + x^2 + x + 1
       over Finite Field of size 11]
[Isogeny of degree 13
  from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 12*x + 12
       over Finite Field of size 13
    to Elliptic Curve defined by y^2 + y = x^3 + x^2 + 12*x + 12
       over Finite Field of size 13]
[Isogeny of degree 17 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 16*x +␣
↪16 over Finite Field of size 17 to Elliptic Curve defined by y^2 + y = x^3 + x^
↪2 + 15 over Finite Field of size 17]
[Isogeny of degree 19 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 18*x +␣
↪18 over Finite Field of size 19 to Elliptic Curve defined by y^2 + y = x^3 + x^
↪2 + 3*x + 12 over Finite Field of size 19]
[Isogeny of degree 23 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 22*x +␣
↪22 over Finite Field of size 23 to Elliptic Curve defined by y^2 + y = x^3 + x^
↪2 + 22*x + 22 over Finite Field of size 23]
[Isogeny of degree 29 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 28*x +␣
↪28 over Finite Field of size 29 to Elliptic Curve defined by y^2 + y = x^3 + x^
↪2 + 7*x + 27 over Finite Field of size 29]
[Isogeny of degree 31 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 30*x +␣
↪30 over Finite Field of size 31 to Elliptic Curve defined by y^2 + y = x^3 + x^
↪2 + 15*x + 16 over Finite Field of size 31]
[Isogeny of degree 37 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 36*x +␣
↪36 over Finite Field of size 37 to Elliptic Curve defined by y^2 + y = x^3 + x^
↪2 + 16*x + 17 over Finite Field of size 37]
[Isogeny of degree 41 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 40*x +␣
↪40 over Finite Field of size 41 to Elliptic Curve defined by y^2 + y = x^3 + x^
↪2 + 33*x + 16 over Finite Field of size 41]
[Isogeny of degree 43 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 42*x +␣
↪42 over Finite Field of size 43 to Elliptic Curve defined by y^2 + y = x^3 + x^
↪2 + 36 over Finite Field of size 43]
[Isogeny of degree 47 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 46*x +␣
↪46 over Finite Field of size 47 to Elliptic Curve defined by y^2 + y = x^3 + x^
↪2 + 42*x + 34 over Finite Field of size 47]
```

Note that not all factors of degree $(l-1)/2$ of the l-division polynomial are kernel polynomials. In this example, the 13-division polynomial factors as a product of 14 irreducible factors of degree 6 each, but only two those are kernel polynomials:

```
sage: F3 = GF(3)
sage: E = EllipticCurve(F3, [0,0,0,-1,0])
sage: Psi13 = E.division_polynomial(13)
sage: len([f for f, e in Psi13.factor() if f.degree() == 6])
14
sage: len(E.isogenies_prime_degree(13))
2
```

Over GF(9) the other factors of degree 6 split into pairs of cubics which can be rearranged to give the remaining 12 kernel polynomials:

```
sage: len(E.change_ring(GF(3^2,'a')).isogenies_prime_degree(13))                #␣
↪needs sage.rings.finite_rings
14
```

See Issue #18589: the following example took 20s before, now only 4s:

```
sage: K.<i> = QuadraticField(-1)                                                #␣
↪needs sage.rings.number_field
sage: E = EllipticCurve(K,[0,0,0,1,0])                                          #␣
↪needs sage.rings.number_field
sage: [phi.codomain().ainvs()                    # long time                    #␣
↪needs sage.rings.number_field
....:  for phi in E.isogenies_prime_degree(37)]
[(0, 0, 0, 840*i + 1081, 0),
 (0, 0, 0, -840*i + 1081, 0)]
```

sage.schemes.elliptic_curves.isogeny_small_degree.**isogenies_prime_degree_genus_0**(*E*, *l=None*, *minimal_models=True*)

Return list of `l` -isogenies with domain `E`.

INPUT:

- `E` – an elliptic curve.
- `l` – either None or 2, 3, 5, 7, or 13.
- `minimal_models` (bool, default `True`) – if `True`, all curves computed will be minimal or semi-minimal models. Over fields of larger degree it can be expensive to compute these so set to `False`.

OUTPUT:

(list) When `l` is None a list of all isogenies of degree 2, 3, 5, 7 and 13, otherwise a list of isogenies of the given degree.

Note: This function would normally be invoked indirectly via `E.isogenies_prime_degree(l)`, which automatically calls the appropriate function.

ALGORITHM:

Cremona and Watkins [CW2005]. See also [KT2013], Chapter 4.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.isogeny_small_degree import isogenies_
↪prime_degree_genus_0
sage: E = EllipticCurve([0,12])
sage: isogenies_prime_degree_genus_0(E, 5)
[]

sage: E = EllipticCurve('1450c1')
sage: isogenies_prime_degree_genus_0(E)
[Isogeny of degree 3
  from Elliptic Curve defined by y^2 + x*y = x^3 + x^2 + 300*x - 1000
       over Rational Field
    to Elliptic Curve defined by y^2 + x*y = x^3 + x^2 - 5950*x - 182250
       over Rational Field]

sage: E = EllipticCurve('50a1')
```

(continues on next page)

(continued from previous page)

```
sage: isogenies_prime_degree_genus_0(E)
[Isogeny of degree 3
  from Elliptic Curve defined by y^2 + x*y + y = x^3 - x - 2 over Rational Field
    to Elliptic Curve defined by y^2 + x*y + y = x^3 - 126*x - 552 over Rational␣
↪Field,
 Isogeny of degree 5
  from Elliptic Curve defined by y^2 + x*y + y = x^3 - x - 2 over Rational Field
    to Elliptic Curve defined by y^2 + x*y + y = x^3 - 76*x + 298 over Rational␣
↪Field]
```

sage.schemes.elliptic_curves.isogeny_small_degree.**isogenies_prime_degree_genus_plus_0**(*E*, *l=None*, *minimal_models=True*)

Return list of `l` -isogenies with domain `E`.

INPUT:

- `E` – an elliptic curve.
- `l` – either None or 11, 17, 19, 23, 29, 31, 41, 47, 59, or 71.
- `minimal_models` (bool, default `True`) – if `True`, all curves computed will be minimal or semi-minimal models. Over fields of larger degree it can be expensive to compute these so set to `False`.

OUTPUT:

(list) When `l` is None a list of all isogenies of degree 11, 17, 19, 23, 29, 31, 41, 47, 59, or 71, otherwise a list of isogenies of the given degree.

Note: This function would normally be invoked indirectly via `E.isogenies_prime_degree(l)`, which automatically calls the appropriate function.

ALGORITHM:

See [KT2013], Chapter 5.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.isogeny_small_degree import isogenies_
↪prime_degree_genus_plus_0

sage: E = EllipticCurve('121a1')
sage: isogenies_prime_degree_genus_plus_0(E, 11)
[Isogeny of degree 11
  from Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 30*x - 76
       over Rational Field
    to Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 305*x + 7888
       over Rational Field]

sage: E = EllipticCurve([1, 1, 0, -660, -7600])
sage: isogenies_prime_degree_genus_plus_0(E, 17)
[Isogeny of degree 17
  from Elliptic Curve defined by y^2 + x*y = x^3 + x^2 - 660*x - 7600
       over Rational Field
```

(continues on next page)

(continued from previous page)

```
    to Elliptic Curve defined by y^2 + x*y = x^3 + x^2 - 878710*x + 316677750
       over Rational Field]

sage: E = EllipticCurve([0, 0, 1, -1862, -30956])
sage: isogenies_prime_degree_genus_plus_0(E, 19)
[Isogeny of degree 19
  from Elliptic Curve defined by y^2 + y = x^3 - 1862*x - 30956
       over Rational Field
    to Elliptic Curve defined by y^2 + y = x^3 - 672182*x + 212325489
       over Rational Field]

sage: # needs sage.rings.number_field
sage: K = QuadraticField(-295,'a')
sage: a = K.gen()
sage: E = EllipticCurve_from_j(-484650135/16777216*a + 4549855725/16777216)
sage: isogenies_prime_degree_genus_plus_0(E, 23)
[Isogeny of degree 23
  from Elliptic Curve defined by
       y^2 = x^3 + (-1446049478419290409 5/140737488355328*a+270742665778826768325/
↪140737488355328)*x
             + (37035998788154488846811217135/590295810358705651712*a-
↪1447451882571839266752561148725/590295810358705651712)
       over Number Field in a with defining polynomial x^2 + 295
       with a = 17.17556403731767?*I
    to Elliptic Curve defined by
       y^2 = x^3 + (-5130542435555445498495/
↪140737488355328*a+1732339550291273610059 25/140737488355328)*x
             + (-1104699335561165691575396879260545/
↪590295810358705651712*a+3169785826904210171629535101419675/
↪590295810358705651712)
       over Number Field in a with defining polynomial x^2 + 295
       with a = 17.17556403731767?*I]

sage: # needs sage.rings.number_field
sage: K = QuadraticField(-199,'a')
sage: a = K.gen()
sage: E = EllipticCurve_from_j(94743000*a + 269989875)
sage: isogenies_prime_degree_genus_plus_0(E, 29)
[Isogeny of degree 29
  from Elliptic Curve defined by
       y^2 = x^3 + (-153477413215038000*a+5140130723072965125)*x
             + (29703621513054700845552 6000*a+285427704716431780097 3582250)
       over Number Field in a with defining polynomial x^2 + 199
       with a = 14.106735979665884?*I
    to Elliptic Curve defined by
       y^2 = x^3 + (2513361613780408050 00*a-3071093219933084341875)*x
             + (-
↪84110642831621685801876432210 00*a+348043377707983895460171847852 50)
       over Number Field in a with defining polynomial x^2 + 199
       with a = 14.106735979665884?*I]

sage: # needs sage.rings.number_field
sage: K = QuadraticField(253,'a')
sage: a = K.gen()
sage: E = EllipticCurve_from_j(208438034112000*a - 3315409892960000)
sage: isogenies_prime_degree_genus_plus_0(E, 31)
[Isogeny of degree 31
```

(continues on next page)

(continued from previous page)

```
  from Elliptic Curve defined by
       y^2 = x^3 + (41463451221854330346779566080000*a-
↪659516565499650372596348006400000)*x
              + (-
↪183291115169544734745834253936982450802524160000*a+291542366110383928366510368064204147260129280
       over Number Field in a with defining polynomial x^2 - 253
       with a = 15.905973720586867?
    to Elliptic Curve defined by
       y^2 = x^3 + (200339763852548615776123686912000*a-
↪318659901902721690428094827520000)*x
              + (744367179141147962911271726018228629485020774400*a-
↪11839884789886475720968595172883889549516865536000)
       over Number Field in a with defining polynomial x^2 - 253
       with a = 15.905973720586867?]

sage: E = EllipticCurve_from_j(GF(5)(1))
sage: isogenies_prime_degree_genus_plus_0(E, 41)
[Isogeny of degree 41
  from Elliptic Curve defined by y^2 = x^3 + x + 2 over Finite Field of size 5
    to Elliptic Curve defined by y^2 = x^3 + x + 3 over Finite Field of size 5,
 Isogeny of degree 41
  from Elliptic Curve defined by y^2 = x^3 + x + 2 over Finite Field of size 5
    to Elliptic Curve defined by y^2 = x^3 + x + 3 over Finite Field of size 5]

sage: # needs sage.rings.number_field
sage: K = QuadraticField(5,'a')
sage: a = K.gen()
sage: E = EllipticCurve_from_j(184068066743177379840*a
....:                          - 411588709724712960000)
sage: isogenies_prime_degree_genus_plus_0(E, 47)        # long time
[Isogeny of degree 47
  from Elliptic Curve defined by
       y^2 = x^3 + (4545620285540803558578520498499758954905 60*a-
↪10164315958371241146686892861765113610240 00)*x
              + (-
↪24945679842989608088144054095039371330383036399948090428096512 0*a+55780235873871044345127332 02
       over Number Field in a with defining polynomial x^2 - 5
       with a = 2.236067977499790?
    to Elliptic Curve defined by
       y^2 = x^3 + (39533118442361013730577638493616965245992960*a-
↪88398740199669828340617478832005245173760000)*x
              +␣
↪(21403032147946661028232052861156236896383010583055536306180325376 0*a-
↪47858634807422069968761632253266616372200449745845231658257612 8000)
       over Number Field in a with defining polynomial x^2 - 5
       with a = 2.236067977499790?]

sage: K = QuadraticField(-66827,'a')                                    #␣
↪needs sage.rings.number_field
sage: a = K.gen()                                                       #␣
↪needs sage.rings.number_field
sage: E = EllipticCurve_from_j(-98669236224000*a + 4401720074240000)    #␣
↪needs sage.rings.number_field
sage: isogenies_prime_degree_genus_plus_0(E, 59)   # long time (5s)
[Isogeny of degree 59
  from Elliptic Curve defined by
       y^2 = x^3 +␣
```

(continues on next page)

(continued from previous page)

```
↪(26058861467821447622979747840000*a+189368104891277363494463471616000)*x
             + (-
↪11691845425641078223229618319806756874407116800 0*a+17012043538294664027185882358514011304812870
      over Number Field in a with defining polynomial x^2 + 66827
      with a = 258.5091874576221?*I
   to Elliptic Curve defined by
      y^2 = x^3 + (-193870840271597868214007750983680 00*a-
↪48820591048681542250527871567134720 00)*x
             + (-25659862010101415428713331477227179429538847260672000*a-
↪25960381484412934859387981190034629728408183819468800 00)
      over Number Field in a with defining polynomial x^2 + 66827
      with a = 258.5091874576221?*I]

sage: E = EllipticCurve_from_j(GF(13)(5))
sage: isogenies_prime_degree_genus_plus_0(E, 71)
[Isogeny of degree 71
  from Elliptic Curve defined by y^2 = x^3 + x + 4 over Finite Field of size 13
    to Elliptic Curve defined by y^2 = x^3 + 10*x + 7 over Finite Field of size␣
↪13,
 Isogeny of degree 71
  from Elliptic Curve defined by y^2 = x^3 + x + 4 over Finite Field of size 13
    to Elliptic Curve defined by y^2 = x^3 + 10*x + 7 over Finite Field of size␣
↪13]

sage: E = EllipticCurve(GF(13), [0,1,1,1,0])
sage: isogenies_prime_degree_genus_plus_0(E)
[Isogeny of degree 17
  from Elliptic Curve defined by y^2 + y = x^3 + x^2 + x over Finite Field of␣
↪size 13
    to Elliptic Curve defined by y^2 + y = x^3 + x^2 + 10*x + 1 over Finite Field␣
↪of size 13,
 Isogeny of degree 17
  from Elliptic Curve defined by y^2 + y = x^3 + x^2 + x over Finite Field of␣
↪size 13
    to Elliptic Curve defined by y^2 + y = x^3 + x^2 + 12*x + 4 over Finite Field␣
↪of size 13,
 Isogeny of degree 29
  from Elliptic Curve defined by y^2 + y = x^3 + x^2 + x over Finite Field of␣
↪size 13
    to Elliptic Curve defined by y^2 + y = x^3 + x^2 + 12*x + 6 over Finite Field␣
↪of size 13,
 Isogeny of degree 29
  from Elliptic Curve defined by y^2 + y = x^3 + x^2 + x over Finite Field of␣
↪size 13
    to Elliptic Curve defined by y^2 + y = x^3 + x^2 + 5*x + 6 over Finite Field␣
↪of size 13,
 Isogeny of degree 41
  from Elliptic Curve defined by y^2 + y = x^3 + x^2 + x over Finite Field of␣
↪size 13
    to Elliptic Curve defined by y^2 + y = x^3 + x^2 + 12*x + 4 over Finite Field␣
↪of size 13,
 Isogeny of degree 41
  from Elliptic Curve defined by y^2 + y = x^3 + x^2 + x over Finite Field of␣
↪size 13
    to Elliptic Curve defined by y^2 + y = x^3 + x^2 + 5*x + 6 over Finite Field␣
↪of size 13]
```

sage.schemes.elliptic_curves.isogeny_small_degree.**isogenies_prime_degree_genus_plus_0_j0**(*E*, *l*, *minimal_models=True*)

Return a list of hyperelliptic `l` -isogenies with domain `E` when $j(E) = 0$.

INPUT:

- `E` – an elliptic curve with j-invariant 0.
- `l` – 11, 17, 19, 23, 29, 31, 41, 47, 59, or 71.
- `minimal_models` (bool, default `True`) – if `True`, all curves computed will be minimal or semi-minimal models. Over fields of larger degree it can be expensive to compute these so set to `False`.

OUTPUT:

(list) a list of all isogenies of degree 11, 17, 19, 23, 29, 31, 41, 47, 59, or 71.

Note: This implementation requires that the characteristic is not 2, 3 or `l`.

Note: This function would normally be invoked indirectly via `E.isogenies_prime_degree(l)`.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.isogeny_small_degree import isogenies_
↪prime_degree_genus_plus_0_j0

sage: u = polygen(QQ)
sage: K.<a> = NumberField(u^4 + 228*u^3 + 486*u^2 - 540*u + 225)                #␣
↪needs sage.rings.number_field
sage: E = EllipticCurve(K, [0, -121/5*a^3 - 20691/5*a^2 - 29403/5*a + 3267])    #␣
↪needs sage.rings.number_field
sage: isogenies_prime_degree_genus_plus_0_j0(E, 11)                             #␣
↪needs sage.rings.number_field
[Isogeny of degree 11
  from Elliptic Curve defined by
       y^2 = x^3 + (-121/5*a^3-20691/5*a^2-29403/5*a+3267) over
       Number Field in a with defining polynomial x^4 + 228*x^3 + 486*x^2 - 540*x␣
↪+ 225
    to Elliptic Curve defined by
       y^2 = x^3 + (-44286*a^2+178596*a-32670)*x
              + (-17863351/5*a^3+125072739/5*a^2-74353653/5*a-682803) over
       Number Field in a with defining polynomial x^4 + 228*x^3 + 486*x^2 - 540*x␣
↪+ 225,
 Isogeny of degree 11
  from Elliptic Curve defined by
       y^2 = x^3 + (-121/5*a^3-20691/5*a^2-29403/5*a+3267) over
       Number Field in a with defining polynomial x^4 + 228*x^3 + 486*x^2 - 540*x␣
↪+ 225
    to Elliptic Curve defined by
       y^2 = x^3 + (-3267*a^3-740157*a^2+600039*a-277695)*x
              + (-17863351/5*a^3-4171554981/5*a^2+3769467867/5*a-272366523) over
       Number Field in a with defining polynomial x^4 + 228*x^3 + 486*x^2 - 540*x␣
```

(continues on next page)

(continued from previous page)

```
↪+ 225]

sage: E = EllipticCurve(GF(5^6,'a'),[0,1])
sage: isogenies_prime_degree_genus_plus_0_j0(E,17)
[Isogeny of degree 17 from Elliptic Curve defined by y^2 = x^3 + 1 over Finite␣
↪Field in a of size 5^6 to Elliptic Curve defined by y^2 = x^3 + 2 over Finite␣
↪Field in a of size 5^6, Isogeny of degree 17 from Elliptic Curve defined by y^2␣
↪= x^3 + 1 over Finite Field in a of size 5^6 to Elliptic Curve defined by y^2 =␣
↪x^3 + 2 over Finite Field in a of size 5^6, Isogeny of degree 17 from Elliptic␣
↪Curve defined by y^2 = x^3 + 1 over Finite Field in a of size 5^6 to Elliptic␣
↪Curve defined by y^2 = x^3 + 2 over Finite Field in a of size 5^6, Isogeny of␣
↪degree 17 from Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field in a␣
↪of size 5^6 to Elliptic Curve defined by y^2 = x^3 + 2 over Finite Field in a␣
↪of size 5^6, Isogeny of degree 17 from Elliptic Curve defined by y^2 = x^3 + 1␣
↪over Finite Field in a of size 5^6 to Elliptic Curve defined by y^2 = x^3 + 2␣
↪over Finite Field in a of size 5^6, Isogeny of degree 17 from Elliptic Curve␣
↪defined by y^2 = x^3 + 1 over Finite Field in a of size 5^6 to Elliptic Curve␣
↪defined by y^2 = x^3 + 2 over Finite Field in a of size 5^6, Isogeny of degree␣
↪17 from Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field in a of size␣
↪5^6 to Elliptic Curve defined by y^2 = x^3 + 2 over Finite Field in a of size 5^
↪6, Isogeny of degree 17 from Elliptic Curve defined by y^2 = x^3 + 1 over␣
↪Finite Field in a of size 5^6 to Elliptic Curve defined by y^2 = x^3 + 2 over␣
↪Finite Field in a of size 5^6, Isogeny of degree 17 from Elliptic Curve defined␣
↪by y^2 = x^3 + 1 over Finite Field in a of size 5^6 to Elliptic Curve defined␣
↪by y^2 = x^3 + 2 over Finite Field in a of size 5^6, Isogeny of degree 17 from␣
↪Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field in a of size 5^6 to␣
↪Elliptic Curve defined by y^2 = x^3 + 2 over Finite Field in a of size 5^6,␣
↪Isogeny of degree 17 from Elliptic Curve defined by y^2 = x^3 + 1 over Finite␣
↪Field in a of size 5^6 to Elliptic Curve defined by y^2 = x^3 + 2 over Finite␣
↪Field in a of size 5^6, Isogeny of degree 17 from Elliptic Curve defined by y^2␣
↪= x^3 + 1 over Finite Field in a of size 5^6 to Elliptic Curve defined by y^2 =␣
↪x^3 + 2 over Finite Field in a of size 5^6, Isogeny of degree 17 from Elliptic␣
↪Curve defined by y^2 = x^3 + 1 over Finite Field in a of size 5^6 to Elliptic␣
↪Curve defined by y^2 = x^3 + 2 over Finite Field in a of size 5^6, Isogeny of␣
↪degree 17 from Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field in a␣
↪of size 5^6 to Elliptic Curve defined by y^2 = x^3 + 2 over Finite Field in a␣
↪of size 5^6, Isogeny of degree 17 from Elliptic Curve defined by y^2 = x^3 + 1␣
↪over Finite Field in a of size 5^6 to Elliptic Curve defined by y^2 = x^3 + 2␣
↪over Finite Field in a of size 5^6, Isogeny of degree 17 from Elliptic Curve␣
↪defined by y^2 = x^3 + 1 over Finite Field in a of size 5^6 to Elliptic Curve␣
↪defined by y^2 = x^3 + 2 over Finite Field in a of size 5^6, Isogeny of degree␣
↪17 from Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field in a of size␣
↪5^6 to Elliptic Curve defined by y^2 = x^3 + 2 over Finite Field in a of size 5^
↪6, Isogeny of degree 17 from Elliptic Curve defined by y^2 = x^3 + 1 over␣
↪Finite Field in a of size 5^6 to Elliptic Curve defined by y^2 = x^3 + 2 over␣
↪Finite Field in a of size 5^6]
```

sage.schemes.elliptic_curves.isogeny_small_degree.**isogenies_prime_degree_genus_plus_0_j1728**

Return a list of `l` -isogenies with domain `E` when $j(E) = 1728$.

INPUT:

- `E` – an elliptic curve with j-invariant 1728.
- `l` – 11, 17, 19, 23, 29, 31, 41, 47, 59, or 71.
- `minimal_models` (bool, default `True`) – if `True`, all curves computed will be minimal or semi-minimal models. Over fields of larger degree it can be expensive to compute these so set to `False`.

OUTPUT:

(list) a list of all isogenies of degree 11, 17, 19, 23, 29, 31, 41, 47, 59, or 71.

Note: This implementation requires that the characteristic is not 2, 3 or `l`.

Note: This function would normally be invoked indirectly via `E.isogenies_prime_degree(l)`.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.isogeny_small_degree import isogenies_
↪prime_degree_genus_plus_0_j1728

sage: # needs sage.rings.number_field
sage: u = polygen(QQ)
sage: K.<a> = NumberField(u^6 - 522*u^5 - 10017*u^4
....:                     + 2484*u^3 - 5265*u^2 + 12150*u - 5103)
sage: E = EllipticCurve(K, [-75295/1335852*a^5 + 13066735/445284*a^4
....:                       + 44903485/74214*a^3 + 17086861/24738*a^2
....:                       + 11373021/16492*a - 1246245/2356, 0])
sage: isogenies_prime_degree_genus_plus_0_j1728(E, 11)
[Isogeny of degree 11
  from Elliptic Curve defined by
       y^2 = x^3 + (-75295/1335852*a^5+13066735/445284*a^4+44903485/74214*a^
↪3+17086861/24738*a^2+11373021/16492*a-1246245/2356)*x
       over Number Field in a with defining polynomial
       x^6 - 522*x^5 - 10017*x^4 + 2484*x^3 - 5265*x^2 + 12150*x - 5103
    to Elliptic Curve defined by
       y^2 = x^3 + (9110695/1335852*a^5-1581074935/445284*a^4-5433321685/74214*a^
↪3-3163057249/24738*a^2+1569269691/16492*a+73825125/2356)*x
              + (-3540460*a^3+30522492*a^2-7043652*a-5031180)
       over Number Field in a with defining polynomial
       x^6 - 522*x^5 - 10017*x^4 + 2484*x^3 - 5265*x^2 + 12150*x - 5103,
 Isogeny of degree 11
  from Elliptic Curve defined by
       y^2 = x^3 + (-75295/1335852*a^5+13066735/445284*a^4+44903485/74214*a^
↪3+17086861/24738*a^2+11373021/16492*a-1246245/2356)*x
       over Number Field in a with defining polynomial
       x^6 - 522*x^5 - 10017*x^4 + 2484*x^3 - 5265*x^2 + 12150*x - 5103
    to Elliptic Curve defined by
       y^2 = x^3 + (9110695/1335852*a^5-1581074935/445284*a^4-5433321685/74214*a^
↪3-3163057249/24738*a^2+1569269691/16492*a+73825125/2356)*x
              + (3540460*a^3-30522492*a^2+7043652*a+5031180)
       over Number Field in a with defining polynomial
       x^6 - 522*x^5 - 10017*x^4 + 2484*x^3 - 5265*x^2 + 12150*x - 5103]
sage: i = QuadraticField(-1,'i').gen()
sage: E = EllipticCurve([-1 - 2*i, 0])
sage: isogenies_prime_degree_genus_plus_0_j1728(E, 17)
[Isogeny of degree 17
```

(continues on next page)

(continued from previous page)

```
  from Elliptic Curve defined by y^2 = x^3 + (-2*i-1)*x
       over Number Field in i with defining polynomial x^2 + 1 with i = 1*I
    to Elliptic Curve defined by y^2 = x^3 + (-82*i-641)*x
       over Number Field in i with defining polynomial x^2 + 1 with i = 1*I,
 Isogeny of degree 17
  from Elliptic Curve defined by y^2 = x^3 + (-2*i-1)*x
       over Number Field in i with defining polynomial x^2 + 1 with i = 1*I
    to Elliptic Curve defined by y^2 = x^3 + (-562*i+319)*x
       over Number Field in i with defining polynomial x^2 + 1 with i = 1*I]
sage: Emin = E.global_minimal_model()
sage: [(p, len(isogenies_prime_degree_genus_plus_0_j1728(Emin, p)))
....:  for p in [17, 29, 41]]
[(17, 2), (29, 2), (41, 2)]
```

sage.schemes.elliptic_curves.isogeny_small_degree.**isogenies_sporadic_Q**(*E*, *l=None*, *minimal_models=True*)

Return a list of sporadic l-isogenies from E (l = 11, 17, 19, 37, 43, 67 or 163). Only for elliptic curves over **Q**.

INPUT:

- `E` – an elliptic curve defined over **Q**.
- `l` – either None or a prime number.

OUTPUT:

(list) If `l` is None, a list of all isogenies with domain `E` and of degree 11, 17, 19, 37, 43, 67 or 163; otherwise a list of isogenies of the given degree.

Note: This function would normally be invoked indirectly via `E.isogenies_prime_degree(l)`, which automatically calls the appropriate function.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.isogeny_small_degree import isogenies_
↪sporadic_Q
sage: E = EllipticCurve('121a1')
sage: isogenies_sporadic_Q(E, 11)
[Isogeny of degree 11
  from Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 30*x - 76
       over Rational Field
    to Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 305*x + 7888
       over Rational Field]
sage: isogenies_sporadic_Q(E, 13)
[]
sage: isogenies_sporadic_Q(E, 17)
[]
sage: isogenies_sporadic_Q(E)
[Isogeny of degree 11
  from Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 30*x - 76
       over Rational Field
    to Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 305*x + 7888
       over Rational Field]
```

(continues on next page)

(continued from previous page)

```
sage: E = EllipticCurve([1, 1, 0, -660, -7600])
sage: isogenies_sporadic_Q(E, 17)
[Isogeny of degree 17
  from Elliptic Curve defined by y^2 + x*y = x^3 + x^2 - 660*x - 7600
       over Rational Field
    to Elliptic Curve defined by y^2 + x*y = x^3 + x^2 - 878710*x + 316677750
       over Rational Field]
sage: isogenies_sporadic_Q(E)
[Isogeny of degree 17
  from Elliptic Curve defined by y^2 + x*y = x^3 + x^2 - 660*x - 7600
       over Rational Field
    to Elliptic Curve defined by y^2 + x*y = x^3 + x^2 - 878710*x + 316677750
       over Rational Field]
sage: isogenies_sporadic_Q(E, 11)
[]

sage: E = EllipticCurve([0, 0, 1, -1862, -30956])
sage: isogenies_sporadic_Q(E, 11)
[]
sage: isogenies_sporadic_Q(E, 19)
[Isogeny of degree 19
  from Elliptic Curve defined by y^2 + y = x^3 - 1862*x - 30956
       over Rational Field
    to Elliptic Curve defined by y^2 + y = x^3 - 672182*x + 212325489
       over Rational Field]
sage: isogenies_sporadic_Q(E)
[Isogeny of degree 19
  from Elliptic Curve defined by y^2 + y = x^3 - 1862*x - 30956
       over Rational Field
    to Elliptic Curve defined by y^2 + y = x^3 - 672182*x + 212325489
       over Rational Field]

sage: E = EllipticCurve([0, -1, 0, -6288, 211072])
sage: E.conductor()
19600
sage: isogenies_sporadic_Q(E,37)
[Isogeny of degree 37
  from Elliptic Curve defined by y^2 = x^3 - x^2 - 6288*x + 211072
       over Rational Field
    to Elliptic Curve defined by y^2 = x^3 - x^2 - 163137088*x - 801950801728
       over Rational Field]

sage: E = EllipticCurve([1, 1, 0, -25178045, 48616918750])
sage: E.conductor()
148225
sage: isogenies_sporadic_Q(E,37)
[Isogeny of degree 37
  from Elliptic Curve defined by y^2 + x*y = x^3 + x^2 - 25178045*x + 48616918750
       over Rational Field
    to Elliptic Curve defined by y^2 + x*y = x^3 + x^2 - 970*x - 13075
       over Rational Field]

sage: E = EllipticCurve([-3440, 77658])
sage: E.conductor()
118336
sage: isogenies_sporadic_Q(E,43)
[Isogeny of degree 43
```

(continues on next page)

(continued from previous page)

```
  from Elliptic Curve defined by y^2 = x^3 - 3440*x + 77658
       over Rational Field
    to Elliptic Curve defined by y^2 = x^3 - 6360560*x - 6174354606
       over Rational Field]

sage: E = EllipticCurve([-29480, -1948226])
sage: E.conductor()
287296
sage: isogenies_sporadic_Q(E,67)
[Isogeny of degree 67
 from Elliptic Curve defined by y^2 = x^3 - 29480*x - 1948226
      over Rational Field
   to Elliptic Curve defined by y^2 = x^3 - 132335720*x + 585954296438
      over Rational Field]

sage: E = EllipticCurve([-34790720, -78984748304])
sage: E.conductor()
425104
sage: isogenies_sporadic_Q(E,163)
[Isogeny of degree 163
  from Elliptic Curve defined by y^2 = x^3 - 34790720*x - 78984748304
       over Rational Field
    to Elliptic Curve defined by y^2 = x^3 - 924354639680*x + 342062961763303088
       over Rational Field]
```

CHAPTER

SEVENTEEN

MODULAR POLYNOMIALS FOR ELLIPTIC CURVES

For a positive integer ℓ, the classical modular polynomial $\Phi_\ell \in \mathbf{Z}[X, Y]$ is characterized by the property that its zero set is exactly the set of pairs of j-invariants connected by a cyclic ℓ-isogeny.

AUTHORS:

- Lorenz Panny (2023)

`sage.schemes.elliptic_curves.mod_poly.`**`classical_modular_polynomial`**(*l*, *j=None*)

Return the classical modular polynomial Φ_ℓ, either as a "generic" bivariate polynomial over $\mathbf{Z}$, or as an "instantiated" modular polynomial where one variable has been replaced by the given j-invariant.

Generic polynomials are cached up to a certain size of ℓ, which significantly accelerates subsequent invocations with the same ℓ. The default bound is $\ell \leq 100$, which can be adjusted using `classical_modular_polynomial.set_cache_bound()` with a different value. Beware that modular polynomials are very big objects and the amount of memory consumed by the cache will grow rapidly when the bound is set to a large value.

INPUT:

- `l` – positive integer.
- `j` – either `None` or a ring element:
  - if `None` is given, the original modular polynomial is returned as an element of $\mathbf{Z}[X, Y]$
  - if a ring element $j \in R$ is given, the evaluation $\Phi_\ell(j, Y)$ is returned as an element of the univariate polynomial ring $R[Y]$

ALGORITHMS:

- The Kohel database `ClassicalModularPolynomialDatabase`
- pari:polmodular

EXAMPLES:

```
sage: classical_modular_polynomial(2)
-X^2*Y^2 + X^3 + 1488*X^2*Y + 1488*X*Y^2 + Y^3 - 162000*X^2 + 40773375*X*Y -␣
↪162000*Y^2 + 8748000000*X + 8748000000*Y - 157464000000000
sage: j = Mod(1728, 419)
sage: classical_modular_polynomial(3, j)
Y^4 + 230*Y^3 + 84*Y^2 + 118*Y + 329
```

Increasing the cache size can be useful for repeated invocations:

```
sage: %timeit classical_modular_polynomial(101)                              #␣
↪not tested
6.11 s ± 1.21 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
sage: %timeit classical_modular_polynomial(101, GF(65537).random_element())  #␣
```

(continues on next page)

(continued from previous page)

```
↪not tested
5.43 s ± 2.71 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)

sage: classical_modular_polynomial.set_cache_bound(150)                    #␣
↪not tested
sage: %timeit classical_modular_polynomial(101)                            #␣
↪not tested
The slowest run took 10.35 times longer than the fastest. This could mean that an␣
↪intermediate result is being cached.
1.84 µs ± 1.84 µs per loop (mean ± std. dev. of 7 runs, 1 loop each)
sage: %timeit classical_modular_polynomial(101, GF(65537).random_element())  #␣
↪not tested
59.8 ms ± 29.4 µs per loop (mean ± std. dev. of 7 runs, 10 loops each)
```

CHAPTER

EIGHTEEN

ELLIPTIC CURVES OVER NUMBER FIELDS

18.1 Elliptic curves over the rational numbers

AUTHORS:

- William Stein (2005): first version
- William Stein (2006-02-26): fixed Lseries_extended which didn't work because of changes elsewhere in Sage.
- David Harvey (2006-09): Added padic_E2, padic_sigma, padic_height, padic_regulator methods.
- David Harvey (2007-02): reworked padic-height related code
- Christian Wuthrich (2007): added padic sha computation
- David Roe (2007-09): moved sha, l-series and p-adic functionality to separate files.
- John Cremona (2008-01)
- Tobias Nagel and Michael Mardaus (2008-07): added integral_points
- John Cremona (2008-07): further work on integral_points
- Christian Wuthrich (2010-01): moved Galois reps and modular parametrization in a separate file
- Simon Spicer (2013-03): Added code for modular degrees and congruence numbers of higher level
- Simon Spicer (2014-08): Added new analytic rank computation functionality

class sage.schemes.elliptic_curves.ell_rational_field.**EllipticCurve_rational_field**(*ainvs*, ***kwds*)

Bases: *EllipticCurve_number_field*

Elliptic curve over the Rational Field.

INPUT:

- `ainvs` – a list or tuple $[a_1, a_2, a_3, a_4, a_6]$ of Weierstrass coefficients

---

Note: This class should not be called directly; use `sage.constructor.EllipticCurve` to construct elliptic curves.

EXAMPLES:

Construction from Weierstrass coefficients (a-invariants), long form:

```
sage: E = EllipticCurve([1,2,3,4,5]); E
Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5 over Rational␣
↪Field
```

Construction from Weierstrass coefficients (a-invariants), short form (sets $a_1 = a_2 = a_3 = 0$):

```
sage: EllipticCurve([4,5]).ainvs()
(0, 0, 0, 4, 5)
```

Constructor from a Cremona label:

```
sage: EllipticCurve('389a1')
Elliptic Curve defined by y^2 + y = x^3 + x^2 - 2*x over Rational Field
```

Constructor from an LMFDB label:

```
sage: EllipticCurve('462.f3')
Elliptic Curve defined by y^2 + x*y = x^3 - 363*x + 1305 over Rational Field
```

CPS_height_bound()

Return the Cremona-Prickett-Siksek height bound. This is a floating point number B such that if P is a rational point on the curve, then $h(P) \leq \hat{h}(P) + B$, where $h(P)$ is the naive logarithmic height of P and $\hat{h}(P)$ is the canonical height.

See also:

silverman_height_bound() for a bound that also works for points over number fields.

EXAMPLES:

```
sage: E = EllipticCurve("11a")
sage: E.CPS_height_bound()
2.8774743273580445
sage: E = EllipticCurve("5077a")
sage: E.CPS_height_bound()
0.0
sage: E = EllipticCurve([1,2,3,4,1])
sage: E.CPS_height_bound()
Traceback (most recent call last):
...
RuntimeError: curve must be minimal.
sage: F = E.quadratic_twist(-19)
sage: F
Elliptic Curve defined by y^2 + x*y + y = x^3 - x^2 + 1376*x - 130 over␣
↪Rational Field
sage: F.CPS_height_bound()
0.6555158376972852
```

IMPLEMENTATION:

Call the corresponding mwrank C++ library function. Note that the formula in the [CPS2006] paper is given for number fields. It is only the implementation in Sage that restricts to the rational field.

Lambda(*s*, *prec*)

Return the value of the Lambda-series of the elliptic curve E at `s`, where `s` can be any complex number.

IMPLEMENTATION:

Fairly *slow* computation using the definitions implemented in Python.

Uses `prec` terms of the power series.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: E.Lambda(1.4 + 0.5*I, 50)
-0.354172680517... + 0.874518681720...*I
```

Np(*p*)

The number of points on E modulo p.

INPUT:

- `p` (int) – a prime, not necessarily of good reduction

OUTPUT:

(int) The number ofpoints on the reduction of E modulo p (including the singular point when p is a prime of bad reduction).

EXAMPLES:

```
sage: E = EllipticCurve([0, -1, 1, -10, -20])
sage: E.Np(2)
5
sage: E.Np(3)
5
sage: E.conductor()
11
sage: E.Np(11)
11
```

This even works when the prime is large:

```
sage: E = EllipticCurve('37a')
sage: E.Np(next_prime(10^30))
1000000000000001426441464441649
```

S_integral_points(*S*, *mw_base='auto'*, *both_signs=False*, *verbose=False*, *proof=None*)

Compute all S-integral points (up to sign) on this elliptic curve.

INPUT:

- `S` – list of primes
- `mw_base` – (default: `'auto'` - calls *gens()*) list of EllipticCurvePoint generating the Mordell-Weil group of E
- `both_signs` – boolean (default: `False`); if `True` the output contains both P and $-P$, otherwise only one of each pair
- `verbose` – boolean (default: `False`); if `True`, some details of the computation are output
- `proof` – boolean (default: `True`); if `True` ALL S-integral points will be returned. If False, the MW basis will be computed with the proof=False flag, and also the time-consuming final call to S_integral_x_coords_with_abs_bounded_by(abs_bound) is omitted. Use this only if the computation takes too long, but be warned that then it cannot be guaranteed that all S-integral points will be found.

OUTPUT:

A sorted list of all the S-integral points on E (up to sign unless both_signs is True)

Note: The complexity increases exponentially in the rank of curve E and in the length of S. The computation time (but not the output!) depends on the Mordell-Weil basis. If mw_base is given but is not a basis for the Mordell-Weil group (modulo torsion), S-integral points which are not in the subgroup generated by the given points will almost certainly not be listed.

EXAMPLES:

A curve of rank 3 with no torsion points:

```
sage: E = EllipticCurve([0,0,1,-7,6])
sage: P1 = E.point((2,0))
sage: P2 = E.point((-1,3))
sage: P3 = E.point((4,6))
sage: a = E.S_integral_points(S=[2,3], mw_base=[P1,P2,P3], verbose=True); a
max_S: 3 len_S: 3 len_tors: 1
lambda 0.485997517468...
k1,k2,k3,k4 7.65200453902598e234 1.31952866480763 3.54035317966420e9 2.
↪42767548272846e17
p= 2 : trying with p_prec =  30
mw_base_p_log_val =  [2, 2, 1]
min_psi =  2 + 2^3 + 2^6 + 2^7 + 2^8 + 2^9 + 2^11 + 2^12 + 2^13 + 2^16 + 2^17␣
↪+ 2^19 + 2^20 + 2^21 + 2^23 + 2^24 + 2^28 + O(2^30)
p= 3 : trying with p_prec =  30
mw_base_p_log_val =  [1, 2, 1]
min_psi =  3 + 3^2 + 2*3^3 + 3^6 + 2*3^7 + 2*3^8 + 3^9 + 2*3^11 + 2*3^12 +␣
↪2*3^13 + 3^15 + 2*3^16 + 3^18 + 2*3^19 + 2*3^22 + 2*3^23 + 2*3^24 + 2*3^27␣
↪+ 3^28 + 3^29 + O(3^30)
mw_base [(1 : -1 : 1), (2 : 0 : 1), (0 : -3 : 1)]
mw_base_log [0.667789378224099, 0.552642660712417, 0.818477222895703]
mp [5, 7]
mw_base_p_log [[2^2 + 2^3 + 2^6 + 2^7 + 2^8 + 2^9 + 2^14 + 2^15 + 2^18 + 2^19␣
↪+ 2^24 + 2^29 + O(2^30), 2^2 + 2^3 + 2^5 + 2^6 + 2^9 + 2^11 + 2^12 + 2^14 +␣
↪2^15 + 2^16 + 2^18 + 2^20 + 2^22 + 2^23 + 2^26 + 2^27 + 2^29 + O(2^30), 2 +␣
↪2^3 + 2^6 + 2^7 + 2^8 + 2^9 + 2^11 + 2^12 + 2^13 + 2^16 + 2^17 + 2^19 + 2^
↪20 + 2^21 + 2^23 + 2^24 + 2^28 + O(2^30)], [2*3^2 + 2*3^5 + 2*3^6 + 2*3^7 +␣
↪3^8 + 3^9 + 2*3^10 + 3^12 + 2*3^14 + 3^15 + 3^17 + 2*3^19 + 2*3^23 + 3^25 +␣
↪3^28 + O(3^30), 2*3 + 2*3^2 + 2*3^3 + 2*3^4 + 2*3^6 + 2*3^7 + 2*3^8 + 3^10␣
↪+ 2*3^12 + 3^13 + 2*3^14 + 3^15 + 3^18 + 3^22 + 3^25 + 2*3^26 + 3^27 + 3^28␣
↪+ O(3^30), 3 + 3^2 + 2*3^3 + 3^6 + 2*3^7 + 2*3^8 + 3^9 + 2*3^11 + 2*3^12 +␣
↪2*3^13 + 3^15 + 2*3^16 + 3^18 + 2*3^19 + 2*3^22 + 2*3^23 + 2*3^24 + 2*3^27␣
↪+ 3^28 + 3^29 + O(3^30)]]
k5,k6,k7 0.321154513240... 1.55246328915... 0.161999172489...
initial bound 2.8057927340...e117
bound_list [58, 58, 58]
bound_list [8, 9, 9]
bound_list [9, 7, 7]
starting search of points using coefficient bound  9
x-coords of S-integral points via linear combination of mw_base and torsion:
[-3, -26/9, -8159/2916, -2759/1024, -151/64, -1343/576, -2, -7/4, -1, -47/256,
↪ 0, 1/4, 4/9, 9/16, 58/81, 7/9, 6169/6561, 1, 17/16, 2, 33/16, 172/81, 9/4,␣
↪25/9, 3, 31/9, 4, 25/4, 1793/256, 8, 625/64, 11, 14, 21, 37, 52, 6142/81,␣
↪93, 4537/36, 342, 406, 816, 207331217/4096]
starting search of extra S-integer points with absolute value bounded by 3.
↪89321964979420
x-coords of points with bounded absolute value
[-3, -2, -1, 0, 1, 2]
```

(continues on next page)

(continued from previous page)

```
Total number of S-integral points: 43
[(-3 : -1 : 1),
(-26/9 : -55/27 : 1),
(-8159/2916 : -390925/157464 : 1),
(-2759/1024 : -93587/32768 : 1),
(-151/64 : -1845/512 : 1),
(-1343/576 : -50399/13824 : 1),
(-2 : -4 : 1),
(-7/4 : -33/8 : 1),
(-1 : -4 : 1),
(-47/256 : -13287/4096 : 1),
(0 : -3 : 1),
(1/4 : -21/8 : 1),
(4/9 : -62/27 : 1),
(9/16 : -133/64 : 1),
(58/81 : -1288/729 : 1),
(7/9 : -44/27 : 1),
(6169/6561 : -641312/531441 : 1),
(1 : -1 : 1),
(17/16 : -39/64 : 1),
(2 : -1 : 1),
(33/16 : -81/64 : 1),
(172/81 : -1079/729 : 1),
(9/4 : -15/8 : 1),
(25/9 : -91/27 : 1),
(3 : -4 : 1),
(31/9 : -143/27 : 1),
(4 : -7 : 1),
(25/4 : -119/8 : 1),
(1793/256 : -73087/4096 : 1),
(8 : -22 : 1),
(625/64 : -15351/512 : 1),
(11 : -36 : 1),
(14 : -52 : 1),
(21 : -96 : 1),
(37 : -225 : 1),
(52 : -375 : 1),
(6142/81 : -481429/729 : 1),
(93 : -897 : 1),
(4537/36 : -305641/216 : 1),
(342 : -6325 : 1),
(406 : -8181 : 1),
(816 : -23310 : 1),
(207331217/4096 : -2985362435769/262144 : 1)]
```

It is not necessary to specify mw_base; if it is not provided, then the Mordell-Weil basis must be computed, which may take much longer.

```
sage: a = E.S_integral_points([2,3])
sage: len(a)
43
```

An example with negative discriminant:

```
sage: EllipticCurve('900d1').S_integral_points([17], both_signs=True)
[(-11 : -27 : 1), (-11 : 27 : 1), (-4 : -34 : 1), (-4 : 34 : 1), (4 : -18 :␣
↪1),
```

(continues on next page)

(continued from previous page)

```
 (4 : 18 : 1), (2636/289 : -98786/4913 : 1), (2636/289 : 98786/4913 : 1),
 (16 : -54 : 1), (16 : 54 : 1)]
```

Output checked with Magma (corrected in 3 cases):

```
sage: [len(e.S_integral_points([2], both_signs=False)) for e in cremona_
↪curves([11..100])] # long time (17s on sage.math, 2011)
[2, 0, 2, 3, 3, 1, 3, 1, 3, 5, 3, 5, 4, 1, 1, 2, 2, 2, 3, 1, 2, 1, 0, 1, 3, 3,
↪ 1, 1, 5, 3, 4, 2, 1, 1, 5, 3, 2, 2, 1, 1, 1, 0, 1, 3, 0, 1, 0, 1, 1, 3, 7,␣
↪1, 3, 3, 3, 1, 1, 2, 3, 1, 2, 3, 1, 2, 1, 3, 3, 1, 1, 1, 0, 1, 3, 3, 1, 1,␣
↪7, 1, 0, 1, 1, 0, 1, 2, 0, 3, 1, 2, 1, 3, 1, 2, 2, 4, 5, 3, 2, 1, 1, 6, 1,␣
↪0, 1, 3, 1, 3, 3, 1, 1, 1, 1, 1, 3, 1, 5, 1, 2, 4, 1, 1, 1, 1, 1, 0, 1, 0,␣
↪2, 2, 0, 0, 1, 0, 1, 1, 6, 1, 0, 1, 1, 0, 4, 3, 1, 2, 1, 2, 3, 1, 1, 1, 1,␣
↪8, 3, 1, 2, 1, 2, 0, 8, 2, 0, 6, 2, 3, 1, 1, 1, 3, 1, 3, 2, 1, 3, 1, 2, 1,␣
↪6, 9, 3, 3, 1, 1, 2, 3, 1, 1, 5, 5, 1, 1, 0, 1, 1, 2, 3, 1, 1, 2, 3, 1, 3,␣
↪1, 1, 1, 1, 0, 0, 1, 3, 3, 1, 3, 1, 1, 2, 2, 0, 0, 6, 1, 0, 1, 1, 1, 1, 3,␣
↪1, 2, 6, 3, 1, 2, 2, 1, 1, 1, 1, 7, 5, 4, 3, 3, 1, 1, 1, 1, 1, 1, 8, 5, 1,␣
↪1, 3, 3, 1, 1, 3, 3, 1, 1, 2, 3, 6, 1, 1, 7, 3, 3, 4, 5, 9, 6, 1, 0, 7, 1,␣
↪1, 3, 1, 1, 2, 3, 1, 2, 1, 1, 1, 1, 1, 1, 1, 7, 8, 2, 3, 1, 1, 1, 1, 0, 0,␣
↪0, 1, 1, 1, 1]
```

An example from [PZGH1999]:

```
sage: E = EllipticCurve([0,0,0,-172,505])
sage: E.rank(), len(E.S_integral_points([3,5,7]))  # long time (5s on sage.
↪math, 2011)
(4, 72)
```

This is curve "7690e1" which failed until Issue #4805 was fixed:

```
sage: EllipticCurve([1,1,1,-301,-1821]).S_integral_points([13,2])
[(-13 : -4 : 1), (-9 : -12 : 1), (-7 : 2 : 1), (21 : -52 : 1),
 (23 : -76 : 1), (63 : -516 : 1), (71 : -620 : 1), (87 : -844 : 1),
 (2711 : -142540 : 1), (7323 : -630376 : 1), (17687 : -2361164 : 1)]
```

- Some parts of this implementation are partially based on the function integral_points()

AUTHORS:

- Tobias Nagel (2008-12)
- Michael Mardaus (2008-12)
- John Cremona (2008-12)

abelian_variety()

Return self as a modular abelian variety.

OUTPUT:

- a modular abelian variety

EXAMPLES:

```
sage: E = EllipticCurve('11a')
sage: E.abelian_variety()
Abelian variety J0(11) of dimension 1
```

(continues on next page)

(continued from previous page)

```
sage: E = EllipticCurve('33a')
sage: E.abelian_variety()
Abelian subvariety of dimension 1 of J0(33)
```

an(*n*)

The n-th Fourier coefficient of the modular form corresponding to this elliptic curve, where n is a positive integer.

EXAMPLES:

```
sage: E = EllipticCurve('37a1')
sage: [E.an(n) for n in range(20) if n>0]
[1, -2, -3, 2, -2, 6, -1, 0, 6, 4, -5, -6, -2, 2, 6, -4, 0, -12, 0]
```

analytic_rank(*algorithm='pari'*, *leading_coefficient=False*)

Return an integer that is *probably* the analytic rank of this elliptic curve.

INPUT:

- algorithm – (default: 'pari'), String
  - 'pari' – use the PARI library function.
  - 'sympow' – use Watkins's program sympow
  - 'rubinstein' – use Rubinstein's L-function C++ program lcalc.
  - 'magma' – use MAGMA
  - 'zero_sum' – Use the rank bounding zero sum method implemented in *analytic_rank_upper_bound()*
  - 'all' – compute with PARI, sympow and lcalc, check that the answers agree, and return the common answer.
- leading_coefficient – (default: False) Boolean; if set to True, return a tuple $(rank, lead)$ where $lead$ is the value of the first non-zero derivative of the L-function of the elliptic curve. Only implemented for algorithm='pari'.

Note: If the curve is loaded from the large Cremona database, then the modular degree is taken from the database.

Of the first three algorithms above, probably Rubinstein's is the most efficient (in some limited testing done). The zero sum method is often *much* faster, but can return a value which is strictly larger than the analytic rank. For curves with conductor <=10^9 using default parameters, testing indicates that for 99.75% of curves the returned rank bound is the true rank.

Note: If you use set_verbose(1), extra information about the computation will be printed when algorithm='zero_sum'.

Note: It is an open problem to *prove* that *any* particular elliptic curve has analytic rank ≥ 4.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: E.analytic_rank(algorithm='pari')
2
sage: E.analytic_rank(algorithm='rubinstein')
2
sage: E.analytic_rank(algorithm='sympow')
2
sage: E.analytic_rank(algorithm='magma')     # optional - magma
2
sage: E.analytic_rank(algorithm='zero_sum')
2
sage: E.analytic_rank(algorithm='all')
2
```

With the optional parameter leading_coefficient set to `True`, a tuple of both the analytic rank and the leading term of the L-series at $s = 1$ is returned. This only works for `algorithm=='pari'`:

```
sage: EllipticCurve([0,-1,1,-10,-20]).analytic_rank(leading_coefficient=True)
(0, 0.25384186085591068...)
sage: EllipticCurve([0,0,1,-1,0]).analytic_rank(leading_coefficient=True)
(1, 0.30599977383405230...)
sage: EllipticCurve([0,1,1,-2,0]).analytic_rank(leading_coefficient=True)
(2, 1.518633000576853...)
sage: EllipticCurve([0,0,1,-7,6]).analytic_rank(leading_coefficient=True)
(3, 10.39109940071580...)
sage: EllipticCurve([0,0,1,-7,36]).analytic_rank(leading_coefficient=True)
(4, 196.170903794579...)
```

analytic_rank_upper_bound(*max_Delta=None*, *adaptive=True*, *N=None*, *root_number='compute'*, *bad_primes=None*, *ncpus=None*)

Return an upper bound for the analytic rank of self, conditional on the Generalized Riemann Hypothesis, via computing the zero sum $\sum_\gamma f(\Delta\gamma)$, where γ ranges over the imaginary parts of the zeros of $L(E, s)$ along the critical strip, $f(x) = (\sin(\pi x)/(\pi x))^2$, and Δ is the tightness parameter whose maximum value is specified by `max_Delta`. This computation can be run on curves with very large conductor (so long as the conductor is known or quickly computable) when Δ is not too large (see below). Uses Bober's rank bounding method as described in [Bob2013].

INPUT:

- `max_Delta` – (default: `None`) If not `None`, a positive real value specifying the maximum Delta value used in the zero sum; larger values of Delta yield better bounds - but runtime is exponential in Delta. If left as `None`, Delta is set to $\min\{\frac{1}{\pi}(\log(N + 1000)/2 - \log(2\pi) - \eta), 2.5\}$, where N is the conductor of the curve attached to self, and η is the Euler-Mascheroni constant $= 0.5772...$; the crossover point is at conductor around $8.3 \cdot 10^8$. For the former value, empirical results show that for about 99.7% of all curves the returned value is the actual analytic rank.
- `adaptive` – (default: `True`) boolean
  - `True` – the computation is first run with small and then successively larger Δ values up to max_Delta. If at any point the computed bound is 0 (or 1 when root_number is -1 or True), the computation halts and that value is returned; otherwise the minimum of the computed bounds is returned.
  - `False` – the computation is run a single time with Δ equal to `max_Delta`, and the resulting bound returned.
- `N` – (default: `None`) If not `None`, a positive integer equal to the conductor of `self`. This is passable so that rank estimation can be done for curves whose (large) conductor has been precomputed.
- `root_number` – (default: "compute") string or integer

  - `"compute"` – the root number of self is computed and used to (possibly) lower the analytic rank estimate by 1.
  - `"ignore"` – the above step is omitted
  - `1` – this value is assumed to be the root number of self. This is passable so that rank estimation can be done for curves whose root number has been precomputed.
  - `-1` – this value is assumed to be the root number of self. This is passable so that rank estimation can be done for curves whose root number has been precomputed.
- `bad_primes` – (default: `None`) If not `None`, a list of the primes of bad reduction for the curve attached to self. This is passable so that rank estimation can be done for curves of large conductor whose bad primes have been precomputed.
- `ncpus` – (default: `None`) If not `None`, a positive integer defining the maximum number of CPUs to be used for the computation. If left as None, the maximum available number of CPUs will be used. Note: Due to parallelization overhead, multiple processors will only be used for Delta values ≥ 1.75.

Note: Output will be incorrect if the incorrect conductor or root number is specified.

Warning: Zero sum computation time is exponential in the tightness parameter Δ, roughly doubling for every increase of 0.1 thereof. Using $\Delta = 1$ (and adaptive=False) will yield a runtime of a few milliseconds; $\Delta = 2$ takes a few seconds, and $\Delta = 3$ may take upwards of an hour. Increase beyond this at your own risk!

OUTPUT:

A non-negative integer greater than or equal to the analytic rank of self.

Note: If you use set_verbose(1), extra information about the computation will be printed.

See also:

`LFunctionZeroSum()` *`root_number()`* `set_verbose()`

EXAMPLES:

For most elliptic curves with small conductor the central zero(s) of $L_E(s)$ are fairly isolated, so small values of Δ will yield tight rank estimates.

```
sage: E = EllipticCurve("11a")
sage: E.rank()
0
sage: E.analytic_rank_upper_bound(max_Delta=1, adaptive=False)
0
sage: E = EllipticCurve([-39,123])
sage: E.rank()
1
sage: E.analytic_rank_upper_bound(max_Delta=1, adaptive=True)
1
```

This is especially true for elliptic curves with large rank.

```
sage: for r in range(9):
....:     E = elliptic_curves.rank(r)[0]
....:     print((r, E.analytic_rank_upper_bound(max_Delta=1,
....:                                           adaptive=False,
....:                                           root_number="ignore")))
(0, 0)
(1, 1)
(2, 2)
(3, 3)
(4, 4)
(5, 5)
(6, 6)
(7, 7)
(8, 8)
```

However, some curves have L-functions with low-lying zeroes, and for these larger values of Δ must be used to get tight estimates.

```
sage: E = EllipticCurve("974b1")
sage: r = E.rank(); r
0
sage: E.analytic_rank_upper_bound(max_Delta=1, root_number="ignore")
1
sage: E.analytic_rank_upper_bound(max_Delta=1.3, root_number="ignore")
0
```

Knowing the root number of E allows us to use smaller Delta values to get tight bounds, thus speeding up runtime considerably.

```
sage: E.analytic_rank_upper_bound(max_Delta=0.6, root_number="compute")
0
```

There are a small number of curves which have pathologically low-lying zeroes. For these curves, this method will produce a bound that is strictly larger than the analytic rank, unless very large values of Delta are used. The following curve ("256944c1" in the Cremona tables) is a rank 0 curve with a zero at 0.0256…; the smallest Delta value for which the zero sum is strictly less than 2 is ~2.815.

```
sage: E = EllipticCurve([0, -1, 0, -7460362000712, -7842981500851012704])
sage: N, r = E.conductor(), E.analytic_rank(); N, r
(256944, 0)
sage: E.analytic_rank_upper_bound(max_Delta=1, adaptive=False)
2
sage: E.analytic_rank_upper_bound(max_Delta=2, adaptive=False)
2
```

This method is can be called on curves with large conductor.

```
sage: E = EllipticCurve([-2934,19238])
sage: E.analytic_rank_upper_bound()
1
```

And it can bound rank on curves with *very* large conductor, so long as you know beforehand/can easily compute the conductor and primes of bad reduction less than $e^{2\pi\Delta}$. The example below is of the rank 28 curve discovered by Elkies that is the elliptic curve of (currently) largest known rank.

```
sage: a4 = -20067762415575526585033208209338542750930230312178956502
sage: a6 =␣
```

(continues on next page)

(continued from previous page)

```
↪3448161179503055646703298569039072037485594435931918036126600829629193944873224342 9
sage: E = EllipticCurve([1, -1, 1, a4, a6])
sage: bad_primes = [2, 3, 5, 7, 11, 13, 17, 19, 48463]
sage: N =␣
↪3455601108357547341532253864901605231198511505793733138900595189472144724781456635380154 14
sage: E.analytic_rank_upper_bound(max_Delta=2.37, adaptive=False,  # long time
....:                             N=N, root_number=1,
....:                             bad_primes=bad_primes, ncpus=2)
32
```

anlist(*n*, *python_ints=False*)

The Fourier coefficients up to and including a_n of the modular form attached to this elliptic curve. The i-th element of the return list is `a[i]`.

INPUT:

- `n` – integer
- `python_ints` – bool (default: `False`); if `True` return a list of Python ints instead of Sage integers

OUTPUT: list of integers

EXAMPLES:

```
sage: E = EllipticCurve([0, -1, 1, -10, -20])
sage: E.anlist(3)
[0, 1, -2, -1]
```

```
sage: E = EllipticCurve([0,1])
sage: E.anlist(20)
[0, 1, 0, 0, 0, 0, 0, -4, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 8, 0]
```

antilogarithm(*z*, *max_denominator=None*)

Return the rational point (if any) associated to this complex number; the inverse of the elliptic logarithm function.

INPUT:

- `z` – a complex number representing an element of $\mathbf{C}/L$ where L is the period lattice of the elliptic curve
- `max_denominator` – integer (optional); parameter controlling the attempted conversion of real numbers to rationals. If not given, `simplest_rational()` will be used; otherwise, `nearby_rational()` will be used with this value of `max_denominator`.

OUTPUT:

- point on the curve: the rational point which is the image of z under the Weierstrass parametrization, if it exists and can be determined from z and the given value of max_denominator (if any); otherwise a `ValueError` exception is raised.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: P = E(-1,1)
sage: z = P.elliptic_logarithm()
sage: E.antilogarithm(z)
(-1 : 1 : 1)
sage: Q = E(0,-1)
sage: z = Q.elliptic_logarithm()
```

(continues on next page)

(continued from previous page)

```
sage: E.antilogarithm(z)
Traceback (most recent call last):
...
ValueError: approximated point not on the curve
sage: E.antilogarithm(z, max_denominator=10)
(0 : -1 : 1)

sage: E = EllipticCurve('11a1')
sage: w1,w2 = E.period_lattice().basis()
sage: [E.antilogarithm(a*w1/5,1) for a in range(5)]
[(0 : 1 : 0), (16 : -61 : 1), (5 : -6 : 1), (5 : 5 : 1), (16 : 60 : 1)]
```

ap(*p*)

The p-th Fourier coefficient of the modular form corresponding to this elliptic curve, where p is prime.

EXAMPLES:

```
sage: E = EllipticCurve('37a1')
sage: [E.ap(p) for p in prime_range(50)]
[-2, -3, -2, -1, -5, -2, 0, 0, 2, 6, -4, -1, -9, 2, -9]
```

aplist(*n*, *python_ints=False*)

The Fourier coefficients a_p of the modular form attached to this elliptic curve, for all primes $p \leq n$.

INPUT:

- n – integer
- python_ints – bool (default: False); if True return a list of Python ints instead of Sage integers

OUTPUT: list of integers

EXAMPLES:

```
sage: e = EllipticCurve('37a')
sage: e.aplist(1)
[]
sage: e.aplist(2)
[-2]
sage: e.aplist(10)
[-2, -3, -2, -1]
sage: v = e.aplist(13); v
[-2, -3, -2, -1, -5, -2]
sage: type(v[0])
<... 'sage.rings.integer.Integer'>
sage: type(e.aplist(13, python_ints=True)[0])
<... 'int'>
```

cm_discriminant()

Return the associated quadratic discriminant if this elliptic curve has Complex Multiplication over the algebraic closure.

A ValueError is raised if the curve does not have CM (see the function *has_cm()*).

EXAMPLES:

```
sage: E = EllipticCurve('32a1')
sage: E.cm_discriminant()
```

(continues on next page)

(continued from previous page)

```
-4
sage: E = EllipticCurve('121b1')
sage: E.cm_discriminant()
-11
sage: E = EllipticCurve('37a1')
sage: E.cm_discriminant()
Traceback (most recent call last):
...
ValueError: Elliptic Curve defined by y^2 + y = x^3 - x
over Rational Field does not have CM
```

conductor(*algorithm='pari'*)

Return the conductor of the elliptic curve.

INPUT:

- `algorithm` – str, (default: "pari")
  - `"pari"` – use the PARI C-library pari:ellglobalred implementation of Tate's algorithm
  - `"mwrank"` – use Cremona's mwrank implementation of Tate's algorithm; can be faster if the curve has integer coefficients (TODO: limited to small conductor until mwrank gets integer factorization)
  - `"gp"` – use the GP interpreter
  - `"generic"` – use the general number field implementation
  - `"all"` – use all four implementations, verify that the results are the same (or raise an error), and output the common value

EXAMPLES:

```
sage: E = EllipticCurve([1, -1, 1, -29372, -1932937])
sage: E.conductor(algorithm="pari")
3006
sage: E.conductor(algorithm="mwrank")
3006
sage: E.conductor(algorithm="gp")
3006
sage: E.conductor(algorithm="generic")
3006
sage: E.conductor(algorithm="all")
3006
```

Note: The conductor computed using each algorithm is cached separately. Thus calling `E.conductor('pari')`, then `E.conductor('mwrank')` and getting the same result checks that both systems compute the same answer.

congruence_number(*M=1*)

The case $M == 1$ corresponds to the classical definition of congruence number: Let X be the subspace of $S_2(\Gamma_0(N))$ spanned by the newform associated with this elliptic curve, and Y be orthogonal complement of X under the Petersson inner product. Let S_X and S_Y be the intersections of X and Y with $S_2(\Gamma_0(N), \mathbf{Z})$. The congruence number is defined to be $[S_X \oplus S_Y : S_2(\Gamma_0(N), \mathbf{Z})]$. It measures congruences between f and elements of $S_2(\Gamma_0(N), \mathbf{Z})$ orthogonal to f.

The congruence number for higher levels, when M>1, is defined as above, but instead considers X to be the subspace of $S_2(\Gamma_0(MN))$ spanned by embeddings into $S_2(\Gamma_0(MN))$ of the newform associated with this

elliptic curve; this subspace has dimension $\sigma_0(M)$, i.e. the number of divisors of M. Let Y be the orthogonal complement in $S_2(\Gamma_0(MN))$ of X under the Petersson inner product, and S_X and S_Y the intersections of X and Y with $S_2(\Gamma_0(MN), \mathbf{Z})$ respectively. Then the congruence number at level MN is $[S_X \oplus S_Y : S_2(\Gamma_0(MN), \mathbf{Z})]$.

INPUT:

- `M` – non-negative integer; congruence number is computed at level MN, where N is the conductor of `self`

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.congruence_number()
2
sage: E.congruence_number()
2
sage: E = EllipticCurve('54b')
sage: E.congruence_number()
6
sage: E.modular_degree()
2
sage: E = EllipticCurve('242a1')
sage: E.modular_degree()
16
sage: E.congruence_number()  # long time (4s on sage.math, 2011)
176
```

Higher level cases:

```
sage: E = EllipticCurve('11a')
sage: for M in range(1,11): print(E.congruence_number(M)) # long time (20s on␣
↪2009 MBP)
1
1
3
2
7
45
12
4
18
245
```

It is a theorem of Ribet that the congruence number (at level N) is equal to the modular degree in the case of square free conductor. It is a conjecture of Agashe, Ribet, and Stein that $ord_p(c_f/m_f) \leq ord_p(N)/2$.

cremona_label (*space=False*)

Return the Cremona label associated to (the minimal model) of this curve, if it is known. If not, raise a `LookupError` exception.

EXAMPLES:

```
sage: E = EllipticCurve('389a1')
sage: E.cremona_label()
'389a1'
```

The default database only contains conductors up to 10000, so any curve with conductor greater than that will cause an error to be raised. The optional package `database_cremona_ellcurve` contains many more curves.

```
sage: E = EllipticCurve([1, -1, 0, -79, 289])
sage: E.conductor()
234446
sage: E.cremona_label()  # optional - database_cremona_ellcurve
'234446a1'
sage: E = EllipticCurve((0, 0, 1, -79, 342))
sage: E.conductor()
19047851
sage: E.cremona_label()
Traceback (most recent call last):
...
LookupError: Cremona database does not contain entry for
Elliptic Curve defined by y^2 + y = x^3 - 79*x + 342 over Rational Field
```

database_attributes()

Return a dictionary containing information about `self` in the elliptic curve database.

If there is no elliptic curve isomorphic to `self` in the database, a `LookupError` is raised.

EXAMPLES:

```
sage: E = EllipticCurve((0, 0, 1, -1, 0))
sage: data = E.database_attributes()
sage: data['conductor']
37
sage: data['cremona_label']
'37a1'
sage: data['rank']
1
sage: data['torsion_order']
1

sage: E = EllipticCurve((8, 13, 21, 34, 55))
sage: E.database_attributes()
Traceback (most recent call last):
...
LookupError: Cremona database does not contain entry for Elliptic Curve
defined by y^2 + 8*x*y + 21*y = x^3 + 13*x^2 + 34*x + 55 over Rational Field
```

database_curve()

Return the curve in the elliptic curve database isomorphic to this curve, if possible. Otherwise raise a `LookupError` exception.

Since Issue #11474, this returns exactly the same curve as *minimal_model()*; the only difference is the additional work of checking whether the curve is in the database.

EXAMPLES:

```
sage: E = EllipticCurve([0,1,2,3,4])
sage: E.database_curve()
Elliptic Curve defined by y^2  = x^3 + x^2 + 3*x + 5 over Rational Field
```

Note: The model of the curve in the database can be different from the Weierstrass model for this curve, e.g., database models are always minimal.

elliptic_exponential(*z*, *embedding=None*)

Compute the elliptic exponential of a complex number with respect to the elliptic curve.

INPUT:

- `z` – a complex number
- `embedding` – ignored (for compatibility with the period_lattice function for elliptic_curve_number_field)

OUTPUT:

The image of z modulo L under the Weierstrass parametrization $\mathbf{C}/L \to E(\mathbf{C})$.

Note: The precision is that of the input `z`, or the default precision of 53 bits if `z` is exact.

EXAMPLES:

```
sage: E = EllipticCurve([1,1,1,-8,6])
sage: P = E([1,-2])
sage: z = P.elliptic_logarithm() # default precision is 100 here
sage: E.elliptic_exponential(z)
(1.0000000000000000000000000000 : -2.0000000000000000000000000000 : 1.
↪0000000000000000000000000000)
sage: z = E([1,-2]).elliptic_logarithm(precision=201)
sage: E.elliptic_exponential(z)
(1.0000000000000000000000000000000000000000000000000000000000 : -2.
↪0000000000000000000000000000000000000000000000000000000000 : 1.
↪0000000000000000000000000000000000000000000000000000000000)
```

```
sage: E = EllipticCurve('389a')
sage: Q = E([3,5])
sage: E.elliptic_exponential(Q.elliptic_logarithm())
(3.0000000000000000000000000000 : 5.0000000000000000000000000000 : 1.
↪0000000000000000000000000000)
sage: P = E([-1,1])
sage: P.elliptic_logarithm()
0.47934825019021931612953301006 + 0.98586885077582410221120384908*I
sage: E.elliptic_exponential(P.elliptic_logarithm())
(-1.0000000000000000000000000000 : 1.0000000000000000000000000000 : 1.
↪0000000000000000000000000000)
```

Some torsion examples:

```
sage: w1,w2 = E.period_lattice().basis()
sage: E.two_division_polynomial().roots(CC,multiplicities=False)
[-2.0403022002854..., 0.13540924022175..., 0.90489296006371...]
sage: [E.elliptic_exponential((a*w1+b*w2)/2)[0] for a,b in [(0,1),(1,1),(1,
↪0)]]
[-2.0403022002854..., 0.13540924022175..., 0.90489296006371...]

sage: E.division_polynomial(3).roots(CC,multiplicities=False)
[-2.88288879135...,
 1.39292799513...,
 0.078313731444316... - 0.492840991709...*I,
 0.078313731444316... + 0.492840991709...*I]
sage: [E.elliptic_exponential((a*w1+b*w2)/3)[0] for a,b in [(0,1),(1,0),(1,1),
↪(2,1)]]
```

(continues on next page)

(continued from previous page)

```
[-2.8828887913533..., 1.39292799513138,
 0.0783137314443... - 0.492840991709...*I,
 0.0783137314443... + 0.492840991709...*I]
```

Observe that this is a group homomorphism (modulo rounding error):

```
sage: z = CC.random_element()
sage: v = 2 * E.elliptic_exponential(z)
sage: w = E.elliptic_exponential(2 * z)
sage: def err(a, b):
....:     err = abs(a - b)
....:     if a + b:
....:         err = min(err, err / abs(a + b))
....:     return err
sage: err(v[0], w[0]) + err(v[1], w[1])  # abs tol 1e-13
0.0
```

eval_modular_form(*points*, *order*)

Evaluate the modular form of this elliptic curve at points in $\mathbf{C}$.

INPUT:

- `points` – a list of points in the upper half-plane
- `order` – a nonnegative integer

The `order` parameter is the number of terms used in the summation.

OUTPUT: A list of values for s in `points`

EXAMPLES:

```
sage: E = EllipticCurve('37a1')
sage: E.eval_modular_form([1.5+I,2.0+I,2.5+I],100)
[-0.0018743978548152085...,
 0.0018604485340371083...,
-0.0018743978548152085...]

sage: E.eval_modular_form(2.1+I, 100) # abs tol 1e-16
[0.0015086436275726707 9 + 0.0010910034111344984 5*I]
```

faltings_height(*stable=False*, *prec=None*)

Return the Faltings height (stable or unstable) of this elliptic curve.

INPUT:

- `stable` – boolean (default: `False`); if `True`, return the *stable* Faltings height, otherwise the unstable height
- `prec` – integer (default: `None`); bit precision of output; if `None`, use standard precision (53 bits)

OUTPUT:

(real) the Faltings height of this elliptic curve.

Note: Different authors normalise the Faltings height differently. We use the formula $-\frac{1}{2}\log(A)$, where A is the area of the fundamental period parallelogram; some authors use $-\frac{1}{2\pi}\log(A)$ instead.

The unstable Faltings height does depend on the model. The *stable* Faltings height is defined to be

$$\frac{1}{12}\log \mathrm{denom}(j) - \frac{1}{12}\log|\Delta| - \frac{1}{2}\log A,$$

This is independent of the model. For the minimal model of a semistable elliptic curve, we have $\mathrm{denom}(j) = |\Delta|$, and the stable and unstable heights agree.

EXAMPLES:

```
sage: E = EllipticCurve('32a1')
sage: E.faltings_height()
-0.617385745351564
sage: E.faltings_height(stable=True)
-1.31053292591151
```

These differ since the curve is not semistable:

```
sage: E.is_semistable()
False
```

If the model is changed, the Faltings height changes but the stable height does not. It is reduced by $\log(u)$ where u is the scale factor:

```
sage: E1 = E.change_weierstrass_model([10,0,0,0])
sage: E1.faltings_height()
-2.91997083834561
sage: E1.faltings_height(stable=True)
-1.31053292591151
sage: E.faltings_height() - log(10.0)
-2.91997083834561
```

For a semistable curve (that is, one with squarefree conductor), the stable and unstable heights are equal. Here we also show that one can specify the (bit) precision of the result:

```
sage: E = EllipticCurve('210a1')
sage: E.is_semistable()
True
sage: E.faltings_height(prec=100)
-0.043427311858075396288912139225
sage: E.faltings_height(stable=True, prec=100)
-0.043427311858075396288912139225
```

galois_representation()

The compatible family of the Galois representation attached to this elliptic curve.

Given an elliptic curve E over $\mathbf{Q}$ and a rational prime number p, the p^n-torsion $E[p^n]$ points of E is a representation of the absolute Galois group of $\mathbf{Q}$. As n varies we obtain the Tate module T_pE which is a a representation of G_K on a free $\mathbf{Z}_p$-module of rank 2. As p varies the representations are compatible.

EXAMPLES:

```
sage: rho = EllipticCurve('11a1').galois_representation()
sage: rho
Compatible family of Galois representations associated to the
 Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field
sage: rho.is_irreducible(7)
True
```

(continues on next page)

(continued from previous page)

```
sage: rho.is_irreducible(5)
False
sage: rho.is_surjective(11)
True
sage: rho.non_surjective()
[5]
sage: rho = EllipticCurve('37a1').galois_representation()
sage: rho.non_surjective()
[]
sage: rho = EllipticCurve('27a1').galois_representation()
sage: rho.is_irreducible(7)
True
sage: rho.non_surjective()    # cm-curve
[0]
```

gens (*proof=None*, ***kwds*)

Return generators for the Mordell-Weil group $E(Q)$ *modulo* torsion.

INPUT:

- `proof` – bool or None (default None), see `proof.elliptic_curve` or `sage.structure.proof`
- `verbose` – (default: None), if specified changes the verbosity of mwrank computations
- `rank1_search` – (default: 10), if the curve has analytic rank 1, try to find a generator by a direct search up to this logarithmic height. If this fails, the usual mwrank procedure is called.
- algorithm – one of the following:
  - `'mwrank_shell'` (default) – call mwrank shell command
  - `'mwrank_lib'` – call mwrank C library
  - `'pari'` – use ellrank in pari
- `only_use_mwrank` – bool (default: `True`) if False, first attempts to use more naive, natively implemented methods
- `use_database` – bool (default: `True`) if True, attempts to find curve and gens in the (optional) database
- `descent_second_limit` – (default: 12) used in 2-descent
- `sat_bound` – (default: 1000) bound on primes used in saturation. If the computed bound on the index of the points found by two-descent in the Mordell-Weil group is greater than this, a warning message will be displayed.
- `pari_effort` – (default: 0) parameter used in when the algorithm `pari` is chosen. It measure of the effort done to find rational points. Values up to 10 can be chosen, the running times increase roughly like the cube of the effort value.

OUTPUT:

- `generators` – list of generators for the Mordell-Weil group modulo torsion

---

Note: If you call this with `proof=False`, then you can use the *`gens_certain()`* method to find out afterwards whether the generators were proved.

IMPLEMENTATION: Uses Cremona's mwrank C++ library or ellrank in pari.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: E.gens()                  # random output
[(-1 : 1 : 1), (0 : 0 : 1)]
sage: E.gens(algorithm="pari")     # random output
[(5/4 : 5/8 : 1), (0 : 0 : 1)]
sage: E = EllipticCurve([0,2429469980725060,0,2751307033881721368336477563 88,
↪0])
sage: len(E.gens(algorithm="pari"))  # not tested (takes too long)
14
```

A non-integral example:

```
sage: E = EllipticCurve([-3/8,-2/3])
sage: E.gens() # random (up to sign)
[(10/9 : 29/54 : 1)]
```

A non-minimal example:

```
sage: E = EllipticCurve('389a1')
sage: E1 = E.change_weierstrass_model([1/20,0,0,0]); E1
Elliptic Curve defined by y^2 + 8000*y = x^3 + 400*x^2 - 320000*x
 over Rational Field
sage: E1.gens() # random (if database not used)
[(-400 : 8000 : 1), (0 : -8000 : 1)]
sage: E1.gens(algorithm="pari")   #random
[(-400 : 8000 : 1), (0 : -8000 : 1)]
```

gens_certain()

Return `True` if the generators have been proven correct.

EXAMPLES:

```
sage: E = EllipticCurve('37a1')
sage: E.gens()                  # random (up to sign)
[(0 : -1 : 1)]
sage: E.gens_certain()
True
```

global_integral_model()

Return a model of `self` which is integral at all primes.

EXAMPLES:

```
sage: E = EllipticCurve([0, 0, 1/216, -7/1296, 1/7776])
sage: F = E.global_integral_model(); F
Elliptic Curve defined by y^2 + y = x^3 - 7*x + 6 over Rational Field
sage: F == EllipticCurve('5077a1')
True
```

has_cm()

Return whether or not this curve has a CM j-invariant.

OUTPUT:

`True` if the j-invariant of this curve is the j-invariant of an imaginary quadratic order, otherwise `False`.

See also:

cm_discriminant() and *has_rational_cm()*

Note: Even if E has CM in this sense (that its j-invariant is a CM j-invariant), since the associated negative discriminant D is not a square in $\mathbf{Q}$, the extra endomorphisms will not be defined over $\mathbf{Q}$. See also the method *has_rational_cm()* which tests whether E has extra endomorphisms defined over $\mathbf{Q}$ or a given extension of $\mathbf{Q}$.

EXAMPLES:

```
sage: E = EllipticCurve('37a1')
sage: E.has_cm()
False
sage: E = EllipticCurve('32a1')
sage: E.has_cm()
True
sage: E.j_invariant()
1728
```

has_good_reduction_outside_S(*S=None*)

Test if this elliptic curve has good reduction outside `S`.

INPUT:

- `S` – list of primes (default: `[]`).

Note: Primality of elements of `S` is not checked, and the output is undefined if `S` is not a list or contains non-primes.

This only tests the given model, so should only be applied to minimal models.

EXAMPLES:

```
sage: EllipticCurve('11a1').has_good_reduction_outside_S([11])
True
sage: EllipticCurve('11a1').has_good_reduction_outside_S([2])
False
sage: EllipticCurve('2310a1').has_good_reduction_outside_S([2,3,5,7])
False
sage: EllipticCurve('2310a1').has_good_reduction_outside_S([2,3,5,7,11])
True
```

has_rational_cm(*field=None*)

Return whether or not this curve has CM defined over $\mathbf{Q}$ or the given field.

INPUT:

- `field` – (default: $\mathbf{Q}$) a field, which should be an extension of $\mathbf{Q}$;

OUTPUT:

`True` if the ring of endomorphisms of this curve over the given field is larger than $\mathbf{Z}$; otherwise `False`. If `field` is `None` the output will always be `False`. See also *cm_discriminant()* and *has_cm()*.

Note: If E has CM but the discriminant D is not a square in the given field K, which will certainly be the case for $K = \mathbf{Q}$ since $D < 0$, then the extra endomorphisms will not be defined over K, and this function

will return `False`. See also *has_cm()*. To obtain the CM discriminant, use *cm_discriminant()*.

EXAMPLES:

```
sage: E = EllipticCurve(j=0)
sage: E.has_cm()
True
sage: E.has_rational_cm()
False
sage: D = E.cm_discriminant(); D
-3
```

If we extend scalars to a field in which the discriminant is a square, the CM becomes rational:

```
sage: E.has_rational_cm(QuadraticField(-3))                                    #␣
↪needs sage.rings.number_field
True

sage: E = EllipticCurve(j=8000)
sage: E.has_cm()
True
sage: E.has_rational_cm()
False
sage: D = E.cm_discriminant(); D
-8
```

Again, we may extend scalars to a field in which the discriminant is a square, where the CM becomes rational:

```
sage: E.has_rational_cm(QuadraticField(-2))                                    #␣
↪needs sage.rings.number_field
True
```

The field need not be a number field provided that it is an extension of $\mathbf{Q}$:

```
sage: E.has_rational_cm(RR)
False
sage: E.has_rational_cm(CC)
True
```

An error is raised if a field is given which is not an extension of $\mathbf{Q}$, i.e., not of characteristic 0:

```
sage: E.has_rational_cm(GF(2))
Traceback (most recent call last):
...
ValueError: Error in has_rational_cm: Finite Field of size 2
is not an extension field of QQ
```

heegner_discriminants(*bound*)

Return the list of self's Heegner discriminants between -1 and -bound.

INPUT:

- `bound (int)` – upper bound for -discriminant

OUTPUT: The list of Heegner discriminants between -1 and -bound for the given elliptic curve.

EXAMPLES:

```
sage: E=EllipticCurve('11a')
sage: E.heegner_discriminants(30)                    # indirect doctest
[-7, -8, -19, -24]
```

heegner_discriminants_list(*n*)

Return the list of self's first n Heegner discriminants smaller than -5.

INPUT:

- `n (int)` – the number of discriminants to compute

OUTPUT: The list of the first n Heegner discriminants smaller than -5 for the given elliptic curve.

EXAMPLES:

```
sage: E=EllipticCurve('11a')
sage: E.heegner_discriminants_list(4)                    # indirect doctest
[-7, -8, -19, -24]
```

heegner_index(*D*, *min_p=2*, *prec=5*, *descent_second_limit=12*, *verbose_mwrank=False*, *check_rank=True*)

Return an interval that contains the index of the Heegner point y_K in the group of K-rational points modulo torsion on this elliptic curve, computed using the Gross-Zagier formula and/or a point search, or possibly half the index if the rank is greater than one.

If the curve has rank > 1, then the returned index is infinity.

Note: If `min_p` is bigger than 2 then the index can be off by any prime less than `min_p`. This function returns the index divided by 2 exactly when the rank of $E(K)$ is greater than 1 and $E(\mathbf{Q})_{/tor} \oplus E^D(\mathbf{Q})_{/tor}$ has index 2 in $E(K)_{/tor}$, where the second factor undergoes a twist.

INPUT:

- `D (int)` – Heegner discriminant
- `min_p (int)` – (default: 2) only rule out primes = min_p dividing the index.
- `verbose_mwrank (bool)` – (default: `False`); print lots of mwrank search status information when computing regulator
- `prec (int)` – (default: 5), use prec*sqrt(N) + 20 terms of L-series in computations, where N is the conductor.
- `descent_second_limit` – (default: 12)- used in 2-descent when computing regulator of the twist
- `check_rank` – whether to check if the rank is at least 2 by computing the Mordell-Weil rank directly.

OUTPUT: an interval that contains the index, or half the index

EXAMPLES:

```
sage: E = EllipticCurve('11a')
sage: E.heegner_discriminants(50)
[-7, -8, -19, -24, -35, -39, -40, -43]
sage: E.heegner_index(-7)
1.00000?
```

```
sage: E = EllipticCurve('37b')
sage: E.heegner_discriminants(100)
[-3, -4, -7, -11, -40, -47, -67, -71, -83, -84, -95]
sage: E.heegner_index(-95)           # long time (1 second)
2.00000?
```

This tests doing direct computation of the Mordell-Weil group.

```
sage: EllipticCurve('675b').heegner_index(-11)
3.0000?
```

Currently discriminants -3 and -4 are not supported:

```
sage: E.heegner_index(-3)
Traceback (most recent call last):
...
ArithmeticError: Discriminant (=-3) must not be -3 or -4.
```

The curve 681b returns the true index, which is 3:

```
sage: E = EllipticCurve('681b')
sage: I = E.heegner_index(-8); I
3.0000?
```

In fact, whenever the returned index has a denominator of 2, the true index is got by multiplying the returned index by 2. Unfortunately, this is not an if and only if condition, i.e., sometimes the index must be multiplied by 2 even though the denominator is not 2.

This example demonstrates the `descent_second_limit` option, which can be used to fine tune the 2-descent used to compute the regulator of the twist:

```
sage: E = EllipticCurve([1,-1,0,-1228,-16267])
sage: E.heegner_index(-8)
Traceback (most recent call last):
...
RuntimeError: ...
```

However when we search higher, we find the points we need:

```
sage: E.heegner_index(-8, descent_second_limit=16, check_rank=False)  # long␣
↪time
2.00000?
```

Two higher rank examples (of ranks 2 and 3):

```
sage: E = EllipticCurve('389a')
sage: E.heegner_index(-7)
+Infinity
sage: E = EllipticCurve('5077a')
sage: E.heegner_index(-7)
+Infinity
sage: E.heegner_index(-7, check_rank=False)
0.001?
sage: E.heegner_index(-7, check_rank=False).lower() == 0
True
```

heegner_index_bound(*D=0*, *prec=5*, *max_height=None*)

Assume `self` has rank 0.

Return a list v of primes such that if an odd prime p divides the index of the Heegner point in the group of rational points modulo torsion, then p is in v.

If 0 is in the interval of the height of the Heegner point computed to the given prec, then this function returns $v = 0$. This does not mean that the Heegner point is torsion, just that it is very likely torsion.

If we obtain no information from a search up to `max_height`, e.g., if the Siksek et al. bound is bigger than `max_height`, then we return $v = -1$.

INPUT:

- `D (int)` – (default: 0) Heegner discriminant; if 0, use the first discriminant -4 that satisfies the Heegner hypothesis
- `verbose (bool)` – (default: `True`)
- `prec (int)` – (default: 5), use $prec \cdot \sqrt(N) + 20$ terms of L-series in computations, where N is the conductor.
- `max_height (float)` – should be = 21; bound on logarithmic naive height used in point searches. Make smaller to make this function faster, at the expense of possibly obtaining a worse answer. A good range is between 13 and 21.

OUTPUT:

- `v` – list or int (bad primes or 0 or -1)
- `D` – the discriminant that was used (this is useful if D was automatically selected).
- `exact` – either False, or the exact Heegner index (up to factors of 2)

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: E.heegner_index_bound()
([2], -7, 2)
```

heegner_point (*D*, *c=1*, *f=None*, *check=True*)

Returns the Heegner point on this curve associated to the quadratic imaginary field $K = \mathbf{Q}(\sqrt{D})$.

If the optional parameter c is given, returns the higher Heegner point associated to the order of conductor c.

INPUT:

- D – a Heegner discriminant
- c – (default: 1) conductor, must be coprime to DN
- f – binary quadratic form or 3-tuple (A, B, C) of coefficients of $AX^2 + BXY + CY^2$
- `check` – bool (default: `True`)

OUTPUT: The Heegner point y_c.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.heegner_discriminants_list(10)
[-7, -11, -40, -47, -67, -71, -83, -84, -95, -104]
sage: P = E.heegner_point(-7); P                          # indirect doctest
Heegner point of discriminant -7 on elliptic curve of conductor 37
sage: z = P.point_exact(); z == E(0, 0, 1)  or -z == E(0, 0, 1)
True
sage: P.curve()
```

(continues on next page)

(continued from previous page)

```
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
sage: P = E.heegner_point(-40).point_exact(); P
(a : -a + 1 : 1)
sage: P = E.heegner_point(-47).point_exact(); P
(a : a^4 + a - 1 : 1)
sage: P[0].parent()
Number Field in a with defining polynomial x^5 - x^4 + x^3 + x^2 - 2*x + 1
```

Working out the details manually:

```
sage: P = E.heegner_point(-47).numerical_approx(prec=200)
sage: f = algdep(P[0], 5); f
x^5 - x^4 + x^3 + x^2 - 2*x + 1
sage: f.discriminant().factor()
47^2
```

The Heegner hypothesis is checked:

```
sage: E = EllipticCurve('389a'); P = E.heegner_point(-5,7);
Traceback (most recent call last):
...
ValueError: N (=389) and D (=-5) must satisfy the Heegner hypothesis
```

We can specify the quadratic form:

```
sage: P = EllipticCurve('389a').heegner_point(-7, 5, (778,925,275)); P
Heegner point of discriminant -7 and conductor 5
 on elliptic curve of conductor 389
sage: P.quadratic_form()
778*x^2 + 925*x*y + 275*y^2
```

heegner_point_height (*D*, *prec=2*, *check_rank=True*)

Use the Gross-Zagier formula to compute the Neron-Tate canonical height over K of the Heegner point corresponding to D, as an interval (it is computed to some precision using L-functions).

If the curve has rank at least 2, then the returned height is the exact Sage integer 0.

INPUT:

- `D (int)` – fundamental discriminant (=/= -3, -4)
- `prec (int)` – (default: 2), use $prec \cdot \sqrt{(N)} + 20$ terms of L-series in computations, where N is the conductor.
- `check_rank` – whether to check if the rank is at least 2 by computing the Mordell-Weil rank directly.

OUTPUT: Interval that contains the height of the Heegner point.

EXAMPLES:

```
sage: E = EllipticCurve('11a')
sage: E.heegner_point_height(-7)
0.22227?
```

Some higher rank examples:

```
sage: E = EllipticCurve('389a')
sage: E.heegner_point_height(-7)
```

(continues on next page)

(continued from previous page)

```
0
sage: E = EllipticCurve('5077a')
sage: E.heegner_point_height(-7)
0
sage: E.heegner_point_height(-7, check_rank=False)
0.0000?
```

heegner_sha_an (*D*, *prec=53*)

Return the conjectural (analytic) order of Sha for E over the field $K = \mathbf{Q}(\sqrt{D})$.

INPUT:

- D – negative integer; the Heegner discriminant
- prec – integer (default: 53); bits of precision to compute analytic order of Sha

OUTPUT:

(floating point number) an approximation to the conjectural order of Sha.

Note: Often you'll want to do `proof.elliptic_curve(False)` when using this function, since often the twisted elliptic curves that come up have enormous conductor, and Sha is nontrivial, which makes provably finding the Mordell-Weil group using 2-descent difficult.

EXAMPLES:

An example where E has conductor 11:

```
sage: E = EllipticCurve('11a')
sage: E.heegner_sha_an(-7)                                  # long time
1.00000000000000
```

The cache works:

```
sage: E.heegner_sha_an(-7) is E.heegner_sha_an(-7)          # long time
True
```

Lower precision:

```
sage: E.heegner_sha_an(-7,10)                               # long time
1.0
```

Checking that the cache works for any precision:

```
sage: E.heegner_sha_an(-7,10) is E.heegner_sha_an(-7,10)    # long time
True
```

Next we consider a rank 1 curve with nontrivial Sha over the quadratic imaginary field K; however, there is no Sha for E over $\mathbf{Q}$ or for the quadratic twist of E:

```
sage: E = EllipticCurve('37a')
sage: E.heegner_sha_an(-40)                                 # long time
4.00000000000000
sage: E.quadratic_twist(-40).sha().an()                     # long time
1
sage: E.sha().an()                                          # long time
1
```

A rank 2 curve:

```
sage: E = EllipticCurve('389a')                                    # long time
sage: E.heegner_sha_an(-7)                                         # long time
1.00000000000000
```

If we remove the hypothesis that $E(K)$ has rank 1 in Conjecture 2.3 in [GZ1986] page 311, then that conjecture is false, as the following example shows:

```
sage: # long time
sage: E = EllipticCurve('65a')
sage: E.heegner_sha_an(-56)
1.00000000000000
sage: E.torsion_order()
2
sage: E.tamagawa_product()
1
sage: E.quadratic_twist(-56).rank()
2
```

height (*precision=None*)

Return the real height of this elliptic curve.

This is used in *integral_points()*.

INPUT:

- `precision` – desired real precision of the result (default real precision if `None`)

EXAMPLES:

```
sage: E = EllipticCurve('5077a1')
sage: E.height()
17.4513334798896
sage: E.height(100)
17.451333479889612702508579399
sage: E = EllipticCurve([0,0,0,0,1])
sage: E.height()
1.38629436111989
sage: E = EllipticCurve([0,0,0,1,0])
sage: E.height()
7.45471994936400
```

integral_model()

Return a model of `self` which is integral at all primes.

EXAMPLES:

```
sage: E = EllipticCurve([0, 0, 1/216, -7/1296, 1/7776])
sage: F = E.global_integral_model(); F
Elliptic Curve defined by y^2 + y = x^3 - 7*x + 6 over Rational Field
sage: F == EllipticCurve('5077a1')
True
```

integral_points (*mw_base='auto'*, *both_signs=False*, *verbose=False*)

Compute all integral points (up to sign) on this elliptic curve.

INPUT:

- `mw_base` – (default: `'auto'` - calls *gens()*) list of EllipticCurvePoint generating the Mordell-Weil group of E
- `both_signs` – boolean (default: `False`); if `True` the output contains both P and $-P$, otherwise only one of each pair
- `verbose` – boolean (default: `False`); if `True`, some details of the computation are output

OUTPUT: A sorted list of all the integral points on E (up to sign unless `both_signs` is `True`)

Note: The complexity increases exponentially in the rank of curve E. The computation time (but not the output!) depends on the Mordell-Weil basis. If `mw_base` is given but is not a basis for the Mordell-Weil group (modulo torsion), integral points which are not in the subgroup generated by the given points will almost certainly not be listed.

EXAMPLES: A curve of rank 3 with no torsion points:

```
sage: E = EllipticCurve([0,0,1,-7,6])
sage: P1 = E.point((2,0)); P2 = E.point((-1,3)); P3 = E.point((4,6))
sage: a = E.integral_points([P1,P2,P3]); a
[(-3 : -1 : 1), (-2 : -4 : 1), (-1 : -4 : 1), (0 : -3 : 1),
 (1 : -1 : 1), (2 : -1 : 1), (3 : -4 : 1), (4 : -7 : 1),
 (8 : -22 : 1), (11 : -36 : 1), (14 : -52 : 1), (21 : -96 : 1),
 (37 : -225 : 1), (52 : -375 : 1), (93 : -897 : 1),
 (342 : -6325 : 1), (406 : -8181 : 1), (816 : -23310 : 1)]
```

```
sage: a = E.integral_points([P1,P2,P3], verbose=True)
Using mw_basis  [(2 : 0 : 1), (3 : -4 : 1), (8 : -22 : 1)]
e1,e2,e3:  -3.0124303725933... 1.0658205476962... 1.94660982489710
Minimal and maximal eigenvalues of height pairing matrix: 0.637920814585005,2.
↪31982967525725
x-coords of points on compact component with  -3 <=x<= 1
[-3, -2, -1, 0, 1]
x-coords of points on non-compact component with  2 <=x<= 6
[2, 3, 4]
starting search of remaining points using coefficient bound 5 and |x| bound 1.
↪53897183921009e25
x-coords of extra integral points:
[2, 3, 4, 8, 11, 14, 21, 37, 52, 93, 342, 406, 816]
Total number of integral points: 18
```

It is not necessary to specify `mw_base`; if it is not provided, then the Mordell-Weil basis must be computed, which may take much longer.

```
sage: E = EllipticCurve([0,0,1,-7,6])
sage: a = E.integral_points(both_signs=True); a
[(-3 : -1 : 1), (-3 : 0 : 1), (-2 : -4 : 1), (-2 : 3 : 1), (-1 : -4 : 1),
 (-1 : 3 : 1), (0 : -3 : 1), (0 : 2 : 1), (1 : -1 : 1), (1 : 0 : 1),
 (2 : -1 : 1), (2 : 0 : 1), (3 : -4 : 1), (3 : 3 : 1), (4 : -7 : 1),
 (4 : 6 : 1), (8 : -22 : 1), (8 : 21 : 1), (11 : -36 : 1), (11 : 35 : 1),
 (14 : -52 : 1), (14 : 51 : 1), (21 : -96 : 1), (21 : 95 : 1),
 (37 : -225 : 1), (37 : 224 : 1), (52 : -375 : 1), (52 : 374 : 1),
 (93 : -897 : 1), (93 : 896 : 1), (342 : -6325 : 1), (342 : 6324 : 1),
 (406 : -8181 : 1), (406 : 8180 : 1), (816 : -23310 : 1), (816 : 23309 : 1)]
```

An example with negative discriminant:

```
sage: EllipticCurve('900d1').integral_points()
[(-11 : -27 : 1), (-4 : -34 : 1), (4 : -18 : 1), (16 : -54 : 1)]
```

Another example with rank 5 and no torsion points:

```
sage: E = EllipticCurve([-879984,319138704])
sage: P1 = E.point((540,1188)); P2 = E.point((576,1836))
sage: P3 = E.point((468,3132)); P4 = E.point((612,3132))
sage: P5 = E.point((432,4428))
sage: a = E.integral_points([P1,P2,P3,P4,P5]); len(a)  # long time (18s on␣
↪sage.math, 2011)
54
```

ALGORITHM:

This function uses the algorithm given in [Coh2007I].

AUTHORS:

- Michael Mardaus (2008-07)
- Tobias Nagel (2008-07)
- John Cremona (2008-07)

integral_short_weierstrass_model()

Return a model of the form $y^2 = x^3 + ax + b$ for this curve with $a, b \in \mathbf{Z}$.

EXAMPLES:

```
sage: E = EllipticCurve('17a1')
sage: E.integral_short_weierstrass_model()
Elliptic Curve defined by y^2  = x^3 - 11*x - 890 over Rational Field
```

integral_x_coords_in_interval(*xmin*, *xmax*)

Return the set of integers x with $xmin \leq x \leq xmax$ which are x-coordinates of rational points on this curve.

INPUT:

- `xmin`, `xmax` (integers) – two integers

OUTPUT:

(set) The set of integers x with $xmin \leq x \leq xmax$ which are x-coordinates of rational points on the elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([0, 0, 1, -7, 6])
sage: xset = E.integral_x_coords_in_interval(-100,100)
sage: sorted(xset)
[-3, -2, -1, 0, 1, 2, 3, 4, 8, 11, 14, 21, 37, 52, 93]
sage: xset = E.integral_x_coords_in_interval(-100, 0)
sage: sorted(xset)
[-3, -2, -1, 0]
```

is_global_integral_model()

Return `True` iff `self` is integral at all primes.

EXAMPLES:

```
sage: E = EllipticCurve([1/2, 1/5, 1/5, 1/5, 1/5])
sage: E.is_global_integral_model()
False
sage: Emin=E.global_integral_model()
sage: Emin.is_global_integral_model()
True
```

is_good(*p*, *check=True*)

Return `True` if `p` is a prime of good reduction for E.

INPUT:

- `p` – a prime

OUTPUT: bool

EXAMPLES:

```
sage: e = EllipticCurve('11a')
sage: e.is_good(-8)
Traceback (most recent call last):
...
ValueError: p must be prime
sage: e.is_good(-8, check=False)
True
```

is_integral()

Return `True` if this elliptic curve has integral coefficients (in Z).

EXAMPLES:

```
sage: E = EllipticCurve(QQ,[1,1]); E
Elliptic Curve defined by y^2 = x^3 + x + 1 over Rational Field
sage: E.is_integral()
True
sage: E2=E.change_weierstrass_model(2,0,0,0); E2
Elliptic Curve defined by y^2 = x^3 + 1/16*x + 1/64 over Rational Field
sage: E2.is_integral()
False
```

is_isogenous(*other*, *proof=True*, *maxp=200*)

Return whether or not self is isogenous to other.

INPUT:

- `other` – another elliptic curve
- `proof` – (default: `True`) if `False`, the function will return `True` whenever the two curves have the same conductor and are isogenous modulo p for p up to `maxp`; otherwise this test is followed by a rigorous test (which may be more time-consuming)
- `maxp` – (default: 200) the maximum prime p for which isogeny modulo p will be checked

OUTPUT:

(bool) True if there is an isogeny from curve `self` to curve `other`.

ALGORITHM:

First the conductors are compared as well as the Traces of Frobenius for good primes up to `maxp`. If any of these tests fail, `False` is returned. If they all pass and `proof` is `False` then `True` is returned, otherwise

a complete set of curves isogenous to `self` is computed and `other` is checked for isomorphism with any of these,

EXAMPLES:

```
sage: E1 = EllipticCurve('14a1')
sage: E6 = EllipticCurve('14a6')
sage: E1.is_isogenous(E6)
True
sage: E1.is_isogenous(EllipticCurve('11a1'))
False
```

```
sage: EllipticCurve('37a1').is_isogenous(EllipticCurve('37b1'))
False
```

```
sage: E = EllipticCurve([2, 16])
sage: EE = EllipticCurve([87, 45])
sage: E.is_isogenous(EE)
False
```

is_local_integral_model(**p*)

Tests if `self` is integral at the prime `p`, or at all the primes if `p` is a list or tuple of primes.

EXAMPLES:

```
sage: E = EllipticCurve([1/2, 1/5, 1/5, 1/5, 1/5])
sage: [E.is_local_integral_model(p) for p in (2,3,5)]
[False, True, False]
sage: E.is_local_integral_model(2,3,5)
False
sage: Eint2=E.local_integral_model(2)
sage: Eint2.is_local_integral_model(2)
True
```

is_minimal()

Return `True` iff this elliptic curve is a reduced minimal model.

The unique minimal Weierstrass equation for this elliptic curve. This is the model with minimal discriminant and $a_1, a_2, a_3 \in \{0, \pm 1\}$.

Todo: This is not very efficient since it just computes the minimal model and compares. A better implementation using the Kraus conditions would be preferable.

EXAMPLES:

```
sage: E = EllipticCurve([10,100,1000,10000,1000000])
sage: E.is_minimal()
False
sage: E = E.minimal_model()
sage: E.is_minimal()
True
```

is_ordinary(*p*, *ell=None*)

Return `True` precisely when the mod-`p` representation attached to this elliptic curve is ordinary at `ell`.

INPUT:

- p – a prime
- ell – a prime (default: p)

OUTPUT: bool

EXAMPLES:

```
sage: E = EllipticCurve('37a1')
sage: E.is_ordinary(37)
True
sage: E = EllipticCurve('32a1')
sage: E.is_ordinary(2)
False
sage: [p for p in prime_range(50) if E.is_ordinary(p)]
[5, 13, 17, 29, 37, 41]
```

is_p_integral(*p*)

Return True if this elliptic curve has p-integral coefficients.

INPUT:

- p – a prime integer

EXAMPLES:

```
sage: E = EllipticCurve(QQ,[1,1]); E
Elliptic Curve defined by y^2 = x^3 + x + 1 over Rational Field
sage: E.is_p_integral(2)
True
sage: E2=E.change_weierstrass_model(2,0,0,0); E2
Elliptic Curve defined by y^2 = x^3 + 1/16*x + 1/64 over Rational Field
sage: E2.is_p_integral(2)
False
sage: E2.is_p_integral(3)
True
```

is_p_minimal(*p*)

Tests if curve is p-minimal at a given prime p.

INPUT:

- p – a prime

OUTPUT:

- True – if curve is p-minimal
- False – if curve is not p-minimal

EXAMPLES:

```
sage: E = EllipticCurve('441a2')
sage: E.is_p_minimal(7)
True
```

```
sage: E = EllipticCurve([0,0,0,0,(2*5*11)**10])
sage: [E.is_p_minimal(p) for p in prime_range(2,24)]
[False, True, False, True, False, True, True, True, True]
```

is_semistable()

Return `True` iff this elliptic curve is semi-stable at all primes.

EXAMPLES:

```
sage: E = EllipticCurve('37a1')
sage: E.is_semistable()
True
sage: E = EllipticCurve('90a1')
sage: E.is_semistable()
False
```

is_supersingular(*p*, *ell=None*)

Return `True` precisely when p is a prime of good reduction and the mod-`p` representation attached to this elliptic curve is supersingular at ell.

INPUT:

- `p` – a prime
- `ell` – a prime (default: `p`)

OUTPUT: bool

EXAMPLES:

```
sage: E = EllipticCurve('37a1')
sage: E.is_supersingular(37)
False
sage: E = EllipticCurve('32a1')
sage: E.is_supersingular(2)
False
sage: E.is_supersingular(7)
True
sage: [p for p in prime_range(50) if E.is_supersingular(p)]
[3, 7, 11, 19, 23, 31, 43, 47]
```

isogenies_prime_degree(*l=None*)

Return a list of ℓ-isogenies from self, where ℓ is a prime.

INPUT:

- `l` – either `None` or a prime or a list of primes

OUTPUT:

(list) ℓ-isogenies for the given ℓ or if ℓ is None, all ℓ-isogenies.

Note: The codomains of the isogenies returned are standard minimal models. This is because the functions `isogenies_prime_degree_genus_0()` and `isogenies_sporadic_Q()` are implemented that way for curves defined over **Q**.

EXAMPLES:

```
sage: E = EllipticCurve([45,32])
sage: E.isogenies_prime_degree()
[]
sage: E = EllipticCurve(j = -262537412640768000)
sage: E.isogenies_prime_degree()
```

(continues on next page)

(continued from previous page)

```
[Isogeny of degree 163
  from Elliptic Curve defined by y^2 + y = x^3 - 2174420*x + 1234136692 over␣
↪Rational Field
    to Elliptic Curve defined by y^2 + y = x^3 - 57772164980*x -␣
↪5344733777551611 over Rational Field]
sage: E1 = E.quadratic_twist(6584935282)
sage: E1.isogenies_prime_degree()
[Isogeny of degree 163
  from Elliptic Curve defined by y^2 = x^3 - 94285835957031797981376080*x +␣
↪3523853116124200413873380542245478308 98 over Rational Field
    to Elliptic Curve defined by y^2 = x^3 -␣
↪25050803755423778405671810695 20*x -␣
↪15260916311095532569780901163187978450180208 06 over Rational Field]

sage: E = EllipticCurve('14a1')
sage: E.isogenies_prime_degree(2)
[Isogeny of degree 2
  from Elliptic Curve defined by y^2 + x*y + y = x^3 + 4*x - 6 over Rational␣
↪Field
    to Elliptic Curve defined by y^2 + x*y + y = x^3 - 36*x - 70 over␣
↪Rational Field]
sage: E.isogenies_prime_degree(3)
[Isogeny of degree 3
  from Elliptic Curve defined by y^2 + x*y + y = x^3 + 4*x - 6 over Rational␣
↪Field
    to Elliptic Curve defined by y^2 + x*y + y = x^3 - x over Rational Field,
 Isogeny of degree 3
  from Elliptic Curve defined by y^2 + x*y + y = x^3 + 4*x - 6 over Rational␣
↪Field
    to Elliptic Curve defined by y^2 + x*y + y = x^3 - 171*x - 874 over␣
↪Rational Field]
sage: E.isogenies_prime_degree(5)
[]
sage: E.isogenies_prime_degree(11)
[]
sage: E.isogenies_prime_degree(29)
[]
sage: E.isogenies_prime_degree(4)
Traceback (most recent call last):
...
ValueError: 4 is not prime.
```

isogeny_class(*algorithm='sage'*, *order=None*)

Return the **Q**-isogeny class of this elliptic curve.

INPUT:

- `algorithm` – string: one of the following:
  - "database" – use the Cremona database (only works if curve is isomorphic to a curve in the database)
  - "sage" (default) – use the native Sage implementation.
- `order` – `None`, string, or list of curves (default: `None`); If not `None` then the curves in the class are reordered after being computed. Note that if the order is `None` then the resulting order will depend on the algorithm.
  - If `order` is "database" or "sage", then the reordering is so that the order of curves matches the order produced by that algorithm.

- If `order` is "lmfdb" then the curves are sorted lexicographically by a-invariants, in the LMFDB database.
- If `order` is a list of curves, then the curves in the class are reordered to be isomorphic with the specified list of curves.

OUTPUT:

An instance of the class *sage.schemes.elliptic_curves.isogeny_class.IsogenyClass_EC_Rational*. This object models a list of minimal models (with containment, index, etc based on isomorphism classes). It also has methods for computing the isogeny matrix and the list of isogenies between curves in this class.

Note: The curves in the isogeny class will all be standard minimal models.

EXAMPLES:

```
sage: isocls = EllipticCurve('37b').isogeny_class(order="lmfdb")
sage: isocls
Elliptic curve isogeny class 37b
sage: isocls.curves
(Elliptic Curve defined by y^2 + y = x^3 + x^2 - 1873*x - 31833 over Rational␣
↪Field,
 Elliptic Curve defined by y^2 + y = x^3 + x^2 - 23*x - 50 over Rational␣
↪Field,
 Elliptic Curve defined by y^2 + y = x^3 + x^2 - 3*x + 1 over Rational Field)
sage: isocls.matrix()
[1 3 9]
[3 1 3]
[9 3 1]
```

```
sage: isocls = EllipticCurve('37b').isogeny_class('database', order="lmfdb");␣
↪isocls.curves
(Elliptic Curve defined by y^2 + y = x^3 + x^2 - 1873*x - 31833 over Rational␣
↪Field,
 Elliptic Curve defined by y^2 + y = x^3 + x^2 - 23*x - 50 over Rational␣
↪Field,
 Elliptic Curve defined by y^2 + y = x^3 + x^2 - 3*x + 1 over Rational Field)
```

This is an example of a curve with a 37-isogeny:

```
sage: E = EllipticCurve([1,1,1,-8,6])
sage: isocls = E.isogeny_class(); isocls
Isogeny class of Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 8*x +␣
↪6 over Rational Field
sage: isocls.matrix()
[ 1 37]
[37  1]
sage: print("\n".join(repr(E) for E in isocls.curves))
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 8*x + 6 over Rational␣
↪Field
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 208083*x - 36621194␣
↪over Rational Field
```

This curve had numerous 2-isogenies:

```
sage: e = EllipticCurve([1,0,0,-39,90])
sage: isocls = e.isogeny_class(); isocls.matrix()
[1 2 4 4 8 8]
[2 1 2 2 4 4]
[4 2 1 4 8 8]
[4 2 4 1 2 2]
[8 4 8 2 1 4]
[8 4 8 2 4 1]
```

See http://math.harvard.edu/~elkies/nature.html for more interesting examples of isogeny structures.

```
sage: E = EllipticCurve(j = -262537412640768000)
sage: isocls = E.isogeny_class(); isocls.matrix()
[  1 163]
[163   1]
sage: print("\n".join(repr(C) for C in isocls.curves))
Elliptic Curve defined by y^2 + y = x^3 - 2174420*x + 1234136692 over␣
↪Rational Field
Elliptic Curve defined by y^2 + y = x^3 - 57772164980*x - 5344733777551611␣
↪over Rational Field
```

The degrees of isogenies are invariant under twists:

```
sage: E = EllipticCurve(j = -262537412640768000)
sage: E1 = E.quadratic_twist(6584935282)
sage: isocls = E1.isogeny_class(); isocls.matrix()
[  1 163]
[163   1]
sage: E1.conductor()
184330929667120636533330496
```

```
sage: E = EllipticCurve('14a1')
sage: isocls = E.isogeny_class(); isocls.matrix()
[ 1  2  3  3  6  6]
[ 2  1  6  6  3  3]
[ 3  6  1  9  2 18]
[ 3  6  9  1 18  2]
[ 6  3  2 18  1  9]
[ 6  3 18  2  9  1]
sage: print("\n".join(repr(C) for C in isocls.curves))
Elliptic Curve defined by y^2 + x*y + y = x^3 + 4*x - 6 over Rational Field
Elliptic Curve defined by y^2 + x*y + y = x^3 - 36*x - 70 over Rational Field
Elliptic Curve defined by y^2 + x*y + y = x^3 - x over Rational Field
Elliptic Curve defined by y^2 + x*y + y = x^3 - 171*x - 874 over Rational␣
↪Field
Elliptic Curve defined by y^2 + x*y + y = x^3 - 11*x + 12 over Rational Field
Elliptic Curve defined by y^2 + x*y + y = x^3 - 2731*x - 55146 over Rational␣
↪Field
sage: isocls2 = isocls.reorder('lmfdb'); isocls2.matrix()
[ 1  2  3  9 18  6]
[ 2  1  6 18  9  3]
[ 3  6  1  3  6  2]
[ 9 18  3  1  2  6]
[18  9  6  2  1  3]
[ 6  3  2  6  3  1]
sage: print("\n".join(repr(C) for C in isocls2.curves))
Elliptic Curve defined by y^2 + x*y + y = x^3 - 2731*x - 55146 over Rational␣
```

(continues on next page)

(continued from previous page)

```
↪Field
Elliptic Curve defined by y^2 + x*y + y = x^3 - 171*x - 874 over Rational␣
↪Field
Elliptic Curve defined by y^2 + x*y + y = x^3 - 36*x - 70 over Rational Field
Elliptic Curve defined by y^2 + x*y + y = x^3 - 11*x + 12 over Rational Field
Elliptic Curve defined by y^2 + x*y + y = x^3 - x over Rational Field
Elliptic Curve defined by y^2 + x*y + y = x^3 + 4*x - 6 over Rational Field
```

```
sage: E = EllipticCurve('11a1')
sage: isocls = E.isogeny_class(); isocls.matrix()
[ 1  5  5]
[ 5  1 25]
[ 5 25  1]
sage: f = isocls.isogenies()[0][1]; f.kernel_polynomial()
x^2 + x - 29/5
```

isogeny_degree(*other*)

Return the minimal degree of an isogeny between `self` and `other`.

INPUT:

- `other` – another elliptic curve

OUTPUT:

The minimal degree of an isogeny from `self` to `other`, or 0 if the curves are not isogenous.

EXAMPLES:

```
sage: E = EllipticCurve([-1056, 13552])
sage: E2 = EllipticCurve([-127776, -18037712])
sage: E.isogeny_degree(E2)
11
```

```
sage: E1 = EllipticCurve('14a1')
sage: E2 = EllipticCurve('14a2')
sage: E3 = EllipticCurve('14a3')
sage: E4 = EllipticCurve('14a4')
sage: E5 = EllipticCurve('14a5')
sage: E6 = EllipticCurve('14a6')
sage: E3.isogeny_degree(E1)
3
sage: E3.isogeny_degree(E2)
6
sage: E3.isogeny_degree(E3)
1
sage: E3.isogeny_degree(E4)
9
sage: E3.isogeny_degree(E5)
2
sage: E3.isogeny_degree(E6)
18
```

```
sage: E1 = EllipticCurve('30a1')
sage: E2 = EllipticCurve('30a2')
sage: E3 = EllipticCurve('30a3')
sage: E4 = EllipticCurve('30a4')
```

(continues on next page)

(continued from previous page)

```
sage: E5 = EllipticCurve('30a5')
sage: E6 = EllipticCurve('30a6')
sage: E7 = EllipticCurve('30a7')
sage: E8 = EllipticCurve('30a8')
sage: E1.isogeny_degree(E1)
1
sage: E1.isogeny_degree(E2)
2
sage: E1.isogeny_degree(E3)
3
sage: E1.isogeny_degree(E4)
4
sage: E1.isogeny_degree(E5)
4
sage: E1.isogeny_degree(E6)
6
sage: E1.isogeny_degree(E7)
12
sage: E1.isogeny_degree(E8)
12
```

```
sage: E1 = EllipticCurve('15a1')
sage: E2 = EllipticCurve('15a2')
sage: E3 = EllipticCurve('15a3')
sage: E4 = EllipticCurve('15a4')
sage: E5 = EllipticCurve('15a5')
sage: E6 = EllipticCurve('15a6')
sage: E7 = EllipticCurve('15a7')
sage: E8 = EllipticCurve('15a8')
sage: E1.isogeny_degree(E1)
1
sage: E7.isogeny_degree(E2)
8
sage: E7.isogeny_degree(E3)
2
sage: E7.isogeny_degree(E4)
8
sage: E7.isogeny_degree(E5)
16
sage: E7.isogeny_degree(E6)
16
sage: E7.isogeny_degree(E8)
4
```

0 is returned when the curves are not isogenous:

```
sage: A = EllipticCurve('37a1')
sage: B = EllipticCurve('37b1')
sage: A.isogeny_degree(B)
0
sage: A.is_isogenous(B)
False
```

isogeny_graph(*order=None*)

Return a graph representing the isogeny class of this elliptic curve, where the vertices are isogenous curves over $\mathbf{Q}$ and the edges are prime degree isogenies.

Note: The vertices are labeled 1 to n rather than 0 to $n-1$ to correspond to LMFDB and Cremona labels.

EXAMPLES:

```
sage: LL = []
sage: for e in cremona_optimal_curves(range(1, 38)):  # long time
....:  G = e.isogeny_graph()
....:  already = False
....:  for H in LL:
....:      if G.is_isomorphic(H):
....:          already = True
....:          break
....:  if not already:
....:      LL.append(G)
sage: graphs_list.show_graphs(LL)  # long time
```

```
sage: E = EllipticCurve('195a')
sage: G = E.isogeny_graph()
sage: for v in G: print("{} {}".format(v, G.get_vertex(v)))
1 Elliptic Curve defined by y^2 + x*y  = x^3 - 110*x + 435 over Rational Field
2 Elliptic Curve defined by y^2 + x*y  = x^3 - 115*x + 392 over Rational Field
3 Elliptic Curve defined by y^2 + x*y  = x^3 + 210*x + 2277 over Rational␣
↪Field
4 Elliptic Curve defined by y^2 + x*y  = x^3 - 520*x - 4225 over Rational␣
↪Field
5 Elliptic Curve defined by y^2 + x*y  = x^3 + 605*x - 19750 over Rational␣
↪Field
6 Elliptic Curve defined by y^2 + x*y  = x^3 - 8125*x - 282568 over Rational␣
↪Field
7 Elliptic Curve defined by y^2 + x*y  = x^3 - 7930*x - 296725 over Rational␣
↪Field
8 Elliptic Curve defined by y^2 + x*y  = x^3 - 130000*x - 18051943 over␣
↪Rational Field
sage: G.plot(edge_labels=True)                                                #␣
↪needs sage.plot
Graphics object consisting of 23 graphics primitives
```

kodaira_symbol(p)

Local Kodaira type of the elliptic curve at p.

INPUT:

- p – an integral prime

OUTPUT:

- the Kodaira type of this elliptic curve at p, as a *KodairaSymbol*

EXAMPLES:

```
sage: E = EllipticCurve('124a')
sage: E.kodaira_type(2)
IV
```

kodaira_type(p)

Local Kodaira type of the elliptic curve at p.

INPUT:

- p – an integral prime

OUTPUT:

- the Kodaira type of this elliptic curve at p, as a *KodairaSymbol*

EXAMPLES:

```
sage: E = EllipticCurve('124a')
sage: E.kodaira_type(2)
IV
```

kodaira_type_old(*p*)

Local Kodaira type of the elliptic curve at p.

INPUT:

- p – an integral prime

OUTPUT:

- the Kodaira type of this elliptic curve at p, as a *KodairaSymbol*

EXAMPLES:

```
sage: E = EllipticCurve('124a')
sage: E.kodaira_type_old(2)
IV
```

kolyvagin_point(*D*, *c=1*, *check=True*)

Return the Kolyvagin point on this curve associated to the quadratic imaginary field $K = \mathbf{Q}(\sqrt{D})$ and conductor c.

INPUT:

- D – a Heegner discriminant
- c – (default: 1) conductor, must be coprime to DN
- check – bool (default: True)

OUTPUT: The Kolyvagin point P of conductor c.

EXAMPLES:

```
sage: E = EllipticCurve('37a1')
sage: P = E.kolyvagin_point(-67); P
Kolyvagin point of discriminant -67 on elliptic curve of conductor 37
sage: P.numerical_approx()  # abs tol 1e-14
(6.00000000000000 : -15.0000000000000 : 1.00000000000000)
sage: P.index()
6
sage: g = E((0,-1,1)) # a generator
sage: E.regulator() == E.regulator_of_points([g])
True
sage: 6*g
(6 : -15 : 1)
```

label(*space=False*)

Return the Cremona label associated to (the minimal model) of this curve, if it is known. If not, raise a LookupError exception.

EXAMPLES:

```
sage: E = EllipticCurve('389a1')
sage: E.cremona_label()
'389a1'
```

The default database only contains conductors up to 10000, so any curve with conductor greater than that will cause an error to be raised. The optional package `database_cremona_ellcurve` contains many more curves.

```
sage: E = EllipticCurve([1, -1, 0, -79, 289])
sage: E.conductor()
234446
sage: E.cremona_label()  # optional - database_cremona_ellcurve
'234446a1'
sage: E = EllipticCurve((0, 0, 1, -79, 342))
sage: E.conductor()
19047851
sage: E.cremona_label()
Traceback (most recent call last):
...
LookupError: Cremona database does not contain entry for
Elliptic Curve defined by y^2 + y = x^3 - 79*x + 342 over Rational Field
```

lmfdb_page()

Open the LMFDB web page of the elliptic curve in a browser.

See http://www.lmfdb.org

EXAMPLES:

```
sage: E = EllipticCurve('5077a1')
sage: E.lmfdb_page()  # optional -- webbrowser
```

local_integral_model(*p*)

Return a model of self which is integral at the prime p.

EXAMPLES:

```
sage: E = EllipticCurve([0, 0, 1/216, -7/1296, 1/7776])
sage: E.local_integral_model(2)
Elliptic Curve defined by y^2 + 1/27*y = x^3 - 7/81*x + 2/243 over Rational␣
↪Field
sage: E.local_integral_model(3)
Elliptic Curve defined by y^2 + 1/8*y = x^3 - 7/16*x + 3/32 over Rational␣
↪Field
sage: E.local_integral_model(2).local_integral_model(3) == EllipticCurve(
↪'5077a1')
True
```

lseries()

Return the L-series of this elliptic curve.

Further documentation is available for the functions which apply to the L-series.

EXAMPLES:

```
sage: E = EllipticCurve('37a1')
sage: E.lseries()
```

(continues on next page)

(continued from previous page)

```
Complex L-series of the Elliptic Curve defined by y^2 + y = x^3 - x over␣
↪Rational Field
```

lseries_gross_zagier(*A*)

Return the Gross-Zagier L-series attached to `self` and an ideal class A.

INPUT:

- `A` – an ideal class in an imaginary quadratic number field K

This L-series $L(E, A, s)$ is defined as the product of a shifted L-function of the quadratic character associated to K and the Dirichlet series whose n-th coefficient is the product of the n-th factor of the L-series of E and the number of integral ideal in A of norm n. For any character χ on the class group of K, one gets $L_K(E, \chi, s) = \sum_A \chi(A)L(E, A, s)$ where A runs through the class group of K.

For the exact definition see section IV of [GZ1986].

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: K.<a> = QuadraticField(-40)
sage: A = K.class_group().gen(0); A
Fractional ideal class (2, 1/2*a)
sage: L = E.lseries_gross_zagier(A)  ; L
Gross Zagier L-series attached to
 Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
 with ideal class Fractional ideal class (2, 1/2*a)
sage: L(1)
0.000000000000000
sage: L.taylor_series(1, 5)
0.000000000000000 - 5.51899839494458*z + 13.6297841350649*z^2 - 16.
↪2292417817675*z^3 + 7.94788823722712*z^4 + O(z^5)
```

These should be equal:

```
sage: L(2) + E.lseries_gross_zagier(A^2)(2)
0.502803417587467
sage: E.lseries()(2) * E.quadratic_twist(-40).lseries()(2)
0.502803417587467
```

manin_constant()

Return the Manin constant of this elliptic curve.

If $\phi : X_0(N) \to E$ is the modular parametrization of minimal degree, then the Manin constant c is defined to be the rational number c such that $\phi^*(\omega_E) = c \cdot \omega_f$ where ω_E is a Néron differential and $\omega_f = f(q)dq/q$ is the differential on $X_0(N)$ corresponding to the newform f attached to the isogeny class of E.

It is known that the Manin constant is an integer. It is conjectured that in each class there is at least one, more precisely the so-called strong Weil curve or $X_0(N)$-optimal curve, that has Manin constant 1.

OUTPUT: An integer.

This function only works if the curve is in the installed Cremona database. Sage includes by default a small database; for the full database you have to install an optional package.

EXAMPLES:

```
sage: EllipticCurve('11a1').manin_constant()
1
sage: EllipticCurve('11a2').manin_constant()
1
sage: EllipticCurve('11a3').manin_constant()
5
```

Check that it works even if the curve is non-minimal:

```
sage: EllipticCurve('11a3').change_weierstrass_model([1/35,0,0,0]).manin_
↪constant()
5
```

Rather complicated examples (see Issue #12080)

```
sage: [ EllipticCurve('27a%s'%i).manin_constant() for i in [1,2,3,4]]
[1, 1, 3, 3]
sage: [ EllipticCurve('80b%s'%i).manin_constant() for i in [1,2,3,4]]
[1, 2, 1, 2]
```

matrix_of_frobenius (*p*, *prec=20*, *check=False*, *check_hypotheses=True*, *algorithm='auto'*)

Returns the matrix of Frobenius on the Monsky Washnitzer cohomology of the short Weierstrass model of the minimal model of the elliptic curve.

INPUT:

- `p` – prime (>= 3) for which E is good and ordinary
- `prec` – (relative) p-adic precision for result (default 20)
- `check` – boolean (default: `False`), whether to perform a consistency check. This will slow down the computation by a constant factor 2. (The consistency check is to verify that its trace is correct to the specified precision. Otherwise, the trace is used to compute one column from the other one (possibly after a change of basis).)
- `check_hypotheses` – boolean, whether to check that this is a curve for which the p-adic sigma function makes sense
- `algorithm` – one of "standard", "sqrtp", or "auto". This selects which version of Kedlaya's algorithm is used. The "standard" one is the one described in Kedlaya's paper. The "sqrtp" one has better performance for large p, but only works when $p > 6N$ (N = prec). The "auto" option selects "sqrtp" whenever possible.

  Note that if the "sqrtp" algorithm is used, a consistency check will automatically be applied, regardless of the setting of the "check" flag.

OUTPUT: a matrix of p-adic number to precision `prec`

See also the documentation of padic_E2.

EXAMPLES:

```
sage: E = EllipticCurve('37a1')
sage: E.matrix_of_frobenius(7)
[             2*7 + 4*7^2 + 5*7^4 + 6*7^5 + 6*7^6 + 7^8 + 4*7^9 + 3*7^10 +␣
↪2*7^11 + 5*7^12 + 4*7^14 + 7^16 + 2*7^17 + 3*7^18 + 4*7^19 + 3*7^20 + O(7^
↪21)                                    2 + 3*7 + 6*7^2 + 7^3 + 3*7^4 + 5*7^5␣
↪+ 3*7^7 + 7^8 + 3*7^9 + 6*7^13 + 7^14 + 7^16 + 5*7^17 + 4*7^18 + 7^19 + O(7^
↪20)]
[    2*7 + 3*7^2 + 7^3 + 3*7^4 + 6*7^5 + 2*7^6 + 3*7^7 + 5*7^8 + 3*7^9 + 2*7^
```

(continues on next page)

(continued from previous page)

```
↪11 + 6*7^12 + 5*7^13 + 4*7^16 + 4*7^17 + 6*7^18 + 6*7^19 + 4*7^20 + O(7^21)␣
↪6 + 4*7 + 2*7^2 + 6*7^3 + 7^4 + 6*7^7 + 5*7^8 + 2*7^9 + 3*7^10 + 4*7^11 + 7^
↪12 + 6*7^13 + 2*7^14 + 6*7^15 + 5*7^16 + 4*7^17 + 3*7^18 + 2*7^19 + O(7^20)]
sage: M = E.matrix_of_frobenius(11,prec=3); M
[   9*11 + 9*11^3 + O(11^4)          10 + 11 + O(11^3)]
[     2*11 + 11^2 + O(11^4) 6 + 11 + 10*11^2 + O(11^3)]
sage: M.det()
11 + O(11^4)
sage: M.trace()
6 + 10*11 + 10*11^2 + O(11^3)
sage: E.ap(11)
-5
sage: E = EllipticCurve('83a1')
sage: E.matrix_of_frobenius(3,6)
[                      2*3 + 3^5 + O(3^6)             2*3 + 2*3^2 + 2*3^3 +␣
↪O(3^6)]
[              2*3 + 3^2 + 2*3^5 + O(3^6) 2 + 2*3^2 + 2*3^3 + 2*3^4 + 3^5 +␣
↪O(3^6)]
```

minimal_model()

Return the unique minimal Weierstrass equation for this elliptic curve.

This is the model with minimal discriminant and $a_1, a_2, a_3 \in \{0, \pm 1\}$.

EXAMPLES:

```
sage: E = EllipticCurve([10,100,1000,10000,1000000])
sage: E.minimal_model()
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 + x + 1 over Rational␣
↪Field
```

minimal_quadratic_twist()

Determine a quadratic twist with minimal conductor. Return a global minimal model of the twist and the fundamental discriminant of the quadratic field over which they are isomorphic.

Note: If there is more than one curve with minimal conductor, the one returned is the one with smallest label (if in the database), or the one with minimal a-invariant list (otherwise).

Note: For curves with j-invariant 0 or 1728 the curve returned is the minimal quadratic twist, not necessarily the minimal twist (which would have conductor 27 or 32 respectively).

EXAMPLES:

```
sage: E = EllipticCurve('121d1')
sage: E.minimal_quadratic_twist()
(Elliptic Curve defined by y^2 + y = x^3 - x^2 over Rational Field, -11)
sage: Et, D = EllipticCurve('32a1').minimal_quadratic_twist()
sage: D
1

sage: E = EllipticCurve('11a1')
sage: Et, D = E.quadratic_twist(-24).minimal_quadratic_twist()
sage: E == Et
```

(continues on next page)

(continued from previous page)

```
True
sage: D
-24

sage: E = EllipticCurve([0,0,0,0,1000])
sage: E.minimal_quadratic_twist()
(Elliptic Curve defined by y^2 = x^3 + 1 over Rational Field, 40)
sage: E = EllipticCurve([0,0,0,1600,0])
sage: E.minimal_quadratic_twist()
(Elliptic Curve defined by y^2 = x^3 + 4*x over Rational Field, 5)
```

If the curve has square-free conductor then it is already minimal (see Issue #14060):

```
sage: E = next(cremona_optimal_curves([2*3*5*7*11]))
sage: (E, 1) == E.minimal_quadratic_twist()
True
```

An example where the minimal quadratic twist is not the minimal twist (which has conductor 27):

```
sage: E = EllipticCurve([0,0,0,0,7])
sage: E.j_invariant()
0
sage: E.minimal_quadratic_twist()[0].conductor()
5292
```

mod5family()

Return the family of all elliptic curves with the same mod-5 representation as `self`.

EXAMPLES:

```
sage: E = EllipticCurve('32a1')
sage: E.mod5family()
Elliptic Curve defined by y^2 = x^3 + 4*x
 over Fraction Field of Univariate Polynomial Ring in t over Rational Field
```

modular_degree(*algorithm='sympow'*, *M=1*)

Return the modular degree at level MN of this elliptic curve. The case $M == 1$ corresponds to the classical definition of modular degree.

When $M > 1$, the function returns the degree of the map from $X_0(MN) \to A$, where A is the abelian variety generated by embeddings of E into $J_0(MN)$.

The result is cached. Subsequent calls, even with a different algorithm, just returned the cached result. The algorithm argument is ignored when $M > 1$.

INPUT:

- `algorithm` – string:
  - `'sympow'` – (default) use Mark Watkin's (newer) C program sympow
  - `'magma'` – requires that MAGMA be installed (also implemented by Mark Watkins)
- `M` – non-negative integer; the modular degree at level MN is returned (see above)

Note: On 64-bit computers ec does not work, so Sage uses sympow even if ec is selected on a 64-bit computer.

The correctness of this function when called with algorithm "sympow" is subject to the following three hypothesis:

- Manin's conjecture: the Manin constant is 1
- Steven's conjecture: the $X_1(N)$-optimal quotient is the curve with minimal Faltings height. (This is proved in most cases.)
- The modular degree fits in a machine double, so it better be less than about 50-some bits. (If you use sympow this constraint does not apply.)

Moreover for all algorithms, computing a certain value of an L-function 'uses a heuristic method that discerns when the real-number approximation to the modular degree is within epsilon [=0.01 for algorithm='sympow'] of the same integer for 3 consecutive trials (which occur maybe every 25000 coefficients or so). Probably it could just round at some point. For rigour, you would need to bound the tail by assuming (essentially) that all the a_n are as large as possible, but in practice they exhibit significant (square root) cancellation. One difficulty is that it doesn't do the sum in 1-2-3-4 order; it uses 1-2-4-8–3-6-12-24-9-18- (Euler product style) instead, and so you have to guess ahead of time at what point to curtail this expansion.' (Quote from an email of Mark Watkins.)

Note: If the curve is loaded from the large Cremona database, then the modular degree is taken from the database.

EXAMPLES:

```
sage: E = EllipticCurve([0, -1, 1, -10, -20])
sage: E
Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field
sage: E.modular_degree()
1
sage: E = EllipticCurve('5077a')
sage: E.modular_degree()
1984
sage: factor(1984)
2^6 * 31
```

```
sage: EllipticCurve([0, 0, 1, -7, 6]).modular_degree()
1984
sage: EllipticCurve([0, 0, 1, -7, 6]).modular_degree(algorithm='sympow')
1984
sage: EllipticCurve([0, 0, 1, -7, 6]).modular_degree(algorithm='magma')  #␣
↪optional - magma
1984
```

We compute the modular degree of the curve with rank 4 having smallest (known) conductor:

```
sage: E = EllipticCurve([1, -1, 0, -79, 289])
sage: factor(E.conductor())  # conductor is 234446
2 * 117223
sage: factor(E.modular_degree())
2^7 * 2617
```

Higher level cases:

```
sage: E = EllipticCurve('11a')
sage: for M in range(1,11): print(E.modular_degree(M=M)) # long time (20s on␣
↪2009 MBP)
```

(continues on next page)

(continued from previous page)

```
1
1
3
2
7
45
12
16
54
245
```

modular_form()

Return the cuspidal modular form associated to this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: f = E.modular_form()
sage: f
q - 2*q^2 - 3*q^3 + 2*q^4 - 2*q^5 + O(q^6)
```

If you need to see more terms in the q-expansion:

```
sage: f.q_expansion(20)
q - 2*q^2 - 3*q^3 + 2*q^4 - 2*q^5 + 6*q^6 - q^7 + 6*q^9 + 4*q^10
 - 5*q^11 - 6*q^12 - 2*q^13 + 2*q^14 + 6*q^15 - 4*q^16 - 12*q^18 + O(q^20)
```

Note: If you just want the q-expansion, use *q_expansion()*.

modular_parametrization()

Return the modular parametrization of this elliptic curve, which is a map from $X_0(N)$ to self, where N is the conductor of `self`.

EXAMPLES:

```
sage: E = EllipticCurve('15a')
sage: phi = E.modular_parametrization(); phi
Modular parameterization
 from the upper half plane
   to Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 10*x - 10
      over Rational Field
sage: z = 0.1 + 0.2j
sage: phi(z)
(8.20822465478531 - 13.1562816054682*I : -8.79855099049364 + 69.
↪4006129342200*I : 1.00000000000000)
```

This map is actually a map on $X_0(N)$, so equivalent representatives in the upper half plane map to the same point:

```
sage: phi((-7*z-1)/(15*z+2))
(8.20822465478524 - 13.1562816054681*I : -8.79855099049... + 69.4006129342...
↪*I : 1.00000000000000)
```

We can also get a series expansion of this modular parameterization:

```
sage: E = EllipticCurve('389a1')
sage: X, Y = E.modular_parametrization().power_series()
sage: X
q^-2 + 2*q^-1 + 4 + 7*q + 13*q^2 + 18*q^3 + 31*q^4 + 49*q^5 + 74*q^6 + 111*q^
↪7 + 173*q^8 + 251*q^9 + 379*q^10 + 560*q^11 + 824*q^12 + 1199*q^13 + 1773*q^
↪14 + 2548*q^15 + 3722*q^16 + 5374*q^17 + O(q^18)
sage: Y
-q^-3 - 3*q^-2 - 8*q^-1 - 17 - 33*q - 61*q^2 - 110*q^3 - 186*q^4 - 320*q^5 -␣
↪528*q^6 - 861*q^7 - 1383*q^8 - 2218*q^9 - 3472*q^10 - 5451*q^11 - 8447*q^12␣
↪- 13020*q^13 - 19923*q^14 - 30403*q^15 - 46003*q^16 + O(q^17)
```

The following should give 0, but only approximately:

```
sage: q = X.parent().gen()
sage: E.defining_polynomial()(X,Y,1) + O(q^11) == 0
True
```

modular_symbol (*sign=1*, *normalize=None*, *implementation='eclib'*, *nap=0*)

Return the modular symbol map associated to this elliptic curve with given sign.

INPUT:

- `sign` – +1 (default) or -1.
- `normalize` – (default: `None`); either 'L_ratio', 'period', or 'none'; ignored unless `implementation` is 'sage'. For 'L_ratio', the modular symbol tries to normalize correctly as explained below by comparing it to `L_ratio` for the curve and some small twists. The normalization 'period' uses the `integral_period_map` for modular symbols which is known to be equal to the desired normalization, up to the sign and a possible power of 2. With normalization 'none', the modular symbol is almost certainly not correctly normalized, i.e. all values will be a fixed scalar multiple of what they should be.
- `implementation` – either 'eclib' (default), 'sage' or 'num'. Here, 'eclib' uses Cremona's `C++` implementation in the `eclib` library, 'sage' uses an implementation within Sage which is often quite a bit slower, and 'num' uses Wuthrich's implementation of numerical modular symbols.
- `nap` – (int, default 0); ignored unless implementation is 'eclib'. The number of ap of E to use in determining the normalisation of the modular symbols. If 0 (the default), then the value of 100*E.conductor().isqrt() is used. Using too small a value can lead to incorrect normalisation.

DEFINITION:

The modular symbol map sends any rational number r to the rational number whichis the ratio of the real or imaginary part (depending on the sign) of the integral of $2\pi i f(z)dz$ from ∞ to r, where f is the newform attached to E, to the real or imaginary period of E.

More precisely: If the sign is +1, then the value returned is the quotient of the real part of this integral by the least positive period Ω_E^+ of E. In particular for $r = 0$, the value is equal to $L(E,1)/\Omega_E^+$ (unlike in `L_ratio` of `lseries()`, where the value is also divided by the number of connected components of $E(\mathbf{R})$). In particular the modular symbol depends on E and not only the isogeny class of E. For sign -1, it is the quotient of the imaginary part of the integral divided by the purely imaginary period of E with smallest positive imaginary part. Note however there is an issue about these normalizations, hence the optional argument `normalize` explained below

ALGORITHM:

For the implementations 'sage' and 'eclib', the used algorithm starts by finding the space of modular symbols within the full space of all modular symbols of that level. This initial step will take a very long time if the conductor is large (e.g. minutes for five digit conductors). Once the space is determined, each evaluation is very fast (logarithmic in the denominator of r).

The implementation 'num' uses a different algorithm. It uses numerical integration along paths in the upper half plane. The bounds are rigorously proved so that the outcome is known to be correct. The initial step costs no time, instead each evaluation will take more time than in the above. More information in the documentation of the class `ModularSymbolNumerical`.

See also:

modular_symbol_numerical()

EXAMPLES:

```
sage: E = EllipticCurve('37a1')
sage: M = E.modular_symbol(); M
Modular symbol with sign 1 over Rational Field attached to
 Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
sage: M(1/2)
0
sage: M(1/5)
1
```

```
sage: E = EllipticCurve('121b1')
sage: M = E.modular_symbol(implementation="sage")
Warning : Could not normalize the modular symbols, maybe all further results
will be multiplied by -1 and a power of 2
sage: M(1/7)
-1/2
```

With the numerical version, rather high conductors can be computed:

```
sage: E = EllipticCurve([999,997])
sage: E.conductor()
16059400956
sage: m = E.modular_symbol(implementation="num")
sage: m(0) # long time
16
```

Different curves in an isogeny class have modular symbols which differ by a nonzero rational factor:

```
sage: E1 = EllipticCurve('11a1')
sage: M1 = E1.modular_symbol()
sage: M1(0)
1/5
sage: E2 = EllipticCurve('11a2')
sage: M2 = E2.modular_symbol()
sage: M2(0)
1
sage: E3 = EllipticCurve('11a3')
sage: M3 = E3.modular_symbol()
sage: M3(0)
1/25
sage: all(5*M1(r)==M2(r)==25*M3(r) for r in QQ.range_by_height(10))
True
```

With the default implementation using `eclib`, the symbols are correctly normalized automatically. With the `Sage` implementation we can choose to normalize using the L-ratio, unless that is 0 (for curves of positive rank) or using periods. Here is an example where the symbol is already normalized:

```
sage: E = EllipticCurve('11a2')
sage: E.modular_symbol(implementation = 'eclib')(0)
1
sage: E.modular_symbol(implementation = 'sage', normalize='L_ratio')(0)
1
sage: E.modular_symbol(implementation = 'sage', normalize='none')(0)
1
sage: E.modular_symbol(implementation = 'sage', normalize='period')(0)
1
```

Here is an example where both normalization methods work, while the non-normalized symbol is incorrect:

```
sage: E = EllipticCurve('11a3')
sage: E.modular_symbol(implementation = 'eclib')(0)
1/25
sage: E.modular_symbol(implementation = 'sage', normalize='none')(0)
1
sage: E.modular_symbol(implementation = 'sage', normalize='L_ratio')(0)
1/25
sage: E.modular_symbol(implementation = 'sage', normalize='period')(0)
1/25
```

Since Issue #10256, the interface for negative modular symbols in eclib is available:

```
sage: E = EllipticCurve('11a1')
sage: Mplus = E.modular_symbol(+1); Mplus
Modular symbol with sign 1 over Rational Field attached to
 Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field
sage: [Mplus(1/i) for i in [1..11]]
[1/5, -4/5, -3/10, 7/10, 6/5, 6/5, 7/10, -3/10, -4/5, 1/5, 0]
sage: Mminus = E.modular_symbol(-1); Mminus
Modular symbol with sign -1 over Rational Field attached to
 Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field
sage: [Mminus(1/i) for i in [1..11]]
[0, 0, 1/2, 1/2, 0, 0, -1/2, -1/2, 0, 0, 0]
```

With older version of eclib, in the default 'eclib' implementation, if `nap` is too small, the normalization may be computed incorrectly (see Issue #31317). This was fixed in eclib version v20210310, since now eclib increase `nap` automatically. The following used to give incorrect results. See Issue #31443:

```
sage: E = EllipticCurve('1590g1')
sage: m = E.modular_symbol(nap=300)      # long time
sage: [m(a/5) for a in [1..4]]           # long time
[13/2, -13/2, -13/2, 13/2]
```

These values are correct, as verified by the numerical implementation:

```
sage: m = E.modular_symbol(implementation='num')
sage: [m(a/5) for a in [1..4]]
[13/2, -13/2, -13/2, 13/2]
```

modular_symbol_numerical (*sign=1*, *prec=20*)

Return the modular symbol as a numerical function.

Just as in *modular_symbol()* this returns a function that maps any rational r to a real number that should be equal to the rational number with an error smaller than the given binary precision. In practice the precision is often much higher. See the examples below. The normalisation is the same.

INPUT:

- `sign` – either +1 (default) or -1
- `prec` – an integer (default 20)

OUTPUT:

- a real number

ALGORITHM:

This method does not compute spaces of modular symbols, so it is suitable for curves of larger conductor than can be handled by *`modular_symbol()`*. It is essentially the same implementation as `modular_symbol` with implementation set to 'num'. However the precision is not automatically chosen to be certain that the output is equal to the rational number it approximates.

For large conductors one should set the `prec` very small.

EXAMPLES:

```
sage: E = EllipticCurve('19a1')
sage: f = E.modular_symbol_numerical(1)
sage: g = E.modular_symbol(1)
sage: f(0), g(0)  # abs tol 1e-11
(0.333333333333333, 1/3)

sage: E = EllipticCurve('5077a1')
sage: f = E.modular_symbol_numerical(-1, prec=2)
sage: f(0)        # abs tol 1e-11
0.000000000000000
sage: f(1/7)      # abs tol 1e-11
0.999844176260303

sage: E = EllipticCurve([123,456])
sage: E.conductor()
104461920
sage: f = E.modular_symbol_numerical(prec=2)
sage: f(0)        # abs tol 1e-11
2.00001004772210
```

`modular_symbol_space`(*sign=1*, *base_ring=Rational Field*, *bound=None*)

Return the space of cuspidal modular symbols associated to this elliptic curve, with given sign and base ring.

INPUT:

- `sign` – 0, -1, or 1
- `base_ring` – a ring

EXAMPLES:

```
sage: f = EllipticCurve('37b')
sage: f.modular_symbol_space()
Modular Symbols subspace of dimension 1 of Modular Symbols space
 of dimension 3 for Gamma_0(37) of weight 2 with sign 1 over Rational Field
sage: f.modular_symbol_space(-1)
Modular Symbols subspace of dimension 1 of Modular Symbols space
 of dimension 2 for Gamma_0(37) of weight 2 with sign -1 over Rational Field
sage: f.modular_symbol_space(0, bound=3)
Modular Symbols subspace of dimension 2 of Modular Symbols space
 of dimension 5 for Gamma_0(37) of weight 2 with sign 0 over Rational Field
```

Note: If you just want the q-expansion, use `q_expansion()`.

mwrank(*options=''*)

Run Cremona's mwrank program on this elliptic curve and return the result as a string.

INPUT:

- `options` (string) – run-time options passed when starting mwrank. The format is as follows (see below for examples of usage):
  - `-v n` (verbosity level) sets verbosity to n (default=1)
  - `-o` (PARI/GP style output flag) turns ON extra PARI/GP short output (default is OFF)
  - `-p n` (precision) sets precision to n decimals (default=15)
  - `-b n` (quartic bound) bound on quartic point search (default=10)
  - `-x n` (n_aux) number of aux primes used for sieving (default=6)
  - `-l` (generator list flag) turns ON listing of points (default ON unless v=0)
  - `-s` (selmer_only flag) if set, computes Selmer rank only (default: not set)
  - `-d` (skip_2nd_descent flag) if set, skips the second descent for curves with 2-torsion (default: not set)
  - `-S n` (sat_bd) upper bound on saturation primes (default=100, -1 for automatic)

OUTPUT:

- (string) – output of mwrank on this curve

Note: The output is a raw string and completely illegible using automatic display, so it is recommended to use print for legible output.

EXAMPLES:

```
sage: E = EllipticCurve('37a1')
sage: E.mwrank() #random
...
sage: print(E.mwrank())
Curve [0,0,1,-1,0] :        Basic pair: I=48, J=-432
disc=255744
...
Generator 1 is [0:-1:1]; height 0.05111...

Regulator = 0.05111...

The rank and full Mordell-Weil basis have been determined unconditionally.
...
```

Options to mwrank can be passed:

```
sage: E = EllipticCurve([0,0,0,877,0])
```

Run mwrank with `'verbose'` flag set to 0 but list generators if found:

```
sage: print(E.mwrank('-v0 -l'))
Curve [0,0,0,877,0] :   0 <= rank <= 1
Regulator = 1
```

Run mwrank again, this time with a higher bound for point searching on homogeneous spaces:

```
sage: print(E.mwrank('-v0 -l -b11'))
Curve [0,0,0,877,0] :   Rank = 1
Generator 1 is␣
↪[29604565304828237474403861024284371796799791624792913256602210:-
↪25625626798892680938877683404551308964866915320435660346478694949007802321978758895980293
↪ height 95.98037...
Regulator = 95.98037...
```

mwrank_curve(*verbose=False*)

Construct an mwrank_EllipticCurve from this elliptic curve

The resulting mwrank_EllipticCurve has available methods from John Cremona's eclib library.

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: EE = E.mwrank_curve()
sage: EE
y^2 + y = x^3 - x^2 - 10 x - 20
sage: type(EE)
<class 'sage.libs.eclib.interface.mwrank_EllipticCurve'>
sage: EE.isogeny_class()
([[0, -1, 1, -10, -20], [0, -1, 1, -7820, -263580], [0, -1, 1, 0, 0]],
 [[0, 5, 5], [5, 0, 0], [5, 0, 0]])
```

newform()

Same as `self.modular_form()`.

EXAMPLES:

```
sage: E = EllipticCurve('37a1')
sage: E.newform()
q - 2*q^2 - 3*q^3 + 2*q^4 - 2*q^5 + O(q^6)
sage: E.newform() == E.modular_form()
True
```

ngens(*proof=None*)

Return the number of generators of this elliptic curve.

Note: See *gens()* for further documentation. The function *ngens()* calls *gens()* if not already done, but only with default parameters. Better results may be obtained by calling *mwrank()* with carefully chosen parameters.

EXAMPLES:

```
sage: E = EllipticCurve('37a1')
sage: E.ngens()
1

sage: E = EllipticCurve([0,0,0,877,0])
```

(continues on next page)

(continued from previous page)

```
sage: E.ngens()
1

sage: print(E.mwrank('-v0 -b12 -l'))
Curve [0,0,0,877,0] :   Rank = 1
Generator 1 is␣
↪[29604565304828237474403861024284371796799791624792913256602210:-
↪25625626798892680938877683404551308964866915320435660346478694 9:490078023219787588959802933
↪ height 95.98037...
Regulator = 95.980...
```

optimal_curve()

Given an elliptic curve that is in the installed Cremona database, return the optimal curve isogenous to it.

EXAMPLES:

The following curve is not optimal:

```
sage: E = EllipticCurve('11a2'); E
Elliptic Curve defined by y^2 + y = x^3 - x^2 - 7820*x - 263580 over Rational␣
↪Field
sage: E.optimal_curve()
Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field
sage: E.optimal_curve().cremona_label()
'11a1'
```

Note that 990h is the special case where the optimal curve isn't the first in the Cremona labeling:

```
sage: E = EllipticCurve('990h4'); E
Elliptic Curve defined by y^2 + x*y + y = x^3 - x^2 + 6112*x - 41533 over␣
↪Rational Field
sage: F = E.optimal_curve(); F
Elliptic Curve defined by y^2 + x*y + y = x^3 - x^2 - 1568*x - 4669 over␣
↪Rational Field
sage: F.cremona_label()
'990h3'
sage: EllipticCurve('990a1').optimal_curve().cremona_label()    # a isn't h.
'990a1'
```

If the input curve is optimal, this function returns that curve (not just a copy of it or a curve isomorphic to it!):

```
sage: E = EllipticCurve('37a1')
sage: E.optimal_curve() is E
True
```

Also, if this curve is optimal but not given by a minimal model, this curve will still be returned, so this function need not return a minimal model in general.

```
sage: F = E.short_weierstrass_model(); F
Elliptic Curve defined by y^2  = x^3 - 16*x + 16 over Rational Field
sage: F.optimal_curve()
Elliptic Curve defined by y^2  = x^3 - 16*x + 16 over Rational Field
```

ordinary_primes(B)

Return a list of all ordinary primes for this elliptic curve up to and possibly including B.

EXAMPLES:

```
sage: e = EllipticCurve('11a')
sage: e.aplist(20)
[-2, -1, 1, -2, 1, 4, -2, 0]
sage: e.ordinary_primes(97)
[3, 5, 7, 11, 13, 17, 23, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83,␣
↪89, 97]
sage: e = EllipticCurve('49a')
sage: e.aplist(20)
[1, 0, 0, 0, 4, 0, 0, 0]
sage: e.supersingular_primes(97)
[3, 5, 13, 17, 19, 31, 41, 47, 59, 61, 73, 83, 89, 97]
sage: e.ordinary_primes(97)
[2, 11, 23, 29, 37, 43, 53, 67, 71, 79]
sage: e.ordinary_primes(3)
[2]
sage: e.ordinary_primes(2)
[2]
sage: e.ordinary_primes(1)
[]
```

padic_E2 (*p*, *prec=20*, *check=False*, *check_hypotheses=True*, *algorithm='auto'*)

Returns the value of the p-adic modular form $E2$ for (E, ω) where ω is the usual invariant differential $dx/(2y + a_1x + a_3)$.

INPUT:

- `p` – prime (= 5) for which E is good and ordinary
- `prec` – (relative) p-adic precision (= 1) for result
- `check` – boolean, whether to perform a consistency check. This will slow down the computation by a constant factor 2. (The consistency check is to compute the whole matrix of frobenius on Monsky-Washnitzer cohomology, and verify that its trace is correct to the specified precision. Otherwise, the trace is used to compute one column from the other one (possibly after a change of basis).)
- `check_hypotheses` – boolean, whether to check that this is a curve for which the p-adic sigma function makes sense
- `algorithm` – one of "standard", "sqrtp", or "auto". This selects which version of Kedlaya's algorithm is used. The "standard" one is the one described in Kedlaya's paper. The "sqrtp" one has better performance for large p, but only works when $p > 6N$ (N = prec). The "auto" option selects "sqrtp" whenever possible.

  Note that if the "sqrtp" algorithm is used, a consistency check will automatically be applied, regardless of the setting of the "check" flag.

OUTPUT: p-adic number to precision prec

Note: If the discriminant of the curve has nonzero valuation at p, then the result will not be returned mod p^{prec}, but it still *will* have prec *digits* of precision.

Todo: Once we have a better implementation of the "standard" algorithm, the algorithm selection strategy for "auto" needs to be revisited.

AUTHORS:

- David Harvey (2006-09-01): partly based on code written by Robert Bradshaw at the MSRI 2006 modular forms workshop

ACKNOWLEDGMENT: - discussion with Eyal Goren that led to the trace trick.

EXAMPLES: Here is the example discussed in the paper "Computation of p-adic Heights and Log Convergence" (Mazur, Stein, Tate) [MST2006]:

```
sage: EllipticCurve([-1, 1/4]).padic_E2(5)
2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + 4*5^10 + 2*5^11 + 2*5^
↪12 + 2*5^14 + 3*5^15 + 3*5^16 + 3*5^17 + 4*5^18 + 2*5^19 + O(5^20)
```

Let's try to higher precision (this is the same answer the MAGMA implementation gives):

```
sage: EllipticCurve([-1, 1/4]).padic_E2(5, 100)
2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + 4*5^10 + 2*5^11 + 2*5^
↪12 + 2*5^14 + 3*5^15 + 3*5^16 + 3*5^17 + 4*5^18 + 2*5^19 + 4*5^20 + 5^21 +␣
↪4*5^22 + 2*5^23 + 3*5^24 + 3*5^26 + 2*5^27 + 3*5^28 + 2*5^30 + 5^31 + 4*5^
↪33 + 3*5^34 + 4*5^35 + 5^36 + 4*5^37 + 4*5^38 + 3*5^39 + 4*5^41 + 2*5^42 +␣
↪3*5^43 + 2*5^44 + 2*5^48 + 3*5^49 + 4*5^50 + 2*5^51 + 5^52 + 4*5^53 + 4*5^
↪54 + 3*5^55 + 2*5^56 + 3*5^57 + 4*5^58 + 4*5^59 + 5^60 + 3*5^61 + 5^62 +␣
↪4*5^63 + 5^65 + 3*5^66 + 2*5^67 + 5^69 + 2*5^70 + 3*5^71 + 3*5^72 + 5^74 +␣
↪5^75 + 5^76 + 3*5^77 + 4*5^78 + 4*5^79 + 2*5^80 + 3*5^81 + 5^82 + 5^83 +␣
↪4*5^84 + 3*5^85 + 2*5^86 + 3*5^87 + 5^88 + 2*5^89 + 4*5^90 + 4*5^92 + 3*5^
↪93 + 4*5^94 + 3*5^95 + 2*5^96 + 4*5^97 + 4*5^98 + 2*5^99 + O(5^100)
```

Check it works at low precision too:

```
sage: EllipticCurve([-1, 1/4]).padic_E2(5, 1)
2 + O(5)
sage: EllipticCurve([-1, 1/4]).padic_E2(5, 2)
2 + 4*5 + O(5^2)
sage: EllipticCurve([-1, 1/4]).padic_E2(5, 3)
2 + 4*5 + O(5^3)
```

TODO: With the old(-er), i.e., = sage-2.4 p-adics we got $5 + O(5^2)$ as output, i.e., relative precision 1, but with the newer p-adics we get relative precision 0 and absolute precision 1.

```
sage: EllipticCurve([1, 1, 1, 1, 1]).padic_E2(5, 1)
O(5)
```

Check it works for different models of the same curve (37a), even when the discriminant changes by a power of p (note that E2 depends on the differential too, which is why it gets scaled in some of the examples below):

```
sage: X1 = EllipticCurve([-1, 1/4])
sage: X1.j_invariant(), X1.discriminant()
 (110592/37, 37)
sage: X1.padic_E2(5, 10)
 2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + O(5^10)
```

```
sage: X2 = EllipticCurve([0, 0, 1, -1, 0])
sage: X2.j_invariant(), X2.discriminant()
 (110592/37, 37)
sage: X2.padic_E2(5, 10)
 2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + O(5^10)
```

```
sage: X3 = EllipticCurve([-1*(2**4), 1/4*(2**6)])
sage: X3.j_invariant(), X3.discriminant() / 2**12
 (110592/37, 37)
sage: 2**(-2) * X3.padic_E2(5, 10)
 2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + O(5^10)
```

```
sage: X4 = EllipticCurve([-1*(7**4), 1/4*(7**6)])
sage: X4.j_invariant(), X4.discriminant() / 7**12
 (110592/37, 37)
sage: 7**(-2) * X4.padic_E2(5, 10)
 2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + O(5^10)
```

```
sage: X5 = EllipticCurve([-1*(5**4), 1/4*(5**6)])
sage: X5.j_invariant(), X5.discriminant() / 5**12
 (110592/37, 37)
sage: 5**(-2) * X5.padic_E2(5, 10)
 2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + O(5^10)
```

```
sage: X6 = EllipticCurve([-1/(5**4), 1/4/(5**6)])
sage: X6.j_invariant(), X6.discriminant() * 5**12
 (110592/37, 37)
sage: 5**2 * X6.padic_E2(5, 10)
 2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + O(5^10)
```

Test check=True vs check=False:

```
sage: EllipticCurve([-1, 1/4]).padic_E2(5, 1, check=False)
2 + O(5)
sage: EllipticCurve([-1, 1/4]).padic_E2(5, 1, check=True)
2 + O(5)
sage: EllipticCurve([-1, 1/4]).padic_E2(5, 30, check=False)
2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + 4*5^10 + 2*5^11 + 2*5^
↪12 + 2*5^14 + 3*5^15 + 3*5^16 + 3*5^17 + 4*5^18 + 2*5^19 + 4*5^20 + 5^21 +␣
↪4*5^22 + 2*5^23 + 3*5^24 + 3*5^26 + 2*5^27 + 3*5^28 + O(5^30)
sage: EllipticCurve([-1, 1/4]).padic_E2(5, 30, check=True)
2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + 5^8 + 3*5^9 + 4*5^10 + 2*5^11 + 2*5^
↪12 + 2*5^14 + 3*5^15 + 3*5^16 + 3*5^17 + 4*5^18 + 2*5^19 + 4*5^20 + 5^21 +␣
↪4*5^22 + 2*5^23 + 3*5^24 + 3*5^26 + 2*5^27 + 3*5^28 + O(5^30)
```

Here's one using the $p^{1/2}$ algorithm:

```
sage: EllipticCurve([-1, 1/4]).padic_E2(3001, 3, algorithm="sqrtp")
1907 + 2819*3001 + 1124*3001^2 + O(3001^3)
```

padic_height (*p*, *prec=20*, *sigma=None*, *check_hypotheses=True*)

Compute the cyclotomic p-adic height.

The equation of the curve must be integral and minimal at p.

INPUT:

- `p` – prime >= 5 for which the curve has semi-stable reduction
- `prec` – integer >= 1 (default 20), desired precision of result
- `sigma` – precomputed value of sigma. If not supplied, this function will call padic_sigma to compute it.

- `check_hypotheses` – boolean, whether to check that this is a curve for which the p-adic height makes sense

OUTPUT: A function that accepts two parameters:

- a **Q**-rational point on the curve whose height should be computed
- optional boolean flag 'check': if False, it skips some input checking, and returns the p-adic height of that point to the desired precision.
- The normalization (sign and a factor 1/2 with respect to some other normalizations that appear in the literature) is chosen in such a way as to make the p-adic Birch Swinnerton-Dyer conjecture hold as stated in [MTT1986].

AUTHORS:

- Jennifer Balakrishnan: original code developed at the 2006 MSRI graduate workshop on modular forms
- David Harvey (2006-09-13): integrated into Sage, optimised to speed up repeated evaluations of the returned height function, addressed some thorny precision questions
- David Harvey (2006-09-30): rewrote to use division polynomials for computing denominator of nP.
- David Harvey (2007-02): cleaned up according to algorithms in "Efficient Computation of p-adic Heights"
- Chris Wuthrich (2007-05): added supersingular and multiplicative heights

EXAMPLES:

```
sage: E = EllipticCurve("37a")
sage: P = E.gens()[0]
sage: h = E.padic_height(5, 10)
sage: h(P)
5 + 5^2 + 5^3 + 3*5^6 + 4*5^7 + 5^9 + O(5^10)
```

An anomalous case:

```
sage: h = E.padic_height(53, 10)
sage: h(P)
26*53^-1 + 30 + 20*53 + 47*53^2 + 10*53^3 + 32*53^4 + 9*53^5 + 22*53^6 +␣
↪35*53^7 + 30*53^8 + 17*53^9 + O(53^10)
```

Boundary case:

```
sage: E.padic_height(5, 3)(P)
5 + 5^2 + O(5^3)
```

A case that works the division polynomial code a little harder:

```
sage: E.padic_height(5, 10)(5*P)
5^3 + 5^4 + 5^5 + 3*5^8 + 4*5^9 + O(5^10)
```

Check that answers agree over a range of precisions:

```
sage: max_prec = 30    # make sure we get past p^2    # long time
sage: full = E.padic_height(5, max_prec)(P)           # long time
sage: for prec in range(1, max_prec):                 # long time
....:     assert E.padic_height(5, prec)(P) == full
```

A supersingular prime for a curve:

```
sage: E = EllipticCurve('37a')
sage: E.is_supersingular(3)
True
sage: h = E.padic_height(3, 5)
sage: h(E.gens()[0])
(3 + 3^3 + O(3^6), 2*3^2 + 3^3 + 3^4 + 3^5 + 2*3^6 + O(3^7))
sage: E.padic_regulator(5)
5 + 5^2 + 5^3 + 3*5^6 + 4*5^7 + 5^9 + 5^10 + 3*5^11 + 3*5^12 + 5^13 + 4*5^14␣
↪+ 5^15 + 2*5^16 + 5^17 + 2*5^18 + 4*5^19 + O(5^20)
sage: E.padic_regulator(3, 5)
(3 + 2*3^2 + 3^3 + O(3^4), 3^2 + 2*3^3 + 3^4 + O(3^5))
```

A torsion point in both the good and supersingular cases:

```
sage: E = EllipticCurve('11a')
sage: P = E.torsion_subgroup().gen(0).element(); P
(5 : 5 : 1)
sage: h = E.padic_height(19, 5)
sage: h(P)
0
sage: h = E.padic_height(5, 5)
sage: h(P)
0
```

The result is not dependent on the model for the curve:

```
sage: E = EllipticCurve([0,0,0,0,2^12*17])
sage: Em = E.minimal_model()
sage: P = E.gens()[0]
sage: Pm = Em.gens()[0]
sage: h = E.padic_height(7)
sage: hm = Em.padic_height(7)
sage: h(P) == hm(Pm)
True
```

padic_height_pairing_matrix(*p*, *prec=20*, *height=None*, *check_hypotheses=True*)

Computes the cyclotomic p-adic height pairing matrix of this curve with respect to the basis `self.gens()` for the Mordell-Weil group for a given odd prime p of good ordinary reduction. The model needs to be integral and minimal at p.

INPUT:

- `p` – prime >= 5
- `prec` – answer will be returned modulo p^{prec}
- `height` – precomputed height function. If not supplied, this function will call padic_height to compute it.
- `check_hypotheses` – boolean, whether to check that this is a curve for which the p-adic height makes sense

OUTPUT: The p-adic cyclotomic height pairing matrix of this curve to the given precision.

AUTHORS:

- David Harvey, Liang Xiao, Robert Bradshaw, Jennifer Balakrishnan: original implementation at the 2006 MSRI graduate workshop on modular forms
- David Harvey (2006-09-13): cleaned up and integrated into Sage, removed some redundant height computations

EXAMPLES:

```
sage: E = EllipticCurve("37a")
sage: E.padic_height_pairing_matrix(5, 10)
[5 + 5^2 + 5^3 + 3*5^6 + 4*5^7 + 5^9 + O(5^10)]
```

A rank two example:

```
sage: e =EllipticCurve('389a')
sage: e._set_gens([e(-1, 1), e(1,0)])  # avoid platform dependent gens
sage: e.padic_height_pairing_matrix(5,10)
[                      3*5 + 2*5^2 + 5^4 + 5^5 + 5^7 + 4*5^9 + O(5^10) 5 +␣
↪4*5^2 + 5^3 + 2*5^4 + 3*5^5 + 4*5^6 + 5^7 + 5^8 + 2*5^9 + O(5^10)]
[5 + 4*5^2 + 5^3 + 2*5^4 + 3*5^5 + 4*5^6 + 5^7 + 5^8 + 2*5^9 + O(5^10)       ␣
↪                 4*5 + 2*5^4 + 3*5^6 + 4*5^7 + 4*5^8 + O(5^10)]
```

An anomalous rank 3 example:

```
sage: e = EllipticCurve("5077a")
sage: e._set_gens([e(-1,3), e(2,0), e(4,6)])
sage: e.padic_height_pairing_matrix(5,4)
[4 + 3*5 + 4*5^2 + 4*5^3 + O(5^4)       4 + 4*5^2 + 2*5^3 + O(5^4)       3*5␣
↪+ 4*5^2 + 5^3 + O(5^4)]
[      4 + 4*5^2 + 2*5^3 + O(5^4)   3 + 4*5 + 3*5^2 + 5^3 + O(5^4)           ␣
↪      2 + 4*5 + O(5^4)]
[      3*5 + 4*5^2 + 5^3 + O(5^4)                 2 + 4*5 + O(5^4)     1 +␣
↪3*5 + 5^2 + 5^3 + O(5^4)]
```

padic_height_via_multiply(*p*, *prec=20*, *E2=None*, *check_hypotheses=True*)

Computes the cyclotomic p-adic height.

The equation of the curve must be minimal at p.

INPUT:

- `p` – prime >= 5 for which the curve has good ordinary reduction
- `prec` – integer >= 2 (default 20), desired precision of result
- `E2` – precomputed value of E2. If not supplied, this function will call padic_E2 to compute it. The value supplied must be correct mod p^{prec-2} (or slightly higher in the anomalous case; see the code for details).
- `check_hypotheses` – boolean, whether to check that this is a curve for which the p-adic height makes sense

OUTPUT: A function that accepts two parameters:

- a **Q**-rational point on the curve whose height should be computed
- optional boolean flag 'check': if False, it skips some input checking, and returns the p-adic height of that point to the desired precision.
- The normalization (sign and a factor 1/2 with respect to some other normalizations that appear in the literature) is chosen in such a way as to make the p-adic Birch Swinnerton-Dyer conjecture hold as stated in [MTT1986].

AUTHORS:

- David Harvey (2008-01): based on the padic_height() function, using the algorithm of [Har2009].

EXAMPLES:

```
sage: E = EllipticCurve("37a")
sage: P = E.gens()[0]
sage: h = E.padic_height_via_multiply(5, 10)
sage: h(P)
5 + 5^2 + 5^3 + 3*5^6 + 4*5^7 + 5^9 + O(5^10)
```

An anomalous case:

```
sage: h = E.padic_height_via_multiply(53, 10)
sage: h(P)
26*53^-1 + 30 + 20*53 + 47*53^2 + 10*53^3 + 32*53^4 + 9*53^5 + 22*53^6 +␣
↪35*53^7 + 30*53^8 + 17*53^9 + O(53^10)
```

Supply the value of E2 manually:

```
sage: E2 = E.padic_E2(5, 8)
sage: E2
2 + 4*5 + 2*5^3 + 5^4 + 3*5^5 + 2*5^6 + O(5^8)
sage: h = E.padic_height_via_multiply(5, 10, E2=E2)
sage: h(P)
5 + 5^2 + 5^3 + 3*5^6 + 4*5^7 + 5^9 + O(5^10)
```

Boundary case:

```
sage: E.padic_height_via_multiply(5, 3)(P)
5 + 5^2 + O(5^3)
```

Check that answers agree over a range of precisions:

```
sage: max_prec = 30    # make sure we get past p^2    # long time
sage: full = E.padic_height(5, max_prec)(P)           # long time
sage: for prec in range(2, max_prec):                 # long time
....:     assert E.padic_height_via_multiply(5, prec)(P) == full
```

padic_lseries(*p*, *normalize=None*, *implementation='eclib'*, *precision=None*)

Return the p-adic L-series of self at p, which is an object whose approx method computes approximation to the true p-adic L-series to any desired precision.

INPUT:

- `p` – prime
- `normalize` – 'L_ratio' (default), 'period' or 'none'; this is describes the way the modular symbols are normalized. See modular_symbol for more details.
- `implementation` – 'eclib' (default), 'sage', 'num' or 'pollackstevens'; Whether to use John Cremona's eclib, the Sage implementation, numerical modular symbols or Pollack-Stevens' implementation of overconvergent modular symbols.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: L = E.padic_lseries(5); L
5-adic L-series of Elliptic Curve defined by y^2 + y = x^3 - x over Rational␣
↪Field
sage: type(L)
<class 'sage.schemes.elliptic_curves.padic_lseries.pAdicLseriesOrdinary'>
```

We compute the 3-adic L-series of two curves of rank 0 and in each case verify the interpolation property for their leading coefficient (i.e., value at 0):

```
sage: e = EllipticCurve('11a')
sage: ms = e.modular_symbol()
sage: [ms(1/11), ms(1/3), ms(0), ms(oo)]
[0, -3/10, 1/5, 0]
sage: ms(0)
1/5
sage: L = e.padic_lseries(3)
sage: P = L.series(5)
sage: P(0)
2 + 3 + 3^2 + 2*3^3 + 2*3^5 + 3^6 + O(3^7)
sage: alpha = L.alpha(9); alpha
2 + 3^2 + 2*3^3 + 2*3^4 + 2*3^6 + 3^8 + O(3^9)
sage: R.<x> = QQ[]
sage: f = x^2 - e.ap(3)*x + 3
sage: f(alpha)
O(3^9)
sage: r = e.lseries().L_ratio(); r
1/5
sage: (1 - alpha^(-1))^2 * r
2 + 3 + 3^2 + 2*3^3 + 2*3^5 + 3^6 + 3^7 + O(3^9)
sage: P(0)
2 + 3 + 3^2 + 2*3^3 + 2*3^5 + 3^6 + O(3^7)
```

Next consider the curve 37b:

```
sage: e = EllipticCurve('37b')
sage: L = e.padic_lseries(3)
sage: P = L.series(5)
sage: alpha = L.alpha(9); alpha
1 + 2*3 + 3^2 + 2*3^5 + 2*3^7 + 3^8 + O(3^9)
sage: r = e.lseries().L_ratio(); r
1/3
sage: (1 - alpha^(-1))^2 * r
3 + 3^2 + 2*3^4 + 2*3^5 + 2*3^6 + 3^7 + O(3^9)
sage: P(0)
3 + 3^2 + 2*3^4 + 2*3^5 + O(3^6)
```

We can use Sage modular symbols instead to compute the L-series:

```
sage: e = EllipticCurve('11a')
sage: L = e.padic_lseries(3, implementation = 'sage')
sage: L.series(5,prec=10)
2 + 3 + 3^2 + 2*3^3 + 2*3^5 + 3^6 + O(3^7) + (1 + 3 + 2*3^2 + 3^3 + O(3^4))*T␣
↪+ (1 + 2*3 + O(3^4))*T^2 + (3 + 2*3^2 + O(3^3))*T^3 + (2*3 + 3^2 + O(3^
↪3))*T^4 + (2 + 2*3 + 2*3^2 + O(3^3))*T^5 + (1 + 3^2 + O(3^3))*T^6 + (2 + 3^
↪2 + O(3^3))*T^7 + (2 + 2*3 + 2*3^2 + O(3^3))*T^8 + (2 + O(3^2))*T^9 + O(T^
↪10)
```

Also the numerical modular symbols can be used. This may allow for much larger conductor in some instances:

```
sage: E = EllipticCurve([101,103])
sage: L = E.padic_lseries(5, implementation="num")
sage: L.series(2)
O(5^4) + (3 + O(5))*T + (1 + O(5))*T^2 + (3 + O(5))*T^3 + O(5)*T^4 + O(T^5)
```

Finally, we can use the overconvergent method of Pollack-Stevens.:

```
sage: e = EllipticCurve('11a')
sage: L = e.padic_lseries(3, implementation = 'pollackstevens', precision = 6)
sage: L.series(5)
2 + 3 + 3^2 + 2*3^3 + 2*3^5 + O(3^6) + (1 + 3 + 2*3^2 + 3^3 + O(3^4))*T + (1␣
↪+ 2*3 + O(3^2))*T^2 + (3 + O(3^2))*T^3 + O(3^0)*T^4 + O(T^5)
sage: L[3]
3 + O(3^2)
```

Another example with a semistable prime.:

```
sage: E = EllipticCurve("11a1")
sage: L = E.padic_lseries(11, implementation = 'pollackstevens', precision=3)
sage: L[1]
10 + 3*11 + O(11^2)
sage: L[3]
O(11^0)
```

padic_regulator(*p*, *prec=20*, *height=None*, *check_hypotheses=True*)

Compute the cyclotomic p-adic regulator of this curve. The model of the curve needs to be integral and minimal at p. Moreover the reduction at p should not be additive.

INPUT:

- `p` – prime >= 5
- `prec` – answer will be returned modulo p^{prec}
- `height` – precomputed height function. If not supplied, this function will call padic_height to compute it.
- `check_hypotheses` – boolean, whether to check that this is a curve for which the p-adic height makes sense

OUTPUT: The p-adic cyclotomic regulator of this curve, to the requested precision.

If the rank is 0, we output 1.

AUTHORS:

- Liang Xiao: original implementation at the 2006 MSRI graduate workshop on modular forms
- David Harvey (2006-09-13): cleaned up and integrated into Sage, removed some redundant height computations
- Chris Wuthrich (2007-05-22): added multiplicative and supersingular cases
- David Harvey (2007-09-20): fixed some precision loss that was occurring

EXAMPLES:

```
sage: E = EllipticCurve("37a")
sage: E.padic_regulator(5, 10)
5 + 5^2 + 5^3 + 3*5^6 + 4*5^7 + 5^9 + O(5^10)
```

An anomalous case:

```
sage: E.padic_regulator(53, 10)
26*53^-1 + 30 + 20*53 + 47*53^2 + 10*53^3 + 32*53^4 + 9*53^5 + 22*53^6 +␣
↪35*53^7 + 30*53^8 + O(53^9)
```

An anomalous case where the precision drops some:

```
sage: E = EllipticCurve("5077a")
sage: E.padic_regulator(5, 10)
5 + 5^2 + 4*5^3 + 2*5^4 + 2*5^5 + 2*5^6 + 4*5^7 + 2*5^8 + 5^9 + O(5^10)
```

Check that answers agree over a range of precisions:

```
sage: max_prec = 30    # make sure we get past p^2    # long time
sage: full = E.padic_regulator(5, max_prec)           # long time
sage: for prec in range(1, max_prec):                 # long time
....:     assert E.padic_regulator(5, prec) == full
```

A case where the generator belongs to the formal group already (Issue #3632):

```
sage: E = EllipticCurve([37,0])
sage: E.padic_regulator(5,10)
2*5^2 + 2*5^3 + 5^4 + 5^5 + 4*5^6 + 3*5^8 + 4*5^9 + O(5^10)
```

The result is not dependent on the model for the curve:

```
sage: E = EllipticCurve([0,0,0,0,2^12*17])
sage: Em = E.minimal_model()
sage: E.padic_regulator(7) == Em.padic_regulator(7)
True
```

Allow a python int as input:

```
sage: E = EllipticCurve('37a')
sage: E.padic_regulator(int(5))
5 + 5^2 + 5^3 + 3*5^6 + 4*5^7 + 5^9 + 5^10 + 3*5^11 + 3*5^12 + 5^13 + 4*5^14␣
↪+ 5^15 + 2*5^16 + 5^17 + 2*5^18 + 4*5^19 + O(5^20)
```

padic_sigma (*p*, *N=20*, *E2=None*, *check=False*, *check_hypotheses=True*)

Computes the p-adic sigma function with respect to the standard invariant differential $dx/(2y + a_1x + a_3)$, as defined by Mazur and Tate in [MT1991], as a power series in the usual uniformiser t at the origin.

The equation of the curve must be minimal at p.

INPUT:

- `p` – prime >= 5 for which the curve has good ordinary reduction
- `N` – integer >= 1 (default 20), indicates precision of result; see OUTPUT section for description
- `E2` – precomputed value of E2. If not supplied, this function will call padic_E2 to compute it. The value supplied must be correct mod p^{N-2}.
- `check` – boolean, whether to perform a consistency check (i.e. verify that the computed sigma satisfies the defining
- `differential equation` – note that this does NOT guarantee correctness of all the returned digits, but it comes pretty close.
- `check_hypotheses` – boolean, whether to check that this is a curve for which the p-adic sigma function makes sense

OUTPUT: A power series $t + \cdots$ with coefficients in $\mathbf{Z}_p$.

The output series will be truncated at $O(t^{N+1})$, and the coefficient of t^n for $n \geq 1$ will be correct to precision $O(p^{N-n+1})$.

In practice this means the following. If $t_0 = p^k u$, where u is a p-adic unit with at least N digits of precision, and $k \geq 1$, then the returned series may be used to compute $\sigma(t_0)$ correctly modulo p^{N+k} (i.e. with N correct p-adic digits).

ALGORITHM: Described in "Efficient Computation of p-adic Heights" (David Harvey) [Har2009] which is basically an optimised version of the algorithm from "p-adic Heights and Log Convergence" (Mazur, Stein, Tate) [MST2006].

Running time is soft-$O(N^2 \log p)$, plus whatever time is necessary to compute E_2.

AUTHORS:

- David Harvey (2006-09-12)
- David Harvey (2007-02): rewrote

EXAMPLES:

```
sage: EllipticCurve([-1, 1/4]).padic_sigma(5, 10)
O(5^11) + (1 + O(5^10))*t + O(5^9)*t^2 + (3 + 2*5^2 + 3*5^3 + 3*5^6 + 4*5^7 +␣
↪O(5^8))*t^3 + O(5^7)*t^4 + (2 + 4*5^2 + 4*5^3 + 5^4 + 5^5 + O(5^6))*t^5 +␣
↪O(5^5)*t^6 + (2 + 2*5 + 5^2 + 4*5^3 + O(5^4))*t^7 + O(5^3)*t^8 + (1 + 2*5 +␣
↪O(5^2))*t^9 + O(5)*t^10 + O(t^11)
```

Run it with a consistency check:

```
sage: EllipticCurve("37a").padic_sigma(5, 10, check=True)
O(5^11) + (1 + O(5^10))*t + O(5^9)*t^2 + (3 + 2*5^2 + 3*5^3 + 3*5^6 + 4*5^7 +␣
↪O(5^8))*t^3 + (3 + 2*5 + 2*5^2 + 2*5^3 + 2*5^4 + 2*5^5 + 2*5^6 + O(5^7))*t^
↪4 + (2 + 4*5^2 + 4*5^3 + 5^4 + 5^5 + O(5^6))*t^5 + (2 + 3*5 + 5^4 + O(5^
↪5))*t^6 + (4 + 3*5 + 2*5^2 + O(5^4))*t^7 + (2 + 3*5 + 2*5^2 + O(5^3))*t^8 +␣
↪(4*5 + O(5^2))*t^9 + (1 + O(5))*t^10 + O(t^11)
```

Boundary cases:

```
sage: EllipticCurve([1, 1, 1, 1, 1]).padic_sigma(5, 1)
 (1 + O(5))*t + O(t^2)
sage: EllipticCurve([1, 1, 1, 1, 1]).padic_sigma(5, 2)
 (1 + O(5^2))*t + (3 + O(5))*t^2 + O(t^3)
```

Supply your very own value of E2:

```
sage: X = EllipticCurve("37a")
sage: my_E2 = X.padic_E2(5, 8)
sage: my_E2 = my_E2 + 5**5    # oops!!!
sage: X.padic_sigma(5, 10, E2=my_E2)
O(5^11) + (1 + O(5^10))*t + O(5^9)*t^2 + (3 + 2*5^2 + 3*5^3 + 4*5^5 + 2*5^6 +␣
↪3*5^7 + O(5^8))*t^3 + (3 + 2*5 + 2*5^2 + 2*5^3 + 2*5^4 + 2*5^5 + 2*5^6 +␣
↪O(5^7))*t^4 + (2 + 4*5^2 + 4*5^3 + 5^4 + 3*5^5 + O(5^6))*t^5 + (2 + 3*5 + 5^
↪4 + O(5^5))*t^6 + (4 + 3*5 + 2*5^2 + O(5^4))*t^7 + (2 + 3*5 + 2*5^2 + O(5^
↪3))*t^8 + (4*5 + O(5^2))*t^9 + (1 + O(5))*t^10 + O(t^11)
```

Check that sigma is "weight 1".

```
sage: f = EllipticCurve([-1, 3]).padic_sigma(5, 10)
sage: g = EllipticCurve([-1*(2**4), 3*(2**6)]).padic_sigma(5, 10)
sage: t = f.parent().gen()
sage: f(2*t)/2
(1 + O(5^10))*t + (4 + 3*5 + 3*5^2 + 3*5^3 + 4*5^4 + 4*5^5 + 3*5^6 + 5^7 +␣
↪O(5^8))*t^3 + (3 + 3*5^2 + 5^4 + 2*5^5 + O(5^6))*t^5 + (4 + 5 + 3*5^3 + O(5^
```

(continues on next page)

(continued from previous page)

```
↪4))*t^7 + (4 + 2*5 + O(5^2))*t^9 + O(5)*t^10 + O(t^11)
sage: g
O(5^11) + (1 + O(5^10))*t + O(5^9)*t^2 + (4 + 3*5 + 3*5^2 + 3*5^3 + 4*5^4 +␣
↪4*5^5 + 3*5^6 + 5^7 + O(5^8))*t^3 + O(5^7)*t^4 + (3 + 3*5^2 + 5^4 + 2*5^5 +␣
↪O(5^6))*t^5 + O(5^5)*t^6 + (4 + 5 + 3*5^3 + O(5^4))*t^7 + O(5^3)*t^8 + (4 +␣
↪2*5 + O(5^2))*t^9 + O(5)*t^10 + O(t^11)
sage: f(2*t)/2 -g
O(t^11)
```

Test that it returns consistent results over a range of precision:

```
sage: # long time
sage: max_N = 30   # get up to at least p^2
sage: E = EllipticCurve([1, 1, 1, 1, 1])
sage: p = 5
sage: E2 = E.padic_E2(5, max_N)
sage: max_sigma = E.padic_sigma(p, max_N, E2=E2)
sage: for N in range(3, max_N):
....:     sigma = E.padic_sigma(p, N, E2=E2)
....:     assert sigma == max_sigma
```

padic_sigma_truncated(*p*, *N=20*, *lamb=0*, *E2=None*, *check_hypotheses=True*)

Compute the p-adic sigma function with respect to the standard invariant differential $dx/(2y + a_1x + a_3)$, as defined by Mazur and Tate in [MT1991], as a power series in the usual uniformiser t at the origin.

The equation of the curve must be minimal at p.

This function differs from *padic_sigma()* in the precision profile of the returned power series; see OUTPUT below.

INPUT:

- `p` – prime >= 5 for which the curve has good ordinary reduction
- `N` – integer >= 2 (default 20), indicates precision of result; see OUTPUT section for description
- `lamb` – integer >= 0, see OUTPUT section for description
- `E2` – precomputed value of E2. If not supplied, this function will call padic_E2 to compute it. The value supplied must be correct mod p^{N-2}.
- `check_hypotheses` – boolean, whether to check that this is a curve for which the p-adic sigma function makes sense

OUTPUT: A power series $t + \cdots$ with coefficients in $\mathbf{Z}_p$.

The coefficient of t^j for $j \geq 1$ will be correct to precision $O(p^{N-2+(3-j)(lamb+1)})$.

ALGORITHM: Described in "Efficient Computation of p-adic Heights" [Har2009], which is basically an optimised version of the algorithm from "p-adic Heights and Log Convergence" (Mazur, Stein, Tate) [MST2006].

Running time is soft-$O(N^2\lambda^{-1}\log p)$, plus whatever time is necessary to compute E_2.

AUTHORS:

- David Harvey (2008-01): wrote based on previous *padic_sigma()* function

EXAMPLES:

```
sage: E = EllipticCurve([-1, 1/4])
sage: E.padic_sigma_truncated(5, 10)
```

(continues on next page)

(continued from previous page)

```
O(5^11) + (1 + O(5^10))*t + O(5^9)*t^2 + (3 + 2*5^2 + 3*5^3 + 3*5^6 + 4*5^7 +␣
↪O(5^8))*t^3 + O(5^7)*t^4 + (2 + 4*5^2 + 4*5^3 + 5^4 + 5^5 + O(5^6))*t^5 +␣
↪O(5^5)*t^6 + (2 + 2*5 + 5^2 + 4*5^3 + O(5^4))*t^7 + O(5^3)*t^8 + (1 + 2*5 +␣
↪O(5^2))*t^9 + O(5)*t^10 + O(t^11)
```

Note the precision of the t^3 coefficient depends only on N, not on lamb:

```
sage: E.padic_sigma_truncated(5, 10, lamb=2)
O(5^17) + (1 + O(5^14))*t + O(5^11)*t^2 + (3 + 2*5^2 + 3*5^3 + 3*5^6 + 4*5^7␣
↪+ O(5^8))*t^3 + O(5^5)*t^4 + (2 + O(5^2))*t^5 + O(t^6)
```

Compare against plain padic_sigma() function over a dense range of N and lamb

```
sage: E = EllipticCurve([1, 2, 3, 4, 7])                                    # long␣
↪time
sage: E2 = E.padic_E2(5, 50)                                                # long␣
↪time
sage: for N in range(2, 10):                                                # long␣
↪time
....:     for lamb in range(10):
....:         correct = E.padic_sigma(5, N + 3*lamb, E2=E2)
....:         compare = E.padic_sigma_truncated(5, N=N, lamb=lamb, E2=E2)
....:         assert compare == correct
```

pari_curve()

Return the PARI curve corresponding to this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([0, 0, 1, -1, 0])
sage: e = E.pari_curve()
sage: type(e)
<... 'cypari2.gen.Gen'>
sage: e.type()
't_VEC'
sage: e.ellan(10)
[1, -2, -3, 2, -2, 6, -1, 0, 6, 4]
```

```
sage: E = EllipticCurve(RationalField(), ['1/3', '2/3'])
sage: e = E.pari_curve()
sage: e[:5]
[0, 0, 0, 1/3, 2/3]
```

When doing certain computations, PARI caches the results:

```
sage: E = EllipticCurve('37a1')
sage: _ = E.__dict__.pop('_pari_curve', None)  # clear cached data
sage: Epari = E.pari_curve()
sage: Epari
[0, 0, 1, -1, 0, 0, -2, 1, -1, 48, -216, 37, 110592/37, Vecsmall([1]),␣
↪[Vecsmall([64, 1])], [0, 0, 0, 0, 0, 0, 0, 0]]
sage: Epari.omega()
[2.99345864623196, -2.45138938198679*I]
sage: Epari
[0, 0, 1, -1, 0, 0, -2, 1, -1, 48, -216, 37, 110592/37, Vecsmall([1]),␣
↪[Vecsmall([64, 1])], [[2.99345864623196, -2.45138938198679*I], 0, [0.
```

(continues on next page)

(continued from previous page)

```
↪837565435283323, 0.269594436405445, -1.10715987168877, 1.37675430809421, 1.
↪94472530697209, 0.567970998877878]~, 0, 0, 0, 0, 0]]
```

This shows that the bug uncovered by Issue #4715 is fixed:

```
sage: Ep = EllipticCurve('903b3').pari_curve()
```

This still works, even when the curve coefficients are large (see Issue #13163):

```
sage: E = EllipticCurve([43826964575647946916034423387881064 97, 28, 3992,␣
↪16777216, 298])
sage: E.pari_curve()
[43826964575647946916034423387881064 97, 28, 3992, 16777216, 298, ...]
sage: E.minimal_model()
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 -␣
↪76864239340837973906759811692291719076741835883261845113911467271436724231670914843924979 8
↪+␣
↪82022804435537614837731086487342718512159885048202147848997526621004596630117099924468609 7
↪over Rational Field
```

pari_mincurve()

Return the PARI curve corresponding to a minimal model for this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve(RationalField(), ['1/3', '2/3'])
sage: e = E.pari_mincurve()
sage: e[:5]
[0, 0, 0, 27, 486]
sage: E.conductor()
47232
sage: e.ellglobalred()
[47232, [1, 0, 0, 0], 2, [2, 7; 3, 2; 41, 1], [[7, 2, 0, 1], [2, -3, 0, 2],␣
↪[1, 5, 0, 1]]]
```

period_lattice(*embedding=None*)

Return the period lattice of the elliptic curve with respect to the differential $dx/(2y + a_1x + a_3)$.

INPUT:

- `embedding` – ignored (for compatibility with the period_lattice function for elliptic_curve_number_field)

OUTPUT:

(period lattice) The *PeriodLattice_ell* object associated to this elliptic curve (with respect to the natural embedding of **Q** into **R**).

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.period_lattice()
Period lattice associated to
 Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
```

point_search(*height_limit*, *verbose=False*, *rank_bound=None*)

Search for points on a curve up to an input bound on the naive logarithmic height.

INPUT:

- `height_limit` – float; bound on naive height
- `verbose` – boolean (default: `False`); if `True`, report on the saturation process otherwise just return the result
- `rank_bound` – boolean (optional); if provided, stop saturating once we find this many independent nontorsion points

OUTPUT: points (list) - list of independent points which generate the subgroup of the Mordell-Weil group generated by the points found and then saturated.

Warning: height_limit is logarithmic, so increasing by 1 will cause the running time to increase by a factor of approximately 4.5 (=exp(1.5)).

IMPLEMENTATION: Uses Michael Stoll's ratpoints module in PARI/GP.

EXAMPLES:

```
sage: E = EllipticCurve('389a1')
sage: E.point_search(1, verbose=False)
[(-1 : 1 : 1), (0 : 0 : 1)]
```

Increasing the height_limit takes longer, but finds no more points:

```
sage: E.point_search(10, verbose=False)  # long time
[(-1 : 1 : 1), (0 : 0 : 1)]
```

In fact this curve has rank 2 so no more than 2 points will ever be output, but we are not using this fact.

```
sage: E.saturation(_)
([(-1 : 1 : 1), (0 : 0 : 1)], 1, 0.152460177943144)
```

What this shows is that if the rank is 2 then the points listed do generate the Mordell-Weil group (mod torsion). Finally,

```
sage: E.rank()
2
```

If we only need one independent generator:

```
sage: E.point_search(5, verbose=False, rank_bound=1)
[(-2 : 0 : 1)]
```

pollack_stevens_modular_symbol (*sign=0*, *implementation='eclib'*)

Create the modular symbol attached to the elliptic curve, suitable for overconvergent calculations.

INPUT:

- `sign` – +1 or -1 or 0 (default), in which case this it is the sum of the two
- `implementation` – either 'eclib' (default) or 'sage'. This determines classical modular symbols which implementation of the underlying classical modular symbols is used

EXAMPLES:

```
sage: E = EllipticCurve('113a1')
sage: symb = E.pollack_stevens_modular_symbol()
sage: symb
```

(continues on next page)

(continued from previous page)

```
Modular symbol of level 113 with values in Sym^0 Q^2
sage: symb.values()
[-1/2, 1, -1, 0, 0, 1, 1, -1, 0, -1, 0, 0, 0, 1, -1, 0, 0, 0, 1, 0, 0]

sage: E = EllipticCurve([0,1])
sage: symb = E.pollack_stevens_modular_symbol(+1)
sage: symb.values()
[-1/6, 1/12, 0, 1/6, 1/12, 1/3, -1/12, 0, -1/6, -1/12, -1/4, -1/6, 1/12]
```

prove_BSD(*E*, *verbosity=0*, *two_desc='mwrank'*, *proof=None*, *secs_hi=5*, *return_BSD=False*)

Attempt to prove the Birch and Swinnerton-Dyer conjectural formula for E, returning a list of primes p for which this function fails to prove BSD(E,p).

Here, BSD(E,p) is the statement: "the Birch and Swinnerton-Dyer formula holds up to a rational number coprime to p."

INPUT:

- `E` – an elliptic curve
- `verbosity` – int, how much information about the proof to print.
  - 0: print nothing
  - 1: print sketch of proof
  - 2: print information about remaining primes
- `two_desc` – string (default `'mwrank'`), what to use for the two-descent. Options are `'mwrank'`, `'pari'`, `'sage'`
- `proof` – bool or None (default: None, see proof.elliptic_curve or sage.structure.proof). If False, this function just immediately returns the empty list.
- `secs_hi` – maximum number of seconds to try to compute the Heegner index before switching over to trying to compute the Heegner index bound. (Rank 0 only!)
- `return_BSD` – bool (default: `False`) whether to return an object which contains information to reconstruct a proof

---

Note: When printing verbose output, phrases such as "by Mazur" are referring to the following list of papers:

REFERENCES:

- [Cha2005]
- [Jet2008]
- [Kat2004]
- [Kol1991]
- [LW2015]
- [LS]
- [Maz1978]
- [Rub1991]
- [SW2013]
- [GJPST2009]

EXAMPLES:

```
sage: EllipticCurve('11a').prove_BSD(verbosity=2)
p = 2: True by 2-descent
True for p not in {2, 5} by Kolyvagin.
Kolyvagin's bound for p = 5 applies by Lawson-Wuthrich
True for p = 5 by Kolyvagin bound
[]

sage: EllipticCurve('14a').prove_BSD(verbosity=2)
p = 2: True by 2-descent
True for p not in {2, 3} by Kolyvagin.
Kolyvagin's bound for p = 3 applies by Lawson-Wuthrich
True for p = 3 by Kolyvagin bound
[]

sage: E = EllipticCurve("20a1")
sage: E.prove_BSD(verbosity=2)
p = 2: True by 2-descent
True for p not in {2, 3} by Kolyvagin.
Kato further implies that #Sha[3] is trivial.
[]

sage: E = EllipticCurve("50b1")
sage: E.prove_BSD(verbosity=2)
p = 2: True by 2-descent
True for p not in {2, 3, 5} by Kolyvagin.
Kolyvagin's bound for p = 3 applies by Lawson-Wuthrich
Kolyvagin's bound for p = 5 applies by Lawson-Wuthrich
True for p = 3 by Kolyvagin bound
True for p = 5 by Kolyvagin bound
[]
sage: E.prove_BSD(two_desc='pari')
[]
```

A rank two curve:

```
sage: E = EllipticCurve('389a')
```

We know nothing with proof=True:

```
sage: E.prove_BSD()
Set of all prime numbers: 2, 3, 5, 7, ...
```

We (think we) know everything with proof=False:

```
sage: E.prove_BSD(proof=False)
[]
```

A curve of rank 0 and prime conductor:

```
sage: E = EllipticCurve('19a')
sage: E.prove_BSD(verbosity=2)
p = 2: True by 2-descent
True for p not in {2, 3} by Kolyvagin.
Kolyvagin's bound for p = 3 applies by Lawson-Wuthrich
True for p = 3 by Kolyvagin bound
[]
```

(continues on next page)

(continued from previous page)

```
sage: E = EllipticCurve('37a')
sage: E.rank()
1
sage: E._EllipticCurve_rational_field__rank
(1, True)
sage: E.analytic_rank = lambda : 0
sage: E.prove_BSD()
Traceback (most recent call last):
...
RuntimeError: It seems that the rank conjecture does not hold for this curve
(Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field)!
This may be a counterexample to BSD, but is more likely a bug.
```

We test the consistency check for the 2-part of Sha:

```
sage: E = EllipticCurve('37a')
sage: S = E.sha(); S
Tate-Shafarevich group for the Elliptic Curve defined by y^2 + y = x^3 - x
 over Rational Field
sage: def foo(use_database):
....:  return 4
sage: S.an = foo
sage: E.prove_BSD()
Traceback (most recent call last):
...
RuntimeError: Apparent contradiction: 0 <= rank(sha[2]) <= 0, but ord_2(sha_
↪an) = 2
```

An example with a Tamagawa number at 5:

```
sage: E = EllipticCurve('123a1')
sage: E.prove_BSD(verbosity=2)
p = 2: True by 2-descent
True for p not in {2, 5} by Kolyvagin.
Kolyvagin's bound for p = 5 applies by Lawson-Wuthrich
True for p = 5 by Kolyvagin bound
[]
```

A curve for which 3 divides the order of the Tate-Shafarevich group:

```
sage: E = EllipticCurve('681b')
sage: E.prove_BSD(verbosity=2)                # long time
p = 2: True by 2-descent...
True for p not in {2, 3} by Kolyvagin....
Remaining primes:
p = 3: irreducible, surjective, non-split multiplicative
    (0 <= ord_p <= 2)
    ord_p(#Sha_an) = 2
[3]
```

A curve for which we need to use `heegner_index_bound`:

```
sage: E = EllipticCurve('198b')
sage: E.prove_BSD(verbosity=1, secs_hi=1)
p = 2: True by 2-descent
```

(continues on next page)

(continued from previous page)

```
True for p not in {2, 3} by Kolyvagin.
[3]
```

The `return_BSD` option gives an object with detailed information about the proof:

```
sage: E = EllipticCurve('26b')
sage: B = E.prove_BSD(return_BSD=True)
sage: B.two_tor_rk
0
sage: B.N
26
sage: B.gens
[]
sage: B.primes
[]
sage: B.heegner_indexes
{-23: 2}
```

q_eigenform(*prec*)

Synonym for `self.q_expansion(prec)`.

EXAMPLES:

```
sage: E = EllipticCurve('37a1')
sage: E.q_eigenform(10)
q - 2*q^2 - 3*q^3 + 2*q^4 - 2*q^5 + 6*q^6 - q^7 + 6*q^9 + O(q^10)
sage: E.q_eigenform(10) == E.q_expansion(10)
True
```

q_expansion(*prec*)

Return the q-expansion to precision prec of the newform attached to this elliptic curve.

INPUT:

- `prec` – an integer

OUTPUT:

a power series (in the variable 'q')

Note: If you want the output to be a modular form and not just a q-expansion, use *modular_form()*.

EXAMPLES:

```
sage: E = EllipticCurve('37a1')
sage: E.q_expansion(20)
q - 2*q^2 - 3*q^3 + 2*q^4 - 2*q^5 + 6*q^6 - q^7 + 6*q^9 + 4*q^10
 - 5*q^11 - 6*q^12 - 2*q^13 + 2*q^14 + 6*q^15 - 4*q^16 - 12*q^18 + O(q^20)
```

quadratic_twist(*D*)

Return the global minimal model of the quadratic twist of this curve by `D`.

EXAMPLES:

```
sage: E = EllipticCurve('37a1')
sage: E7 = E.quadratic_twist(7); E7
```

(continues on next page)

(continued from previous page)

```
Elliptic Curve defined by y^2 = x^3 - 784*x + 5488 over Rational Field
sage: E7.conductor()
29008
sage: E7.quadratic_twist(7) == E
True
```

rank (*use_database=True*, *verbose=False*, *only_use_mwrank=True*, *algorithm='mwrank_lib'*, *proof=None*, *pari_effort=0*)

Return the rank of this elliptic curve, assuming no conjectures.

If we fail to provably compute the rank, raises a RuntimeError exception.

INPUT:

- `use_database` – boolean (default: `True`); if `True`, try to look up the rank in the Cremona database
- `verbose` – (default: `False`) if specified changes the verbosity of mwrank computations
- `algorithm` – (default: `'mwrank_lib'`) one of:
  - `'mwrank_shell'` – call mwrank shell command
  - `'mwrank_lib'` – call mwrank c library
  - `'pari'` – call ellrank in pari
- `only_use_mwrank` – (default: `True`) if `False` try using analytic rank methods first
- `proof` – bool (default: `None`, see `proof.elliptic_curve` or `sage.structure.proof`); note that results obtained from databases are considered `proof=True`
- `pari_effort` – (default: 0) parameter used in when the algorithm `pari` is chosen. It measure of the effort done to find rational points. Values up to 10 can be chosen; the running times increase roughly like the cube of the effort value.

OUTPUT: the rank of the elliptic curve as `Integer`

IMPLEMENTATION: Uses L-functions, mwrank, pari, and databases.

EXAMPLES:

```
sage: EllipticCurve('11a').rank()
0
sage: EllipticCurve('37a').rank()
1
sage: EllipticCurve('389a').rank()
2
sage: EllipticCurve('5077a').rank()
3
sage: EllipticCurve([1, -1, 0, -79, 289]).rank()    # This will use the␣
↪default proof behavior of True
4
sage: EllipticCurve([0, 0, 1, -79, 342]).rank(proof=False)
5
sage: EllipticCurve([0, 0, 1, -79, 342]).rank(algorithm="pari")
5
```

Examples with denominators in defining equations:

```
sage: E = EllipticCurve([0, 0, 0, 0, -675/4])
sage: E.rank()
0
sage: E = EllipticCurve([0, 0, 1/2, 0, -1/5])
sage: E.rank()
1
sage: E.minimal_model().rank()
1
```

A large example where mwrank doesn't determine the result with certainty, but pari does:

```
sage: EllipticCurve([1,0,0,0,37455]).rank(proof=False)
0
sage: EllipticCurve([1,0,0,0,37455]).rank(proof=True)
Traceback (most recent call last):
...
RuntimeError: rank not provably correct (lower bound: 0)
sage: EllipticCurve([1,0,0,0,37455]).rank(algorithm="pari")
0
```

rank_bound(*algorithm='pari'*)

Return the upper bound on the rank of the curve, computed using a 2-descent.

INPUT:

- `algorithm` – (default: `'pari'`) either `'pari'` or `'mwrank'`

In many cases, this is the actual rank of the curve.

EXAMPLES:

```
sage: E = EllipticCurve("389a1")
sage: E.rank_bound()
2
```

The following is the curve 571a1, which has rank 0, but Sha of order 4, yet pari, using the Cassels pairing is able to show that the rank is 0. The 2-descent in mwrank only determines a weaker upper bound:

```
sage: E = EllipticCurve([0, -1, 1, -929, -10595])
sage: E.rank_bound()
0
sage: E.rank_bound(algorithm="mwrank")
2
```

In the following last example, both algorithm only determine a rank bound larger than the actual rank:

```
sage: E = EllipticCurve([1, 1, 1, -896670, -327184905])
sage: E.rank_bound()
2
sage: E.rank_bound(algorithm="mwrank")
2
sage: E.rank(only_use_mwrank=False) # uses L-function
0
```

real_components()

Return the number of real components.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.real_components ()
2
sage: E = EllipticCurve('37b')
sage: E.real_components ()
2
sage: E = EllipticCurve('11a')
sage: E.real_components ()
1
```

reduction(*p*)

Return the reduction of the elliptic curve at a prime of good reduction.

Note: The actual reduction is done in `self.change_ring(GF(p))`; the reduction is performed after changing to a model which is minimal at p.

INPUT:

- `p` – a (positive) prime number

OUTPUT: an elliptic curve over the finite field $\mathbf{F}_p$

EXAMPLES:

```
sage: E = EllipticCurve('389a1')
sage: E.reduction(2)
Elliptic Curve defined by y^2 + y = x^3 + x^2 over Finite Field of size 2
sage: E.reduction(3)
Elliptic Curve defined by y^2 + y = x^3 + x^2 + x over Finite Field of size 3
sage: E.reduction(5)
Elliptic Curve defined by y^2 + y = x^3 + x^2 + 3*x over Finite Field of size␣
↪5
sage: E.reduction(38)
Traceback (most recent call last):
...
AttributeError: p must be prime.
sage: E.reduction(389)
Traceback (most recent call last):
...
AttributeError: The curve must have good reduction at p.
sage: E = EllipticCurve([5^4, 5^6])
sage: E.reduction(5)
Elliptic Curve defined by y^2 = x^3 + x + 1 over Finite Field of size 5
```

regulator(*proof=None*, *precision=53*, ***kwds*)

Return the regulator of this curve, which must be defined over $\mathbf{Q}$.

INPUT:

- `proof` – bool or `None` (default: `None`, see proof.[tab] or sage.structure.proof). Note that results from databases are considered proof = True
- `precision` – (int, default 53): the precision in bits of the result
- `**kwds` – passed to *`gens()`* method

EXAMPLES:

```
sage: E = EllipticCurve([0, 0, 1, -1, 0])
sage: E.regulator()
0.0511114082399688
sage: EllipticCurve('11a').regulator()
1.00000000000000
sage: EllipticCurve('37a').regulator()
0.0511114082399688
sage: EllipticCurve('389a').regulator()
0.152460177943144
sage: EllipticCurve('5077a').regulator()
0.41714355875838...
sage: EllipticCurve([1, -1, 0, -79, 289]).regulator()
1.50434488827528
sage: EllipticCurve([0, 0, 1, -79, 342]).regulator(proof=False)  # long time␣
↪(6s on sage.math, 2011)
14.790527570131...
```

root_number(*p=None*)

Return the root number of this elliptic curve.

This is 1 if the order of vanishing of the L-function $L(E, s)$ at 1 is even, and -1 if it is odd.

INPUT:

- *p* – (optional) if given, return the local root number at `p`

EXAMPLES:

```
sage: EllipticCurve('11a1').root_number()
1
sage: EllipticCurve('37a1').root_number()
-1
sage: EllipticCurve('389a1').root_number()
1
sage: type(EllipticCurve('389a1').root_number())
<... 'sage.rings.integer.Integer'>

sage: E = EllipticCurve('100a1')
sage: E.root_number(2)
-1
sage: E.root_number(5)
1
sage: E.root_number(7)
1
```

The root number is cached:

```
sage: E.root_number(2) is E.root_number(2)
True
sage: E.root_number()
1
```

satisfies_heegner_hypothesis(*D*)

Returns `True` precisely when D is a fundamental discriminant that satisfies the Heegner hypothesis for this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: E.satisfies_heegner_hypothesis(-7)
True
sage: E.satisfies_heegner_hypothesis(-11)
False
```

saturation (*points*, *verbose=False*, *max_prime=-1*, *min_prime=2*)

Given a list of rational points on E, compute the saturation in $E(Q)$ of the subgroup they generate.

INPUT:

- `points (list)` – list of points on E
- `verbose (bool)` – (default: `False`) if `True`, give verbose output
- `max_prime` – int (default: -1); if -1 (the default), an upper bound is computed for the primes at which the subgroup may not be saturated, and saturation is performed for all primes up to this bound; otherwise, the bound used is the minimum of `max_prime` and the computed bound
- **`min_prime (int)` – (default: 2) only do p-saturation**
  at primes p greater than or equal to this

Note: To saturate at a single prime p, set `max_prime` and `min_prime` both to p. One situation where this is useful is after mapping saturated points from another elliptic curve by a p-isogeny, since the images may not be p-saturated but will be saturated at all other primes.

OUTPUT:

- `saturation (list)` – points that form a basis for the saturation
- `index (int)` – the index of the group generated by points in their saturation
- `regulator (real with default precision)` – regulator of saturated points.

ALGORITHM:

Uses Cremona's `eclib` package, which computes a bound on the saturation index. To p-saturate, or prove p-saturation, we consider the reductions of the points modulo primes q of good reduction such that $E(\mathbf{F}_q)$ has order divisible by p.

Note: In versons of `eclib` up to `v20190909`, division of points in `eclib` was done using floating point methods, without automatic handling of precision, so that p-saturation sometimes failed unless `mwrank_set_precision()` was called in advance with a suitably high bit precision. Since version `v20210310` of `eclib`, division is done using exact methods based on division polynomials, and p-saturation cannot fail in this way.

Note: The computed index of saturation may be large, in which case saturation may take a long time. For example, the rank 4 curve `EllipticCurve([0,1,1,-9872,374262])` has a saturation index bound of 11816 and takes around 40 seconds to prove saturation.

EXAMPLES:

```
sage: E = EllipticCurve('37a1')
sage: P=E(0,0)
sage: Q=5*P; Q
```

(continues on next page)

(continued from previous page)

```
(1/4 : -5/8 : 1)
sage: E.saturation([Q])
([(0 : 0 : 1)], 5, 0.0511114082399688)
```

selmer_rank(*algorithm='pari'*)

Return the rank of the 2-Selmer group of the curve.

INPUT:

- `algorithm` – (default: `'pari'`) either `'pari'` or `'mwrank'`

EXAMPLES: This example has rank 1, Sha[2] of order 4 and a single rational 2-torsion point:

```
sage: E = EllipticCurve([1, 1, 1, 508, -2551])
sage: E.selmer_rank()
4
sage: E.selmer_rank(algorithm="mwrank")
4
```

The following is the curve 960d1, which has rank 0, but Sha of order 4:

```
sage: E = EllipticCurve([0, -1, 0, -900, -10098])
sage: E.selmer_rank()
3
sage: E.selmer_rank(algorithm="mwrank")
3
```

This curve has rank 1, and 4 elements in Sha[2]. Yet the order of Sha is 16, so that group is the product of two cyclic groups of order 4:

```
sage: E = EllipticCurve([1, 0, 0, -150752, -22541610])
sage: E.selmer_rank()
4
```

Instead in this last example of rank 0, Sha is a product of four cyclic groups of order 2:

```
sage: E = EllipticCurve([1, 0, 0, -49280, -4214808])
sage: E.selmer_rank()
5
sage: E.rank()
0
```

sha()

Return an object of class 'sage.schemes.elliptic_curves.sha_tate.Sha' attached to this elliptic curve.

This can be used in functions related to bounding the order of Sha (The Tate-Shafarevich group of the curve).

EXAMPLES:

```
sage: E = EllipticCurve('37a1')
sage: S = E.sha()
sage: S
Tate-Shafarevich group for the Elliptic Curve
 defined by y^2 + y = x^3 - x over Rational Field
sage: S.bound_kolyvagin()
([2], 1)
```

silverman_height_bound(*algorithm='default'*)

Return the Silverman height bound.

This is a positive real (floating point) number B such that for all points P on the curve over any number field, $|h(P) - \hat{h}(P)| \leq B$, where $h(P)$ is the naive logarithmic height of P and $\hat{h}(P)$ is the canonical height.

INPUT:

- `algorithm` – one of the following:
  - `'default'` (default) - compute using a Python implementation in Sage
  - `'mwrank'` – use a C++ implementation in the mwrank library

Note:

- The CPS_height_bound is often better (i.e. smaller) than the Silverman bound, but it only applies for points over the base field, whereas the Silverman bound works over all number fields.
- The Silverman bound is also fairly straightforward to compute over number fields, but isn't implemented here.
- Silverman's paper is 'The Difference Between the Weil Height and the Canonical Height on Elliptic Curves', Math. Comp., Volume 55, Number 192, pages 723-743. We use a correction by Bremner with 0.973 replaced by 0.961, as explained in the source code to mwrank (htconst.cc).

EXAMPLES:

```
sage: E = EllipticCurve('37a1')
sage: E.silverman_height_bound()
4.825400758180918
sage: E.silverman_height_bound(algorithm='mwrank')
4.825400758180918
sage: E.CPS_height_bound()
0.16397076103046915
```

simon_two_descent(*verbose=0*, *lim1=5*, *lim3=50*, *limtriv=3*, *maxprob=20*, *limbigprime=30*, *known_points=None*)

Return lower and upper bounds on the rank of the Mordell-Weil group $E(\mathbf{Q})$ and a list of points of infinite order.

Warning: This function is deprecated as the functionality of Simon's script for elliptic curves over the rationals has been ported over to pari. Use *rank()* with the keyword `algorithm='pari'` instead.

INPUT:

- `verbose` – 0, 1, 2, or 3 (default: 0), the verbosity level
- `lim1` – (default: 5) limit on trivial points on quartics
- `lim3` – (default: 50) limit on points on ELS quartics
- `limtriv` – (default: 3) limit on trivial points on E
- `maxprob` – (default: 20)
- `limbigprime` – (default: 30) to distinguish between small and large prime numbers. Use probabilistic tests for large primes. If 0, don't any probabilistic tests.

- known_points – (default: None) list of known points on the curve

OUTPUT: a triple (lower, upper, list) consisting of

- lower (integer) – lower bound on the rank
- upper (integer) – upper bound on the rank
- list – list of points of infinite order in $E(\mathbf{Q})$

The integer upper is in fact an upper bound on the dimension of the 2-Selmer group, hence on the dimension of $E(\mathbf{Q})/2E(\mathbf{Q})$. It is equal to the dimension of the 2-Selmer group except possibly if $E(\mathbf{Q})[2]$ has dimension 1. In that case, upper may exceed the dimension of the 2-Selmer group by an even number, due to the fact that the algorithm does not perform a second descent.

To obtain a list of generators, use E.gens().

IMPLEMENTATION:

Uses Denis Simon's PARI/GP scripts from http://www.math.unicaen.fr/~simon/

EXAMPLES:

We compute the ranks of the curves of lowest known conductor up to rank 8. Amazingly, each of these computations finishes almost instantly!

```
sage: E = EllipticCurve('11a1')
sage: E.simon_two_descent()
doctest:warning
...
DeprecationWarning: Use E.rank(algorithm="pari") instead, as this script has␣
↪been ported over to pari.
See https://github.com/sagemath/sage/issues/35621 for details.
(0, 0, [])
sage: E = EllipticCurve('37a1')
sage: E.simon_two_descent()
(1, 1, [(0 : 0 : 1)])
sage: E = EllipticCurve('389a1')
sage: E._known_points = []  # clear cached points
sage: E.simon_two_descent()
(2, 2, [(5/4 : 5/8 : 1), (-3/4 : 7/8 : 1)])
sage: E = EllipticCurve('5077a1')
sage: E.simon_two_descent()
(3, 3, [(1 : 0 : 1), (2 : 0 : 1), (0 : 2 : 1)])
```

In this example Simon's program does not find any points, though it does correctly compute the rank of the 2-Selmer group.

```
sage: E = EllipticCurve([1, -1, 0, -751055859, -7922219731979])
sage: E.simon_two_descent()
(1, 1, [])
```

The rest of these entries were taken from Tom Womack's page http://tom.womack.net/maths/conductors.htm

```
sage: E = EllipticCurve([1, -1, 0, -79, 289])
sage: E.simon_two_descent()
(4, 4, [(6 : -1 : 1), (4 : 3 : 1), (5 : -2 : 1), (8 : 7 : 1)])
sage: E = EllipticCurve([0, 0, 1, -79, 342])
sage: E.simon_two_descent()  # long time (9s on sage.math, 2011)
(5, 5, [(5 : 8 : 1), (10 : 23 : 1), (3 : 11 : 1), (-3 : 23 : 1), (0 : 18 :␣
↪1)])
```

(continues on next page)

(continued from previous page)

```
sage: E = EllipticCurve([1, 1, 0, -2582, 48720])
sage: r, s, G = E.simon_two_descent(); r,s
(6, 6)
sage: E = EllipticCurve([0, 0, 0, -10012, 346900])
sage: r, s, G = E.simon_two_descent(); r,s  # long time
(7, 7)
sage: E = EllipticCurve([0, 0, 1, -23737, 960366])
sage: r, s, G = E.simon_two_descent(); r,s  # long time
(8, 8)
```

Example from Issue #10832:

```
sage: E = EllipticCurve([1,0,0,-6664,86543])
sage: E.simon_two_descent()
(2, 3, [(-1/4 : 2377/8 : 1), (323/4 : 1891/8 : 1)])
sage: E.rank()
2
sage: E.gens()
[(-1/4 : 2377/8 : 1), (323/4 : 1891/8 : 1)]
```

Example where the lower bound is known to be 1 despite that the algorithm has not found any points of infinite order

```
sage: E = EllipticCurve([1, 1, 0, -23611790086, 1396491910863060])
sage: E.simon_two_descent()
(1, 2, [])
sage: E.rank()
1
sage: E.gens()      # uses mwrank
[(4311692542083/48594841 : -13035144436525227/338754636611 : 1)]
```

Example for Issue #5153:

```
sage: E = EllipticCurve([3,0])
sage: E.simon_two_descent()
(1, 2, [(1 : 2 : 1)])
```

The upper bound on the 2-Selmer rank returned by this method need not be sharp. In following example, the upper bound equals the actual 2-Selmer rank plus 2 (see Issue #10735):

```
sage: E = EllipticCurve('438e1')
sage: E.simon_two_descent()
(0, 3, [])
sage: E.selmer_rank()  # uses mwrank
1
```

supersingular_primes(*B*)

Return a list of all supersingular primes for this elliptic curve up to and possibly including B.

EXAMPLES:

```
sage: e = EllipticCurve('11a')
sage: e.aplist(20)
[-2, -1, 1, -2, 1, 4, -2, 0]
sage: e.supersingular_primes(1000)
[2, 19, 29, 199, 569, 809]
```

```
sage: e = EllipticCurve('27a')
sage: e.aplist(20)
[0, 0, 0, -1, 0, 5, 0, -7]
sage: e.supersingular_primes(97)
[2, 5, 11, 17, 23, 29, 41, 47, 53, 59, 71, 83, 89]
sage: e.ordinary_primes(97)
[7, 13, 19, 31, 37, 43, 61, 67, 73, 79, 97]
sage: e.supersingular_primes(3)
[2]
sage: e.supersingular_primes(2)
[2]
sage: e.supersingular_primes(1)
[]
```

tamagawa_exponent(p)

The Tamagawa index of the elliptic curve at p.

This is the index of the component group $E(\mathbf{Q}_p)/E^0(\mathbf{Q}_p)$. It equals the Tamagawa number (as the component group is cyclic) except for types I_m^* (m even) when the group can be $C_2 \times C_2$.

EXAMPLES:

```
sage: E = EllipticCurve('816a1')
sage: E.tamagawa_number(2)
4
sage: E.tamagawa_exponent(2)
2
sage: E.kodaira_symbol(2)
I2*
```

```
sage: E = EllipticCurve('200c4')
sage: E.kodaira_symbol(5)
I4*
sage: E.tamagawa_number(5)
4
sage: E.tamagawa_exponent(5)
2
```

See Issue #4715:

```
sage: E = EllipticCurve('117a3')
sage: E.tamagawa_exponent(13)
4
```

tamagawa_number(p)

The Tamagawa number of the elliptic curve at p.

This is the order of the component group $E(\mathbf{Q}_p)/E^0(\mathbf{Q}_p)$.

EXAMPLES:

```
sage: E = EllipticCurve('11a')
sage: E.tamagawa_number(11)
5
sage: E = EllipticCurve('37b')
sage: E.tamagawa_number(37)
3
```

tamagawa_number_old(p)

The Tamagawa number of the elliptic curve at p.

This is the order of the component group $E(\mathbf{Q}_p)/E^0(\mathbf{Q}_p)$.

EXAMPLES:

```
sage: E = EllipticCurve('11a')
sage: E.tamagawa_number_old(11)
5
sage: E = EllipticCurve('37b')
sage: E.tamagawa_number_old(37)
3
```

tamagawa_product()

Return the product of the Tamagawa numbers.

EXAMPLES:

```
sage: E = EllipticCurve('54a')
sage: E.tamagawa_product ()
3
```

tate_curve(p)

Create the Tate curve over the p-adics associated to this elliptic curve.

This Tate curve is a p-adic curve with split multiplicative reduction of the form $y^2 + xy = x^3 + s_4x + s_6$ which is isomorphic to the given curve over the algebraic closure of $\mathbf{Q}_p$. Its points over $\mathbf{Q}_p$ are isomorphic to $\mathbf{Q}_p^\times/q^{\mathbf{Z}}$ for a certain parameter $q \in \mathbf{Z}_p$.

INPUT:

- p – a prime where the curve has split multiplicative reduction

EXAMPLES:

```
sage: e = EllipticCurve('130a1')
sage: e.tate_curve(2)
2-adic Tate curve associated to the Elliptic Curve
 defined by y^2 + x*y + y = x^3 - 33*x + 68 over Rational Field
```

The input curve must have multiplicative reduction at the prime.

```
sage: e.tate_curve(3)
Traceback (most recent call last):
...
ValueError: the elliptic curve must have multiplicative reduction at 3
```

We compute with $p = 5$:

```
sage: T = e.tate_curve(5); T
5-adic Tate curve associated to the Elliptic Curve
 defined by y^2 + x*y + y = x^3 - 33*x + 68 over Rational Field
```

We find the Tate parameter q:

```
sage: T.parameter(prec=5)
3*5^3 + 3*5^4 + 2*5^5 + 2*5^6 + 3*5^7 + O(5^8)
```

We compute the $\mathcal{L}$-invariant of the curve:

```
sage: T.L_invariant(prec=10)
5^3 + 4*5^4 + 2*5^5 + 2*5^6 + 2*5^7 + 3*5^8 + 5^9 + O(5^10)
```

three_selmer_rank(*algorithm='UseSUnits'*)

Return the 3-selmer rank of this elliptic curve, computed using Magma.

INPUT:

- `algorithm` – 'Heuristic' (which is usually much faster in large examples), 'FindCubeRoots', or 'UseSUnits' (default)

OUTPUT: nonnegative integer

EXAMPLES: A rank 0 curve:

```
sage: EllipticCurve('11a').three_selmer_rank()       # optional - magma
0
```

A rank 0 curve with rational 3-isogeny but no 3-torsion

```
sage: EllipticCurve('14a3').three_selmer_rank()      # optional - magma
0
```

A rank 0 curve with rational 3-torsion:

```
sage: EllipticCurve('14a1').three_selmer_rank()      # optional - magma
1
```

A rank 1 curve with rational 3-isogeny:

```
sage: EllipticCurve('91b').three_selmer_rank()       # optional - magma
2
```

A rank 0 curve with nontrivial 3-Sha. The Heuristic option makes this about twice as fast as without it.

```
sage: EllipticCurve('681b').three_selmer_rank(algorithm='Heuristic')    # long␣
↪time (10 seconds); optional - magma
2
```

torsion_order()

Return the order of the torsion subgroup.

EXAMPLES:

```
sage: e = EllipticCurve('11a')
sage: e.torsion_order()
5
sage: type(e.torsion_order())
<... 'sage.rings.integer.Integer'>
sage: e = EllipticCurve([1,2,3,4,5])
sage: e.torsion_order()
1
sage: type(e.torsion_order())
<... 'sage.rings.integer.Integer'>
```

torsion_points()

Return the torsion points of this elliptic curve as a sorted list.

OUTPUT: A list of all the torsion points on this elliptic curve.

EXAMPLES:

```
sage: EllipticCurve('11a').torsion_points()
[(0 : 1 : 0), (5 : -6 : 1), (5 : 5 : 1), (16 : -61 : 1), (16 : 60 : 1)]
sage: EllipticCurve('37b').torsion_points()
[(0 : 1 : 0), (8 : -19 : 1), (8 : 18 : 1)]
```

Some curves with large torsion groups:

```
sage: E = EllipticCurve([-1386747, 368636886])
sage: T = E.torsion_subgroup(); T
Torsion Subgroup isomorphic to Z/8 + Z/2 associated to the
 Elliptic Curve defined by y^2 = x^3 - 1386747*x + 368636886 over
 Rational Field
sage: E.torsion_points()
[(0 : 1 : 0),
 (-1293 : 0 : 1),
 (-933 : -29160 : 1),
 (-933 : 29160 : 1),
 (-285 : -27216 : 1),
 (-285 : 27216 : 1),
 (147 : -12960 : 1),
 (147 : 12960 : 1),
 (282 : 0 : 1),
 (1011 : 0 : 1),
 (1227 : -22680 : 1),
 (1227 : 22680 : 1),
 (2307 : -97200 : 1),
 (2307 : 97200 : 1),
 (8787 : -816480 : 1),
 (8787 : 816480 : 1)]
sage: EllipticCurve('210b5').torsion_points()
[(0 : 1 : 0),
 (-41/4 : 37/8 : 1),
 (-5 : -103 : 1),
 (-5 : 107 : 1),
 (10 : -208 : 1),
 (10 : 197 : 1),
 (37 : -397 : 1),
 (37 : 359 : 1),
 (100 : -1153 : 1),
 (100 : 1052 : 1),
 (415 : -8713 : 1),
 (415 : 8297 : 1)]
sage: EllipticCurve('210e2').torsion_points()
[(0 : 1 : 0),
 (-36 : 18 : 1),
 (-26 : -122 : 1),
 (-26 : 148 : 1),
 (-8 : -122 : 1),
 (-8 : 130 : 1),
 (4 : -62 : 1),
 (4 : 58 : 1),
 (31/4 : -31/8 : 1),
 (28 : -14 : 1),
 (34 : -122 : 1),
 (34 : 88 : 1),
 (64 : -482 : 1),
```

(continues on next page)

(continued from previous page)

```
 (64 : 418 : 1),
 (244 : -3902 : 1),
 (244 : 3658 : 1)]
```

torsion_subgroup()

Return the torsion subgroup of this elliptic curve.

OUTPUT: The EllipticCurveTorsionSubgroup instance associated to this elliptic curve.

Note: To see the torsion points as a list, use *torsion_points()*.

EXAMPLES:

```
sage: EllipticCurve('11a').torsion_subgroup()
Torsion Subgroup isomorphic to Z/5 associated to the
 Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field
sage: EllipticCurve('37b').torsion_subgroup()
Torsion Subgroup isomorphic to Z/3 associated to the
 Elliptic Curve defined by y^2 + y = x^3 + x^2 - 23*x - 50 over Rational Field
```

```
sage: e = EllipticCurve([-1386747,368636886]); e
Elliptic Curve defined by y^2 = x^3 - 1386747*x + 368636886 over Rational␣
↪Field
sage: G = e.torsion_subgroup(); G
Torsion Subgroup isomorphic to Z/8 + Z/2 associated to the
 Elliptic Curve defined by y^2 = x^3 - 1386747*x + 368636886 over
 Rational Field
sage: G.0*3 + G.1
(1227 : 22680 : 1)
sage: G.1
(282 : 0 : 1)
sage: list(G)
[(0 : 1 : 0), (147 : -12960 : 1), (2307 : -97200 : 1), (-933 : -29160 : 1),
 (1011 : 0 : 1), (-933 : 29160 : 1), (2307 : 97200 : 1), (147 : 12960 : 1),
 (-1293 : 0 : 1), (1227 : 22680 : 1), (-285 : 27216 : 1), (8787 : 816480 : 1),
 (282 : 0 : 1), (8787 : -816480 : 1), (-285 : -27216 : 1), (1227 : -22680 :␣
↪1)]
```

two_descent(*verbose=True*, *selmer_only=False*, *first_limit=20*, *second_limit=8*, *n_aux=-1*, *second_descent=1*)

Compute 2-descent data for this curve.

INPUT:

- `verbose` – (default: `True`) print what mwrank is doing; if `False`, **no output** is printed
- `selmer_only` – (default: `False`) selmer_only switch
- `first_limit` – (default: 20) firstlim is bound on x+z second_limit- (default: 8) secondlim is bound on log max x,z , i.e. logarithmic
- `n_aux` – (default: -1) n_aux only relevant for general 2-descent when 2-torsion trivial; n_aux=-1 causes default to be used (depends on method)
- `second_descent` – (default: `True`) second_descent only relevant for descent via 2-isogeny

OUTPUT:

Return `True` if the descent succeeded, i.e. if the lower bound and the upper bound for the rank are the same. In this case, generators and the rank are cached. A return value of `False` indicates that either rational points were not found, or that Sha[2] is nontrivial and mwrank was unable to determine this for sure.

EXAMPLES:

```
sage: E = EllipticCurve('37a1')
sage: E.two_descent(verbose=False)
True
```

two_descent_simon(*verbose=0*, *lim1=5*, *lim3=50*, *limtriv=3*, *maxprob=20*, *limbigprime=30*, *known_points=None*)

Return lower and upper bounds on the rank of the Mordell-Weil group $E(\mathbf{Q})$ and a list of points of infinite order.

Warning: This function is deprecated as the functionality of Simon's script for elliptic curves over the rationals has been ported over to pari. Use *rank()* with the keyword `algorithm='pari'` instead.

INPUT:

- `verbose` – 0, 1, 2, or 3 (default: 0), the verbosity level
- `lim1` – (default: 5) limit on trivial points on quartics
- `lim3` – (default: 50) limit on points on ELS quartics
- `limtriv` – (default: 3) limit on trivial points on E
- `maxprob` – (default: 20)
- `limbigprime` – (default: 30) to distinguish between small and large prime numbers. Use probabilistic tests for large primes. If 0, don't any probabilistic tests.
- `known_points` – (default: None) list of known points on the curve

OUTPUT: a triple `(lower, upper, list)` consisting of

- `lower` (integer) – lower bound on the rank
- `upper` (integer) – upper bound on the rank
- `list` – list of points of infinite order in $E(\mathbf{Q})$

The integer `upper` is in fact an upper bound on the dimension of the 2-Selmer group, hence on the dimension of $E(\mathbf{Q})/2E(\mathbf{Q})$. It is equal to the dimension of the 2-Selmer group except possibly if $E(\mathbf{Q})[2]$ has dimension 1. In that case, `upper` may exceed the dimension of the 2-Selmer group by an even number, due to the fact that the algorithm does not perform a second descent.

To obtain a list of generators, use E.gens().

IMPLEMENTATION:

Uses Denis Simon's PARI/GP scripts from http://www.math.unicaen.fr/~simon/

EXAMPLES:

We compute the ranks of the curves of lowest known conductor up to rank 8. Amazingly, each of these computations finishes almost instantly!

```
sage: E = EllipticCurve('11a1')
sage: E.simon_two_descent()
doctest:warning
...
DeprecationWarning: Use E.rank(algorithm="pari") instead, as this script has␣
↪been ported over to pari.
See https://github.com/sagemath/sage/issues/35621 for details.
(0, 0, [])
sage: E = EllipticCurve('37a1')
sage: E.simon_two_descent()
(1, 1, [(0 : 0 : 1)])
sage: E = EllipticCurve('389a1')
sage: E._known_points = []  # clear cached points
sage: E.simon_two_descent()
(2, 2, [(5/4 : 5/8 : 1), (-3/4 : 7/8 : 1)])
sage: E = EllipticCurve('5077a1')
sage: E.simon_two_descent()
(3, 3, [(1 : 0 : 1), (2 : 0 : 1), (0 : 2 : 1)])
```

In this example Simon's program does not find any points, though it does correctly compute the rank of the 2-Selmer group.

```
sage: E = EllipticCurve([1, -1, 0, -751055859, -7922219731979])
sage: E.simon_two_descent()
(1, 1, [])
```

The rest of these entries were taken from Tom Womack's page http://tom.womack.net/maths/conductors.htm

```
sage: E = EllipticCurve([1, -1, 0, -79, 289])
sage: E.simon_two_descent()
(4, 4, [(6 : -1 : 1), (4 : 3 : 1), (5 : -2 : 1), (8 : 7 : 1)])
sage: E = EllipticCurve([0, 0, 1, -79, 342])
sage: E.simon_two_descent()  # long time (9s on sage.math, 2011)
(5, 5, [(5 : 8 : 1), (10 : 23 : 1), (3 : 11 : 1), (-3 : 23 : 1), (0 : 18 :␣
↪1)])
sage: E = EllipticCurve([1, 1, 0, -2582, 48720])
sage: r, s, G = E.simon_two_descent(); r,s
(6, 6)
sage: E = EllipticCurve([0, 0, 0, -10012, 346900])
sage: r, s, G = E.simon_two_descent(); r,s  # long time
(7, 7)
sage: E = EllipticCurve([0, 0, 1, -23737, 960366])
sage: r, s, G = E.simon_two_descent(); r,s  # long time
(8, 8)
```

Example from Issue #10832:

```
sage: E = EllipticCurve([1,0,0,-6664,86543])
sage: E.simon_two_descent()
(2, 3, [(-1/4 : 2377/8 : 1), (323/4 : 1891/8 : 1)])
sage: E.rank()
2
sage: E.gens()
[(-1/4 : 2377/8 : 1), (323/4 : 1891/8 : 1)]
```

Example where the lower bound is known to be 1 despite that the algorithm has not found any points of infinite order

```
sage: E = EllipticCurve([1, 1, 0, -23611790086, 1396491910863060])
sage: E.simon_two_descent()
(1, 2, [])
sage: E.rank()
1
sage: E.gens()      # uses mwrank
[(4311692542083/48594841 : -13035144436525227/338754636611 : 1)]
```

Example for Issue #5153:

```
sage: E = EllipticCurve([3,0])
sage: E.simon_two_descent()
(1, 2, [(1 : 2 : 1)])
```

The upper bound on the 2-Selmer rank returned by this method need not be sharp. In following example, the upper bound equals the actual 2-Selmer rank plus 2 (see Issue #10735):

```
sage: E = EllipticCurve('438e1')
sage: E.simon_two_descent()
(0, 3, [])
sage: E.selmer_rank()  # uses mwrank
1
```

sage.schemes.elliptic_curves.ell_rational_field.**cremona_curves**(*conductors*)

Return iterator over all known curves (in database) with conductor in the list of conductors.

EXAMPLES:

```
sage: [(E.label(), E.rank()) for E in cremona_curves(srange(35,40))]
[('35a1', 0),
 ('35a2', 0),
 ('35a3', 0),
 ('36a1', 0),
 ('36a2', 0),
 ('36a3', 0),
 ('36a4', 0),
 ('37a1', 1),
 ('37b1', 0),
 ('37b2', 0),
 ('37b3', 0),
 ('38a1', 0),
 ('38a2', 0),
 ('38a3', 0),
 ('38b1', 0),
 ('38b2', 0),
 ('39a1', 0),
 ('39a2', 0),
 ('39a3', 0),
 ('39a4', 0)]
```

sage.schemes.elliptic_curves.ell_rational_field.**cremona_optimal_curves**(*conductors*)

Return iterator over all known optimal curves (in database) with conductor in the list of conductors.

EXAMPLES:

```
sage: [(E.label(), E.rank()) for E in cremona_optimal_curves(srange(35,40))]
[('35a1', 0),
 ('36a1', 0),
 ('37a1', 1),
 ('37b1', 0),
 ('38a1', 0),
 ('38b1', 0),
 ('39a1', 0)]
```

There is one case – 990h3 – when the optimal curve isn't labeled with a 1:

```
sage: [e.cremona_label() for e in cremona_optimal_curves([990])]
['990a1', '990b1', '990c1', '990d1', '990e1', '990f1', '990g1',
 '990h3', '990i1', '990j1', '990k1', '990l1']
```

sage.schemes.elliptic_curves.ell_rational_field.**elliptic_curve_congruence_graph**(*curves*)

Return the congruence graph for this set of elliptic curves.

INPUT:

- `curves` – a list of elliptic curves

OUTPUT:

The graph with each curve as a vertex (labelled by its Cremona label) and an edge from E to F labelled p if and only if E is congruent to F mod p

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_rational_field import elliptic_curve_
↪congruence_graph
sage: curves = list(cremona_optimal_curves([11..30]))
sage: G = elliptic_curve_congruence_graph(curves)
sage: G
Graph on 12 vertices
```

sage.schemes.elliptic_curves.ell_rational_field.**integral_points_with_bounded_mw_coeffs**(*E*, *mw_*
N,
x_bo

Return the set of integers x which are x-coordinates of points on the curve E which are linear combinations of the generators (basis and torsion points) with coefficients bounded by N.

INPUT:

- `E` – an elliptic curve
- `mw_base` – a list of points on E (generators)
- `N` – a positive integer (bound on coefficients)
- `x_bound` – a positive real number (upper bound on size of x-coordinates)

OUTPUT:

(list) list of integral points on E which are linear combinations of the given points with coefficients bounded by N in absolute value.

18.2 Tables of elliptic curves of given rank

The default database of curves contains the following data:

Rank	Number of curves	Maximal conductor
0	30427	9999
1	31871	9999
2	2388	9999
3	836	119888
4	10	1175648
5	5	37396136
6	5	6663562874
7	5	896913586322
8	6	457532830151317
9	7	~9.612839e+21
10	6	~1.971057e+21
11	6	~1.803406e+24
12	1	~2.696017e+29
14	1	~3.627533e+37
15	1	~1.640078e+56
17	1	~2.750021e+56
19	1	~1.373776e+65
20	1	~7.381324e+73
21	1	~2.611208e+85
22	1	~2.272064e+79
23	1	~1.139647e+89
24	1	~3.257638e+95
28	1	~3.455601e+141

Note that lists for r>=4 are not exhaustive; there may well be curves of the given rank with conductor less than the listed maximal conductor, which are not included in the tables.

AUTHORS:

- William Stein (2007-10-07): initial version
- Simon Spicer (2014-10-24): Added examples of more high-rank curves

See also the functions *cremona_curves()* and *cremona_optimal_curves()* which enable easy looping through the Cremona elliptic curve database.

class sage.schemes.elliptic_curves.ec_database.**EllipticCurves**

Bases: object

rank(*rank*, *tors=0*, *n=10*, *labels=False*)

Return a list of at most n curves with given rank and torsion order.

INPUT:

- rank (int) – the desired rank
- tors (int, default 0) – the desired torsion order (ignored if 0)
- n (int, default 10) – the maximum number of curves returned.
- labels (bool, default: False) – if True, return Cremona labels instead of curves.

OUTPUT:

(list) A list at most n of elliptic curves of required rank.

EXAMPLES:

```
sage: elliptic_curves.rank(n=5, rank=3, tors=2, labels=True)
['59450i1', '59450i2', '61376c1', '61376c2', '65481c1']
```

```
sage: elliptic_curves.rank(n=5, rank=0, tors=5, labels=True)
['11a1', '11a3', '38b1', '50b1', '50b2']
```

```
sage: elliptic_curves.rank(n=5, rank=1, tors=7, labels=True)
['574i1', '4730k1', '6378c1']
```

```
sage: e = elliptic_curves.rank(6)[0]; e.ainvs(), e.conductor()
((1, 1, 0, -2582, 48720), 5187563742)
sage: e = elliptic_curves.rank(7)[0]; e.ainvs(), e.conductor()
((0, 0, 0, -10012, 346900), 382623908456)
sage: e = elliptic_curves.rank(8)[0]; e.ainvs(), e.conductor()
((1, -1, 0, -106384, 13075804), 249649566346838)
```

For large conductors, the labels are not known:

```
sage: L = elliptic_curves.rank(6, n=3); L
[Elliptic Curve defined by y^2 + x*y = x^3 + x^2 - 2582*x + 48720 over␣
↪Rational Field,
 Elliptic Curve defined by y^2 + y = x^3 - 7077*x + 235516 over Rational␣
↪Field,
 Elliptic Curve defined by y^2 + x*y = x^3 - x^2 - 2326*x + 43456 over␣
↪Rational Field]
sage: L[0].cremona_label()
Traceback (most recent call last):
...
LookupError: Cremona database does not contain entry for Elliptic Curve
defined by y^2 + x*y = x^3 + x^2 - 2582*x + 48720 over Rational Field
sage: elliptic_curves.rank(6, n=3, labels=True)
[]
```

18.3 Elliptic curves over number fields

An elliptic curve E over a number field K can be given by a Weierstrass equation whose coefficients lie in K or by using `base_extend` on an elliptic curve defined over a subfield.

One major difference to elliptic curves over $\mathbf{Q}$ is that there might not exist a global minimal equation over K, when K does not have class number one. Another difference is the lack of understanding of modularity for general elliptic curves over general number fields.

Currently Sage can obtain local information about E/K_v for finite places v, it has an interface to Denis Simon's script for 2-descent, it can compute the torsion subgroup of the Mordell-Weil group $E(K)$, and it can work with isogenies defined over K.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2 + 1)
sage: E = EllipticCurve([0, 4+i])
sage: E.discriminant()
-3456*i - 6480
sage: P= E([i,2])
sage: P+P
(-2*i + 9/16 : -9/4*i - 101/64 : 1)
```

```
sage: E.has_good_reduction(2 + i)
True
sage: E.local_data(4+i)
Local data at Fractional ideal (i + 4):
  Reduction type: bad additive
  Local minimal model: Elliptic Curve defined by y^2 = x^3 + (i+4)
                       over Number Field in i with defining polynomial x^2 + 1
  Minimal discriminant valuation: 2
  Conductor exponent: 2
  Kodaira Symbol: II
  Tamagawa Number: 1
sage: E.tamagawa_product_bsd()
1
```

```
sage: E.simon_two_descent()
(1, 1, [(i : 2 : 1)])
```

```
sage: E.torsion_order()
1
```

```
sage: E.isogenies_prime_degree(3)
[Isogeny of degree 3
  from Elliptic Curve defined by y^2 = x^3 + (i+4)
       over Number Field in i with defining polynomial x^2 + 1
    to Elliptic Curve defined by y^2 = x^3 + (-27*i-108)
       over Number Field in i with defining polynomial x^2 + 1]
```

AUTHORS:

- Robert Bradshaw 2007
- John Cremona
- Chris Wuthrich

REFERENCE:

- [Sil] Silverman, Joseph H. The arithmetic of elliptic curves. Second edition. Graduate Texts in Mathematics, 106. Springer, 2009.
- [Sil2] Silverman, Joseph H. Advanced topics in the arithmetic of elliptic curves. Graduate Texts in Mathematics, 151. Springer, 1994.

class sage.schemes.elliptic_curves.ell_number_field.**EllipticCurve_number_field**(*K*, *ainvs*)

Bases: *EllipticCurve_field*

Elliptic curve over a number field.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2 + 1)
sage: EllipticCurve([i, i - 1, i + 1, 24*i + 15, 14*i + 35])
Elliptic Curve defined by
 y^2 + i*x*y + (i+1)*y = x^3 + (i-1)*x^2 + (24*i+15)*x + (14*i+35)
 over Number Field in i with defining polynomial x^2 + 1
```

base_extend(*R*)

Return the base extension of `self` to R.

EXAMPLES:

```
sage: E = EllipticCurve('11a3')
sage: K = QuadraticField(-5, 'a')
sage: E.base_extend(K)
Elliptic Curve defined by y^2 + y = x^3 + (-1)*x^2 over Number Field in a
 with defining polynomial x^2 + 5 with a = 2.236067977499790?*I
```

Check that non-torsion points are remembered when extending the base field (see Issue #16034):

```
sage: E = EllipticCurve([1, 0, 1, -1751, -31352])
sage: K.<d> = QuadraticField(5)
sage: E.gens()
[(52 : 111 : 1)]
sage: EK = E.base_extend(K)
sage: EK.gens()
[(52 : 111 : 1)]
```

cm_discriminant()

Return the CM discriminant of the j-invariant of this curve, or 0.

OUTPUT:

An integer D which is either 0 if this curve E does not have Complex Multiplication) (CM), or an imaginary quadratic discriminant if $j(E)$ is the j-invariant of the order with discriminant D.

Note: If E has CM but the discriminant D is not a square in the base field K then the extra endomorphisms will not be defined over K. See also *has_rational_cm()*.

EXAMPLES:

```
sage: EllipticCurve(j=0).cm_discriminant()
-3
sage: EllipticCurve(j=1).cm_discriminant()
Traceback (most recent call last):
...
ValueError: Elliptic Curve defined by y^2 + x*y = x^3 + 36*x + 3455
over Rational Field does not have CM
sage: EllipticCurve(j=1728).cm_discriminant()
-4
sage: EllipticCurve(j=8000).cm_discriminant()
-8
sage: K.<a> = QuadraticField(5)
sage: EllipticCurve(j=282880*a + 632000).cm_discriminant()
-20
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)
```

(continues on next page)

(continued from previous page)

```
sage: EllipticCurve(j=31710790944000*a^2 + 39953093016000*a + 50337742902000).
↪cm_discriminant()
-108
```

conductor()

Return the conductor of this elliptic curve as a fractional ideal of the base field.

OUTPUT:

(fractional ideal) The conductor of the curve.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2 + 1)
sage: EllipticCurve([i, i - 1, i + 1, 24*i + 15, 14*i + 35]).conductor()
Fractional ideal (21*i - 3)
sage: K.<a> = NumberField(x^2 - x + 3)
sage: EllipticCurve([1 + a, -1 + a, 1 + a, -11 + a, 5 - 9*a]).conductor()
Fractional ideal (-6*a)
```

A not so well known curve with everywhere good reduction:

```
sage: K.<a> = NumberField(x^2 - 38)
sage: E = EllipticCurve([0,0,0, 217968148569327655682438l0*a -␣
↪134364590724198567128296995, 121774567239345229314269094644186997594*a -␣
↪750668847495706904791115375024037711300])
sage: E.conductor()
Fractional ideal (1)
```

An example which used to fail (see Issue #5307):

```
sage: K.<w> = NumberField(x^2 + x + 6)
sage: E = EllipticCurve([w, -1, 0, -w-6, 0])
sage: E.conductor()
Fractional ideal (86304, w + 5898)
```

An example raised in Issue #11346:

```
sage: K.<g> = NumberField(x^2 - x - 1)
sage: E1 = EllipticCurve(K, [0, 0, 0, -1/48, -161/864])
sage: [(p.smallest_integer(), e) for p,e in E1.conductor().factor()]
[(2, 4), (3, 1), (5, 1)]
```

galois_representation()

The compatible family of the Galois representation attached to this elliptic curve.

Given an elliptic curve E over a number field K and a rational prime number p, the p^n-torsion $E[p^n]$ points of E is a representation of the absolute Galois group of K. As n varies we obtain the Tate module T_pE which is a a representation of G_K on a free $\mathbf{Z}_p$-module of rank 2. As p varies the representations are compatible.

EXAMPLES:

```
sage: x = polygen(ZZ, 'x')
sage: K = NumberField(x**2 + 1, 'a')
sage: E = EllipticCurve('11a1').change_ring(K)
sage: rho = E.galois_representation()
sage: rho
```

(continues on next page)

(continued from previous page)

```
Compatible family of Galois representations associated to the
 Elliptic Curve defined by y^2 + y = x^3 + (-1)*x^2 + (-10)*x + (-20)
  over Number Field in a with defining polynomial x^2 + 1
sage: rho.is_surjective(3)
True
sage: rho.is_surjective(5)  # long time (4s on sage.math, 2014)
False
sage: rho.non_surjective()
[5]
```

gens (***kwds*)

Return some points of infinite order on this elliptic curve.

Contrary to what the name of this method suggests, the points it returns do not always generate a subgroup of full rank in the Mordell-Weil group, nor are they necessarily linearly independent. Moreover, the number of points can be smaller or larger than what one could expect after calling *rank()* or *rank_bounds()*.

Note: The optional parameters control the Simon two descent algorithm; see the documentation of *simon_two_descent()* for more details.

INPUT:

- `verbose` – 0, 1, 2, or 3 (default: 0), the verbosity level
- `lim1` – (default: 2) limit on trivial points on quartics
- `lim3` – (default: 4) limit on points on ELS quartics
- `limtriv` – (default: 2) limit on trivial points on elliptic curve
- `maxprob` – (default: 20)
- `limbigprime` – (default: 30) to distinguish between small and large prime numbers. Use probabilistic tests for large primes. If 0, do not use probabilistic tests.
- `known_points` – (default: None) list of known points on the curve

OUTPUT:

A set of points of infinite order given by the Simon two-descent.

Note: For non-quadratic number fields, this code does return, but it takes a long time.

EXAMPLES:

```
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^2 + 23, 'a')
sage: E = EllipticCurve(K,[0,0,0,101,0])
sage: E.gens()
[(23831509/8669448*a - 2867471/8669448 : 76507317707/18049790736*a -␣
↪424166479633/18049790736 : 1),
 (-2031032029/969232392*a + 58813561/969232392 : -15575984630401/
↪21336681877488*a + 451041199309/21336681877488 : 1),
 (-186948623/4656964 : 549438861195/10049728312*a : 1)]
```

It can happen that no points are found if the height bounds used in the search are too small (see Issue #10745):

```
sage: K.<t> = NumberField(x^4 + x^2 - 7)
sage: E = EllipticCurve(K, [1, 0, 5*t^2 + 16, 0, 0])
sage: E.gens(lim1=1, lim3=1)
[]
sage: E.rank()
1
sage: gg=E.gens(lim3=13); gg  # long time (about 4s)
[(... : 1)]
```

Check that the the point found has infinite order, and that it is on the curve:

```
sage: P=gg[0]; P.order()  # long time
+Infinity
sage: E.defining_polynomial()(*P)  # long time
0
```

Here is a curve of rank 2:

```
sage: K.<t> = NumberField(x^2 - 17)
sage: E = EllipticCurve(K, [-4, 0])
sage: E.gens()
[(-1/2*t + 1/2 : -1/2*t + 1/2 : 1), (-t + 3 : -2*t + 10 : 1)]
sage: E.rank()
2
```

Test that points of finite order are not included (see Issue #13593):

```
sage: E = EllipticCurve("17a3")
sage: K.<t> = NumberField(x^2 + 3)
sage: EK = E.base_extend(K)
sage: EK.rank()
0
sage: EK.gens()
[]
```

IMPLEMENTATION:

For curves over quadratic fields which are base-changes from **Q**, we delegate the work to *gens_quadratic()* where methods over **Q** suffice. Otherwise, we use Denis Simon's PARI/GP scripts from http://www.math.unicaen.fr/~simon/.

gens_quadratic(***kwds*)

Return generators for the Mordell-Weil group modulo torsion, for a curve which is a base change from **Q** to a quadratic field.

EXAMPLES:

```
sage: E = EllipticCurve([1,2,3,40,50])
sage: E.conductor()
2123582
sage: E.gens()
[(5 : 17 : 1)]
sage: K.<i> = QuadraticField(-1)
sage: EK = E.change_ring(K)
sage: EK.gens_quadratic()
[(5 : 17 : 1), (-13 : 48*i + 5 : 1)]

sage: E.change_ring(QuadraticField(3, 'a')).gens_quadratic()
```

(continues on next page)

(continued from previous page)

```
[(5 : 17 : 1), (-1 : 2*a - 1 : 1), (11/4 : 33/4*a - 23/8 : 1)]

sage: K.<a> = QuadraticField(-7)
sage: E = EllipticCurve([0,0,0,197,0])
sage: E.conductor()
2483776
sage: E.gens()
[(47995604297578081/7389879786648100 : -25038161802544048018837479/
↪63526665583012979412100 : 1)]
sage: K.<a> = QuadraticField(7)
sage: E.change_ring(K).gens_quadratic()
[(-1209642055/59583566*a + 1639995844/29791783 : -377240626321899/
↪1720892553212*a + 138577803462855/245841793316 : 1),
 (1/28 : 393/392*a : 1),
 (-61*a + 162 : 1098*a - 2916 : 1)]

sage: E = EllipticCurve([1, a])
sage: E.gens_quadratic()
Traceback (most recent call last):
...
ValueError: gens_quadratic() requires the elliptic curve to be a base change␣
↪from Q
```

global_integral_model()

Return a model of self which is integral at all primes.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2 + 1)
sage: E = EllipticCurve([i/5,i/5,i/5,i/5,i/5])
sage: P1, P2 = K.primes_above(5)
sage: E.global_integral_model()
Elliptic Curve defined by
 y^2 + (-i)*x*y + (-25*i)*y = x^3 + 5*i*x^2 + 125*i*x + 3125*i
 over Number Field in i with defining polynomial x^2 + 1
```

Issue #7935:

```
sage: K.<a> = NumberField(x^2 - 38)
sage: E = EllipticCurve([a,1/2])
sage: E.global_integral_model()
Elliptic Curve defined by y^2 = x^3 + 1444*a*x + 27436
 over Number Field in a with defining polynomial x^2 - 38
```

Issue #9266:

```
sage: K.<s> = NumberField(x^2 - 5)
sage: w = (1+s)/2
sage: E = EllipticCurve(K, [2,w])
sage: E.global_integral_model()
Elliptic Curve defined by y^2 = x^3 + 2*x + (1/2*s+1/2)
 over Number Field in s with defining polynomial x^2 - 5
```

Issue #12151:

```
sage: K.<v> = NumberField(x^2 + 161*x - 150)
sage: E = EllipticCurve([25105/216*v - 3839/36, 634768555/7776*v - 98002625/
```

(continues on next page)

(continued from previous page)

```
↪1296, 634768555/7776*v - 98002625/1296, 0, 0])
sage: M = E.global_integral_model(); M # choice varies, not tested
Elliptic Curve defined by
y^2 + (2094779518028859*v-1940492905300351)*x*y +␣
↪(47799726847254419310117823445416530407112750 0*v-
↪44279137744134685291993077384950287195809750 0)*y = x^3 +␣
↪(26519784690047674853185542622500*v-24566525306469707225840460652500)*x^2
 over Number Field in v with defining polynomial x^2 + 161*x - 150
```

Issue #14476:

```
sage: R.<t> = QQ[]
sage: K.<g> = NumberField(t^4 - t^3 - 3*t^2 - t + 1)
sage: E = EllipticCurve([ -43/625*g^3 + 14/625*g^2 - 4/625*g + 706/625, -4862/
↪78125*g^3 - 4074/78125*g^2 - 711/78125*g + 10304/78125,  -4862/78125*g^3 -␣
↪4074/78125*g^2 - 711/78125*g + 10304/78125, 0,0])
sage: E.global_integral_model()
Elliptic Curve defined by
 y^2 + (15*g^3-48*g-42)*x*y + (-111510*g^3-162162*g^2-44145*g+37638)*y = x^3␣
↪+ (-954*g^3-1134*g^2+81*g+576)*x^2
 over Number Field in g with defining polynomial t^4 - t^3 - 3*t^2 - t + 1
```

global_minimal_model (*proof=None*, *semi_global=False*)

Return a model of self that is integral, and minimal.

Note: Over fields of class number greater than 1, a global minimal model may not exist. If it does not, set the parameter `semi_global` to `True` to obtain a model minimal at all but one prime.

INPUT:

- `proof` – whether to only use provably correct methods (default controlled by global proof module). Note that the proof module is number_field, not elliptic_curves, since the functions that actually need the flag are in number fields.
- `semi_global` (boolean, default: `False`) – if there is no global minimal mode, return a semi-global minimal model (minimal at all but one prime) instead, if True; raise an error if False. No effect if a global minimal model exists.

OUTPUT:

A global integral and minimal model, or an integral model minimal at all but one prime of there is no global minimal model and the flag `semi_global` is True.

EXAMPLES:

```
sage: K.<a> = NumberField(x^2 - 38)
sage: E = EllipticCurve([0,0,0, 21796814856932765568243810*a -␣
↪13436459072419856712829699 5, 12177456723934522931426909464418699759 4*a -␣
↪75066884749570690479111537502 40377113 00])
sage: E2 = E.global_minimal_model()
sage: E2
Elliptic Curve defined by
 y^2 + a*x*y + (a+1)*y = x^3 + (a+1)*x^2 + (4*a+15)*x + (4*a+21)
 over Number Field in a with defining polynomial x^2 - 38
sage: E2.local_data()
[]
```

See Issue #11347:

```
sage: K.<g> = NumberField(x^2 - x - 1)
sage: E = EllipticCurve(K, [0, 0, 0, -1/48, 161/864])
sage: E2 = E.integral_model().global_minimal_model(); E2
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2
 over Number Field in g with defining polynomial x^2 - x - 1
sage: [(p.norm(), e) for p, e in E2.conductor().factor()]
[(9, 1), (5, 1)]
sage: [(p.norm(), e) for p, e in E2.discriminant().factor()]
[(-5, 2), (9, 1)]
```

See Issue #14472, this used not to work over a relative extension:

```
sage: K1.<w> = NumberField(x^2 + x + 1)
sage: m = polygen(K1)
sage: K2.<v> = K1.extension(m^2 - w + 1)
sage: E = EllipticCurve([0*v, -432])
sage: E.global_minimal_model()
Elliptic Curve defined by y^2 + y = x^3
 over Number Field in v with defining polynomial x^2 - w + 1 over its base␣
↪field
```

See Issue #18662: for fields of class number greater than 1, even when global minimal models did exist, their computation was not implemented. Now it is:

```
sage: K.<a> = NumberField(x^2 - 10)
sage: K.class_number()
2
sage: E = EllipticCurve([0, 0, 0, -186408*a - 589491, 78055704*a + 246833838])
sage: E.discriminant().norm()
16375845905239507992576
sage: E.discriminant().norm().factor()
2^31 * 3^27
sage: E.has_global_minimal_model()
True
sage: Emin = E.global_minimal_model(); Emin
Elliptic Curve defined by
 y^2 + (a+1)*x*y + (a+1)*y = x^3 + (-a)*x^2 + (a-12)*x + (-2*a+2)
 over Number Field in a with defining polynomial x^2 - 10
sage: Emin.discriminant().norm()
3456
sage: Emin.discriminant().norm().factor()
2^7 * 3^3
```

If there is no global minimal model, this method will raise an error unless you set the parameter `semi_global` to `True`:

```
sage: K.<a> = NumberField(x^2 - 10)
sage: K.class_number()
2
sage: E = EllipticCurve([a, a, 0, 3*a+8, 4*a+3])
sage: E.has_global_minimal_model()
False
sage: E.global_minimal_model()
Traceback (most recent call last):
...
ValueError: Elliptic Curve defined by
```

(continues on next page)

(continued from previous page)

```
y^2 + a*x*y = x^3 + a*x^2 + (3*a+8)*x + (4*a+3) over Number Field
in a with defining polynomial x^2 - 10 has no global minimal model!
For a semi-global minimal model use semi_global=True
sage: E.global_minimal_model(semi_global=True)
Elliptic Curve defined by
 y^2 + a*x*y = x^3 + a*x^2 + (3*a+8)*x + (4*a+3) over Number Field in a
 with defining polynomial x^2 - 10
```

An example of a curve with everywhere good reduction but which has no model with unit discriminant:

```
sage: K.<a> = NumberField(x^2 - x - 16)
sage: K.class_number()
2
sage: E = EllipticCurve([0,0,0,-15221331*a - 53748576, -79617688290*a -␣
↪281140318368])
sage: Emin = E.global_minimal_model(semi_global=True)
sage: Emin.ainvs()
(a, a - 1, a, 605*a - 2728, 15887*a - 71972)
sage: Emin.discriminant()
-17*a - 16
sage: Emin.discriminant().norm()
-4096
sage: Emin.minimal_discriminant_ideal()
Fractional ideal (1)
sage: E.conductor()
Fractional ideal (1)
```

global_minimality_class()

Return the obstruction to this curve having a global minimal model.

OUTPUT:

An ideal class of the base number field, which is trivial if and only if the elliptic curve has a global minimal model, and which can be used to find global and semi-global minimal models.

EXAMPLES:

A curve defined over a field of class number 2 with no global minimal model was a nontrivial minimality class:

```
sage: K.<a> = NumberField(x^2 - 10)
sage: K.class_number()
2
sage: E = EllipticCurve([0, 0, 0, -22500, 750000*a])
sage: E.global_minimality_class()
Fractional ideal class (10, 5*a)
sage: E.global_minimality_class().order()
2
```

Over the same field, a curve defined by a non-minimal model has trivial class, showing that a global minimal model does exist:

```
sage: K.<a> = NumberField(x^2 - 10)
sage: E = EllipticCurve([0, 0, 0, 4536*a+14148, -163728*a-474336])
sage: E.is_global_minimal_model()
False
sage: E.global_minimality_class()
Trivial principal fractional ideal class
```

Over a field of class number 1 the result is always the trivial class:

```
sage: K.<a> = NumberField(x^2 - 5)
sage: E = EllipticCurve([0, 0, 0, K(16), K(64)])
sage: E.global_minimality_class()
Trivial principal fractional ideal class

sage: E = EllipticCurve([0, 0, 0, 16, 64])
sage: E.base_field()
Rational Field
sage: E.global_minimality_class()
1
```

has_additive_reduction(P)

Return True if this elliptic curve has (bad) additive reduction at the prime P.

INPUT:

- `P` – a prime ideal of the base field of self, or a field element generating such an ideal.

OUTPUT:

(bool) True if the curve has additive reduction at P, else False.

EXAMPLES:

```
sage: E = EllipticCurve('27a1')
sage: [(p, E.has_additive_reduction(p)) for p in prime_range(15)]
[(2, False), (3, True), (5, False), (7, False), (11, False), (13, False)]

sage: K.<a> = NumberField(x^3 - 2)
sage: P17a, P17b = [P for P,e in K.factor(17)]
sage: E = EllipticCurve([0,0,0,0,2*a+1])
sage: [(p, E.has_additive_reduction(p)) for p in [P17a,P17b]]
[(Fractional ideal (4*a^2 - 2*a + 1), False),
 (Fractional ideal (2*a + 1), True)]
```

has_bad_reduction(P)

Return True if this elliptic curve has bad reduction at the prime P.

INPUT:

- `P` – a prime ideal of the base field of self, or a field element generating such an ideal.

OUTPUT:

(bool) True if the curve has bad reduction at P, else False.

Note: This requires determining a local integral minimal model; we do not just check that the discriminant of the current model has valuation zero.

EXAMPLES:

```
sage: E = EllipticCurve('14a1')
sage: [(p, E.has_bad_reduction(p)) for p in prime_range(15)]
[(2, True), (3, False), (5, False), (7, True), (11, False), (13, False)]

sage: K.<a> = NumberField(x^3 - 2)
sage: P17a, P17b = [P for P,e in K.factor(17)]
```

(continues on next page)

(continued from previous page)

```
sage: E = EllipticCurve([0,0,0,0,2*a+1])
sage: [(p, E.has_bad_reduction(p)) for p in [P17a,P17b]]
[(Fractional ideal (4*a^2 - 2*a + 1), False),
 (Fractional ideal (2*a + 1), True)]
```

has_cm()

Return whether or not this curve has a CM j-invariant.

OUTPUT:

`True` if this curve has CM over the algebraic closure of the base field, otherwise `False`. See also *cm_discriminant()* and *has_rational_cm()*.

Note: Even if E has CM in this sense (that its j-invariant is a CM j-invariant), if the associated negative discriminant D is not a square in the base field K, the extra endomorphisms will not be defined over K. See also the method *has_rational_cm()* which tests whether E has extra endomorphisms defined over K or a given extension of K.

EXAMPLES:

```
sage: EllipticCurve(j=0).has_cm()
True
sage: EllipticCurve(j=1).has_cm()
False
sage: EllipticCurve(j=1728).has_cm()
True
sage: EllipticCurve(j=8000).has_cm()
True
sage: K.<a> = QuadraticField(5)
sage: EllipticCurve(j=282880*a + 632000).has_cm()
True
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)
sage: EllipticCurve(j=31710790944000*a^2 + 39953093016000*a + 50337742902000).
↪has_cm()
True
```

has_global_minimal_model()

Return whether this elliptic curve has a global minimal model.

OUTPUT:

Boolean, True iff a global minimal model exists, i.e. an integral model which is minimal at every prime.

EXAMPLES:

```
sage: K.<a> = NumberField(x^2 - 10)
sage: E = EllipticCurve([0, 0, 0, 4536*a+14148, -163728*a-474336])
sage: E.is_global_minimal_model()
False
sage: E.has_global_minimal_model()
True
```

has_good_reduction(P)

Return True if this elliptic curve has good reduction at the prime P.

INPUT:

- P – a prime ideal of the base field of self, or a field element generating such an ideal.

OUTPUT:

(bool) – True if the curve has good reduction at P, else False.

Note: This requires determining a local integral minimal model; we do not just check that the discriminant of the current model has valuation zero.

EXAMPLES:

```
sage: E = EllipticCurve('14a1')
sage: [(p, E.has_good_reduction(p)) for p in prime_range(15)]
[(2, False), (3, True), (5, True), (7, False), (11, True), (13, True)]

sage: K.<a> = NumberField(x^3 - 2)
sage: P17a, P17b = [P for P,e in K.factor(17)]
sage: E = EllipticCurve([0,0,0,0,2*a+1])
sage: [(p, E.has_good_reduction(p)) for p in [P17a,P17b]]
[(Fractional ideal (4*a^2 - 2*a + 1), True),
 (Fractional ideal (2*a + 1), False)]
```

has_multiplicative_reduction(P)

Return True if this elliptic curve has (bad) multiplicative reduction at the prime P.

Note: See also `has_split_multiplicative_reduction()` and `has_nonsplit_multiplicative_reduction()`.

INPUT:

- P – a prime ideal of the base field of self, or a field element generating such an ideal.

OUTPUT:

(bool) True if the curve has multiplicative reduction at P, else False.

EXAMPLES:

```
sage: E = EllipticCurve('14a1')
sage: [(p, E.has_multiplicative_reduction(p)) for p in prime_range(15)]
[(2, True), (3, False), (5, False), (7, True), (11, False), (13, False)]

sage: K.<a> = NumberField(x^3 - 2)
sage: P17a, P17b = [P for P,e in K.factor(17)]
sage: E = EllipticCurve([0,0,0,0,2*a+1])
sage: [(p, E.has_multiplicative_reduction(p)) for p in [P17a,P17b]]
[(Fractional ideal (4*a^2 - 2*a + 1), False),
 (Fractional ideal (2*a + 1), False)]
```

has_nonsplit_multiplicative_reduction(P)

Return True if this elliptic curve has (bad) non-split multiplicative reduction at the prime P.

INPUT:

- P – a prime ideal of the base field of self, or a field element generating such an ideal.

OUTPUT:

(bool) True if the curve has non-split multiplicative reduction at P, else False.

EXAMPLES:

```
sage: E = EllipticCurve('14a1')
sage: [(p, E.has_nonsplit_multiplicative_reduction(p)) for p in prime_
↪range(15)]
[(2, True), (3, False), (5, False), (7, False), (11, False), (13, False)]

sage: K.<a> = NumberField(x^3 - 2)
sage: P17a, P17b = [P for P,e in K.factor(17)]
sage: E = EllipticCurve([0,0,0,0,2*a+1])
sage: [(p, E.has_nonsplit_multiplicative_reduction(p)) for p in [P17a,P17b]]
[(Fractional ideal (4*a^2 - 2*a + 1), False),
 (Fractional ideal (2*a + 1), False)]
```

has_rational_cm(*field=None*)

Return whether or not this curve has CM defined over its base field or a given extension.

INPUT:

- `field` – a field, which should be an extension of the base field of the curve. If `field` is `None` (the default), it is taken to be the base field of the curve.

OUTPUT:

`True` if the ring of endomorphisms of this curve over the given field is larger than **Z**; otherwise `False`. See also *cm_discriminant()* and *has_cm()*.

Note: If E has CM but the discriminant D is not a square in the given field K then the extra endomorphisms will not be defined over K, and this function will return `False`. See also *has_cm()*. To obtain the CM discriminant, use *cm_discriminant()*.

EXAMPLES:

```
sage: E = EllipticCurve(j=0)
sage: E.has_cm()
True
sage: E.has_rational_cm()
False
sage: D = E.cm_discriminant(); D
-3
sage: E.has_rational_cm(QuadraticField(D))
True

sage: E = EllipticCurve(j=1728)
sage: E.has_cm()
True
sage: E.has_rational_cm()
False
sage: D = E.cm_discriminant(); D
-4
sage: E.has_rational_cm(QuadraticField(D))
True
```

Higher degree examples:

```
sage: K.<a> = QuadraticField(5)
sage: E = EllipticCurve(j=282880*a + 632000)
```

(continues on next page)

(continued from previous page)

```
sage: E.has_cm()
True
sage: E.has_rational_cm()
False
sage: E.cm_discriminant()
-20
sage: E.has_rational_cm(K.extension(x^2 + 5, 'b'))
True
```

An error is raised if a field is given which is not an extension of the base field:

```
sage: E.has_rational_cm(QuadraticField(-20))
Traceback (most recent call last):
...
ValueError: Error in has_rational_cm: Number Field in a
with defining polynomial x^2 + 20 with a = 4.472135954999579?*I
is not an extension field of Number Field in a
with defining polynomial x^2 - 5 with a = 2.236067977499790?

sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)
sage: E = EllipticCurve(j=31710790944000*a^2 + 39953093016000*a +␣
↪50337742902000)
sage: E.has_cm()
True
sage: E.has_rational_cm()
False
sage: D = E.cm_discriminant(); D
-108
sage: E.has_rational_cm(K.extension(x^2 + 108,'b'))
True
```

has_split_multiplicative_reduction(*P*)

Return True if this elliptic curve has (bad) split multiplicative reduction at the prime P.

INPUT:

- `P` – a prime ideal of the base field of self, or a field element generating such an ideal.

OUTPUT:

(bool) True if the curve has split multiplicative reduction at P, else False.

EXAMPLES:

```
sage: E = EllipticCurve('14a1')
sage: [(p, E.has_split_multiplicative_reduction(p)) for p in prime_range(15)]
[(2, False), (3, False), (5, False), (7, True), (11, False), (13, False)]

sage: K.<a> = NumberField(x^3 - 2)
sage: P17a, P17b = [P for P,e in K.factor(17)]
sage: E = EllipticCurve([0,0,0,0,2*a+1])
sage: [(p, E.has_split_multiplicative_reduction(p)) for p in [P17a,P17b]]
[(Fractional ideal (4*a^2 - 2*a + 1), False),
 (Fractional ideal (2*a + 1), False)]
```

height_function()

Return the canonical height function attached to self.

EXAMPLES:

```
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^2 - 5)
sage: E = EllipticCurve(K, '11a3')
sage: E.height_function()
EllipticCurveCanonicalHeight object associated to
 Elliptic Curve defined by y^2 + y = x^3 + (-1)*x^2
 over Number Field in a with defining polynomial x^2 - 5
```

height_pairing_matrix(*points=None*, *precision=None*, *normalised=True*)

Return the height pairing matrix of the given points.

INPUT:

- `points` (list or `None` (default)) – a list of points on this curve, or `None`, in which case self.gens() will be used.
- `precision` (int or `None` (default)) – number of bits of precision of result, or `None`, for default RealField precision.
- `normalised` (bool, default `True`) – if `True`, use normalised heights which are independent of base change. Otherwise use the non-normalised Néron-Tate height, as required for the regulator in the BSD conjecture.

EXAMPLES:

```
sage: E = EllipticCurve([0, 0, 1, -1, 0])
sage: E.height_pairing_matrix()
[0.0511114082399688]
```

For rank 0 curves, the result is a valid 0x0 matrix:

```
sage: EllipticCurve('11a').height_pairing_matrix()
[]
sage: E = EllipticCurve('5077a1')
sage: E.height_pairing_matrix([E.lift_x(x) for x in [-2,-7/4,1]],␣
↪precision=100)
[  1.3685725053539301120518194471  -1.3095767070865761992624519454 -0.
↪63486715783715592064475542573]
[ -1.3095767070865761992624519454   2.7173593928122930896610589220   1.
↪09981843056672921397775714 32]
[-0.63486715783715592064475542573   1.09981843056672921397775714 32  0.
↪66820516565192793503314205089]

sage: E = EllipticCurve('389a1')
sage: E = EllipticCurve('389a1')
sage: P, Q = E.point([-1,1,1]), E.point([0,-1,1])
sage: E.height_pairing_matrix([P,Q])
[0.686667083305587 0.268478098806726]
[0.268478098806726 0.327000773651605]
```

Over a number field:

```
sage: x = polygen(QQ)
sage: K.<t> = NumberField(x^2 + 47)
sage: EK = E.base_extend(K)
sage: EK.height_pairing_matrix([EK(P),EK(Q)])
[0.686667083305587 0.268478098806726]
[0.268478098806726 0.327000773651605]
```

```
sage: K.<i> = QuadraticField(-1)
sage: E = EllipticCurve([0,0,0,i,i])
sage: P = E(-9+4*i, -18-25*i)
sage: Q = E(i,-i)
sage: E.height_pairing_matrix([P,Q])
[  2.16941934493768 -0.870059380421505]
[-0.870059380421505  0.424585837470709]
sage: E.regulator_of_points([P,Q])
0.164101403936070
```

When the parameter `normalised` is set to `False`, each height is multiplied by the degree d of the base field, and the regulator of r points is multiplied by d^r:

```
sage: E.height_pairing_matrix([P,Q], normalised=False)
[ 4.33883868987537 -1.74011876084301]
[-1.74011876084301 0.849171674941418]
sage: E.regulator_of_points([P,Q], normalised=False)
0.656405615744281
```

integral_model()

Return a model of self which is integral at all primes.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2 + 1)
sage: E = EllipticCurve([i/5,i/5,i/5,i/5,i/5])
sage: P1, P2 = K.primes_above(5)
sage: E.global_integral_model()
Elliptic Curve defined by
 y^2 + (-i)*x*y + (-25*i)*y = x^3 + 5*i*x^2 + 125*i*x + 3125*i
 over Number Field in i with defining polynomial x^2 + 1
```

Issue #7935:

```
sage: K.<a> = NumberField(x^2 - 38)
sage: E = EllipticCurve([a,1/2])
sage: E.global_integral_model()
Elliptic Curve defined by y^2 = x^3 + 1444*a*x + 27436
 over Number Field in a with defining polynomial x^2 - 38
```

Issue #9266:

```
sage: K.<s> = NumberField(x^2 - 5)
sage: w = (1+s)/2
sage: E = EllipticCurve(K, [2,w])
sage: E.global_integral_model()
Elliptic Curve defined by y^2 = x^3 + 2*x + (1/2*s+1/2)
 over Number Field in s with defining polynomial x^2 - 5
```

Issue #12151:

```
sage: K.<v> = NumberField(x^2 + 161*x - 150)
sage: E = EllipticCurve([25105/216*v - 3839/36, 634768555/7776*v - 98002625/
↪1296, 634768555/7776*v - 98002625/1296, 0, 0])
sage: M = E.global_integral_model(); M # choice varies, not tested
Elliptic Curve defined by
y^2 + (2094779518028859*v-1940492905300351)*x*y +␣
```

(continues on next page)

(continued from previous page)

```
↪(47799726847254419310117823445416530407112750 0*v-
↪44279137744134685291993077384950287195809750 0)*y = x^3 +␣
↪(26519784690047674853185542622500*v-24566525306469707225840460652500)*x^2
 over Number Field in v with defining polynomial x^2 + 161*x - 150
```

Issue #14476:

```
sage: R.<t> = QQ[]
sage: K.<g> = NumberField(t^4 - t^3 - 3*t^2 - t + 1)
sage: E = EllipticCurve([ -43/625*g^3 + 14/625*g^2 - 4/625*g + 706/625, -4862/
↪78125*g^3 - 4074/78125*g^2 - 711/78125*g + 10304/78125,  -4862/78125*g^3 -␣
↪4074/78125*g^2 - 711/78125*g + 10304/78125, 0,0])
sage: E.global_integral_model()
Elliptic Curve defined by
 y^2 + (15*g^3-48*g-42)*x*y + (-111510*g^3-162162*g^2-44145*g+37638)*y = x^3␣
↪+ (-954*g^3-1134*g^2+81*g+576)*x^2
 over Number Field in g with defining polynomial t^4 - t^3 - 3*t^2 - t + 1
```

is_Q_curve(*maxp=100*, *certificate=False*, *verbose=False*)

Return `True` if this is a **Q**-curve, with optional certificate.

INPUT:

- `maxp` (int, default 100): bound on primes used for checking necessary local conditions. The result will not depend on this, but using a larger value may return `False` faster.
- `certificate` (bool, default `False`): if `True` then a second value is returned giving a certificate for the **Q**-curve property.

OUTPUT:

If `certificate` is `False`: either `True` (if E is a **Q**-curve), or `False`.

If `certificate` is `True`: a tuple consisting of a boolean flag as before and a certificate, defined as follows:

- when the flag is `True`, so E is a **Q**-curve:
  - either {'CM':D} where D is a negative discriminant, when E has potential CM with discriminant D;
  - otherwise {'CM': 0, 'core_poly': f, 'rho': ρ, 'r': r, 'N': N}, when E is a non-CM **Q**-curve, where the core polynomial f is an irreducible monic polynomial over QQ of degree 2^ρ, all of whose roots are j-invariants of curves isogenous to E, the core level N is a square-free integer with r prime factors which is the LCM of the degrees of the isogenies between these conjugates. For example, if there exists a curve E' isogenous to E with $j(E') = j \in \mathbf{Q}$, then the certificate is {'CM':0, 'r':0, 'rho':0, 'core_poly': x-j, 'N':1}.
- when the flag is `False`, so E is not a **Q**-curve, the certificate is a prime p such that the reductions of E at the primes dividing p are inconsistent with the property of being a **Q**-curve. See the documentation for `sage.src.schemes.elliptic_curves.Qcurves.is_Q_curve()` for details.

ALGORITHM:

See the documentation for `sage.src.schemes.elliptic_curves.Qcurves.is_Q_curve()`, and [CrNa2020] for details.

EXAMPLES:

A non-CM curve over **Q** and a CM curve over **Q** are both trivially **Q**-curves:

```
sage: E = EllipticCurve([1,2,3,4,5])
sage: flag, cert = E.is_Q_curve(certificate=True)
sage: flag
True
sage: cert
{'CM': 0, 'N': 1, 'core_poly': x, 'r': 0, 'rho': 0}

sage: E = EllipticCurve(j=8000)
sage: flag, cert = E.is_Q_curve(certificate=True)
sage: flag
True
sage: cert
{'CM': -8}
```

A non-$\mathbf{Q}$-curve over a quartic field. The local data at bad primes above 3 is inconsistent:

```
sage: R.<x> = PolynomialRing(QQ)
sage: K.<a> = NumberField(R([3, 0, -5, 0, 1]))
sage: E = EllipticCurve([K([-3,-4,1,1]), K([4,-1,-1,0]), K([-2,0,1,0]), K([-
↪621,778,138,-178]), K([9509,2046,-24728,10380])])
sage: E.is_Q_curve(certificate=True, verbose=True)
Checking whether Elliptic Curve defined by y^2 + (a^3+a^2-4*a-3)*x*y + (a^2-
↪2)*y = x^3 + (-a^2-a+4)*x^2 + (-178*a^3+138*a^2+778*a-621)*x + (10380*a^3-
↪24728*a^2+2046*a+9509) over Number Field in a with defining polynomial x^4 -
↪ 5*x^2 + 3 is a Q-curve
No: inconsistency at the 2 primes dividing 3
- potentially multiplicative: [True, False]
(False, 3)
```

A non-$\mathbf{Q}$-curve over a quadratic field. The local data at bad primes is consistent, but the local test at good primes above 13 is not:

```
sage: K.<a> = NumberField(R([-10, 0, 1]))
sage: E = EllipticCurve([K([0,1]), K([-1,-1]), K([0,0]), K([-236,40]), K([-
↪1840,464])])
sage: E.is_Q_curve(certificate=True, verbose=True)
Checking whether Elliptic Curve defined by y^2 + a*x*y = x^3 + (-a-1)*x^2 +␣
↪(40*a-236)*x + (464*a-1840) over Number Field in a with defining polynomial␣
↪x^2 - 10 is a Q-curve
Applying local tests at good primes above p<=100
No: inconsistency at the 2 ordinary primes dividing 13
- Frobenius discriminants mod squares: [-1, -3]
No: local test at p=13 failed
(False, 13)
```

A quadratic $\mathbf{Q}$-curve with CM discriminant -15 (so the j-invariant is not in $\mathbf{Q}$):

```
sage: R.<x> = PolynomialRing(QQ)
sage: K.<a> = NumberField(R([-1, -1, 1]))
sage: E = EllipticCurve([K([1,0]), K([-1,0]), K([0,1]), K([0,-2]), K([0,1])])
sage: E.is_Q_curve(certificate=True, verbose=True)
Checking whether Elliptic Curve defined by y^2 + x*y + a*y = x^3 + (-1)*x^2 +␣
↪(-2*a)*x + a over Number Field in a with defining polynomial x^2 - x - 1 is␣
↪a Q-curve
Yes: E is CM (discriminant -15)
(True, {'CM': -15})
```

An example over $\mathbf{Q}(\sqrt{2},\sqrt{3})$. The j-invariant is in $\mathbf{Q}(\sqrt{6})$, so computations will be done over that field, and

in fact there is an isogenous curve with rational j, so we have a so-called rational $\mathbf{Q}$-curve:

```
sage: K.<a> = NumberField(R([1, 0, -4, 0, 1]))
sage: E = EllipticCurve([K([-2,-4,1,1]), K([0,1,0,0]), K([0,1,0,0]), K([-4780,
↪9170,1265,-2463]), K([163923,-316598,-43876,84852])])
sage: flag, cert = E.is_Q_curve(certificate=True) # long time
sage: flag # long time
True
sage: cert # long time
{'CM': 0, 'N': 1, 'core_degs': [1], 'core_poly': x - 85184/3, 'r': 0, 'rho':␣
↪0}
```

Over the same field, a so-called strict $\mathbf{Q}$-curve which is not isogenous to one with rational j, but whose core field is quadratic. In fact the isogeny class over K consists of 6 curves, four with conjugate quartic j-invariants and 2 with quadratic conjugate j-invariants in $\mathbf{Q}(\sqrt{3})$ (but which are not base-changes from the quadratic subfield):

```
sage: E = EllipticCurve([K([0,-3,0,1]), K([1,4,0,-1]), K([0,0,0,0]), K([-2,-
↪16,0,4]), K([-19,-32,4,8])])
sage: flag, cert = E.is_Q_curve(certificate=True) # long time
sage: flag # long time
True
sage: cert # long time
{'CM': 0,
 'N': 2,
 'core_degs': [1, 2],
 'core_poly': x^2 - 840064*x + 1593413632,
 'r': 1,
 'rho': 1}
```

is_global_integral_model()

Return whether `self` is integral at all primes.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2 + 1)
sage: E = EllipticCurve([i/5,i/5,i/5,i/5,i/5])
sage: P1, P2 = K.primes_above(5)
sage: Emin = E.global_integral_model()
sage: Emin.is_global_integral_model()
True
```

is_global_minimal_model()

Return whether this elliptic curve is a global minimal model.

OUTPUT:

Boolean, False if E is not integral, or if E is non-minimal at some prime, else True.

EXAMPLES:

```
sage: K.<a> = NumberField(x^2 - 10)
sage: E = EllipticCurve([0, 0, 0, -22500, 750000*a])
sage: E.is_global_minimal_model()
False
sage: E.non_minimal_primes()
[Fractional ideal (2, a), Fractional ideal (5, a)]
```

(continues on next page)

(continued from previous page)

```
sage: E = EllipticCurve([0, 0, 0, -3024, 46224])
sage: E.is_global_minimal_model()
False
sage: E.non_minimal_primes()
[2, 3]
sage: Emin = E.global_minimal_model()
sage: Emin.is_global_minimal_model()
True
```

A necessary condition to be a global minimal model is that the model must be globally integral:

```
sage: E = EllipticCurve([0,0,0,1/2,1/3])
sage: E.is_global_minimal_model()
False
sage: Emin.is_global_minimal_model()
True
sage: Emin.ainvs()
(0, 1, 1, -2, 0)
```

is_isogenous (*other*, *proof=True*, *maxnorm=100*)

Return whether or not self is isogenous to other.

INPUT:

- `other` – another elliptic curve.
- `proof` (default: `True`) – If `False`, the function will return `True` whenever the two curves have the same conductor and are isogenous modulo p for all primes p of norm up to `maxnorm`. If `True`, the function returns False when the previous condition does not hold, and if it does hold we compute the complete isogeny class to see if the curves are indeed isogenous.
- `maxnorm` (integer, default 100) – The maximum norm of primes p for which isogeny modulo p will be checked.

OUTPUT:

(bool) True if there is an isogeny from curve `self` to curve `other`.

EXAMPLES:

```
sage: x = polygen(QQ, 'x')
sage: F = NumberField(x^2 - 2, 's'); F
Number Field in s with defining polynomial x^2 - 2
sage: E1 = EllipticCurve(F, [7,8])
sage: E2 = EllipticCurve(F, [0,5,0,1,0])
sage: E3 = EllipticCurve(F, [0,-10,0,21,0])
sage: E1.is_isogenous(E2)
False
sage: E1.is_isogenous(E1)
True
sage: E2.is_isogenous(E2)
True
sage: E2.is_isogenous(E1)
False
sage: E2.is_isogenous(E3)
True
```

```
sage: x = polygen(QQ, 'x')
sage: F = NumberField(x^2 - 2, 's'); F
Number Field in s with defining polynomial x^2 - 2
sage: E = EllipticCurve('14a1')
sage: EE = EllipticCurve('14a2')
sage: E1 = E.change_ring(F)
sage: E2 = EE.change_ring(F)
sage: E1.is_isogenous(E2)
True
```

```
sage: x = polygen(QQ, 'x')
sage: F = NumberField(x^2 - 2, 's'); F
Number Field in s with defining polynomial x^2 - 2
sage: k.<a> = NumberField(x^3 + 7)
sage: E = EllipticCurve(F, [7,8])
sage: EE = EllipticCurve(k, [2, 2])
sage: E.is_isogenous(EE)
Traceback (most recent call last):
...
ValueError: Second argument must be defined over the same number field.
```

Some examples from Cremona's 1981 tables:

```
sage: K.<i> = QuadraticField(-1)
sage: E1 = EllipticCurve([i + 1, 0, 1, -240*i - 400, -2869*i - 2627])
sage: E1.conductor()
Fractional ideal (-4*i - 7)
sage: E2 = EllipticCurve([1+i,0,1,0,0])
sage: E2.conductor()
Fractional ideal (-4*i - 7)
sage: E1.is_isogenous(E2) # long time
True
sage: E1.is_isogenous(E2, proof=False) # faster  (~170ms)
True
```

In this case E1 and E2 are in fact 9-isogenous, as may be deduced from the following:

```
sage: E3 = EllipticCurve([i + 1, 0, 1, -5*i - 5, -2*i - 5])
sage: E3.is_isogenous(E1)
True
sage: E3.is_isogenous(E2)
True
sage: E1.isogeny_degree(E2) # long time
9
```

is_local_integral_model(*P)

Tests if self is integral at the prime ideal P, or at all the primes if P is a list or tuple.

INPUT:

- `*P` – a prime ideal, or a list or tuple of primes.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2 + 1)
sage: P1, P2 = K.primes_above(5)
sage: E = EllipticCurve([i/5,i/5,i/5,i/5,i/5])
sage: E.is_local_integral_model(P1, P2)
```

(continues on next page)

(continued from previous page)

```
False
sage: Emin = E.local_integral_model(P1, P2)
sage: Emin.is_local_integral_model(P1, P2)
True
```

isogenies_prime_degree (*l=None*, *algorithm='Billerey'*, *minimal_models=True*)

Return a list of ℓ-isogenies from self, where ℓ is a prime.

INPUT:

- `l` – either None or a prime or a list of primes.
- `algorithm` (string, default 'Billerey') – the algorithm to use to compute the reducible primes when `l` is None. Ignored for CM curves or if `l` is provided. Values are 'Billerey' (default), 'Larson', and 'heuristic'.
- `minimal_models` (bool, default `True`) – if `True`, all curves computed will be minimal or semi-minimal models. Over fields of larger degree it can be expensive to compute these so set to `False`.

OUTPUT:

(list) ℓ-isogenies for the given ℓ or if ℓ is None, all isogenies of prime degree (see below for the CM case).

Note: Over **Q**, the codomains of the isogenies returned are standard minimal models. Over other number fields they are global minimal models if these exist, otherwise models which are minimal at all but one prime.

Note: For curves with rational CM, isogenies of primes degree exist for infinitely many primes ℓ, though there are only finitely many isogenous curves up to isomorphism. The list returned only includes one isogeny of prime degree for each codomain.

EXAMPLES:

```
sage: K.<i> = QuadraticField(-1)
sage: E = EllipticCurve(K, [0,0,0,0,1])
sage: isogs = E.isogenies_prime_degree()
sage: [phi.degree() for phi in isogs]
[2, 3]

sage: pol = PolynomialRing(QQ,'x')([1,-3,5,-5,5,-3,1])
sage: L.<a> = NumberField(pol)
sage: js = hilbert_class_polynomial(-23).roots(L, multiplicities=False);␣
↪len(js)
3
sage: E = EllipticCurve(j=js[0])
sage: len(E.isogenies_prime_degree())  # long time
3
```

Set `minimal_models` to False to avoid computing minimal models of the isogenous curves, since that can be time-consuming since it requires computation of the class group:

```
sage: proof.number_field(False)
sage: K.<z> = CyclotomicField(53)
sage: E = EllipticCurve(K, [0,6,0,2,0])
sage: E.isogenies_prime_degree(2, minimal_models=False)
[Isogeny of degree 2
  from Elliptic Curve defined by y^2 = x^3 + 6*x^2 + 2*x
```

(continues on next page)

(continued from previous page)

```
      over Cyclotomic Field of order 53 and degree 52
   to Elliptic Curve defined by y^2 = x^3 + 6*x^2 + (-8)*x + (-48)
      over Cyclotomic Field of order 53 and degree 52]
sage: E.isogenies_prime_degree(2, minimal_models=True) # not tested (10s)
[Isogeny of degree 2
  from Elliptic Curve defined by y^2 = x^3 + 6*x^2 + 2*x
      over Cyclotomic Field of order 53 and degree 52
   to Elliptic Curve defined by y^2 = x^3 + (-20)*x + (-16)
      over Cyclotomic Field of order 53 and degree 52]
```

isogeny_class(*reducible_primes=None*, *algorithm='Billerey'*, *minimal_models=True*)

Return the isogeny class of this elliptic curve.

INPUT:

- `reducible_primes` (list of ints, or None (default)) – if not None then this should be a list of primes; in computing the isogeny class, only composites isogenies of these degrees will be used.
- `algorithm` (string, default `'Billerey'`) – the algorithm to use to compute the reducible primes. Ignored for CM curves or if `reducible_primes` is provided. Values are `'Billerey'` (default), `'Larson'`, and `'heuristic'`.
- `minimal_models` (bool, default `True`) – if `True`, all curves in the class will be minimal or semi-minimal models. Over fields of larger degree it can be expensive to compute these so set to `False`.

OUTPUT:

An instance of the class *sage.schemes.elliptic_curves.isogeny_class.IsogenyClass_EC_NumberField*. From this object may be obtained a list of curves in the class, a matrix of the degrees of the isogenies between them, and the isogenies themselves.

Note: If using the algorithm `'heuristic'` for non-CM curves, the result is not guaranteed to be the complete isogeny class, since only reducible primes up to the default bound in `reducible_primes_naive()` (currently 1000) are tested. However, no examples of non-CM elliptic curves with reducible primes greater than 100 have yet been computed so the output is likely to be correct.

Note: By default, the curves in the isogeny class will all be minimal models if these exist (for example, when the class number is 1); otherwise they will be minimal at all but one prime. This behaviour can be switched off if desired, for example over fields where the computation of the class group would be too expensive.

EXAMPLES:

```
sage: K.<i> = QuadraticField(-1)
sage: E = EllipticCurve(K, [0,0,0,0,1])
sage: C = E.isogeny_class(); C
Isogeny class of Elliptic Curve defined by y^2 = x^3 + 1
 over Number Field in i with defining polynomial x^2 + 1 with i = 1*I
```

The curves in the class (sorted):

```
sage: [E1.ainvs() for E1 in C]
[(0, 0, 0, 0, -27),
(0, 0, 0, 0, 1),
(i + 1, i, i + 1, -i + 3, 4*i),
(i + 1, i, i + 1, -i + 33, -58*i)]
```

The matrix of degrees of cyclic isogenies between curves:

```
sage: C.matrix()
[1 3 6 2]
[3 1 2 6]
[6 2 1 3]
[2 6 3 1]
```

The array of isogenies themselves is not filled out but only contains those used to construct the class, the other entries containing the integer 0. This will be changed when the class *EllipticCurveIsogeny* allowed composition. In this case we used 2-isogenies to go from 0 to 2 and from 1 to 3, and 3-isogenies to go from 0 to 1 and from 2 to 3:

```
sage: isogs = C.isogenies()
sage: [((i,j), isogs[i][j].degree())
....:  for i in range(4) for j in range(4) if isogs[i][j] != 0]
[((0, 1), 3),
 ((0, 3), 2),
 ((1, 0), 3),
 ((1, 2), 2),
 ((2, 1), 2),
 ((2, 3), 3),
 ((3, 0), 2),
 ((3, 2), 3)]
sage: [((i,j), isogs[i][j].x_rational_map())
....:  for i in range(4) for j in range(4) if isogs[i][j] != 0]
[((0, 1), (1/9*x^3 - 12)/x^2),
 ((0, 3), (-1/2*i*x^2 + i*x - 12*i)/(x - 3)),
 ((1, 0), (x^3 + 4)/x^2),
 ((1, 2), (-1/2*i*x^2 - i*x - 2*i)/(x + 1)),
 ((2, 1), (1/2*i*x^2 - x)/(x + 3/2*i)),
 ((2, 3), (x^3 + 4*i*x^2 - 10*x - 10*i)/(x^2 + 4*i*x - 4)),
 ((3, 0), (1/2*i*x^2 + x + 4*i)/(x - 5/2*i)),
 ((3, 2), (1/9*x^3 - 4/3*i*x^2 - 34/3*x + 226/9*i)/(x^2 - 8*i*x - 16))]
```

The isogeny class may be visualized by obtaining its graph and plotting it:

```
sage: G = C.graph()
sage: G.show(edge_labels=True) # long time

sage: K.<i> = QuadraticField(-1)
sage: E = EllipticCurve([1+i, -i, i, 1, 0])
sage: C = E.isogeny_class(); C # long time
Isogeny class of
 Elliptic Curve defined by y^2 + (i+1)*x*y + i*y = x^3 + (-i)*x^2 + x
  over Number Field in i with defining polynomial x^2 + 1 with i = 1*I
sage: len(C) # long time
6
sage: C.matrix() # long time
[ 1  3  9 18  6  2]
[ 3  1  3  6  2  6]
[ 9  3  1  2  6 18]
[18  6  2  1  3  9]
[ 6  2  6  3  1  3]
[ 2  6 18  9  3  1]
sage: [E1.ainvs() for E1 in C] # long time
[(i + 1, i - 1, i, -i - 1, -i + 1),
 (i + 1, i - 1, i, 14*i + 4, 7*i + 14),
```

(continues on next page)

(continued from previous page)

```
 (i + 1, i - 1, i, 59*i + 99, 372*i - 410),
 (i + 1, -i, i, -240*i - 399, 2869*i + 2627),
 (i + 1, -i, i, -5*i - 4, 2*i + 5),
 (i + 1, -i, i, 1, 0)]
```

An example with CM by $\sqrt{-5}$:

```
sage: pol = PolynomialRing(QQ,'x')([1,0,3,0,1])
sage: K.<c> = NumberField(pol)
sage: j = 1480640 + 565760*c^2
sage: E = EllipticCurve(j=j)
sage: E.has_cm()
True
sage: E.has_rational_cm()
True
sage: E.cm_discriminant()
-20
sage: C = E.isogeny_class()
sage: len(C)
2
sage: C.matrix()
[1 2]
[2 1]
sage: [E.ainvs() for E in C]
[(0, 0, 0, 83490*c^2 - 147015, -64739840*c^2 - 84465260),
(0, 0, 0, -161535*c^2 + 70785, -62264180*c^3 + 6229080*c)]
sage: C.isogenies()[0][1]
Isogeny of degree 2
 from Elliptic Curve defined by
      y^2 = x^3 + (83490*c^2-147015)*x + (-64739840*c^2-84465260)
      over Number Field in c with defining polynomial x^4 + 3*x^2 + 1
   to Elliptic Curve defined by
      y^2 = x^3 + (-161535*c^2+70785)*x + (-62264180*c^3+6229080*c)
      over Number Field in c with defining polynomial x^4 + 3*x^2 + 1
```

An example with CM by $\sqrt{-23}$ (class number 3):

```
sage: pol = PolynomialRing(QQ,'x')([1,-3,5,-5,5,-3,1])
sage: L.<a> = NumberField(pol)
sage: js = hilbert_class_polynomial(-23).roots(L,multiplicities=False);␣
↪len(js)
3
sage: E = EllipticCurve(j=js[0])
sage: E.has_rational_cm()
True
sage: len(E.isogenies_prime_degree())  # long time
3
sage: C = E.isogeny_class(); len(C)  # long time
6
```

The reason for the isogeny class having size six while the class number is only 3 is that the class also contains three curves with CM by the order of discriminant $-92 = 4 \cdot (-23)$, which also has class number 3. The curves in the class are sorted first by CM discriminant (then lexicographically using a-invariants):

```
sage: [F.cm_discriminant() for F in C]  # long time
[-23, -23, -23, -92, -92, -92]
```

2 splits in the order with discriminant -23, into two primes of order 3 in the class group, each of which induces a 2-isogeny to a curve with the same endomorphism ring; the third 2-isogeny is to a curve with the smaller endomorphism ring:

```
sage: [phi.codomain().cm_discriminant() for phi in E.isogenies_prime_
↪degree()]  # long time
[-92, -23, -23]

sage: C.matrix()  # long time # random
[1 2 2 4 4 2]
[2 1 2 4 2 4]
[2 2 1 2 4 4]
[4 4 2 1 3 3]
[4 2 4 3 1 3]
[2 4 4 3 3 1]
```

The graph of this isogeny class has a shape which does not occur over $\mathbf{Q}$: a triangular prism. Note that for curves without CM, the graph has an edge between two curves if and only if they are connected by an isogeny of prime degree, and this degree is uniquely determined by the two curves, but in the CM case this property does not hold, since for pairs of curves in the class with the same endomorphism ring O, the set of degrees of isogenies between them is the set of integers represented by a primitive integral binary quadratic form of discriminant $\mathrm{disc}(O)$, and this form represents infinitely many primes. In the matrix we give a small prime represented by the appropriate form. In this example, the matrix is formed by four 3×3 blocks. The isogenies of degree 2 indicated by the upper left 3×3 block of the matrix could be replaced by isogenies of any degree represented by the quadratic form $2x^2 + xy + 3y^2$ of discriminant -23. Similarly in the lower right block, the entries of 3 could be represented by any integers represented by the quadratic form $3x^2 + 2xy + 8y^2$ of discriminant -92. In the top right block and lower left blocks, by contrast, the prime entries 2 are unique determined:

```
sage: G = C.graph()  # long time
sage: G.adjacency_matrix()  # long time # random
[0 1 1 0 0 1]
[1 0 1 0 1 0]
[1 1 0 1 0 0]
[0 0 1 0 1 1]
[0 1 0 1 0 1]
[1 0 0 1 1 0]
sage: Graph(polytopes.simplex(2).prism().adjacency_matrix()).is_isomorphic(G)
↪# long time
True
```

To display the graph without any edge labels:

```
sage: G.show()  # not tested
```

To display the graph with edge labels: by default, for curves with rational CM, the labels are the coefficients of the associated quadratic forms:

```
sage: G.show(edge_labels=True)  # not tested
```

For an alternative view, first relabel the edges using only 2 labels to distinguish between isogenies between curves with the same endomorphism ring and isogenies between curves with different endomorphism rings, then use a 3-dimensional plot which can be rotated:

```
sage: for i, j, l in G.edge_iterator():  # long time
....:     G.set_edge_label(i, j, l.count(','))
sage: G.show3d(color_by_label=True)  # long time
```

A class number 6 example. First we set up the fields: `pol` defines the same field as `pol26` but is simpler:

```
sage: pol26 = hilbert_class_polynomial(-4*26)
sage: pol = x^6 - x^5 + 2*x^4 + x^3 - 2*x^2 - x - 1
sage: K.<a> = NumberField(pol)
sage: L.<b> = K.extension(x^2 + 26)
```

Only 2 of the j-invariants with discriminant -104 are in K, though all are in L:

```
sage: len(pol26.roots(K))
2
sage: len(pol26.roots(L))
6
```

We create an elliptic curve defined over K with one of the j-invariants in K:

```
sage: j1 = pol26.roots(K)[0][0]
sage: E = EllipticCurve(j=j1)
sage: E.has_cm()
True
sage: E.has_rational_cm()
False
sage: E.has_rational_cm(L)
True
```

Over K the isogeny class has size 4, with 2 curves for each of the 2 K-rational j-invariants:

```
sage: C = E.isogeny_class(); len(C)  # long time (~11s)
4
sage: C.matrix()                     # long time
[ 1 13  2 26]
[13  1 26  2]
[ 2 26  1 13]
[26  2 13  1]
sage: len(Set([EE.j_invariant() for EE in C.curves]))  # long time
2
```

Over L, the isogeny class grows to size 6 (the class number):

```
sage: EL = E.change_ring(L)
sage: CL = EL.isogeny_class(minimal_models=False) # long time
sage: len(CL) # long time
6
sage: s1 = Set([EE.j_invariant() for EE in CL.curves]) # long time
sage: s2 = Set(pol26.roots(L, multiplicities=False)) # long time
sage: s1 == s2 # long time
True
```

In each position in the matrix of degrees, we see primes (or 1). In fact the set of degrees of cyclic isogenies from curve i to curve j is infinite, and is the set of all integers represented by one of the primitive binary quadratic forms of discriminant -104, from which we have selected a small prime:

```
sage: CL.matrix() # long time # random (see :issue:`19229`)
[1 2 3 3 5 5]
[2 1 5 5 3 3]
[3 5 1 3 2 5]
[3 5 3 1 5 2]
```

(continues on next page)

(continued from previous page)

```
[5 3 2 5 1 3]
[5 3 5 2 3 1]
```

To see the array of binary quadratic forms:

```
sage: CL.qf_matrix()  # long time # random (see :issue:`19229`)
[[[1], [2, 0, 13], [3, -2, 9], [3, -2, 9], [5, -4, 6], [5, -4, 6]],
 [[2, 0, 13], [1], [5, -4, 6], [5, -4, 6], [3, -2, 9], [3, -2, 9]],
 [[3, -2, 9], [5, -4, 6], [1], [3, -2, 9], [2, 0, 13], [5, -4, 6]],
 [[3, -2, 9], [5, -4, 6], [3, -2, 9], [1], [5, -4, 6], [2, 0, 13]],
 [[5, -4, 6], [3, -2, 9], [2, 0, 13], [5, -4, 6], [1], [3, -2, 9]],
 [[5, -4, 6], [3, -2, 9], [5, -4, 6], [2, 0, 13], [3, -2, 9], [1]]]
```

As in the non-CM case, the isogeny class may be visualized by obtaining its graph and plotting it. Since there are more edges than in the non-CM case, it may be preferable to omit the edge_labels:

```
sage: G = C.graph()
sage: G.show(edge_labels=False) # long time
```

It is possible to display a 3-dimensional plot, with colours to represent the different edge labels, in a form which can be rotated!:

```
sage: G.show3d(color_by_label=True) # long time
```

Over larger number fields several options make computations tractable. Here we use algorithm 'heuristic' which avoids a rigorous computation of the reducible primes, only testing those less than 1000, and setting `minimal_models` to `False` avoid having to compute the class group of K. To obtain minimal models set proof.number_field(False); the class group computation takes an additional 10s:

```
sage: K.<z> = CyclotomicField(53)
sage: E = EllipticCurve(K,[0,6,0,2,0])
sage: C = E.isogeny_class(algorithm='heuristic', minimal_models=False); C #␣
↪long time (10s)
Isogeny class of Elliptic Curve defined by y^2 = x^3 + 6*x^2 + 2*x
 over Cyclotomic Field of order 53 and degree 52
sage: C.curves # long time
[Elliptic Curve defined by y^2 = x^3 + 6*x^2 + (-8)*x + (-48)
  over Cyclotomic Field of order 53 and degree 52,
 Elliptic Curve defined by y^2 = x^3 + 6*x^2 + 2*x
  over Cyclotomic Field of order 53 and degree 52]
```

isogeny_degree(*other*)

Return the minimal degree of an isogeny between self and other, or 0 if no isogeny exists.

INPUT:

- `other` – another elliptic curve.

OUTPUT:

(int) The degree of an isogeny from `self` to `other`, or 0.

EXAMPLES:

```
sage: x = QQ['x'].0
sage: F = NumberField(x^2 - 2, 's'); F
Number Field in s with defining polynomial x^2 - 2
```

(continues on next page)

(continued from previous page)

```
sage: E = EllipticCurve('14a1')
sage: EE = EllipticCurve('14a2')
sage: E1 = E.change_ring(F)
sage: E2 = EE.change_ring(F)
sage: E1.isogeny_degree(E2)  # long time
2
sage: E2.isogeny_degree(E2)
1
sage: E5 = EllipticCurve('14a5').change_ring(F)
sage: E1.isogeny_degree(E5)  # long time
6

sage: E = EllipticCurve('11a1')
sage: [E2.label() for E2 in cremona_curves([11..20]) if E.isogeny_degree(E2)]
['11a1', '11a2', '11a3']

sage: K.<i> = QuadraticField(-1)
sage: E = EllipticCurve([1+i, -i, i, 1, 0])
sage: C = E.isogeny_class() # long time
sage: [E.isogeny_degree(F) for F in C] # long time
[2, 6, 18, 9, 3, 1]
```

kodaira_symbol(*P*, *proof=None*)

Return the Kodaira Symbol of this elliptic curve at the prime P.

INPUT:

- `P` – either None or a prime ideal of the base field of self.
- `proof` – whether to only use provably correct methods (default controlled by global proof module). Note that the proof module is number_field, not elliptic_curves, since the functions that actually need the flag are in number fields.

OUTPUT:

The Kodaira Symbol of the curve at `P`, represented as a string.

EXAMPLES:

```
sage: K.<a> = NumberField(x^2 - 5)
sage: E = EllipticCurve([20, 225, 750, 625*a + 6875, 31250*a + 46875])
sage: bad_primes = E.discriminant().support(); bad_primes
[Fractional ideal (-a), Fractional ideal (7/2*a - 81/2),
 Fractional ideal (-a - 52), Fractional ideal (2)]
sage: [E.kodaira_symbol(P) for P in bad_primes]
[I0, I1, I1, II]
sage: K.<a> = QuadraticField(-11)
sage: E = EllipticCurve('11a1').change_ring(K)
sage: [E.kodaira_symbol(P) for P in K(11).support()]
[I10]
```

lll_reduce(*points*, *height_matrix=None*, *precision=None*)

Return an LLL-reduced basis from a given basis, with transform matrix.

INPUT:

- `points` – a list of points on this elliptic curve, which should be independent.
- `height_matrix` – the height-pairing matrix of the points, or `None`. If `None`, it will be computed.

- `precision` – number of bits of precision of intermediate computations (default: `None`, for default RealField precision; ignored if `height_matrix` is supplied)

OUTPUT: A tuple (newpoints, U) where U is a unimodular integer matrix, new_points is the transform of points by U, such that new_points has LLL-reduced height pairing matrix

---

Note: If the input points are not independent, the output depends on the undocumented behaviour of PARI's pari:qflllgram function when applied to a Gram matrix which is not positive definite.

EXAMPLES:

Some examples over **Q**:

```
sage: E = EllipticCurve([0, 1, 1, -2, 42])
sage: Pi = E.gens(); Pi
[(-4 : 1 : 1), (-3 : 5 : 1), (-11/4 : 43/8 : 1), (-2 : 6 : 1)]
sage: Qi, U = E.lll_reduce(Pi)
sage: all(sum(U[i,j]*Pi[i] for i in range(4)) == Qi[j] for j in range(4))
True
sage: sorted(Qi)
[(-4 : 1 : 1), (-3 : 5 : 1), (-2 : 6 : 1), (0 : 6 : 1)]
sage: U.det()
1
sage: E.regulator_of_points(Pi)
4.59088036960573
sage: E.regulator_of_points(Qi)
4.59088036960574
```

```
sage: E = EllipticCurve([1,0,1,-120039822036992245303534619191166796374,
↪5042249924849106700108017991680827267594437562229114151116])
sage: xi = [2005024558054813068,
....:       -4690836759490453344,
....:       4700156326649806635,
....:       6785546256295273860,
....:       6823803569166584943,
....:       7788809602110240789,
....:       27385442304350994620556,
....:       54284682060285253719/4,
....:       -9420023526039507513 9/25,
....:       -34636610553318417246 47/576,
....:       -66840659340335069706 37/676,
....:       -95607738619264034419 8/2209,
....:       -27067471797013364392578/2809,
....:       -25538866857137199063309/3721,
....:       -10263250117602590518943 31/108241,
....:       93513612307294812506273 34/1366561,
....:       10100878635879432897339615/1423249,
....:       11499655868211022625340735/17522596,
....:       11035225366508100251781 1734/21353641,
....:       41428009642603309414366 8538257/285204544,
....:       36101712290699828042930087436/4098432361,
....:       45442463408503524215460183165/5424617104,
....:       98388601334470070767858 7482584/141566320009,
....:       11246143357168510532811 76544216033/152487126016]
sage: points = [E.lift_x(x) for x in xi]
sage: newpoints, U = E.lll_reduce(points)  # long time (35s on sage.math,␣
↪2011)
```

(continues on next page)

(continued from previous page)

```
sage: [P[0] for P in newpoints]             # long time
[6823803569166584943, 5949539878899294213, 2005024558054813068,␣
↪5864879778877955778, 23955263915878682727/4, 5922188321411938518,␣
↪5286988283823825378, 11465667352242779838, -11451575907286171572,␣
↪3502708072571012181, 15001439351832387091 84/225, 27180522378120223419/4, -
↪58118741641906044615 81/625, 26807786527159569093, 7041412654828066743,␣
↪475656155255883588, 2657574547267660178 91/49, 7272142121019825303,␣
↪5062867917383369341 5/4, 6951643522366348968, 6842515151518070703,␣
↪1115937503896508468 85/16, 2607467890531740394315/9, -1829928525835506297]
```

An example to show the explicit use of the height pairing matrix:

```
sage: E = EllipticCurve([0, 1, 1, -2, 42])
sage: Pi = E.gens()
sage: H = E.height_pairing_matrix(Pi,3)
sage: E.lll_reduce(Pi,height_matrix=H)
(
                                                                [1 0 0 1]
                                                                [0 1 0 1]
                                                                [0 0 0 1]
[(-4 : 1 : 1), (-3 : 5 : 1), (-2 : 6 : 1), (1 : -7 : 1)], [0 0 1 1]
)
```

Some examples over number fields (see Issue #9411):

```
sage: K.<a> = QuadraticField(-23, 'a')
sage: E = EllipticCurve(K, [0,0,1,-1,0])
sage: P = E(-2, -(a+1)/2)
sage: Q = E(0,-1)
sage: E.lll_reduce([P,Q])
(
                                          [0 1]
[(0 : -1 : 1), (-2 : -1/2*a - 1/2 : 1)], [1 0]
)
```

```
sage: K.<a> = QuadraticField(-5)
sage: E = EllipticCurve(K, [0,a])
sage: points = [E.point([-211/841*a - 6044/841,-209584/24389*a + 53634/
↪24389]),
....:           E.point([-17/18*a - 1/9, -109/108*a - 277/108])]
sage: E.lll_reduce(points)
(
[(-a + 4 : -3*a + 7 : 1), (-17/18*a - 1/9 : 109/108*a + 277/108 : 1)],
[ 1  0]
[ 1 -1]
)
```

local_data(*P=None*, *proof=None*, *algorithm='pari'*, *globally=False*)

Local data for this elliptic curve at the prime P.

INPUT:

- `P` – either None, a prime ideal of the base field of self, or an element of the base field that generates a prime ideal.
- `proof` – whether to only use provably correct methods (default controlled by global proof module). Note that the proof module is number_field, not elliptic_curves, since the functions that actually need the flag are in number fields.

- `algorithm` (string, default: "pari") – Ignored unless the base field is **Q**. If "pari", use the PARI C-library pari:ellglobalred implementation of Tate's algorithm over **Q**. If "generic", use the general number field implementation.
- `globally` – whether the local algorithm uses global generators for the prime ideals. Default is False, which will not require any information about the class group. If True, a generator for P will be used if P is principal. Otherwise, or if `globally` is False, the minimal model returned will preserve integrality at other primes, but not minimality.

OUTPUT:

If P is specified, returns the `EllipticCurveLocalData` object associated to the prime P for this curve. Otherwise, returns a list of such objects, one for each prime P in the support of the discriminant of this model.

Note: The model is not required to be integral on input.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2 + 1)
sage: E = EllipticCurve([1 + i, 0, 1, 0, 0])
sage: E.local_data()
[Local data at Fractional ideal (2*i + 1):
   Reduction type: bad non-split multiplicative
   Local minimal model: Elliptic Curve defined by y^2 + (i+1)*x*y + y = x^3
                        over Number Field in i with defining polynomial x^2 +␣
↪1
   Minimal discriminant valuation: 1
   Conductor exponent: 1
   Kodaira Symbol: I1
   Tamagawa Number: 1,
 Local data at Fractional ideal (-2*i + 3):
   Reduction type: bad split multiplicative
   Local minimal model: Elliptic Curve defined by y^2 + (i+1)*x*y + y = x^3
                        over Number Field in i with defining polynomial x^2 +␣
↪1
   Minimal discriminant valuation: 2
   Conductor exponent: 1
   Kodaira Symbol: I2
   Tamagawa Number: 2]
sage: E.local_data(K.ideal(3))
Local data at Fractional ideal (3):
  Reduction type: good
  Local minimal model: Elliptic Curve defined by y^2 + (i+1)*x*y + y = x^3
                       over Number Field in i with defining polynomial x^2 + 1
  Minimal discriminant valuation: 0
  Conductor exponent: 0
  Kodaira Symbol: I0
  Tamagawa Number: 1
sage: E.local_data(2*i + 1)
Local data at Fractional ideal (2*i + 1):
  Reduction type: bad non-split multiplicative
  Local minimal model: Elliptic Curve defined by y^2 + (i+1)*x*y + y = x^3
                       over Number Field in i with defining polynomial x^2 + 1
  Minimal discriminant valuation: 1
  Conductor exponent: 1
  Kodaira Symbol: I1
  Tamagawa Number: 1
```

An example raised in Issue #3897:

```
sage: E = EllipticCurve([1,1])
sage: E.local_data(3)
Local data at Principal ideal (3) of Integer Ring:
  Reduction type: good
  Local minimal model: Elliptic Curve defined by y^2 = x^3 + x + 1
                       over Rational Field
  Minimal discriminant valuation: 0
  Conductor exponent: 0
  Kodaira Symbol: I0
  Tamagawa Number: 1
```

local_integral_model(**P*)

Return a model of self which is integral at the prime ideal P.

Note: The integrality at other primes is not affected, even if P is non-principal.

INPUT:

- `*P` – a prime ideal, or a list or tuple of primes.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2 + 1)
sage: P1, P2 = K.primes_above(5)
sage: E = EllipticCurve([i/5,i/5,i/5,i/5,i/5])
sage: E.local_integral_model((P1,P2))
Elliptic Curve defined by
 y^2 + (-i)*x*y + (-25*i)*y = x^3 + 5*i*x^2 + 125*i*x + 3125*i
 over Number Field in i with defining polynomial x^2 + 1
```

local_minimal_model(*P*, *proof=None*, *algorithm='pari'*)

Return a model which is integral at all primes and minimal at P.

INPUT:

- `P` – either None or a prime ideal of the base field of self.
- `proof` – whether to only use provably correct methods (default controlled by global proof module). Note that the proof module is number_field, not elliptic_curves, since the functions that actually need the flag are in number fields.
- `algorithm` (string, default: "pari") – Ignored unless the base field is **Q**. If "pari", use the PARI C-library pari:ellglobalred implementation of Tate's algorithm over **Q**. If "generic", use the general number field implementation.

OUTPUT:

A model of the curve which is minimal (and integral) at P.

Note: The model is not required to be integral on input.

For principal P, a generator is used as a uniformizer, and integrality or minimality at other primes is not affected. For non-principal P, the minimal model returned will preserve integrality at other primes, but not minimality.

EXAMPLES:

```
sage: K.<a> = NumberField(x^2 - 5)
sage: E = EllipticCurve([20, 225, 750, 1250*a + 6250, 62500*a + 15625])
sage: P = K.ideal(a)
sage: E.local_minimal_model(P).ainvs()
(0, 1, 0, 2*a - 34, -4*a + 66)
```

minimal_discriminant_ideal()

Return the minimal discriminant ideal of this elliptic curve.

OUTPUT:

The integral ideal D whose valuation at every prime P is that of the local minimal model for E at P. If E has a global minimal model, this will be the principal ideal generated by the discriminant of any such model, but otherwise it can be a proper divisor of the discriminant of any model.

EXAMPLES:

```
sage: K.<a> = NumberField(x^2 - x - 57)
sage: K.class_number()
3
sage: E = EllipticCurve([a, -a, a, -5692-820*a, -259213-36720*a])
sage: K.ideal(E.discriminant())
Fractional ideal (90118662980*a + 636812084644)
sage: K.ideal(E.discriminant()).factor()
(Fractional ideal (2))^2 * (Fractional ideal (3, a + 2))^12
```

Here the minimal discriminant ideal is principal but there is no global minimal model since the quotient is the 12th power of a non-principal ideal:

```
sage: E.minimal_discriminant_ideal()
Fractional ideal (4)
sage: E.minimal_discriminant_ideal().factor()
(Fractional ideal (2))^2
```

If (and only if) the curve has everywhere good reduction the result is the unit ideal:

```
sage: K.<a> = NumberField(x^2 - 26)
sage: E = EllipticCurve([a, a-1, a+1, 4*a+10, 2*a+6])
sage: E.conductor()
Fractional ideal (1)
sage: E.discriminant()
-104030*a - 530451
sage: E.minimal_discriminant_ideal()
Fractional ideal (1)
```

Over $\mathbf{Q}$, the result returned is an ideal of $\mathbf{Z}$ rather than a fractional ideal of $\mathbf{Q}$:

```
sage: E = EllipticCurve([1,2,3,4,5])
sage: E.minimal_discriminant_ideal()
Principal ideal (10351) of Integer Ring
```

non_minimal_primes()

Return a list of primes at which this elliptic curve is not minimal.

OUTPUT:

A list of prime ideals (or prime numbers when the base field is $\mathbf{Q}$, empty if this is a global minimal model.

EXAMPLES:

```
sage: K.<a> = NumberField(x^2 - 10)
sage: E = EllipticCurve([0, 0, 0, -22500, 750000*a])
sage: E.non_minimal_primes()
[Fractional ideal (2, a), Fractional ideal (5, a)]
sage: K.ideal(E.discriminant()).factor()
(Fractional ideal (2, a))^24 * (Fractional ideal (3, a + 1))^5 * (Fractional␣
↪ideal (3, a + 2))^5 * (Fractional ideal (5, a))^24 * (Fractional ideal (7))
sage: E.minimal_discriminant_ideal().factor()
(Fractional ideal (2, a))^12 * (Fractional ideal (3, a + 1))^5 * (Fractional␣
↪ideal (3, a + 2))^5 * (Fractional ideal (7))
```

Over **Q**, the primes returned are integers, not ideals:

```
sage: E = EllipticCurve([0, 0, 0, -3024, 46224])
sage: E.non_minimal_primes()
[2, 3]
sage: Emin = E.global_minimal_model()
sage: Emin.non_minimal_primes()
[]
```

If the model is not globally integral, a `ValueError` is raised:

```
sage: E = EllipticCurve([0, 0, 0, 1/2, 1/3])
sage: E.non_minimal_primes()
Traceback (most recent call last):
...
ValueError: non_minimal_primes only defined for integral models
```

period_lattice(*embedding*)

Return the period lattice of the elliptic curve for the given embedding of its base field with respect to the differential $dx/(2y + a_1x + a_3)$.

INPUT:

- `embedding` – an embedding of the base number field into **R** or **C**.

Note: The precision of the embedding is ignored: we only use the given embedding to determine which embedding into `QQbar` to use. Once the lattice has been initialized, periods can be computed to arbitrary precision.

EXAMPLES:

First define a field with two real embeddings:

```
sage: K.<a> = NumberField(x^3 - 2)
sage: E = EllipticCurve([0,0,0,a,2])
sage: embs = K.embeddings(CC); len(embs)
3
```

For each embedding we have a different period lattice:

```
sage: E.period_lattice(embs[0])
Period lattice associated to Elliptic Curve defined by y^2 = x^3 + a*x + 2
 over Number Field in a with defining polynomial x^3 - 2
 with respect to the embedding Ring morphism:
  From: Number Field in a with defining polynomial x^3 - 2
```

(continues on next page)

(continued from previous page)

```
  To:   Algebraic Field
  Defn: a |--> -0.6299605249474365? - 1.0911236359717222?*I

sage: E.period_lattice(embs[1])
Period lattice associated to Elliptic Curve defined by y^2 = x^3 + a*x + 2
 over Number Field in a with defining polynomial x^3 - 2
 with respect to the embedding Ring morphism:
  From: Number Field in a with defining polynomial x^3 - 2
  To:   Algebraic Field
  Defn: a |--> -0.6299605249474365? + 1.0911236359717222?*I

sage: E.period_lattice(embs[2])
Period lattice associated to Elliptic Curve defined by y^2 = x^3 + a*x + 2
 over Number Field in a with defining polynomial x^3 - 2
 with respect to the embedding Ring morphism:
  From: Number Field in a with defining polynomial x^3 - 2
  To:   Algebraic Field
  Defn: a |--> 1.259921049894873?
```

Although the original embeddings have only the default precision, we can obtain the basis with higher precision later:

```
sage: L = E.period_lattice(embs[0])
sage: L.basis()
(1.86405007647981 - 0.903761485143226*I, -0.149344633143919 - 2.
↪06619546272945*I)

sage: L.basis(prec=100)
(1.8640500764798108425920506200 - 0.90376148514322594749786960975*I,
 -0.14934463314391922099120107422 - 2.0661954627294548995621225062*I)
```

rank(***kwds*)

Return the rank of this elliptic curve, if it can be determined.

Note: The optional parameters control the Simon two descent algorithm; see the documentation of *simon_two_descent()* for more details.

INPUT:

- `verbose` – 0, 1, 2, or 3 (default: 0), the verbosity level
- `lim1` – (default: 2) limit on trivial points on quartics
- `lim3` – (default: 4) limit on points on ELS quartics
- `limtriv` – (default: 2) limit on trivial points on elliptic curve
- `maxprob` – (default: 20)
- `limbigprime` – (default: 30) to distinguish between small and large prime numbers. Use probabilistic tests for large primes. If 0, do not use probabilistic tests.
- `known_points` – (default: None) list of known points on the curve

OUTPUT:

If the upper and lower bounds given by Simon two-descent are the same, then the rank has been uniquely identified and we return this. Otherwise, we raise a `ValueError` with an error message specifying the

upper and lower bounds.

Note: For non-quadratic number fields, this code does return, but it takes a long time.

EXAMPLES:

```
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^2 + 23, 'a')
sage: E = EllipticCurve(K, '37')
sage: E == loads(dumps(E))
True
sage: E.rank()
2
```

Here is a curve with two-torsion in the Tate-Shafarevich group, so here the bounds given by the algorithm do not uniquely determine the rank:

```
sage: E = EllipticCurve("15a5")
sage: K.<t> = NumberField(x^2 - 6)
sage: EK = E.base_extend(K)
sage: EK.rank(lim1=1, lim3=1, limtriv=1)
Traceback (most recent call last):
...
ValueError: There is insufficient data to determine the rank -
2-descent gave lower bound 0 and upper bound 2
```

IMPLEMENTATION:

Uses Denis Simon's PARI/GP scripts from http://www.math.unicaen.fr/~simon/.

rank_bounds(**kwds)

Return the lower and upper bounds using *simon_two_descent()*. The results of *simon_two_descent()* are cached.

Note: The optional parameters control the Simon two descent algorithm; see the documentation of *simon_two_descent()* for more details.

INPUT:

- `verbose` – 0, 1, 2, or 3 (default: 0), the verbosity level
- `lim1` – (default: 2) limit on trivial points on quartics
- `lim3` – (default: 4) limit on points on ELS quartics
- `limtriv` – (default: 2) limit on trivial points on elliptic curve
- `maxprob` – (default: 20)
- `limbigprime` – (default: 30) to distinguish between small and large prime numbers. Use probabilistic tests for large primes. If 0, do not use probabilistic tests.
- `known_points` – (default: None) list of known points on the curve

OUTPUT:

lower and upper bounds for the rank of the Mordell-Weil group

Note: For non-quadratic number fields, this code does return, but it takes a long time.

EXAMPLES:

```
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^2 + 23, 'a')
sage: E = EllipticCurve(K, '37')
sage: E == loads(dumps(E))
True
sage: E.rank_bounds()
(2, 2)
```

Here is a curve with two-torsion, again the bounds coincide:

```
sage: Qrt5.<rt5> = NumberField(x^2 - 5)
sage: E = EllipticCurve([0, 5-rt5, 0, rt5, 0])
sage: E.rank_bounds()
(1, 1)
```

Finally an example with non-trivial 2-torsion in Sha. So the 2-descent will not be able to determine the rank, but can only give bounds:

```
sage: E = EllipticCurve("15a5")
sage: K.<t> = NumberField(x^2 - 6)
sage: EK = E.base_extend(K)
sage: EK.rank_bounds(lim1=1, lim3=1, limtriv=1)
(0, 2)
```

IMPLEMENTATION:

Uses Denis Simon's PARI/GP scripts from http://www.math.unicaen.fr/~simon/.

rational_points(***kwds*)

Find rational points on the elliptic curve, all arguments are passed on to `sage.schemes.generic.algebraic_scheme.rational_points()`.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.rational_points(bound=8) # long time
[(-1 : -1 : 1),
 (-1 : 0 : 1),
 (0 : -1 : 1),
 (0 : 0 : 1),
 (0 : 1 : 0),
 (1/4 : -5/8 : 1),
 (1/4 : -3/8 : 1),
 (1 : -1 : 1),
 (1 : 0 : 1),
 (2 : -3 : 1),
 (2 : 2 : 1)]
```

Check that Issue #26677 is fixed:

```
sage: E = EllipticCurve("11a1")
sage: E.rational_points(bound=5)
[(0 : 1 : 0), (5 : 5 : 1)]
```

(continues on next page)

(continued from previous page)

```
sage: E.rational_points(bound=6) # long time
[(0 : 1 : 0), (5 : -6 : 1), (5 : 5 : 1)]
```

An example over a number field:

```
sage: E = EllipticCurve([1,0])
sage: pts = E.rational_points(bound=2, F=QuadraticField(-1))
sage: pts
[(-a : 0 : 1), (0 : 0 : 1), (0 : 1 : 0), (a : 0 : 1)]
sage: pts[0] + pts[1]
(a : 0 : 1)
```

real_components(*embedding*)

Return the number of real components with respect to a real embedding of the base field.

EXAMPLES:

```
sage: K.<a> = QuadraticField(5)
sage: embs = K.real_embeddings()
sage: E = EllipticCurve([0,1,1,a,a])
sage: [e(E.discriminant()) > 0 for e in embs]
[True, False]
sage: [E.real_components(e) for e in embs]
[2, 1]
```

reducible_primes(*algorithm='Billerey'*, *max_l=None*, *num_l=None*, *verbose=False*)

Return a finite set of primes ℓ for which E has a K-rational ℓ-isogeny.

For curves without CM the list returned is exactly the finite set of primes ℓ for which the mod-ℓ Galois representation is reducible. For curves with CM this set is infinite; we return a (not necessarily minimal) finite list of primes ℓ such that every curve isogenous to this curve can be obtained by a finite sequence of isogenies of degree one of the primes in the list.

INPUT:

- `algorithm` (string) – only relevant for non-CM curves. Either 'Billerey", to use the methods of [Bil2011], 'Larson' to use Larson's implementation using Galois representations, or 'heuristic' (see below).
- `max_l` (int or `None`) – only relevant for non-CM curves and algorithms 'Billerey' and 'heuristic. Controls the maximum prime used in either algorithm. If `None`, use the default for that algorithm.
- `num_l` (int or `None`) – only relevant for non-CM curves and algorithm 'Billerey'. Controls the maximum number of primes used in the algorithm. If `None`, use the default for that algorithm.

Note: The 'heuristic' algorithm only checks primes up to the bound `max_l`. This is faster but not guaranteed to be complete. Both the Billerey and Larson algorithms are rigorous.

EXAMPLES:

```
sage: x = polygen(ZZ, 'x')
sage: K = NumberField(x**2 - 29, 'a'); a = K.gen()
sage: E = EllipticCurve([1, 0, ((5 + a)/2)**2, 0, 0])
sage: rho = E.galois_representation()
sage: rho.reducible_primes() # long time
[3, 5]
```

(continues on next page)

(continued from previous page)

```
sage: E.reducible_primes() # long time
[3, 5]
sage: K = NumberField(x**2 + 1, 'a')
sage: E = EllipticCurve_from_j(K(1728)) # CM over K
sage: rho = E.galois_representation()
sage: rho.reducible_primes() # CM curves always return [0]
[0]
sage: E.reducible_primes()
[2, 5]
sage: E = EllipticCurve_from_j(K(0)) # CM but NOT over K
sage: rho = E.galois_representation()
sage: rho.reducible_primes() # long time
[2, 3]
sage: E.reducible_primes()
[2, 3]
sage: E = EllipticCurve_from_j(K(2268945/128)).global_minimal_model() # c.f.␣
↪[Sut2012]
sage: rho = E.galois_representation()
sage: rho.isogeny_bound() # ..but there is no 7-isogeny, long time
[7]
sage: rho.reducible_primes() # long time
[]
sage: E.reducible_primes() # long time
[]
```

reduction (*place*)

Return the reduction of the elliptic curve at a place of good reduction.

INPUT:

- `place` – a prime ideal in the base field of the curve

OUTPUT:

An elliptic curve over a finite field, the residue field of the place.

EXAMPLES:

```
sage: K.<i> = QuadraticField(-1)
sage: EK = EllipticCurve([0, 0, 0, i, i+3])
sage: v = K.fractional_ideal(2*i+3)
sage: EK.reduction(v)
Elliptic Curve defined by y^2  = x^3 + 5*x + 8
 over Residue field of Fractional ideal (2*i + 3)
sage: EK.reduction(K.ideal(1+i))
Traceback (most recent call last):
...
ValueError: The curve must have good reduction at the place.
sage: EK.reduction(K.ideal(2))
Traceback (most recent call last):
...
ValueError: The ideal must be prime.
sage: K = QQ.extension(x^2 + x + 1, "a")
sage: E = EllipticCurve([1024*K.0, 1024*K.0])
sage: E.reduction(2*K)
Elliptic Curve defined by y^2 + (abar+1)*y = x^3
 over Residue field in abar of Fractional ideal (2)
```

regulator_of_points (*points=[]*, *precision=None*, *normalised=True*)

Return the regulator of the given points on this curve.

INPUT:

- `points` – (default: empty list) a list of points on this curve
- `precision` – int or None (default: None): the precision in bits of the result (default real precision if None)
- `normalised` (bool, default `True`) – if `True`, use normalised heights which are independent of base change. Otherwise use the non-normalised Néron-Tate height, as required for the regulator in the BSD conjecture

EXAMPLES:

```
sage: E = EllipticCurve('37a1')
sage: P = E(0,0)
sage: Q = E(1,0)
sage: E.regulator_of_points([P,Q])
0.000000000000000
sage: 2*P == Q
True
```

```
sage: E = EllipticCurve('5077a1')
sage: points = [E.lift_x(x) for x in [-2,-7/4,1]]
sage: E.regulator_of_points(points)
0.417143558758384
sage: E.regulator_of_points(points, precision=100)
0.41714355875838396981711954462
```

```
sage: E = EllipticCurve('389a')
sage: E.regulator_of_points()
1.00000000000000
sage: points = [P,Q] = [E(-1,1), E(0,-1)]
sage: E.regulator_of_points(points)
0.152460177943144
sage: E.regulator_of_points(points, precision=100)
0.15246017794314375162432475705
sage: E.regulator_of_points(points, precision=200)
0.15246017794314375162432475704945582324372707748663081784028
sage: E.regulator_of_points(points, precision=300)
0.
↪152460177943143751624324757049455823243727077486630817840280980046053225683562463604114816
```

Examples over number fields:

```
sage: K.<a> = QuadraticField(97)
sage: E = EllipticCurve(K, [1,1])
sage: P = E(0,1)
sage: P.height()
0.476223106404866
sage: E.regulator_of_points([P])
0.476223106404866
```

When the parameter `normalised` is set to `False`, each height is multiplied by the degree d of the base field, and the regulator of r points is multiplied by d^r:

```
sage: P.height(normalised=False)
0.952446212809731
sage: E.regulator_of_points([P], normalised=False)
0.952446212809731
```

```
sage: E = EllipticCurve('11a1')
sage: x = polygen(QQ)
sage: K.<t> = NumberField(x^2 + 47)
sage: EK = E.base_extend(K)
sage: T = EK(5, 5)
sage: T.order()
5
sage: P = EK(-2, -1/2*t - 1/2)
sage: P.order()
+Infinity
sage: EK.regulator_of_points([P,T]) # random very small output
-1.23259516440783e-32
sage: EK.regulator_of_points([P,T]).abs() < 1e-30
True
```

```
sage: E = EllipticCurve('389a1')
sage: P,Q = E.gens()
sage: E.regulator_of_points([P,Q])
0.152460177943144
sage: K.<t> = NumberField(x^2 + 47)
sage: EK = E.base_extend(K)
sage: EK.regulator_of_points([EK(P),EK(Q)])
0.152460177943144
```

```
sage: K.<i> = QuadraticField(-1)
sage: E = EllipticCurve([0,0,0,i,i])
sage: P = E(-9+4*i, -18-25*i)
sage: Q = E(i, -i)
sage: E.height_pairing_matrix([P,Q])
[  2.16941934493768 -0.870059380421505]
[-0.870059380421505  0.424585837470709]
sage: E.regulator_of_points([P,Q])
0.164101403936070
```

saturation (*points*, *verbose=False*, *max_prime=0*, *one_prime=0*, *odd_primes_only=False*, *lower_ht_bound=None*, *reg=None*, *debug=False*)

Given a list of rational points on E over K, compute the saturation in $E(K)$ of the subgroup they generate.

INPUT:

- `points (list)` – list of points on E. Points of finite order are ignored; the remaining points should be independent, or an error is raised.
- `verbose` (bool) – (default: `False`), if `True`, give verbose output.
- `max_prime` (int, default 0) – saturation is performed for all primes up to `max_prime`. If `max_prime` is 0, perform saturation at *all* primes, i.e., compute the true saturation.
- `odd_primes_only` (bool, default `False`) – only do saturation at odd primes.
- `one_prime` (int, default 0) – if nonzero, only do saturation at this prime.

The following two inputs are optional, and may be provided to speed up the computation.

- `lower_ht_bound` (real, default `None`) – lower bound of the regulator $E(K)$, if known.
- `reg` (real, default `None`) – regulator of the span of points, if known.
- `debug` (int, default 0) – used for debugging and testing.

OUTPUT:

- `saturation` (list) – points that form a basis for the saturation.
- `index` (int) – the index of the group generated by the input points in their saturation.
- `regulator` (real with default precision, or `None`) – regulator of saturated points.

EXAMPLES:

```
sage: K.<i> = QuadraticField(-1)
sage: E = EllipticCurve('389a1')
sage: EK = E.change_ring(K)
sage: P = EK(-1,1); Q = EK(0,-1)

sage: EK.saturation([2*P], max_prime=2)
([(-1 : 1 : 1)], 2, 0.686667083305587)
sage: EK.saturation([12*P], max_prime=2)
([(26/361 : -5720/6859 : 1)], 4, 6.18000374975028)
sage: EK.saturation([12*P], lower_ht_bound=0.1)
([(-1 : 1 : 1)], 12, 0.686667083305587)
sage: EK.saturation([2*P, Q], max_prime=2)
([(-1 : 1 : 1), (0 : -1 : 1)], 2, 0.152460177943144)
sage: EK.saturation([P + Q, P - Q], lower_ht_bound=.1, debug=2)
([(-1 : 1 : 1), (1 : 0 : 1)], 2, 0.152460177943144)
sage: EK.saturation([P + Q, 17*Q], lower_ht_bound=0.1)  # long time
([(4 : 8 : 1), (0 : -1 : 1)], 17, 0.152460177943143)

sage: R = EK(i-2,-i-3)
sage: EK.saturation([P + R, P + Q, Q + R], lower_ht_bound=0.1)
([(841/1369*i - 171/1369 : 41334/50653*i - 74525/50653 : 1),
  (4 : 8 : 1),
  (-1/25*i + 18/25 : -69/125*i - 58/125 : 1)],
 2,
 0.103174443217351)
sage: EK.saturation([26*Q], lower_ht_bound=0.1)
([(0 : -1 : 1)], 26, 0.327000773651605)
```

Another number field:

```
sage: E = EllipticCurve('389a1')
sage: K.<a> = NumberField(x^3 - x + 1)
sage: EK = E.change_ring(K)
sage: P = EK(-1,1); Q = EK(0,-1)
sage: EK.saturation([P + Q, P - Q], lower_ht_bound=0.1)
([(-1 : 1 : 1), (1 : 0 : 1)], 2, 0.152460177943144)
sage: EK.saturation([3*P, P + 5*Q], lower_ht_bound=0.1)
([(-185/2209 : -119510/103823 : 1), (80041/34225 : -26714961/6331625 : 1)],
 15,
 0.152460177943144)
```

A different curve:

```
sage: K.<a> = QuadraticField(3)
sage: E = EllipticCurve('37a1')
```

(continues on next page)

(continued from previous page)

```
sage: EK = E.change_ring(K)
sage: P = EK(0,0); Q = EK(2-a, 2*a-4)
sage: EK.saturation([3*P - Q, 3*P + Q], lower_ht_bound=.01)
([(0 : 0 : 1), (1/2*a : -1/4*a - 1/4 : 1)], 6, 0.0317814530725985)
```

The points must be linearly independent:

```
sage: EK.saturation([2*P, 3*Q, P-Q])
Traceback (most recent call last):
...
ValueError: points not linearly independent in saturation()
```

Degenerate case:

```
sage: EK.saturation([])
([], 1, 1.00000000000000)
```

ALGORITHM:

For rank 1 subgroups, simply do trial division up to the maximal prime divisor. For higher rank subgroups, perform trial division on all linear combinations for small primes, and look for projections $E(K) \to \oplus E(k) \otimes \mathbf{F}_p$ which are either full rank or provide p-divisible linear combinations, where the k here are residue fields of K.

simon_two_descent(*verbose=0*, *lim1=2*, *lim3=4*, *limtriv=2*, *maxprob=20*, *limbigprime=30*, *known_points=None*)

Return lower and upper bounds on the rank of the Mordell-Weil group $E(K)$ and a list of points.

This method is used internally by the `rank()`, `rank_bounds()` and `gens()` methods.

INPUT:

- `self` – an elliptic curve E over a number field K
- `verbose` – 0, 1, 2, or 3 (default: 0), the verbosity level
- `lim1` – (default: 2) limit on trivial points on quartics
- `lim3` – (default: 4) limit on points on ELS quartics
- `limtriv` – (default: 2) limit on trivial points on E
- `maxprob` – (default: 20)
- `limbigprime` – (default: 30) to distinguish between small and large prime numbers. Use probabilistic tests for large primes. If 0, do not use probabilistic tests.
- `known_points` – (default: None) list of known points on the curve

OUTPUT: a triple `(lower, upper, list)` consisting of

- `lower` (integer) – lower bound on the rank
- `upper` (integer) – upper bound on the rank
- `list` – list of points in $E(K)$

The integer `upper` is in fact an upper bound on the dimension of the 2-Selmer group, hence on the dimension of $E(K)/2E(K)$. It is equal to the dimension of the 2-Selmer group except possibly if $E(K)[2]$ has dimension 1. In that case, `upper` may exceed the dimension of the 2-Selmer group by an even number, due to the fact that the algorithm does not perform a second descent.

Note: For non-quadratic number fields, this code does return, but it takes a long time.

ALGORITHM:

Uses Denis Simon's PARI/GP scripts from https://simond.users.lmno.cnrs.fr/.

EXAMPLES:

```
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^2 + 23, 'a')
sage: E = EllipticCurve(K, '37')
sage: E == loads(dumps(E))
True
sage: E.simon_two_descent()
(2, 2, [(0 : 0 : 1), (1/18*a + 7/18 : -5/54*a - 17/54 : 1)])
sage: E.simon_two_descent(lim1=5, lim3=5, limtriv=10, maxprob=7,␣
↪limbigprime=10)
(2, 2, [(-1 : 0 : 1), (-2 : -1/2*a - 1/2 : 1)])
```

```
sage: K.<a> = NumberField(x^2 + 7, 'a')
sage: E = EllipticCurve(K, [0,0,0,1,a]); E
Elliptic Curve defined by y^2 = x^3 + x + a
 over Number Field in a with defining polynomial x^2 + 7

sage: v = E.simon_two_descent(verbose=1); v
 elliptic curve: Y^2 = x^3 + Mod(1, y^2 + 7)*x + Mod(y, y^2 + 7)
 Trivial points on the curve = [[1, 1, 0], [Mod(1/2*y + 3/2, y^2 + 7), Mod(-y␣
↪- 2, y^2 + 7), 1]]
#S(E/K)[2]    = 2
#E(K)/2E(K)   = 2
#III(E/K)[2]  = 1
rank(E/K)     = 1
 listpoints = [[Mod(1/2*y + 3/2, y^2 + 7), Mod(-y - 2, y^2 + 7), 1]]
(1, 1, [(1/2*a + 3/2 : -a - 2 : 1)])

sage: v = E.simon_two_descent(verbose=2)
K = bnfinit(y^2 + 7);
a = Mod(y,K.pol);
bnfellrank(K, [0, 0, 0, 1, a], [[Mod(1/2*y + 3/2, y^2 + 7), Mod(-y - 2, y^2 +␣
↪7)]]);
...
v = [1, 1, [[Mod(1/2*y + 3/2, y^2 + 7), Mod(-y - 2, y^2 + 7)]]]
sage: v
(1, 1, [(1/2*a + 3/2 : -a - 2 : 1)])
```

A curve with 2-torsion:

```
sage: K.<a> = NumberField(x^2 + 7)
sage: E = EllipticCurve(K, '15a')
sage: E.simon_two_descent()  # long time (3s on sage.math, 2013), points can␣
↪vary
(1, 3, [...])
```

Check that the bug reported in Issue #15483 is fixed:

```
sage: K.<s> = QuadraticField(229)
sage: c4 = 2173 - 235*(1 - s)/2
```

(continues on next page)

(continued from previous page)

```
sage: c6 = -124369 + 15988*(1 - s)/2
sage: E = EllipticCurve([-c4/48, -c6/864])
sage: E.simon_two_descent()
(0, 0, [])

sage: R.<t> = QQ[]
sage: L.<g> = NumberField(t^3 - 9*t^2 + 13*t - 4)
sage: E1 = EllipticCurve(L, [1-g*(g-1), -g^2*(g-1), -g^2*(g-1), 0, 0])
sage: E1.rank()  # long time (about 5 s)
0

sage: K = CyclotomicField(43).subfields(3)[0][0]
sage: E = EllipticCurve(K, '37')
sage: E.simon_two_descent()  # long time (4s on sage.math, 2013)
(3,
 3,
 [(1/8*zeta43_0^2 - 3/8*zeta43_0 - 1/4 : -5/16*zeta43_0^2 + 7/16*zeta43_0 + 1/
↪8 : 1),
  (0 : 0 : 1)])
```

tamagawa_exponent(*P*, *proof=None*)

Return the Tamagawa index of this elliptic curve at the prime P.

INPUT:

- `P` – either None or a prime ideal of the base field of self.
- `proof` – whether to only use provably correct methods (default controlled by global proof module). Note that the proof module is number_field, not elliptic_curves, since the functions that actually need the flag are in number fields.

OUTPUT:

(positive integer) The Tamagawa index of the curve at P.

EXAMPLES:

```
sage: K.<a> = NumberField(x^2 - 5)
sage: E = EllipticCurve([20, 225, 750, 625*a + 6875, 31250*a + 46875])
sage: [E.tamagawa_exponent(P) for P in E.discriminant().support()]
[1, 1, 1, 1]
sage: K.<a> = QuadraticField(-11)
sage: E = EllipticCurve('11a1').change_ring(K)
sage: [E.tamagawa_exponent(P) for P in K(11).support()]
[10]
```

tamagawa_number(*P*, *proof=None*)

Return the Tamagawa number of this elliptic curve at the prime P.

INPUT:

- `P` – either None or a prime ideal of the base field of self.
- `proof` – whether to only use provably correct methods (default controlled by global proof module). Note that the proof module is number_field, not elliptic_curves, since the functions that actually need the flag are in number fields.

OUTPUT:

(positive integer) The Tamagawa number of the curve at P.

EXAMPLES:

```
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^2 - 5)
sage: E = EllipticCurve([20, 225, 750, 625*a + 6875, 31250*a + 46875])
sage: [E.tamagawa_number(P) for P in E.discriminant().support()]
[1, 1, 1, 1]
sage: K.<a> = QuadraticField(-11)
sage: E = EllipticCurve('11a1').change_ring(K)
sage: [E.tamagawa_number(P) for P in K(11).support()]
[10]
```

tamagawa_numbers()

Return a list of all Tamagawa numbers for all prime divisors of the conductor (in order).

EXAMPLES:

```
sage: e = EllipticCurve('30a1')
sage: e.tamagawa_numbers()
[2, 3, 1]
sage: vector(e.tamagawa_numbers())
(2, 3, 1)
sage: K.<a> = NumberField(x^2 + 3)
sage: eK = e.base_extend(K)
sage: eK.tamagawa_numbers()
[4, 6, 1]
```

tamagawa_product()

Return the product of the Tamagawa numbers c_v where v runs over all prime ideals of K.

Note: See also tamagawa_product_bsd(), which includes an additional factor when the model is not globally minimal, as required by the BSD formula.

OUTPUT:

A positive integer.

EXAMPLES:

```
sage: K.<i> = NumberField(x^2 + 1)
sage: E = EllipticCurve([0, 2+i])
sage: E.tamagawa_product()
1

sage: E = EllipticCurve([(2*i+1)^2, i*(2*i+1)^7])
sage: E.tamagawa_product()
4
```

An example over $\mathbf{Q}$:

```
sage: E = EllipticCurve('30a')
sage: E.tamagawa_product()
6
```

An example with everywhere good reduction, where the product is empty:

```
sage: x = polygen(QQ)
sage: K.<a> = NumberField(x^2 - 38)
sage: E = EllipticCurve( [a, -a + 1, a + 1, -5*a + 15, -5*a + 21])
sage: E.tamagawa_numbers()
[]
sage: E.tamagawa_product()
1
```

tamagawa_product_bsd()

Given an elliptic curve E over a number field K, this function returns the integer $C(E/K)$ that appears in the Birch and Swinnerton-Dyer conjecture accounting for the local information at finite places. If the model is a global minimal model then $C(E/K)$ is simply the product of the Tamagawa numbers c_v where v runs over all prime ideals of K. Otherwise, if the model has to be changed at a place v a correction factor appears. The definition is such that $C(E/K)$ times the periods at the infinite places is invariant under change of the Weierstrass model. See [Tate1966] and [DD2010] for details.

Note: This definition differs from the definition of `tamagawa_product` for curves defined over **Q**. Over the rational number it is always defined to be the product of the Tamagawa numbers, so the two definitions only agree when the model is global minimal.

OUTPUT:

A rational number

EXAMPLES:

```
sage: K.<i> = NumberField(x^2 + 1)
sage: E = EllipticCurve([0, 2+i])
sage: E.tamagawa_product_bsd()
1

sage: E = EllipticCurve([(2*i+1)^2, i*(2*i+1)^7])
sage: E.tamagawa_product_bsd()
4
```

An example where the Neron model changes over K:

```
sage: K.<t> = NumberField(x^5 - 10*x^3 + 5*x^2 + 10*x + 1)
sage: E = EllipticCurve(K, '75a1')
sage: E.tamagawa_product_bsd()
5
sage: da = E.local_data()
sage: [dav.tamagawa_number() for dav in da]
[1, 1]
```

An example over **Q** (Issue #9413):

```
sage: E = EllipticCurve('30a')
sage: E.tamagawa_product_bsd()
6
```

torsion_order()

Return the order of the torsion subgroup of this elliptic curve.

OUTPUT:

(integer) the order of the torsion subgroup of this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: x = polygen(ZZ, 'x')
sage: K.<t> = NumberField(x^4 + x^3 + 11*x^2 + 41*x + 101)
sage: EK = E.base_extend(K)
sage: EK.torsion_order()  # long time (2s on sage.math, 2014)
25
```

```
sage: E = EllipticCurve('15a1')
sage: K.<t> = NumberField(x^2 + 2*x + 10)
sage: EK = E.base_extend(K)
sage: EK.torsion_order()
16
```

```
sage: E = EllipticCurve('19a1')
sage: K.<t> = NumberField(x^9 - 3*x^8 - 4*x^7 + 16*x^6 - 3*x^5 - 21*x^4 + 5*x^
↪3 + 7*x^2 - 7*x + 1)
sage: EK = E.base_extend(K)
sage: EK.torsion_order()
9
```

```
sage: K.<i> = QuadraticField(-1)
sage: EK = EllipticCurve([0, 0, 0, i, i + 3])
sage: EK.torsion_order()
1
```

torsion_points()

Return a list of the torsion points of this elliptic curve.

OUTPUT:

(list) A sorted list of the torsion points.

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: E.torsion_points()
[(0 : 1 : 0), (5 : -6 : 1), (5 : 5 : 1), (16 : -61 : 1), (16 : 60 : 1)]
sage: x = polygen(ZZ, 'x')
sage: K.<t> = NumberField(x^4 + x^3 + 11*x^2 + 41*x + 101)
sage: EK = E.base_extend(K)
sage: EK.torsion_points()  # long time (1s on sage.math, 2014)
[(0 : 1 : 0),
 (t : 1/11*t^3 + 6/11*t^2 + 19/11*t + 48/11 : 1),
 (1/11*t^3 - 5/11*t^2 + 19/11*t - 40/11 : -6/11*t^3 - 3/11*t^2 - 26/11*t -␣
↪321/11 : 1),
 (1/11*t^3 - 5/11*t^2 + 19/11*t - 40/11 : 6/11*t^3 + 3/11*t^2 + 26/11*t + 310/
↪11 : 1),
 (t : -1/11*t^3 - 6/11*t^2 - 19/11*t - 59/11 : 1),
 (16 : 60 : 1),
 (-3/55*t^3 - 7/55*t^2 - 2/55*t - 133/55 : 6/55*t^3 + 3/55*t^2 + 25/11*t +␣
↪156/55 : 1),
 (14/121*t^3 - 15/121*t^2 + 90/121*t + 232/121 : 16/121*t^3 - 69/121*t^2 +␣
↪293/121*t - 46/121 : 1),
 (-26/121*t^3 + 20/121*t^2 - 219/121*t - 995/121 : -15/121*t^3 - 156/121*t^2␣
↪+ 232/121*t - 2887/121 : 1),
 (10/121*t^3 + 49/121*t^2 + 168/121*t + 73/121 : -32/121*t^3 - 60/121*t^2 +␣
```

(continues on next page)

(continued from previous page)

```
↪261/121*t + 686/121 : 1),
 (5 : 5 : 1),
 (-9/121*t^3 - 21/121*t^2 - 127/121*t - 377/121 : -7/121*t^3 + 24/121*t^2 +␣
↪197/121*t + 16/121 : 1),
 (3/55*t^3 + 7/55*t^2 + 2/55*t + 78/55 : 7/55*t^3 - 24/55*t^2 + 9/11*t + 17/
↪55 : 1),
 (-5/121*t^3 + 36/121*t^2 - 84/121*t + 24/121 : -34/121*t^3 + 27/121*t^2 -␣
↪305/121*t - 829/121 : 1),
 (5/121*t^3 - 14/121*t^2 - 158/121*t - 453/121 : 49/121*t^3 + 129/121*t^2 +␣
↪315/121*t + 86/121 : 1),
 (5 : -6 : 1),
 (5/121*t^3 - 14/121*t^2 - 158/121*t - 453/121 : -49/121*t^3 - 129/121*t^2 -␣
↪315/121*t - 207/121 : 1),
 (-5/121*t^3 + 36/121*t^2 - 84/121*t + 24/121 : 34/121*t^3 - 27/121*t^2 + 305/
↪121*t + 708/121 : 1),
 (3/55*t^3 + 7/55*t^2 + 2/55*t + 78/55 : -7/55*t^3 + 24/55*t^2 - 9/11*t - 72/
↪55 : 1),
 (-9/121*t^3 - 21/121*t^2 - 127/121*t - 377/121 : 7/121*t^3 - 24/121*t^2 -␣
↪197/121*t - 137/121 : 1),
 (16 : -61 : 1),
 (10/121*t^3 + 49/121*t^2 + 168/121*t + 73/121 : 32/121*t^3 + 60/121*t^2 -␣
↪261/121*t - 807/121 : 1),
 (-26/121*t^3 + 20/121*t^2 - 219/121*t - 995/121 : 15/121*t^3 + 156/121*t^2 -␣
↪232/121*t + 2766/121 : 1),
 (14/121*t^3 - 15/121*t^2 + 90/121*t + 232/121 : -16/121*t^3 + 69/121*t^2 -␣
↪293/121*t - 75/121 : 1),
 (-3/55*t^3 - 7/55*t^2 - 2/55*t - 133/55 : -6/55*t^3 - 3/55*t^2 - 25/11*t -␣
↪211/55 : 1)]
```

```
sage: E = EllipticCurve('15a1')
sage: K.<t> = NumberField(x^2 + 2*x + 10)
sage: EK = E.base_extend(K)
sage: EK.torsion_points()
[(0 : 1 : 0),
 (-7 : -5*t - 2 : 1),
 (-7 : 5*t + 8 : 1),
 (-13/4 : 9/8 : 1),
 (-2 : -2 : 1),
 (-2 : 3 : 1),
 (-t - 2 : -t - 7 : 1),
 (-t - 2 : 2*t + 8 : 1),
 (-1 : 0 : 1),
 (t : t - 5 : 1),
 (t : -2*t + 4 : 1),
 (1/2 : -5/4*t - 2 : 1),
 (1/2 : 5/4*t + 1/2 : 1),
 (3 : -2 : 1),
 (8 : -27 : 1),
 (8 : 18 : 1)]
```

```
sage: K.<i> = QuadraticField(-1)
sage: EK = EllipticCurve(K, [0,0,0,0,-1])
sage: EK.torsion_points()
 [(0 : 1 : 0),
  (-2 : -3*i : 1),
  (-2 : 3*i : 1),
```

(continues on next page)

(continued from previous page)

```
  (0 : -i : 1),
  (0 : i : 1),
  (1 : 0 : 1)]
```

torsion_subgroup()

Return the torsion subgroup of this elliptic curve.

OUTPUT: The `EllipticCurveTorsionSubgroup` associated to this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: x = polygen(ZZ, 'x')
sage: K.<t> = NumberField(x^4 + x^3 + 11*x^2 + 41*x + 101)
sage: EK = E.base_extend(K)
sage: tor = EK.torsion_subgroup()  # long time (2s on sage.math, 2014)
sage: tor  # long time
Torsion Subgroup isomorphic to Z/5 + Z/5 associated to the Elliptic Curve
 defined by y^2 + y = x^3 + (-1)*x^2 + (-10)*x + (-20) over Number Field
 in t with defining polynomial x^4 + x^3 + 11*x^2 + 41*x + 101
sage: tor.gens()  # long time
((16 : 60 : 1), (t : 1/11*t^3 + 6/11*t^2 + 19/11*t + 48/11 : 1))
```

```
sage: E = EllipticCurve('15a1')
sage: K.<t> = NumberField(x^2 + 2*x + 10)
sage: EK = E.base_extend(K)
sage: EK.torsion_subgroup()
Torsion Subgroup isomorphic to Z/4 + Z/4 associated to the
 Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 + (-10)*x + (-10)
 over Number Field in t with defining polynomial x^2 + 2*x + 10
```

```
sage: E = EllipticCurve('19a1')
sage: K.<t> = NumberField(x^9-3*x^8-4*x^7+16*x^6-3*x^5-21*x^4+5*x^3+7*x^2-
↪7*x+1)
sage: EK = E.base_extend(K)
sage: EK.torsion_subgroup()
Torsion Subgroup isomorphic to Z/9 associated to the Elliptic Curve defined
 by y^2 + y = x^3 + x^2 + (-9)*x + (-15) over Number Field in t with defining
 polynomial x^9 - 3*x^8 - 4*x^7 + 16*x^6 - 3*x^5 - 21*x^4 + 5*x^3 + 7*x^2 -␣
↪7*x + 1
```

```
sage: K.<i> = QuadraticField(-1)
sage: EK = EllipticCurve([0, 0, 0, i, i+3])
sage: EK.torsion_subgroup ()
Torsion Subgroup isomorphic to Trivial group associated to the
 Elliptic Curve defined by y^2 = x^3 + i*x + (i+3)
 over Number Field in i with defining polynomial x^2 + 1 with i = 1*I
```

See also:

Use *division_field()* to determine the field of definition of the ℓ-torsion subgroup.

18.4 Canonical heights for elliptic curves over number fields

Also, rigorous lower bounds for the canonical height of non-torsion points, implementing the algorithms in [CS2006] (over **Q**) and [Tho2010], which also refer to [CPS2006].

AUTHORS:

- Robert Bradshaw (2010): initial version
- John Cremona (2014): added many docstrings and doctests

class sage.schemes.elliptic_curves.height.**EllipticCurveCanonicalHeight**(*E*)

Bases: object

Class for computing canonical heights of points on elliptic curves defined over number fields, including rigorous lower bounds for the canonical height of non-torsion points.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.height import EllipticCurveCanonicalHeight
sage: E = EllipticCurve([0,0,0,0,1])
sage: EllipticCurveCanonicalHeight(E)
EllipticCurveCanonicalHeight object associated to
 Elliptic Curve defined by y^2 = x^3 + 1 over Rational Field
```

Normally this object would be created like this:

```
sage: E.height_function()
EllipticCurveCanonicalHeight object associated to
 Elliptic Curve defined by y^2 = x^3 + 1 over Rational Field
```

B(*n*, *mu*)

Return the value $B_n(\mu)$.

INPUT:

- n (int) – a positive integer
- mu (real) – a positive real number

OUTPUT:

The real value $B_n(\mu)$ as defined in [Tho2010], section 5.

EXAMPLES:

Example 10.2 from [Tho2010]:

```
sage: K.<i> = QuadraticField(-1)                                                #␣
↪needs sage.rings.number_field
sage: E = EllipticCurve([0, 1-i, i, -i, 0])                                     #␣
↪needs sage.rings.number_field
sage: H = E.height_function()                                                   #␣
↪needs sage.rings.number_field
```

In [Tho2010] the value is given as 0.772:

```
sage: RealField(12)( H.B(5, 0.01) )                                            #␣
↪needs sage.rings.number_field
0.777
```

DE (*n*)

Return the value $D_E(n)$.

INPUT:

- n (int) – a positive integer

OUTPUT:

The value $D_E(n)$ as defined in [Tho2010], section 4.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: K.<i> = QuadraticField(-1)
sage: E = EllipticCurve([0, 0, 0, 1+5*i, 3+i])
sage: H = E.height_function()
sage: [H.DE(n) for n in srange(1,6)]
[0, 2*log(5) + 2*log(2), 0, 2*log(13) + 2*log(5) + 4*log(2), 0]
```

ME ()

Return the norm of the ideal M_E.

OUTPUT:

The norm of the ideal M_E as defined in [Tho2010], section 3.1. This is 1 if E is a global minimal model, and in general measures the non-minimality of E.

EXAMPLES:

```
sage: K.<i> = QuadraticField(-1)                                                #␣
↪needs sage.rings.number_field
sage: E = EllipticCurve([0, 0, 0, 1+5*i, 3+i])                                  #␣
↪needs sage.rings.number_field
sage: H = E.height_function()                                                   #␣
↪needs sage.rings.number_field
sage: H.ME()                                                                    #␣
↪needs sage.rings.number_field
1
sage: E = EllipticCurve([0,0,0,0,1])
sage: E.height_function().ME()
1
sage: E = EllipticCurve([0,0,0,0,64])
sage: E.height_function().ME()
4096
sage: E.discriminant()/E.minimal_model().discriminant()
4096
```

S (*xi1*, *xi2*, *v*)

Return the union of intervals $S^{(v)}(\xi_1, \xi_2)$.

INPUT:

- xi1, xi2 (real) – real numbers with $\xi_1 \leq \xi_2$.
- v (embedding) – a real embedding of the field.

OUTPUT:

The union of intervals $S^{(v)}(\xi_1, \xi_2)$ defined in [Tho2010] section 6.1.

EXAMPLES:

An example over **Q**:

```
sage: E = EllipticCurve('389a')
sage: v = QQ.places()[0]
sage: H = E.height_function()
sage: H.S(2, 3, v)
([0.224512677391895, 0.274544821597130] U [0.725455178402870, 0.
↪775487322608105])
```

An example over a number field:

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)
sage: E = EllipticCurve([0,0,0,0,a])
sage: v = K.real_places()[0]
sage: H = E.height_function()
sage: H.S(9, 10, v)
([0.078119444725347..., 0.082342373201640...] U [0.91765762679836..., 0.
↪92188055527465...])
```

Sn (*xi1*, *xi2*, *n*, *v*)

Return the union of intervals $S_n^{(v)}(\xi_1, \xi_2)$.

INPUT:

- `xi1`, `xi2` (real) – real numbers with $\xi_1 \leq \xi_2$.
- `n` (integer) – a positive integer.
- `v` (embedding) – a real embedding of the field.

OUTPUT:

The union of intervals $S_n^{(v)}(\xi_1, \xi_2)$ defined in [Tho2010] (Lemma 6.1).

EXAMPLES:

An example over **Q**:

```
sage: E = EllipticCurve('389a')
sage: v = QQ.places()[0]
sage: H = E.height_function()
sage: H.S(2, 3, v), H.Sn(2, 3, 1, v)
(([0.224512677391895, 0.274544821597130] U [0.725455178402870, 0.
↪775487322608105]),
([0.224512677391895, 0.274544821597130] U [0.725455178402870, 0.
↪775487322608105]))
sage: H.Sn(2, 3, 6, v)
([0.0374187795653158, 0.0457574702661884] U [0.120909196400478, 0.
↪129247887101351] U [0.204085446231982, 0.212424136932855] U [0.
↪287575863067145, 0.295914553768017] U [0.370752112898649, 0.
↪379090803599522] U [0.454242529733812, 0.462581220434684] U [0.
↪537418779565316, 0.545757470266188] U [0.620909196400478, 0.
↪629247887101351] U [0.704085446231982, 0.712424136932855] U [0.
↪787575863067145, 0.795914553768017] U [0.870752112898649, 0.
↪879090803599522] U [0.954242529733812, 0.962581220434684])
```

An example over a number field:

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)
sage: E = EllipticCurve([0,0,0,0,a])
sage: v = K.real_places()[0]
sage: H = E.height_function()
sage: H.S(2, 3, v), H.Sn(2, 3, 1, v)
(([0.142172065860075, 0.172845716928584] U [0.827154283071416, 0.
↪857827934139925]),
 ([0.142172065860075, 0.172845716928584] U [0.827154283071416, 0.
↪857827934139925]))
sage: H.Sn(2, 3, 6, v)
([0.0236953443100124, 0.0288076194880974] U [0.137859047178569, 0.
↪142971322356654] U [0.190362010976679, 0.195474286154764] U [0.
↪304525713845236, 0.309637989023321] U [0.357028677643346, 0.
↪362140952821431] U [0.471192380511903, 0.476304655689988] U [0.
↪523695344310012, 0.528807619488097] U [0.637859047178569, 0.
↪642971322356654] U [0.690362010976679, 0.695474286154764] U [0.
↪804525713845236, 0.809637989023321] U [0.857028677643346, 0.
↪862140952821431] U [0.971192380511903, 0.976304655689988])
```

alpha(*v*, *tol=0.01*)

Return the constant α_v associated to the embedding v.

INPUT:

- v – an embedding of the base field into **R** or **C**

OUTPUT:

The constant α_v. In the notation of [CPS2006] and [Tho2010] (section 3.2), $\alpha_v^3 = \epsilon_v$. The result is cached since it only depends on the curve.

EXAMPLES:

Example 1 from [CPS2006]:

```
sage: # needs sage.rings.number_field
sage: K.<i> = QuadraticField(-1)
sage: E = EllipticCurve([0, 0, 0, 1 + 5*i, 3 + i])
sage: H = E.height_function()
sage: alpha = H.alpha(K.places()[0])
sage: alpha
1.12272013439355
```

Compare with $\log(\epsilon_v) = 0.344562...$ in [CPS2006]:

```
sage: 3*alpha.log()                                                 #␣
↪needs sage.rings.number_field
0.347263296676126
```

base_field()

Return the base field.

EXAMPLES:

```
sage: E = EllipticCurve([0,0,0,0,1])
sage: H = E.height_function()
sage: H.base_field()
Rational Field
```

complex_intersection_is_empty(*Bk*, *v*, *verbose=False*, *use_half=True*)

Returns True iff an intersection of $T_n^{(v)}$ sets is empty.

INPUT:

- `Bk` (list) – a list of reals.
- `v` (embedding) – a complex embedding of the number field.
- `verbose` (boolean, default: `False`) – verbosity flag.
- `use_half` (boolean, default: `False`) – if True, use only half the fundamental region.

OUTPUT:

True or False, according as the intersection of the unions of intervals $T_n^{(v)}(-b, b)$ for b in the list `Bk` (see [Tho2010], section 7) is empty or not. When `Bk` is the list of $b = \sqrt{B_n(\mu)}$ for $n = 1, 2, 3, \ldots$ for some $\mu > 0$ this means that all non-torsion points on E with everywhere good reduction have canonical height strictly greater than μ, by [Tho2010], Proposition 7.8.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)
sage: E = EllipticCurve([0,0,0,0,a])
sage: v = K.complex_embeddings()[0]
sage: H = E.height_function()
```

The following two lines prove that the heights of non-torsion points on E with everywhere good reduction have canonical height strictly greater than 0.02, but fail to prove the same for 0.03. For the first proof, using only $n = 1, 2, 3$ is not sufficient:

```
sage: H.complex_intersection_is_empty([H.B(n,0.02) for n in [1,2,3]], v)  #␣
↪long time, needs sage.rings.number_field
False
sage: H.complex_intersection_is_empty([H.B(n,0.02) for n in [1,2,3,4]], v)  #␣
↪needs sage.rings.number_field
True
sage: H.complex_intersection_is_empty([H.B(n,0.03) for n in [1,2,3,4]], v)  #␣
↪long time, needs sage.rings.number_field
False
```

Using $n \leq 6$ enables us to prove the lower bound 0.03. Note that it takes longer when the result is `False` than when it is `True`:

```
sage: H.complex_intersection_is_empty([H.B(n,0.03) for n in [1..6]], v)     #␣
↪needs sage.rings.number_field
True
```

curve()

Return the elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve([0,0,0,0,1])
sage: H = E.height_function()
sage: H.curve()
Elliptic Curve defined by y^2 = x^3 + 1 over Rational Field
```

e_p(*p*)

Return the exponent of the group over the residue field at p.

INPUT:

- p – a prime ideal of K (or a prime number if $K = \mathbf{Q}$).

OUTPUT:

A positive integer e_p, the exponent of the group of nonsingular points on the reduction of the elliptic curve modulo p. The result is cached.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: K.<i> = QuadraticField(-1)
sage: E = EllipticCurve([0, 0, 0, 1 + 5*i, 3 + i])
sage: H = E.height_function()
sage: H.e_p(K.prime_above(2))
2
sage: H.e_p(K.prime_above(3))
10
sage: H.e_p(K.prime_above(5))
9
sage: E.conductor().norm().factor()
2^10 * 20921
sage: p1, p2 = K.primes_above(20921)
sage: E.local_data(p1)
Local data at Fractional ideal (-40*i + 139):
Reduction type: bad split multiplicative
...
sage: H.e_p(p1)
20920
sage: E.local_data(p2)
Local data at Fractional ideal (40*i + 139):
Reduction type: good
...
sage: H.e_p(p2)
20815
```

fk_intervals(*v=None*, *N=20*, *domain=Complex Interval Field with 53 bits of precision*)

Return a function approximating the Weierstrass function, with error.

INPUT:

- v (embedding) – an embedding of the number field. If None (default) use the real embedding if the field is $\mathbf{Q}$ and raise an error for other fields.
- N (int) – The number of terms to use in the q-expansion of $\wp$.
- domain (complex field) – the model of $\mathbf{C}$ to use, for example CDF of CIF (default).

OUTPUT:

A pair of functions fk, err which can be evaluated at complex numbers z (in the correct domain) to give an approximation to $\wp(z)$ and an upper bound on the error, respectively. The Weierstrass function returned is with respect to the normalised lattice $[1, \tau]$ associated to the given embedding.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: L = E.period_lattice()
sage: w1, w2 = L.normalised_basis()
sage: z = CDF(0.3, 0.4)
```

Compare the value give by the standard elliptic exponential (scaled since `fk` is with respect to the normalised lattice):

```
sage: L.elliptic_exponential(z*w2, to_curve=False)[0] * w2 ** 2
-1.82543539306049 - 2.49336319992847*I
```

to the value given by this function, and see the error:

```
sage: fk, err = E.height_function().fk_intervals(N=10)
sage: fk(CIF(z))
-1.825435393060497? - 2.493363199928477?*I
sage: err(CIF(z))
2.71750621458744e-31
```

The same, but in the domain `CDF` instead of `CIF`:

```
sage: fk, err = E.height_function().fk_intervals(N=10, domain=CDF)
sage: fk(z)
-1.8254353930604... - 2.493363199928...*I
```

min (*tol*, *n_max*, *verbose=False*)

Returns a lower bound for all points of infinite order.

INPUT:

- `tol` – tolerance in output (see below).
- `n_max` – how many multiples to use in iteration.
- `verbose` (boolean, default: `False`) – verbosity flag.

OUTPUT:

A positive real μ for which it has been established rigorously that every point of infinite order on the elliptic curve (defined over its ground field) has canonical height greater than μ, and such that it is not possible (at least without increasing `n_max`) to prove the same for $\mu \cdot$ tol.

EXAMPLES:

Example 1 from [CS2006] (where the same lower bound of 0.1126 was given):

```
sage: E = EllipticCurve([1, 0, 1, 421152067, 105484554028056]) # 60490d1
sage: E.height_function().min(.0001, 5)
0.0011263287309893311
```

Example 10.1 from [Tho2010] (where a lower bound of 0.18 was given):

```
sage: # needs sage.rings.number_field
sage: K.<i> = QuadraticField(-1)
sage: E = EllipticCurve([0, 0, 0, 91 - 26*i, -144 - 323*i])
sage: H = E.height_function()
sage: H.min(0.1, 4)                  # long time
0.1621049443313762
```

Example 10.2 from [Tho2010]:

```
sage: # needs sage.rings.number_field
sage: K.<i> = QuadraticField(-1)
sage: E = EllipticCurve([0, 1 - i, i, -i, 0])
sage: H = E.height_function()
sage: H.min(0.01, 5)                # long time
0.020153685521979152
```

In this example the point $P = (0, 0)$ has height 0.023 so our lower bound is quite good:

```
sage: P = E((0,0))                                                          #␣
↪needs sage.rings.number_field
sage: P.height()                                                            #␣
↪needs sage.rings.number_field
0.0230242154471211
```

Example 10.3 from [Tho2010] (where the same bound of 0.0625 is given):

```
sage: # needs sage.rings.number_field
sage: K.<a> = NumberField(x^3 - 2)
sage: E = EllipticCurve([0, 0, 0, -3*a - a^2, a^2])
sage: H = E.height_function()
sage: H.min(0.1, 5)                 # long time
0.0625
```

More examples over $\mathbf{Q}$:

```
sage: E = EllipticCurve('37a')
sage: h = E.height_function()
sage: h.min(.01, 5)
0.03987318057488725
sage: E.gen(0).height()
0.0511114082399688
```

After base change the lower bound can decrease:

```
sage: K.<a> = QuadraticField(-5)                                            #␣
↪needs sage.rings.number_field
sage: E.change_ring(K).height_function().min(0.5, 10)       # long time,␣
↪needs sage.rings.number_field
0.04419417382415922

sage: E = EllipticCurve('389a')
sage: h = E.height_function()
sage: h.min(0.1, 5)
0.05731275270029196
sage: [P.height() for P in E.gens()]
[0.686667083305587, 0.327000773651605]
```

min_gr (*tol*, *n_max*, *verbose=False*)

Returns a lower bound for points of infinite order with good reduction.

INPUT:

- `tol` – tolerance in output (see below).
- `n_max` – how many multiples to use in iteration.
- `verbose` (boolean, default: `False`) – verbosity flag.

OUTPUT:

A positive real μ for which it has been established rigorously that every point of infinite order on the elliptic curve (defined over its ground field), which has good reduction at all primes, has canonical height greater than μ, and such that it is not possible (at least without increasing `n_max`) to prove the same for $\mu \cdot$ tol.

EXAMPLES:

Example 1 from [CS2006] (where a lower bound of 1.9865 was given):

```
sage: E = EllipticCurve([1, 0, 1, 421152067, 105484554028056]) # 60490d1
sage: E.height_function().min_gr(.0001, 5)
1.98684388146518
```

Example 10.1 from [Tho2010] (where a lower bound of 0.18 was given):

```
sage: # needs sage.rings.number_field
sage: K.<i> = QuadraticField(-1)
sage: E = EllipticCurve([0, 0, 0, 91 - 26*i, -144 - 323*i])
sage: H = E.height_function()
sage: H.min_gr(0.1, 4)              # long time
0.1621049443313762
```

Example 10.2 from [Tho2010]:

```
sage: # needs sage.rings.number_field
sage: K.<i> = QuadraticField(-1)
sage: E = EllipticCurve([0, 1 - i, i, -i, 0])
sage: H = E.height_function()
sage: H.min_gr(0.01, 5)             # long time
0.020153685521979152
```

In this example the point $P = (0, 0)$ has height 0.023 so our lower bound is quite good:

```
sage: P = E((0,0))                                                          #␣
↪needs sage.rings.number_field
sage: P.has_good_reduction()                                                #␣
↪needs sage.rings.number_field
True
sage: P.height()                                                            #␣
↪needs sage.rings.number_field
0.0230242154471211
```

Example 10.3 from [Tho2010] (where the same bound of 0.25 is given):

```
sage: # needs sage.rings.number_field
sage: K.<a> = NumberField(x^3 - 2)
sage: E = EllipticCurve([0, 0, 0, -3*a - a^2, a^2])
sage: H = E.height_function()
sage: H.min_gr(0.1, 5)              # long time
0.25
```

psi (*xi*, *v*)

Return the normalised elliptic log of a point with this x-coordinate.

INPUT:

- `xi` (real) – the real x-coordinate of a point on the curve in the connected component with respect to a real embedding.

- v (embedding) – a real embedding of the number field.

OUTPUT:

A real number in the interval [0.5,1] giving the elliptic logarithm of a point on E with x-coordinate xi, on the connected component with respect to the embedding v, scaled by the real period.

EXAMPLES:

An example over **Q**:

```
sage: E = EllipticCurve('389a')
sage: v = QQ.places()[0]
sage: L = E.period_lattice(v)
sage: P = E.lift_x(10/9)
sage: L(P)
0.958696500380439
sage: L(P) / L.real_period()
0.384985810227885
sage: H = E.height_function()
sage: H.psi(10/9, v)
0.615014189772115
```

An example over a number field:

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)
sage: E = EllipticCurve([0,0,0,0,a])
sage: P = E.lift_x(1/3*a^2 + a + 5/3)
sage: v = K.real_places()[0]
sage: L = E.period_lattice(v)
sage: L(P)
3.51086196882538
sage: L(P) / L.real_period()
0.867385122699931
sage: xP = v(P.x())
sage: H = E.height_function()
sage: H.psi(xP, v)
0.867385122699931
sage: H.psi(1.23, v)
0.785854718241495
```

real_intersection_is_empty(*Bk*, *v*)

Returns True iff an intersection of $S_n^{(v)}$ sets is empty.

INPUT:

- Bk (list) – a list of reals.
- v (embedding) – a real embedding of the number field.

OUTPUT:

True or False, according as the intersection of the unions of intervals $S_n^{(v)}(-b, b)$ for b in the list Bk is empty or not. When Bk is the list of $b = B_n(\mu)$ for $n = 1, 2, 3, \ldots$ for some $\mu > 0$ this means that all non-torsion points on E with everywhere good reduction have canonical height strictly greater than μ, by [Tho2010], Proposition 6.2.

EXAMPLES:

An example over **Q**:

```
sage: E = EllipticCurve('389a')
sage: v = QQ.places()[0]
sage: H = E.height_function()
```

The following two lines prove that the heights of non-torsion points on E with everywhere good reduction have canonical height strictly greater than 0.2, but fail to prove the same for 0.3:

```
sage: H.real_intersection_is_empty([H.B(n,0.2) for n in srange(1,10)], v)
True
sage: H.real_intersection_is_empty([H.B(n,0.3) for n in srange(1,10)], v)
False
```

An example over a number field:

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)
sage: E = EllipticCurve([0,0,0,0,a])
sage: v = K.real_places()[0]
sage: H = E.height_function()
```

The following two lines prove that the heights of non-torsion points on E with everywhere good reduction have canonical height strictly greater than 0.07, but fail to prove the same for 0.08:

```
sage: H.real_intersection_is_empty([H.B(n,0.07) for n in srange(1,5)], v)  #␣
↪long time, needs sage.rings.number_field
True
sage: H.real_intersection_is_empty([H.B(n,0.08) for n in srange(1,5)], v)   #␣
↪needs sage.rings.number_field
False
```

tau(*v*)

Return the normalised upper half-plane parameter τ for the period lattice with respect to the embedding v.

INPUT:

- v (embedding) – a real or complex embedding of the number field.

OUTPUT:

(Complex) $\tau = \omega_1/\omega_2$ in the fundamental region of the upper half-plane.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: H = E.height_function()
sage: H.tau(QQ.places()[0])
1.22112736076463*I
```

test_mu(*mu*, *N*, *verbose=True*)

Return True if we can prove that μ is a lower bound.

INPUT:

- mu (real) – a positive real number
- N (integer) – upper bound on the multiples to be used.
- verbose (boolean, default: True) – verbosity flag.

OUTPUT:

`True` or `False`, according to whether we succeed in proving that μ is a lower bound for the canonical heights of points of infinite order with everywhere good reduction.

Note: A `True` result is rigorous; `False` only means that the attempt failed: trying again with larger N may yield `True`.

EXAMPLES:

```
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)                                          #␣
↪needs sage.rings.number_field
sage: E = EllipticCurve([0,0,0,0,a])                                        #␣
↪needs sage.rings.number_field
sage: H = E.height_function()                                               #␣
↪needs sage.rings.number_field
```

This curve does have a point of good reduction whose canonical point is approximately 1.68:

```
sage: P = E.gens(lim3=5)[0]; P                                              #␣
↪needs sage.rings.number_field
(1/3*a^2 + a + 5/3 : -2*a^2 - 4/3*a - 5/3 : 1)
sage: P.height()                                                            #␣
↪needs sage.rings.number_field
1.68038085233673
sage: P.has_good_reduction()                                                #␣
↪needs sage.rings.number_field
True
```

Using $N = 5$ we can prove that 0.1 is a lower bound (in fact we only need $N = 2$), but not that 0.2 is:

```
sage: H.test_mu(0.1, 5)                                                     #␣
↪needs sage.rings.number_field
B_1(0.100000000000000) = 1.51580969677387
B_2(0.100000000000000) = 0.932072561526720
True
sage: H.test_mu(0.2, 5)                                                     #␣
↪needs sage.rings.number_field
B_1(0.200000000000000) = 2.04612906979932
B_2(0.200000000000000) = 3.09458988474327
B_3(0.200000000000000) = 27.6251108409484
B_4(0.200000000000000) = 1036.24722370223
B_5(0.200000000000000) = 3.67090854562318e6
False
```

Since 0.1 is a lower bound we can deduce that the point P is either primitive or divisible by either 2 or 3. In fact it is primitive:

```
sage: (P.height()/0.1).sqrt()                                               #␣
↪needs sage.rings.number_field
4.09924487233530
sage: P.division_points(2)                                                  #␣
↪needs sage.rings.number_field
[]
sage: P.division_points(3)                                                  #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.rings.number_field
[]
```

wp_c(*v*)

Return a bound for the Weierstrass $\wp$-function.

INPUT:

- v (embedding) – a real or complex embedding of the number field.

OUTPUT:

(Real) $c > 0$ such that

$$|\wp(z) - z^{-}2| \leq \frac{c^2|z|^2}{1 - c|z|^2}$$

whenever $c|z|^2 < 1$. Given the recurrence relations for the Laurent series expansion of $\wp$, it is easy to see that there is such a constant c. [Reference?]

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: H = E.height_function()
sage: H.wp_c(QQ.places()[0])
2.68744508779950

sage: # needs sage.rings.number_field
sage: K.<i> = QuadraticField(-1)
sage: E = EllipticCurve([0, 0, 0, 1 + 5*i, 3 + i])
sage: H = E.height_function()
sage: H.wp_c(K.places()[0])
2.66213425640096
```

wp_intervals(*v=None*, *N=20*, *abs_only=False*)

Return a function approximating the Weierstrass function.

INPUT:

- v (embedding) – an embedding of the number field. If None (default) use the real embedding if the field is $\mathbf{Q}$ and raise an error for other fields.
- N (int, default 20) – The number of terms to use in the q-expansion of $\wp$.
- abs_only (boolean, default: False) – flag to determine whether (if True) the error adjustment should use the absolute value or (if False) the real and imaginary parts.

OUTPUT:

A function wp which can be evaluated at complex numbers z to give an approximation to $\wp(z)$. The Weierstrass function returned is with respect to the normalised lattice $[1, \tau]$ associated to the given embedding. For z which are not near a lattice point the function fk is used, otherwise a better approximation is used.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: wp = E.height_function().wp_intervals()
sage: z = CDF(0.3, 0.4)
sage: wp(CIF(z))
-1.825435393060493? - 2.493363199928?*I
```

(continues on next page)

(continued from previous page)

```
sage: L = E.period_lattice()
sage: w1, w2 = L.normalised_basis()
sage: L.elliptic_exponential(z*w2, to_curve=False)[0] * w2^2
-1.82543539306049 - 2.49336319992847*I

sage: z = CDF(0.3, 0.1)
sage: wp(CIF(z))
8.5918243572165? - 5.4751982004351?*I
sage: L.elliptic_exponential(z*w2, to_curve=False)[0] * w2^2
8.59182435721650 - 5.47519820043503*I
```

wp_on_grid(*v*, *N*, *half=False*)

Return an array of the values of $\wp$ on an $N \times N$ grid.

INPUT:

- `v` (embedding) – an embedding of the number field.
- `N` (int) – The number of terms to use in the q-expansion of $\wp$.
- `half` (boolean, default: `False`) – if True, use an array of size $N \times N/2$ instead of $N \times N$.

OUTPUT:

An array of size either $N \times N/2$ or $N \times N$ whose (i, j) entry is the value of the Weierstrass $\wp$-function at $(i + .5)/N + (j + .5) * \tau/N$, a grid of points in the fundamental region for the lattice $[1, \tau]$.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: H = E.height_function()
sage: v = QQ.places()[0]
```

The array of values on the grid shows symmetry, since $\wp$ is even:

```
sage: H.wp_on_grid(v,4)
array([[25.43920182,  5.28760943,  5.28760943, 25.43920182],
       [ 6.05099485,  1.83757786,  1.83757786,  6.05099485],
       [ 6.05099485,  1.83757786,  1.83757786,  6.05099485],
       [25.43920182,  5.28760943,  5.28760943, 25.43920182]])
```

The array of values on the half-grid:

```
sage: H.wp_on_grid(v,4,True)
array([[25.43920182,  5.28760943],
       [ 6.05099485,  1.83757786],
       [ 6.05099485,  1.83757786],
       [25.43920182,  5.28760943]])
```

class sage.schemes.elliptic_curves.height.**UnionOfIntervals**(*endpoints*)

Bases: `object`

A class representing a finite union of closed intervals in **R** which can be scaled, shifted, intersected, etc.

The intervals are represented as an ordered list of their endpoints, which may include $-\infty$ and $+\infty$.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.height import UnionOfIntervals
sage: R = UnionOfIntervals([1, 2, 3, infinity]); R
([1, 2] U [3, +Infinity])
sage: R + 5
([6, 7] U [8, +Infinity])
sage: ~R
([-Infinity, 1] U [2, 3])
sage: ~R | (10*R + 100)
([-Infinity, 1] U [2, 3] U [110, 120] U [130, +Infinity])
```

Todo: Unify *UnionOfIntervals* with the class RealSet introduced by Issue #13125; see Issue #16063.

finite_endpoints()

Returns the finite endpoints of this union of intervals.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.height import UnionOfIntervals
sage: UnionOfIntervals([0, 1]).finite_endpoints()
[0, 1]
sage: UnionOfIntervals([-infinity, 0, 1, infinity]).finite_endpoints()
[0, 1]
```

classmethod intersection(*L*)

Return the intersection of a list of UnionOfIntervals.

INPUT:

- L (list) – a list of UnionOfIntervals instances

OUTPUT:

A new UnionOfIntervals instance representing the intersection of the UnionOfIntervals in the list.

Note: This is a class method.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.height import UnionOfIntervals
sage: A = UnionOfIntervals([1,3,5,7]); A
([1, 3] U [5, 7])
sage: B = A + 1; B
([2, 4] U [6, 8])
sage: A.intersection([A,B])
([2, 3] U [6, 7])
```

intervals()

Returns the intervals in self, as a list of 2-tuples.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.height import UnionOfIntervals
sage: UnionOfIntervals(list(range(10))).intervals()
[(0, 1), (2, 3), (4, 5), (6, 7), (8, 9)]
sage: UnionOfIntervals([-infinity, pi, 17, infinity]).intervals()           #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.symbolic
[(-Infinity, pi), (17, +Infinity)]
```

is_empty()

Returns whether self is empty.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.height import UnionOfIntervals
sage: UnionOfIntervals([3, 4]).is_empty()
False
sage: all = UnionOfIntervals([-infinity, infinity])
sage: all.is_empty()
False
sage: (~all).is_empty()
True
sage: A = UnionOfIntervals([0, 1]) & UnionOfIntervals([2, 3])
sage: A.is_empty()
True
```

static join(*L*, *condition*)

Utility function to form the union or intersection of a list of UnionOfIntervals.

INPUT:

- `L` (list) – a list of UnionOfIntervals instances
- `condition` (function) – either `any` or `all`, or some other boolean function of a list of boolean values.

OUTPUT:

A new UnionOfIntervals instance representing the subset of 'RR' equal to those reals in any/all/condition of the UnionOfIntervals in the list.

Note: This is a static method for the class.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.height import UnionOfIntervals
sage: A = UnionOfIntervals([1,3,5,7]); A
([1, 3] U [5, 7])
sage: B = A + 1; B
([2, 4] U [6, 8])
sage: A.join([A,B], any) # union
([1, 4] U [5, 8])
sage: A.join([A,B], all) # intersection
([2, 3] U [6, 7])
sage: A.join([A,B], sum) # symmetric difference
([1, 2] U [3, 4] U [5, 6] U [7, 8])
```

classmethod union(*L*)

Return the union of a list of UnionOfIntervals.

INPUT:

- `L` (list) – a list of UnionOfIntervals instances

OUTPUT:

A new UnionOfIntervals instance representing the union of the UnionOfIntervals in the list.

Note: This is a class method.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.height import UnionOfIntervals
sage: A = UnionOfIntervals([1,3,5,7]); A
([1, 3] U [5, 7])
sage: B = A + 1; B
([2, 4] U [6, 8])
sage: A.union([A,B])
([1, 4] U [5, 8])
```

sage.schemes.elliptic_curves.height.**eps**(*err*, *is_real*)

Return a Real or Complex interval centered on 0 with radius err.

INPUT:

- `err` (real) – a positive real number, the radius of the interval
- `is_real` (boolean) – if True, returns a real interval in RIF, else a complex interval in CIF

OUTPUT:

An element of RIF or CIF (as specified), centered on 0, with given radius.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.height import eps
sage: eps(0.01, True)
0.0?
sage: eps(0.01, False)
0.0? + 0.0?*I
```

sage.schemes.elliptic_curves.height.**inf_max_abs**(f, g, D)

Returns $\inf_D(\max(|f|,|g|))$.

INPUT:

- `f`, `g` (polynomials) – real univariate polynomials
- `D` (*UnionOfIntervals*) – a subset of $\mathbf{R}$

OUTPUT:

A real number approximating the value of $\inf_D(\max(|f|,|g|))$.

ALGORITHM:

The extreme values must occur at an endpoint of a subinterval of D or at a point where one of f, f', g, g', $f \pm g$ is zero.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.height import inf_max_abs,␣
↪UnionOfIntervals
sage: x = polygen(RR)
sage: f = (x-10)^4 + 1
```

(continues on next page)

(continued from previous page)

```
sage: g = 2*x^3 + 100
sage: inf_max_abs(f, g, UnionOfIntervals([1,2,3,4,5,6]))
425.638201706391
sage: r0 = (f - g).roots()[0][0]
sage: r0
5.46053402234697
sage: max(abs(f(r0)), abs(g(r0)))
425.638201706391
```

sage.schemes.elliptic_curves.height.**min_on_disk**(*f*, *tol*, *max_iter=10000*)

Returns the minimum of a real-valued complex function on a square.

INPUT:

- `f` – a function from CIF to RIF
- `tol` (real) – a positive real number
- `max_iter` (integer, default 10000) – a positive integer bounding the number of iterations to be used

OUTPUT:

A 2-tuple (s, t), where $t = f(s)$ and s is a CIF element contained in the disk $|z| \leq 1$, at which f takes its minimum value.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.height import min_on_disk
sage: f = lambda x: (x^2 + 100).abs()
sage: s, t = min_on_disk(f, 0.0001)
sage: s, f(s), t
(0.01? + 1.00?*I, 99.01?, 99.0000000000000)
```

sage.schemes.elliptic_curves.height.**nonneg_region**(*f*)

Returns the UnionOfIntervals representing the region where `f` is non-negative.

INPUT:

- `f` (polynomial) – a univariate polynomial over $\mathbf{R}$.

OUTPUT:

A UnionOfIntervals representing the set $\{x \in \mathbf{R} midf(x) \geq 0\}$.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.height import nonneg_region
sage: x = polygen(RR)
sage: nonneg_region(x^2 - 1)
([-Infinity, -1.00000000000000] U [1.00000000000000, +Infinity])
sage: nonneg_region(1 - x^2)
([-1.00000000000000, 1.00000000000000])
sage: nonneg_region(1 - x^3)
([-Infinity, 1.00000000000000])
sage: nonneg_region(x^3 - 1)
([1.00000000000000, +Infinity])
sage: nonneg_region((x-1)*(x-2))
([-Infinity, 1.00000000000000] U [2.00000000000000, +Infinity])
sage: nonneg_region(-(x-1)*(x-2))
([1.00000000000000, 2.00000000000000])
```

(continues on next page)

(continued from previous page)

```
sage: nonneg_region((x-1)*(x-2)*(x-3))
([1.00000000000000, 2.00000000000000] U [3.00000000000000, +Infinity])
sage: nonneg_region(-(x-1)*(x-2)*(x-3))
([-Infinity, 1.00000000000000] U [2.00000000000000, 3.00000000000000])
sage: nonneg_region(x^4 + 1)
([-Infinity, +Infinity])
sage: nonneg_region(-x^4 - 1)
()
```

sage.schemes.elliptic_curves.height.**rat_term_CIF**(*z*, *try_strict=True*)

Compute the value of $u/(1-u)^2$ in `CIF`, where $u = \exp(2\pi i z)$.

INPUT:

- `z` (complex) – a CIF element
- `try_strict` (bool) – flag

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.height import rat_term_CIF
sage: z = CIF(0.5,0.2)
sage: rat_term_CIF(z)
-0.172467461182437? + 0.?e-16*I
sage: rat_term_CIF(z, False)
-0.172467461182437? + 0.?e-16*I
```

18.5 Saturation of Mordell-Weil groups of elliptic curves over number fields

Points $P_1, \ldots, P_r$ in $E(K)$, where E is an elliptic curve over a number field K, are said to be p-saturated if no linear combination $\sum n_i P_i$ is divisible by p in $E(K)$ except trivially when all n_i are multiples of p. The points are said to be saturated if they are p-saturated at all primes; this is always true for all but finitely many primes since $E(K)$ is a finitely-generated Abelian group.

The process of p-saturating a given set of points is implemented here. The naive algorithm simply checks all $(p^r-1)/(p-1)$ projective combinations of the points, testing each to see if it can be divided by p. If this occurs then we replace one of the points and continue. The function `p_saturation()` does one step of this, while `full_p_saturation()` repeats until the points are p-saturated. A more sophisticated algorithm for p-saturation is implemented which is much more efficient for large p and r, and involves computing the reduction of the points modulo auxiliary primes to obtain linear conditions modulo p which must be satisfied by the coefficients a_i of any nontrivial relation. When the points are already p-saturated this sieving technique can prove their saturation quickly.

The method `saturation()` of the class `EllipticCurve_number_field` applies full p-saturation at any given set of primes, or can compute a bound on the primes p at which the given points may not be p-saturated. This involves computing a lower bound for the canonical height of points of infinite order, together with estimates from the geometry of numbers.

AUTHORS:

- Robert Bradshaw
- John Cremona

class sage.schemes.elliptic_curves.saturation.**EllipticCurveSaturator**(*E*, *verbose=False*)

Bases: SageObject

Class for saturating points on an elliptic curve over a number field.

INPUT:

- E – an elliptic curve defined over a number field, or **Q**.
- verbose (boolean, default False) – verbosity flag.

Note: This function is not normally called directly by users, who may access the data via methods of the EllipticCurve classes.

---

add_reductions(*q*)

Add reduction data at primes above q if not already there.

INPUT:

- q – a prime number not dividing the defining polynomial of self.__field.

OUTPUT:

Returns nothing, but updates self._reductions dictionary for key q to a dict whose keys are the roots of the defining polynomial mod q and values tuples (nq, Eq) where Eq is an elliptic curve over $GF(q)$ and nq its cardinality. If q divides the conductor norm or order discriminant nothing is added.

EXAMPLES:

Over **Q**:

```
sage: from sage.schemes.elliptic_curves.saturation import␣
↪EllipticCurveSaturator
sage: E = EllipticCurve('11a1')
sage: saturator = EllipticCurveSaturator(E)
sage: saturator._reductions
{}
sage: saturator.add_reductions(19)
sage: saturator._reductions
{19: {0: (20, Elliptic Curve defined by y^2 + y = x^3 + 18*x^2 + 9*x + 18
              over Finite Field of size 19)}}
```

Over a number field:

```
sage: x = polygen(QQ);  K.<a> = NumberField(x^2 + 2)
sage: E = EllipticCurve(K, [0,1,0,a,a])
sage: from sage.schemes.elliptic_curves.saturation import␣
↪EllipticCurveSaturator
sage: saturator = EllipticCurveSaturator(E)
sage: for q in primes(20):
....:     saturator.add_reductions(q)
sage: saturator._reductions
{2:  {},
 3:  {},
 5:  {},
 7:  {},
 11: {3:  (16, Elliptic Curve defined by y^2 = x^3 + x^2 + 3*x + 3
               over Finite Field of size 11),
      8:  (8,  Elliptic Curve defined by y^2 = x^3 + x^2 + 8*x + 8
```

(continues on next page)

(continued from previous page)

```
                over Finite Field of size 11)},
 13: {},
 17: {7:  (20, Elliptic Curve defined by y^2 = x^3 + x^2 + 7*x + 7
                over Finite Field of size 17),
      10: (18, Elliptic Curve defined by y^2 = x^3 + x^2 + 10*x + 10
                over Finite Field of size 17)},
 19: {6:  (16, Elliptic Curve defined by y^2 = x^3 + x^2 + 6*x + 6
                over Finite Field of size 19),
      13: (12, Elliptic Curve defined by y^2 = x^3 + x^2 + 13*x + 13
                over Finite Field of size 19)}}
```

full_p_saturation(*Plist*, *p*)

Full p-saturation of `Plist`.

INPUT:

- `Plist` (list) – a list of independent points on one elliptic curve.
- `p` (integer) – a prime number.

OUTPUT:

(`newPlist`, `exponent`) where `newPlist` has the same length as `Plist` and spans the p-saturation of the span of `Plist`, which contains that span with index `p**exponent`.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.saturation import␣
↪EllipticCurveSaturator
sage: E = EllipticCurve('389a')
sage: K.<i> = QuadraticField(-1)
sage: EK = E.change_ring(K)
sage: P = EK(1 + i, -1 - 2*i)
sage: saturator = EllipticCurveSaturator(EK, verbose=True)
sage: saturator.full_p_saturation([8*P], 2)
 --starting full 2-saturation
Points were not 2-saturated, exponent was 3
([(i + 1 : -2*i - 1 : 1)], 3)

sage: Q = EK(0, 0)
sage: R = EK(-1, 1)
sage: saturator = EllipticCurveSaturator(EK, verbose=False)
sage: saturator.full_p_saturation([P, Q, R], 3)
([(i + 1 : -2*i - 1 : 1), (0 : 0 : 1), (-1 : 1 : 1)], 0)
```

An example where the points are not 7-saturated and we gain index exponent 1. Running this example with verbose=True would show that it uses the code for when the reduction has p-rank 2 (which occurs for the reduction modulo $(16 - 5i)$), which uses the Weil pairing:

```
sage: saturator.full_p_saturation([P, Q + 3*R, Q - 4*R], 7)
([(i + 1 : -2*i - 1 : 1),
  (2869/676 : 154413/17576 : 1),
  (-7095/502681 : -366258864/356400829 : 1)], 1)
```

p_saturation(*Plist*, *p*, *sieve=True*)

Checks whether the list of points is p-saturated.

INPUT:

- `Plist` (list) – a list of independent points on one elliptic curve.
- `p` (integer) – a prime number.
- `sieve` (boolean) – if True, use a sieve (when there are at least 2 points); otherwise test all combinations.

Note: The sieve is much more efficient when the points are saturated and the number of points or the prime are large.

OUTPUT:

Either `False` if the points are p-saturated, or (`i`, `newP`) if they are not p-saturated, in which case after replacing the i'th point with `newP`, the subgroup generated contains that generated by `Plist` with index p.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.saturation import␣
↪EllipticCurveSaturator
sage: E = EllipticCurve('389a')
sage: K.<i> = QuadraticField(-1)
sage: EK = E.change_ring(K)
sage: P = EK(1 + i, -1 - 2*i)
sage: saturator = EllipticCurveSaturator(EK)
sage: saturator.p_saturation([P], 2)
False
sage: saturator.p_saturation([2*P], 2)
(0, (i + 1 : -2*i - 1 : 1))

sage: Q = EK(0, 0)
sage: R = EK(-1, 1)
sage: saturator.p_saturation([P, Q, R], 3)
False
```

Here we see an example where 19-saturation is proved, with the verbose flag set to True so that we can see what is going on:

```
sage: saturator = EllipticCurveSaturator(EK, verbose=True)
sage: saturator.p_saturation([P, Q, R], 19)
Using sieve method to saturate...
E has 19-torsion over Finite Field of size 197, projecting points
--> [(15 : 168 : 1), (0 : 0 : 1), (196 : 1 : 1)]
--rank is now 1
E has 19-torsion over Finite Field of size 197, projecting points
--> [(184 : 27 : 1), (0 : 0 : 1), (196 : 1 : 1)]
--rank is now 2
E has 19-torsion over Finite Field of size 293, projecting points
--> [(139 : 16 : 1), (0 : 0 : 1), (292 : 1 : 1)]
 --rank is now 3
Reached full rank: points were 19-saturated
False
```

An example where the points are not 11-saturated:

```
sage: saturator = EllipticCurveSaturator(EK, verbose=False)
sage: res = saturator.p_saturation([P + 5*Q, P - 6*Q, R], 11); res
(0, (-5783311/14600041*i + 1396143/14600041
     : 37679338314/55786756661*i + 3813624227/55786756661 : 1))
```

That means that the 0'th point may be replaced by the displayed point to achieve an index gain of 11:

```
sage: saturator.p_saturation([res[1], P - 6*Q, R], 11)
False
```

sage.schemes.elliptic_curves.saturation.**p_projections**(*Eq*, *Plist*, *p*, *debug=False*)

INPUT:

- `Eq` – An elliptic curve over a finite field.
- `Plist` – a list of points on Eq.
- `p` – a prime number.

OUTPUT:

A list of $r \leq 2$ vectors in $\mathbf{F}_{p^n}$, the images of the points in $G \otimes \mathbf{F}_p$, where r is the number of vectors is the p-rank of Eq.

ALGORITHM:

First project onto the p-primary part of Eq. If that has p-rank 1 (i.e. is cyclic), use discrete logs there to define a map to $\mathbf{F}_p$, otherwise use the Weil pairing to define two independent maps to $\mathbf{F}_p$.

EXAMPLES:

This curve has three independent rational points:

```
sage: E = EllipticCurve([0,0,1,-7,6])
```

We reduce modulo 409 where its order is $3^2 \cdot 7^2$; the 3-primary part is non-cyclic while the 7-primary part is cyclic of order 49:

```
sage: F = GF(409)
sage: EF = E.change_ring(F)
sage: G = EF.abelian_group()
sage: G
Additive abelian group isomorphic to Z/147 + Z/3
 embedded in Abelian group of points on Elliptic Curve
  defined by y^2 + y = x^3 + 402*x + 6 over Finite Field of size 409
sage: G.order().factor()
3^2 * 7^2
```

We construct three points and project them to the p-primary parts for $p = 2, 3, 5, 7$, yielding 0,2,0,1 vectors of length 3 modulo p respectively. The exact vectors output depend on the computed generators of G:

```
sage: Plist = [EF([-2,3]), EF([0,2]), EF([1,0])]
sage: from sage.schemes.elliptic_curves.saturation import p_projections
sage: [(p, p_projections(EF, Plist, p)) for p in primes(11)]  # random
[(2, []), (3, [(0, 2, 2), (2, 2, 1)]), (5, []), (7, [(5, 1, 1)])]
sage: [(p, len(p_projections(EF, Plist, p))) for p in primes(11)]
[(2, 0), (3, 2), (5, 0), (7, 1)]
```

sage.schemes.elliptic_curves.saturation.**reduce_mod_q**(*x*, *amodq*)

The reduction of `x` modulo the prime ideal defined by `amodq`.

INPUT:

- `x` – an element of a number field K.
- `amodq` – an element of $GF(q)$ which is a root mod q of the defining polynomial of K. This defines a degree 1 prime ideal $Q = (q, \alpha - a)$ of $K = \mathbf{Q}(\alpha)$, where a mod q = `amodq`.

OUTPUT:

The image of x in the residue field of K at the prime Q.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.saturation import reduce_mod_q
sage: x = polygen(QQ)
sage: pol = x^3 - x^2 - 3*x + 1
sage: K.<a> = NumberField(pol)
sage: [(q, [(amodq, reduce_mod_q(1 - a + a^4, amodq))
....:       for amodq in sorted(pol.roots(GF(q), multiplicities=False))])
....:  for q in primes(50, 70)]
[(53, []),
 (59, [(36, 28)]),
 (61, [(40, 35)]),
 (67, [(10, 8), (62, 28), (63, 60)])]
```

18.6 Torsion subgroups of elliptic curves over number fields (including Q)

AUTHORS:

- Nick Alexander: original implementation over **Q**
- Chris Wuthrich: original implementation over number fields
- John Cremona: rewrote p-primary part to use division polynomials, added some features, unified Number Field and **Q** code.

class sage.schemes.elliptic_curves.ell_torsion.**EllipticCurveTorsionSubgroup**(E)

Bases: AdditiveAbelianGroupWrapper

The torsion subgroup of an elliptic curve over a number field.

EXAMPLES:

Examples over **Q**:

```
sage: E = EllipticCurve([-4, 0]); E
Elliptic Curve defined by y^2 = x^3 - 4*x over Rational Field
sage: G = E.torsion_subgroup(); G
Torsion Subgroup isomorphic to Z/2 + Z/2 associated to the
 Elliptic Curve defined by y^2 = x^3 - 4*x over Rational Field
sage: G.order()
4
sage: G.gen(0)
(-2 : 0 : 1)
sage: G.gen(1)
(0 : 0 : 1)
sage: G.ngens()
2
```

```
sage: E = EllipticCurve([17, -120, -60, 0, 0]); E
Elliptic Curve defined by y^2 + 17*x*y - 60*y = x^3 - 120*x^2 over Rational Field
sage: G = E.torsion_subgroup(); G
Torsion Subgroup isomorphic to Trivial group associated to the
```

(continues on next page)

(continued from previous page)

```
 Elliptic Curve defined by y^2 + 17*x*y - 60*y = x^3 - 120*x^2 over Rational Field
sage: G.gens()
()
sage: e = EllipticCurve([0, 330761566545336520666099468884, 0,
....:     34789753614434217964212032179072902312771611933875860480 0,
....:     ␣
↪1141128154369274295519023032806804247788154621049857648870032370285851781352816640000])
sage: e.torsion_order()
16
```

Constructing points from the torsion subgroup:

```
sage: E = EllipticCurve('14a1')
sage: T = E.torsion_subgroup()
sage: [E(t) for t in T]
[(0 : 1 : 0),
 (9 : 23 : 1),
 (2 : 2 : 1),
 (1 : -1 : 1),
 (2 : -5 : 1),
 (9 : -33 : 1)]
```

An example where the torsion subgroup is not cyclic:

```
sage: E = EllipticCurve([0,0,0,-49,0])
sage: T = E.torsion_subgroup()
sage: [E(t) for t in T]
[(0 : 1 : 0), (0 : 0 : 1), (-7 : 0 : 1), (7 : 0 : 1)]
```

An example where the torsion subgroup is trivial:

```
sage: E = EllipticCurve('37a1')
sage: T = E.torsion_subgroup()
sage: T
Torsion Subgroup isomorphic to Trivial group associated to the
 Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
sage: [E(t) for t in T]
[(0 : 1 : 0)]
```

Examples over other Number Fields:

```
sage: # needs sage.rings.number_field
sage: E = EllipticCurve('11a1')
sage: x = polygen(ZZ, 'x')
sage: K.<i> = NumberField(x^2 + 1)
sage: EK = E.change_ring(K)
sage: from sage.schemes.elliptic_curves.ell_torsion import␣
↪EllipticCurveTorsionSubgroup
sage: EllipticCurveTorsionSubgroup(EK)
Torsion Subgroup isomorphic to Z/5 associated to the
 Elliptic Curve defined by y^2 + y = x^3 + (-1)*x^2 + (-10)*x + (-20)
  over Number Field in i with defining polynomial x^2 + 1

sage: E = EllipticCurve('11a1')
sage: K.<i> = NumberField(x^2 + 1)                                          #␣
↪needs sage.rings.number_field
sage: EK = E.change_ring(K)                                                 #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.rings.number_field
sage: T = EK.torsion_subgroup()                                                #␣
↪needs sage.rings.number_field
sage: T.ngens()
1
sage: T.gen(0)
(5 : -6 : 1)
```

Note: this class is normally constructed indirectly as follows:

```
sage: # needs sage.rings.number_field
sage: T = EK.torsion_subgroup(); T
Torsion Subgroup isomorphic to Z/5 associated to the
 Elliptic Curve defined by y^2 + y = x^3 + (-1)*x^2 + (-10)*x + (-20)
  over Number Field in i with defining polynomial x^2 + 1
sage: type(T)
<class 'sage.schemes.elliptic_curves.ell_torsion.EllipticCurveTorsionSubgroup_
↪with_category'>
```

AUTHORS:

- Nick Alexander: initial implementation over $\mathbf{Q}$.
- Chris Wuthrich: initial implementation over number fields.
- John Cremona: additional features and unification.

curve()

Return the curve of this torsion subgroup.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: E = EllipticCurve('11a1')
sage: x = polygen(ZZ, 'x')
sage: K.<i> = NumberField(x^2 + 1)
sage: EK = E.change_ring(K)
sage: T = EK.torsion_subgroup()
sage: T.curve() is EK
True
```

points()

Return a list of all the points in this torsion subgroup.

The list is cached.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<i> = NumberField(x^2 + 1)
sage: E = EllipticCurve(K, [0,0,0,1,0])
sage: tor = E.torsion_subgroup()
sage: tor.points()
[(0 : 1 : 0), (0 : 0 : 1), (-i : 0 : 1), (i : 0 : 1)]
```

sage.schemes.elliptic_curves.ell_torsion.**torsion_bound**(*E*, *number_of_places=20*)

Return an upper bound on the order of the torsion subgroup.

INPUT:

- `E` – an elliptic curve over **Q** or a number field
- `number_of_places` (positive integer, default = 20) – the number of places that will be used to find the bound

OUTPUT:

(integer) An upper bound on the torsion order.

ALGORITHM:

An upper bound on the order of the torsion group of the elliptic curve is obtained by counting points modulo several primes of good reduction. Note that the upper bound returned by this function is a multiple of the order of the torsion group, and in general will be greater than the order.

To avoid nontrivial arithmetic in the base field (in particular, to avoid having to compute the maximal order) we only use prime P above rational primes p which do not divide the discriminant of the equation order.

EXAMPLES:

```
sage: CDB = CremonaDatabase()
sage: from sage.schemes.elliptic_curves.ell_torsion import torsion_bound
sage: [torsion_bound(E) for E in CDB.iter([14])]
[6, 6, 6, 6, 6, 6]
sage: [E.torsion_order() for E in CDB.iter([14])]
[6, 6, 2, 6, 2, 6]
```

An example over a relative number field (see Issue #16011):

```
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: F.<a> = QuadraticField(5)
sage: K.<b> = F.extension(x^2 - 3)
sage: E = EllipticCurve(K, [0,0,0,b,1])
sage: E.torsion_subgroup().order()
1
```

An example of a base-change curve from **Q** to a degree 16 field:

```
sage: # needs sage.rings.number_field
sage: from sage.schemes.elliptic_curves.ell_torsion import torsion_bound
sage: f = PolynomialRing(QQ,'x')([5643417737593488384,0,
....:     -11114515801179776,0,-455989850911004,0,379781901872,
....:     0,14339154953,0,-1564048,0,-194542,0,-32,0,1])
sage: K = NumberField(f,'a')
sage: E = EllipticCurve(K, [1, -1, 1, 824579, 245512517])
sage: torsion_bound(E)
16
sage: E.torsion_subgroup().invariants()
(4, 4)
```

18.7 Galois representations attached to elliptic curves

Given an elliptic curve E over $\mathbf{Q}$ and a rational prime number p, the p^n-torsion $E[p^n]$ points of E is a representation of the absolute Galois group $G_{\mathbf{Q}}$ of $\mathbf{Q}$. As n varies we obtain the Tate module T_pE which is a a representation of $G_{\mathbf{Q}}$ on a free $\mathbf{Z}_p$-module of rank 2. As p varies the representations are compatible.

Currently sage can decide whether the Galois module $E[p]$ is reducible, i.e., if E admits an isogeny of degree p, and whether the image of the representation on $E[p]$ is surjective onto $\mathrm{Aut}(E[p]) = GL_2(\mathbf{F}_p)$.

The following are the most useful functions for the class `GaloisRepresentation`.

For the reducibility:

- `is_reducible(p)`
- `is_irreducible(p)`
- `reducible_primes()`

For the image:

- `is_surjective(p)`
- `non_surjective()`
- `image_type(p)`

For the classification of the representation

- `is_semistable(p)`
- `is_unramified(p, ell)`
- `is_crystalline(p)`

EXAMPLES:

```
sage: E = EllipticCurve('196a1')
sage: rho = E.galois_representation()
sage: rho.is_irreducible(7)
True
sage: rho.is_reducible(3)
True
sage: rho.is_irreducible(2)
True
sage: rho.is_surjective(2)
False
sage: rho.is_surjective(3)
False
sage: rho.is_surjective(5)
True
sage: rho.reducible_primes()
[3]
sage: rho.non_surjective()
[2, 3]
sage: rho.image_type(2)
'The image is cyclic of order 3.'
sage: rho.image_type(3)
'The image is contained in a Borel subgroup as there is a 3-isogeny.'
sage: rho.image_type(5)
'The image is all of GL_2(F_5).'
```

For semi-stable curve it is known that the representation is surjective if and only if it is irreducible:

```
sage: E = EllipticCurve('11a1')
sage: rho = E.galois_representation()
sage: rho.non_surjective()
[5]
sage: rho.reducible_primes()
[5]
```

For cm curves it is not true that there are only finitely many primes for which the Galois representation mod p is surjective onto $GL_2(\mathbf{F}_p)$:

```
sage: E = EllipticCurve('27a1')
sage: rho = E.galois_representation()
sage: rho.non_surjective()
[0]
sage: rho.reducible_primes()
[3]
sage: E.has_cm()
True
sage: rho.image_type(11)
'The image is contained in the normalizer of a non-split Cartan group. (cm)'
```

REFERENCES:

- [Ser1972]
- [Ser1987]
- [Coj2005]

AUTHORS:

- chris wuthrich (02/10): moved from ell_rational_field.py.

class sage.schemes.elliptic_curves.gal_reps.**GaloisRepresentation**(E)

Bases: SageObject

The compatible family of Galois representation attached to an elliptic curve over the rational numbers.

Given an elliptic curve E over $\mathbf{Q}$ and a rational prime number p, the p^n-torsion $E[p^n]$ points of E is a representation of the absolute Galois group. As n varies we obtain the Tate module T_pE which is a representation of the absolute Galois group on a free $\mathbf{Z}_p$-module of rank 2. As p varies the representations are compatible.

EXAMPLES:

```
sage: rho = EllipticCurve('11a1').galois_representation()
sage: rho
Compatible family of Galois representations associated to the Elliptic Curve␣
↪defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field
```

elliptic_curve()

The elliptic curve associated to this representation.

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: rho = E.galois_representation()
sage: rho.elliptic_curve() == E
True
```

image_classes(*p*, *bound=10000*)

This function returns, given the representation ρ a list of p values that add up to 1, representing the frequency of the conjugacy classes of the projective image of ρ in $PGL_2(\mathbf{F}_p)$.

Let M be a matrix in $GL_2(\mathbf{F}_p)$, then define $u(M) = \mathrm{tr}(M)^2/\det(M)$, which only depends on the conjugacy class of M in $PGL_2(\mathbf{F}_p)$. Hence this defines a map $u : PGL_2(\mathbf{F}_p) \to \mathbf{F}_p$, which is almost a bijection between conjugacy classes of the source and $\mathbf{F}_p$ (the elements of order p and the identity map to 4 and both classes of elements of order 2 map to 0).

This function returns the frequency with which the values of u appeared among the images of the Frobenius elements a_ℓ`at`ℓ for good primes $\ell \neq p$ below a given `bound`.

INPUT:

- a prime `p`
- a natural number `bound` (default: 10000)

OUTPUT:

- a list of p real numbers in the interval $[0, 1]$ adding up to 1

EXAMPLES:

```
sage: E = EllipticCurve('14a1')
sage: rho = E.galois_representation()
sage: rho.image_classes(5)
[0.2095, 0.1516, 0.2445, 0.1728, 0.2217]

sage: E = EllipticCurve('11a1')
sage: rho = E.galois_representation()
sage: rho.image_classes(5)
[0.2467, 0.0000, 0.5049, 0.0000, 0.2484]
```

```
sage: EllipticCurve('27a1').galois_representation().image_classes(5)
[0.5839, 0.1645, 0.0000, 0.1702, 0.08143]
sage: EllipticCurve('30a1').galois_representation().image_classes(5)
[0.1956, 0.1801, 0.2543, 0.1728, 0.1972]
sage: EllipticCurve('32a1').galois_representation().image_classes(5)
[0.6319, 0.0000, 0.2492, 0.0000, 0.1189]
sage: EllipticCurve('900a1').galois_representation().image_classes(5)
[0.5852, 0.1679, 0.0000, 0.1687, 0.07824]
sage: EllipticCurve('441a1').galois_representation().image_classes(5)
[0.5860, 0.1646, 0.0000, 0.1679, 0.08150]
sage: EllipticCurve('648a1').galois_representation().image_classes(5)
[0.3945, 0.3293, 0.2388, 0.0000, 0.03749]
```

```
sage: EllipticCurve('784h1').galois_representation().image_classes(7)
[0.5049, 0.0000, 0.0000, 0.0000, 0.4951, 0.0000, 0.0000]
sage: EllipticCurve('49a1').galois_representation().image_classes(7)
[0.5045, 0.0000, 0.0000, 0.0000, 0.4955, 0.0000, 0.0000]

sage: EllipticCurve('121c1').galois_representation().image_classes(11)
[0.1001, 0.0000, 0.0000, 0.0000, 0.1017, 0.1953, 0.1993, 0.0000, 0.0000, 0.
↪2010, 0.2026]
sage: EllipticCurve('121d1').galois_representation().image_classes(11)
[0.08869, 0.07974, 0.08706, 0.08137, 0.1001, 0.09439, 0.09764, 0.08218, 0.
↪08625, 0.1017, 0.1009]
```

(continues on next page)

(continued from previous page)

```
sage: EllipticCurve('441f1').galois_representation().image_classes(13)
[0.08232, 0.1663, 0.1663, 0.1663, 0.08232, 0.0000, 0.1549, 0.0000, 0.0000, 0.
↪0000, 0.0000, 0.1817, 0.0000]
```

REMARKS:

Conjugacy classes of subgroups of $PGL_2(\mathbf{F}_5)$

For the case $p = 5$, the order of an element determines almost the value of u:

u	0	1	2	3	4
orders	2	3	4	6	1 or 5

Here we give here the full table of all conjugacy classes of subgroups with the values that `image_classes` should give (as `bound` tends to ∞). Comparing with the output of the above examples, it is now easy to guess what the image is.

subgroup	order	frequencies of values of u
trivial	1	[0.0000, 0.0000, 0.0000, 0.0000, 1.000]
cyclic	2	[0.5000, 0.0000, 0.0000, 0.0000, 0.5000]
cyclic	2	[0.5000, 0.0000, 0.0000, 0.0000, 0.5000]
cyclic	3	[0.0000, 0.6667, 0.0000, 0.0000, 0.3333]
Klein	4	[0.7500, 0.0000, 0.0000, 0.0000, 0.2500]
cyclic	4	[0.2500, 0.0000, 0.5000, 0.0000, 0.2500]
Klein	4	[0.7500, 0.0000, 0.0000, 0.0000, 0.2500]
cyclic	5	[0.0000, 0.0000, 0.0000, 0.0000, 1.000]
cyclic	6	[0.1667, 0.3333, 0.0000, 0.3333, 0.1667]
S_3	6	[0.5000, 0.3333, 0.0000, 0.0000, 0.1667]
S_3	6	[0.5000, 0.3333, 0.0000, 0.0000, 0.1667]
D_4	8	[0.6250, 0.0000, 0.2500, 0.0000, 0.1250]
D_5	10	[0.5000, 0.0000, 0.0000, 0.0000, 0.5000]
A_4	12	[0.2500, 0.6667, 0.0000, 0.0000, 0.08333]
D_6	12	[0.5833, 0.1667, 0.0000, 0.1667, 0.08333]
Borel	20	[0.2500, 0.0000, 0.5000, 0.0000, 0.2500]
S_4	24	[0.3750, 0.3333, 0.2500, 0.0000, 0.04167]
PSL_2	60	[0.2500, 0.3333, 0.0000, 0.0000, 0.4167]
PGL_2	120	[0.2083, 0.1667, 0.2500, 0.1667, 0.2083]

image_type(p)

Return a string describing the image of the mod-p representation. The result is provably correct, but only indicates what sort of an image we have. If one wishes to determine the exact group one needs to work a bit harder. The probabilistic method of image_classes or Sutherland's galrep package can give a very good guess what the image should be.

INPUT:

- `p` a prime number

OUTPUT: A string.

EXAMPLES:

```
sage: E = EllipticCurve('14a1')
sage: rho = E.galois_representation()
sage: rho.image_type(5)
'The image is all of GL_2(F_5).'

sage: E = EllipticCurve('11a1')
sage: rho = E.galois_representation()
sage: rho.image_type(5)
'The image is meta-cyclic inside a Borel subgroup as there is a 5-torsion␣
↪point on the curve.'

sage: EllipticCurve('27a1').galois_representation().image_type(5)
'The image is contained in the normalizer of a non-split Cartan group. (cm)'
sage: EllipticCurve('30a1').galois_representation().image_type(5)
'The image is all of GL_2(F_5).'
sage: EllipticCurve("324b1").galois_representation().image_type(5)
'The image in PGL_2(F_5) is the exceptional group S_4.'

sage: E = EllipticCurve([0,0,0,-56,4848])
sage: rho = E.galois_representation()

sage: rho.image_type(5)
'The image is contained in the normalizer of a split Cartan group.'

sage: EllipticCurve('49a1').galois_representation().image_type(7)
'The image is contained in a Borel subgroup as there is a 7-isogeny.'

sage: EllipticCurve('121c1').galois_representation().image_type(11)
'The image is contained in a Borel subgroup as there is a 11-isogeny.'
sage: EllipticCurve('121d1').galois_representation().image_type(11)
'The image is all of GL_2(F_11).'
sage: EllipticCurve('441f1').galois_representation().image_type(13)
'The image is contained in a Borel subgroup as there is a 13-isogeny.'

sage: EllipticCurve([1,-1,1,-5,2]).galois_representation().image_type(5)
'The image is contained in the normalizer of a non-split Cartan group.'
sage: EllipticCurve([0,0,1,-25650,1570826]).galois_representation().image_
↪type(5)
'The image is contained in the normalizer of a split Cartan group.'
sage: EllipticCurve([1,-1,1,-2680,-50053]).galois_representation().image_
↪type(7)    # the dots (...) in the output fix #11937 (installed 'Kash' may␣
↪give additional output); long time (2s on sage.math, 2014)
'The image is a... group of order 18.'
sage: EllipticCurve([1,-1,0,-107,-379]).galois_representation().image_type(7)␣
↪      # the dots (...) in the output fix #11937 (installed 'Kash' may give␣
↪additional output); long time (1s on sage.math, 2014)
'The image is a... group of order 36.'
sage: EllipticCurve([0,0,1,2580,549326]).galois_representation().image_type(7)
'The image is contained in the normalizer of a split Cartan group.'
```

Test Issue #14577:

```
sage: EllipticCurve([0, 1, 0, -4788, 109188]).galois_representation().image_
↪type(13)
'The image in PGL_2(F_13) is the exceptional group S_4.'
```

Test Issue #14752:

```
sage: EllipticCurve([0, 0, 0, -1129345880,-86028258620304]).galois_
↪representation().image_type(11)
'The image is contained in the normalizer of a non-split Cartan group.'
```

For $p = 2$:

```
sage: E = EllipticCurve('11a1')
sage: rho = E.galois_representation()
sage: rho.image_type(2)
'The image is all of GL_2(F_2), i.e. a symmetric group of order 6.'

sage: rho = EllipticCurve('14a1').galois_representation()
sage: rho.image_type(2)
'The image is cyclic of order 2 as there is exactly one rational 2-torsion␣
↪point.'

sage: rho = EllipticCurve('15a1').galois_representation()
sage: rho.image_type(2)
'The image is trivial as all 2-torsion points are rational.'

sage: rho = EllipticCurve('196a1').galois_representation()
sage: rho.image_type(2)
'The image is cyclic of order 3.'
```

$p = 3$:

```
sage: rho = EllipticCurve('33a1').galois_representation()
sage: rho.image_type(3)
'The image is all of GL_2(F_3).'

sage: rho = EllipticCurve('30a1').galois_representation()
sage: rho.image_type(3)
'The image is meta-cyclic inside a Borel subgroup as there is a 3-torsion␣
↪point on the curve.'

sage: rho = EllipticCurve('50b1').galois_representation()
sage: rho.image_type(3)
'The image is contained in a Borel subgroup as there is a 3-isogeny.'

sage: rho = EllipticCurve('3840h1').galois_representation()
sage: rho.image_type(3)
'The image is contained in a dihedral group of order 8.'

sage: rho = EllipticCurve('32a1').galois_representation()
sage: rho.image_type(3)
'The image is a semi-dihedral group of order 16, gap.SmallGroup([16,8]).'
```

ALGORITHM: Mainly based on Serre's paper.

is_crystalline(p)

Return true is the p-adic Galois representation to $GL_2(\mathbf{Z}_p)$ is crystalline.

For an elliptic curve E, this is to ask whether E has good reduction at p.

INPUT:

- `p` a prime

OUTPUT: A boolean.

EXAMPLES:

```
sage: rho = EllipticCurve('64a1').galois_representation()
sage: rho.is_crystalline(5)
True
sage: rho.is_crystalline(2)
False
```

is_irreducible(*p*)

Return True if the mod p representation is irreducible.

INPUT:

- p – a prime number

OUTPUT: A boolean.

EXAMPLES:

```
sage: rho = EllipticCurve('37b').galois_representation()
sage: rho.is_irreducible(2)
True
sage: rho.is_irreducible(3)
False
sage: rho.is_reducible(2)
False
sage: rho.is_reducible(3)
True
```

is_ordinary(*p*)

Return true if the p-adic Galois representation to $GL_2(\mathbf{Z}_p)$ is ordinary, i.e. if the image of the decomposition group in Gal($\bar{\mathbf{Q}}/\mathbf{Q}$) above he prime p maps into a Borel subgroup.

For an elliptic curve E, this is to ask whether E is ordinary at p, i.e. good ordinary or multiplicative.

INPUT:

- p a prime

OUTPUT: A boolean.

EXAMPLES:

```
sage: rho = EllipticCurve('11a3').galois_representation()
sage: rho.is_ordinary(11)
True
sage: rho.is_ordinary(5)
True
sage: rho.is_ordinary(19)
False
```

is_potentially_crystalline(*p*)

Return true is the p-adic Galois representation to $GL_2(\mathbf{Z}_p)$ is potentially crystalline, i.e. if there is a finite extension $K/\mathbf{Q}_p$ such that the p-adic representation becomes crystalline.

For an elliptic curve E, this is to ask whether E has potentially good reduction at p.

INPUT:

- p a prime

OUTPUT: A boolean.

EXAMPLES:

```
sage: rho = EllipticCurve('37b1').galois_representation()
sage: rho.is_potentially_crystalline(37)
False
sage: rho.is_potentially_crystalline(7)
True
```

is_potentially_semistable(p)

Return true if the p-adic Galois representation to $GL_2(\mathbf{Z}_p)$ is potentially semistable.

For an elliptic curve E, this returns True always

INPUT:

- p a prime

OUTPUT: A boolean.

EXAMPLES:

```
sage: rho = EllipticCurve('27a2').galois_representation()
sage: rho.is_potentially_semistable(3)
True
```

is_quasi_unipotent(p, *ell*)

Return true if the Galois representation to $GL_2(\mathbf{Z}_p)$ is quasi-unipotent at $\ell \neq p$, i.e. if there is a finite extension $K/\mathbf{Q}$ such that the inertia group at a place above ℓ in $\mathrm{Gal}(\bar{\mathbf{Q}}/K)$ maps into a Borel subgroup.

For a Galois representation attached to an elliptic curve E, this returns always True.

INPUT:

- p a prime
- ell a different prime

OUTPUT: A boolean.

EXAMPLES:

```
sage: rho = EllipticCurve('11a3').galois_representation()
sage: rho.is_quasi_unipotent(11,13)
True
```

is_reducible(p)

Return True if the mod-p representation is reducible. This is equivalent to the existence of an isogeny defined over $\mathbf{Q}$ of degree p from the elliptic curve.

INPUT:

- p – a prime number

OUTPUT: A boolean.

The answer is cached.

EXAMPLES:

```
sage: rho = EllipticCurve('121a').galois_representation()
sage: rho.is_reducible(7)
False
sage: rho.is_reducible(11)
True
sage: EllipticCurve('11a').galois_representation().is_reducible(5)
True
sage: rho = EllipticCurve('11a2').galois_representation()
sage: rho.is_reducible(5)
True
sage: EllipticCurve('11a2').torsion_order()
1
```

is_semistable(*p*)

Return true if the p-adic Galois representation to $GL_2(\mathbf{Z}_p)$ is semistable.

For an elliptic curve E, this is to ask whether E has semistable reduction at p.

INPUT:

- `p` a prime

OUTPUT: A boolean.

EXAMPLES:

```
sage: rho = EllipticCurve('20a3').galois_representation()
sage: rho.is_semistable(2)
False
sage: rho.is_semistable(3)
True
sage: rho.is_semistable(5)
True
```

is_surjective(*p*, *A=1000*)

Return True if the mod-p representation is surjective onto $Aut(E[p]) = GL_2(\mathbf{F}_p)$.

False if it is not, or None if we were unable to determine whether it is or not.

INPUT:

- `p` (integer) – a prime number
- `A` (integer) – a bound on the number of a_p to use

OUTPUT:

- (boolean) – True if the mod-p representation is surjective and False if not.

The answer is cached.

EXAMPLES:

```
sage: rho = EllipticCurve('37b').galois_representation()
sage: rho.is_surjective(2)
True
sage: rho.is_surjective(3)
False
```

```
sage: rho = EllipticCurve('121a1').galois_representation()
sage: rho.non_surjective()
[11]
sage: rho.is_surjective(5)
True
sage: rho.is_surjective(11)
False

sage: rho = EllipticCurve('121d1').galois_representation()
sage: rho.is_surjective(5)
False
sage: rho.is_surjective(11)
True
```

Here is a case, in which the algorithm does not return an answer:

```
sage: rho = EllipticCurve([0,0,1,2580,549326]).galois_representation()
sage: rho.is_surjective(7)
```

In these cases, one can use image_type to get more information about the image:

```
sage: rho.image_type(7)
'The image is contained in the normalizer of a split Cartan group.'
```

REMARKS:

1. If $p \geq 5$ then the mod-p representation is surjective if and only if the p-adic representation is surjective. When $p = 2, 3$ there are counterexamples. See papers of Dokchitsers and Elkies for more details.
2. For the primes $p = 2$ and 3, this will always answer either True or False. For larger primes it might give None.

is_unipotent (*p*, *ell*)

Return true if the Galois representation to $GL_2(\mathbf{Z}_p)$ is unipotent at $\ell \neq p$, i.e. if the inertia group at a place above ℓ in $\mathrm{Gal}(\bar{\mathbf{Q}}/\mathbf{Q})$ maps into a Borel subgroup.

For a Galois representation attached to an elliptic curve E, this returns True if E has semi-stable reduction at ℓ.

INPUT:

- `p` a prime
- `ell` a different prime

OUTPUT: A boolean.

EXAMPLES:

```
sage: rho = EllipticCurve('120a1').galois_representation()
sage: rho.is_unipotent(2,5)
True
sage: rho.is_unipotent(5,2)
False
sage: rho.is_unipotent(5,7)
True
sage: rho.is_unipotent(5,3)
True
sage: rho.is_unipotent(5,5)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: unipotent is not defined for l = p, use semistable instead.
```

is_unramified(*p*, *ell*)

Return true if the Galois representation to $GL_2(\mathbf{Z}_p)$ is unramified at ℓ, i.e. if the inertia group at a place above ℓ in $\mathrm{Gal}(\bar{\mathbf{Q}}/\mathbf{Q})$ has trivial image in $GL_2(\mathbf{Z}_p)$.

For a Galois representation attached to an elliptic curve E, this returns True if $\ell \neq p$ and E has good reduction at ℓ.

INPUT:

- `p` a prime
- `ell` another prime

OUTPUT: A boolean.

EXAMPLES:

```
sage: rho = EllipticCurve('20a3').galois_representation()
sage: rho.is_unramified(5,7)
True
sage: rho.is_unramified(5,5)
False
sage: rho.is_unramified(7,5)
False
```

This says that the 5-adic representation is unramified at 7, but the 7-adic representation is ramified at 5.

non_surjective(*A=1000*)

Return a list of primes p such that the mod-p representation *might* not be surjective. If p is not in the returned list, then the mod-p representation is provably surjective.

By a theorem of Serre, there are only finitely many primes in this list, except when the curve has complex multiplication.

If the curve has CM, we simply return the sequence [0] and do no further computation.

INPUT:

- `A` – an integer (default 1000). By increasing this parameter the resulting set might get smaller.

OUTPUT:

- `list` – if the curve has CM, returns [0]. Otherwise, returns a list of primes where mod-p representation is very likely not surjective. At any prime not in this list, the representation is definitely surjective.

EXAMPLES:

```
sage: E = EllipticCurve([0, 0, 1, -38, 90])  # 361A
sage: E.galois_representation().non_surjective()   # CM curve
[0]
```

```
sage: E = EllipticCurve([0, -1, 1, 0, 0]) # X_1(11)
sage: E.galois_representation().non_surjective()
[5]

sage: E = EllipticCurve([0, 0, 1, -1, 0]) # 37A
sage: E.galois_representation().non_surjective()
```

(continues on next page)

(continued from previous page)

```
[]

sage: E = EllipticCurve([0,-1,1,-2,-1])    # 141C
sage: E.galois_representation().non_surjective()
[13]
```

```
sage: E = EllipticCurve([1,-1,1,-9965,385220]) # 9999a1
sage: rho = E.galois_representation()
sage: rho.non_surjective()
[2]

sage: E = EllipticCurve('324b1')
sage: rho = E.galois_representation()
sage: rho.non_surjective()
[3, 5]
```

ALGORITHM: We first find an upper bound B on the possible primes. If E is semi-stable, we can take $B = 11$ by a result of Mazur. There is a bound by Serre in the case that the j-invariant is not integral in terms of the smallest prime of good reduction. Finally there is an unconditional bound by Cojocaru, but which depends on the conductor of E. For the prime below that bound we call `is_surjective`.

reducible_primes()

Return a list of the primes p such that the mod-p representation is reducible. For all other primes the representation is irreducible.

EXAMPLES:

```
sage: rho = EllipticCurve('225a').galois_representation()
sage: rho.reducible_primes()
[3]
```

18.8 Galois representations for elliptic curves over number fields

This file contains the code to compute for which primes the Galois representation attached to an elliptic curve (over an arbitrary number field) is surjective. The functions in this file are called by the `is_surjective` and `non_surjective` methods of an elliptic curve over a number field.

EXAMPLES:

```
sage: x = polygen(ZZ, 'x')
sage: K = NumberField(x**2 - 29, 'a'); a = K.gen()
sage: E = EllipticCurve([1, 0, ((5 + a)/2)**2, 0, 0])
sage: rho = E.galois_representation()
sage: rho.is_surjective(29) # Cyclotomic character not surjective.
False
sage: rho.is_surjective(31) # See Section 5.10 of [Ser1972].
True
sage: rho.non_surjective()  # long time (4s on sage.math, 2014)
[3, 5, 29]

sage: E = EllipticCurve_from_j(1728).change_ring(K) # CM
sage: E.galois_representation().non_surjective()  # long time (2s on sage.math, 2014)
[0]
```

AUTHORS:

- Eric Larson (2012-05-28): initial version.
- Eric Larson (2014-08-13): added isogeny_bound function.
- John Cremona (2016, 2017): various efficiency improvements to _semistable_reducible_primes
- John Cremona (2017): implementation of Billerey's algorithm to find all reducible primes

REFERENCES:

- [Ser1972]
- [Sut2012]

sage.schemes.elliptic_curves.gal_reps_number_field.**Billerey_B_bound**(*E*, *max_l=200*, *num_l=8*, *small_prime_bound=0*, *debug=False*)

Compute Billerey's bound B.

We compute B_l for l up to `max_l` (at most) until `num_l` nonzero values are found (at most). Return the list of primes dividing all B_l computed, excluding those dividing 6 or ramified or of bad reduction or less than small_prime_bound. If no non-zero values are found return [0].

INPUT:

- `E` – an elliptic curve over a number field K, given by a global integral model.
- `max_l` (int, default 200) – maximum size of primes l to check.
- `num_l` (int, default 8) – maximum number of primes l to check.
- `small_prime_bound` (int, default 0) – remove primes less than this from the output.
- `debug` (bool, default `False`) – if `True` prints details.

Note: The purpose of the small_prime_bound is that it is faster to deal with these using the local test; by ignoring them here, we enable the algorithm to terminate sooner when there are no large reducible primes, which is always the case in practice.

EXAMPLES:

```
sage: x = polygen(ZZ, 'x')
sage: K = NumberField(x**2 - 29, 'a'); a = K.gen()
sage: E = EllipticCurve([1, 0, ((5 + a)/2)**2, 0, 0])
sage: from sage.schemes.elliptic_curves.gal_reps_number_field import Billerey_B_
↪bound
sage: Billerey_B_bound(E)
[5]
```

If we do not use enough primes l, extraneous primes will be included which are not reducible primes:

```
sage: Billerey_B_bound(E, num_l=6)
[5, 7]
```

Similarly if we do not use large enough primes l:

```
sage: Billerey_B_bound(E, max_l=50, num_l=8)
[5, 7]
sage: Billerey_B_bound(E, max_l=100, num_l=8)
[5]
```

This curve does have a rational 5-isogeny:

```
sage: len(E.isogenies_prime_degree(5))
1
```

sage.schemes.elliptic_curves.gal_reps_number_field.**Billerey_B_l**(*E*, *l*, *B=0*)

Return Billerey's B_l, adapted from the definition in [Bil2011], after (9).

INPUT:

- `E` – an elliptic curve over a number field K, given by a global integral model.
- `l` (int) – a rational prime
- `B` (int) – 0 or LCM of previous B_l: the prime-to-B part of this B_l is ignored.

EXAMPLES:

```
sage: x = polygen(ZZ, 'x')
sage: K = NumberField(x**2 - 29, 'a'); a = K.gen()
sage: E = EllipticCurve([1, 0, ((5 + a)/2)**2, 0, 0])
sage: from sage.schemes.elliptic_curves.gal_reps_number_field import Billerey_B_l
sage: [Billerey_B_l(E,l) for l in primes(15)]
[1123077552537600,
 227279663773903886745600,
 0,
 0,
 269247154818492941287713746693964214802283882086400,
 0]
```

sage.schemes.elliptic_curves.gal_reps_number_field.**Billerey_P_l**(*E*, *l*)

Return Billerey's P_l^* as defined in [Bil2011], equation (9).

INPUT:

- `E` – an elliptic curve over a number field K, given by a global integral model.
- `l` – a rational prime

EXAMPLES:

```
sage: x = polygen(ZZ, 'x')
sage: K = NumberField(x**2 - 29, 'a'); a = K.gen()
sage: E = EllipticCurve([1, 0, ((5 + a)/2)**2, 0, 0])
sage: from sage.schemes.elliptic_curves.gal_reps_number_field import Billerey_P_l
sage: [Billerey_P_l(E,l) for l in primes(10)]
[x^2 + 8143*x + 16777216,
 x^2 + 451358*x + 282429536481,
 x^4 - 664299076*x^3 + 205155493652343750*x^2 - 39595310449600219726562500*x +␣
↪355271367880050092935562133789062 5,
 x^4 - 207302404*x^3 - 3774237985386893 66394*x^2 -␣
↪39715249826471656586987520004*x + 367033682172941254412302110320336601888 01]
```

sage.schemes.elliptic_curves.gal_reps_number_field.**Billerey_R_bound**(*E*, *max_l=200*, *num_l=8*, *small_prime_bound=None*, *debug=False*)

Compute Billerey's bound R.

We compute R_q for q dividing primes ℓ up to `max_l` (at most) until `num_l` nonzero values are found (at most). Return the list of primes dividing all `R_q` computed, excluding those dividing 6 or ramified or of bad reduction or less than small_prime_bound. If no non-zero values are found return [0].

INPUT:

- `E` – an elliptic curve over a number field K, given by a global integral model.
- `max_l` (int, default 200) – maximum size of rational primes l for which the primes q above l are checked.
- `num_l` (int, default 8) – maximum number of rational primes l for which the primes q above l are checked.
- `small_prime_bound` (int, default 0) – remove primes less than this from the output.
- `debug` (bool, default `False`) – if `True` prints details.

Note: The purpose of the `small_prime_bound` is that it is faster to deal with these using the local test; by ignoring them here, we enable the algorithm to terminate sooner when there are no large reducible primes, which is always the case in practice.

EXAMPLES:

```
sage: x = polygen(ZZ, 'x')
sage: K = NumberField(x**2 - 29, 'a'); a = K.gen()
sage: E = EllipticCurve([1, 0, ((5 + a)/2)**2, 0, 0])
sage: from sage.schemes.elliptic_curves.gal_reps_number_field import Billerey_R_
↪bound
sage: Billerey_R_bound(E)
[5]
```

We may get no bound at all if we do not use enough primes:

```
sage: Billerey_R_bound(E, max_l=2, debug=False)
[0]
```

Or we may get a bound but not a good one if we do not use enough primes:

```
sage: Billerey_R_bound(E, num_l=1, debug=False)
[5, 17, 67, 157]
```

In this case two primes is enough to restrict the set of possible reducible primes to just $\{5\}$. This curve does have a rational 5-isogeny:

```
sage: Billerey_R_bound(E, num_l=2, debug=False)
[5]
sage: len(E.isogenies_prime_degree(5))
1
```

`sage.schemes.elliptic_curves.gal_reps_number_field.`**`Billerey_R_q`**`(E, q, B=0)`

Return Billerey's R_q, adapted from the definition in [Bil2011], Theorem 2.8.

INPUT:

- `E` – an elliptic curve over a number field K, given by a global integral model.
- `q` – a prime ideal of K
- `B` (int) – 0 or LCM of previous R_q: the prime-to-B part of this R_q is ignored.

EXAMPLES:

```
sage: x = polygen(ZZ, 'x')
sage: K = NumberField(x**2 - 29, 'a'); a = K.gen()
sage: E = EllipticCurve([1, 0, ((5 + a)/2)**2, 0, 0])
sage: from sage.schemes.elliptic_curves.gal_reps_number_field import Billerey_R_q
sage: [Billerey_R_q(E,K.prime_above(l)) for l in primes(10)]
[1123077552537600,
22727966377390388674560 0,
51956919562116960000000000000000,
25248593382055636182992640000000 0]
```

sage.schemes.elliptic_curves.gal_reps_number_field.**Frobenius_filter**(*E*, *L*, *patience=100*)

Determine which primes in L might have an image contained in a Borel subgroup, by checking of traces of Frobenius.

Note: This function will sometimes return primes for which the image is not contained in a Borel subgroup. This issue cannot always be fixed by increasing patience as it may be a result of a failure of a local-global principle for isogenies.

INPUT:

- `E` – EllipticCurve over a number field.
- `L` – a list of prime numbers.
- `patience` (int), default 100 – a positive integer bounding the number of traces of Frobenius to use while trying to prove irreducibility.

OUTPUT:

- list – The list of all primes ℓ in L for which the mod ℓ image might be contained in a Borel subgroup of $GL_2(\mathbf{F}_\ell)$.

EXAMPLES:

```
sage: E = EllipticCurve('11a1') # has a 5-isogeny
sage: sage.schemes.elliptic_curves.gal_reps_number_field.Frobenius_filter(E,
↪primes(40))  # long time
[5]
```

Example to show that the output may contain primes where the representation is in fact reducible. Over $\mathbf{Q}$ the following is essentially the unique such example by [Sut2012]:

```
sage: E = EllipticCurve_from_j(2268945/128)
sage: sage.schemes.elliptic_curves.gal_reps_number_field.Frobenius_filter(E, [7,␣
↪11])  # long time
[7]
```

This curve does possess a 7-isogeny modulo every prime of good reduction, but has no rational 7-isogeny:

```
sage: E.isogenies_prime_degree(7)
[]
```

A number field example:

```
sage: K.<i> = QuadraticField(-1)
sage: E = EllipticCurve([1+i, -i, i, -399-240*i,  2627+2869*i])
```

(continues on next page)

(continued from previous page)

```
sage: sage.schemes.elliptic_curves.gal_reps_number_field.Frobenius_filter(E,␣
↪primes(20))  # long time
[2, 3]
```

Here the curve really does possess isogenies of degrees 2 and 3:

```
sage: [len(E.isogenies_prime_degree(l)) for l in [2,3]]
[1, 1]
```

class sage.schemes.elliptic_curves.gal_reps_number_field.**GaloisRepresentation**(*E*)

Bases: SageObject

The compatible family of Galois representation attached to an elliptic curve over a number field.

Given an elliptic curve E over a number field K and a rational prime number p, the p^n-torsion $E[p^n]$ points of E is a representation of the absolute Galois group G_K of K. As n varies we obtain the Tate module T_pE which is a representation of G_K on a free $\mathbf{Z}_p$-module of rank 2. As p varies the representations are compatible.

EXAMPLES:

```
sage: x = polygen(ZZ, 'x')
sage: K = NumberField(x**2 + 1, 'a')
sage: E = EllipticCurve('11a1').change_ring(K)
sage: rho = E.galois_representation()
sage: rho
Compatible family of Galois representations associated to the Elliptic Curve
 defined by y^2 + y = x^3 + (-1)*x^2 + (-10)*x + (-20)
 over Number Field in a with defining polynomial x^2 + 1
```

elliptic_curve()

Return the elliptic curve associated to this representation.

EXAMPLES:

```
sage: x = polygen(ZZ, 'x')
sage: K = NumberField(x**2 + 1, 'a'); a = K.gen()
sage: E = EllipticCurve_from_j(a)
sage: rho = E.galois_representation()
sage: rho.elliptic_curve() == E
True
```

is_surjective(*p*, *A=100*)

Return `True` if the mod-p representation is (provably) surjective onto $Aut(E[p]) = GL_2(\mathbf{F}_p)$. Return `False` if it is (probably) not.

INPUT:

- `p` – a prime number.
- `A` – (integer) a bound on the number of traces of Frobenius to use while trying to prove surjectivity.

EXAMPLES:

```
sage: x = polygen(ZZ, 'x')
sage: K = NumberField(x**2 - 29, 'a'); a = K.gen()
sage: E = EllipticCurve([1, 0, ((5 + a)/2)**2, 0, 0])
sage: rho = E.galois_representation()
sage: rho.is_surjective(29) # Cyclotomic character not surjective.
```

(continues on next page)

(continued from previous page)

```
False
sage: rho.is_surjective(7) # See Section 5.10 of [Ser1972].
True
```

If E is defined over $\mathbf{Q}$, then the exceptional primes for $E_{/K}$ are the same as the exceptional primes for E, except for those primes that are ramified in $K/\mathbf{Q}$ or are less than $[K : \mathbf{Q}]$:

```
sage: K = NumberField(x**2 + 11, 'a')
sage: E = EllipticCurve([2, 14])
sage: rhoQQ = E.galois_representation()
sage: rhoK = E.change_ring(K).galois_representation()
sage: rhoQQ.is_surjective(2) == rhoK.is_surjective(2)
False
sage: rhoQQ.is_surjective(3) == rhoK.is_surjective(3)
True
sage: rhoQQ.is_surjective(5) == rhoK.is_surjective(5)
True
```

For CM curves, the mod-p representation is never surjective:

```
sage: K.<a> = NumberField(x^2 - x + 1)
sage: E = EllipticCurve([0,0,0,0,a])
sage: E.has_cm()
True
sage: rho = E.galois_representation()
sage: any(rho.is_surjective(p) for p in [2,3,5,7])
False
```

isogeny_bound(*A=100*)

Return a list of primes p including all primes for which the image of the mod-p representation is contained in a Borel.

Note: For the actual list of primes p at which the representation is reducible see *reducible_primes()*.

INPUT:

- `A` – int (a bound on the number of traces of Frobenius to use while trying to prove the mod-p representation is not contained in a Borel).

OUTPUT:

- `list` – A list of primes which contains (but may not be equal to) all p for which the image of the mod-p representation is contained in a Borel subgroup. At any prime not in this list, the image is definitely not contained in a Borel. If E has CM defined over K, the list [0] is returned.

EXAMPLES:

```
sage: x = polygen(ZZ, 'x')
sage: K = NumberField(x**2 - 29, 'a'); a = K.gen()
sage: E = EllipticCurve([1, 0, ((5 + a)/2)**2, 0, 0])
sage: rho = E.galois_representation()
sage: rho.isogeny_bound() # See Section 5.10 of [Ser1972].  # long time
[3, 5]
sage: K = NumberField(x**2 + 1, 'a')
sage: E = EllipticCurve_from_j(K(1728))              # CM over K
sage: E.galois_representation().isogeny_bound()
```

(continues on next page)

(continued from previous page)

```
[0]
sage: E = EllipticCurve_from_j(K(0))                    # CM NOT over K
sage: E.galois_representation().isogeny_bound()         # long time
[2, 3]
sage: E = EllipticCurve_from_j(K(2268945/128))          # c.f. [Sut2012]
sage: rho = E.galois_representation()
sage: rho.isogeny_bound()  # No 7-isogeny, but...      # long time
[7]
```

For curves with rational CM, there are infinitely many primes p for which the mod-p representation is reducible, and [0] is returned:

```
sage: K.<a> = NumberField(x^2 - x + 1)
sage: E = EllipticCurve([0,0,0,0,a])
sage: E.has_rational_cm()
True
sage: rho = E.galois_representation()
sage: rho.isogeny_bound()
[0]
```

An example (an elliptic curve with everywhere good reduction over an imaginary quadratic field with quite large discriminant), which failed until fixed at Issue #21776:

```
sage: K.<a> = NumberField(x^2 - x + 112941801)
sage: E = EllipticCurve([a+1,a-1,a,-23163076*a + 266044005933275,
↪57560769602038*a - 836483958630700313803])
sage: E.conductor().norm()
1
sage: GR = E.galois_representation()
sage: GR.isogeny_bound()
[]
```

non_surjective(*A=100*)

Return a list of primes p including all primes for which the mod-p representation might not be surjective.

INPUT:

- `A` – int (a bound on the number of traces of Frobenius to use while trying to prove surjectivity).

OUTPUT:

- `list` – A list of primes where mod-p representation is very likely not surjective. At any prime not in this list, the representation is definitely surjective. If E has CM, the list [0] is returned.

EXAMPLES:

```
sage: x = polygen(ZZ, 'x')
sage: K = NumberField(x**2 - 29, 'a'); a = K.gen()
sage: E = EllipticCurve([1, 0, ((5 + a)/2)**2, 0, 0])
sage: rho = E.galois_representation()
sage: rho.non_surjective() # See Section 5.10 of [Ser1972].  # long time
[3, 5, 29]
sage: K = NumberField(x**2 + 3, 'a'); a = K.gen()
sage: E = EllipticCurve([0, -1, 1, -10, -20]).change_ring(K) # X_0(11)
sage: rho = E.galois_representation()
sage: rho.non_surjective()  # long time (4s on sage.math, 2014)
[3, 5]
sage: K = NumberField(x**2 + 1, 'a'); a = K.gen()
```

(continues on next page)

(continued from previous page)

```
sage: E = EllipticCurve_from_j(1728).change_ring(K) # CM
sage: rho = E.galois_representation()
sage: rho.non_surjective()
[0]
sage: K = NumberField(x**2 - 5, 'a'); a = K.gen()
sage: E = EllipticCurve_from_j(146329141248*a - 327201914880) # CM
sage: rho = E.galois_representation()
sage: rho.non_surjective() # long time (3s on sage.math, 2014)
[0]
```

reducible_primes()

Return a list of primes p for which the mod-p representation is reducible, or [0] for CM curves.

OUTPUT:

- `list` – A list of those primes p for which the mod-p representation is contained in a Borel subgroup, i.e. is reducible. If E has CM *defined over K*, the list [0] is returned (in this case the representation is reducible for infinitely many primes).

EXAMPLES:

```
sage: x = polygen(ZZ, 'x')
sage: K = NumberField(x**2 - 29, 'a'); a = K.gen()
sage: E = EllipticCurve([1, 0, ((5 + a)/2)**2, 0, 0])
sage: rho = E.galois_representation()
sage: rho.reducible_primes() # See Section 5.10 of [Ser1972].  # long time
[3, 5]

sage: K = NumberField(x**2 + 1, 'a')
sage: E = EllipticCurve_from_j(K(1728))              # CM over K
sage: E.galois_representation().reducible_primes()
[0]
sage: E = EllipticCurve_from_j(K(0))                 # CM but NOT over K
sage: E.galois_representation().reducible_primes()   # long time
[2, 3]
sage: E = EllipticCurve_from_j(K(2268945/128))       # c.f. [Sut2012]
sage: rho = E.galois_representation()
sage: rho.isogeny_bound()  # No 7-isogeny, but...    # long time
[7]
sage: rho.reducible_primes()                         # long time
[]
```

For curves with rational CM, there are infinitely many primes p for which the mod-p representation is reducible, and [0] is returned:

```
sage: K.<a> = NumberField(x^2 - x + 1)
sage: E = EllipticCurve([0,0,0,0,a])
sage: E.has_rational_cm()
True
sage: rho = E.galois_representation()
sage: rho.reducible_primes()
[0]
```

sage.schemes.elliptic_curves.gal_reps_number_field.**deg_one_primes_iter**(*K*, *principal_only=False*)

Return an iterator over degree 1 primes of `K`.

INPUT:

- K – a number field
- principal_only – bool; if True, only yield principal primes

OUTPUT:

An iterator over degree 1 primes of K up to the given norm, optionally yielding only principal primes.

EXAMPLES:

```
sage: K.<a> = QuadraticField(-5)
sage: from sage.schemes.elliptic_curves.gal_reps_number_field import deg_one_
↪primes_iter
sage: it = deg_one_primes_iter(K)
sage: [next(it) for _ in range(6)]
[Fractional ideal (2, a + 1),
 Fractional ideal (3, a + 1),
 Fractional ideal (3, a + 2),
 Fractional ideal (a),
 Fractional ideal (7, a + 3),
 Fractional ideal (7, a + 4)]
sage: it = deg_one_primes_iter(K, True)
sage: [next(it) for _ in range(6)]
[Fractional ideal (a),
 Fractional ideal (-2*a + 3),
 Fractional ideal (2*a + 3),
 Fractional ideal (a + 6),
 Fractional ideal (a - 6),
 Fractional ideal (-3*a + 4)]
```

sage.schemes.elliptic_curves.gal_reps_number_field.**reducible_primes_Billerey**(*E*, *num_l=None*, *max_l=None*, *verbose=False*)

Return a finite set of primes ℓ containing all those for which E has a K-rational ell-isogeny, where K is the base field of E: i.e., the mod-ℓ representation is irreducible for all ℓ outside the set returned.

INPUT:

- E – an elliptic curve defined over a number field K.
- max_l (int or None (default)) – the maximum prime ℓ to use for the B-bound and R-bound. If None, a default value will be used.
- num_l (int or None (default)) – the number of primes ℓ to use for the B-bound and R-bound. If None, a default value will be used.

Note: If E has CM then [0] is returned. In this case use the function sage.schemes.elliptic_curves.isogeny_class.possible_isogeny_degrees

---

We first compute Billeray's B_bound using at most num_l primes of size up to max_l. If that fails we compute Billeray's R_bound using at most num_q primes of size up to max_q.

Provided that one of these methods succeeds in producing a finite list of primes we check these using a local condition, and finally test that the primes returned actually are reducible. Otherwise we return [0].

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.gal_reps_number_field import reducible_
↪primes_Billerey
sage: x = polygen(ZZ, 'x')
sage: K = NumberField(x**2 - 29, 'a'); a = K.gen()
sage: E = EllipticCurve([1, 0, ((5 + a)/2)**2, 0, 0])
sage: reducible_primes_Billerey(E)  # long time
[3, 5]
sage: K = NumberField(x**2 + 1, 'a')
sage: E = EllipticCurve_from_j(K(1728)) # CM over K
sage: reducible_primes_Billerey(E)  # long time
[0]
sage: E = EllipticCurve_from_j(K(0)) # CM but NOT over K
sage: reducible_primes_Billerey(E)  # long time
[2, 3]
```

An example where a prime is not reducible but passes the test:

```
sage: E = EllipticCurve_from_j(K(2268945/128)).global_minimal_model() # c.f.␣
↪[Sut2012]
sage: reducible_primes_Billerey(E)  # long time
[7]
```

sage.schemes.elliptic_curves.gal_reps_number_field.**reducible_primes_naive**(*E*, *max_l=None*, *num_P=None*, *verbose=False*)

Return locally reducible primes ℓ up to `max_l`.

The list of primes ℓ returned consists of all those up to `max_l` such that E mod P has an ℓ-isogeny, where K is the base field of E, for `num_P` primes P of K. In most cases E then has a K-rational ℓ-isogeny, but there are rare exceptions.

INPUT:

- `E` – an elliptic curve defined over a number field K
- `max_l` (int or `None` (default)) – the maximum prime ℓ to test.
- `num_P` (int or `None` (default)) – the number of primes P of K to use in testing each ℓ.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.gal_reps_number_field import reducible_
↪primes_naive
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^4 - 5*x^2 + 3)
sage: E = EllipticCurve(K, [a^2 - 2, -a^2 + 3, a^2 - 2, -50*a^2 + 35, 95*a^2 -␣
↪67])
sage: reducible_primes_naive(E,num_P=10)
[2, 5, 53, 173, 197, 241, 293, 317, 409, 557, 601, 653, 677, 769, 773, 797]
sage: reducible_primes_naive(E,num_P=15)
[2, 5, 197, 557, 653, 769]
sage: reducible_primes_naive(E,num_P=20)
[2, 5]
sage: reducible_primes_naive(E)  # long time
[2, 5]
sage: [phi.degree() for phi in E.isogenies_prime_degree()]  # long time
[2, 2, 2, 5]
```

18.9 Isogeny class of elliptic curves over number fields

AUTHORS:

- David Roe (2012-03-29) – initial version.
- John Cremona (2014-08) – extend to number fields.

class sage.schemes.elliptic_curves.isogeny_class.**IsogenyClass_EC**(*E*, *label=None*, *empty=False*)

Bases: SageObject

Isogeny class of an elliptic curve.

Note: The current implementation chooses a curve from each isomorphism class in the isogeny class. Over **Q** this is a unique reduced minimal model in each isomorphism class. Over number fields the model chosen may change in future.

---

graph()

Return a graph whose vertices correspond to curves in this class, and whose edges correspond to prime degree isogenies.

Note: There are only finitely many possible isogeny graphs for curves over **Q** [Maz1978b]. This function tries to lay out the graph nicely by special casing each isogeny graph. This could also be done over other number fields, such as quadratic fields.

Note: The vertices are labeled 1 to n rather than 0 to n-1 to match LMFDB and Cremona labels for curves over **Q**.

EXAMPLES:

```
sage: isocls = EllipticCurve('15a3').isogeny_class()
sage: G = isocls.graph()
sage: sorted(G._pos.items())
[(1, [-0.8660254, 0.5]), (2, [-0.8660254, 1.5]), (3, [-1.7320508, 0]),
 (4, [0, 0]), (5, [0, -1]), (6, [0.8660254, 0.5]),
 (7, [0.8660254, 1.5]), (8, [1.7320508, 0])]
```

index(*C*)

Return the index of a curve in this class.

INPUT:

- C – an elliptic curve in this isogeny class.

OUTPUT:

- i – an integer so that the i th curve in the class is isomorphic to C

EXAMPLES:

```
sage: E = EllipticCurve('990j1')
sage: iso = E.isogeny_class(order="lmfdb") # orders lexicographically on a-
↪invariants
```

(continues on next page)

(continued from previous page)

```
sage: iso.index(E.short_weierstrass_model())
2
```

isogenies (*fill=False*)

Return a list of lists of isogenies and 0s, corresponding to the entries of *matrix()*

INPUT:

- `fill` – boolean (default `False`). Whether to only return prime degree isogenies. Currently only implemented for `fill=False`.

OUTPUT:

- a list of lists, where the `j` th entry of the `i` th list is either zero or a prime degree isogeny from the `i` th curve in this class to the `j` th curve.

Warning: The domains and codomains of the isogenies will have the same Weierstrass equation as the curves in this class, but they may not be identical python objects in the current implementation.

EXAMPLES:

```
sage: isocls = EllipticCurve('15a3').isogeny_class()
sage: f = isocls.isogenies()[0][1]; f
Isogeny of degree 2
  from Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 5*x + 2 over␣
↪Rational Field
    to Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 80*x + 242 over␣
↪Rational Field
sage: f.domain() == isocls.curves[0] and f.codomain() == isocls.curves[1]
True
```

matrix (*fill=True*)

Return the matrix whose entries give the minimal degrees of isogenies between curves in this class.

INPUT:

- `fill` – boolean (default `True`). If `False` then the matrix will contain only zeros and prime entries; if `True` it will fill in the other degrees.

EXAMPLES:

```
sage: isocls = EllipticCurve('15a3').isogeny_class()
sage: isocls.matrix()
[ 1  2  2  4  4  8  8]
[ 2  1  4  4  8  8 16 16]
[ 2  4  1  4  8  8 16 16]
[ 2  4  4  1  2  2  4  4]
[ 4  8  8  2  1  4  8  8]
[ 4  8  8  2  4  1  2  2]
[ 8 16 16  4  8  2  1  4]
[ 8 16 16  4  8  2  4  1]
sage: isocls.matrix(fill=False)
[0 2 2 2 0 0 0 0]
[2 0 0 0 0 0 0 0]
[2 0 0 0 0 0 0 0]
[2 0 0 0 2 2 0 0]
```

(continues on next page)

(continued from previous page)

```
[0 0 0 2 0 0 0 0]
[0 0 0 2 0 0 2 2]
[0 0 0 0 0 2 0 0]
[0 0 0 0 0 2 0 0]
```

qf_matrix()

Return the array whose entries are quadratic forms representing the degrees of isogenies between curves in this class (CM case only).

OUTPUT:

a $2x2$ array (list of lists) of list, each of the form [2] or [2,1,3] representing the coefficients of an integral quadratic form in 1 or 2 variables whose values are the possible isogeny degrees between the i'th and j'th curve in the class.

EXAMPLES:

```
sage: pol = PolynomialRing(QQ,'x')([1,0,3,0,1])
sage: K.<c> = NumberField(pol)
sage: j = 1480640 + 565760*c^2
sage: E = EllipticCurve(j=j)
sage: C = E.isogeny_class()
sage: C.qf_matrix()
[[[1], [2, 2, 3]], [[2, 2, 3], [1]]]
```

reorder(*order*)

Return a new isogeny class with the curves reordered.

INPUT:

- `order` – None, a string or an iterable over all curves in this class. See *sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field.isogeny_class()* for more details.

OUTPUT:

- Another *IsogenyClass_EC* with the curves reordered (and matrices and maps changed as appropriate)

EXAMPLES:

```
sage: isocls = EllipticCurve('15a1').isogeny_class()
sage: print("\n".join(repr(C) for C in isocls.curves))
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 10*x - 10 over Rational␣
↪Field
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 5*x + 2 over Rational␣
↪Field
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 + 35*x - 28 over Rational␣
↪Field
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 135*x - 660 over␣
↪Rational Field
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 80*x + 242 over␣
↪Rational Field
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 over Rational Field
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 110*x - 880 over␣
↪Rational Field
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 2160*x - 39540 over␣
↪Rational Field
```

(continues on next page)

(continued from previous page)

```
sage: isocls2 = isocls.reorder('lmfdb')
sage: print("\n".join(repr(C) for C in isocls2.curves))
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 2160*x - 39540 over␣
↪Rational Field
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 135*x - 660 over␣
↪Rational Field
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 110*x - 880 over␣
↪Rational Field
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 80*x + 242 over␣
↪Rational Field
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 10*x - 10 over Rational␣
↪Field
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 5*x + 2 over Rational␣
↪Field
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 over Rational Field
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 + 35*x - 28 over Rational␣
↪Field
```

class sage.schemes.elliptic_curves.isogeny_class.**IsogenyClass_EC_NumberField**(*E*, *reducible_primes=None*, *algorithm='Billerey'*, *minimal_models=True*)

Bases: *IsogenyClass_EC*

Isogeny classes for elliptic curves over number fields.

copy()

Return a copy (mostly used in reordering).

EXAMPLES:

```
sage: K.<i> = QuadraticField(-1)
sage: E = EllipticCurve(K, [0,0,0,0,1])
sage: C = E.isogeny_class()
sage: C2 = C.copy()
sage: C is C2
False
sage: C == C2
True
```

class sage.schemes.elliptic_curves.isogeny_class.**IsogenyClass_EC_Rational**(*E*, *algorithm='sage'*, *label=None*, *empty=False*)

Bases: *IsogenyClass_EC_NumberField*

Isogeny classes for elliptic curves over $\mathbf{Q}$.

copy()

Return a copy (mostly used in reordering).

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: C = E.isogeny_class()
sage: C2 = C.copy()
sage: C is C2
False
sage: C == C2
True
```

sage.schemes.elliptic_curves.isogeny_class.**isogeny_degrees_cm**(*E*, *verbose=False*)

Return a list of primes ℓ sufficient to generate the isogeny class of E, where E has CM.

INPUT:

- `E` – An elliptic curve defined over a number field.

OUTPUT:

A finite list of primes ℓ such that every curve isogenous to this curve can be obtained by a finite sequence of isogenies of degree one of the primes in the list. This list is not necessarily minimal.

ALGORITHM:

For curves with CM by the order O of discriminant d, the Galois representation is always non-surjective and the curve will admit ℓ-isogenies for infinitely many primes ℓ, but there are only finitely many codomains E'. The primes can be divided according to the discriminant d' of the CM order O' associated to E: either $O = O'$, or one contains the other with index ℓ, since $\ell O \subset O'$ and vice versa.

Case (1): $O = O'$. The degrees of all isogenies between E and E' are precisely the integers represented by one of the classes of binary quadratic forms Q of discriminant d. Hence to obtain all possible isomorphism classes of codomain E', we need only use one prime ℓ represented by each such class Q. It would in fact suffice to use primes represented by forms which generate the class group. Here we simply omit the principal class and one from each pair of inverse classes, and include a prime represented by each of the remaining forms.

Case (2): $[O' : O] = \ell$: so $d = \ell^2 d$;. We include all prime divisors of d.

Case (3): $[O : O'] = \ell$: we may assume that ℓ does not divide d as we have already included these, so ℓ either splits or is inert in O; the class numbers satisfy $h(O') = (\ell \pm 1)h(O)$ accordingly. We include all primes ℓ such that $\ell \pm 1$ divides the degree $[K : \mathbf{Q}]$.

For curves with only potential CM we proceed as in the CM case, using $2[K : \mathbf{Q}]$ instead of $[K : \mathbf{Q}]$.

EXAMPLES:

For curves with CM by a quadratic order of class number greater than 1, we use the structure of the class group to only give one prime in each ideal class:

```
sage: pol = PolynomialRing(QQ,'x')([1,-3,5,-5,5,-3,1])
sage: L.<a> = NumberField(pol)
sage: j = hilbert_class_polynomial(-23).roots(L, multiplicities=False)[0]
sage: E = EllipticCurve(j=j)
sage: from sage.schemes.elliptic_curves.isogeny_class import isogeny_degrees_cm
sage: isogeny_degrees_cm(E, verbose=True)
CM case, discriminant = -23
initial primes: {2}
upward primes: {}
downward ramified primes: {}
```

(continues on next page)

(continued from previous page)

```
downward split primes: {2, 3}
downward inert primes: {5}
primes generating the class group: [2]
Set of primes before filtering: {2, 3, 5}
List of primes after filtering: [2, 3]
[2, 3]
```

sage.schemes.elliptic_curves.isogeny_class.**possible_isogeny_degrees**(*E*, *algorithm='Billerey'*, *max_l=None*, *num_l=None*, *exact=True*, *verbose=False*)

Return a list of primes ℓ sufficient to generate the isogeny class of E.

INPUT:

- `E` – An elliptic curve defined over a number field.
- `algorithm` (string, default `'Billerey'`) – Algorithm to be used for non-CM curves: either `'Billerey'`, `'Larson'`, or `'heuristic'`. Only relevant for non-CM curves and base fields other than **Q**.
- `max_l` (int or `None`) – only relevant for non-CM curves and algorithms `'Billerey'` and `'heuristic'`. Controls the maximum prime used in either algorithm. If `None`, use the default for that algorithm.
- `num_l` (int or `None`) – only relevant for non-CM curves and algorithm `'Billerey'`. Controls the maximum number of primes used in the algorithm. If `None`, use the default for that algorithm.
- `exact` (bool, default `True`) – if `True`, perform an additional check that the primes returned are all reducible. If `False`, skip this step, in which case some of the primes returned may be irreducible.

OUTPUT:

A finite list of primes ℓ such that every curve isogenous to this curve can be obtained by a finite sequence of isogenies of degree one of the primes in the list.

ALGORITHM:

For curves without CM, the set may be taken to be the finite set of primes at which the Galois representation is not surjective, since the existence of an ℓ-isogeny is equivalent to the image of the mod-ℓ Galois representation being contained in a Borel subgroup. Two rigorous algorithms have been implemented to determine this set, due to Larson and Billeray respectively. We also provide a non-rigorous 'heuristic' algorithm which only tests reducible primes up to a bound depending on the degree of the base field.

For curves with CM see the documentation for *isogeny_degrees_cm()*.

EXAMPLES:

For curves without CM we determine the primes at which the mod p Galois representation is reducible, i.e. contained in a Borel subgroup:

```
sage: from sage.schemes.elliptic_curves.isogeny_class import possible_isogeny_
↪degrees
sage: E = EllipticCurve('11a1')
sage: possible_isogeny_degrees(E)
[5]
```

(continues on next page)

(continued from previous page)

```
sage: possible_isogeny_degrees(E, algorithm='Larson')
[5]
sage: possible_isogeny_degrees(E, algorithm='Billerey')
[5]
sage: possible_isogeny_degrees(E, algorithm='heuristic')
[5]
```

We check that in this case E really does have rational 5-isogenies:

```
sage: [phi.degree() for phi in E.isogenies_prime_degree()]
[5, 5]
```

Over an extension field:

```
sage: E3 = E.change_ring(CyclotomicField(3))
sage: possible_isogeny_degrees(E3)
[5]
sage: [phi.degree() for phi in E3.isogenies_prime_degree()]
[5, 5]
```

A higher degree example (LMFDB curve 5.5.170701.1-4.1-b1):

```
sage: K.<a> = NumberField(x^5 - x^4 - 6*x^3 + 4*x + 1)
sage: E = EllipticCurve(K, [a^3 - a^2 - 5*a + 1, a^4 - a^3 - 5*a^2 - a + 1,
....:                       -a^4 + 2*a^3 + 5*a^2 - 5*a - 3, a^4 - a^3 - 5*a^2 - a,
....:                       -3*a^4 + 4*a^3 + 17*a^2 - 6*a - 12])
sage: possible_isogeny_degrees(E, algorithm='heuristic')
[2]
sage: possible_isogeny_degrees(E, algorithm='Billerey')
[2]
sage: possible_isogeny_degrees(E, algorithm='Larson')
[2]
```

LMFDB curve 4.4.8112.1-108.1-a5:

```
sage: K.<a> = NumberField(x^4 - 5*x^2 + 3)
sage: E = EllipticCurve(K, [a^2 - 2, -a^2 + 3, a^2 - 2, -50*a^2 + 35, 95*a^2 -␣
↪67])
sage: possible_isogeny_degrees(E, exact=False, algorithm='Billerey')
[2, 5]
sage: possible_isogeny_degrees(E, exact=False, algorithm='Larson')
[2, 5]
sage: possible_isogeny_degrees(E, exact=False, algorithm='heuristic')
[2, 5]
sage: possible_isogeny_degrees(E)
[2, 5]
```

This function only returns the primes which are isogeny degrees:

```
sage: Set(E.isogeny_class().matrix().list())
{1, 2, 4, 5, 20, 10}
```

For curves with CM by a quadratic order of class number greater than 1, we use the structure of the class group to only give one prime in each ideal class:

```
sage: pol = PolynomialRing(QQ,'x')([1,-3,5,-5,5,-3,1])
sage: L.<a> = NumberField(pol)
sage: j = hilbert_class_polynomial(-23).roots(L, multiplicities=False)[0]
sage: E = EllipticCurve(j=j)
sage: from sage.schemes.elliptic_curves.isogeny_class import possible_isogeny_
↪degrees
sage: possible_isogeny_degrees(E, verbose=True)
CM case, discriminant = -23
initial primes: {2}
upward primes: {}
downward ramified primes: {}
downward split primes: {2, 3}
downward inert primes: {5}
primes generating the class group: [2]
Set of primes before filtering: {2, 3, 5}
List of primes after filtering: [2, 3]
[2, 3]
```

18.10 Tate-Shafarevich group

If E is an elliptic curve over a global field K, the Tate-Shafarevich group is the subgroup of elements in $H^1(K, E)$ which map to zero under every global-to-local restriction map $H^1(K, E) \to H^1(K_v, E)$, one for each place v of K.

The group is usually denoted by the Russian letter Sha (Ш), in this document it will be denoted by Sha.

Sha is known to be an abelian torsion group. It is conjectured that the Tate-Shafarevich group is finite for any elliptic curve over a global field. But it is not known in general.

A theorem of Kolyvagin and Gross-Zagier using Heegner points shows that if the L-series of an elliptic curve $E/\mathbf{Q}$ does not vanish at 1 or has a simple zero there, then Sha is finite.

A theorem of Kato, together with theorems from Iwasawa theory, allows for certain primes p to show that the p-primary part of Sha is finite and gives an effective upper bound for it.

The (p-adic) conjecture of Birch and Swinnerton-Dyer predicts the order of Sha from the leading term of the (p-adic) L-series of the elliptic curve.

Sage can compute a few things about Sha. The commands `an`, `an_numerical` and `an_padic` compute the conjectural order of Sha as a real or p-adic number. With `p_primary_bound` one can find an upper bound of the size of the p-primary part of Sha. Finally, if the analytic rank is at most 1, then `bound_kato` and `bound_kolyvagin` find all primes for which the theorems of Kato and Kolyvagin respectively do not prove the triviality the p-primary part of Sha.

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: S = E.sha()
sage: S.bound_kato()
[2]
sage: S.bound_kolyvagin()
([2, 5], 1)
sage: S.an_padic(7,3)
1 + O(7^5)
sage: S.an()
1
sage: S.an_numerical()
```

(continues on next page)

(continued from previous page)

```
1.00000000000000

sage: E = EllipticCurve('389a')
sage: S = E.sha(); S
Tate-Shafarevich group for the
 Elliptic Curve defined by y^2 + y = x^3 + x^2 - 2*x over Rational Field
sage: S.an_numerical()
1.00000000000000
sage: S.p_primary_bound(5)
0
sage: S.an_padic(5)
1 + O(5)
sage: S.an_padic(5,prec=4)  # long time (2s on sage.math, 2011)
1 + O(5^3)
```

AUTHORS:

- William Stein (2007) – initial version
- Chris Wuthrich (April 2009) – reformat docstrings
- Aly Deines, Chris Wuthrich, Jeaninne Van Order (2016-03): Added functionality that tests the Skinner-Urban condition.

class sage.schemes.elliptic_curves.sha_tate.**Sha**(E)

Bases: SageObject

The Tate-Shafarevich group associated to an elliptic curve.

If E is an elliptic curve over a global field K, the Tate-Shafarevich group is the subgroup of elements in $H^1(K, E)$ which map to zero under every global-to-local restriction map $H^1(K, E) \to H^1(K_v, E)$, one for each place v of K.

EXAMPLES:

```
sage: E = EllipticCurve('571a1')
sage: E._set_gens([])   # curve has rank 0, but non-trivial Sha[2]
sage: S = E.sha()
sage: S.bound_kato()
[2]
sage: S.bound_kolyvagin()
([2], 1)
sage: S.an_padic(7,3)
4 + O(7^5)
sage: S.an()
4
sage: S.an_numerical()
4.00000000000000

sage: E = EllipticCurve('389a')
sage: S = E.sha(); S
Tate-Shafarevich group for the
 Elliptic Curve defined by y^2 + y = x^3 + x^2 - 2*x over Rational Field
sage: S.an_numerical()
1.00000000000000
sage: S.p_primary_bound(5)  # long time
0
sage: S.an_padic(5)         # long time
1 + O(5)
```

(continues on next page)

(continued from previous page)

```
sage: S.an_padic(5,prec=4)  # very long time
1 + O(5^3)
```

an(*use_database=False*, *descent_second_limit=12*)

Return the Birch and Swinnerton-Dyer conjectural order of Sha as a provably correct integer, unless the analytic rank is > 1, in which case this function returns a numerical value.

INPUT:

- `use_database` – bool (default: `False`); if `True`, try to use any databases installed to lookup the analytic order of Sha, if possible. The order of Sha is computed if it cannot be looked up.
- `descent_second_limit` – int (default: 12); limit to use on point searching for the quartic twist in the hard case

This result is proved correct if the order of vanishing is 0 and the Manin constant is <= 2.

If the optional parameter `use_database` is `True` (default: `False`), this function returns the analytic order of Sha as listed in Cremona's tables, if this curve appears in Cremona's tables.

NOTE:

If you come across the following error:

```
sage: E = EllipticCurve([0, 0, 1, -34874, -2506691])
sage: E.sha().an()
Traceback (most recent call last):
...
RuntimeError: Unable to compute the rank, hence generators, with certainty
(lower bound=0, generators found=[]).  This could be because Sha(E/Q)[2] is
nontrivial. Try increasing descent_second_limit then trying this command␣
↪again.
```

You can increase the `descent_second_limit` (in the above example, set to the default, 12) option to try again:

```
sage: E.sha().an(descent_second_limit=16)  # long time (2s on sage.math, 2011)
1
```

EXAMPLES:

```
sage: E = EllipticCurve([0, -1, 1, -10, -20])    # 11A  = X_0(11)
sage: E.sha().an()
1
sage: E = EllipticCurve([0, -1, 1, 0, 0])        # X_1(11)
sage: E.sha().an()
1

sage: EllipticCurve('14a4').sha().an()
1
sage: EllipticCurve('14a4').sha().an(use_database=True)    # will be faster if␣
↪you have large Cremona database installed
1
```

The smallest conductor curve with nontrivial Sha:

```
sage: E = EllipticCurve([1,1,1,-352,-2689])      # 66b3
sage: E.sha().an()
4
```

The four optimal quotients with nontrivial Sha and conductor <= 1000:

```
sage: E = EllipticCurve([0, -1, 1, -929, -10595])       # 571A
sage: E.sha().an()
4
sage: E = EllipticCurve([1, 1, 0, -1154, -15345])       # 681B
sage: E.sha().an()
9
sage: E = EllipticCurve([0, -1, 0, -900, -10098])       # 960D
sage: E.sha().an()
4
sage: E = EllipticCurve([0, 1, 0, -20, -42])            # 960N
sage: E.sha().an()
4
```

The smallest conductor curve of rank > 1:

```
sage: E = EllipticCurve([0, 1, 1, -2, 0])       # 389A (rank 2)
sage: E.sha().an()
1.00000000000000
```

The following are examples that require computation of the Mordell- Weil group and regulator:

```
sage: E = EllipticCurve([0, 0, 1, -1, 0])                    # 37A  (rank 1)
sage: E.sha().an()
1

sage: E = EllipticCurve("1610f3")
sage: E.sha().an()
4
```

In this case the input curve is not minimal, and if this function did not transform it to be minimal, it would give nonsense:

```
sage: E = EllipticCurve([0, -432*6^2])
sage: E.sha().an()
1
```

See Issue #10096: this used to give the wrong result 6.0000 before since the minimal model was not used:

```
sage: E = EllipticCurve([1215*1216, 0]) # non-minimal model
sage: E.sha().an()  # long time (2s on sage.math, 2011)
1.00000000000000
sage: E.minimal_model().sha().an()  # long time (1s on sage.math, 2011)
1.00000000000000
```

an_numerical (*prec=None*, *use_database=True*, *proof=None*)

Return the numerical analytic order of Sha, which is a floating point number in all cases.

INPUT:

- `prec` – integer (default: 53) bits precision – used for the L-series computation, period, regulator, etc.
- `use_database` – whether the rank and generators should be looked up in the database if possible. Default is `True`
- `proof` – bool or `None` (default: `None`, see proof.[tab] or sage.structure.proof) proof option passed onto regulator and rank computation.

Note: See also the `an()` command, which will return a provably correct integer when the rank is 0 or 1.

Warning: If the curve's generators are not known, computing them may be very time-consuming. Also, computation of the L-series derivative will be time-consuming for large rank and large conductor, and the computation time for this may increase substantially at greater precision. However, use of very low precision less than about 10 can cause the underlying PARI library functions to fail.

EXAMPLES:

```
sage: EllipticCurve('11a').sha().an_numerical()
1.00000000000000
sage: EllipticCurve('37a').sha().an_numerical()
1.00000000000000
sage: EllipticCurve('389a').sha().an_numerical()
1.00000000000000
sage: EllipticCurve('66b3').sha().an_numerical()
4.00000000000000
sage: EllipticCurve('5077a').sha().an_numerical()
1.00000000000000
```

A rank 4 curve:

```
sage: EllipticCurve([1, -1, 0, -79, 289]).sha().an_numerical()  # long time␣
↪(3s on sage.math, 2011)
1.00000000000000
```

A rank 5 curve:

```
sage: EllipticCurve([0, 0, 1, -79, 342]).sha().an_numerical(prec=10,␣
↪proof=False)  # long time (22s on sage.math, 2011)
1.0
```

See Issue #1115:

```
sage: sha = EllipticCurve('37a1').sha()
sage: [sha.an_numerical(prec) for prec in range(40,100,10)]  # long time (3s␣
↪on sage.math, 2013)
[1.0000000000,
 1.0000000000000,
 1.0000000000000000,
 1.0000000000000000000,
 1.0000000000000000000000,
 1.0000000000000000000000000]
```

an_padic(*p*, *prec=0*, *use_twists=True*)

Return the conjectural order of $Sha(E/\mathbf{Q})$, according to the p-adic analogue of the Birch and Swinnerton-Dyer conjecture as formulated in [MTT1986] and [BP1993].

INPUT:

- `p` – a prime > 3
- `prec` (optional) – the precision used in the computation of the p-adic L-Series

- `use_twists` (default: `True`) – If `True` the algorithm may change to a quadratic twist with minimal conductor to do the modular symbol computations rather than using the modular symbols of the curve itself. If `False` it forces the computation using the modular symbols of the curve itself.

OUTPUT: p-adic number - that conjecturally equals $\#Sha(E/\mathbf{Q})$.

If `prec` is set to zero (default) then the precision is set so that at least the first p-adic digit of conjectural $\#Sha(E/\mathbf{Q})$ is determined.

EXAMPLES:

Good ordinary examples:

```
sage: EllipticCurve('11a1').sha().an_padic(5)     # rank 0
1 + O(5^22)
sage: EllipticCurve('43a1').sha().an_padic(5)     # rank 1
1 + O(5)
sage: EllipticCurve('389a1').sha().an_padic(5,4) # rank 2, long time (2s on␣
↪sage.math, 2011)
1 + O(5^3)
sage: EllipticCurve('858k2').sha().an_padic(7)    # rank 0, non trivial sha,␣
↪long time (10s on sage.math, 2011)
7^2 + O(7^24)
sage: EllipticCurve('300b2').sha().an_padic(3)    # 9 elements in sha, long␣
↪time (2s on sage.math, 2011)
3^2 + O(3^24)
sage: EllipticCurve('300b2').sha().an_padic(7, prec=6)  # long time
2 + 7 + O(7^8)
```

Exceptional cases:

```
sage: EllipticCurve('11a1').sha().an_padic(11) # rank 0
1 + O(11^22)
sage: EllipticCurve('130a1').sha().an_padic(5) # rank 1
1 + O(5)
```

Non-split, but rank 0 case (Issue #7331):

```
sage: EllipticCurve('270b1').sha().an_padic(5) # rank 0, long time (2s on␣
↪sage.math, 2011)
1 + O(5^22)
```

The output has the correct sign:

```
sage: EllipticCurve('123a1').sha().an_padic(41) # rank 1, long time (3s on␣
↪sage.math, 2011)
1 + O(41)
```

Supersingular cases:

```
sage: EllipticCurve('34a1').sha().an_padic(5) # rank 0
1 + O(5^22)
sage: EllipticCurve('53a1').sha().an_padic(5) # rank 1, long time (11s on␣
↪sage.math, 2011)
1 + O(5)
```

Cases that use a twist to a lower conductor:

```
sage: EllipticCurve('99a1').sha().an_padic(5)
1 + O(5)
sage: EllipticCurve('240d3').sha().an_padic(5)  # sha has 4 elements here
4 + O(5)
sage: EllipticCurve('448c5').sha().an_padic(7, prec=4, use_twists=False)  #␣
↪long time (2s on sage.math, 2011)
2 + 7 + O(7^6)
sage: EllipticCurve([-19,34]).sha().an_padic(5)  # see trac #6455, long time␣
↪(4s on sage.math, 2011)
1 + O(5)
```

Test for Issue #15737:

```
sage: E = EllipticCurve([-100,0])
sage: s = E.sha()
sage: s.an_padic(13)
1 + O(13^20)
```

bound()

Compute a provably correct bound on the order of the Tate-Shafarevich group of this curve.

The bound is either `False` (no bound) or a list `B` of primes such that any prime divisor of the order of Sha is in this list.

EXAMPLES:

```
sage: EllipticCurve('37a').sha().bound()
([2], 1)
```

bound_kato()

Return a list of primes p such that the theorems of Kato's [Kat2004] and others (e.g., as explained in a thesis of Grigor Grigorov [Gri2005]) imply that if p divides the order of $Sha(E/\mathbf{Q})$ then p is in the list.

If $L(E,1) = 0$, then this function gives no information, so it returns `False`.

THEOREM: Suppose $L(E,1) \neq 0$ and $p \neq 2$ is a prime such that

- E does not have additive reduction at p,
- either the p-adic representation is surjective or has its image contained in a Borel subgroup.

Then $ord_p(\#Sha(E))$ is bounded from above by the p-adic valuation of $L(E,1) \cdot \#E(\mathbf{Q})_{tor}^2/(\Omega_E \cdot \prod c_v)$.

If the L-series vanishes, the method `p_primary_bound` can be used instead.

EXAMPLES:

```
sage: E = EllipticCurve([0, -1, 1, -10, -20])    # 11A  = X_0(11)
sage: E.sha().bound_kato()
[2]
sage: E = EllipticCurve([0, -1, 1, 0, 0])        # X_1(11)
sage: E.sha().bound_kato()
[2]
sage: E = EllipticCurve([1,1,1,-352,-2689])      # 66B3
sage: E.sha().bound_kato()
[2]
```

For the following curve one really has that 25 divides the order of Sha (by [GJPST2009]):

```
sage: E = EllipticCurve([1, -1, 0, -332311, -73733731])   # 1058D1
sage: E.sha().bound_kato()                                # long time (about 1 second)
[2, 5, 23]
sage: E.galois_representation().non_surjective()  # long time (about 1 second)
[]
```

For this one, Sha is divisible by 7:

```
sage: E = EllipticCurve([0, 0, 0, -4062871, -3152083138])   # 3364C1
sage: E.sha().bound_kato()                                  # long time (< 10 seconds)
[2, 7, 29]
```

No information about curves of rank > 0:

```
sage: E = EllipticCurve([0, 0, 1, -1, 0])       # 37A  (rank 1)
sage: E.sha().bound_kato()
False
```

bound_kolyvagin (*D=0*, *regulator=None*, *ignore_nonsurj_hypothesis=False*)

Given a fundamental discriminant $D \neq -3, -4$ that satisfies the Heegner hypothesis for E, return a list of primes so that Kolyvagin's theorem (as in Gross's paper) implies that any prime divisor of Sha is in this list.

INPUT:

- `D` – (optional) a fundamental discriminant < -4 that satisfies the Heegner hypothesis for E; if not given, use the first such D
- `regulator` – (optional) regulator of $E(K)$; if not given, will be computed (which could take a long time)
- `ignore_nonsurj_hypothesis` (optional: default `False`) – If `True`, then gives the bound coming from Heegner point index, but without any hypothesis on surjectivity of the mod-p representation.

OUTPUT:

- list – a list of primes such that if p divides $Sha(E/K)$, then p is in this list, unless E/K has complex multiplication or analytic rank greater than 2 (in which case we return 0).
- index – the odd part of the index of the Heegner point in the full group of K-rational points on E. (If E has CM, returns 0.)

REMARKS:

1) We do not have to assume that the Manin constant is 1 (or a power of 2). If the Manin constant were divisible by a prime, that prime would get included in the list of bad primes.

2) We assume the Gross-Zagier theorem is true under the hypothesis that $gcd(N, D) = 1$, instead of the stronger hypothesis $gcd(2 \cdot N, D) = 1$ that is in the original Gross-Zagier paper. That Gross-Zagier is true when $gcd(N, D) = 1$ is "well-known" to the experts, but does not seem to written up well in the literature.

3) Correctness of the computation is guaranteed using interval arithmetic, under the assumption that the regulator, square root, and period lattice are computed to precision at least 10^{-10}, i.e., they are correct up to addition or a real number with absolute value less than 10^{-10}.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.sha().bound_kolyvagin()
([2], 1)
```

(continues on next page)

(continued from previous page)

```
sage: E = EllipticCurve('141a')
sage: E.sha().an()
1
sage: E.sha().bound_kolyvagin()
([2, 7], 49)
```

We get no information when the curve has rank 2.:

```
sage: E = EllipticCurve('389a')
sage: E.sha().bound_kolyvagin()
(0, 0)
sage: E = EllipticCurve('681b')
sage: E.sha().an()
9
sage: E.sha().bound_kolyvagin()
([2, 3], 9)
```

p_primary_bound(p)

Return a provable upper bound for the order of the p-primary part $Sha(E)(p)$ of the Tate-Shafarevich group.

INPUT:

- p – a prime > 2

OUTPUT:

- e – a non-negative integer such that p^e is an upper bound for the order of $Sha(E)(p)$

In particular, if this algorithm does not fail, then it proves that the p-primary part of Sha is finite. This works also for curves of rank > 1.

Note also that this bound is sharp if one assumes the main conjecture of Iwasawa theory of elliptic curves. One may use the method p_primary_order for checking if the extra conditions hold under which the main conjecture is known by the work of Skinner and Urban. This then returns the provable p-primary part of the Tate-Shafarevich group.

Currently the algorithm is only implemented when the following conditions are verified:

- The p-adic Galois representation must be surjective or must have its image contained in a Borel subgroup.
- The reduction at p is not allowed to be additive.
- If the reduction at p is non-split multiplicative, then the rank must be 0.
- If $p = 3$, then the reduction at 3 must be good ordinary or split multiplicative, and the rank must be 0.

ALGORITHM:

The algorithm is described in [SW2013]. The results for the reducible case can be found in [Wu2004]. The main ingredient is Kato's result on the main conjecture in Iwasawa theory.

EXAMPLES:

```
sage: e = EllipticCurve('11a3')
sage: e.sha().p_primary_bound(3)
0
sage: e.sha().p_primary_bound(5)
0
sage: e.sha().p_primary_bound(7)
0
sage: e.sha().p_primary_bound(11)
```

(continues on next page)

(continued from previous page)

```
0
sage: e.sha().p_primary_bound(13)
0

sage: e = EllipticCurve('389a1')
sage: e.sha().p_primary_bound(5)
0
sage: e.sha().p_primary_bound(7)
0
sage: e.sha().p_primary_bound(11)
0
sage: e.sha().p_primary_bound(13)
0

sage: e = EllipticCurve('858k2')
sage: e.sha().p_primary_bound(3)  # long time (10s on sage.math, 2011)
0
```

Some checks for Issue #6406 and Issue #16959:

```
sage: e.sha().p_primary_bound(7)  # long time
2

sage: E = EllipticCurve('608b1')
sage: E.sha().p_primary_bound(5)
Traceback (most recent call last):
...
ValueError: The p-adic Galois representation is not surjective or reducible.
Current knowledge about Euler systems does not provide an upper bound
in this case. Try an_padic for a conjectural bound.

sage: E.sha().an_padic(5)           # long time
1 + O(5^22)

sage: E = EllipticCurve("5040bi1")
sage: E.sha().p_primary_bound(5)    # long time
0
```

p_primary_order(p)

Return the order of the p-primary part of the Tate-Shafarevich group.

This uses the result of Skinner and Urban [SU2014] on the main conjecture in Iwasawa theory. In particular the elliptic curve must have good ordinary reduction at p, the residual Galois representation must be surjective. Furthermore there must be an auxiliary prime ℓ dividing the conductor of the curve exactly once such that the residual representation is ramified at p.

INPUT:

- p – an odd prime

OUTPUT:

- e – a non-negative integer such that p^e is the order of the p-primary order if the conditions are satisfied and raises a `ValueError` otherwise.

EXAMPLES:

```
sage: E = EllipticCurve("389a1")  # rank 2
sage: E.sha().p_primary_order(5)
0
sage: E = EllipticCurve("11a1")
sage: E.sha().p_primary_order(7)
0
sage: E.sha().p_primary_order(5)
Traceback (most recent call last):
...
ValueError: The order is not provably known using Skinner-Urban.
Try running p_primary_bound to get a bound.
```

two_selmer_bound()

Return the 2-rank, i.e. the $\mathbf{F}_2$-dimension of the 2-torsion part of Sha, provided we can determine the rank of E.

EXAMPLES:

```
sage: sh = EllipticCurve('571a1').sha()
sage: sh.two_selmer_bound()
2
sage: sh.an()
4

sage: sh = EllipticCurve('66a1').sha()
sage: sh.two_selmer_bound()
0
sage: sh.an()
1

sage: sh = EllipticCurve('960d1').sha()
sage: sh.two_selmer_bound()
2
sage: sh.an()
4
```

18.11 Complex multiplication for elliptic curves

This module implements the functions

- hilbert_class_polynomial
- is_HCP
- cm_j_invariants
- cm_orders
- discriminants_with_bounded_class_number
- cm_j_invariants_and_orders
- largest_fundamental_disc_with_class_number
- is_cm_j_invariant

AUTHORS:

- Robert Bradshaw

- John Cremona
- William Stein

sage.schemes.elliptic_curves.cm.**OrderClassNumber**(*D0*, *h0*, *f*)

Return the class number h(f**2 * D0), given h(D0)=h0.

INPUT:

- `D0` (integer) – a negative fundamental discriminant
- `h0` (integer) – the class number of the (maximal) imaginary quadratic order of discriminant `D0`
- `f` (integer) – a positive integer

OUTPUT:

(integer) the class number of the imaginary quadratic order of discriminant `D0*f**2`

ALGORITHM:

We use the formula for the class number of the order $\mathcal{O}_D$ in terms of the class number of the maximal order $\mathcal{O}_{D_0}$; see [Cox1989] Theorem 7.24:

$$h(D) = \frac{h(D_0)f}{[\mathcal{O}_{D_0}^\times : \mathcal{O}_D^\times]} \prod_{p \mid f} \left(1 - \left(\frac{D_0}{p}\right)\frac{1}{p}\right)$$

EXAMPLES:

```
sage: # needs sage.libs.pari
sage: from sage.schemes.elliptic_curves.cm import OrderClassNumber
sage: D0 = -4
sage: h = D0.class_number()
sage: [OrderClassNumber(D0,h,f) for f in srange(1,20)]
[1, 1, 2, 2, 2, 4, 4, 4, 6, 4, 6, 8, 6, 8, 8, 8, 8, 12, 10]
sage: all([OrderClassNumber(D0,h,f) == (D0*f**2).class_number() for f in srange(1,
↪20)])
True
```

sage.schemes.elliptic_curves.cm.**cm_j_invariants**(*proof=None*)

Return a list of all CM j-invariants in the field K.

INPUT:

- `K` – a number field
- `proof` – (default: proof.number_field())

OUTPUT:

(list) – A list of CM j-invariants in the field K.

EXAMPLES:

```
sage: cm_j_invariants(QQ)
[-262537412640768000, -147197952000, -884736000, -12288000, -884736,
 -32768, -3375, 0, 1728, 8000, 54000, 287496, 16581375]
```

Over imaginary quadratic fields there are no more than over QQ:

```
sage: cm_j_invariants(QuadraticField(-1, 'i'))                                  #␣
↪needs sage.rings.number_field
[-262537412640768000, -147197952000, -884736000, -12288000, -884736,
 -32768, -3375, 0, 1728, 8000, 54000, 287496, 16581375]
```

Over real quadratic fields there may be more, for example:

```
sage: len(cm_j_invariants(QuadraticField(5, 'a')))                              #␣
↪needs sage.rings.number_field
31
```

Over number fields K of many higher degrees this also works:

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)
sage: cm_j_invariants(K)
[-262537412640768000, -147197952000, -884736000, -884736, -32768,
 8000, -3375, 16581375, 1728, 287496, 0, 54000, -12288000,
 31710790944000*a^2 + 39953093016000*a + 50337742902000]
sage: K.<a> = NumberField(x^4 - 2)
sage: len(cm_j_invariants(K))
23
```

sage.schemes.elliptic_curves.cm.**cm_j_invariants_and_orders**(*proof=None*)

Return a list of all CM j-invariants in the field K, together with the associated orders.

INPUT:

- `K` – a number field
- `proof` – (default: proof.number_field())

OUTPUT:

(list) A list of 3-tuples (D, f, j) where j is a CM j-invariant in K with quadratic fundamental discriminant D and conductor f.

EXAMPLES:

```
sage: cm_j_invariants_and_orders(QQ)
[(-3, 3, -12288000), (-3, 2, 54000), (-3, 1, 0), (-4, 2, 287496), (-4, 1, 1728),
 (-7, 2, 16581375), (-7, 1, -3375), (-8, 1, 8000), (-11, 1, -32768),
 (-19, 1, -884736), (-43, 1, -884736000), (-67, 1, -147197952000),
 (-163, 1, -262537412640768000)]
```

Over an imaginary quadratic field there are no more than over QQ:

```
sage: cm_j_invariants_and_orders(QuadraticField(-1, 'i'))                       #␣
↪needs sage.rings.number_field
[(-163, 1, -262537412640768000), (-67, 1, -147197952000),
 (-43, 1, -884736000), (-19, 1, -884736), (-11, 1, -32768),
 (-8, 1, 8000), (-7, 1, -3375), (-7, 2, 16581375), (-4, 1, 1728),
 (-4, 2, 287496), (-3, 1, 0), (-3, 2, 54000), (-3, 3, -12288000)]
```

Over real quadratic fields there may be more:

```
sage: v = cm_j_invariants_and_orders(QuadraticField(5,'a')); len(v)             #␣
↪needs sage.rings.number_field
31
sage: [(D, f) for D, f, j in v if j not in QQ]                                  #␣
↪needs sage.rings.number_field
[(-235, 1), (-235, 1), (-115, 1), (-115, 1), (-40, 1), (-40, 1),
 (-35, 1), (-35, 1), (-20, 1), (-20, 1), (-15, 1), (-15, 1), (-15, 2),
 (-15, 2), (-4, 5), (-4, 5), (-3, 5), (-3, 5)]
```

Over number fields K of many higher degrees this also works:

```
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)                                              #␣
↪needs sage.rings.number_field
sage: cm_j_invariants_and_orders(K)                                             #␣
↪needs sage.rings.number_field
[(-163, 1, -262537412640768000), (-67, 1, -147197952000),
 (-43, 1, -884736000), (-19, 1, -884736), (-11, 1, -32768),
 (-8, 1, 8000), (-7, 1, -3375), (-7, 2, 16581375), (-4, 1, 1728),
 (-4, 2, 287496), (-3, 1, 0), (-3, 2, 54000), (-3, 3, -12288000),
 (-3, 6, 31710790944000*a^2 + 39953093016000*a + 50337742902000)]
```

sage.schemes.elliptic_curves.cm.**cm_orders**(*proof=None*)

Return a list of all pairs (D, f) where there is a CM order of discriminant Df^2 with class number h, with D a fundamental discriminant.

INPUT:

- h – positive integer
- `proof` – (default: proof.number_field())

OUTPUT:

- list of 2-tuples (D, f) sorted lexicographically by $(|D|, f)$

EXAMPLES:

```
sage: cm_orders(0)
[]
sage: v = cm_orders(1); v
[(-3, 1), (-3, 2), (-3, 3), (-4, 1), (-4, 2), (-7, 1), (-7, 2), (-8, 1),
 (-11, 1), (-19, 1), (-43, 1), (-67, 1), (-163, 1)]
sage: type(v[0][0]), type(v[0][1])
(<... 'sage.rings.integer.Integer'>, <... 'sage.rings.integer.Integer'>)
sage: # needs sage.libs.pari
sage: v = cm_orders(2); v
 [(-3, 4), (-3, 5), (-3, 7), (-4, 3), (-4, 4), (-4, 5), (-7, 4), (-8, 2),
  (-8, 3), (-11, 3), (-15, 1), (-15, 2), (-20, 1), (-24, 1), (-35, 1),
  (-40, 1), (-51, 1), (-52, 1), (-88, 1), (-91, 1), (-115, 1), (-123, 1),
  (-148, 1), (-187, 1), (-232, 1), (-235, 1), (-267, 1), (-403, 1), (-427, 1)]
sage: len(v)
29
sage: set([hilbert_class_polynomial(D*f^2).degree() for D,f in v])
{2}
```

Any degree up to 100 is implemented, but may be slow:

```
sage: # needs sage.libs.pari
sage: cm_orders(3)
[(-3, 6), (-3, 9), (-11, 2), (-19, 2), (-23, 1), (-23, 2), (-31, 1), (-31, 2),
 (-43, 2), (-59, 1), (-67, 2), (-83, 1), (-107, 1), (-139, 1), (-163, 2),
 (-211, 1), (-283, 1), (-307, 1), (-331, 1), (-379, 1), (-499, 1), (-547, 1),
 (-643, 1), (-883, 1), (-907, 1)]
sage: len(cm_orders(4))
84
```

sage.schemes.elliptic_curves.cm.**discriminants_with_bounded_class_number**(*hmax*, *B=None*, *proof=None*)

Return a dictionary with keys class numbers $h \leq hmax$ and values the list of all pairs (D_0, f), with $D_0 < 0$ a fundamental discriminant such that $D = D_0 f^2$ has class number h. If the optional bound B is given, return only those pairs with $|D| \leq B$.

INPUT:

- `hmax` – integer
- B – integer or None; if None returns all pairs
- `proof` – this code calls the PARI function pari:qfbclassno, so it could give wrong answers when `proof``==``False` (though only for discriminants greater than $2 \cdot 10^{10}$). The default is the current value of `proof.number_field()`.

OUTPUT:

- dictionary

Note: In case B is not given, then `hmax` must be at most 100; we use the tables from [Watkins2004] and [Klaise2012] to compute a B that captures all h up to $hmax$.

EXAMPLES:

```
sage: # needs sage.libs.pari
sage: from sage.schemes.elliptic_curves.cm import discriminants_with_bounded_
↪class_number
sage: v = discriminants_with_bounded_class_number(3)
sage: sorted(v)
[1, 2, 3]
sage: v[1]
[(-3, 1), (-3, 2), (-3, 3), (-4, 1), (-4, 2), (-7, 1), (-7, 2), (-8, 1),
 (-11, 1), (-19, 1), (-43, 1), (-67, 1), (-163, 1)]
sage: v[2]
[(-3, 4), (-3, 5), (-3, 7), (-4, 3), (-4, 4), (-4, 5), (-7, 4), (-8, 2),
 (-8, 3), (-11, 3), (-15, 1), (-15, 2), (-20, 1), (-24, 1), (-35, 1), (-40, 1),
 (-51, 1), (-52, 1), (-88, 1), (-91, 1), (-115, 1), (-123, 1), (-148, 1),
 (-187, 1), (-232, 1), (-235, 1), (-267, 1), (-403, 1), (-427, 1)]
sage: v[3]
[(-3, 6), (-3, 9), (-11, 2), (-19, 2), (-23, 1), (-23, 2), (-31, 1), (-31, 2),
 (-43, 2), (-59, 1), (-67, 2), (-83, 1), (-107, 1), (-139, 1), (-163, 2),
 (-211, 1), (-283, 1), (-307, 1), (-331, 1), (-379, 1), (-499, 1), (-547, 1),
 (-643, 1), (-883, 1), (-907, 1)]
sage: v = discriminants_with_bounded_class_number(8, proof=False)
sage: sorted(len(v[h]) for h in v)
[13, 25, 29, 29, 38, 84, 101, 208]
```

Find all class numbers for discriminant up to 50:

```
sage: sage.schemes.elliptic_curves.cm.discriminants_with_bounded_class_
↪number(hmax=5, B=50)
{1: [(-3, 1), (-3, 2), (-3, 3), (-4, 1), (-4, 2), (-7, 1), (-7, 2), (-8, 1), (-11,
↪ 1), (-19, 1), (-43, 1)], 2: [(-3, 4), (-4, 3), (-8, 2), (-15, 1), (-20, 1), (-
↪24, 1), (-35, 1), (-40, 1)], 3: [(-11, 2), (-23, 1), (-31, 1)], 4: [(-39, 1)],␣
↪5: [(-47, 1)]}
```

`sage.schemes.elliptic_curves.cm.`**`hilbert_class_polynomial`**(*algorithm=None*)

Return the Hilbert class polynomial for discriminant D.

INPUT:

- `D` (int) – a negative integer congruent to 0 or 1 modulo 4.
- `algorithm` (string, default None).

OUTPUT:

(integer polynomial) The Hilbert class polynomial for the discriminant D.

ALGORITHM:

- If `algorithm` = "arb" (default): Use FLINT's implementation inherited from Arb which uses complex interval arithmetic.
- If `algorithm` = "sage": Use complex approximations to the roots.
- If `algorithm` = "magma": Call the appropriate Magma function (if available).

AUTHORS:

- Sage implementation originally by Eduardo Ocampo Alvarez and AndreyTimofeev
- Sage implementation corrected by John Cremona (using corrected precision bounds from Andreas Enge)
- Magma implementation by David Kohel

EXAMPLES:

```
sage: # needs sage.libs.flint
sage: hilbert_class_polynomial(-4)
x - 1728
sage: hilbert_class_polynomial(-7)
x + 3375
sage: hilbert_class_polynomial(-23)
x^3 + 3491750*x^2 - 5151296875*x + 12771880859375
sage: hilbert_class_polynomial(-37*4)
x^2 - 39660183801072000*x - 7898242515936467904000000
sage: hilbert_class_polynomial(-37*4, algorithm="magma") # optional - magma
x^2 - 39660183801072000*x - 7898242515936467904000000
sage: hilbert_class_polynomial(-163)
x + 262537412640768000
sage: hilbert_class_polynomial(-163, algorithm="sage")
x + 262537412640768000
sage: hilbert_class_polynomial(-163, algorithm="magma") # optional - magma
x + 262537412640768000
```

`sage.schemes.elliptic_curves.cm.`**`is_HCP`**(*f*, *check_monic_irreducible=True*)

Determine whether a polynomial is a Hilbert Class Polynomial.

INPUT:

- `f` – a polynomial in $\mathbf{Z}[X]$.
- `check_monic_irreducible` (boolean, default `True`) – if `True`, check that `f` is a monic, irreducible, integer polynomial.

OUTPUT:

(integer) – either D if `f` is the Hilbert Class Polynomial H_D for discriminant D, or 0 if not an HCP.

ALGORITHM:

Cremona and Sutherland: Algorithm 2 of [CreSuth2023].

EXAMPLES:

Even for large degrees this is fast. We test the largest discriminant of class number 100, for which the HCP has coefficients with thousands of digits:

```
sage: from sage.schemes.elliptic_curves.cm import is_HCP
sage: D = -1856563
sage: D.class_number()                                                      #␣
↪needs sage.libs.pari
100

sage: # needs sage.libs.flint
sage: H = hilbert_class_polynomial(D)
sage: H.degree()
100
sage: max(H).ndigits()
2774
sage: is_HCP(H)
-1856563
```

Testing polynomials which are not HCPs is faster:

```
sage: is_HCP(H+1)                                                           #␣
↪needs sage.libs.flint
0
```

sage.schemes.elliptic_curves.cm.**is_cm_j_invariant**(*algorithm='CremonaSutherland'*, *method=None*)

Return whether or not this is a CM j-invariant, and the CM discriminant if it is.

INPUT:

- `j` – an element of a number field K
- `algorithm` (string, default 'CremonaSutherland') – the algorithm used, either 'CremonaSutherland' (the default, very much faster for all but very small degrees), 'exhaustive' or 'reduction'
- `method` (string) – deprecated name for `algorithm`

OUTPUT:

A pair (bool, (d,f)) which is either (False, None) if j is not a CM j-invariant or (True, (d,f)) if j is the j-invariant of the imaginary quadratic order of discriminant $D = df^2$ where d is the associated fundamental discriminant and f the index.

ALGORITHM:

The default algorithm used is to test whether the minimal polynomial of `j` is a Hilbert CLass Polynomail, using *`is_HCP()`* which implements Algorithm 2 of [CreSuth2023] by Cremona and Sutherland.

Two older algorithms are available, both of which are much slower except for very small degrees.

Method 'exhaustive' makes use of the complete and unconditionsl classification of all orders of class number up to 100, and hence will raise an error if j is an algebraic integer of degree greater than this.

Method 'reduction' constructs an elliptic curve over the number field $\mathbf{Q}(j)$ and computes its traces of Frobenius at several primes of degree 1.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.cm import is_cm_j_invariant
sage: is_cm_j_invariant(0)
(True, (-3, 1))
```

(continues on next page)

(continued from previous page)

```
sage: is_cm_j_invariant(8000)
(True, (-8, 1))

sage: # needs sage.rings.number_field
sage: K.<a> = QuadraticField(5)
sage: is_cm_j_invariant(282880*a + 632000)
(True, (-20, 1))
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)
sage: is_cm_j_invariant(31710790944000*a^2 + 39953093016000*a + 50337742902000)
(True, (-3, 6))
```

An example of large degree. This is only possible using the default algorithm:

```
sage: from sage.schemes.elliptic_curves.cm import is_cm_j_invariant
sage: D = -1856563
sage: H = hilbert_class_polynomial(D)                                       #␣
↪needs sage.libs.flint
sage: H.degree()                                                            #␣
↪needs sage.libs.flint
100
sage: K.<j> = NumberField(H)                                                #␣
↪needs sage.libs.flint sage.rings.number_field
sage: is_cm_j_invariant(j)                                                  #␣
↪needs sage.libs.flint sage.rings.number_field
(True, (-1856563, 1))
```

sage.schemes.elliptic_curves.cm.**largest_disc_with_class_number**(h)

Return largest absolute value of any negative discriminant with class number h, and the number of fundamental negative discriminants with that class number. This is known (unconditionally) for h up to 100, by work of Mark Watkins [Watkins2004] for fundamental discriminants, extended to all discriminants of class number $h \leq 100$ by Klaise [Klaise2012].

Note: The class number of a negative discriminant D is the same as the class number of the unique imaginary quadratic order of discriminant D, so this function gives the number of such orders of each class number $h \leq 100$. It is easy to extend this to larger class number conditional on the GRH, but much harder to obyain unconditional results.

INPUT:

- h – integer

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.cm import largest_disc_with_class_number
sage: largest_disc_with_class_number(0)
(0, 0)
sage: largest_disc_with_class_number(1)
(163, 13)
sage: largest_disc_with_class_number(2)
(427, 29)
sage: largest_disc_with_class_number(10)
(13843, 123)
sage: largest_disc_with_class_number(100)
(1856563, 2311)
```

(continues on next page)

(continued from previous page)

```
sage: largest_disc_with_class_number(101)
Traceback (most recent call last):
...
NotImplementedError: largest discriminant not available for class number 101
```

For most $h \leq 100$, the largest fundamental discriminant with class number h is also the largest discriminant, but this is not the case for some h:

```
sage: from sage.schemes.elliptic_curves.cm import largest_disc_with_class_number,␣
↪largest_fundamental_disc_with_class_number
sage: [h for h in range(1,101) if largest_disc_with_class_number(h)[0] != largest_
↪fundamental_disc_with_class_number(h)[0]]
[6, 8, 12, 16, 20, 30, 40, 42, 52, 70]
sage: largest_fundamental_disc_with_class_number(6)
(3763, 51)
sage: largest_disc_with_class_number(6)
(4075, 101)
```

sage.schemes.elliptic_curves.cm.**largest_fundamental_disc_with_class_number**(*h*)

Return largest absolute value of any fundamental negative discriminant with class number h, and the number of fundamental negative discriminants with that class number. This is known (unconditionally) for h up to 100, by work of Mark Watkins ([Watkins2004]).

Note: The class number of a fundamental negative discriminant D is the same as the class number of the imaginary quadratic field $\mathbf{Q}(\sqrt{D})$, so this function gives the number of such fields of each class number $h \leq 100$. It is easy to extend this to larger class number conditional on the GRH, but much harder to obtain unconditional results.

INPUT:

- h – integer

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.cm import largest_fundamental_disc_with_
↪class_number
sage: largest_fundamental_disc_with_class_number(0)
(0, 0)
sage: largest_fundamental_disc_with_class_number(1)
(163, 9)
sage: largest_fundamental_disc_with_class_number(2)
(427, 18)
sage: largest_fundamental_disc_with_class_number(10)
(13843, 87)
sage: largest_fundamental_disc_with_class_number(100)
(1856563, 1736)
sage: largest_fundamental_disc_with_class_number(101)
Traceback (most recent call last):
...
NotImplementedError: largest fundamental discriminant not available for class␣
↪number 101
```

18.12 Testing whether elliptic curves over number fields are Q-curves

AUTHORS:

- John Cremona (February 2021)

The code here implements the algorithm of Cremona and Najman presented in [CrNa2020].

`sage.schemes.elliptic_curves.Qcurves.`**`Step4Test`**(*E*, *B*, *oldB=0*, *verbose=False*)

Apply local Q-curve test to E at all primes up to B.

INPUT:

- E (elliptic curve): an elliptic curve defined over a number field
- B (integer): upper bound on primes to test
- `oldB` (integer, default 0): lower bound on primes to test
- `verbose` (boolean, default `False`): verbosity flag

OUTPUT:

Either (`False`, p), if the local test at p proves that E is not a $\mathbf{Q}$-curve, or (`True`, 0) if all local tests at primes between `oldB` and `B` fail to prove that E is not a $\mathbf{Q}$-curve.

ALGORITHM (see [CrNa2020] for details):

This local test at p only applies if E has good reduction at all of the primes lying above p in the base field K of E. It tests whether (1) E is either ordinary at all $P \mid p$, or supersingular at all; (2) if ordinary at all, it tests that the squarefree part of $a_P^2 - 4N(P)$ is the same for all $P \mid p$.

EXAMPLES:

A non-$\mathbf{Q}$-curve over a quartic field (with LMFDB label '4.4.8112.1-12.1-a1') fails this test at $p = 13$:

```
sage: from sage.schemes.elliptic_curves.Qcurves import Step4Test
sage: R.<x> = PolynomialRing(QQ)
sage: K.<a> = NumberField(R([3, 0, -5, 0, 1]))                                  #␣
↪needs sage.rings.number_field
sage: E = EllipticCurve([K([-3,-4,1,1]), K([4,-1,-1,0]), K([-2,0,1,0]),         #␣
↪needs sage.rings.number_field
....:                    K([-621,778,138,-178]), K([9509,2046,-24728,10380])])
sage: Step4Test(E, 100, verbose=True)                                           #␣
↪needs sage.rings.number_field
No: inconsistency at the 2 ordinary primes dividing 13
- Frobenius discriminants mod squares: [-3, -1]
(False, 13)
```

A $\mathbf{Q}$-curve over a sextic field (with LMFDB label '6.6.1259712.1-64.1-a6') passes this test for all $p < 100$:

```
sage: from sage.schemes.elliptic_curves.Qcurves import Step4Test
sage: R.<x> = PolynomialRing(QQ)
sage: K.<a> = NumberField(R([-3, 0, 9, 0, -6, 0, 1]))                           #␣
↪needs sage.rings.number_field
sage: E = EllipticCurve([K([1,-3,0,1,0,0]), K([5,-3,-6,1,1,0]),                 #␣
↪needs sage.rings.number_field
....:                    K([1,-3,0,1,0,0]), K([-139,-129,331,277,-76,-63]),
....:                    K([2466,1898,-5916,-4582,1361,1055])])
sage: Step4Test(E, 100, verbose=True)                                           #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.rings.number_field
(True, 0)
```

sage.schemes.elliptic_curves.Qcurves.**conjugacy_test**(*jlist*, *verbose=False*)

Test whether a list of algebraic numbers contains a complete conjugacy class of 2-power degree.

INPUT:

- `jlist` (list): a list of algebraic numbers in the same field
- `verbose` (boolean, default `False`): verbosity flag

OUTPUT:

A possibly empty list of irreducible polynomials over **Q** of 2-power degree all of whose roots are in the list.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: from sage.schemes.elliptic_curves.Qcurves import conjugacy_test
sage: conjugacy_test([3])
[x - 3]
sage: K.<a> = QuadraticField(2)
sage: conjugacy_test([K(3), a])
[x - 3]
sage: conjugacy_test([K(3), 3 + a])
[x - 3]
sage: conjugacy_test([3 + a])
[]
sage: conjugacy_test([3 + a, 3 - a])
[x^2 - 6*x + 7]
sage: x = polygen(QQ)
sage: f = x^3 - 3
sage: K.<a> = f.splitting_field()
sage: js = f.roots(K, multiplicities=False)
sage: conjugacy_test(js)
[]
sage: f = x^4 - 3
sage: K.<a> = NumberField(f)
sage: js = f.roots(K, multiplicities=False)
sage: conjugacy_test(js)
[]
sage: K.<a> = f.splitting_field()
sage: js = f.roots(K, multiplicities=False)
sage: conjugacy_test(js)
[x^4 - 3]
```

sage.schemes.elliptic_curves.Qcurves.**is_Q_curve**(*E*, *maxp=100*, *certificate=False*, *verbose=False*)

Return whether `E` is a **Q**-curve, with optional certificate.

INPUT:

- `E` (elliptic curve) – an elliptic curve over a number field.
- `maxp` (int, default 100): bound on primes used for checking necessary local conditions. The result will not depend on this, but using a larger value may return `False` faster.
- `certificate` (bool, default `False`): if `True` then a second value is returned giving a certificate for the **Q**-curve property.

OUTPUT:

If `certificate` is `False`: either `True` (if E is a **Q**-curve), or `False`.

If `certificate` is `True`: a tuple consisting of a boolean flag as before and a certificate, defined as follows:

- when the flag is `True`, so E is a **Q**-curve:
  - either {'CM':D} where D is a negative discriminant, when E has potential CM with discriminant D;
  - otherwise {'CM': 0, 'core_poly': f, 'rho': ρ, 'r': r, 'N': N}, when E is a non-CM **Q**-curve, where the core polynomial f is an irreducible monic polynomial over QQ of degree 2^ρ, all of whose roots are j-invariants of curves isogenous to E, the core level N is a square-free integer with r prime factors which is the LCM of the degrees of the isogenies between these conjugates. For example, if there exists a curve E' isogenous to E with $j(E') = j \in \mathbf{Q}$, then the certificate is {'CM':0, 'r':0, 'rho':0, 'core_poly': x-j, 'N':1}.
- when the flag is `False`, so E is not a **Q**-curve, the certificate is a prime p such that the reductions of E at the primes dividing p are inconsistent with the property of being a **Q**-curve. See the ALGORITHM section for details.

ALGORITHM:

See [CrNa2020] for details.

1. If E has rational j-invariant, or has CM, then return `True`.

2. Replace E by a curve defined over $K = \mathbf{Q}(j(E))$. Let N be the conductor norm.

3. For all primes $p \mid N$ check that the valuations of j at all $P \mid p$ are either all negative or all non-negative; if not, return `False`.

4. For $p \leq maxp$, $p \not| N$, check that either E is ordinary mod P for all $P \mid p$, or E is supersingular mod P for all $P \mid p$; if neither, return `False`. If all are ordinary, check that the integers $a_P(E)^2 - 4N(P)$ have the same square-free part; if not, return `False`.

5. Compute the K-isogeny class of E using the "heuristic" option (which is faster, but not guaranteed to be complete). Check whether the set of j-invariants of curves in the class of 2-power degree contains a complete Galois orbit. If so, return `True`.

6. Otherwise repeat step 4 for more primes, and if still undecided, repeat Step 5 without the "heuristic" option, to get the complete K-isogeny class (which will probably be no bigger than before). Now return `True` if the set of j-invariants of curves in the class contains a complete Galois orbit, otherwise return `False`.

EXAMPLES:

A non-CM curve over **Q** and a CM curve over **Q** are both trivially **Q**-curves:

```
sage: from sage.schemes.elliptic_curves.Qcurves import is_Q_curve
sage: E = EllipticCurve([1,2,3,4,5])
sage: flag, cert = is_Q_curve(E, certificate=True)
sage: flag
True
sage: cert
{'CM': 0, 'N': 1, 'core_poly': x, 'r': 0, 'rho': 0}

sage: E = EllipticCurve(j=8000)
sage: flag, cert = is_Q_curve(E, certificate=True)
sage: flag
True
sage: cert
{'CM': -8}
```

A non-$\mathbf{Q}$-curve over a quartic field. The local data at bad primes above 3 is inconsistent:

```
sage: from sage.schemes.elliptic_curves.Qcurves import is_Q_curve
sage: R.<x> = PolynomialRing(QQ)
sage: K.<a> = NumberField(R([3, 0, -5, 0, 1]))                                  #␣
↪needs sage.rings.number_field
sage: E = EllipticCurve([K([-3,-4,1,1]), K([4,-1,-1,0]), K([-2,0,1,0]),         #␣
↪needs sage.rings.number_field
....:                    K([-621,778,138,-178]), K([9509,2046,-24728,10380])])
sage: is_Q_curve(E, certificate=True, verbose=True)                             #␣
↪needs sage.rings.number_field
Checking whether Elliptic Curve defined by y^2 + (a^3+a^2-4*a-3)*x*y + (a^2-2)*y␣
↪= x^3 + (-a^2-a+4)*x^2 + (-178*a^3+138*a^2+778*a-621)*x + (10380*a^3-24728*a^
↪2+2046*a+9509) over Number Field in a with defining polynomial x^4 - 5*x^2 + 3␣
↪is a Q-curve
No: inconsistency at the 2 primes dividing 3
- potentially multiplicative: [True, False]
(False, 3)
```

A non-$\mathbf{Q}$-curve over a quadratic field. The local data at bad primes is consistent, but the local test at good primes above 13 is not:

```
sage: K.<a> = NumberField(R([-10, 0, 1]))                                       #␣
↪needs sage.rings.number_field
sage: E = EllipticCurve([K([0,1]), K([-1,-1]), K([0,0]),                        #␣
↪needs sage.rings.number_field
....:                    K([-236,40]), K([-1840,464])])
sage: is_Q_curve(E, certificate=True, verbose=True)                             #␣
↪needs sage.rings.number_field
Checking whether Elliptic Curve defined by y^2 + a*x*y = x^3 + (-a-1)*x^2 + (40*a-
↪236)*x + (464*a-1840) over Number Field in a with defining polynomial x^2 - 10␣
↪is a Q-curve
Applying local tests at good primes above p<=100
No: inconsistency at the 2 ordinary primes dividing 13
- Frobenius discriminants mod squares: [-1, -3]
No: local test at p=13 failed
(False, 13)
```

A quadratic $\mathbf{Q}$-curve with CM discriminant -15 (j-invariant not in $\mathbf{Q}$):

```
sage: from sage.schemes.elliptic_curves.Qcurves import is_Q_curve
sage: R.<x> = PolynomialRing(QQ)
sage: K.<a> = NumberField(R([-1, -1, 1]))                                       #␣
↪needs sage.rings.number_field
sage: E = EllipticCurve([K([1,0]), K([-1,0]), K([0,1]), K([0,-2]), K([0,1])])   #␣
↪needs sage.rings.number_field
sage: is_Q_curve(E, certificate=True, verbose=True)                             #␣
↪needs sage.rings.number_field
Checking whether Elliptic Curve defined by y^2 + x*y + a*y = x^3 + (-1)*x^2 + (-
↪2*a)*x + a over Number Field in a with defining polynomial x^2 - x - 1 is a Q-
↪curve
Yes: E is CM (discriminant -15)
(True, {'CM': -15})
```

An example over $\mathbf{Q}(\sqrt{2}, \sqrt{3})$. The j-invariant is in $\mathbf{Q}(\sqrt{6})$, so computations will be done over that field, and in fact there is an isogenous curve with rational j, so we have a so-called rational $\mathbf{Q}$-curve:

```
sage: # needs sage.rings.number_field
sage: K.<a> = NumberField(R([1, 0, -4, 0, 1]))
sage: E = EllipticCurve([K([-2,-4,1,1]), K([0,1,0,0]), K([0,1,0,0]),
....:                    K([-4780,9170,1265,-2463]),
....:                    K([163923,-316598,-43876,84852])])
sage: flag, cert = is_Q_curve(E, certificate=True)
sage: flag
True
sage: cert
{'CM': 0, 'N': 1, 'core_degs': [1], 'core_poly': x - 85184/3, 'r': 0, 'rho': 0}
```

Over the same field, a so-called strict $\mathbf{Q}$-curve which is not isogenous to one with rational j, but whose core field is quadratic. In fact the isogeny class over K consists of 6 curves, four with conjugate quartic j-invariants and 2 with quadratic conjugate j-invariants in $\mathbf{Q}(\sqrt{3})$ (but which are not base-changes from the quadratic subfield):

```
sage: # needs sage.rings.number_field
sage: E = EllipticCurve([K([0,-3,0,1]), K([1,4,0,-1]), K([0,0,0,0]),
....:                    K([-2,-16,0,4]), K([-19,-32,4,8])])
sage: flag, cert = is_Q_curve(E, certificate=True)
sage: flag
True
sage: cert
{'CM': 0,
 'N': 2,
 'core_degs': [1, 2],
 'core_poly': x^2 - 840064*x + 1593413632,
 'r': 1,
 'rho': 1}
```

The following relate to elliptic curves over local nonarchimedean fields.

18.13 Local data for elliptic curves over number fields

Let E be an elliptic curve over a number field K (including $\mathbf{Q}$). There are several local invariants at a finite place v that can be computed via Tate's algorithm (see [Sil1994] IV.9.4 or [Tate1975]).

These include the type of reduction (good, additive, multiplicative), a minimal equation of E over K_v, the Tamagawa number c_v, defined to be the index $[E(K_v) : E^0(K_v)]$ of the points with good reduction among the local points, and the exponent of the conductor f_v.

The functions in this file will typically be called by using `local_data`.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<i> = NumberField(x^2 + 1)
sage: E = EllipticCurve([(2+i)^2, (2+i)^7])
sage: pp = K.fractional_ideal(2+i)
sage: da = E.local_data(pp)
sage: da.has_bad_reduction()
True
sage: da.has_multiplicative_reduction()
False
sage: da.kodaira_symbol()
I0*
```

(continues on next page)

(continued from previous page)

```
sage: da.tamagawa_number()
4
sage: da.minimal_model()
Elliptic Curve defined by y^2 = x^3 + (4*i+3)*x + (-29*i-278)
 over Number Field in i with defining polynomial x^2 + 1
```

An example to show how the Neron model can change as one extends the field:

```
sage: E = EllipticCurve([0,-1])
sage: E.local_data(2)
Local data at Principal ideal (2) of Integer Ring:
  Reduction type: bad additive
  Local minimal model: Elliptic Curve defined by y^2 = x^3 - 1 over Rational Field
  Minimal discriminant valuation: 4
  Conductor exponent: 4
  Kodaira Symbol: II
  Tamagawa Number: 1

sage: # needs sage.rings.number_field
sage: EK = E.base_extend(K)
sage: EK.local_data(1+i)
Local data at Fractional ideal (i + 1):
  Reduction type: bad additive
  Local minimal model: Elliptic Curve defined by y^2 = x^3 + (-1)
                       over Number Field in i with defining polynomial x^2 + 1
  Minimal discriminant valuation: 8
  Conductor exponent: 2
  Kodaira Symbol: IV*
  Tamagawa Number: 3
```

Or how the minimal equation changes:

```
sage: E = EllipticCurve([0,8])
sage: E.is_minimal()
True

sage: # needs sage.rings.number_field
sage: EK = E.base_extend(K)
sage: da = EK.local_data(1+i)
sage: da.minimal_model()
Elliptic Curve defined by y^2 = x^3 + (-i)
 over Number Field in i with defining polynomial x^2 + 1
```

AUTHORS:

- John Cremona: First version 2008-09-21 (refactoring code from `ell_number_field.py` and `ell_rational_field.py`)
- Chris Wuthrich: more documentation 2010-01

class sage.schemes.elliptic_curves.ell_local_data.**EllipticCurveLocalData**(*E*, *P*, *proof=None*, *algorithm='pari'*, *globally=False*)

Bases: `SageObject`

The class for the local reduction data of an elliptic curve.

Currently supported are elliptic curves defined over **Q**, and elliptic curves defined over a number field, at an arbitrary prime or prime ideal.

INPUT:

- `E` – an elliptic curve defined over a number field, or **Q**.
- `P` – a prime ideal of the field, or a prime integer if the field is **Q**.
- `proof` (bool) – if `True`, only use provably correct methods (default controlled by global proof module). Note that the proof module is number_field, not elliptic_curves, since the functions that actually need the flag are in number fields.
- `algorithm` (string, default: "pari") – Ignored unless the base field is **Q**. If "pari", use the PARI C-library `ellglobalred` implementation of Tate's algorithm over **Q**. If "generic", use the general number field implementation.

Note: This function is not normally called directly by users, who may access the data via methods of the EllipticCurve classes.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_local_data import␣
↪EllipticCurveLocalData
sage: E = EllipticCurve('14a1')
sage: EllipticCurveLocalData(E,2)
Local data at Principal ideal (2) of Integer Ring:
  Reduction type: bad non-split multiplicative
  Local minimal model: Elliptic Curve defined by y^2 + x*y + y = x^3 + 4*x - 6
                       over Rational Field
  Minimal discriminant valuation: 6
  Conductor exponent: 1
  Kodaira Symbol: I6
  Tamagawa Number: 2
```

bad_reduction_type()

Return the type of bad reduction of this reduction data.

OUTPUT:

(int or `None`):

- +1 for split multiplicative reduction
- -1 for non-split multiplicative reduction
- 0 for additive reduction
- `None` for good reduction

EXAMPLES:

```
sage: E = EllipticCurve('14a1')
sage: [(p,E.local_data(p).bad_reduction_type()) for p in prime_range(15)]
[(2, -1), (3, None), (5, None), (7, 1), (11, None), (13, None)]

sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
```

(continues on next page)

(continued from previous page)

```
sage: K.<a> = NumberField(x^3 - 2)
sage: P17a, P17b = [P for P,e in K.factor(17)]
sage: E = EllipticCurve([0, 0, 0, 0, 2*a+1])
sage: [(p,E.local_data(p).bad_reduction_type()) for p in [P17a,P17b]]
[(Fractional ideal (4*a^2 - 2*a + 1), None), (Fractional ideal (2*a + 1), 0)]
```

conductor_valuation()

Return the valuation of the conductor from this local reduction data.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_local_data import␣
↪EllipticCurveLocalData
sage: E = EllipticCurve([0,0,0,0,64]); E
Elliptic Curve defined by y^2 = x^3 + 64 over Rational Field
sage: data = EllipticCurveLocalData(E,2)
sage: data.conductor_valuation()
2
```

discriminant_valuation()

Return the valuation of the minimal discriminant from this local reduction data.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_local_data import␣
↪EllipticCurveLocalData
sage: E = EllipticCurve([0,0,0,0,64]); E
Elliptic Curve defined by y^2 = x^3 + 64 over Rational Field
sage: data = EllipticCurveLocalData(E,2)
sage: data.discriminant_valuation()
4
```

has_additive_reduction()

Return `True` if there is additive reduction.

EXAMPLES:

```
sage: E = EllipticCurve('27a1')
sage: [(p, E.local_data(p).has_additive_reduction()) for p in prime_range(15)]
[(2, False), (3, True), (5, False), (7, False), (11, False), (13, False)]
```

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)
sage: P17a, P17b = [P for P,e in K.factor(17)]
sage: E = EllipticCurve([0, 0, 0, 0, 2*a+1])
sage: [(p, E.local_data(p).has_additive_reduction()) for p in [P17a,P17b]]
[(Fractional ideal (4*a^2 - 2*a + 1), False),
 (Fractional ideal (2*a + 1), True)]
```

has_bad_reduction()

Return `True` if there is bad reduction.

EXAMPLES:

```
sage: E = EllipticCurve('14a1')
sage: [(p,E.local_data(p).has_bad_reduction()) for p in prime_range(15)]
[(2, True), (3, False), (5, False), (7, True), (11, False), (13, False)]
```

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)
sage: P17a, P17b = [P for P,e in K.factor(17)]
sage: E = EllipticCurve([0, 0, 0, 0, 2*a+1])
sage: [(p,E.local_data(p).has_bad_reduction()) for p in [P17a,P17b]]
[(Fractional ideal (4*a^2 - 2*a + 1), False),
 (Fractional ideal (2*a + 1), True)]
```

has_good_reduction()

Return `True` if there is good reduction.

EXAMPLES:

```
sage: E = EllipticCurve('14a1')
sage: [(p,E.local_data(p).has_good_reduction()) for p in prime_range(15)]
[(2, False), (3, True), (5, True), (7, False), (11, True), (13, True)]

sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)
sage: P17a, P17b = [P for P,e in K.factor(17)]
sage: E = EllipticCurve([0, 0, 0, 0, 2*a+1])
sage: [(p,E.local_data(p).has_good_reduction()) for p in [P17a,P17b]]
[(Fractional ideal (4*a^2 - 2*a + 1), True),
 (Fractional ideal (2*a + 1), False)]
```

has_multiplicative_reduction()

Return `True` if there is multiplicative reduction.

Note: See also `has_split_multiplicative_reduction()` and `has_nonsplit_multiplicative_reduction()`.

EXAMPLES:

```
sage: E = EllipticCurve('14a1')
sage: [(p, E.local_data(p).has_multiplicative_reduction()) for p in prime_
↪range(15)]
[(2, True), (3, False), (5, False), (7, True), (11, False), (13, False)]
```

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)
sage: P17a, P17b = [P for P,e in K.factor(17)]
sage: E = EllipticCurve([0, 0, 0, 0, 2*a+1])
sage: [(p,E.local_data(p).has_multiplicative_reduction()) for p in [P17a,
↪P17b]]
[(Fractional ideal (4*a^2 - 2*a + 1), False), (Fractional ideal (2*a + 1),␣
↪False)]
```

has_nonsplit_multiplicative_reduction()

Return `True` if there is non-split multiplicative reduction.

EXAMPLES:

```
sage: E = EllipticCurve('14a1')
sage: [(p, E.local_data(p).has_nonsplit_multiplicative_reduction())
....:  for p in prime_range(15)]
[(2, True), (3, False), (5, False), (7, False), (11, False), (13, False)]
```

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)
sage: P17a, P17b = [P for P,e in K.factor(17)]
sage: E = EllipticCurve([0, 0, 0, 0, 2*a+1])
sage: [(p, E.local_data(p).has_nonsplit_multiplicative_reduction())
....:  for p in [P17a,P17b]]
[(Fractional ideal (4*a^2 - 2*a + 1), False), (Fractional ideal (2*a + 1),␣
↪False)]
```

has_split_multiplicative_reduction()

Return `True` if there is split multiplicative reduction.

EXAMPLES:

```
sage: E = EllipticCurve('14a1')
sage: [(p, E.local_data(p).has_split_multiplicative_reduction())
....:  for p in prime_range(15)]
[(2, False), (3, False), (5, False), (7, True), (11, False), (13, False)]
```

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)
sage: P17a, P17b = [P for P,e in K.factor(17)]
sage: E = EllipticCurve([0, 0, 0, 0, 2*a+1])
sage: [(p,E .local_data(p).has_split_multiplicative_reduction())
....:  for p in [P17a,P17b]]
[(Fractional ideal (4*a^2 - 2*a + 1), False),
 (Fractional ideal (2*a + 1), False)]
```

kodaira_symbol()

Return the Kodaira symbol from this local reduction data.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_local_data import␣
↪EllipticCurveLocalData
sage: E = EllipticCurve([0,0,0,0,64]); E
Elliptic Curve defined by y^2 = x^3 + 64 over Rational Field
sage: data = EllipticCurveLocalData(E,2)
sage: data.kodaira_symbol()
IV
```

minimal_model(*reduce=True*)

Return the (local) minimal model from this local reduction data.

INPUT:

- `reduce` – (default: `True`) if set to `True` and if the initial elliptic curve had globally integral coefficients, then the elliptic curve returned by Tate's algorithm will be "reduced" as specified in _reduce_model() for curves over number fields.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_local_data import␣
↪EllipticCurveLocalData
sage: E = EllipticCurve([0,0,0,0,64]); E
Elliptic Curve defined by y^2 = x^3 + 64 over Rational Field
sage: data = EllipticCurveLocalData(E, 2)
sage: data.minimal_model()
Elliptic Curve defined by y^2 = x^3 + 1 over Rational Field
sage: data.minimal_model() == E.local_minimal_model(2)
True
```

To demonstrate the behaviour of the parameter `reduce`:

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 + x + 1)
sage: E = EllipticCurve(K, [0, 0, a, 0, 1])
sage: E.local_data(K.ideal(a-1)).minimal_model()
Elliptic Curve defined by y^2 + a*y = x^3 + 1
 over Number Field in a with defining polynomial x^3 + x + 1
sage: E.local_data(K.ideal(a-1)).minimal_model(reduce=False)
Elliptic Curve defined by y^2 + (a+2)*y = x^3 + 3*x^2 + 3*x + (-a+1)
 over Number Field in a with defining polynomial x^3 + x + 1

sage: E = EllipticCurve([2, 1, 0, -2, -1])
sage: E.local_data(ZZ.ideal(2), algorithm="generic").minimal_
↪model(reduce=False)
Elliptic Curve defined by y^2 + 2*x*y + 2*y = x^3 + x^2 - 4*x - 2 over␣
↪Rational Field
sage: E.local_data(ZZ.ideal(2), algorithm="pari").minimal_model(reduce=False)
Traceback (most recent call last):
...
ValueError: the argument reduce must not be False if algorithm=pari is used
sage: E.local_data(ZZ.ideal(2), algorithm="generic").minimal_model()
Elliptic Curve defined by y^2 = x^3 - x^2 - 3*x + 2 over Rational Field
sage: E.local_data(ZZ.ideal(2), algorithm="pari").minimal_model()
Elliptic Curve defined by y^2 = x^3 - x^2 - 3*x + 2 over Rational Field
```

Issue #14476:

```
sage: # needs sage.rings.number_field
sage: t = QQ['t'].0
sage: K.<g> = NumberField(t^4 - t^3-3*t^2 - t +1)
sage: E = EllipticCurve([-2*g^3 + 10/3*g^2 + 3*g - 2/3,
....:                    -11/9*g^3 + 34/9*g^2 - 7/3*g + 4/9,
....:                    -11/9*g^3 + 34/9*g^2 - 7/3*g + 4/9, 0, 0])
sage: vv = K.fractional_ideal(g^2 - g - 2)
sage: E.local_data(vv).minimal_model()
Elliptic Curve defined by
 y^2 + (-2*g^3+10/3*g^2+3*g-2/3)*x*y + (-11/9*g^3+34/9*g^2-7/3*g+4/9)*y
  = x^3 + (-11/9*g^3+34/9*g^2-7/3*g+4/9)*x^2
 over Number Field in g with defining polynomial t^4 - t^3 - 3*t^2 - t + 1
```

prime()

Return the prime ideal associated with this local reduction data.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_local_data import␣
↪EllipticCurveLocalData
sage: E = EllipticCurve([0,0,0,0,64]); E
Elliptic Curve defined by y^2 = x^3 + 64 over Rational Field
sage: data = EllipticCurveLocalData(E,2)
sage: data.prime()
Principal ideal (2) of Integer Ring
```

tamagawa_exponent()

Return the Tamagawa index from this local reduction data.

This is the exponent of $E(K_v)/E^0(K_v)$; in most cases it is the same as the Tamagawa index.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_local_data import␣
↪EllipticCurveLocalData
sage: E = EllipticCurve('816a1')
sage: data = EllipticCurveLocalData(E, 2)
sage: data.kodaira_symbol()
I2*
sage: data.tamagawa_number()
4
sage: data.tamagawa_exponent()
2

sage: E = EllipticCurve('200c4')
sage: data = EllipticCurveLocalData(E, 5)
sage: data.kodaira_symbol()
I4*
sage: data.tamagawa_number()
4
sage: data.tamagawa_exponent()
2
```

tamagawa_number()

Return the Tamagawa number from this local reduction data.

This is the index $[E(K_v) : E^0(K_v)]$.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_local_data import␣
↪EllipticCurveLocalData
sage: E = EllipticCurve([0,0,0,0,64]); E
Elliptic Curve defined by y^2 = x^3 + 64 over Rational Field
sage: data = EllipticCurveLocalData(E,2)
sage: data.tamagawa_number()
3
```

sage.schemes.elliptic_curves.ell_local_data.**check_prime**(K, P)

Function to check that P determines a prime of K, and return that ideal.

INPUT:

- K – a number field (including **Q**).
- P – an element of K or a (fractional) ideal of K.

OUTPUT:

- If K is **Q**: the prime integer equal to or which generates P.
- If K is not **Q**: the prime ideal equal to or generated by P.

Note: If P is not a prime and does not generate a prime, a TypeError is raised.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_local_data import check_prime
sage: check_prime(QQ, 3)
3
sage: check_prime(QQ, QQ(3))
3
sage: check_prime(QQ, ZZ.ideal(31))
31

sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^2 - 5)
sage: check_prime(K, a)
Fractional ideal (a)
sage: check_prime(K, a + 1)
Fractional ideal (a + 1)
sage: [check_prime(K, P) for P in K.primes_above(31)]
[Fractional ideal (5/2*a + 1/2), Fractional ideal (5/2*a - 1/2)]
sage: L.<b> = NumberField(x^2 + 3)
sage: check_prime(K, L.ideal(5))
Traceback (most recent call last):
...
TypeError: The ideal Fractional ideal (5) is not a prime ideal of
Number Field in a with defining polynomial x^2 - 5
sage: check_prime(K, L.ideal(b))
Traceback (most recent call last):
...
TypeError: No compatible natural embeddings found for
Number Field in a with defining polynomial x^2 - 5 and
Number Field in b with defining polynomial x^2 + 3
```

18.14 Kodaira symbols

Kodaira symbols encode the type of reduction of an elliptic curve at a (finite) place.

The standard notation for Kodaira Symbols is as a string which is one of I_m, II, III, IV, I_m^*, II*, III*, IV*, where m denotes a non-negative integer. These have been encoded by single integers by different people. For convenience we give here the conversion table between strings, the eclib coding and the PARI encoding.

Kodaira Symbol	Eclib coding	PARI Coding
I_0	0	1
I_0^*	1	-1
I_m $(m > 0)$	$10m$	$m + 4$
I_m^* $(m > 0)$	$10m + 1$	$-(m + 4)$
II	2	2
III	3	3
IV	4	4
II^*	7	-2
III^*	6	-3
IV^*	5	-4

AUTHORS:

- David Roe <roed@math.harvard.edu>
- John Cremona

sage.schemes.elliptic_curves.kodaira_symbol.**KodairaSymbol**(*symbol*)

Return the specified Kodaira symbol.

INPUT:

- `symbol` (string or integer) – Either a string of the form "I0", "I1", …, "In", "II", "III", "IV", "I0*", "I1*", …, "In*", "II*", "III*", or "IV*", or an integer encoding a Kodaira symbol using PARI's conventions.

OUTPUT:

(KodairaSymbol) The corresponding Kodaira symbol.

EXAMPLES:

```
sage: KS = KodairaSymbol
sage: [KS(n) for n in range(1,10)]
[I0, II, III, IV, I1, I2, I3, I4, I5]
sage: [KS(-n) for n in range(1,10)]
[I0*, II*, III*, IV*, I1*, I2*, I3*, I4*, I5*]
sage: all(KS(str(KS(n))) == KS(n) for n in range(-10,10) if n != 0)
True
```

class sage.schemes.elliptic_curves.kodaira_symbol.**KodairaSymbol_class**(*symbol*)

Bases: `SageObject`

Class to hold a Kodaira symbol of an elliptic curve over a p-adic local field.

Users should use the `KodairaSymbol()` function to construct Kodaira Symbols rather than use the class constructor directly.

18.15 Tate's parametrisation of p-adic curves with multiplicative reduction

Let E be an elliptic curve defined over the p-adic numbers $\mathbf{Q}_p$. Suppose that E has multiplicative reduction, i.e. that the j-invariant of E has negative valuation, say n. Then there exists a parameter q in $\mathbf{Z}_p$ of valuation n such that the points of E defined over the algebraic closure $\bar{\mathbf{Q}}_p$ are in bijection with $\bar{\mathbf{Q}}_p^\times / q^{\mathbf{Z}}$. More precisely there exists the series $s_4(q)$ and $s_6(q)$ such that the $y^2 + xy = x^3 + s_4(q)x + s_6(q)$ curve is isomorphic to E over $\bar{\mathbf{Q}}_p$ (or over $\mathbf{Q}_p$ if the reduction is *split* multiplicative). There is a p-adic analytic map from $\bar{\mathbf{Q}}_p^\times$ to this curve with kernel $q^{\mathbf{Z}}$. Points of good reduction correspond to points of valuation 0 in $\bar{\mathbf{Q}}_p^\times$.

See chapter V of [Sil1994] for more details.

AUTHORS:

- Chris Wuthrich (23/05/2007): first version
- William Stein (2007-05-29): added some examples; editing.
- Chris Wuthrich (04/09): reformatted docstrings.

class sage.schemes.elliptic_curves.ell_tate_curve.**TateCurve**(*E*, *p*)

Bases: SageObject

Tate's p-adic uniformisation of an elliptic curve with multiplicative reduction.

Note: Some of the methods of this Tate curve only work when the reduction is split multiplicative over $\mathbf{Q}_p$.

EXAMPLES:

```
sage: e = EllipticCurve('130a1')
sage: eq = e.tate_curve(5); eq
5-adic Tate curve associated to the Elliptic Curve
 defined by y^2 + x*y + y = x^3 - 33*x + 68 over Rational Field
sage: eq == loads(dumps(eq))
True
```

REFERENCES: [Sil1994]

E2(*prec=20*)

Return the value of the p-adic Eisenstein series of weight 2 evaluated on the elliptic curve having split multiplicative reduction.

INPUT:

- `prec` – the p-adic precision, default is 20.

EXAMPLES:

```
sage: eq = EllipticCurve('130a1').tate_curve(5)
sage: eq.E2(prec=10)
4 + 2*5^2 + 2*5^3 + 5^4 + 2*5^5 + 5^7 + 5^8 + 2*5^9 + O(5^10)

sage: T = EllipticCurve('14').tate_curve(7)
sage: T.E2(30)
2 + 4*7 + 7^2 + 3*7^3 + 6*7^4 + 5*7^5 + 2*7^6 + 7^7 + 5*7^8 + 6*7^9 + 5*7^10␣
↪+ 2*7^11 + 6*7^12 + 4*7^13 + 3*7^15 + 5*7^16 + 4*7^17 + 4*7^18 + 2*7^20 + 7^
↪21 + 5*7^22 + 4*7^23 + 4*7^24 + 3*7^25 + 6*7^26 + 3*7^27 + 6*7^28 + O(7^30)
```

L_invariant (*prec=20*)

Return the *mysterious* $\mathcal{L}$-invariant associated to an elliptic curve with split multiplicative reduction.

One instance where this constant appears is in the exceptional case of the p-adic Birch and Swinnerton-Dyer conjecture as formulated in [MTT1986]. See [Col2004] for a detailed discussion.

INPUT:

- `prec` – the p-adic precision, default is 20.

EXAMPLES:

```
sage: eq = EllipticCurve('130a1').tate_curve(5)
sage: eq.L_invariant(prec=10)
5^3 + 4*5^4 + 2*5^5 + 2*5^6 + 2*5^7 + 3*5^8 + 5^9 + O(5^10)
```

curve (*prec=20*)

Return the p-adic elliptic curve of the form $y^2 + xy = x^3 + s_4x + s_6$.

This curve with split multiplicative reduction is isomorphic to the given curve over the algebraic closure of $\mathbf{Q}_p$.

INPUT:

- `prec` – the p-adic precision, default is 20.

EXAMPLES:

```
sage: eq = EllipticCurve('130a1').tate_curve(5)
sage: eq.curve(prec=5)
Elliptic Curve defined by y^2 + (1+O(5^5))*x*y =
 x^3 + (2*5^4+5^5+2*5^6+5^7+3*5^8+O(5^9))*x + (2*5^3+5^4+2*5^5+5^7+O(5^8))
 over 5-adic Field with capped relative precision 5
```

is_split ()

Return `True` if the given elliptic curve has split multiplicative reduction.

EXAMPLES:

```
sage: eq = EllipticCurve('130a1').tate_curve(5)
sage: eq.is_split()
True

sage: eq = EllipticCurve('37a1').tate_curve(37)
sage: eq.is_split()
False
```

lift (*P*, *prec=20*)

Given a point P in the formal group of the elliptic curve E with split multiplicative reduction, this produces an element u in $\mathbf{Q}_p^\times$ mapped to the point P by the Tate parametrisation. The algorithm return the unique such element in $1 + p\mathbf{Z}_p$.

INPUT:

- `P` – a point on the elliptic curve.
- `prec` – the p-adic precision, default is 20.

EXAMPLES:

```
sage: e = EllipticCurve('130a1')
sage: eq = e.tate_curve(5)
sage: P = e([-6,10])
sage: l = eq.lift(12*P, prec=10); l
1 + 4*5 + 5^3 + 5^4 + 4*5^5 + 5^6 + 5^7 + 4*5^8 + 5^9 + O(5^10)
```

Now we map the lift l back and check that it is indeed right.:

```
sage: eq.parametrisation_onto_original_curve(l)
(4*5^-2 + 2*5^-1 + 4*5 + 3*5^3 + 5^4 + 2*5^5 + 4*5^6 + O(5^7)
 : 2*5^-3 + 5^-1 + 4 + 4*5 + 5^2 + 3*5^3 + 4*5^4 + O(5^6) : 1 + O(5^10))
sage: e5 = e.change_ring(Qp(5,9))
sage: e5(12*P)
(4*5^-2 + 2*5^-1 + 4*5 + 3*5^3 + 5^4 + 2*5^5 + 4*5^6 + O(5^7)
 : 2*5^-3 + 5^-1 + 4 + 4*5 + 5^2 + 3*5^3 + 4*5^4 + O(5^6) : 1 + O(5^9))
```

original_curve()

Return the elliptic curve the Tate curve was constructed from.

EXAMPLES:

```
sage: eq = EllipticCurve('130a1').tate_curve(5)
sage: eq.original_curve()
Elliptic Curve defined by y^2 + x*y + y = x^3 - 33*x + 68
 over Rational Field
```

padic_height(*prec=20*)

Return the canonical p-adic height function on the original curve.

INPUT:

- `prec` – the p-adic precision, default is 20.

OUTPUT:

- A function that can be evaluated on rational points of E.

EXAMPLES:

```
sage: e = EllipticCurve('130a1')
sage: eq = e.tate_curve(5)
sage: h = eq.padic_height(prec=10)
sage: P = e.gens()[0]
sage: h(P)
2*5^-1 + 1 + 2*5 + 2*5^2 + 3*5^3 + 3*5^6 + 5^7 + O(5^9)
```

Check that it is a quadratic function:

```
sage: h(3*P)-3^2*h(P)
O(5^9)
```

padic_regulator(*prec=20*)

Compute the canonical p-adic regulator on the extended Mordell-Weil group as in [MTT1986] (with the correction of [Wer1998] and sign convention in [SW2013].)

The p-adic Birch and Swinnerton-Dyer conjecture predicts that this value appears in the formula for the leading term of the p-adic L-function.

INPUT:

- `prec` – the p-adic precision, default is 20.

EXAMPLES:

```
sage: eq = EllipticCurve('130a1').tate_curve(5)
sage: eq.padic_regulator()
2*5^-1 + 1 + 2*5 + 2*5^2 + 3*5^3 + 3*5^6 + 5^7 + 3*5^9 + 3*5^10 + 3*5^12 +␣
↪4*5^13 + 3*5^15 + 2*5^16 + 3*5^18 + 4*5^19 +  4*5^20 + 3*5^21 + 4*5^22 +␣
↪O(5^23)
```

parameter(*prec=20*)

Return the Tate parameter q such that the curve is isomorphic over the algebraic closure of $\mathbf{Q}_p$ to the curve $\mathbf{Q}_p^\times/q^{\mathbf{Z}}$.

INPUT:

- `prec` – the p-adic precision, default is 20.

EXAMPLES:

```
sage: eq = EllipticCurve('130a1').tate_curve(5)
sage: eq.parameter(prec=5)
3*5^3 + 3*5^4 + 2*5^5 + 2*5^6 + 3*5^7 + O(5^8)
```

parametrisation_onto_original_curve(*u*, *prec=None*)

Given an element u in $\mathbf{Q}_p^\times$, this computes its image on the original curve under the p-adic uniformisation of E.

INPUT:

- `u` – a non-zero p-adic number.
- `prec` – the p-adic precision, default is the relative precision of `u` otherwise 20.

EXAMPLES:

```
sage: eq = EllipticCurve('130a1').tate_curve(5)
sage: eq.parametrisation_onto_original_curve(1+5+5^2+O(5^10))
(4*5^-2 + 4*5^-1 + 4 + 2*5^3 + 3*5^4 + 2*5^6 + O(5^7) :
 3*5^-3 + 5^-2 + 4*5^-1 + 1 + 4*5 + 5^2 + 3*5^5 + O(5^6) :
 1 + O(5^10))
sage: eq.parametrisation_onto_original_curve(1+5+5^2+O(5^10), prec=20)
Traceback (most recent call last):
...
ValueError: requested more precision than the precision of u
```

Here is how one gets a 4-torsion point on E over $\mathbf{Q}_5$:

```
sage: R = Qp(5,30)
sage: i = R(-1).sqrt()
sage: T = eq.parametrisation_onto_original_curve(i, prec=30); T
(2 + 3*5 + 4*5^2 + 2*5^3 + 5^4 + 4*5^5 + 2*5^7 + 5^8 + 5^9 + 5^12 + 3*5^13 +␣
↪3*5^14 + 5^15 + 4*5^17 + 5^18 + 3*5^19 + 2*5^20 + 4*5^21 + 5^22 + 3*5^23 +␣
↪3*5^24 + 4*5^25 + 3*5^26 + 3*5^27 + 3*5^28 + 3*5^29 + O(5^30) : 3*5 + 5^2 +␣
↪5^4 + 3*5^5 + 3*5^7 + 2*5^8 + 4*5^9 + 5^10 + 2*5^11 + 4*5^13 + 2*5^14 + 4*5^
↪15 + 4*5^16 + 3*5^17 + 2*5^18 + 4*5^20 + 2*5^21 + 2*5^22 + 4*5^23 + 4*5^24␣
↪+ 4*5^25 + 5^26 + 3*5^27 + 2*5^28 + O(5^30) : 1 + O(5^30))
sage: 4*T
(0 : 1 + O(5^30) : 0)
```

parametrisation_onto_tate_curve(*u*, *prec=None*)

Given an element u in $\mathbf{Q}_p^\times$, this computes its image on the Tate curve under the p-adic uniformisation of E.

INPUT:

- `u` – a non-zero p-adic number.
- `prec` – the p-adic precision, default is the relative precision of `u` otherwise 20.

EXAMPLES:

```
sage: eq = EllipticCurve('130a1').tate_curve(5)
sage: eq.parametrisation_onto_tate_curve(1+5+5^2+O(5^10), prec=10)
(5^-2 + 4*5^-1 + 1 + 2*5 + 3*5^2 + 2*5^5 + 3*5^6 + O(5^7)
 : 4*5^-3 + 2*5^-1 + 4 + 2*5 + 3*5^4 + 2*5^5 + O(5^6) : 1 + O(5^10))
sage: eq.parametrisation_onto_tate_curve(1+5+5^2+O(5^10))
(5^-2 + 4*5^-1 + 1 + 2*5 + 3*5^2 + 2*5^5 + 3*5^6 + O(5^7)
 : 4*5^-3 + 2*5^-1 + 4 + 2*5 + 3*5^4 + 2*5^5 + O(5^6) : 1 + O(5^10))
sage: eq.parametrisation_onto_tate_curve(1+5+5^2+O(5^10), prec=20)
Traceback (most recent call last):
...
ValueError: requested more precision than the precision of u
```

prime()

Return the residual characteristic p.

EXAMPLES:

```
sage: eq = EllipticCurve('130a1').tate_curve(5)
sage: eq.original_curve()
Elliptic Curve defined by y^2 + x*y + y = x^3 - 33*x + 68
 over Rational Field
sage: eq.prime()
5
```

Analytic properties over $\mathbf{C}$.

18.16 Weierstrass $\wp$-function for elliptic curves

The Weierstrass $\wp$ function associated to an elliptic curve over a field k is a Laurent series of the form

$$\wp(z) = \frac{1}{z^2} + c_2 \cdot z^2 + c_4 \cdot z^4 + \cdots .$$

If the field is contained in $\mathbb{C}$, then this is the series expansion of the map from $\mathbb{C}$ to $E(\mathbb{C})$ whose kernel is the period lattice of E.

Over other fields, like finite fields, this still makes sense as a formal power series with coefficients in k - at least its first $p-2$ coefficients where p is the characteristic of k. It can be defined via the formal group as $x + c$ in the variable $z = \log_E(t)$ for a constant c such that the constant term c_0 in $\wp(z)$ is zero.

EXAMPLES:

```
sage: E = EllipticCurve([0,1])
sage: E.weierstrass_p()
z^-2 - 1/7*z^4 + 1/637*z^10 - 1/84721*z^16 + O(z^20)
```

REFERENCES:

- [BMSS2006]

AUTHORS:

- Dan Shumov 04/09: original implementation
- Chris Wuthrich 11/09: major restructuring
- Jeroen Demeyer (2014-03-06): code clean up, fix characteristic bound for quadratic algorithm (see Issue #15855)

sage.schemes.elliptic_curves.ell_wp.**compute_wp_fast**(*k*, *A*, *B*, *m*)

Computes the Weierstrass function of an elliptic curve defined by short Weierstrass model: $y^2 = x^3 + Ax + B$. It does this with as fast as polynomial of degree m can be multiplied together in the base ring, i.e. $O(M(n))$ in the notation of [BMSS2006].

Let p be the characteristic of the underlying field: Then we must have either $p = 0$, or $p > m + 3$.

INPUT:

- `k` – the base field of the curve
- `A` – and
- `B` – as the coefficients of the short Weierstrass model $y^2 = x^3 + Ax + B$, and
- `m` – the precision to which the function is computed to.

OUTPUT:

the Weierstrass $\wp$ function as a Laurent series to precision m.

ALGORITHM:

This function uses the algorithm described in section 3.3 of [BMSS2006].

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_wp import compute_wp_fast
sage: compute_wp_fast(QQ, 1, 8, 7)
z^-2 - 1/5*z^2 - 8/7*z^4 + 1/75*z^6 + O(z^7)

sage: k = GF(37)
sage: compute_wp_fast(k, k(1), k(8), 5)
z^-2 + 22*z^2 + 20*z^4 + O(z^5)
```

sage.schemes.elliptic_curves.ell_wp.**compute_wp_pari**(*E*, *prec*)

Computes the Weierstrass $\wp$-function with the `ellwp` function from PARI.

EXAMPLES:

```
sage: E = EllipticCurve([0,1])
sage: from sage.schemes.elliptic_curves.ell_wp import compute_wp_pari
sage: compute_wp_pari(E, prec=20)
z^-2 - 1/7*z^4 + 1/637*z^10 - 1/84721*z^16 + O(z^20)
sage: compute_wp_pari(E, prec=30)
z^-2 - 1/7*z^4 + 1/637*z^10 - 1/84721*z^16
+ 3/38548055*z^22 - 4/8364927935*z^28 + O(z^30)
```

sage.schemes.elliptic_curves.ell_wp.**compute_wp_quadratic**(*k*, *A*, *B*, *prec*)

Compute the truncated Weierstrass function of an elliptic curve defined by short Weierstrass model: $y^2 = x^3 + Ax + B$. Uses an algorithm that is of complexity $O(prec^2)$.

Let p be the characteristic of the underlying field. Then we must have either p = 0, or p > prec + 2.

INPUT:

- k – the field of definition of the curve
- A – and
- B – the coefficients of the elliptic curve
- prec – the precision to which we compute the series.

OUTPUT:

A Laurent series approximating the Weierstrass $\wp$-function to precision prec.

ALGORITHM:

This function uses the algorithm described in section 3.2 of [BMSS2006].

EXAMPLES:

```
sage: E = EllipticCurve([7,0])
sage: E.weierstrass_p(prec=10, algorithm='quadratic')
z^-2 - 7/5*z^2 + 49/75*z^6 + O(z^10)

sage: E = EllipticCurve(GF(103), [1,2])
sage: E.weierstrass_p(algorithm='quadratic')
z^-2 + 41*z^2 + 88*z^4 + 11*z^6 + 57*z^8 + 55*z^10 + 73*z^12
 + 11*z^14 + 17*z^16 + 50*z^18 + O(z^20)

sage: from sage.schemes.elliptic_curves.ell_wp import compute_wp_quadratic
sage: compute_wp_quadratic(E.base_ring(), E.a4(), E.a6(), prec=10)
z^-2 + 41*z^2 + 88*z^4 + 11*z^6 + 57*z^8 + O(z^10)
```

sage.schemes.elliptic_curves.ell_wp.**solve_linear_differential_system**(*a*, *b*, *c*, *alpha*)

Solves a system of linear differential equations: $af' + bf = c$ and $f'(0) = \alpha$ where a, b, and c are power series in one variable and α is a constant in the coefficient ring.

ALGORITHM:

due to Brent and Kung '78.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_wp import solve_linear_differential_
↪system
sage: k = GF(17)
sage: R.<x> = PowerSeriesRing(k)
sage: a = 1 + x + O(x^7); b = x + O(x^7); c = 1 + x^3 + O(x^7); alpha = k(3)
sage: f = solve_linear_differential_system(a, b, c, alpha)
sage: f
3 + x + 15*x^2 + x^3 + 10*x^5 + 3*x^6 + 13*x^7 + O(x^8)
sage: a*f.derivative() + b*f - c
O(x^7)
sage: f(0) == alpha
True
```

sage.schemes.elliptic_curves.ell_wp.**weierstrass_p**(*E*, *prec=20*, *algorithm=None*)

Compute the Weierstrass $\wp$-function on an elliptic curve.

INPUT:

- E – an elliptic curve

- `prec` – precision
- `algorithm` – string or `None` (default: `None`): a choice of algorithm among `"pari"`, `"fast"`, `"quadratic"`; or `None` to let this function determine the best algorithm to use.

OUTPUT:

a Laurent series in one variable z with coefficients in the base field k of E.

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: E.weierstrass_p(prec=10)
z^-2 + 31/15*z^2 + 2501/756*z^4 + 961/675*z^6 + 77531/41580*z^8 + O(z^10)
sage: E.weierstrass_p(prec=8)
z^-2 + 31/15*z^2 + 2501/756*z^4 + 961/675*z^6 + O(z^8)
sage: Esh = E.short_weierstrass_model()
sage: Esh.weierstrass_p(prec=8)
z^-2 + 13392/5*z^2 + 1080432/7*z^4 + 59781888/25*z^6 + O(z^8)

sage: E.weierstrass_p(prec=8, algorithm='pari')
z^-2 + 31/15*z^2 + 2501/756*z^4 + 961/675*z^6 + O(z^8)
sage: E.weierstrass_p(prec=8, algorithm='quadratic')
z^-2 + 31/15*z^2 + 2501/756*z^4 + 961/675*z^6 + O(z^8)

sage: k = GF(11)
sage: E = EllipticCurve(k, [1,1])
sage: E.weierstrass_p(prec=6, algorithm='fast')
z^-2 + 2*z^2 + 3*z^4 + O(z^6)
sage: E.weierstrass_p(prec=7, algorithm='fast')
Traceback (most recent call last):
...
ValueError: for computing the Weierstrass p-function via the fast algorithm,
the characteristic (11) of the underlying field must be greater than prec + 4 = 11
sage: E.weierstrass_p(prec=8)
z^-2 + 2*z^2 + 3*z^4 + 5*z^6 + O(z^8)
sage: E.weierstrass_p(prec=8, algorithm='quadratic')
z^-2 + 2*z^2 + 3*z^4 + 5*z^6 + O(z^8)
sage: E.weierstrass_p(prec=8, algorithm='pari')
z^-2 + 2*z^2 + 3*z^4 + 5*z^6 + O(z^8)
sage: E.weierstrass_p(prec=9)
Traceback (most recent call last):
...
NotImplementedError: currently no algorithms for computing the Weierstrass
p-function for that characteristic / precision pair is implemented.
Lower the precision below char(k) - 2
sage: E.weierstrass_p(prec=9, algorithm="quadratic")
Traceback (most recent call last):
...
ValueError: for computing the Weierstrass p-function via the quadratic
algorithm, the characteristic (11) of the underlying field must be greater
than prec + 2 = 11
sage: E.weierstrass_p(prec=9, algorithm='pari')
Traceback (most recent call last):
...
ValueError: for computing the Weierstrass p-function via pari, the
characteristic (11) of the underlying field must be greater than prec + 2 = 11
```

18.17 Period lattices of elliptic curves and related functions

Let E be an elliptic curve defined over a number field K (including $\mathbf{Q}$). We attach a period lattice (a discrete rank 2 subgroup of $\mathbf{C}$) to each embedding of K into $\mathbf{C}$.

In the case of real embeddings, the lattice is stable under complex conjugation and is called a real lattice. These have two types: rectangular, (the real curve has two connected components and positive discriminant) or non-rectangular (one connected component, negative discriminant).

The periods are computed to arbitrary precision using the AGM (Gauss's Arithmetic-Geometric Mean).

EXAMPLES:

```
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)                                              #␣
↪needs sage.rings.number_field
sage: E = EllipticCurve([0,1,0,a,a])                                            #␣
↪needs sage.rings.number_field
```

First we try a real embedding:

```
sage: emb = K.embeddings(RealField())[0]                                        #␣
↪needs sage.rings.number_field
sage: L = E.period_lattice(emb); L                                              #␣
↪needs sage.rings.number_field
Period lattice associated to Elliptic Curve defined by y^2 = x^3 + x^2 + a*x + a
 over Number Field in a with defining polynomial x^3 - 2
 with respect to the embedding Ring morphism:
   From: Number Field in a with defining polynomial x^3 - 2
   To:   Algebraic Real Field
   Defn: a |--> 1.259921049894873?
```

The first basis period is real:

```
sage: L.basis()                                                                 #␣
↪needs sage.rings.number_field
(3.81452977217855, 1.90726488608927 + 1.34047785962440*I)
sage: L.is_real()                                                               #␣
↪needs sage.rings.number_field
True
```

For a basis ω_1, ω_2 normalised so that ω_1/ω_2 is in the fundamental region of the upper half-plane, use the method *normalised_basis()* instead:

```
sage: L.normalised_basis()                                                      #␣
↪needs sage.rings.number_field
(1.90726488608927 - 1.34047785962440*I, -1.90726488608927 - 1.34047785962440*I)
```

Next a complex embedding:

```
sage: emb = K.embeddings(ComplexField())[0]                                     #␣
↪needs sage.rings.number_field
sage: L = E.period_lattice(emb); L                                              #␣
↪needs sage.rings.number_field
Period lattice associated to Elliptic Curve defined by y^2 = x^3 + x^2 + a*x + a
 over Number Field in a with defining polynomial x^3 - 2
 with respect to the embedding Ring morphism:
   From: Number Field in a with defining polynomial x^3 - 2
```

(continues on next page)

(continued from previous page)

```
   To:   Algebraic Field
   Defn: a |--> -0.6299605249474365? - 1.091123635971722?*I
```

In this case, the basis ω_1, ω_2 is always normalised so that $\tau = \omega_1/\omega_2$ is in the fundamental region in the upper half plane:

```
sage: # needs sage.rings.number_field
sage: w1, w2 = L.basis(); w1, w2
(-1.37588604166076 - 2.58560946624443*I, -2.10339907847356 + 0.428378776460622*I)
sage: L.is_real()
False
sage: tau = w1/w2; tau
0.387694505032876 + 1.30821088214407*I
sage: L.normalised_basis()
(-1.37588604166076 - 2.58560946624443*I, -2.10339907847356 + 0.428378776460622*I)
```

We test that bug Issue #8415 (caused by a PARI bug fixed in v2.3.5) is OK:

```
sage: # needs sage.rings.number_field
sage: E = EllipticCurve('37a')
sage: K.<a> = QuadraticField(-7)
sage: EK = E.change_ring(K)
sage: EK.period_lattice(K.complex_embeddings()[0])
Period lattice associated to Elliptic Curve defined by y^2 + y = x^3 + (-1)*x
 over Number Field in a with defining polynomial x^2 + 7
  with a = 2.645751311064591?*I
 with respect to the embedding Ring morphism:
   From: Number Field in a with defining polynomial x^2 + 7
         with a = 2.645751311064591?*I
   To:   Algebraic Field
   Defn: a |--> -2.645751311064591?*I
```

REFERENCES:

- [CT2013]

AUTHORS:

- ?: initial version.
- John Cremona:
  - Adapted to handle real embeddings of number fields, September 2008.
  - Added basis_matrix function, November 2008
  - Added support for complex embeddings, May 2009.
  - Added complex elliptic logs, March 2010; enhanced, October 2010.

class sage.schemes.elliptic_curves.period_lattice.**PeriodLattice**(*base_ring*, *rank*, *degree*, *sparse=False*, *coordinate_ring=None*, *category=None*)

Bases: FreeModule_generic_pid

The class for the period lattice of an algebraic variety.

class sage.schemes.elliptic_curves.period_lattice.**PeriodLattice_ell**(*E*, *embedding=None*)

Bases: *PeriodLattice*

The class for the period lattice of an elliptic curve.

Currently supported are elliptic curves defined over $\mathbf{Q}$, and elliptic curves defined over a number field with a real or complex embedding, where the lattice constructed depends on that embedding.

basis (*prec=None*, *algorithm='sage'*)

Return a basis for this period lattice as a 2-tuple.

INPUT:

- `prec` (default: `None`) – precision in bits (default precision if `None`).
- `algorithm` (string, default `'sage'`) – choice of implementation (for real embeddings only) between `'sage'` (native Sage implementation) or `'pari'` (use the PARI library: only available for real embeddings).

OUTPUT:

(tuple of Complex) (ω_1, ω_2) where the lattice is $\mathbf{Z}\omega_1 + \mathbf{Z}\omega_2$. If the lattice is real then ω_1 is real and positive, $\Im(\omega_2) > 0$ and $\Re(\omega_1/\omega_2)$ is either 0 (for rectangular lattices) or $\frac{1}{2}$ (for non-rectangular lattices). Otherwise, ω_1/ω_2 is in the fundamental region of the upper half-plane. If the latter normalisation is required for real lattices, use the method *normalised_basis()* instead.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.period_lattice().basis()
(2.99345864623196, 2.45138938198679*I)
```

This shows that the issue reported at Issue #3954 is fixed:

```
sage: E = EllipticCurve('37a')
sage: b1 = E.period_lattice().basis(prec=30)
sage: b2 = E.period_lattice().basis(prec=30)
sage: b1 == b2
True
```

This shows that the issue reported at Issue #4064 is fixed:

```
sage: E = EllipticCurve('37a')
sage: E.period_lattice().basis(prec=30)[0].parent()
Real Field with 30 bits of precision
sage: E.period_lattice().basis(prec=100)[0].parent()
Real Field with 100 bits of precision
```

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)
sage: emb = K.embeddings(RealField())[0]
sage: E = EllipticCurve([0,1,0,a,a])
sage: L = E.period_lattice(emb)
sage: L.basis(64)
(3.81452977217854509, 1.90726488608927255 + 1.34047785962440202*I)

sage: # needs sage.rings.number_field
sage: emb = K.embeddings(ComplexField())[0]
sage: L = E.period_lattice(emb)
```

(continues on next page)

(continued from previous page)

```
sage: w1, w2 = L.basis(); w1, w2
(-1.37588604166076 - 2.58560946624443*I, -2.10339907847356 + 0.
↪428378776460622*I)
sage: L.is_real()
False
sage: tau = w1/w2; tau
0.387694505032876 + 1.30821088214407*I
```

basis_matrix(*prec=None*, *normalised=False*)

Return the basis matrix of this period lattice.

INPUT:

- `prec` (int or `None` (default)) – real precision in bits (default real precision if `None`).
- `normalised` (bool, default `False`) – if `True` and the embedding is real, use the normalised basis (see *normalised_basis()*) instead of the default.

OUTPUT:

A 2×2 real matrix whose rows are the lattice basis vectors, after identifying $\mathbf{C}$ with $\mathbf{R}^2$.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.period_lattice().basis_matrix()
[ 2.99345864623196 0.000000000000000]
[0.000000000000000  2.45138938198679]
```

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)
sage: emb = K.embeddings(RealField())[0]
sage: E = EllipticCurve([0,1,0,a,a])
sage: L = E.period_lattice(emb)
sage: L.basis_matrix(64)
[ 3.81452977217854509 0.000000000000000000]
[ 1.90726488608927255  1.34047785962440202]
```

See Issue #4388:

```
sage: L = EllipticCurve('11a1').period_lattice()
sage: L.basis_matrix()
[ 1.26920930427955 0.000000000000000]
[0.634604652139777  1.45881661693850]
sage: L.basis_matrix(normalised=True)
[0.634604652139777 -1.45881661693850]
[-1.26920930427955 0.000000000000000]
```

```
sage: L = EllipticCurve('389a1').period_lattice()
sage: L.basis_matrix()
[ 2.49021256085505 0.000000000000000]
[0.000000000000000  1.97173770155165]
sage: L.basis_matrix(normalised=True)
[ 2.49021256085505 0.000000000000000]
[0.000000000000000 -1.97173770155165]
```

complex_area (*prec=None*)

Return the area of a fundamental domain for the period lattice of the elliptic curve.

INPUT:

- `prec` (int or `None` (default)) – real precision in bits (default real precision if `None`).

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.period_lattice().complex_area()
7.33813274078958
```

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)
sage: embs = K.embeddings(ComplexField())
sage: E = EllipticCurve([0,1,0,a,a])
sage: [E.period_lattice(emb).is_real() for emb in K.embeddings(CC)]
[False, False, True]
sage: [E.period_lattice(emb).complex_area() for emb in embs]
[6.02796894766694, 6.02796894766694, 5.11329270448345]
```

coordinates (*z*, *rounding=None*)

Return the coordinates of a complex number w.r.t. the lattice basis

INPUT:

- `z` (complex) – A complex number.
- **`rounding` (default `None`) – whether and how to round the** output (see below).

OUTPUT:

When `rounding` is `None` (the default), returns a tuple of reals x, y such that $z = xw_1 + yw_2$ where w_1, w_2 are a basis for the lattice (normalised in the case of complex embeddings).

When `rounding` is `'round'`, returns a tuple of integers n_1, n_2 which are the closest integers to the x, y defined above. If z is in the lattice these are the coordinates of z with respect to the lattice basis.

When `rounding` is `'floor'`, returns a tuple of integers n_1, n_2 which are the integer parts to the x, y defined above. These are used in *reduce()*

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: L = E.period_lattice()
sage: w1, w2 = L.basis(prec=100)
sage: P = E([-1,1])
sage: zP = P.elliptic_logarithm(precision=100); zP
0.47934825019021931612953301006 + 0.98586885077582410221120384908*I
sage: L.coordinates(zP)
(0.19249290511394227352563996419, 0.50000000000000000000000000000)
sage: sum([x*w for x, w in zip(L.coordinates(zP), L.basis(prec=100))])
0.47934825019021931612953301006 + 0.98586885077582410221120384908*I

sage: L.coordinates(12*w1 + 23*w2)
(12.000000000000000000000000000, 23.000000000000000000000000000)
sage: L.coordinates(12*w1 + 23*w2, rounding='floor')
```

(continues on next page)

(continued from previous page)

```
(11, 22)
sage: L.coordinates(12*w1 + 23*w2, rounding='round')
(12, 23)
```

curve()

Return the elliptic curve associated with this period lattice.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: L = E.period_lattice()
sage: L.curve() is E
True
```

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)
sage: E = EllipticCurve([0,1,0,a,a])
sage: L = E.period_lattice(K.embeddings(RealField())[0])
sage: L.curve() is E
True

sage: L = E.period_lattice(K.embeddings(ComplexField())[0])                    #␣
↪needs sage.rings.number_field
sage: L.curve() is E                                                           #␣
↪needs sage.rings.number_field
True
```

e_log_RC(*xP*, *yP*, *prec=None*, *reduce=True*)

Return the elliptic logarithm of a real or complex point.

INPUT:

- `xP`, `yP` (real or complex) – Coordinates of a point on the embedded elliptic curve associated with this period lattice.
- `prec` (default: `None`) – real precision in bits (default real precision if None).
- `reduce` (default: `True`) – if `True`, the result is reduced with respect to the period lattice basis.

OUTPUT:

(complex number) The elliptic logarithm of the point (xP, yP) with respect to this period lattice. If E is the elliptic curve and $\sigma : K \to \mathbf{C}$ the embedding, the returned value z is such that $z \pmod L$ maps to $(xP, yP) = \sigma(P)$ under the standard Weierstrass isomorphism from $\mathbf{C}/L$ to $\sigma(E)$. If `reduce` is `True`, the output is reduced so that it is in the fundamental period parallelogram with respect to the normalised lattice basis.

ALGORITHM:

Uses the complex AGM. See [CT2013] for details.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: L = E.period_lattice()
sage: P = E([-1,1])
sage: xP, yP = [RR(c) for c in P.xy()]
```

The elliptic log from the real coordinates:

```
sage: L.e_log_RC(xP, yP)
0.479348250190219 + 0.985868850775824*I
```

The same elliptic log from the algebraic point:

```
sage: L(P)
0.479348250190219 + 0.985868850775824*I
```

A number field example:

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)
sage: E = EllipticCurve([0,0,0,0,a])
sage: v = K.real_places()[0]
sage: L = E.period_lattice(v)
sage: P = E.lift_x(1/3*a^2 + a + 5/3)
sage: L(P)
3.51086196882538
sage: xP, yP = [v(c) for c in P.xy()]
sage: L.e_log_RC(xP, yP)
3.51086196882538
```

Elliptic logs of real points which do not come from algebraic points:

```
sage: # needs sage.rings.number_field
sage: ER = EllipticCurve([v(ai) for ai in E.a_invariants()])
sage: P = ER.lift_x(12.34)
sage: xP, yP = P.xy()
sage: xP, yP
(12.3400000000000, -43.3628968710567)
sage: L.e_log_RC(xP, yP)
0.284656841192041
sage: xP, yP = ER.lift_x(0).xy()
sage: L.e_log_RC(xP, yP)
1.34921304541057
```

Elliptic logs of complex points:

```
sage: # needs sage.rings.number_field
sage: v = K.complex_embeddings()[0]
sage: L = E.period_lattice(v)
sage: P = E.lift_x(1/3*a^2 + a + 5/3)
sage: L(P)
1.68207104397706 - 1.87873661686704*I
sage: xP, yP = [v(c) for c in P.xy()]
sage: L.e_log_RC(xP, yP)
1.68207104397706 - 1.87873661686704*I
sage: EC = EllipticCurve([v(ai) for ai in E.a_invariants()])
sage: xP, yP = EC.lift_x(0).xy()
sage: L.e_log_RC(xP, yP)
2.06711431204080 - 1.73451485683471*I
```

ei()

Return the x-coordinates of the 2-division points of the elliptic curve associated with this period lattice, as elements of `QQbar`.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: L = E.period_lattice()
sage: L.ei()
[-1.1071598716887 68?, 0.2695944364054446?, 0.8375654352833230?]
```

In the following example, we should have one purely real 2-division point coordinate, and two conjugate purely imaginary coordinates.

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)
sage: E = EllipticCurve([0,1,0,a,a])
sage: L = E.period_lattice(K.embeddings(RealField())[0])
sage: x1,x2,x3 = L.ei()
sage: abs(x1.real()) + abs(x2.real()) < 1e-14
True
sage: x1.imag(), x2.imag(), x3
(-1.122462048309373?, 1.122462048309373?, -1.000000000000000?)
```

```
sage: L = E.period_lattice(K.embeddings(ComplexField())[0])                    #␣
↪needs sage.rings.number_field
sage: L.ei()                                                                   #␣
↪needs sage.rings.number_field
[-1.000000000000000? + 0.?e-1...*I,
 -0.9720806486198328? - 0.561231024154687?*I,
 0.9720806486198328? + 0.561231024154687?*I]
```

elliptic_exponential(*z*, *to_curve=True*)

Return the elliptic exponential of a complex number.

INPUT:

- `z` (complex) – A complex number (viewed modulo this period lattice).
- `to_curve` (bool, default `True`): see below.

OUTPUT:

- If `to_curve` is False, a 2-tuple of real or complex numbers representing the point $(x, y) = (\wp(z), \wp'(z))$ where $\wp$ denotes the Weierstrass $\wp$-function with respect to this lattice.
- If `to_curve` is True, the point $(X, Y) = (x - b_2/12, y - (a_1(x - b_2/12) - a_3)/2)$ as a point in $E(\mathbf{R})$ or $E(\mathbf{C})$, with $(x, y) = (\wp(z), \wp'(z))$ as above, where E is the elliptic curve over **R** or **C** whose period lattice this is.
- If the lattice is real and z is also real then the output is a pair of real numbers if `to_curve` is `True`, or a point in $E(\mathbf{R})$ if `to_curve` is `False`.

Note: The precision is taken from that of the input `z`.

EXAMPLES:

```
sage: E = EllipticCurve([1,1,1,-8,6])
sage: P = E(1, -2)
sage: L = E.period_lattice()
sage: z = L(P); z
```

(continues on next page)

(continued from previous page)

```
1.17044757240090
sage: L.elliptic_exponential(z)
(0.999999999999999 : -2.00000000000000 : 1.00000000000000)
sage: _.curve()
Elliptic Curve defined by y^2 + 1.00000000000000*x*y + 1.00000000000000*y
 = x^3 + 1.00000000000000*x^2 - 8.00000000000000*x + 6.00000000000000
 over Real Field with 53 bits of precision
sage: L.elliptic_exponential(z,to_curve=False)
(1.41666666666667, -2.00000000000000)
sage: z = L(P, prec=201); z
1.17044757240089592298992188482371493504472561677451007994189
sage: L.elliptic_exponential(z)
(1.00000000000000000000000000000000000000000000000000000000000
 : -2.00000000000000000000000000000000000000000000000000000000000
 : 1.00000000000000000000000000000000000000000000000000000000000)
```

Examples over number fields:

```
sage: # needs sage.rings.number_field
sage: x = polygen(QQ)
sage: K.<a> = NumberField(x^3 - 2)
sage: embs = K.embeddings(CC)
sage: E = EllipticCurve('37a')
sage: EK = E.change_ring(K)
sage: Li = [EK.period_lattice(e) for e in embs]
sage: P = EK(-1, -1)
sage: Q = EK(a - 1, 1 - a^2)
sage: zi = [L.elliptic_logarithm(P) for L in Li]
sage: [c.real() for c in Li[0].elliptic_exponential(zi[0])]
[-1.00000000000000, -1.00000000000000, 1.00000000000000]
sage: [c.real() for c in Li[0].elliptic_exponential(zi[1])]
[-1.00000000000000, -1.00000000000000, 1.00000000000000]
sage: [c.real() for c in Li[0].elliptic_exponential(zi[2])]
[-1.00000000000000, -1.00000000000000, 1.00000000000000]

sage: # needs sage.rings.number_field
sage: zi = [L.elliptic_logarithm(Q) for L in Li]
sage: Li[0].elliptic_exponential(zi[0])
(-1.62996052494744 - 1.09112363597172*I
 : 1.79370052598410 - 1.37472963699860*I
 : 1.00000000000000)
sage: [embs[0](c) for c in Q]
[-1.62996052494744 - 1.09112363597172*I,
 1.79370052598410 - 1.37472963699860*I,
 1.00000000000000]
sage: Li[1].elliptic_exponential(zi[1])
(-1.62996052494744 + 1.09112363597172*I
 : 1.79370052598410 + 1.37472963699860*I
 : 1.00000000000000)
sage: [embs[1](c) for c in Q]
[-1.62996052494744 + 1.09112363597172*I,
 1.79370052598410 + 1.37472963699860*I,
 1.00000000000000]
sage: [c.real() for c in Li[2].elliptic_exponential(zi[2])]
[0.259921049894873, -0.587401051968199, 1.00000000000000]
sage: [embs[2](c) for c in Q]
[0.259921049894873, -0.587401051968200, 1.00000000000000]
```

Test to show that Issue #8820 is fixed:

```
sage: # needs sage.rings.number_field
sage: E = EllipticCurve('37a')
sage: K.<a> = QuadraticField(-5)
sage: L = E.change_ring(K).period_lattice(K.places()[0])
sage: L.elliptic_exponential(CDF(.1,.1))
(0.0000142854026029... - 49.9960001066650*I
 : 249.520141250950 + 250.019855549131*I : 1.00000000000000)
sage: L.elliptic_exponential(CDF(.1,.1), to_curve=False)
(0.0000142854026029447 - 49.9960001066650*I,
 500.040282501900 + 500.039711098263*I)
```

$z = 0$ is treated as a special case:

```
sage: E = EllipticCurve([1,1,1,-8,6])
sage: L = E.period_lattice()
sage: L.elliptic_exponential(0)
(0.000000000000000 : 1.00000000000000 : 0.000000000000000)
sage: L.elliptic_exponential(0, to_curve=False)
(+infinity, +infinity)
```

```
sage: # needs sage.rings.number_field
sage: E = EllipticCurve('37a')
sage: K.<a> = QuadraticField(-5)
sage: L = E.change_ring(K).period_lattice(K.places()[0])
sage: P = L.elliptic_exponential(0); P
(0.000000000000000 : 1.00000000000000 : 0.000000000000000)
sage: P.parent()
Abelian group of points on Elliptic Curve defined by
 y^2 + 1.00000000000000*y = x^3 + (-1.00000000000000)*x
 over Complex Field with 53 bits of precision
```

Very small z are handled properly (see Issue #8820):

```
sage: # needs sage.rings.number_field
sage: K.<a> = QuadraticField(-1)
sage: E = EllipticCurve([0,0,0,a,0])
sage: L = E.period_lattice(K.complex_embeddings()[0])
sage: L.elliptic_exponential(1e-100)
(0.000000000000000 : 1.00000000000000 : 0.000000000000000)
```

The elliptic exponential of z is returned as $(0 : 1 : 0)$ if the coordinates of z with respect to the period lattice are approximately integral:

```
sage: (100/log(2.0,10))/0.8
415.241011860920
sage: L.elliptic_exponential((RealField(415)(1e-100))).is_zero()            #␣
↪needs sage.rings.number_field
True
sage: L.elliptic_exponential((RealField(420)(1e-100))).is_zero()            #␣
↪needs sage.rings.number_field
False
```

elliptic_logarithm(*P*, *prec=None*, *reduce=True*)

Return the elliptic logarithm of a point.

INPUT:

- `P` (point) – A point on the elliptic curve associated with this period lattice.
- `prec` (default: `None`) – real precision in bits (default real precision if `None`).
- `reduce` (default: `True`) – if `True`, the result is reduced with respect to the period lattice basis.

OUTPUT:

(complex number) The elliptic logarithm of the point P with respect to this period lattice. If E is the elliptic curve and $\sigma : K \to \mathbf{C}$ the embedding, the returned value z is such that $z \pmod L$ maps to $\sigma(P)$ under the standard Weierstrass isomorphism from $\mathbf{C}/L$ to $\sigma(E)$. If `reduce` is `True`, the output is reduced so that it is in the fundamental period parallelogram with respect to the normalised lattice basis.

ALGORITHM:

Uses the complex AGM. See [CT2013] for details.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: L = E.period_lattice()
sage: E.discriminant() > 0
True
sage: L.real_flag
1
sage: P = E([-1,1])
sage: P.is_on_identity_component ()
False
sage: L.elliptic_logarithm(P, prec=96)
0.47934825019021931612953301 01 + 0.985868850775824102211203849 1*I
sage: Q=E([3,5])
sage: Q.is_on_identity_component()
True
sage: L.elliptic_logarithm(Q, prec=96)
1.9311282715425594424885852 0
```

Note that this is actually the inverse of the Weierstrass isomorphism:

```
sage: L.elliptic_exponential(_)  # abs tol 1e-26
(3.0000000000000000000000000000 : 5.0000000000000000000000000000 : 1.
↪0000000000000000000000000000)
```

An example with negative discriminant, and a torsion point:

```
sage: E = EllipticCurve('11a1')
sage: L = E.period_lattice()
sage: E.discriminant() < 0
True
sage: L.real_flag
-1
sage: P = E([16,-61])
sage: L.elliptic_logarithm(P)
0.253841860855911
sage: L.real_period() / L.elliptic_logarithm(P)
5.00000000000000
```

An example where precision is problematic:

```
sage: E = EllipticCurve([1, 0, 1, -85357462, 303528987048])  #18074g1
sage: P = E([4458713781401/835903744, -64466909836503771/24167649046528, 1])
```

(continues on next page)

(continued from previous page)

```
sage: L = E.period_lattice()
sage: L.ei()
[5334.003952567705? - 1.964393150436?e-6*I,
 5334.003952567705? + 1.964393150436?e-6*I,
 -10668.257905135412?]
sage: L.elliptic_logarithm(P,prec=100)
0.27656204014107061464076203097
```

Some complex examples, taken from the paper by Cremona and Thongjunthug:

```
sage: # needs sage.rings.number_field
sage: K.<i> = QuadraticField(-1)
sage: a4 = 9*i - 10
sage: a6 = 21 - i
sage: E = EllipticCurve([0,0,0,a4,a6])
sage: e1 = 3 - 2*i; e2 = 1 + i; e3 = -4 + i
sage: emb = K.embeddings(CC)[1]
sage: L = E.period_lattice(emb)
sage: P = E(2 - i, 4 + 2*i)
```

By default, the output is reduced with respect to the normalised lattice basis, so that its coordinates with respect to that basis lie in the interval [0,1):

```
sage: z = L.elliptic_logarithm(P, prec=100); z                                  #␣
↪needs sage.rings.number_field
0.70448375537782208460499649302 - 0.79246725643650979858266018068*I
sage: L.coordinates(z)                                                          #␣
↪needs sage.rings.number_field
(0.46247636364807931766105406092, 0.79497588726808704200760395829)
```

Using `reduce=False` this step can be omitted. In this case the coordinates are usually in the interval [-0.5,0.5), but this is not guaranteed. This option is mainly for testing purposes:

```
sage: z = L.elliptic_logarithm(P, prec=100, reduce=False); z                    #␣
↪needs sage.rings.number_field
0.57002153834710752778063503023 + 0.46476340520469798857457031393*I
sage: L.coordinates(z)                                                          #␣
↪needs sage.rings.number_field
(0.46247636364807931766105406092, -0.20502411273191295799239604171)
```

The elliptic logs of the 2-torsion points are half-periods:

```
sage: L.elliptic_logarithm(E(e1, 0), prec=100)                                  #␣
↪needs sage.rings.number_field
0.64607575874356525952487867052 + 0.22379609053909448304176885364*I
sage: L.elliptic_logarithm(E(e2, 0), prec=100)                                  #␣
↪needs sage.rings.number_field
0.71330686725892253793705940192 - 0.40481924028150941053684639367*I
sage: L.elliptic_logarithm(E(e3, 0), prec=100)                                  #␣
↪needs sage.rings.number_field
0.067231108515357278412180731396 - 0.62861533082060389357861524731*I
```

We check this by doubling and seeing that the resulting coordinates are integers:

```
sage: L.coordinates(2*L.elliptic_logarithm(E(e1, 0), prec=100))                 #␣
↪needs sage.rings.number_field
```

(continues on next page)

(continued from previous page)

```
(1.0000000000000000000000000000, 0.0000000000000000000000000000)
sage: L.coordinates(2*L.elliptic_logarithm(E(e2, 0), prec=100))                #␣
↪needs sage.rings.number_field
(1.0000000000000000000000000000, 1.0000000000000000000000000000)
sage: L.coordinates(2*L.elliptic_logarithm(E(e3, 0), prec=100))                #␣
↪needs sage.rings.number_field
(0.0000000000000000000000000000, 1.0000000000000000000000000000)
```

```
sage: # needs sage.rings.number_field
sage: a4 = -78*i + 104
sage: a6 = -216*i - 312
sage: E = EllipticCurve([0,0,0,a4,a6])
sage: emb = K.embeddings(CC)[1]
sage: L = E.period_lattice(emb)
sage: P = E(3 + 2*i, 14 - 7*i)
sage: L.elliptic_logarithm(P)
0.297147783912228 - 0.546125549639461*I
sage: L.coordinates(L.elliptic_logarithm(P))
(0.628653378040238, 0.371417754610223)
sage: e1 = 1 + 3*i; e2 = -4 - 12*i; e3 = -e1 - e2
sage: L.coordinates(L.elliptic_logarithm(E(e1, 0)))
(0.500000000000000, 0.500000000000000)
sage: L.coordinates(L.elliptic_logarithm(E(e2, 0)))
(1.00000000000000, 0.500000000000000)
sage: L.coordinates(L.elliptic_logarithm(E(e3, 0)))
(0.500000000000000, 0.000000000000000)
```

gens (*prec=None*, *algorithm='sage'*)

Return a basis for this period lattice as a 2-tuple.

This is an alias for *basis()*. See the docstring there for a more in-depth explanation and further examples.

INPUT:

- `prec` (default: `None`) – precision in bits (default precision if `None`).
- `algorithm` (string, default `'sage'`) – choice of implementation (for real embeddings only) between `'sage'` (native Sage implementation) or `'pari'` (use the PARI library: only available for real embeddings).

OUTPUT:

(tuple of Complex) (ω_1, ω_2) where the lattice is $\mathbf{Z}\omega_1 + \mathbf{Z}\omega_2$. If the lattice is real then ω_1 is real and positive, $\Im(\omega_2) > 0$ and $\Re(\omega_1/\omega_2)$ is either 0 (for rectangular lattices) or $\frac{1}{2}$ (for non-rectangular lattices). Otherwise, ω_1/ω_2 is in the fundamental region of the upper half-plane. If the latter normalisation is required for real lattices, use the method *normalised_basis()* instead.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.period_lattice().gens()
(2.99345864623196, 2.45138938198679*I)

sage: E.period_lattice().gens(prec=100)
(2.9934586462319596298320099794, 2.4513893819867900608542248319*I)
```

is_real()

Return `True` if this period lattice is real.

EXAMPLES:

```
sage: f = EllipticCurve('11a')
sage: f.period_lattice().is_real()
True
```

```
sage: # needs sage.rings.number_field
sage: K.<i> = QuadraticField(-1)
sage: E = EllipticCurve(K, [0,0,0,i,2*i])
sage: emb = K.embeddings(ComplexField())[0]
sage: L = E.period_lattice(emb)
sage: L.is_real()
False
```

```
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)                                    #␣
↪needs sage.rings.number_field
sage: E = EllipticCurve([0,1,0,a,a])                                  #␣
↪needs sage.rings.number_field
sage: [E.period_lattice(emb).is_real() for emb in K.embeddings(CC)]   #␣
↪needs sage.rings.number_field
[False, False, True]
```

ALGORITHM:

The lattice is real if it is associated to a real embedding; such lattices are stable under conjugation.

is_rectangular()

Return `True` if this period lattice is rectangular.

Note: Only defined for real lattices; a `RuntimeError` is raised for non-real lattices.

EXAMPLES:

```
sage: f = EllipticCurve('11a')
sage: f.period_lattice().basis()
(1.26920930427955, 0.634604652139777 + 1.45881661693850*I)
sage: f.period_lattice().is_rectangular()
False
```

```
sage: f = EllipticCurve('37b')
sage: f.period_lattice().basis()
(1.08852159290423, 1.76761067023379*I)
sage: f.period_lattice().is_rectangular()
True
```

ALGORITHM:

The period lattice is rectangular precisely if the discriminant of the Weierstrass equation is positive, or equivalently if the number of real components is 2.

normalised_basis(*prec=None*, *algorithm='sage'*)

Return a normalised basis for this period lattice as a 2-tuple.

INPUT:

- `prec` (default: `None`) – precision in bits (default precision if `None`).

- `algorithm` (string, default `'sage'`) – choice of implementation (for real embeddings only) between `'sage'` (native Sage implementation) or `'pari'` (use the PARI library: only available for real embeddings).

OUTPUT:

(tuple of Complex) (ω_1, ω_2) where the lattice has the form $\mathbf{Z}\omega_1 + \mathbf{Z}\omega_2$. The basis is normalised so that ω_1/ω_2 is in the fundamental region of the upper half-plane. For an alternative normalisation for real lattices (with the first period real), use the method *basis()* instead.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.period_lattice().normalised_basis()
(2.99345864623196, -2.45138938198679*I)
```

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)
sage: emb = K.embeddings(RealField())[0]
sage: E = EllipticCurve([0,1,0,a,a])
sage: L = E.period_lattice(emb)
sage: L.normalised_basis(64)
(1.90726488608927255 - 1.34047785962440202*I,
-1.90726488608927255 - 1.34047785962440202*I)

sage: # needs sage.rings.number_field
sage: emb = K.embeddings(ComplexField())[0]
sage: L = E.period_lattice(emb)
sage: w1, w2 = L.normalised_basis(); w1, w2
(-1.37588604166076 - 2.58560946624443*I,
 -2.10339907847356 + 0.428378776460622*I)
sage: L.is_real()
False
sage: tau = w1/w2; tau
0.387694505032876 + 1.30821088214407*I
```

omega (*prec=None*, *bsd_normalise=False*)

Return the real or complex volume of this period lattice.

INPUT:

- `prec` (int or `None` (default)) – real precision in bits (default real precision if `None`)
- `bsd_normalise` (bool, default `False`) – flag to use BSD normalisation in the complex case.

OUTPUT:

(real) For real lattices, this is the real period times the number of connected components. For non-real lattices it is the complex area, or double the area if `bsd_normalise` is `True`.

Note: If the curve is given by a *global minimal* Weierstrass equation, then with `bsd_normalise = True`, this gives the correct period in the BSD conjecture: the product of this quantity over all embeddings appears in the BSD formula. In general a correction factor is required to make allowance for the model.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.period_lattice().omega()
5.98691729246392
```

This is not a minimal model:

```
sage: E = EllipticCurve([0, -432*6^2])
sage: E.period_lattice().omega()
0.486109385710056
```

If you were to plug the above omega into the BSD conjecture, you would get an incorrect value, out by a factor of 2. The following works though:

```
sage: F = E.minimal_model()
sage: F.period_lattice().omega()
0.972218771420113
```

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)
sage: emb = K.embeddings(RealField())[0]
sage: E = EllipticCurve([0,1,0,a,a])
sage: L = E.period_lattice(emb)
sage: L.omega(64)
3.81452977217854509
```

A complex example (taken from J.E.Cremona and E.Whitley, *Periods of cusp forms and elliptic curves over imaginary quadratic fields*, Mathematics of Computation 62 No. 205 (1994), 407-429). See Issue #29645 and Issue #29782:

```
sage: # needs sage.rings.number_field
sage: K.<i> = QuadraticField(-1)
sage: E = EllipticCurve([0,1-i,i,-i,0])
sage: L = E.period_lattice(K.embeddings(CC)[0])
sage: L.omega()
8.80694160502647
sage: L.omega(prec=200)
8.80694160502647414932507436322954622278586307653921140700 32
sage: L.omega(bsd_normalise=True)
17.6138832100529
```

real_period(*prec=None*, *algorithm='sage'*)

Return the real period of this period lattice.

INPUT:

- `prec` (integer or `None` (default)) – real precision in bits (default real precision if `None`)
- `algorithm` (string, default `'sage'`) – choice of implementation (for real embeddings only) between `'sage'` (native Sage implementation) or `'pari'` (use the PARI library: only available for real embeddings).

Note: Only defined for real lattices; a `RuntimeError` is raised for non-real lattices.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.period_lattice().real_period()
2.99345864623196
```

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)
sage: emb = K.embeddings(RealField())[0]
sage: E = EllipticCurve([0,1,0,a,a])
sage: L = E.period_lattice(emb)
sage: L.real_period(64)
3.81452977217854509
```

reduce(*z*)

Reduce a complex number modulo the lattice

INPUT:

- z (complex) – A complex number.

OUTPUT:

(complex) the reduction of z modulo the lattice, lying in the fundamental period parallelogram with respect to the lattice basis. For curves defined over the reals (i.e. real embeddings) the output will be real when possible.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: L = E.period_lattice()
sage: w1, w2 = L.basis(prec=100)
sage: P = E([-1,1])
sage: zP = P.elliptic_logarithm(precision=100); zP
0.47934825019021931612953301006 + 0.98586885077582410221120384908*I
sage: z = zP + 10*w1 - 20*w2; z
25.381473858740770069343110929 - 38.448885180257139986236950114*I
sage: L.reduce(z)
0.47934825019021931612953301006 + 0.98586885077582410221120384908*I
sage: L.elliptic_logarithm(2*P)
0.958696500380439
sage: L.reduce(L.elliptic_logarithm(2*P))
0.958696500380439
sage: L.reduce(L.elliptic_logarithm(2*P) + 10*w1 - 20*w2)
0.958696500380444
```

sigma(*z*, *prec=None*, *flag=0*)

Return the value of the Weierstrass sigma function for this elliptic curve period lattice.

INPUT:

- z – a complex number
- **prec (default: None) – real precision in bits**
  (default real precision if None).
- flag –

  0: (default) ???;

  1: computes an arbitrary determination of log(sigma(z))

  2, 3: same using the product expansion instead of theta series. ???

Note: The reason for the ???'s above, is that the PARI documentation for ellsigma is very vague. Also this is only implemented for curves defined over **Q**.

Todo: This function does not use any of the PeriodLattice functions and so should be moved to ell_rational_field.

EXAMPLES:

```
sage: EllipticCurve('389a1').period_lattice().sigma(CC(2,1))
2.60912163570108 - 0.200865080824587*I
```

tau (*prec=None*, *algorithm='sage'*)

Return the upper half-plane parameter in the fundamental region.

INPUT:

- `prec` (default: `None`) – precision in bits (default precision if `None`).
- `algorithm` (string, default 'sage') – choice of implementation (for real embeddings only) between 'sage' (native Sage implementation) or 'pari' (use the PARI library: only available for real embeddings).

OUTPUT:

(Complex) $\tau = \omega_1/\omega_2$ where the lattice has the form $\mathbf{Z}\omega_1 + \mathbf{Z}\omega_2$, normalised so that $\tau = \omega_1/\omega_2$ is in the fundamental region of the upper half-plane.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: L = E.period_lattice()
sage: L.tau()
1.22112736076463*I
```

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^3 - 2)
sage: emb = K.embeddings(RealField())[0]
sage: E = EllipticCurve([0,1,0,a,a])
sage: L = E.period_lattice(emb)
sage: tau = L.tau(); tau
-0.338718341018919 + 0.940887817679340*I
sage: tau.abs()
1.00000000000000
sage: -0.5 <= tau.real() <= 0.5
True

sage: # needs sage.rings.number_field
sage: emb = K.embeddings(ComplexField())[0]
sage: L = E.period_lattice(emb)
sage: tau = L.tau(); tau
0.387694505032876 + 1.30821088214407*I
sage: tau.abs()
1.36444961115933
sage: -0.5 <= tau.real() <= 0.5
True
```

sage.schemes.elliptic_curves.period_lattice.**extended_agm_iteration**(*a*, *b*, *c*)

Internal function for the extended AGM used in elliptic logarithm computation.

INPUT:

- a, b, c (real or complex) – three real or complex numbers.

OUTPUT:

(3-tuple) (a_0, b_0, c_0), the limit of the iteration $(a, b, c) \mapsto ((a+b)/2, \sqrt{ab}, (c+\sqrt{(c^2+b^2-a^2)})/2)$.

EXAMPLES:

```
sage: # needs sage.rings.real_mpfr
sage: from sage.schemes.elliptic_curves.period_lattice import extended_agm_
↪iteration
sage: extended_agm_iteration(RR(1), RR(2), RR(3))
(1.45679103104691, 1.45679103104691, 3.21245294970054)
sage: extended_agm_iteration(CC(1,2), CC(2,3), CC(3,4))
(1.46242448156430 + 2.47791311676267*I,
 1.46242448156430 + 2.47791311676267*I,
 3.22202144343535 + 4.28383734262540*I)
```

sage.schemes.elliptic_curves.period_lattice.**normalise_periods**(*w1*, *w2*)

Normalise the period basis (w_1, w_2) so that w_1/w_2 is in the fundamental region.

INPUT:

- w1, w2 – two complex numbers with non-real ratio

OUTPUT:

(tuple) $((\omega_1', \omega_2'), [a, b, c, d])$ where a, b, c, d are integers such that

- $ad - bc = \pm 1$;
- $(\omega_1', \omega_2') = (a\omega_1 + b\omega_2, c\omega_1 + d\omega_2)$;
- $\tau = \omega_1'/\omega_2'$ is in the upper half plane;
- $|\tau| \geq 1$ and $|\Re(\tau)| \leq \frac{1}{2}$.

EXAMPLES:

```
sage: # needs sage.rings.real_mpfr sage.symbolic
sage: from sage.schemes.elliptic_curves.period_lattice import reduce_tau,␣
↪normalise_periods
sage: w1 = CC(1.234, 3.456)
sage: w2 = CC(1.234, 3.456000001)
sage: w1/w2    # in lower half plane!
0.999999999743367 - 9.16334785827644e-11*I
sage: w1w2, abcd = normalise_periods(w1, w2)
sage: a,b,c,d = abcd
sage: w1w2 == (a*w1+b*w2, c*w1+d*w2)
True
sage: w1w2[0]/w1w2[1]
1.23400010389203e9*I
sage: a*d-b*c # note change of orientation
-1
```

sage.schemes.elliptic_curves.period_lattice.**reduce_tau**(*tau*)

Transform a point in the upper half plane to the fundamental region.

INPUT:

- `tau` (complex) – a complex number with positive imaginary part

OUTPUT:

(tuple) $(\tau', [a, b, c, d])$ where a, b, c, d are integers such that

- $ad - bc = 1$;
- $\tau' = (a\tau + b)/(c\tau + d)$;
- $|\tau'| \geq 1$;
- $|\Re(\tau')| \leq \frac{1}{2}$.

EXAMPLES:

```
sage: # needs sage.rings.real_mpfr sage.symbolic
sage: from sage.schemes.elliptic_curves.period_lattice import reduce_tau
sage: reduce_tau(CC(1.23,3.45))
(0.230000000000000 + 3.45000000000000*I, [1, -1, 0, 1])
sage: reduce_tau(CC(1.23,0.0345))
(-0.463960069171512 + 1.35591888067914*I, [-5, 6, 4, -5])
sage: reduce_tau(CC(1.23,0.0000345))
(0.130000000001761 + 2.89855072463768*I, [13, -16, 100, -123])
```

18.18 Regions in fundamental domains of period lattices

This module is used to represent sub-regions of a fundamental parallelogram of the period lattice of an elliptic curve, used in computing minimum height bounds.

In particular, these are the approximating sets `S^{(v)}` in section 3.2 of Thotsaphon Thongjunthug's Ph.D. Thesis and paper [Tho2010].

AUTHORS:

- Robert Bradshaw (2010): initial version
- John Cremona (2014): added some docstrings and doctests

class sage.schemes.elliptic_curves.period_lattice_region.**PeriodicRegion**

Bases: `object`

EXAMPLES:

```
sage: import numpy as np
sage: from sage.schemes.elliptic_curves.period_lattice_region import␣
↪PeriodicRegion
sage: S = PeriodicRegion(CDF(2), CDF(2*I), np.zeros((4, 4)))
sage: S.plot()                                                          #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
sage: data = np.zeros((4, 4))
sage: data[1,1] = True
sage: S = PeriodicRegion(CDF(2), CDF(2*I+1), data)
sage: S.plot()                                                          #␣
↪needs sage.plot
Graphics object consisting of 5 graphics primitives
```

border(*raw=True*)

Returns the boundary of this region as set of tile boundaries.

If raw is true, returns a list with respect to the internal bitmap, otherwise returns complex intervals covering the border.

EXAMPLES:

```
sage: import numpy as np
sage: from sage.schemes.elliptic_curves.period_lattice_region import␣
↪PeriodicRegion
sage: data = np.zeros((4, 4))
sage: data[1, 1] = True
sage: PeriodicRegion(CDF(1), CDF(I), data).border()
[(1, 1, 0), (2, 1, 0), (1, 1, 1), (1, 2, 1)]
sage: PeriodicRegion(CDF(2), CDF(I-1/2), data).border()
[(1, 1, 0), (2, 1, 0), (1, 1, 1), (1, 2, 1)]

sage: PeriodicRegion(CDF(1), CDF(I), data).border(raw=False)
[0.25000000000000000? + 1.?*I,
 0.50000000000000000? + 1.?*I,
 1.? + 0.25000000000000000?*I,
 1.? + 0.50000000000000000?*I]
sage: PeriodicRegion(CDF(2), CDF(I-1/2), data).border(raw=False)
[0.3? + 1.?*I,
 0.8? + 1.?*I,
 1.? + 0.25000000000000000?*I,
 1.? + 0.50000000000000000?*I]

sage: data[1:3, 2] = True
sage: PeriodicRegion(CDF(1), CDF(I), data).border()
[(1, 1, 0), (2, 1, 0), (1, 1, 1), (1, 2, 0), (1, 3, 1), (3, 2, 0), (2, 2, 1),␣
↪(2, 3, 1)]
```

contract(*corners=True*)

Opposite (but not inverse) of expand; removes neighbors of complement.

EXAMPLES:

```
sage: import numpy as np
sage: from sage.schemes.elliptic_curves.period_lattice_region import␣
↪PeriodicRegion
sage: data = np.zeros((10, 10))
sage: data[1:4,1:4] = True
sage: S = PeriodicRegion(CDF(1), CDF(I + 1/2), data)
sage: S.plot()                                                              #␣
↪needs sage.plot
Graphics object consisting of 13 graphics primitives
sage: S.contract().plot()                                                   #␣
↪needs sage.plot
Graphics object consisting of 5 graphics primitives
sage: S.contract().data.sum()
1
sage: S.contract().contract().is_empty()
True
```

data

ds()

Returns the sides of each parallelogram tile.

EXAMPLES:

```
sage: import numpy as np
sage: from sage.schemes.elliptic_curves.period_lattice_region import␣
↪PeriodicRegion
sage: data = np.zeros((4, 4))
sage: S = PeriodicRegion(CDF(2), CDF(2*I), data, full=False)
sage: S.ds()
(0.5, 0.25*I)
sage: _ = S._ensure_full()
sage: S.ds()
(0.5, 0.25*I)

sage: data = np.zeros((8, 8))
sage: S = PeriodicRegion(CDF(1), CDF(I + 1/2), data)
sage: S.ds()
(0.125, 0.0625 + 0.125*I)
```

expand(*corners=True*)

Returns a region containing this region by adding all neighbors of internal tiles.

EXAMPLES:

```
sage: import numpy as np
sage: from sage.schemes.elliptic_curves.period_lattice_region import␣
↪PeriodicRegion
sage: data = np.zeros((4, 4))
sage: data[1,1] = True
sage: S = PeriodicRegion(CDF(1), CDF(I + 1/2), data)
sage: S.plot()                                                              #␣
↪needs sage.plot
Graphics object consisting of 5 graphics primitives
sage: S.expand().plot()                                                     #␣
↪needs sage.plot
Graphics object consisting of 13 graphics primitives
sage: S.expand().data
array([[1, 1, 1, 0],
       [1, 1, 1, 0],
       [1, 1, 1, 0],
       [0, 0, 0, 0]], dtype=int8)
sage: S.expand(corners=False).plot()                                        #␣
↪needs sage.plot
Graphics object consisting of 13 graphics primitives
sage: S.expand(corners=False).data
array([[0, 1, 0, 0],
       [1, 1, 1, 0],
       [0, 1, 0, 0],
       [0, 0, 0, 0]], dtype=int8)
```

full

innermost_point()

Return a point well inside the region, specifically the center of (one of) the last tile(s) to be removed on contraction.

EXAMPLES:

```
sage: import numpy as np
sage: from sage.schemes.elliptic_curves.period_lattice_region import␣
↪PeriodicRegion
sage: data = np.zeros((10, 10))
sage: data[1:4, 1:4] = True
sage: data[1, 0:8] = True
sage: S = PeriodicRegion(CDF(1), CDF(I+1/2), data)
sage: S.innermost_point()
0.375 + 0.25*I
sage: S.plot() + point(S.innermost_point())                                    #␣
↪needs sage.plot
Graphics object consisting of 24 graphics primitives
```

is_empty()

Returns whether this region is empty.

EXAMPLES:

```
sage: import numpy as np
sage: from sage.schemes.elliptic_curves.period_lattice_region import␣
↪PeriodicRegion
sage: data = np.zeros((4, 4))
sage: PeriodicRegion(CDF(2), CDF(2*I), data).is_empty()
True
sage: data[1,1] = True
sage: PeriodicRegion(CDF(2), CDF(2*I), data).is_empty()
False
```

plot(***kwds*)

Plot this region in the fundamental lattice. If `full` is `False`, plots only the lower half. Note that the true nature of this region is periodic.

EXAMPLES:

```
sage: import numpy as np
sage: from sage.schemes.elliptic_curves.period_lattice_region import␣
↪PeriodicRegion
sage: data = np.zeros((10, 10))
sage: data[2, 2:8] = True
sage: data[2:5, 2] = True
sage: data[3, 3] = True
sage: S = PeriodicRegion(CDF(1), CDF(I + 1/2), data)
sage: plot(S) + plot(S.expand(), rgbcolor=(1, 0, 1), thickness=2)              #␣
↪needs sage.plot
Graphics object consisting of 46 graphics primitives
```

refine(*condition=None*, *times=1*)

Recursive function to refine the current tiling.

INPUT:

- `condition` (function, default None) – if not None, only keep tiles in the refinement which satisfy the condition.
- `times` (int, default 1) – the number of times to refine; each refinement step halves the mesh size.

OUTPUT:

The refined PeriodicRegion.

EXAMPLES:

```
sage: import numpy as np
sage: from sage.schemes.elliptic_curves.period_lattice_region import␣
↪PeriodicRegion
sage: data = np.zeros((4, 4))
sage: S = PeriodicRegion(CDF(2), CDF(2*I), data, full=False)
sage: S.ds()
(0.5, 0.25*I)
sage: S = S.refine()
sage: S.ds()
(0.25, 0.125*I)
sage: S = S.refine(2)
sage: S.ds()
(0.125, 0.0625*I)
```

verify(*condition*)

Given a condition that should hold for every line segment on the boundary, verify that it actually does so.

INPUT:

- `condition` (function) – a boolean-valued function on **C**.

OUTPUT:

True or False according to whether the condition holds for all lines on the boundary.

EXAMPLES:

```
sage: import numpy as np
sage: from sage.schemes.elliptic_curves.period_lattice_region import␣
↪PeriodicRegion
sage: data = np.zeros((4, 4))
sage: data[1, 1] = True
sage: S = PeriodicRegion(CDF(1), CDF(I), data)
sage: S.border()
[(1, 1, 0), (2, 1, 0), (1, 1, 1), (1, 2, 1)]
sage: condition = lambda z: z.real().abs()<1/2
sage: S.verify(condition)
False
sage: condition = lambda z: z.real().abs()<1
sage: S.verify(condition)
True
```

w1

w2

Modularity and L-series over $\mathbf{Q}$.

18.19 Modular parametrization of elliptic curves over Q

By the work of Taylor–Wiles et al. it is known that there is a surjective morphism

$$\phi_E : X_0(N) \to E.$$

from the modular curve $X_0(N)$, where N is the conductor of E. The map sends the cusp ∞ to the origin of E.

EXAMPLES:

```
sage: phi = EllipticCurve('11a1').modular_parametrization()
sage: phi
Modular parameterization
 from the upper half plane
   to Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field
sage: phi(0.5+CDF(I))
(285684.320516... + 7.0...e-11*I : 1.526964169...e8 + 5.6...e-8*I : 1.00000000000000)
sage: phi.power_series(prec = 7)
(q^-2 + 2*q^-1 + 4 + 5*q + 8*q^2 + q^3 + 7*q^4 + O(q^5),
 -q^-3 - 3*q^-2 - 7*q^-1 - 13 - 17*q - 26*q^2 - 19*q^3 + O(q^4))
```

AUTHORS:

- Chris Wuthrich (02/10): moved from ell_rational_field.py.

class sage.schemes.elliptic_curves.modular_parametrization.**ModularParameterization**(E)

Bases: object

This class represents the modular parametrization of an elliptic curve

$$\phi_E : X_0(N) \to E.$$

Evaluation is done by passing through the lattice representation of E.

EXAMPLES:

```
sage: phi = EllipticCurve('11a1').modular_parametrization()
sage: phi
Modular parameterization
 from the upper half plane
   to Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20
      over Rational Field
```

curve()

Return the curve associated to this modular parametrization.

EXAMPLES:

```
sage: E = EllipticCurve('15a')
sage: phi = E.modular_parametrization()
sage: phi.curve() is E
True
```

map_to_complex_numbers(*z*, *prec=None*)

Evaluate `self` at a point $z \in X_0(N)$ where z is given by a representative in the upper half plane, returning a point in the complex numbers.

All computations are done with `prec` bits of precision. If `prec` is not given, use the precision of z. Use self(z) to compute the image of z on the Weierstrass equation of the curve.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: E = EllipticCurve('37a'); phi = E.modular_parametrization()
sage: x = polygen(ZZ, 'x')
sage: tau = (sqrt(7)*I - 17)/74
sage: z = phi.map_to_complex_numbers(tau); z
0.929592715285395 - 1.22569469099340*I
sage: E.elliptic_exponential(z)
(...e-16 - ...e-16*I : ...e-16 + ...e-16*I : 1.00000000000000)
sage: phi(tau)
(...e-16 - ...e-16*I : ...e-16 + ...e-16*I : 1.00000000000000)
```

power_series (*prec=20*)

Return the power series of this modular parametrization.

The curve must be a minimal model. The prec parameter determines the number of significant terms. This means that X will be given up to O(q^(prec-2)) and Y will be given up to O(q^(prec-3)).

OUTPUT: A list of two Laurent series `[X(x),Y(x)]` of degrees -2, -3 respectively, which satisfy the equation of the elliptic curve. There are modular functions on $\Gamma_0(N)$ where N is the conductor.

The series should satisfy the differential equation

$$\frac{\mathrm{d}X}{2Y + a_1X + a_3} = \frac{f(q)\,\mathrm{d}q}{q}$$

where f is `self.curve().q_expansion()`.

EXAMPLES:

```
sage: E = EllipticCurve('389a1')
sage: phi = E.modular_parametrization()
sage: X, Y = phi.power_series(prec=10)
sage: X
q^-2 + 2*q^-1 + 4 + 7*q + 13*q^2 + 18*q^3 + 31*q^4 + 49*q^5 + 74*q^6 + 111*q^
↪7 + O(q^8)
sage: Y
-q^-3 - 3*q^-2 - 8*q^-1 - 17 - 33*q - 61*q^2 - 110*q^3 - 186*q^4 - 320*q^5 -␣
↪528*q^6 + O(q^7)
sage: X,Y = phi.power_series()
sage: X
q^-2 + 2*q^-1 + 4 + 7*q + 13*q^2 + 18*q^3 + 31*q^4 + 49*q^5 + 74*q^6 + 111*q^
↪7 + 173*q^8 + 251*q^9 + 379*q^10 + 560*q^11 + 824*q^12 + 1199*q^13 + 1773*q^
↪14 + 2548*q^15 + 3722*q^16 + 5374*q^17 + O(q^18)
sage: Y
-q^-3 - 3*q^-2 - 8*q^-1 - 17 - 33*q - 61*q^2 - 110*q^3 - 186*q^4 - 320*q^5 -␣
↪528*q^6 - 861*q^7 - 1383*q^8 - 2218*q^9 - 3472*q^10 - 5451*q^11 - 8447*q^12␣
↪- 13020*q^13 - 19923*q^14 - 30403*q^15 - 46003*q^16 + O(q^17)
```

The following should give 0, but only approximately:

```
sage: q = X.parent().gen()
sage: E.defining_polynomial()(X,Y,1) + O(q^11) == 0
True
```

Note that below we have to change variable from x to q:

```
sage: a1,_,a3,_,_ = E.a_invariants()
sage: f = E.q_expansion(17)
```

(continues on next page)

(continued from previous page)

```
sage: q = f.parent().gen()
sage: f/q == (X.derivative()/(2*Y+a1*X+a3))
True
```

18.20 Modular symbols attached to elliptic curves over Q

To an elliptic curve E over the rational numbers with conductor N, one can associate a space of modular symbols of level N, because E is known to be modular. The space is two-dimensional and contains a subspace on which complex conjugation acts as multiplication by $+1$ and one on which it acts by -1.

There are three implementations of modular symbols, two within `Sage` and one in Cremona's `eclib` library. One can choose here which one is used.

Associated to E there is a canonical generator in each space. They are maps $[.]^+$ and $[.]^-$, both $\mathbf{Q} \to \mathbf{Q}$. They are normalized such that

$$[r]^+\Omega^+ + [r]^-\Omega^- = \int_\infty^r 2\pi i f(z)dz$$

where f is the newform associated to the isogeny class of E and Ω^+ is the smallest positive period of the Néron differential of E and Ω^- is the smallest positive purely imaginary period. Note that it depends on E rather than on its isogeny class.

From `eclib` version v20161230, both plus and minus symbols are available and are correctly normalized. In the `Sage` implementation, the computation of the space provides initial generators which are not necessarily correctly normalized; here we implement two methods that try to find the correct scaling factor.

Modular symbols are used to compute p-adic L-functions.

EXAMPLES:

```
sage: E = EllipticCurve("19a1")
sage: m = E.modular_symbol()
sage: m(0)
1/3
sage: m(1/17)
-2/3
sage: m2 = E.modular_symbol(-1, implementation="sage")
sage: m2(0)
0
sage: m2(1/5)
1/2

sage: V = E.modular_symbol_space()
sage: V
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 2
 for Gamma_0(19) of weight 2 with sign 1 over Rational Field
sage: V.q_eigenform(30)
q - 2*q^3 - 2*q^4 + 3*q^5 - q^7 + q^9 + 3*q^11 + 4*q^12 - 4*q^13 - 6*q^15 + 4*q^16
 - 3*q^17 + q^19 - 6*q^20 + 2*q^21 + 4*q^25 + 4*q^27 + 2*q^28 + 6*q^29 + O(q^30)
```

For more details on modular symbols consult the following

REFERENCES:

- [MTT1986]
- [Cre1997]

- [SW2013]

AUTHORS:

- William Stein (2007): first version
- Chris Wuthrich (2008): add scaling and reference to eclib
- John Cremona (2016): reworked eclib interface

class sage.schemes.elliptic_curves.ell_modular_symbols.**ModularSymbol**

Bases: SageObject

A modular symbol attached to an elliptic curve, which is the map $\mathbf{Q} \to \mathbf{Q}$ obtained by sending r to the normalized symmetrized (or anti-symmetrized) integral ∞ to r.

This is as defined in [MTT1986], but normalized to depend on the curve and not only its isogeny class as in [SW2013].

See the documentation of `E.modular_symbol()` in elliptic curves over the rational numbers for help.

base_ring()

Return the base ring for this modular symbol.

EXAMPLES:

```
sage: m = EllipticCurve('11a1').modular_symbol()
sage: m.base_ring()
Rational Field
```

elliptic_curve()

Return the elliptic curve of this modular symbol.

EXAMPLES:

```
sage: m = EllipticCurve('11a1').modular_symbol()
sage: m.elliptic_curve()
Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field
```

sign()

Return the sign of this elliptic curve modular symbol.

EXAMPLES:

```
sage: m = EllipticCurve('11a1').modular_symbol()
sage: m.sign()
1
sage: m = EllipticCurve('11a1').modular_symbol(sign=-1, implementation="sage")
sage: m.sign()
-1
```

class sage.schemes.elliptic_curves.ell_modular_symbols.**ModularSymbolECLIB**(*E*, *sign*, *nap=1000*)

Bases: *ModularSymbol*

Modular symbols attached to E using `eclib`.

Note that the normalization used within `eclib` differs from the normalization chosen here by a factor of 2 in the case of elliptic curves with negative discriminant (with one real component) since the convention there is to write

the above integral as $[r]^+x + [r]^-yi$, where the lattice is $\langle 2x, x + yi\rangle$, so that $\Omega^+ = 2x$ and $\Omega^- = 2yi$. We allow for this below.

INPUT:

- `E` – an elliptic curve
- `sign` – an integer, -1 or 1
- `nap` – (int, default 1000): the number of ap of E to use in determining the normalisation of the modular symbols.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_modular_symbols import␣
↪ModularSymbolECLIB
sage: E = EllipticCurve('11a1')
sage: M = ModularSymbolECLIB(E,+1)
sage: M
Modular symbol with sign 1 over Rational Field attached to
 Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field
sage: M(0)
1/5
sage: E = EllipticCurve('11a2')
sage: M = ModularSymbolECLIB(E,+1)
sage: M(0)
1
```

This is a rank 1 case with vanishing positive twists:

```
sage: E = EllipticCurve('121b1')
sage: M = ModularSymbolECLIB(E,+1)
sage: M(0)
0
sage: M(1/7)
1/2

sage: M = EllipticCurve('121d1').modular_symbol(implementation="eclib")
sage: M(0)
2

sage: E = EllipticCurve('15a1')
sage: [C.modular_symbol(implementation="eclib")(0) for C in E.isogeny_class()]
[1/4, 1/8, 1/4, 1/2, 1/8, 1/16, 1/2, 1]
```

Since Issue #10256, the interface for negative modular symbols in eclib is available:

```
sage: E = EllipticCurve('11a1')
sage: Mplus = E.modular_symbol(+1); Mplus
Modular symbol with sign 1 over Rational Field attached to
 Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field
sage: [Mplus(1/i) for i in [1..11]]
[1/5, -4/5, -3/10, 7/10, 6/5, 6/5, 7/10, -3/10, -4/5, 1/5, 0]
sage: Mminus = E.modular_symbol(-1); Mminus
Modular symbol with sign -1 over Rational Field attached to
 Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field
sage: [Mminus(1/i) for i in [1..11]]
[0, 0, 1/2, 1/2, 0, 0, -1/2, -1/2, 0, 0, 0]
```

The scaling factor relative to eclib's normalization is 1/2 for curves of negative discriminant:

```
sage: [E.discriminant() for E in cremona_curves([14])]
[-21952, 941192, -1835008, -28, 25088, 98]
sage: [E.modular_symbol()._scaling for E in cremona_curves([14])]
[1/2, 1, 1/2, 1/2, 1, 1]
```

TESTS (for Issue #10236):

```
sage: E = EllipticCurve('11a1')
sage: m = E.modular_symbol(implementation="eclib")
sage: m(1/7)
7/10
sage: m(0)
1/5
```

If `nap` is too small, the normalization in eclib used to be incorrect (see Issue #31317), but since `eclib` version v20210310 the value of `nap` is increased automatically by `eclib`:

```
sage: from sage.schemes.elliptic_curves.ell_modular_symbols import␣
↪ModularSymbolECLIB
sage: E = EllipticCurve('1590g1')
sage: m = ModularSymbolECLIB(E, sign=+1, nap=300)
sage: [m(a/5) for a in [1..4]]
[13/2, -13/2, -13/2, 13/2]
```

These values are correct, and increasing `nap` has no effect. The correct values may verified by the numerical implementation:

```
sage: m = ModularSymbolECLIB(E, sign=+1, nap=400)
sage: [m(a/5) for a in [1..4]]
[13/2, -13/2, -13/2, 13/2]
sage: m = E.modular_symbol(implementation='num')
sage: [m(a/5) for a in [1..4]]
[13/2, -13/2, -13/2, 13/2]
```

class sage.schemes.elliptic_curves.ell_modular_symbols.**ModularSymbolSage**(*E*, *sign*, *normalize='L_ratio'*)

Bases: *ModularSymbol*

Modular symbols attached to E using `sage`.

INPUT:

- `E` – an elliptic curve
- `sign` – an integer, -1 or 1
- `normalize` – either 'L_ratio' (default), 'period', or 'none'; For 'L_ratio', the modular symbol is correctly normalized by comparing it to the quotient of $L(E,1)$ by the least positive period for the curve and some small twists. The normalization 'period' uses the integral_period_map for modular symbols and is known to be equal to the above normalization up to the sign and a possible power of 2. For 'none', the modular symbol is almost certainly not correctly normalized, i.e. all values will be a fixed scalar multiple of what they should be. But the initial computation of the modular symbol is much faster, though evaluation of it after computing it won't be any faster.

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: from sage.schemes.elliptic_curves.ell_modular_symbols import␣
↪ModularSymbolSage
sage: M = ModularSymbolSage(E, +1)
sage: M
Modular symbol with sign 1 over Rational Field attached to
 Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field
sage: M(0)
1/5
sage: E = EllipticCurve('11a2')
sage: M = ModularSymbolSage(E, +1)
sage: M(0)
1
sage: M = ModularSymbolSage(E, -1)
sage: M(1/3)
1/2
```

This is a rank 1 case with vanishing positive twists. The modular symbol is adjusted by -2:

```
sage: E = EllipticCurve('121b1')
sage: M = ModularSymbolSage(E, -1, normalize='L_ratio')
sage: M(1/3)
1
sage: M._scaling
1

sage: M = EllipticCurve('121d1').modular_symbol(implementation="sage")
sage: M(0)
2
sage: M = EllipticCurve('121d1').modular_symbol(implementation="sage",
....:                                           normalize='none')
sage: M(0)
1

sage: E = EllipticCurve('15a1')
sage: [C.modular_symbol(implementation="sage", normalize='L_ratio')(0)
....:  for C in E.isogeny_class()]
[1/4, 1/8, 1/4, 1/2, 1/8, 1/16, 1/2, 1]
sage: [C.modular_symbol(implementation="sage", normalize='period')(0)
....:  for C in E.isogeny_class()]
[1/8, 1/16, 1/8, 1/4, 1/16, 1/32, 1/4, 1/2]
sage: [C.modular_symbol(implementation="sage", normalize='none')(0)
....:  for C in E.isogeny_class()]
[1, 1, 1, 1, 1, 1, 1, 1]
```

sage.schemes.elliptic_curves.ell_modular_symbols.**modular_symbol_space**(*E*, *sign*, *base_ring*, *bound=None*)

Creates the space of modular symbols of a given sign over a give base_ring, attached to the isogeny class of the elliptic curve `E`.

INPUT:

- `E` – an elliptic curve over **Q**
- `sign` – integer, -1, 0, or 1
- `base_ring` – ring

- `bound` – (default: None) maximum number of Hecke operators to use to cut out modular symbols factor. If None, use enough to provably get the correct answer.

OUTPUT: a space of modular symbols

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_modular_symbols import modular_symbol_
↪space
sage: E = EllipticCurve('11a1')
sage: M = modular_symbol_space(E, -1, GF(37))
sage: M
Modular Symbols space of dimension 1 for Gamma_0(11) of weight 2 with sign -1
 over Finite Field of size 37
```

18.21 Modular symbols by numerical integration

We describe here the method for computing modular symbols by numerical approximations of the integral of the modular form on a path between cusps.

More precisely, let E be an elliptic curve and f the newform associated to the isogeny class of E. If

$$\lambda(r \to r') = 2\pi i \int_r^{r'} f(\tau) d\tau$$

then the modular symbol $[r]^+$ is defined as the quotient of the real part of $\lambda(\infty \to r)$ by the least positive real period of E. Similarly for the negative modular symbol, it is the quotient of the imaginary part of the above by the smallest positive imaginary part of a period on the imaginary axis.

The theorem of Manin-Drinfeld shows that the modular symbols are rational numbers with small denominator. They are used for the computation of special values of the L-function of E twisted by Dirichlet characters and for the computation of p-adic L-functions.

ALGORITHM:

The implementation of modular symbols in eclib and directly in sage uses the algorithm described in Cremona's book [Cre1997] and Stein's book [St2007]. First the space of all modular symbols of the given level is computed, then the space corresponding to the given newform is determined. Even if these initial steps may take a while, the evaluation afterwards is instantaneous. All computations are done with rational numbers and hence are exact.

Instead the method used here (see [Wu2018] for details) is by evaluating the above integrals $\lambda(r \to r')$ by numerical approximation. Since we know precise bounds on the denominator, we can make rigorous estimates on the error to guarantee that the result is proven to be the correct rational number.

The paths over which we integrate are split up and Atkin-Lehner operators are used to compute the badly converging part of the integrals by using the Fourier expansion at other cusps than ∞.

Note: There is one assumption for the correctness of these computations: The Manin constant for the X_0-optimal curve should be 1 if the curve lies outside the Cremona tables. This is known for all curves in the Cremona table, but only conjectured for general curves.

EXAMPLES:

The most likely usage for the code is through the functions `modular_symbol` with implementation set to "num" and through `modular_symbol_numerical`:

```
sage: E = EllipticCurve("5077a1")
sage: M = E.modular_symbol(implementation="num")
sage: M(0)
0
sage: M(1/123)
4
sage: Mn = E.modular_symbol_numerical(sign=-1, prec=30)
sage: Mn(3/123)       # abs tol 1e-11
3.00000000000018
```

In more details. A numerical modular symbols `M` is created from an elliptic curve with a chosen `sign` (though the other sign will also be accessible, too):

```
sage: E = EllipticCurve([101,103])
sage: E.conductor()
35261176
sage: M = E.modular_symbol(implementation="num", sign=-1)
sage: M
Numerical modular symbol attached to
 Elliptic Curve defined by y^2 = x^3 + 101*x + 103 over Rational Field
```

We can then compute the value $[13/17]^-$ and $[1/17]^+$ by calling the function `M`. The value of $[0]^+ = 0$ tells us that the rank of this curve is positive:

```
sage: M(13/17)
-1/2
sage: M(1/17,sign=+1)
-3
sage: M(0, sign=+1)
0
```

One can compute the numerical approximation to these rational numbers to any proven binary precision:

```
sage: M.approximative_value(13/17, prec=2) # abs tol 1e-4
-0.500003172770455
sage: M.approximative_value(13/17, prec=4) # abs tol 1e-6
-0.500000296037388
sage: M.approximative_value(0, sign=+1, prec=6) # abs tol 1e-8
0.000000000000000
```

There are a few other things that one can do with `M`. The Manin symbol $M(c:d)$ for a point $(c:d)$ in the projective line can be computed.:

```
sage: M.manin_symbol(1,5)
-1
```

In some cases useful, there is a function that returns all $[a/m]^+$ for a fixed denominator m. This is rather quicker for small m than computing them individually:

```
sage: M.all_values_for_one_denominator(7)
{1/7: 0, 2/7: 3/2, 3/7: 3/2, 4/7: -3/2, 5/7: -3/2, 6/7: 0}
```

Finally a word of warning. The algorithm is fast when the cusps involved are unitary. If the curve is semistable, all cusps are unitary. But rational numbers with a prime p dividing the denominator once, but the conductor more than once, are very difficult. For instance for the above example, a seemingly harmless command like `M(1/2)` would take a very very long time to return a value. However it is possible to compute them for smaller conductors:

```
sage: E = EllipticCurve("664a1")
sage: M = E.modular_symbol(implementation="num")
sage: M(1/2)
0
```

The problem with non-unitary cusps is dealt with rather easily when one can twist to a semistable curve, like in this example:

```
sage: C = EllipticCurve("11a1")
sage: E = C.quadratic_twist(101)
sage: M = E.modular_symbol(implementation="num")
sage: M(1/101)
41
```

AUTHORS:

- Chris Wuthrich (2013-16)

class sage.schemes.elliptic_curves.mod_sym_num.**ModularSymbolNumerical**

Bases: object

This class assigning to an elliptic curve over **Q** a modular symbol. Unlike the other implementations this class does not precompute a basis for this space. Instead at each call, it evaluates the integral using numerical approximation. All bounds are very strictly implemented and the output is a correct proven rational number.

INPUT:

- E – an elliptic curve over the rational numbers.
- sign – either -1 or +1 (default). This sets the default value of sign throughout the class. Both are still accessible.

OUTPUT: a modular symbol

EXAMPLES:

```
sage: E = EllipticCurve("5077a1")
sage: M = E.modular_symbol(implementation="num")
sage: M(0)
0
sage: M(77/57)
-1
sage: M(33/37, -1)
2
sage: M = E.modular_symbol(sign=-1, implementation="num")
sage: M(2/7)
2

sage: from sage.schemes.elliptic_curves.mod_sym_num \
....: import ModularSymbolNumerical
sage: M = ModularSymbolNumerical(EllipticCurve("11a1"))
sage: M(1/3, -1)
1/2
sage: M(1/2)
-4/5
```

all_values_for_one_denominator(*m*, *sign=0*)

Given an integer m and a sign, this returns the modular symbols $[a/m]$ for all a coprime to m using partial sums. This is much quicker than computing them one by one.

This will only work if m is relatively small and if the cusps a/m are unitary.

INPUT:

- `m` – a natural number
- `sign` – optional either +1 or -1, or 0 (default), in which case the sign passed to the class is taken.

OUTPUT: a dictionary of fractions with denominator m giving rational numbers.

EXAMPLES:

```
sage: E = EllipticCurve('5077a1')
sage: M = E.modular_symbol(implementation="num")
sage: M.all_values_for_one_denominator(7)
{1/7: 3, 2/7: 0, 3/7: -3, 4/7: -3, 5/7: 0, 6/7: 3}
sage: [M(a/7) for a in [1..6]]
[3, 0, -3, -3, 0, 3]
sage: M.all_values_for_one_denominator(3,-1)
{1/3: 4, 2/3: -4}

sage: E = EllipticCurve('11a1')
sage: M = E.modular_symbol(implementation="num")
sage: M.all_values_for_one_denominator(12)
{1/12: 1/5, 5/12: -23/10, 7/12: -23/10, 11/12: 1/5}
sage: M.all_values_for_one_denominator(12, -1)
{1/12: 0, 5/12: 1/2, 7/12: -1/2, 11/12: 0}

sage: E = EllipticCurve('20a1')
sage: M = E.modular_symbol(implementation="num")
sage: M.all_values_for_one_denominator(4)
{1/4: 0, 3/4: 0}
sage: M.all_values_for_one_denominator(8)
{1/8: 1/2, 3/8: -1/2, 5/8: -1/2, 7/8: 1/2}
```

approximative_value(*r*, *sign=0*, *prec=20*, *use_twist=True*)

The numerical modular symbol evaluated at rational.

It returns a real number, which should be equal to a rational number to the given binary precision `prec`. In practice the precision is often much higher. See the examples below.

INPUT:

- `r` – a rational (or integer)
- `sign` – optional either +1 or -1, or 0 (default), in which case the sign passed to the class is taken.
- `prec` – an integer (default 20)
- `use_twist` – True (default) allows to use a quadratic twist of the curve to lower the conductor.

OUTPUT: a real number

EXAMPLES:

```
sage: E = EllipticCurve("5077a1")
sage: M = E.modular_symbol(implementation="num")
sage: M.approximative_value(123/567)  # abs tol 1e-11
-4.00000000000845
sage: M.approximative_value(123/567,prec=2) # abs tol 1e-9
-4.00002815242902
```

(continues on next page)

(continued from previous page)

```
sage: E = EllipticCurve([11,88])
sage: E.conductor()
1715296
sage: M = E.modular_symbol(implementation="num")
sage: M.approximative_value(0,prec=2)    # abs tol 1e-11
-0.00001176374317982166
sage: M.approximative_value(1/7,prec=2)  # abs tol 1e-11
0.999981178147778
sage: M.approximative_value(1/7,prec=10) # abs tol 1e-11
0.999999972802649
```

clear_cache()

Clear the cached values in all methods of this class

EXAMPLES:

```
sage: E = EllipticCurve("11a1")
sage: M = E.modular_symbol(implementation="num")
sage: M(0)
1/5
sage: M.clear_cache()
sage: M(0)
1/5
```

elliptic_curve()

Return the elliptic curve of this modular symbol.

EXAMPLES:

```
sage: E = EllipticCurve("15a4")
sage: M = E.modular_symbol(implementation="num")
sage: M.elliptic_curve()
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 + 35*x - 28 over Rational␣
↪Field
```

manin_symbol(*u*, *v*, *sign=0*)

Given a pair (u, v) presenting a point in $\mathbb{P}^1(\mathbb{Z}/N\mathbb{Z})$ and hence a coset of $\Gamma_0(N)$, this computes the value of the Manin symbol $M(u : v)$.

INPUT:

- `u` – an integer
- `v` – an integer such that $(u : v)$ is a projective point modulo N
- `sign` – optional either +1 or -1, or 0 (default), in which case the sign passed to the class is taken.

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: M = E.modular_symbol(implementation="num")
sage: M.manin_symbol(1,3)
-1/2
sage: M.manin_symbol(1,3, sign=-1)
-1/2
sage: M.manin_symbol(1,5)
1
sage: M.manin_symbol(1,5)
```

(continues on next page)

(continued from previous page)

```
1

sage: E = EllipticCurve('14a1')
sage: M = E.modular_symbol(implementation="num")
sage: M.manin_symbol(1,2)
-1/2
sage: M.manin_symbol(17,6)
-1/2
sage: M.manin_symbol(-1,12)
-1/2
```

transportable_symbol(*r*, *rr*, *sign=0*)

Return the symbol $[r']^+ - [r]^+$ where $r' = \gamma(r)$ for some $\gamma \in \Gamma_0(N)$. These symbols can be computed by transporting the path into the upper half plane close to one of the unitary cusps. Here we have implemented it only to move close to $i\infty$ and 0.

INPUT:

- `r` and `rr` – two rational numbers
- `sign` – optional either +1 or -1, or 0 (default), in which case the sign passed to the class is taken.

OUTPUT: a rational number

EXAMPLES:

```
sage: E = EllipticCurve("11a1")
sage: M = E.modular_symbol(implementation="num")
sage: M.transportable_symbol(0/1,-2/7)
-1/2

sage: E = EllipticCurve("37a1")
sage: M = E.modular_symbol(implementation="num")
sage: M.transportable_symbol(0/1,-1/19)
0
sage: M.transportable_symbol(0/1,-1/19,-1)
0

sage: E = EllipticCurve("5077a1")
sage: M = E.modular_symbol(implementation="num")
sage: M.transportable_symbol(0/1,-35/144)
-3
sage: M.transportable_symbol(0/1,-35/144,-1)
0
sage: M.transportable_symbol(0/1, -7/31798)
0
sage: M.transportable_symbol(0/1, -7/31798, -1)
-5
```

18.22 L-series for elliptic curves

AUTHORS:

- Simon Spicer (2014-08-15): Added LFunctionZeroSum class interface method
- Jeroen Demeyer (2013-10-17): Compute L series with arbitrary precision instead of floats.
- William Stein et al. (2005 and later)

class sage.schemes.elliptic_curves.lseries_ell.**Lseries_ell**(E)

Bases: SageObject

An elliptic curve L-series.

L1_vanishes()

Returns whether or not $L(E,1) = 0$. The result is provably correct if the Manin constant of the associated optimal quotient is <= 2. This hypothesis on the Manin constant is true for all curves of conductor <= 40000 (by Cremona) and all semistable curves (i.e., squarefree conductor).

ALGORITHM: see *L_ratio()*.

EXAMPLES:

```
sage: E = EllipticCurve([0, -1, 1, -10, -20])    # 11A  = X_0(11)
sage: E.lseries().L1_vanishes()
False
sage: E = EllipticCurve([0, -1, 1, 0, 0])        # X_1(11)
sage: E.lseries().L1_vanishes()
False
sage: E = EllipticCurve([0, 0, 1, -1, 0])        # 37A  (rank 1)
sage: E.lseries().L1_vanishes()
True
sage: E = EllipticCurve([0, 1, 1, -2, 0])        # 389A (rank 2)
sage: E.lseries().L1_vanishes()
True
sage: E = EllipticCurve([0, 0, 1, -38, 90])      # 361A (CM curve))
sage: E.lseries().L1_vanishes()
True
sage: E = EllipticCurve([0,-1,1,-2,-1])          # 141C (13-isogeny)
sage: E.lseries().L1_vanishes()
False
```

AUTHORS: William Stein, 2005-04-20.

L_ratio()

Return the ratio $L(E,1)/\Omega$ as an exact rational number.

The result is *provably* correct if the Manin constant of the associated optimal quotient is ≤ 2. This hypothesis on the Manin constant is true for all semistable curves (i.e., squarefree conductor), by a theorem of Mazur from his *Rational Isogenies of Prime Degree* paper.

EXAMPLES:

```
sage: E = EllipticCurve([0, -1, 1, -10, -20])    # 11A  = X_0(11)
sage: E.lseries().L_ratio()
1/5
sage: E = EllipticCurve([0, -1, 1, 0, 0])        # X_1(11)
sage: E.lseries().L_ratio()
1/25
```

(continues on next page)

(continued from previous page)

```
sage: E = EllipticCurve([0, 0, 1, -1, 0])        # 37A  (rank 1)
sage: E.lseries().L_ratio()
0
sage: E = EllipticCurve([0, 1, 1, -2, 0])        # 389A (rank 2)
sage: E.lseries().L_ratio()
0
sage: E = EllipticCurve([0, 0, 1, -38, 90])      # 361A (CM curve))
sage: E.lseries().L_ratio()
0
sage: E = EllipticCurve([0,-1,1,-2,-1])          # 141C (13-isogeny)
sage: E.lseries().L_ratio()
1
sage: E = EllipticCurve(RationalField(), [1, 0, 0, 1/24624, 1/886464])
sage: E.lseries().L_ratio()
2
```

See Issue #3651 and Issue #15299:

```
sage: EllipticCurve([0,0,0,-193^2,0]).sha().an()
4
sage: EllipticCurve([1, 0, 1, -131, 558]).sha().an()  # long time
1.00000000000000
```

ALGORITHM: Compute the root number. If it is -1 then $L(E, s)$ vanishes to odd order at 1, hence vanishes. If it is +1, use a result about modular symbols and Mazur's *Rational Isogenies* paper to determine a provably correct bound (assuming Manin constant is <= 2) so that we can determine whether $L(E, 1) = 0$.

AUTHORS: William Stein, 2005-04-20.

at1(*k=None*, *prec=None*)

Compute $L(E, 1)$ using k terms of the series for $L(E, 1)$ as explained in Section 7.5.3 of Henri Cohen's book *A Course in Computational Algebraic Number Theory*. If the argument k is not specified, then it defaults to $\sqrt{N}$, where N is the conductor.

INPUT:

- `k` – number of terms of the series. If zero or `None`, use $k = \sqrt{N}$, where N is the conductor.
- `prec` – numerical precision in bits. If zero or `None`, use a reasonable automatic default.

OUTPUT:

A tuple of real numbers `(L, err)` where `L` is an approximation for $L(E, 1)$ and `err` is a bound on the error in the approximation.

This function is disjoint from the PARI pari:elllseries command, which is for a similar purpose. To use that command (via the PARI C library), simply type `E.pari_mincurve().elllseries(1)`.

ALGORITHM:

- Compute the root number eps. If it is -1, return 0.
- Compute the Fourier coefficients a_n, for n up to and including k.
- Compute the sum

$$2 \cdot \sum_{n=1}^{k} \frac{a_n}{n} \cdot \exp(-2 * pi * n/\sqrt{N}),$$

where N is the conductor of E.

- Compute a bound on the tail end of the series, which is

$$2e^{-2\pi(k+1)/\sqrt{N}}/(1-e^{-2\pi/\sqrt{N}}).$$

For a proof see [Grigov-Jorza-Patrascu-Patrikis-Stein].

EXAMPLES:

```
sage: L, err = EllipticCurve('11a1').lseries().at1()
sage: L, err
(0.253804, 0.000181444)
sage: parent(L)
Real Field with 24 bits of precision
sage: E = EllipticCurve('37b')
sage: E.lseries().at1()
(0.7257177, 0.000800697)
sage: E.lseries().at1(100)
(0.72568106193615278233620554102639654873675603361763, 1.52469e-45)
sage: L,err = E.lseries().at1(100, prec=128)
sage: L
0.72568106193615278233620554102639654873
sage: parent(L)
Real Field with 128 bits of precision
sage: err
1.70693e-37
sage: parent(err)
Real Field with 24 bits of precision and rounding RNDU
```

Rank 1 through 3 elliptic curves:

```
sage: E = EllipticCurve('37a1')
sage: E.lseries().at1()
(0.0000000, 0.000000)
sage: E = EllipticCurve('389a1')
sage: E.lseries().at1()
(-0.001769566, 0.00911776)
sage: E = EllipticCurve('5077a1')
sage: E.lseries().at1()
(0.0000000, 0.000000)
```

deriv_at1(*k=None*, *prec=None*)

Compute $L'(E,1)$ using k terms of the series for $L'(E,1)$, under the assumption that $L(E,1)=0$.

The algorithm used is from Section 7.5.3 of Henri Cohen's book *A Course in Computational Algebraic Number Theory*.

INPUT:

- `k` – number of terms of the series. If zero or `None`, use $k=\sqrt{N}$, where N is the conductor.
- `prec` – numerical precision in bits. If zero or `None`, use a reasonable automatic default.

OUTPUT:

A tuple of real numbers `(L1, err)` where `L1` is an approximation for $L'(E,1)$ and `err` is a bound on the error in the approximation.

Warning: This function only makes sense if $L(E)$ has positive order of vanishing at 1, or equivalently if $L(E,1)=0$.

ALGORITHM:

- Compute the root number eps. If it is 1, return 0.
- Compute the Fourier coefficients a_n, for n up to and including k.
- Compute the sum

$$2 \cdot \sum_{n=1}^{k} (a_n/n) \cdot E_1(2\pi n/\sqrt{N}),$$

  where N is the conductor of E, and E_1 is the exponential integral function.

- Compute a bound on the tail end of the series, which is

$$2e^{-2\pi(k+1)/\sqrt{N}}/(1 - e^{-2\pi/\sqrt{N}}).$$

  For a proof see [Grigorov-Jorza-Patrascu-Patrikis-Stein]. This is exactly the same as the bound for the approximation to $L(E, 1)$ produced by *at1()*.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.lseries().deriv_at1()                                                  #␣
↪needs sage.symbolic
(0.3059866, 0.000801045)
sage: E.lseries().deriv_at1(100)                                               #␣
↪needs sage.symbolic
(0.305999773834052301820483683321676474452637774590771998..., 1.52493e-45)
sage: E.lseries().deriv_at1(1000)                                              #␣
↪needs sage.symbolic
(0.305999773834052301820483683321676474452637774590771998..., 2.75031e-449)
```

With less numerical precision, the error is bounded by numerical accuracy:

```
sage: # needs sage.symbolic
sage: L, err = E.lseries().deriv_at1(100, prec=64)
sage: L, err
(0.305999773834052302, 5.55318e-18)
sage: parent(L)
Real Field with 64 bits of precision
sage: parent(err)
Real Field with 24 bits of precision and rounding RNDU
```

Rank 2 and rank 3 elliptic curves:

```
sage: E = EllipticCurve('389a1')
sage: E.lseries().deriv_at1()                                                  #␣
↪needs sage.symbolic
(0.0000000, 0.000000)
sage: E = EllipticCurve((1, 0, 1, -131, 558))  # curve 59450i1
sage: E.lseries().deriv_at1()                                                  #␣
↪needs sage.symbolic
(-0.00010911444, 0.142428)
sage: E.lseries().deriv_at1(4000)                                              #␣
↪needs sage.symbolic
(6.990...e-50, 1.31318e-43)
```

dokchitser(*prec=53*, *max_imaginary_part=0*, *max_asymp_coeffs=40*, *algorithm=None*)

Return an interface for computing with the L-series of this elliptic curve.

This provides a way to compute Taylor expansions and higher derivatives of L-series.

INPUT:

- `prec` – optional integer (default 53) bits precision
- `max_imaginary_part` – optional real number (default 0)
- `max_asymp_coeffs` – optional integer (default 40)
- `algorithm` – optional string: 'gp' (default), 'pari' or 'magma'

If algorithm is "gp", this returns an interface to Tim Dokchitser's program for computing with the L-functions.

If algorithm is "pari", this returns instead an interface to Pari's own general implementation of L-functions.

Note: If algorithm='magma', then the precision is in digits rather than bits and the object returned is a Magma L-series, which has different functionality from the Sage L-series.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: L = E.lseries().dokchitser()
sage: L(2)
0.381575408260711
sage: L = E.lseries().dokchitser(algorithm='magma')          # optional - magma
sage: L.Evaluate(2)                                          # optional - magma
0.381575408260711211293710409580086636677097533988921 16
```

If the curve has too large a conductor, it is not possible to compute with the L-series using this command. Instead a `RuntimeError` is raised:

```
sage: e = EllipticCurve([1,1,0,-63900,-1964465932632])
sage: L = e.lseries().dokchitser(15, algorithm='gp')
Traceback (most recent call last):
...
RuntimeError: unable to create L-series, due to precision or other limits in␣
↪PARI
```

Using the "pari" algorithm:

```
sage: E = EllipticCurve('37a')
sage: L = E.lseries().dokchitser(algorithm="pari")
sage: L(2)
0.381575408260711
```

elliptic_curve()

Return the elliptic curve that this L-series is attached to.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: L = E.lseries()
sage: L.elliptic_curve ()
Elliptic Curve defined by y^2 + y = x^3 + x^2 - 2*x over Rational Field
```

sympow(*n*, *prec*)

Return $L(Sym^{(n)}(E, \text{edge}))$ to `prec` digits of precision.

INPUT:

- `n` – integer
- `prec` – integer

OUTPUT:

- (string) – real number to `prec` digits of precision as a string.

Note: Before using this function for the first time for a given `n`, you may have to type `sympow('-new_data <n>')`, where `<n>` is replaced by your value of `n`. This command takes a long time to run.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: a = E.lseries().sympow(2,16)    # not tested - requires precomputing
↪"sympow('-new_data 2')"
sage: a                               # not tested
'2.492262044273650E+00'
sage: RR(a)                           # not tested
2.49226204427365
```

sympow_derivs(*n*, *prec*, *d*)

Return 0-th to d-th derivatives of $L(Sym^{(n)}(E, \text{edge}))$ to `prec` digits of precision.

INPUT:

- n – integer
- prec – integer
- d – integer

OUTPUT:

- a string, exactly as output by sympow

Note: To use this function you may have to run a few commands like `sympow('-new_data 1d2')`, each which takes a few minutes. If this function fails it will indicate what commands have to be run.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: print(E.lseries().sympow_derivs(1,16,2))     # not tested -- requires␣
↪precomputing "sympow('-new_data 2')"
sympow 1.018 RELEASE  (c) Mark Watkins --- see README and COPYING for details
Minimal model of curve  is [0,0,1,-1,0]
At 37: Inertia Group is  C1 MULTIPLICATIVE REDUCTION
Conductor is 37
sp 1: Conductor at 37 is 1+0, root number is 1
sp 1: Euler factor at 37 is 1+1*x
1st sym power conductor is 37, global root number is -1
NT 1d0: 35
```

(continues on next page)

(continued from previous page)

```
NT 1d1: 32
NT 1d2: 28
Maximal number of terms is 35
Done with small primes 1049
Computed:  1d0  1d1  1d2
Checked out:  1d1
 1n0: 3.837774351482055E-01
 1w0: 3.777214305638848E-01
 1n1: 3.059997738340522E-01
 1w1: 3.059997738340524E-01
 1n2: 1.519054910249753E-01
 1w2: 1.545605024269432E-01
```

taylor_series(*a=1*, *prec=53*, *series_prec=6*, *var='z'*)

Return the Taylor series of this L-series about a to the given precision (in bits) and the number of terms.

The output is a series in `var`, where you should view `var` as equal to $s - a$. Thus this function returns the formal power series whose coefficients are $L^{(n)}(a)/n!$.

INPUT:

- `a` – complex number
- `prec` – integer, precision in bits (default 53)
- `series_prec` – integer (default 6)
- `var` – variable (default 'z')

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: L = E.lseries()
sage: L.taylor_series(series_prec=3)    # abs tol 1e-14
-1.27685190980159e-23 + (7.23588070754027e-24)*z + 0.759316500288427*z^2 +␣
↪O(z^3)  # 32-bit
1.34667664606157e-19 + (-7.63157535163667e-20)*z + 0.759316500288427*z^2 +␣
↪O(z^3)  # 64-bit
```

twist_values(*s*, *dmin*, *dmax*)

Return values of $L(E, s, \chi_d)$ for each quadratic character χ_d for $d_{\text{min}} \leq d \leq d_{\text{max}}$.

Note: The L-series is normalized so that the center of the critical strip is 1.

INPUT:

- `s` – complex numbers
- `dmin` – integer
- `dmax` – integer

OUTPUT:

- list of pairs $(d, L(E, s, \chi_d))$

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: vals = E.lseries().twist_values(1, -12, -4)
sage: vals[0][0]
-11
sage: vals[0][1] # abs tol 1e-8
1.47824342 + 0.0*I
sage: vals[1][0]
-8
sage: vals[1][1] # abs tol 1e-8
0.0 + 0.0*I
sage: vals[2][0]
-7
sage: vals[2][1] # abs tol 1e-8
1.85307619 + 0.0*I
sage: vals[3][0]
-4
sage: vals[3][1] # abs tol 1e-8
2.45138938 + 0.0*I
sage: F = E.quadratic_twist(-8)
sage: F.rank()
1
sage: F = E.quadratic_twist(-7)
sage: F.rank()
0
```

twist_zeros(*n*, *dmin*, *dmax*)

Return first n real parts of nontrivial zeros of $L(E, s, \chi_d)$ for each quadratic character χ_d with $d_{\min} \leq d \leq d_{\max}$.

Note: The L-series is normalized so that the center of the critical strip is 1.

INPUT:

- `n` – integer
- `dmin` – integer
- `dmax` – integer

OUTPUT:

- **dict – keys are the discriminants d, and** values are list of corresponding zeros.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.lseries().twist_zeros(3, -4, -3)          # long time
{-4: [1.60813783, 2.96144840, 3.89751747], -3: [2.06170900, 3.48216881, 4.
↪45853219]}
```

values_along_line(*s0*, *s1*, *number_samples*)

Return values of $L(E, s)$ at `number_samples` equally-spaced sample points along the line from s_0 to s_1 in the complex plane.

Note: The L-series is normalized so that the center of the critical strip is 1.

INPUT:

- s0, s1 – complex numbers
- number_samples – integer

OUTPUT:

list – list of pairs (s, $L(E,s)$), where the s are
equally spaced sampled points on the line from s0 to s1.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.lseries().values_along_line(1, 0.5 + 20*I, 5)
[(0.500000000, ...),
 (0.400000000 + 4.00000000*I, 3.31920245 - 2.60028054*I),
 (0.300000000 + 8.00000000*I, -0.886341185 - 0.422640337*I),
 (0.200000000 + 12.0000000*I, -3.50558936 - 0.108531690*I),
 (0.100000000 + 16.0000000*I, -3.87043288 - 1.88049411*I)]
```

zero_sums (*N=None*)

Return an LFunctionZeroSum class object for efficient computation of sums over the zeros of self.

This can be used to bound analytic rank from above without having to compute with the L-series directly.

INPUT:

- N – (default: None) If not None, the conductor of the elliptic curve attached to self. This is passable so that zero sum computations can be done on curves for which the conductor has been precomputed.

OUTPUT:

A LFunctionZeroSum_EllipticCurve instance.

EXAMPLES:

```
sage: E = EllipticCurve("5077a")
sage: E.lseries().zero_sums()
Zero sum estimator for L-function attached to
 Elliptic Curve defined by y^2 + y = x^3 - 7*x + 6 over Rational Field
```

zeros (*n*)

Return the imaginary parts of the first n nontrivial zeros on the critical line of the L-function in the upper half plane, as 32-bit reals.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.lseries().zeros(2)
[0.000000000, 5.00317001]

sage: a = E.lseries().zeros(20)             # long time
sage: point([(1,x) for x in a])             # graph  (long time)
Graphics object consisting of 1 graphics primitive
```

AUTHORS: Uses Rubinstein's L-functions calculator.

zeros_in_interval (*x*, *y*, *stepsize*)

Return the imaginary parts of (most of) the nontrivial zeros on the critical line $\Re(s) = 1$ with positive imaginary part between x and y, along with a technical quantity for each.

INPUT:

- `x` – positive floating point number
- `y` – positive floating point number
- `stepsize` – positive floating point number

OUTPUT:

- list of pairs `(zero, S(T))`.

Rubinstein writes: The first column outputs the imaginary part of the zero, the second column a quantity related to `S(T)` (it increases roughly by 2 whenever a sign change, i.e. pair of zeros, is missed). Higher up the critical strip you should use a smaller stepsize so as not to miss zeros.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.lseries().zeros_in_interval(6, 10, 0.1)      # long time
[(6.87039122, 0.248922780), (8.01433081, -0.140168533), (9.93309835, -0.
↪129943029)]
```

18.23 Heegner points on elliptic curves over the rational numbers

AUTHORS:

- William Stein (August 2009)– most of the initial version
- Robert Bradshaw (July 2009) – an early version of some specific code

EXAMPLES:

```
sage: E = EllipticCurve('433a')
sage: P = E.heegner_point(-8,3)
sage: z = P.point_exact(201); z
(-4/3 : 1/9*a : 1)
sage: parent(z)
Abelian group of points on Elliptic Curve defined by y^2 + x*y = x^3 + 1
 over Number Field in a with defining polynomial x^2 - 12*x + 111
sage: parent(z[0]).discriminant()
-3
sage: E.quadratic_twist(-3).rank()
1
sage: K.<a> = QuadraticField(-8)
sage: K.factor(3)
(Fractional ideal (1/2*a + 1)) * (Fractional ideal (-1/2*a + 1))
```

Next try an inert prime:

```
sage: K.factor(5)
Fractional ideal (5)
sage: P = E.heegner_point(-8,5)
sage: z = P.point_exact(300)
sage: z[0].charpoly().factor()
(x^6 + x^5 - 1/4*x^4 + 19/10*x^3 + 31/20*x^2 - 7/10*x + 49/100)^2
sage: z[1].charpoly().factor()
x^12 - x^11 + 6/5*x^10 - 33/40*x^9 - 89/320*x^8 + 3287/800*x^7
 - 5273/1600*x^6 + 993/4000*x^5 + 823/320*x^4 - 2424/625*x^3
 + 12059/12500*x^2 + 3329/25000*x + 123251/250000
```

(continues on next page)

(continued from previous page)

```
sage: f = P.x_poly_exact(300); f
x^6 + x^5 - 1/4*x^4 + 19/10*x^3 + 31/20*x^2 - 7/10*x + 49/100
sage: f.discriminant().factor()
-1 * 2^-9 * 5^-9 * 7^2 * 281^2 * 1021^2
```

We find some Mordell-Weil generators in the rank 1 case using Heegner points:

```
sage: E = EllipticCurve('43a'); P = E.heegner_point(-7)
sage: P.x_poly_exact()
x
sage: z = P.point_exact(); z == E(0,0,1) or -z == E(0,0,1)
True

sage: E = EllipticCurve('997a')
sage: E.rank()
1
sage: E.heegner_discriminants_list(10)
[-19, -23, -31, -35, -39, -40, -52, -55, -56, -59]
sage: P = E.heegner_point(-19)
sage: P.x_poly_exact()
x - 141/49
sage: z = P.point_exact(); z == E(141/49, -162/343, 1)  or -z == E(141/49, -162/343,␣
↪1)
True
```

Here we find that the Heegner point generates a subgroup of index 3:

```
sage: E = EllipticCurve('92b1')
sage: E.heegner_discriminants_list(1)
[-7]
sage: P = E.heegner_point(-7)
sage: z = P.point_exact(); z == E(0, 1, 1)  or -z == E(0, 1, 1)
True
sage: E.regulator()
0.0498083972980648
sage: z.height()
0.448275575682583
sage: P = E(1,1); P # a generator
(1 : 1 : 1)
sage: -3*P
(0 : 1 : 1)
sage: E.tamagawa_product()
3
```

The above is consistent with the following analytic computation:

```
sage: E.heegner_index(-7)
3.0000?
```

class sage.schemes.elliptic_curves.heegner.**GaloisAutomorphism**(*parent*)

Bases: SageObject

An abstract automorphism of a ring class field.

Todo: make *GaloisAutomorphism* derive from GroupElement, so that one gets powers for free, etc.

---

domain()

Return the domain of this automorphism.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: G = E.heegner_point(-7,5).ring_class_field().galois_group()
sage: s = G.complex_conjugation()
sage: s.domain()
Ring class field extension of QQ[sqrt(-7)] of conductor 5
```

parent()

Return the parent of this automorphism, which is a Galois group of a ring class field.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: G = E.heegner_point(-7,5).ring_class_field().galois_group()
sage: s = G.complex_conjugation()
sage: s.parent()
Galois group of Ring class field extension of QQ[sqrt(-7)] of conductor 5
```

class sage.schemes.elliptic_curves.heegner.**GaloisAutomorphismComplexConjugation**(*parent*)

Bases: *GaloisAutomorphism*

The complex conjugation automorphism of a ring class field.

EXAMPLES:

```
sage: G = heegner_point(37,-7,5).ring_class_field().galois_group()
sage: conj = G.complex_conjugation()
sage: conj
Complex conjugation automorphism of Ring class field extension
 of QQ[sqrt(-7)] of conductor 5
sage: conj.domain()
Ring class field extension of QQ[sqrt(-7)] of conductor 5
```

order()

EXAMPLES:

```
sage: G = heegner_point(37,-7,5).ring_class_field().galois_group()
sage: conj = G.complex_conjugation()
sage: conj.order()
2
```

class sage.schemes.elliptic_curves.heegner.**GaloisAutomorphismQuadraticForm**(*parent*, *quadratic_form*, *alpha=None*)

Bases: *GaloisAutomorphism*

An automorphism of a ring class field defined by a quadratic form.

EXAMPLES:

```
sage: H = heegner_points(389,-20,3)
sage: sigma = H.ring_class_field().galois_group(H.quadratic_field())[0]; sigma
Class field automorphism defined by x^2 + 45*y^2
sage: type(sigma)
<class 'sage.schemes.elliptic_curves.heegner.GaloisAutomorphismQuadraticForm'>
sage: loads(dumps(sigma)) == sigma
True
```

alpha()

Optional data that specified element corresponding element of $(\mathcal{O}_K/c\mathcal{O}_K)^*/(\mathbf{Z}/c\mathbf{Z})^*$, via class field theory.

This is a generator of the ideal corresponding to this automorphism.

EXAMPLES:

```
sage: K3 = heegner_points(389,-52,3).ring_class_field()
sage: K1 = heegner_points(389,-52,1).ring_class_field()
sage: G = K3.galois_group(K1)
sage: orb = sorted([g.alpha() for g in G]); orb # random (the sign depends on␣
↪the database being installed or not)
[1, 1/2*sqrt_minus_52 + 1, -1/2*sqrt_minus_52, 1/2*sqrt_minus_52 - 1]
sage: sorted([x^2 for x in orb]) # this is just for testing
[-13, -sqrt_minus_52 - 12, sqrt_minus_52 - 12, 1]

sage: K5 = heegner_points(389,-52,5).ring_class_field()
sage: K1 = heegner_points(389,-52,1).ring_class_field()
sage: G = K5.galois_group(K1)
sage: orb = sorted([g.alpha() for g in G]); orb # random (the sign depends on␣
↪the database being installed or not)
[1, -1/2*sqrt_minus_52, 1/2*sqrt_minus_52 + 1, 1/2*sqrt_minus_52 - 1,
 1/2*sqrt_minus_52 - 2, -1/2*sqrt_minus_52 - 2]
sage: sorted([x^2 for x in orb]) # just for testing
[-13, -sqrt_minus_52 - 12, sqrt_minus_52 - 12,
 -2*sqrt_minus_52 - 9, 2*sqrt_minus_52 - 9, 1]
```

ideal()

Return ideal of ring of integers of quadratic imaginary field corresponding to this quadratic form. This is the ideal

$$I = \left(A, \frac{-B+c\sqrt{D}}{2}\right)\mathcal{O}_K.$$

EXAMPLES:

```
sage: E = EllipticCurve('389a'); F= E.heegner_point(-20,3).ring_class_field()
sage: G = F.galois_group(F.quadratic_field())
sage: G[1].ideal()
Fractional ideal (2, 1/2*sqrt_minus_20 + 1)
sage: [s.ideal().gens() for s in G]
[(1, 3/2*sqrt_minus_20), (2, 3/2*sqrt_minus_20 - 1),
 (5, 3/2*sqrt_minus_20), (7, 3/2*sqrt_minus_20 - 2)]
```

order()

Return the multiplicative order of this Galois group automorphism.

EXAMPLES:

```
sage: K3 = heegner_points(389,-52,3).ring_class_field()
sage: K1 = heegner_points(389,-52,1).ring_class_field()
```

(continues on next page)

(continued from previous page)

```
sage: G = K3.galois_group(K1)
sage: sorted([g.order() for g in G])
[1, 2, 4, 4]
sage: K5 = heegner_points(389,-52,5).ring_class_field()
sage: K1 = heegner_points(389,-52,1).ring_class_field()
sage: G = K5.galois_group(K1)
sage: sorted([g.order() for g in G])
[1, 2, 3, 3, 6, 6]
```

p1_element()

Return element of the projective line corresponding to this automorphism.

This only makes sense if this automorphism is in the Galois group Gal(K_c/K_1).

EXAMPLES:

```
sage: K3 = heegner_points(389,-52,3).ring_class_field()
sage: K1 = heegner_points(389,-52,1).ring_class_field()
sage: G = K3.galois_group(K1)
sage: sorted([g.p1_element() for g in G])
[(0, 1), (1, 0), (1, 1), (1, 2)]

sage: K5 = heegner_points(389,-52,5).ring_class_field()
sage: K1 = heegner_points(389,-52,1).ring_class_field()
sage: G = K5.galois_group(K1)
sage: sorted([g.p1_element() for g in G])
[(0, 1), (1, 0), (1, 1), (1, 2), (1, 3), (1, 4)]
```

quadratic_form()

Return reduced quadratic form corresponding to this Galois automorphism.

EXAMPLES:

```
sage: H = heegner_points(389,-20,3); s = H.ring_class_field().galois_group(H.
↪quadratic_field())[0]
sage: s.quadratic_form()
x^2 + 45*y^2
```

class sage.schemes.elliptic_curves.heegner.**GaloisGroup**(*field*, *base=Rational Field*)

Bases: SageObject

A Galois group of a ring class field.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: G = E.heegner_point(-7,5).ring_class_field().galois_group(); G
Galois group of Ring class field extension of QQ[sqrt(-7)] of conductor 5
sage: G.field()
Ring class field extension of QQ[sqrt(-7)] of conductor 5
sage: G.cardinality()
12
sage: G.complex_conjugation()
Complex conjugation automorphism of Ring class field extension of QQ[sqrt(-7)]
 of conductor 5
```

base_field()

Return the base field, which the field fixed by all the automorphisms in this Galois group.

EXAMPLES:

```
sage: x = heegner_point(37,-7,5)
sage: Kc = x.ring_class_field(); Kc
Ring class field extension of QQ[sqrt(-7)] of conductor 5
sage: K = x.quadratic_field()
sage: G = Kc.galois_group(); G
Galois group of Ring class field extension of QQ[sqrt(-7)] of conductor 5
sage: G.base_field()
Rational Field
sage: G.cardinality()
12
sage: Kc.absolute_degree()
12
sage: G = Kc.galois_group(K); G
Galois group of Ring class field extension of QQ[sqrt(-7)] of conductor 5
 over Number Field in sqrt_minus_7 with defining polynomial x^2 + 7
  with sqrt_minus_7 = 2.645751311064591?*I
sage: G.cardinality()
6
sage: G.base_field()
Number Field in sqrt_minus_7 with defining polynomial x^2 + 7
 with sqrt_minus_7 = 2.645751311064591?*I
sage: G = Kc.galois_group(Kc); G
Galois group of Ring class field extension of QQ[sqrt(-7)] of conductor 5
 over Ring class field extension of QQ[sqrt(-7)] of conductor 5
sage: G.cardinality()
1
sage: G.base_field()
Ring class field extension of QQ[sqrt(-7)] of conductor 5
```

cardinality()

Return the cardinality of this Galois group.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: G = E.heegner_point(-7,5).ring_class_field().galois_group(); G
Galois group of Ring class field extension of QQ[sqrt(-7)] of conductor 5
sage: G.cardinality()
12
sage: G = E.heegner_point(-7).ring_class_field().galois_group()
sage: G.cardinality()
2
sage: G = E.heegner_point(-7,55).ring_class_field().galois_group()
sage: G.cardinality()
120
```

complex_conjugation()

Return the automorphism of `self` determined by complex conjugation. The base field must be the rational numbers.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: G = E.heegner_point(-7,5).ring_class_field().galois_group()
sage: G.complex_conjugation()
```

(continues on next page)

(continued from previous page)

```
Complex conjugation automorphism of Ring class field extension
 of QQ[sqrt(-7)] of conductor 5
```

field()

Return the ring class field that this Galois group acts on.

EXAMPLES:

```
sage: G = heegner_point(389,-7,5).ring_class_field().galois_group()
sage: G.field()
Ring class field extension of QQ[sqrt(-7)] of conductor 5
```

is_kolyvagin()

Return `True` if conductor c is prime to the discriminant of the quadratic field, c is squarefree and each prime dividing c is inert.

EXAMPLES:

```
sage: K5 = heegner_points(389,-52,5).ring_class_field()
sage: K1 = heegner_points(389,-52,1).ring_class_field()
sage: K5.galois_group(K1).is_kolyvagin()
True
sage: K7 = heegner_points(389,-52,7).ring_class_field()
sage: K7.galois_group(K1).is_kolyvagin()
False
sage: K25 = heegner_points(389,-52,25).ring_class_field()
sage: K25.galois_group(K1).is_kolyvagin()
False
```

kolyvagin_generators()

Assuming this Galois group G is of the form $G = \mathrm{Gal}(K_c/K_1)$, with $c = p_1 \dots p_n$ satisfying the Kolyvagin hypothesis, this function returns noncanonical choices of lifts of generators for each of the cyclic factors of G corresponding to the primes dividing c. Thus the i-th returned valued is an element of G that maps to the identity element of $\mathrm{Gal}(K_p/K_1)$ for all $p \neq p_i$ and to a choice of generator of $\mathrm{Gal}(K_{p_i}/K_1)$.

OUTPUT:

- list of elements of `self`

EXAMPLES:

```
sage: K3 = heegner_points(389,-52,3).ring_class_field()
sage: K1 = heegner_points(389,-52,1).ring_class_field()
sage: G = K3.galois_group(K1)
sage: G.kolyvagin_generators()
(Class field automorphism defined by 9*x^2 - 6*x*y + 14*y^2,)

sage: K5 = heegner_points(389,-52,5).ring_class_field()
sage: K1 = heegner_points(389,-52,1).ring_class_field()
sage: G = K5.galois_group(K1)
sage: G.kolyvagin_generators()
(Class field automorphism defined by 17*x^2 - 14*x*y + 22*y^2,)
```

lift_of_hilbert_class_field_galois_group()

Assuming this Galois group G is of the form $G = \mathrm{Gal}(K_c/K)$, this function returns noncanonical choices of lifts of the elements of the quotient group $\mathrm{Gal}(K_1/K)$.

OUTPUT:

- tuple of elements of self

EXAMPLES:

```
sage: K5 = heegner_points(389,-52,5).ring_class_field()
sage: G = K5.galois_group(K5.quadratic_field())
sage: G.lift_of_hilbert_class_field_galois_group()
(Class field automorphism defined by x^2 + 325*y^2,
 Class field automorphism defined by 2*x^2 + 2*x*y + 163*y^2)
sage: G.cardinality()
12
sage: K5.quadratic_field().class_number()
2
```

class sage.schemes.elliptic_curves.heegner.**HeegnerPoint**(N, D, c)

Bases: SageObject

A Heegner point of level N, discriminant D and conductor c is any point on a modular curve or elliptic curve that is concocted in some way from a quadratic imaginary τ in the upper half plane with $\Delta(\tau) = Dc = \Delta(N\tau)$.

EXAMPLES:

```
sage: x = sage.schemes.elliptic_curves.heegner.HeegnerPoint(389,-7,13); x
Heegner point of level 389, discriminant -7, and conductor 13
sage: type(x)
<class 'sage.schemes.elliptic_curves.heegner.HeegnerPoint'>
sage: loads(dumps(x)) == x
True
```

conductor()

Return the conductor of this Heegner point.

EXAMPLES:

```
sage: heegner_point(389,-7,5).conductor()
5
sage: E = EllipticCurve('37a1'); P = E.kolyvagin_point(-67,7); P
Kolyvagin point of discriminant -67 and conductor 7
 on elliptic curve of conductor 37
sage: P.conductor()
7
sage: E = EllipticCurve('389a'); P = E.heegner_point(-7, 5); P.conductor()
5
```

discriminant()

Return the discriminant of the quadratic imaginary field associated to this Heegner point.

EXAMPLES:

```
sage: heegner_point(389,-7,5).discriminant()
-7
sage: E = EllipticCurve('37a1'); P = E.kolyvagin_point(-67,7); P
Kolyvagin point of discriminant -67 and conductor 7
 on elliptic curve of conductor 37
sage: P.discriminant()
-67
sage: E = EllipticCurve('389a'); P = E.heegner_point(-7, 5); P.discriminant()
-7
```

level()

Return the level of this Heegner point, which is the level of the modular curve $X_0(N)$ on which this is a Heegner point.

EXAMPLES:

```
sage: heegner_point(389,-7,5).level()
389
```

quadratic_field()

Return the quadratic number field of discriminant D.

EXAMPLES:

```
sage: x = heegner_point(37,-7,5)
sage: x.quadratic_field()
Number Field in sqrt_minus_7 with defining polynomial x^2 + 7
 with sqrt_minus_7 = 2.645751311064591?*I

sage: E = EllipticCurve('37a'); P = E.heegner_point(-40)
sage: P.quadratic_field()
Number Field in sqrt_minus_40 with defining polynomial x^2 + 40
 with sqrt_minus_40 = 6.324555320336759?*I
sage: P.quadratic_field() is P.quadratic_field()
True
sage: type(P.quadratic_field())
<class 'sage.rings.number_field.number_field.NumberField_quadratic_with_
↪category'>
```

quadratic_order()

Return the order in the quadratic imaginary field of conductor c, where c is the conductor of this Heegner point.

EXAMPLES:

```
sage: heegner_point(389,-7,5).quadratic_order()
Order of conductor 10 generated by 5*sqrt_minus_7
 in Number Field in sqrt_minus_7 with defining polynomial x^2 + 7
 with sqrt_minus_7 = 2.645751311064591?*I
sage: heegner_point(389,-7,5).quadratic_order().basis()
[1, 5*sqrt_minus_7]

sage: E = EllipticCurve('37a'); P = E.heegner_point(-40,11)
sage: P.quadratic_order()
Order of conductor 22 generated by 11*sqrt_minus_40
 in Number Field in sqrt_minus_40 with defining polynomial x^2 + 40
 with sqrt_minus_40 = 6.324555320336759?*I
sage: P.quadratic_order().basis()
[1, 11*sqrt_minus_40]
```

ring_class_field()

Return the ring class field associated to this Heegner point. This is an extension K_c over K, where K is the quadratic imaginary field and c is the conductor associated to this Heegner point. This Heegner point is defined over K_c and the Galois group $Gal(K_c/K)$ acts transitively on the Galois conjugates of this Heegner point.

EXAMPLES:

```
sage: E = EllipticCurve('389a'); K.<a> = QuadraticField(-5)
sage: len(K.factor(5))
1
sage: len(K.factor(23))
2
sage: E.heegner_point(-7, 5).ring_class_field().degree_over_K()
6
sage: E.heegner_point(-7, 23).ring_class_field().degree_over_K()
22
sage: E.heegner_point(-7, 5*23).ring_class_field().degree_over_K()
132
sage: E.heegner_point(-7, 5^2).ring_class_field().degree_over_K()
30
sage: E.heegner_point(-7, 7).ring_class_field().degree_over_K()
7
```

class sage.schemes.elliptic_curves.heegner.**HeegnerPointOnEllipticCurve**(*E*, *x*, *check=True*)

Bases: *HeegnerPoint*

A Heegner point on a curve associated to an order in a quadratic imaginary field.

EXAMPLES:

```
sage: E = EllipticCurve('37a'); P = E.heegner_point(-7,5); P
Heegner point of discriminant -7 and conductor 5 on elliptic curve of conductor 37
sage: type(P)
<class 'sage.schemes.elliptic_curves.heegner.HeegnerPointOnEllipticCurve'>
```

conjugates_over_K()

Return the $Gal(K_c/K)$ conjugates of this Heegner point.

EXAMPLES:

```
sage: E = EllipticCurve('77a')
sage: y = E.heegner_point(-52,5); y
Heegner point of discriminant -52 and conductor 5
 on elliptic curve of conductor 77
sage: print([z.quadratic_form() for z in y.conjugates_over_K()])
[77*x^2 + 52*x*y + 13*y^2, 154*x^2 + 206*x*y + 71*y^2, 539*x^2 + 822*x*y +␣
↪314*y^2,
 847*x^2 + 1284*x*y + 487*y^2, 1001*x^2 + 52*x*y + y^2, 1078*x^2 + 822*x*y +␣
↪157*y^2,
 1309*x^2 + 360*x*y + 25*y^2, 1309*x^2 + 2054*x*y + 806*y^2,
 1463*x^2 + 976*x*y + 163*y^2, 2233*x^2 + 2824*x*y + 893*y^2,
 2387*x^2 + 2054*x*y + 442*y^2, 3619*x^2 + 3286*x*y + 746*y^2]
sage: y.quadratic_form()
77*x^2 + 52*x*y + 13*y^2
```

curve()

Return the elliptic curve on which this is a Heegner point.

EXAMPLES:

```
sage: E = EllipticCurve('389a'); P = E.heegner_point(-7, 5)
sage: P.curve()
Elliptic Curve defined by y^2 + y = x^3 + x^2 - 2*x over Rational Field
```

(continues on next page)

(continued from previous page)

```
sage: P.curve() is E
True
```

heegner_point_on_X0N()

Return Heegner point on $X_0(N)$ that maps to this Heegner point on E.

EXAMPLES:

```
sage: E = EllipticCurve('37a'); P = E.heegner_point(-7,5); P
Heegner point of discriminant -7 and conductor 5 on elliptic curve
 of conductor 37
sage: P.heegner_point_on_X0N()
Heegner point 5/74*sqrt(-7) - 11/74 of discriminant -7 and conductor 5
 on X_0(37)
```

kolyvagin_cohomology_class(*n=None*)

Return the Kolyvagin class associated to this Heegner point.

INPUT:

- n – positive integer that divides the gcd of a_p and $p+1$ for all p dividing the conductor. If n is `None`, choose the largest valid n.

EXAMPLES:

```
sage: y = EllipticCurve('389a').heegner_point(-7,5)
sage: y.kolyvagin_cohomology_class(3)
Kolyvagin cohomology class c(5) in H^1(K,E[3])
```

kolyvagin_point()

Return the Kolyvagin point corresponding to this Heegner point.

This is the point obtained by applying the Kolyvagin operator $J_c I_c$ in the group ring of the Galois group to this Heegner point. It is a point that defines an element of $H^1(K, E[n])$, under certain hypotheses on n.

EXAMPLES:

```
sage: E = EllipticCurve('37a1'); y = E.heegner_point(-7); y
Heegner point of discriminant -7 on elliptic curve of conductor 37
sage: P = y.kolyvagin_point(); P
Kolyvagin point of discriminant -7 on elliptic curve of conductor 37
sage: P.numerical_approx()  # abs tol 1e-15
(-3.36910401903861e-16 - 2.22076195576076e-16*I
 : 3.33066907387547e-16 + 2.22076195576075e-16*I : 1.00000000000000)
```

map_to_complex_numbers(*prec=53*)

Return the point in the subfield M of the complex numbers (well defined only modulo the period lattice) corresponding to this Heegner point.

EXAMPLES:

We compute a nonzero Heegner point over a ring class field on a curve of rank 2:

```
sage: E = EllipticCurve('389a'); y = E.heegner_point(-7,5)
sage: y.map_to_complex_numbers()
1.49979679635196 + 0.369156204821526*I
sage: y.map_to_complex_numbers(100)
1.4997967963519640592142411892 + 0.36915620482152626830089145962*I
```

(continues on next page)

(continued from previous page)

```
sage: y.map_to_complex_numbers(10)
1.5 + 0.37*I
```

Here we see that the Heegner point is 0 since it lies in the lattice:

```
sage: E = EllipticCurve('389a'); y = E.heegner_point(-7)
sage: y.map_to_complex_numbers(10)
0.0034 - 3.9*I
sage: y.map_to_complex_numbers()
4.71844785465692e-15 - 3.94347540310330*I
sage: E.period_lattice().basis()
(2.49021256085505, 1.97173770155165*I)
sage: 2*E.period_lattice().basis()[1]
3.94347540310330*I
```

You can also directly coerce to the complex field:

```
sage: E = EllipticCurve('389a'); y = E.heegner_point(-7)
sage: z = ComplexField(100)(y); z # real part approx. 0
-... - 3.9434754031032964088448153963*I
sage: E.period_lattice().elliptic_exponential(z)
(0.00000000000000000000000000000 : 1.0000000000000000000000000000 : 0.
↪0000000000000000000000000000)
```

numerical_approx(*prec=53*, *algorithm=None*)

Return a numerical approximation to this Heegner point computed using a working precision of prec bits.

Warning: The answer is *not* provably correct to prec bits! A priori, due to rounding and other errors, it is possible that not a single digit is correct.

INPUT:

- prec – (default: `None`) the working precision

EXAMPLES:

```
sage: E = EllipticCurve('37a'); P = E.heegner_point(-7); P
Heegner point of discriminant -7 on elliptic curve of conductor 37
sage: P.numerical_approx()  # abs tol 1e-15
(-3.36910401903861e-16 - 2.22076195576076e-16*I
 : 3.33066907387547e-16 + 2.22076195576075e-16*I : 1.00000000000000)
sage: P.numerical_approx(10)  # expect random digits
(0.0030 - 0.0028*I : -0.0030 + 0.0028*I : 1.0)
sage: P.numerical_approx(100)[0]  # expect random digits
8.4...e-31 + 6.0...e-31*I
sage: E = EllipticCurve('37a'); P = E.heegner_point(-40); P
Heegner point of discriminant -40 on elliptic curve of conductor 37
sage: P.numerical_approx()  # abs tol 1e-14
(-3.15940603400359e-16 + 1.41421356237309*I
 : 1.00000000000000 - 1.41421356237309*I : 1.00000000000000)
```

A rank 2 curve, where all Heegner points of conductor 1 are 0:

```
sage: E = EllipticCurve('389a'); E.rank()
2
```

(continues on next page)

(continued from previous page)

```
sage: P = E.heegner_point(-7); P
Heegner point of discriminant -7 on elliptic curve of conductor 389
sage: P.numerical_approx()
(0.000000000000000 : 1.00000000000000 : 0.000000000000000)
```

However, Heegner points of bigger conductor are often nonzero:

```
sage: E = EllipticCurve('389a'); P = E.heegner_point(-7, 5); P
Heegner point of discriminant -7 and conductor 5 on elliptic curve
 of conductor 389
sage: numerical_approx(P)
(0.675507556926807 + 0.344749649302635*I
 : -0.377142931401887 + 0.843366227137146*I : 1.00000000000000)
sage: P.numerical_approx()
(0.6755075569268... + 0.3447496493026...*I
 : -0.3771429314018... + 0.8433662271371...*I : 1.00000000000000)
sage: E.heegner_point(-7, 11).numerical_approx()
(0.1795583794118... + 0.02035501750912...*I
 : -0.5573941377055... + 0.2738940831635...*I : 1.00000000000000)
sage: E.heegner_point(-7, 13).numerical_approx()
(1.034302915374... - 3.302744319777...*I
 : 1.323937875767... + 6.908264226850...*I : 1.00000000000000)
```

We find (probably) the defining polynomial of the x-coordinate of P, which defines a class field. The shape of the discriminant below is strong confirmation – but not proof – that this polynomial is correct:

```
sage: f = P.numerical_approx(70)[0].algdep(6); f
1225*x^6 + 1750*x^5 - 21675*x^4 - 380*x^3 + 110180*x^2 - 129720*x + 48771
sage: f.discriminant().factor()
2^6 * 3^2 * 5^11 * 7^4 * 13^2 * 19^6 * 199^2 * 719^2 * 26161^2
```

point_exact (*prec=53*, *algorithm='lll'*, *var='a'*, *optimize=False*)

Return exact point on the elliptic curve over a number field defined by computing this Heegner point to the given number of bits of precision. A ValueError is raised if the precision is clearly insignificant to define a point on the curve.

Warning: It is in theory possible for this function to not raise a ValueError, find a point on the curve, but via some very unlikely coincidence that point is not actually this Heegner point.

Warning: Currently we make an arbitrary choice of y-coordinate for the lift of the x-coordinate.

INPUT:

- `prec` – integer (default: 53)
- `algorithm` – see the description of the algorithm parameter for the `x_poly_exact` method.
- `var` – string (default: 'a')
- `optimize` – bool (default; False) if `True`, try to optimize defining polynomial for the number field that the point is defined over. Off by default, since this can be very expensive.

EXAMPLES:

```
sage: E = EllipticCurve('389a'); P = E.heegner_point(-7, 5); P
Heegner point of discriminant -7 and conductor 5
 on elliptic curve of conductor 389
sage: z = P.point_exact(200, optimize=True)
sage: z[1].charpoly()
x^12 + 6*x^11 + 90089/1715*x^10 + 71224/343*x^9 + 52563964/588245*x^8 -␣
↪483814934/588245*x^7 - 156744579/16807*x^6 - 2041518032/84035*x^5 +␣
↪1259355443184/14706125*x^4 + 3094420220918/14706125*x^3 + 123060442043827/
↪367653125*x^2 + 82963044474852/367653125*x + 211679465261391/1838265625
sage: f = P.numerical_approx(500)[1].algdep(12); f / f.leading_coefficient()
x^12 + 6*x^11 + 90089/1715*x^10 + 71224/343*x^9 + 52563964/588245*x^8 -␣
↪483814934/588245*x^7 - 156744579/16807*x^6 - 2041518032/84035*x^5 +␣
↪1259355443184/14706125*x^4 + 3094420220918/14706125*x^3 + 123060442043827/
↪367653125*x^2 + 82963044474852/367653125*x + 211679465261391/1838265625

sage: E = EllipticCurve('5077a')
sage: P = E.heegner_point(-7)
sage: P.point_exact(prec=100)
(0 : 1 : 0)
```

quadratic_form()

Return the integral primitive positive definite binary quadratic form associated to this Heegner point.

EXAMPLES:

```
sage: EllipticCurve('389a').heegner_point(-7, 5).quadratic_form()
389*x^2 + 147*x*y + 14*y^2

sage: P = EllipticCurve('389a').heegner_point(-7, 5, (778,925,275)); P
Heegner point of discriminant -7 and conductor 5 on elliptic curve
 of conductor 389
sage: P.quadratic_form()
778*x^2 + 925*x*y + 275*y^2
```

satisfies_kolyvagin_hypothesis(*n=None*)

Return `True` if this Heegner point and n satisfy the Kolyvagin hypothesis, i.e., that each prime dividing the conductor c of `self` is inert in K and coprime to ND. Moreover, if n is not `None`, also check that for each prime p dividing c we have that $n|\gcd(a_p(E), p+1)$.

INPUT:

- n – positive integer

EXAMPLES:

```
sage: EllipticCurve('389a').heegner_point(-7).satisfies_kolyvagin_hypothesis()
True
sage: EllipticCurve('389a').heegner_point(-7, 5).satisfies_kolyvagin_
↪hypothesis()
True
sage: EllipticCurve('389a').heegner_point(-7, 11).satisfies_kolyvagin_
↪hypothesis()
False
```

tau()

Return τ in the upper half plane that maps via the modular parametrization to this Heegner point.

EXAMPLES:

```
sage: E = EllipticCurve('389a'); P = E.heegner_point(-7, 5)
sage: P.tau()
5/778*sqrt_minus_7 - 147/778
```

x_poly_exact (*prec=53*, *algorithm='lll'*)

Return irreducible polynomial over the rational numbers satisfied by the x coordinate of this Heegner point. A ValueError is raised if the precision is clearly insignificant to define a point on the curve.

Warning: It is in theory possible for this function to not raise a ValueError, find a polynomial, but via some very unlikely coincidence that point is not actually this Heegner point.

INPUT:

- `prec` – integer (default: 53)
- `algorithm` – 'conjugates' or 'lll' (default); if 'conjugates', compute numerically all the conjugates `y[i]` of the Heegner point and construct the characteristic polynomial as the product $f(X) = (X - y[i])$. If 'lll', compute only one of the conjugates `y[0]`, then uses the LLL algorithm to guess $f(X)$.

EXAMPLES:

We compute some x-coordinate polynomials of some conductor 1 Heegner points:

```
sage: E = EllipticCurve('37a')
sage: v = E.heegner_discriminants_list(10)
sage: [E.heegner_point(D).x_poly_exact() for D in v]
[x, x, x^2 + 2, x^5 - x^4 + x^3 + x^2 - 2*x + 1, x - 6,
 x^7 - 2*x^6 + 9*x^5 - 10*x^4 - x^3 + 8*x^2 - 5*x + 1,
 x^3 + 5*x^2 + 10*x + 4, x^4 - 10*x^3 + 10*x^2 + 12*x - 12,
 x^8 - 5*x^7 + 7*x^6 + 13*x^5 - 10*x^4 - 4*x^3 + x^2 - 5*x + 7,
 x^6 - 2*x^5 + 11*x^4 - 24*x^3 + 30*x^2 - 16*x + 4]
```

We compute x-coordinate polynomials for some Heegner points of conductor bigger than 1 on a rank 2 curve:

```
sage: E = EllipticCurve('389a'); P = E.heegner_point(-7, 5); P
Heegner point of discriminant -7 and conductor 5
 on elliptic curve of conductor 389
sage: P.x_poly_exact()
Traceback (most recent call last):
...
ValueError: insufficient precision to determine Heegner point
(fails discriminant test)
sage: P.x_poly_exact(120)
x^6 + 10/7*x^5 - 867/49*x^4 - 76/245*x^3 + 3148/35*x^2 - 25944/245*x + 48771/
↪1225
sage: E.heegner_point(-7, 11).x_poly_exact(500)
x^10 + 282527/52441*x^9 + 27049007420/2750058481*x^8 - 22058564794/
↪2750058481*x^7
 - 140054237301/2750058481*x^6 + 696429998952/30250643291*x^5
 + 2791387923058/30250643291*x^4 - 3148473886134/30250643291*x^3
 + 1359454055022/30250643291*x^2 - 250620385365/30250643291*x
 + 181599685425/332757076201
```

Here we compute a Heegner point of conductor 5 on a rank 3 curve:

```
sage: E = EllipticCurve('5077a'); P = E.heegner_point(-7,5); P
Heegner point of discriminant -7 and conductor 5
 on elliptic curve of conductor 5077
sage: P.x_poly_exact(500)
x^6 + 1108754853727159228/72351048803252547*x^5 + 88875505551184048168/
↪1953478317687818769*x^4 - 22162002711660986621 32/3255797196146364615*x^3 +␣
↪1494162750416883944985 1/9767391588439093845*x^2 - 3456417460183342963918/
↪3255797196146364615*x + 1306572835857500500459/5426328660243941025
```

See Issue #34121:

```
sage: E = EllipticCurve('11a1')
sage: P = E.heegner_point(-7)
sage: PE = P.point_exact()
sage: PE
(a : -4*a + 3 : 1)
sage: all(c.parent().disc() == -7 for c in PE)
True
```

class sage.schemes.elliptic_curves.heegner.**HeegnerPointOnX0N**(*N*, *D*, *c=1*, *f=None*, *check=True*)

Bases: *HeegnerPoint*

A Heegner point as a point on the modular curve $X_0(N)$, which we view as the upper half plane modulo the action of $\Gamma_0(N)$.

EXAMPLES:

```
sage: x = heegner_point(37, -7, 5); x
Heegner point 5/74*sqrt(-7) - 11/74 of discriminant -7 and conductor 5 on X_0(37)
sage: type(x)
<class 'sage.schemes.elliptic_curves.heegner.HeegnerPointOnX0N'>
sage: x.level()
37
sage: x.conductor()
5
sage: x.discriminant()
-7
sage: x.quadratic_field()
Number Field in sqrt_minus_7 with defining polynomial x^2 + 7
 with sqrt_minus_7 = 2.645751311064591?*I
sage: x.quadratic_form()
37*x^2 + 11*x*y + 2*y^2
sage: x.quadratic_order()
Order of conductor 10 generated by 5*sqrt_minus_7
 in Number Field in sqrt_minus_7 with defining polynomial x^2 + 7
 with sqrt_minus_7 = 2.645751311064591?*I
sage: x.tau()
5/74*sqrt_minus_7 - 11/74
sage: loads(dumps(x)) == x
True
```

atkin_lehner_act(*Q=None*)

Given an integer Q dividing the level N such that $\gcd(Q, N/Q) = 1$, return the image of this Heegner point under the Atkin-Lehner operator W_Q.

INPUT:

- Q – positive divisor of N; if not given, default to N

EXAMPLES:

```
sage: x = heegner_point(389, -7, 5)
sage: x.atkin_lehner_act()
Heegner point 5/199168*sqrt(-7) - 631/199168 of discriminant -7
 and conductor 5 on X_0(389)

sage: x = heegner_point(45, D=-11, c=1); x
Heegner point 1/90*sqrt(-11) - 13/90 of discriminant -11 on X_0(45)
sage: x.atkin_lehner_act(5)
Heegner point 1/90*sqrt(-11) + 23/90 of discriminant -11 on X_0(45)
sage: y = x.atkin_lehner_act(9); y
Heegner point 1/90*sqrt(-11) - 23/90 of discriminant -11 on X_0(45)
sage: z = y.atkin_lehner_act(9); z
Heegner point 1/90*sqrt(-11) - 13/90 of discriminant -11 on X_0(45)
sage: z == x
True
```

galois_orbit_over_K()

Return the $Gal(K_c/K)$-orbit of this Heegner point.

EXAMPLES:

```
sage: x = heegner_point(389, -7, 3); x
Heegner point 3/778*sqrt(-7) - 223/778 of discriminant -7
 and conductor 3 on X_0(389)
sage: x.galois_orbit_over_K()
[Heegner point 3/778*sqrt(-7) - 223/778 of discriminant -7 and conductor 3 on␣
↪X_0(389),
 Heegner point 3/1556*sqrt(-7) - 223/1556 of discriminant -7 and conductor 3␣
↪on X_0(389),
 Heegner point 3/1556*sqrt(-7) - 1001/1556 of discriminant -7 and conductor 3␣
↪on X_0(389),
 Heegner point 3/3112*sqrt(-7) - 223/3112 of discriminant -7 and conductor 3␣
↪on X_0(389)]
```

map_to_curve(E)

Return the image of this Heegner point on the elliptic curve E, which must also have conductor N, where N is the level of `self`.

EXAMPLES:

```
sage: x = heegner_point(389, -7, 5); x
Heegner point 5/778*sqrt(-7) - 147/778 of discriminant -7
 and conductor 5 on X_0(389)
sage: y = x.map_to_curve(EllipticCurve('389a')); y
Heegner point of discriminant -7 and conductor 5
 on elliptic curve of conductor 389
sage: y.curve().cremona_label()
'389a1'
sage: y.heegner_point_on_X0N()
Heegner point 5/778*sqrt(-7) - 147/778 of discriminant -7
 and conductor 5 on X_0(389)
```

You can also directly apply the modular parametrization of the elliptic curve:

```
sage: x = heegner_point(37,-7); x
Heegner point 1/74*sqrt(-7) - 17/74 of discriminant -7 on X_0(37)
```

(continues on next page)

(continued from previous page)

```
sage: E = EllipticCurve('37a'); phi = E.modular_parametrization()
sage: phi(x)
Heegner point of discriminant -7 on elliptic curve of conductor 37
```

plot(***kwds*)

Draw a point at (x, y) where this Heegner point is represented by the point $\tau = x + iy$ in the upper half plane.

The `kwds` get passed onto the point plotting command.

EXAMPLES:

```
sage: heegner_point(389,-7,1).plot(pointsize=50)
Graphics object consisting of 1 graphics primitive
```

quadratic_form()

Return the integral primitive positive-definite binary quadratic form associated to this Heegner point.

EXAMPLES:

```
sage: heegner_point(389, -7, 5).quadratic_form()
389*x^2 + 147*x*y + 14*y^2
```

reduced_quadratic_form()

Return reduced binary quadratic corresponding to this Heegner point.

EXAMPLES:

```
sage: x = heegner_point(389, -7, 5)
sage: x.quadratic_form()
389*x^2 + 147*x*y + 14*y^2
sage: x.reduced_quadratic_form()
4*x^2 - x*y + 11*y^2
```

tau()

Return an element `tau` in the upper half plane that corresponds to this particular Heegner point.

Actually, `tau` is in the quadratic imaginary field K associated to this Heegner point.

EXAMPLES:

```
sage: x = heegner_point(37, -7, 5); tau = x.tau(); tau
5/74*sqrt_minus_7 - 11/74
sage: 37 * tau.minpoly()
37*x^2 + 11*x + 2
sage: x.quadratic_form()
37*x^2 + 11*x*y + 2*y^2
```

class sage.schemes.elliptic_curves.heegner.**HeegnerPoints**(N)

Bases: SageObject

The set of Heegner points with given parameters.

EXAMPLES:

```
sage: H = heegner_points(389); H
Set of all Heegner points on X_0(389)
```

(continues on next page)

(continued from previous page)

```
sage: type(H)
<class 'sage.schemes.elliptic_curves.heegner.HeegnerPoints_level'>
sage: isinstance(H, sage.schemes.elliptic_curves.heegner.HeegnerPoints)
True
```

level()

Return the level N of the modular curve $X_0(N)$.

EXAMPLES:

```
sage: heegner_points(389).level()
389
```

class sage.schemes.elliptic_curves.heegner.**HeegnerPoints_level**(N)

Bases: *HeegnerPoints*

Return the infinite set of all Heegner points on $X_0(N)$ for all quadratic imaginary fields.

EXAMPLES:

```
sage: H = heegner_points(11); H
Set of all Heegner points on X_0(11)
sage: type(H)
<class 'sage.schemes.elliptic_curves.heegner.HeegnerPoints_level'>
sage: loads(dumps(H)) == H
True
```

discriminants(*n=10*, *weak=False*)

Return the first n quadratic imaginary discriminants that satisfy the Heegner hypothesis for N.

INPUT:

- n – nonnegative integer
- `weak` – bool (default: `False`); if `True` only require weak Heegner hypothesis, which is the same as usual but without the condition that $\gcd(D, N) = 1$.

EXAMPLES:

```
sage: X = heegner_points(37)
sage: X.discriminants(5)
[-7, -11, -40, -47, -67]
```

The default is 10:

```
sage: X.discriminants()
[-7, -11, -40, -47, -67, -71, -83, -84, -95, -104]
sage: X.discriminants(15)
[-7, -11, -40, -47, -67, -71, -83, -84, -95, -104, -107, -115, -120, -123, -
↪127]
```

The discriminant -111 satisfies only the weak Heegner hypothesis, since it is divisible by 37:

```
sage: X.discriminants(15, weak=True)
[-7, -11, -40, -47, -67, -71, -83, -84, -95, -104, -107, -111, -115, -120, -
↪123]
```

reduce_mod(*ell*)

Return object that allows for computation with Heegner points of level N modulo the prime ℓ, represented using quaternion algebras.

INPUT:

- ℓ – prime

EXAMPLES:

```
sage: heegner_points(389).reduce_mod(7).quaternion_algebra()
Quaternion Algebra (-1, -7) with base ring Rational Field
```

class sage.schemes.elliptic_curves.heegner.**HeegnerPoints_level_disc**(N, D)

Bases: *HeegnerPoints*

Set of Heegner points of given level and all conductors associated to a quadratic imaginary field.

EXAMPLES:

```
sage: H = heegner_points(389,-7); H
Set of all Heegner points on X_0(389) associated to QQ[sqrt(-7)]
sage: type(H)
<class 'sage.schemes.elliptic_curves.heegner.HeegnerPoints_level_disc'>
sage: H._repr_()
'Set of all Heegner points on X_0(389) associated to QQ[sqrt(-7)]'
sage: H.discriminant()
-7
sage: H.quadratic_field()
Number Field in sqrt_minus_7 with defining polynomial x^2 + 7
 with sqrt_minus_7 = 2.645751311064591?*I
sage: H.kolyvagin_conductors()
[1, 3, 5, 13, 15, 17, 19, 31, 39, 41]

sage: loads(dumps(H)) == H
True
```

discriminant()

Return the discriminant of the quadratic imaginary extension K.

EXAMPLES:

```
sage: heegner_points(389,-7).discriminant()
-7
```

kolyvagin_conductors(*r=None*, *n=10*, *E=None*, *m=None*)

Return the first n conductors that are squarefree products of distinct primes inert in the quadratic imaginary field $K = \mathbf{Q}(\sqrt{D})$. If r is specified, return only conductors that are a product of r distinct primes all inert in K. If $r = 0$, always return the list `[1]`, no matter what.

If the optional elliptic curve E and integer m are given, then only include conductors c such that for each prime divisor p of c we have $m \mid \gcd(a_p(E), p + 1)$.

INPUT:

- r – (default: `None`) nonnegative integer or `None`
- n – positive integer
- E – an elliptic curve

- m – a positive integer

EXAMPLES:

```
sage: H = heegner_points(389, -7)
sage: H.kolyvagin_conductors(0)
[1]
sage: H.kolyvagin_conductors(1)
[3, 5, 13, 17, 19, 31, 41, 47, 59, 61]
sage: H.kolyvagin_conductors(1,15)
[3, 5, 13, 17, 19, 31, 41, 47, 59, 61, 73, 83, 89, 97, 101]
sage: H.kolyvagin_conductors(1,5)
[3, 5, 13, 17, 19]
sage: H.kolyvagin_conductors(1, 5, EllipticCurve('389a'), 3)
[5, 17, 41, 59, 83]
sage: H.kolyvagin_conductors(2, 5, EllipticCurve('389a'), 3)
[85, 205, 295, 415, 697]
```

quadratic_field()

Return the quadratic imaginary field $K = \mathbf{Q}(\sqrt{D})$.

EXAMPLES:

```
sage: E = EllipticCurve('389a'); K = E.heegner_point(-7,5).ring_class_field()
sage: K.quadratic_field()
Number Field in sqrt_minus_7 with defining polynomial x^2 + 7
 with sqrt_minus_7 = 2.645751311064591?*I
```

class sage.schemes.elliptic_curves.heegner.**HeegnerPoints_level_disc_cond**(N, D, c=1)

Bases: *HeegnerPoints_level*, *HeegnerPoints_level_disc*

The set of Heegner points of given level, discriminant, and conductor.

EXAMPLES:

```
sage: H = heegner_points(389,-7,5); H
All Heegner points of conductor 5 on X_0(389) associated to QQ[sqrt(-7)]
sage: type(H)
<class 'sage.schemes.elliptic_curves.heegner.HeegnerPoints_level_disc_cond'>
sage: H.discriminant()
-7
sage: H.level()
389

sage: len(H.points())
12
sage: H.points()[0]
Heegner point 5/778*sqrt(-7) - 147/778 of discriminant -7 and conductor 5 on X_
↪0(389)
sage: H.betas()
(147, 631)

sage: H.quadratic_field()
Number Field in sqrt_minus_7 with defining polynomial x^2 + 7
 with sqrt_minus_7 = 2.645751311064591?*I
sage: H.ring_class_field()
Ring class field extension of QQ[sqrt(-7)] of conductor 5
```

(continues on next page)

(continued from previous page)

```
sage: H.kolyvagin_conductors()
[1, 3, 5, 13, 15, 17, 19, 31, 39, 41]
sage: H.satisfies_kolyvagin_hypothesis()
True

sage: H = heegner_points(389,-7,5)
sage: loads(dumps(H)) == H
True
```

betas()

Return the square roots of Dc^2 modulo $4N$ all reduced mod $2N$, without multiplicity.

EXAMPLES:

```
sage: X = heegner_points(45,-11,1); X
All Heegner points of conductor 1 on X_0(45) associated to QQ[sqrt(-11)]
sage: [x.quadratic_form() for x in X]
[45*x^2 + 13*x*y + y^2,
 45*x^2 + 23*x*y + 3*y^2,
 45*x^2 + 67*x*y + 25*y^2,
 45*x^2 + 77*x*y + 33*y^2]
sage: X.betas()
(13, 23, 67, 77)
sage: X.points(13)
(Heegner point 1/90*sqrt(-11) - 13/90 of discriminant -11 on X_0(45),)
sage: [x.quadratic_form() for x in X.points(13)]
[45*x^2 + 13*x*y + y^2]
```

conductor()

Return the level of the conductor.

EXAMPLES:

```
sage: heegner_points(389,-7,5).conductor()
5
```

plot(**args*, ***kwds*)

Returns plot of all the representatives in the upper half plane of the Heegner points in this set of Heegner points.

The inputs to this function get passed onto the point command.

EXAMPLES:

```
sage: heegner_points(389,-7,5).plot(pointsize=50, rgbcolor='red')           #␣
↪needs sage.plot
Graphics object consisting of 12 graphics primitives
sage: heegner_points(53,-7,15).plot(pointsize=50, rgbcolor='purple')        #␣
↪needs sage.plot
Graphics object consisting of 48 graphics primitives
```

points(*beta=None*)

Return the Heegner points in `self`. If β is given, return only those Heegner points with given β, i.e., whose quadratic form has B congruent to β modulo $2N$.

Use `self.beta()` to get a list of betas.

EXAMPLES:

```
sage: H = heegner_points(389, -7, 5); H
All Heegner points of conductor 5 on X_0(389) associated to QQ[sqrt(-7)]
sage: H.points()
(Heegner point 5/778*sqrt(-7) - 147/778 of discriminant -7 and conductor 5
  on X_0(389),
 ...,
 Heegner point 5/5446*sqrt(-7) - 757/778 of discriminant -7 and conductor 5
  on X_0(389))
sage: H.betas()
(147, 631)
sage: [x.tau() for x in H.points(147)]
[5/778*sqrt_minus_7 - 147/778, 5/1556*sqrt_minus_7 - 147/1556,
 5/1556*sqrt_minus_7 - 925/1556, 5/3112*sqrt_minus_7 - 1703/3112,
 5/3112*sqrt_minus_7 - 2481/3112, 5/5446*sqrt_minus_7 - 21/778]

sage: [x.tau() for x in H.points(631)]
[5/778*sqrt_minus_7 - 631/778, 5/1556*sqrt_minus_7 - 631/1556,
 5/1556*sqrt_minus_7 - 1409/1556, 5/3112*sqrt_minus_7 - 631/3112,
 5/3112*sqrt_minus_7 - 1409/3112, 5/5446*sqrt_minus_7 - 757/778]
```

The result is cached and is a tuple (since it is immutable):

```
sage: H.points() is H.points()
True
sage: type(H.points())
<... 'tuple'>
```

ring_class_field()

Return the ring class field associated to this set of Heegner points. This is an extension K_c over K, where K is the quadratic imaginary field and c the conductor associated to this Heegner point. This Heegner point is defined over K_c and the Galois group $Gal(K_c/K)$ acts transitively on the Galois conjugates of this Heegner point.

EXAMPLES:

```
sage: heegner_points(389,-7,5).ring_class_field()
Ring class field extension of QQ[sqrt(-7)] of conductor 5
```

satisfies_kolyvagin_hypothesis()

Return `True` if `self` satisfies the Kolyvagin hypothesis, i.e., that each prime dividing the conductor c of `self` is inert in K and coprime to ND.

EXAMPLES:

The prime 5 is inert, but the prime 11 is not:

```
sage: heegner_points(389,-7,5).satisfies_kolyvagin_hypothesis()
True
sage: heegner_points(389,-7,11).satisfies_kolyvagin_hypothesis()
False
```

class sage.schemes.elliptic_curves.heegner.**HeegnerQuatAlg**(*level*, *ell*)

Bases: SageObject

Heegner points viewed as supersingular points on the modular curve $X_0(N)/\mathbf{F}_\ell$.

EXAMPLES:

```
sage: H = heegner_points(11).reduce_mod(13); H
Heegner points on X_0(11) over F_13
sage: type(H)
<class 'sage.schemes.elliptic_curves.heegner.HeegnerQuatAlg'>
sage: loads(dumps(H)) == H
True
```

brandt_module()

Return the Brandt module of right ideal classes that we used to represent the set of supersingular points on the modular curve.

EXAMPLES:

```
sage: heegner_points(11).reduce_mod(3).brandt_module()
Brandt module of dimension 2 of level 3*11 of weight 2 over Rational Field
```

cyclic_subideal_p1(I, c)

Compute dictionary mapping 2-tuples that defined normalized elements of $P^1(\mathbf{Z}/c\mathbf{Z})$

INPUT:

- I – right ideal of Eichler order or in quaternion algebra
- c – square free integer (currently must be odd prime and coprime to level, discriminant, characteristic, etc.

OUTPUT:

- dictionary mapping 2-tuples (u,v) to ideals

EXAMPLES:

```
sage: H = heegner_points(11).reduce_mod(7)
sage: I = H.brandt_module().right_ideals()[0]
sage: sorted(H.cyclic_subideal_p1(I, 3).items())
[((0, 1),
  Fractional ideal (2 + 2*j + 32*k, 2*i + 8*j + 82*k, 12*j + 60*k, 132*k)),
 ((1, 0),
  Fractional ideal (2 + 10*j + 28*k, 2*i + 4*j + 62*k, 12*j + 60*k, 132*k)),
 ((1, 1),
  Fractional ideal (2 + 2*j + 76*k, 2*i + 4*j + 106*k, 12*j + 60*k, 132*k)),
 ((1, 2),
  Fractional ideal (2 + 10*j + 116*k, 2*i + 8*j + 38*k, 12*j + 60*k, 132*k))]
sage: len(H.cyclic_subideal_p1(I, 17))
18
```

ell()

Return the prime ℓ modulo which we are working.

EXAMPLES:

```
sage: heegner_points(11).reduce_mod(3).ell()
3
```

galois_group_over_hilbert_class_field(D, c)

Return the Galois group of the extension of ring class fields K_c over the Hilbert class field K_1 of the quadratic imaginary field of discriminant D.

INPUT:

- D – fundamental discriminant
- c – conductor (square-free integer)

EXAMPLES:

```
sage: N = 37; D = -7; ell = 17; c = 41; p = 3
sage: H = heegner_points(N).reduce_mod(ell)
sage: H.galois_group_over_hilbert_class_field(D, c)
Galois group of Ring class field extension of QQ[sqrt(-7)] of conductor 41
 over Hilbert class field of QQ[sqrt(-7)]
```

galois_group_over_quadratic_field(*D*, *c*)

Return the Galois group of the extension of ring class fields K_c over the quadratic imaginary field K of discriminant D.

INPUT:

- D – fundamental discriminant
- c – conductor (square-free integer)

EXAMPLES:

```
sage: N = 37; D = -7; ell = 17; c = 41; p = 3
sage: H = heegner_points(N).reduce_mod(ell)
sage: H.galois_group_over_quadratic_field(D, c)
Galois group of Ring class field extension of QQ[sqrt(-7)] of conductor 41
 over Number Field in sqrt_minus_7 with defining polynomial x^2 + 7
  with sqrt_minus_7 = 2.645751311064591?*I
```

heegner_conductors(*D*, *n=5*)

Return the first n negative fundamental discriminants coprime to $N\ell$ such that ℓ is inert in the corresponding quadratic imaginary field and that field satisfies the Heegner hypothesis.

INPUT:

- D – negative integer; a fundamental Heegner discriminant
- n – positive integer (default: 5)

OUTPUT: A list.

EXAMPLES:

```
sage: H = heegner_points(11).reduce_mod(3)
sage: H.heegner_conductors(-7)
[1, 2, 4, 5, 8]
sage: H.heegner_conductors(-7, 10)
[1, 2, 4, 5, 8, 10, 13, 16, 17, 19]
```

heegner_discriminants(*n=5*)

Return the first n negative fundamental discriminants coprime to $N\ell$ such that ℓ is inert in the corresponding quadratic imaginary field and that field satisfies the Heegner hypothesis, and N is the level.

INPUT:

- n – positive integer (default: 5)

OUTPUT: A list.

EXAMPLES:

```
sage: H = heegner_points(11).reduce_mod(3)
sage: H.heegner_discriminants()
[-7, -19, -40, -43, -52]
sage: H.heegner_discriminants(10)
[-7, -19, -40, -43, -52, -79, -127, -139, -151, -184]
```

heegner_divisor(*D*, *c=1*)

Return Heegner divisor as an element of the Brandt module corresponding to the discriminant D and conductor c, which both must be coprime to $N\ell$.

More precisely, we compute the sum of the reductions of the $\mathrm{Gal}(K_1/K)$-conjugates of each choice of y_1, where the choice comes from choosing the ideal $\mathcal{N}$. Then we apply the Hecke operator T_c to this sum.

INPUT:

- D – discriminant (negative integer)
- c – conductor (positive integer)

OUTPUT: A Brandt module element.

EXAMPLES:

```
sage: H = heegner_points(11).reduce_mod(7)
sage: H.heegner_discriminants()
[-8, -39, -43, -51, -79]
sage: H.heegner_divisor(-8)
(1, 0, 0, 1, 0, 0)
sage: H.heegner_divisor(-39)
(1, 2, 2, 1, 2, 0)
sage: H.heegner_divisor(-43)
(1, 0, 0, 1, 0, 0)
sage: H.heegner_divisor(-51)
(1, 0, 0, 1, 0, 2)
sage: H.heegner_divisor(-79)
(3, 2, 2, 3, 0, 0)

sage: sum(H.heegner_divisor(-39).element())
8
sage: QuadraticField(-39,'a').class_number()
4
```

kolyvagin_cyclic_subideals(*I*, *p*, *alpha_quaternion*)

Return list of pairs (J, n) where J runs through the cyclic subideals of I of index $(\mathbf{Z}/p\mathbf{Z})^2$, and $J \sim \alpha^n(J_0)$ for some fixed choice of cyclic subideal J_0.

INPUT:

- I – right ideal of the quaternion algebra
- p – prime number
- `alpha_quaternion` – image in the quaternion algebra of generator α for $(\mathcal{O}_K/c\mathcal{O}_K)^*/(\mathbf{Z}/c\mathbf{Z})^*$.

OUTPUT: A list of 2-tuples.

EXAMPLES:

```
sage: N = 37; D = -7; ell = 17; c = 5
sage: H = heegner_points(N).reduce_mod(ell)
sage: I = H.brandt_module().right_ideals()[49]
```

(continues on next page)

(continued from previous page)

```
sage: f = H.optimal_embeddings(D, 1, I.left_order())[1]
sage: g = H.kolyvagin_generators(f.domain().number_field(), c)
sage: alpha_quaternion = f(g[0]); alpha_quaternion
1 - 77/192*i - 5/128*j - 137/384*k
sage: H.kolyvagin_cyclic_subideals(I, 5, alpha_quaternion)
[(Fractional ideal (2 + 2/3*i + 364*j + 231928/3*k,
                    4/3*i + 946*j + 69338/3*k,
                    1280*j + 49920*k, 94720*k), 0),
 (Fractional ideal (2 + 2/3*i + 108*j + 31480/3*k,
                    4/3*i + 434*j + 123098/3*k,
                    1280*j + 49920*k, 94720*k), 1),
 (Fractional ideal (2 + 2/3*i + 876*j + 7672/3*k,
                    4/3*i + 434*j + 236762/3*k,
                    1280*j + 49920*k, 94720*k), 2),
 (Fractional ideal (2 + 2/3*i + 364*j + 61432/3*k,
                    4/3*i + 178*j + 206810/3*k,
                    1280*j + 49920*k, 94720*k), 3),
 (Fractional ideal (2 + 2/3*i + 876*j + 178168/3*k,
                    4/3*i + 1202*j + 99290/3*k,
                    1280*j + 49920*k, 94720*k), 4),
 (Fractional ideal (2 + 2/3*i + 1132*j + 208120/3*k,
                    4/3*i + 946*j + 183002/3*k,
                    1280*j + 49920*k, 94720*k), 5)]
```

kolyvagin_generator(K, p)

Return element in K that maps to the multiplicative generator for the quotient group

$$(\mathcal{O}_K/p\mathcal{O}_K)^*/(\mathbf{Z}/p\mathbf{Z})^*$$

of the form $\sqrt{D} + n$ with $n \geq 1$ minimal.

INPUT:

- K – quadratic imaginary field
- p – inert prime

EXAMPLES:

```
sage: N = 37; D = -7; ell = 17; p = 5
sage: H = heegner_points(N).reduce_mod(ell)
sage: I = H.brandt_module().right_ideals()[49]
sage: f = H.optimal_embeddings(D, 1, I.left_order())[0]
sage: H.kolyvagin_generator(f.domain().number_field(), 5)
a + 1
```

This function requires that p be prime, but `kolyvagin_generators` works in general:

```
sage: H.kolyvagin_generator(f.domain().number_field(), 5*17)
Traceback (most recent call last):
...
NotImplementedError: p must be prime
sage: H.kolyvagin_generators(f.domain().number_field(), 5*17)
[-34*a + 1, 35*a + 106]
```

kolyvagin_generators(K, c)

Return elements in $\mathcal{O}_K$ that map to multiplicative generators for the factors of the quotient group

$$(\mathcal{O}_K/c\mathcal{O}_K)^*/(\mathbf{Z}/c\mathbf{Z})^*$$

corresponding to the prime divisors of c. Each generator is of the form $\sqrt{D} + n$ with $n \geq 1$ minimal.

INPUT:

- K – quadratic imaginary field
- c – square free product of inert prime

EXAMPLES:

```
sage: N = 37; D = -7; ell = 17; p = 5
sage: H = heegner_points(N).reduce_mod(ell)
sage: I = H.brandt_module().right_ideals()[49]
sage: f = H.optimal_embeddings(D, 1, I.left_order())[0]
sage: H.kolyvagin_generators(f.domain().number_field(), 5*17)
[-34*a + 1, 35*a + 106]
```

kolyvagin_point_on_curve(*D*, *c*, *E*, *p*, *bound=10*)

Compute image of the Kolyvagin divisor P_c in $E(\mathbf{F}_{\ell^2})/pE(\mathbf{F}_{\ell^2})$.

Note that this image is by definition only well defined up to scalars. However, doing multiple computations will always yield the same result, and working modulo different ℓ is compatible (since we always choose the same generator for $\mathrm{Gal}(K_c/K_1)$).

INPUT:

- D – fundamental negative discriminant
- c – conductor
- E – elliptic curve of conductor the level of self
- p – odd prime number such that we consider image in $E(\mathbf{F}_{\ell^2})/pE(\mathbf{F}_{\ell^2})$
- `bound` – integer (default: 10)

EXAMPLES:

```
sage: N = 37; D = -7; ell = 17; c = 41; p = 3
sage: H = heegner_points(N).reduce_mod(ell)
sage: H.kolyvagin_point_on_curve(D, c, EllipticCurve('37a'), p)
[1, 1]
```

kolyvagin_sigma_operator(*D*, *c*, *r*, *bound=None*)

Return the action of the Kolyvagin sigma operator on the r-th basis vector.

INPUT:

- D – fundamental discriminant
- c – conductor (square-free integer, need not be prime)
- r – nonnegative integer
- `bound` – (default: `None`), if given, controls precision of computation of theta series, which could impact performance, but does not impact correctness

EXAMPLES:

We first try to verify Kolyvagin's conjecture for a rank 2 curve by working modulo 5, but we are unlucky with $c = 17$:

```
sage: N = 389; D = -7; ell = 5; c = 17; q = 3
sage: H = heegner_points(N).reduce_mod(ell)
sage: E = EllipticCurve('389a')
sage: V = H.modp_dual_elliptic_curve_factor(E, q, 5)  # long time (4s on sage.
↪math, 2012)
sage: k118 = H.kolyvagin_sigma_operator(D, c, 118)
sage: k104 = H.kolyvagin_sigma_operator(D, c, 104)
sage: [b.dot_product(k104.element().change_ring(GF(3)))  # long time
....:  for b in V.basis()]
[0, 0]
sage: [b.dot_product(k118.element().change_ring(GF(3)))  # long time
....:  for b in V.basis()]
[0, 0]
```

Next we try again with $c = 41$ and this does work, in that we get something nonzero, when dotting with V:

```
sage: c = 41
sage: k118 = H.kolyvagin_sigma_operator(D, c, 118)
sage: k104 = H.kolyvagin_sigma_operator(D, c, 104)
sage: [b.dot_product(k118.element().change_ring(GF(3)))  # long time
....:  for b in V.basis()]
[2, 0]
sage: [b.dot_product(k104.element().change_ring(GF(3)))  # long time
....:  for b in V.basis()]
[1, 0]
```

By the way, the above is the first ever provable verification of Kolyvagin's conjecture for any curve of rank at least 2.

Another example, but where the curve has rank 1:

```
sage: N = 37; D = -7; ell = 17; c = 41; q = 3
sage: H = heegner_points(N).reduce_mod(ell)
sage: H.heegner_divisor(D,1).element().nonzero_positions()
[49, 51]
sage: k49 = H.kolyvagin_sigma_operator(D, c, 49); k49
(79, 32, 31, 11, 53, 37, 1, 23, 15, 7, 0, 0, 0, 64, 32, 34, 53, 0, 27, 27, 0,␣
↪0, 0, 26, 0, 0, 18, 0, 22, 0, 53, 19, 27, 10, 0, 0, 0, 30, 35, 38, 0, 0, 0,␣
↪53, 0, 0, 4, 0, 0, 0, 0, 0)
sage: k51 = H.kolyvagin_sigma_operator(D, c, 51); k51
(20, 12, 57, 0, 0, 0, 0, 52, 23, 15, 0, 7, 0, 0, 19, 4, 0, 73, 11, 0, 104, 31,
↪ 0, 38, 31, 0, 0, 31, 5, 47, 0, 27, 35, 0, 57, 32, 24, 10, 0, 8, 0, 31, 41,␣
↪0, 0, 0, 16, 0, 0, 0, 0, 0)
sage: V = H.modp_dual_elliptic_curve_factor(EllipticCurve('37a'), q, 5); V
Vector space of degree 52 and dimension 2 over Ring of integers modulo 3
Basis matrix:
2 x 52 dense matrix over Ring of integers modulo 3
sage: [b.dot_product(k49.element().change_ring(GF(q))) for b in V.basis()]
[1, 1]
sage: [b.dot_product(k51.element().change_ring(GF(q))) for b in V.basis()]
[1, 1]
```

An example with c a product of two primes:

```
sage: N = 389; D = -7; ell = 5; q = 3
sage: H = heegner_points(N).reduce_mod(ell)
sage: V = H.modp_dual_elliptic_curve_factor(EllipticCurve('389a'), q, 5)
sage: k = H.kolyvagin_sigma_operator(D, 17*41, 104)      # long time
```

(continues on next page)

(continued from previous page)

```
sage: k                                                    # long time
(990, 656, 219, ..., 246, 534, 1254)
sage: [b.dot_product(k.element().change_ring(GF(3))) for b in V.basis()]   #␣
↪long time (but only because depends on something slow)
[0, 0]
```

left_orders()

Return the left orders associated to the representative right ideals in the Brandt module.

EXAMPLES:

```
sage: heegner_points(11).reduce_mod(3).left_orders()
[Order of Quaternion Algebra (-1, -3) with base ring Rational Field
  with basis (1/2 + 1/2*j + 7*k, 1/2*i + 13/2*k, j + 3*k, 11*k),
 Order of Quaternion Algebra (-1, -3) with base ring Rational Field
  with basis (1/2 + 1/2*j + 7*k, 1/4*i + 1/2*j + 63/4*k, j + 14*k, 22*k)]
```

level()

Return the level.

EXAMPLES:

```
sage: heegner_points(11).reduce_mod(3).level()
11
```

modp_dual_elliptic_curve_factor(*E*, *p*, *bound=10*)

Return the factor of the Brandt module space modulo p corresponding to the elliptic curve E, cut out using Hecke operators up to `bound`.

INPUT:

- E – elliptic curve of conductor equal to the level of `self`
- p – prime number
- `bound` – positive integer (default: 10)

EXAMPLES:

```
sage: N = 37; D = -7; ell = 17; c = 41; q = 3
sage: H = heegner_points(N).reduce_mod(ell)
sage: V = H.modp_dual_elliptic_curve_factor(EllipticCurve('37a'), q, 5); V
Vector space of degree 52 and dimension 2 over Ring of integers modulo 3
 Basis matrix: 2 x 52 dense matrix over Ring of integers modulo 3
```

modp_splitting_data(p)

Return mod p splitting data for the quaternion algebra at the unramified prime p. This is a pair of 2×2 matrices A, B over the finite field $\mathbf{F}_p$ such that if the quaternion algebra has generators i, j, k, then the homomorphism sending i to A and j to B maps any maximal order homomorphically onto the ring of 2×2 matrices.

Because of how the homomorphism is defined, we must assume that the prime p is odd.

INPUT:

- p – unramified odd prime

OUTPUT: A 2-tuple of matrices over finite field.

EXAMPLES:

```
sage: H = heegner_points(11).reduce_mod(7)
sage: H.quaternion_algebra()
Quaternion Algebra (-1, -7) with base ring Rational Field
sage: I, J = H.modp_splitting_data(13)
sage: I
[ 0 12]
[ 1  0]
sage: J
[7 3]
[3 6]
sage: I^2
[12  0]
[ 0 12]
sage: J^2
[6 0]
[0 6]
sage: I*J == -J*I
True
```

The following is a good test because of the asserts in the code:

```
sage: v = [H.modp_splitting_data(p) for p in primes(13,200)]
```

Some edge cases:

```
sage: H.modp_splitting_data(11)
(
[ 0 10]  [6 1]
[ 1  0], [1 5]
)
```

Proper error handling:

```
sage: H.modp_splitting_data(7)
Traceback (most recent call last):
...
ValueError: p (=7) must be an unramified prime

sage: H.modp_splitting_data(2)
Traceback (most recent call last):
...
ValueError: p must be odd
```

modp_splitting_map(*p*)

Return (algebra) map from the (p-integral) quaternion algebra to the set of 2×2 matrices over $\mathbf{F}_p$.

INPUT:

- p – prime number

EXAMPLES:

```
sage: H = heegner_points(11).reduce_mod(7)
sage: f = H.modp_splitting_map(13)
sage: B = H.quaternion_algebra(); B
Quaternion Algebra (-1, -7) with base ring Rational Field
sage: i, j, k = H.quaternion_algebra().gens()
sage: a = 2 + i - j + 3*k; b = 7 + 2*i - 4*j + k
```

(continues on next page)

(continued from previous page)

```
sage: f(a*b)
[12  3]
[10  5]
sage: f(a)*f(b)
[12  3]
[10  5]
```

optimal_embeddings(D, c, R)

INPUT:

- D – negative fundamental discriminant
- c – integer coprime
- R – Eichler order

EXAMPLES:

```
sage: H = heegner_points(11).reduce_mod(3)
sage: R = H.left_orders()[0]
sage: H.optimal_embeddings(-7, 1, R)
[Embedding sending sqrt(-7) to i - j - k,
 Embedding sending sqrt(-7) to -i + j + k]
sage: H.optimal_embeddings(-7, 2, R)
[Embedding sending 2*sqrt(-7) to 5*i - k,
 Embedding sending 2*sqrt(-7) to -5*i + k,
 Embedding sending 2*sqrt(-7) to 2*i - 2*j - 2*k,
 Embedding sending 2*sqrt(-7) to -2*i + 2*j + 2*k]
```

quadratic_field(D)

Return our fixed choice of quadratic imaginary field of discriminant D.

INPUT:

- D – fundamental discriminant

OUTPUT: A quadratic number field.

EXAMPLES:

```
sage: H = heegner_points(389).reduce_mod(5)
sage: H.quadratic_field(-7)
Number Field in sqrt_minus_7 with defining polynomial x^2 + 7
 with sqrt_minus_7 = 2.645751311064591?*I
```

quaternion_algebra()

Return the rational quaternion algebra used to implement self.

EXAMPLES:

```
sage: heegner_points(389).reduce_mod(7).quaternion_algebra()
Quaternion Algebra (-1, -7) with base ring Rational Field
```

rational_kolyvagin_divisor(D, c)

Return the Kolyvagin divisor as an element of the Brandt module corresponding to the discriminant D and conductor c, which both must be coprime to $N\ell$.

INPUT:

- D – discriminant (negative integer)

- c – conductor (positive integer)

OUTPUT: Brandt module element (or tuple of them).

EXAMPLES:

```
sage: N = 389; D = -7; ell = 5; c = 17; q = 3
sage: H = heegner_points(N).reduce_mod(ell)
sage: k = H.rational_kolyvagin_divisor(D, c); k  # long time (5s on sage.math,
↪ 2013)
(2, 0, 0, 0, 0, 0, 16, 0, 0, 0, 0, 4, 0, 0, 9, 11, 0, 6, 0, 0, 7, 0, 0, 0, 0,␣
↪14, 12, 13, 15, 17, 0, 0, 0, 0, 8, 0, 0, 0, 0, 10, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 5, 0, 0, 0, 0, 3, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,␣
↪0, 0, 0, 0, 0, 0, 0)
sage: V = H.modp_dual_elliptic_curve_factor(EllipticCurve('389a'), q, 2)
sage: [b.dot_product(k.element().change_ring(GF(q))) for b in V.basis()]  #␣
↪long time
[0, 0]
sage: k = H.rational_kolyvagin_divisor(D, 59)
sage: [b.dot_product(k.element().change_ring(GF(q))) for b in V.basis()]
[2, 0]
```

right_ideals()

Return representative right ideals in the Brandt module.

EXAMPLES:

```
sage: heegner_points(11).reduce_mod(3).right_ideals()
(Fractional ideal (2 + 2*j + 28*k, 2*i + 26*k, 4*j + 12*k, 44*k),
 Fractional ideal (2 + 2*j + 28*k, 2*i + 4*j + 38*k, 8*j + 24*k, 88*k))
```

satisfies_heegner_hypothesis(D, c=*1*)

The fundamental discriminant D must be coprime to $N\ell$, and must define a quadratic imaginary field K in which ℓ is inert. Also, all primes dividing N must split in K, and c must be squarefree and coprime to $ND\ell$.

INPUT:

- D – negative integer
- c – positive integer (default: 1)

OUTPUT: A boolean.

EXAMPLES:

```
sage: H = heegner_points(11).reduce_mod(7)
sage: H.satisfies_heegner_hypothesis(-5)
False
sage: H.satisfies_heegner_hypothesis(-7)
False
sage: H.satisfies_heegner_hypothesis(-8)
True
sage: [D for D in [-1,-2..-100] if H.satisfies_heegner_hypothesis(D)]
[-8, -39, -43, -51, -79, -95]
```

class sage.schemes.elliptic_curves.heegner.**HeegnerQuatAlgEmbedding**(D, c, R, *beta*)

Bases: SageObject

The homomorphism $\mathcal{O} \to R$, where $\mathcal{O}$ is the order of conductor c in the quadratic field of discriminant D, and R is an Eichler order in a quaternion algebra.

EXAMPLES:

```
sage: H = heegner_points(11).reduce_mod(3); R = H.left_orders()[0]
sage: f = H.optimal_embeddings(-7, 2, R)[1]; f
Embedding sending 2*sqrt(-7) to -5*i + k
sage: type(f)
<class 'sage.schemes.elliptic_curves.heegner.HeegnerQuatAlgEmbedding'>
sage: loads(dumps(f)) == f
True
```

beta()

Return the element β in the quaternion algebra order that $c\sqrt{D}$ maps to.

EXAMPLES:

```
sage: H = heegner_points(11).reduce_mod(3); R = H.left_orders()[0]
sage: H.optimal_embeddings(-7, 2, R)[1].beta()
-5*i + k
```

codomain()

Return the codomain of this embedding.

EXAMPLES:

```
sage: H = heegner_points(11).reduce_mod(3); R = H.left_orders()[0]
sage: H.optimal_embeddings(-7, 2, R)[0].codomain()
Order of Quaternion Algebra (-1, -3) with base ring Rational Field
 with basis (1/2 + 1/2*j + 7*k, 1/2*i + 13/2*k, j + 3*k, 11*k)
```

conjugate()

Return the conjugate of this embedding, which is also an embedding.

EXAMPLES:

```
sage: H = heegner_points(11).reduce_mod(3); R = H.left_orders()[0]
sage: f = H.optimal_embeddings(-7, 2, R)[1]
sage: f.conjugate()
Embedding sending 2*sqrt(-7) to 5*i - k
sage: f
Embedding sending 2*sqrt(-7) to -5*i + k
```

domain()

Return the domain of this embedding.

EXAMPLES:

```
sage: H = heegner_points(11).reduce_mod(3); R = H.left_orders()[0]
sage: H.optimal_embeddings(-7, 2, R)[0].domain()
Order of conductor 4 generated by 2*a
 in Number Field in a with defining polynomial x^2 + 7
 with a = 2.645751311064591?*I
```

domain_conductor()

Return the conductor of the domain.

EXAMPLES:

```
sage: H = heegner_points(11).reduce_mod(3); R = H.left_orders()[0]
sage: H.optimal_embeddings(-7, 2, R)[0].domain_conductor()
2
```

domain_gen()

Return the specific generator $c\sqrt{D}$ for the domain order.

EXAMPLES:

```
sage: H = heegner_points(11).reduce_mod(3); R = H.left_orders()[0]
sage: f = H.optimal_embeddings(-7, 2, R)[0]
sage: f.domain_gen()
2*a
sage: f.domain_gen()^2
-28
```

matrix()

Return matrix over $\mathbf{Q}$ of this morphism, with respect to the basis $1, c\sqrt{D}$ of the domain and the basis $1, i, j, k$ of the ambient rational quaternion algebra (which contains the domain).

EXAMPLES:

```
sage: H = heegner_points(11).reduce_mod(3); R = H.left_orders()[0]
sage: f = H.optimal_embeddings(-7, 1, R)[1]; f
Embedding sending sqrt(-7) to -i + j + k
sage: f.matrix()
[ 1  0  0  0]
[ 0 -1  1  1]
sage: f.conjugate().matrix()
[ 1  0  0  0]
[ 0  1 -1 -1]
```

class sage.schemes.elliptic_curves.heegner.**KolyvaginCohomologyClass**(*kolyvagin_point*, *n*)

Bases: SageObject

A Kolyvagin cohomology class in $H^1(K, E[n])$ or $H^1(K, E)[n]$ attached to a Heegner point.

EXAMPLES:

```
sage: y = EllipticCurve('37a').heegner_point(-7)
sage: c = y.kolyvagin_cohomology_class(3); c
Kolyvagin cohomology class c(1) in H^1(K,E[3])
sage: type(c)
<class 'sage.schemes.elliptic_curves.heegner.KolyvaginCohomologyClassEn'>
sage: loads(dumps(c)) == c
True
sage: y.kolyvagin_cohomology_class(5)
Kolyvagin cohomology class c(1) in H^1(K,E[5])
```

conductor()

Return the integer c such that this cohomology class is associated to the Heegner point y_c.

EXAMPLES:

```
sage: y = EllipticCurve('37a').heegner_point(-7, 5)
sage: t = y.kolyvagin_cohomology_class()
```

(continues on next page)

(continued from previous page)

```
sage: t.conductor()
5
```

heegner_point()

Return the Heegner point y_c to which this cohomology class is associated.

EXAMPLES:

```
sage: y = EllipticCurve('37a').heegner_point(-7, 5)
sage: t = y.kolyvagin_cohomology_class()
sage: t.heegner_point()
Heegner point of discriminant -7 and conductor 5
 on elliptic curve of conductor 37
```

kolyvagin_point()

Return the Kolyvagin point P_c to which this cohomology class is associated.

EXAMPLES:

```
sage: y = EllipticCurve('37a').heegner_point(-7, 5)
sage: t = y.kolyvagin_cohomology_class()
sage: t.kolyvagin_point()
Kolyvagin point of discriminant -7 and conductor 5
 on elliptic curve of conductor 37
```

n()

Return the integer n so that this is a cohomology class in $H^1(K, E[n])$ or $H^1(K, E)[n]$.

EXAMPLES:

```
sage: y = EllipticCurve('37a').heegner_point(-7)
sage: t = y.kolyvagin_cohomology_class(3); t
Kolyvagin cohomology class c(1) in H^1(K,E[3])
sage: t.n()
3
```

class sage.schemes.elliptic_curves.heegner.**KolyvaginCohomologyClassEn**(*kolyvagin_point*, *n*)

Bases: *KolyvaginCohomologyClass*

EXAMPLES:

class sage.schemes.elliptic_curves.heegner.**KolyvaginPoint**(*heegner_point*)

Bases: *HeegnerPoint*

A Kolyvagin point.

EXAMPLES:

We create a few Kolyvagin points:

```
sage: EllipticCurve('11a1').kolyvagin_point(-7)
Kolyvagin point of discriminant -7 on elliptic curve of conductor 11
sage: EllipticCurve('37a1').kolyvagin_point(-7)
Kolyvagin point of discriminant -7 on elliptic curve of conductor 37
sage: EllipticCurve('37a1').kolyvagin_point(-67)
```

(continues on next page)

(continued from previous page)

```
Kolyvagin point of discriminant -67 on elliptic curve of conductor 37
sage: EllipticCurve('389a1').kolyvagin_point(-7, 5)
Kolyvagin point of discriminant -7 and conductor 5
 on elliptic curve of conductor 389
```

One can also associated a Kolyvagin point to a Heegner point:

```
sage: y = EllipticCurve('37a1').heegner_point(-7); y
Heegner point of discriminant -7 on elliptic curve of conductor 37
sage: y.kolyvagin_point()
Kolyvagin point of discriminant -7 on elliptic curve of conductor 37
```

curve()

Return the elliptic curve over $\mathbf{Q}$ on which this Kolyvagin point sits.

EXAMPLES:

```
sage: E = EllipticCurve('37a1'); P = E.kolyvagin_point(-67, 3)
sage: P.curve()
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
```

heegner_point()

This Kolyvagin point P_c is associated to some Heegner point y_c via Kolyvagin's construction. This function returns that point y_c.

EXAMPLES:

```
sage: E = EllipticCurve('37a1')
sage: P = E.kolyvagin_point(-67); P
Kolyvagin point of discriminant -67 on elliptic curve of conductor 37
sage: y = P.heegner_point(); y
Heegner point of discriminant -67 on elliptic curve of conductor 37
sage: y.kolyvagin_point() is P
True
```

index(**args*, ***kwds*)

Return index of this Kolyvagin point in the full group of K_c rational points on E.

When the conductor is 1, this is computed numerically using the Gross-Zagier formula and explicit point search, and it may be off by 2. See the documentation for `E.heegner_index`, where E is the curve attached to `self`.

EXAMPLES:

```
sage: E = EllipticCurve('37a1'); P = E.kolyvagin_point(-67); P.index()
6
```

kolyvagin_cohomology_class(*n=None*)

INPUT:

- n – positive integer that divides the gcd of a_p and $p+1$ for all p dividing the conductor. If n is `None`, choose the largest valid n.

EXAMPLES:

```
sage: y = EllipticCurve('389a').heegner_point(-7, 5)
sage: P = y.kolyvagin_point()
sage: P.kolyvagin_cohomology_class(3)
Kolyvagin cohomology class c(5) in H^1(K,E[3])

sage: y = EllipticCurve('37a').heegner_point(-7, 5).kolyvagin_point()
sage: y.kolyvagin_cohomology_class()
Kolyvagin cohomology class c(5) in H^1(K,E[2])
```

mod(*p*, *prec=53*)

Return the trace of the reduction Q modulo a prime over p of this Kolyvagin point as an element of $E(\mathbf{F}_p)$, where p is any prime that is inert in K that is coprime to NDc.

The point Q is only well defined up to an element of $(p+1)E(\mathbf{F}_p)$, i.e., it gives a well defined element of the abelian group $E(\mathbf{F}_p)/(p+1)E(\mathbf{F}_p)$.

See [St2011b], Proposition 5.4 for a proof of the above well-definedness assertion.

EXAMPLES:

A Kolyvagin point on a rank 1 curve:

```
sage: E = EllipticCurve('37a1'); P = E.kolyvagin_point(-67)
sage: P.mod(2)
(1 : 1 : 1)
sage: P.mod(3)
(1 : 0 : 1)
sage: P.mod(5)
(2 : 2 : 1)
sage: P.mod(7)
(6 : 0 : 1)
sage: P.trace_to_real_numerical()
(1.61355529131986 : -2.18446840788880 : 1.00000000000000)
sage: P._trace_exact_conductor_1()  # the actual point we're reducing
(1357/841 : -53277/24389 : 1)
sage: (P._trace_exact_conductor_1().height() / E.regulator()).sqrt()
12.0000000000000
```

Here the Kolyvagin point is a torsion point (since E has rank 1), and we reduce it modulo several primes.:

```
sage: E = EllipticCurve('11a1'); P = E.kolyvagin_point(-7)
sage: P.mod(3,70)  # long time (4s on sage.math, 2013)
(1 : 2 : 1)
sage: P.mod(5,70)
(1 : 4 : 1)
sage: P.mod(7,70)
Traceback (most recent call last):
...
ValueError: p must be coprime to conductors and discriminant
sage: P.mod(11,70)
Traceback (most recent call last):
...
ValueError: p must be coprime to conductors and discriminant
sage: P.mod(13,70)
(3 : 4 : 1)
```

numerical_approx(*prec=53*)

Return a numerical approximation to this Kolyvagin point using prec bits of working precision.

INPUT:

- `prec` – precision in bits (default: 53)

EXAMPLES:

```
sage: P = EllipticCurve('37a1').kolyvagin_point(-7); P
Kolyvagin point of discriminant -7 on elliptic curve of conductor 37
sage: P.numerical_approx() # approx. (0 : 0 : 1)
(...e-16 - ...e-16*I : ...e-16 + ...e-16*I : 1.00000000000000)
sage: P.numerical_approx(100)[0].abs() < 2.0^-99
True

sage: P = EllipticCurve('389a1').kolyvagin_point(-7, 5); P
Kolyvagin point of discriminant -7 and conductor 5
 on elliptic curve of conductor 389
```

Numerical approximation is only implemented for points of conductor 1:

```
sage: P.numerical_approx()
Traceback (most recent call last):
...
NotImplementedError
```

plot (*prec=53*, **args*, ***kwds*)

Plot a Kolyvagin point P_1 if it is defined over the rational numbers.

EXAMPLES:

```
sage: E = EllipticCurve('37a'); P = E.heegner_point(-11).kolyvagin_point()
sage: P.plot(prec=30, pointsize=50, rgbcolor='red') + E.plot()              #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
```

point_exact (*prec=53*)

INPUT:

- `prec` – precision in bits (default: 53)

EXAMPLES:

A rank 1 curve:

```
sage: E = EllipticCurve('37a1'); P = E.kolyvagin_point(-67)
sage: P.point_exact()
(6 : -15 : 1)
sage: P.point_exact(40)
(6 : -15 : 1)
sage: P.point_exact(20)
Traceback (most recent call last):
...
RuntimeError: insufficient precision to find exact point
```

A rank 0 curve:

```
sage: E = EllipticCurve('11a1'); P = E.kolyvagin_point(-7)
sage: P.point_exact()
(-1/2*sqrt_minus_7 + 1/2 : -2*sqrt_minus_7 - 2 : 1)
```

A rank 2 curve:

```
sage: E = EllipticCurve('389a1'); P = E.kolyvagin_point(-7)
sage: P.point_exact()
(0 : 1 : 0)
```

satisfies_kolyvagin_hypothesis(*n=None*)

Return `True` if this Kolyvagin point satisfies the Heegner hypothesis for n, so that it defines a Galois equivariant element of $E(K_c)/nE(K_c)$.

EXAMPLES:

```
sage: y = EllipticCurve('389a').heegner_point(-7,5); P = y.kolyvagin_point()
sage: P.kolyvagin_cohomology_class(3)
Kolyvagin cohomology class c(5) in H^1(K,E[3])
sage: P.satisfies_kolyvagin_hypothesis(3)
True
sage: P.satisfies_kolyvagin_hypothesis(5)
False
sage: P.satisfies_kolyvagin_hypothesis(7)
False
sage: P.satisfies_kolyvagin_hypothesis(11)
False
```

trace_to_real_numerical(*prec=53*)

Return the trace of this Kolyvagin point down to the real numbers, computed numerically using prec bits of working precision.

EXAMPLES:

```
sage: E = EllipticCurve('37a1'); P = E.kolyvagin_point(-67)
sage: PP = P.numerical_approx()
sage: [c.real() for c in PP]
[6.00000000000000, -15.0000000000000, 1.00000000000000]
sage: all(c.imag().abs() < 1e-14 for c in PP)
True
sage: P.trace_to_real_numerical()
(1.61355529131986 : -2.18446840788880 : 1.00000000000000)
sage: P.trace_to_real_numerical(prec=80)  # abs tol 1e-21
(1.6135552913198573127230 : -2.184468407888802328 9187 : 1.
↪00000000000000000000)
```

class sage.schemes.elliptic_curves.heegner.**RingClassField**(*D*, *c*, *check=True*)

Bases: SageObject

A Ring class field of a quadratic imaginary field of given conductor.

Note: This is a *ring* class field, not a ray class field. In general, the ring class field of given conductor is a subfield of the ray class field of the same conductor.

EXAMPLES:

```
sage: heegner_point(37,-7).ring_class_field()
Hilbert class field of QQ[sqrt(-7)]
sage: heegner_point(37,-7,5).ring_class_field()
Ring class field extension of QQ[sqrt(-7)] of conductor 5
sage: heegner_point(37,-7,55).ring_class_field()
Ring class field extension of QQ[sqrt(-7)] of conductor 55
```

absolute_degree()

Return the absolute degree of this field over $\mathbf{Q}$.

EXAMPLES:

```
sage: E = EllipticCurve('389a'); K = E.heegner_point(-7,5).ring_class_field()
sage: K.absolute_degree()
12
sage: K.degree_over_K()
6
```

conductor()

Return the conductor of this ring class field.

EXAMPLES:

```
sage: E = EllipticCurve('389a'); K5 = E.heegner_point(-7,5).ring_class_field()
sage: K5.conductor()
5
```

degree_over_H()

Return the degree of this field over the Hilbert class field H of K.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: E.heegner_point(-59).ring_class_field().degree_over_H()
1
sage: E.heegner_point(-59).ring_class_field().degree_over_K()
3
sage: QuadraticField(-59,'a').class_number()
3
```

Some examples in which prime dividing c is inert:

```
sage: heegner_point(37,-7,3).ring_class_field().degree_over_H()
4
sage: heegner_point(37,-7,3^2).ring_class_field().degree_over_H()
12
sage: heegner_point(37,-7,3^3).ring_class_field().degree_over_H()
36
```

The prime dividing c is split. For example, in the first case O_K/cO_K is isomorphic to a direct sum of two copies of `GF(2)`, so the units are trivial:

```
sage: heegner_point(37,-7,2).ring_class_field().degree_over_H()
1
sage: heegner_point(37,-7,4).ring_class_field().degree_over_H()
2
sage: heegner_point(37,-7,8).ring_class_field().degree_over_H()
4
```

Now c is ramified:

```
sage: heegner_point(37,-7,7).ring_class_field().degree_over_H()
7
sage: heegner_point(37,-7,7^2).ring_class_field().degree_over_H()
49
```

Check that Issue #15218 is solved:

```
sage: E = EllipticCurve("19a");
sage: s = E.heegner_point(-3,2).ring_class_field().galois_group().complex_
↪conjugation()
sage: H = s.domain(); H.absolute_degree()
2
```

degree_over_K()

Return the relative degree of this ring class field over the quadratic imaginary field K.

EXAMPLES:

```
sage: E = EllipticCurve('389a'); P = E.heegner_point(-7,5)
sage: K5 = P.ring_class_field(); K5
Ring class field extension of QQ[sqrt(-7)] of conductor 5
sage: K5.degree_over_K()
6
sage: type(K5.degree_over_K())
<... 'sage.rings.integer.Integer'>

sage: E = EllipticCurve('389a'); E.heegner_point(-20).ring_class_field().
↪degree_over_K()
2
sage: E.heegner_point(-20,3).ring_class_field().degree_over_K()
4
sage: kronecker(-20,11)
-1
sage: E.heegner_point(-20,11).ring_class_field().degree_over_K()
24
```

degree_over_Q()

Return the absolute degree of this field over $\mathbf{Q}$.

EXAMPLES:

```
sage: E = EllipticCurve('389a'); K = E.heegner_point(-7,5).ring_class_field()
sage: K.absolute_degree()
12
sage: K.degree_over_K()
6
```

discriminant_of_K()

Return the discriminant of the quadratic imaginary field K contained in `self`.

EXAMPLES:

```
sage: E = EllipticCurve('389a'); K5 = E.heegner_point(-7,5).ring_class_field()
sage: K5.discriminant_of_K()
-7
```

galois_group(*base=Rational Field*)

Return the Galois group of `self` over base.

INPUT:

- `base` – (default: $\mathbf{Q}$) a subfield of `self` or $\mathbf{Q}$

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: A = E.heegner_point(-7,5).ring_class_field()
sage: A.galois_group()
Galois group of Ring class field extension of QQ[sqrt(-7)] of conductor 5
sage: B = E.heegner_point(-7).ring_class_field()
sage: C = E.heegner_point(-7,15).ring_class_field()
sage: A.galois_group()
Galois group of Ring class field extension of QQ[sqrt(-7)] of conductor 5
sage: A.galois_group(B)
Galois group of Ring class field extension of QQ[sqrt(-7)] of conductor 5
 over Hilbert class field of QQ[sqrt(-7)]
sage: A.galois_group().cardinality()
12
sage: A.galois_group(B).cardinality()
6
sage: C.galois_group(A)
Galois group of Ring class field extension of QQ[sqrt(-7)] of conductor 15
 over Ring class field extension of QQ[sqrt(-7)] of conductor 5
sage: C.galois_group(A).cardinality()
4
```

is_subfield(*M*)

Return `True` if this ring class field is a subfield of the ring class field M. If M is not a ring class field, then a `TypeError` is raised.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: A = E.heegner_point(-7,5).ring_class_field()
sage: B = E.heegner_point(-7).ring_class_field()
sage: C = E.heegner_point(-20).ring_class_field()
sage: D = E.heegner_point(-7,15).ring_class_field()
sage: B.is_subfield(A)
True
sage: B.is_subfield(B)
True
sage: B.is_subfield(D)
True
sage: B.is_subfield(C)
False
sage: A.is_subfield(B)
False
sage: A.is_subfield(D)
True
```

quadratic_field()

Return the quadratic imaginary field $K = \mathbf{Q}(\sqrt{D})$.

EXAMPLES:

```
sage: E = EllipticCurve('389a'); K = E.heegner_point(-7,5).ring_class_field()
sage: K.quadratic_field()
Number Field in sqrt_minus_7 with defining polynomial x^2 + 7
 with sqrt_minus_7 = 2.645751311064591?*I
```

ramified_primes()

Return the primes of $\mathbf{Z}$ that ramify in this ring class field.

EXAMPLES:

```
sage: E = EllipticCurve('389a'); K55 = E.heegner_point(-7,55).ring_class_
↪field()
sage: K55.ramified_primes()
[5, 7, 11]
sage: E.heegner_point(-7).ring_class_field().ramified_primes()
[7]
```

sage.schemes.elliptic_curves.heegner.**class_number**(*D*)

Return the class number of the quadratic field with fundamental discriminant D.

INPUT:

- D – integer

EXAMPLES:

```
sage: sage.schemes.elliptic_curves.heegner.class_number(-20)
2
sage: sage.schemes.elliptic_curves.heegner.class_number(-23)
3
sage: sage.schemes.elliptic_curves.heegner.class_number(-163)
1
```

A `ValueError` is raised when D is not a fundamental discriminant:

```
sage: sage.schemes.elliptic_curves.heegner.class_number(-5)
Traceback (most recent call last):
...
ValueError: D (=-5) must be a fundamental discriminant
```

sage.schemes.elliptic_curves.heegner.**ell_heegner_discriminants**(*self*, *bound*)

Return the list of self's Heegner discriminants between -1 and -bound.

INPUT:

- `bound (int)` – upper bound for -discriminant

OUTPUT: The list of Heegner discriminants between -1 and -bound for the given elliptic curve.

EXAMPLES:

```
sage: E=EllipticCurve('11a')
sage: E.heegner_discriminants(30)                    # indirect doctest
[-7, -8, -19, -24]
```

sage.schemes.elliptic_curves.heegner.**ell_heegner_discriminants_list**(*self*, *n*)

Return the list of self's first n Heegner discriminants smaller than -5.

INPUT:

- `n (int)` – the number of discriminants to compute

OUTPUT: The list of the first n Heegner discriminants smaller than -5 for the given elliptic curve.

EXAMPLES:

```
sage: E=EllipticCurve('11a')
sage: E.heegner_discriminants_list(4)                    # indirect doctest
[-7, -8, -19, -24]
```

sage.schemes.elliptic_curves.heegner.**ell_heegner_point**(*self*, *D*, *c=1*, *f=None*, *check=True*)

Returns the Heegner point on this curve associated to the quadratic imaginary field $K = \mathbf{Q}(\sqrt{D})$.

If the optional parameter c is given, returns the higher Heegner point associated to the order of conductor c.

INPUT:

- D – a Heegner discriminant
- c – (default: 1) conductor, must be coprime to DN
- f – binary quadratic form or 3-tuple (A, B, C) of coefficients of $AX^2 + BXY + CY^2$
- `check` – bool (default: `True`)

OUTPUT: The Heegner point y_c.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.heegner_discriminants_list(10)
[-7, -11, -40, -47, -67, -71, -83, -84, -95, -104]
sage: P = E.heegner_point(-7); P                              # indirect doctest
Heegner point of discriminant -7 on elliptic curve of conductor 37
sage: z = P.point_exact(); z == E(0, 0, 1)  or -z == E(0, 0, 1)
True
sage: P.curve()
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
sage: P = E.heegner_point(-40).point_exact(); P
(a : -a + 1 : 1)
sage: P = E.heegner_point(-47).point_exact(); P
(a : a^4 + a - 1 : 1)
sage: P[0].parent()
Number Field in a with defining polynomial x^5 - x^4 + x^3 + x^2 - 2*x + 1
```

Working out the details manually:

```
sage: P = E.heegner_point(-47).numerical_approx(prec=200)
sage: f = algdep(P[0], 5); f
x^5 - x^4 + x^3 + x^2 - 2*x + 1
sage: f.discriminant().factor()
47^2
```

The Heegner hypothesis is checked:

```
sage: E = EllipticCurve('389a'); P = E.heegner_point(-5,7);
Traceback (most recent call last):
...
ValueError: N (=389) and D (=-5) must satisfy the Heegner hypothesis
```

We can specify the quadratic form:

```
sage: P = EllipticCurve('389a').heegner_point(-7, 5, (778,925,275)); P
Heegner point of discriminant -7 and conductor 5
 on elliptic curve of conductor 389
sage: P.quadratic_form()
778*x^2 + 925*x*y + 275*y^2
```

sage.schemes.elliptic_curves.heegner.**heegner_index**(*self*, *D*, *min_p=2*, *prec=5*, *descent_second_limit=12*, *verbose_mwrank=False*, *check_rank=True*)

Return an interval that contains the index of the Heegner point y_K in the group of K-rational points modulo torsion on this elliptic curve, computed using the Gross-Zagier formula and/or a point search, or possibly half the index if the rank is greater than one.

If the curve has rank > 1, then the returned index is infinity.

Note: If `min_p` is bigger than 2 then the index can be off by any prime less than `min_p`. This function returns the index divided by 2 exactly when the rank of $E(K)$ is greater than 1 and $E(\mathbf{Q})_{/tor} \oplus E^D(\mathbf{Q})_{/tor}$ has index 2 in $E(K)_{/tor}$, where the second factor undergoes a twist.

INPUT:

- `D (int)` – Heegner discriminant
- `min_p (int)` – (default: 2) only rule out primes = min_p dividing the index.
- `verbose_mwrank (bool)` – (default: `False`); print lots of mwrank search status information when computing regulator
- `prec (int)` – (default: 5), use prec*sqrt(N) + 20 terms of L-series in computations, where N is the conductor.
- `descent_second_limit` – (default: 12)- used in 2-descent when computing regulator of the twist
- `check_rank` – whether to check if the rank is at least 2 by computing the Mordell-Weil rank directly.

OUTPUT: an interval that contains the index, or half the index

EXAMPLES:

```
sage: E = EllipticCurve('11a')
sage: E.heegner_discriminants(50)
[-7, -8, -19, -24, -35, -39, -40, -43]
sage: E.heegner_index(-7)
1.00000?
```

```
sage: E = EllipticCurve('37b')
sage: E.heegner_discriminants(100)
[-3, -4, -7, -11, -40, -47, -67, -71, -83, -84, -95]
sage: E.heegner_index(-95)          # long time (1 second)
2.00000?
```

This tests doing direct computation of the Mordell-Weil group.

```
sage: EllipticCurve('675b').heegner_index(-11)
3.0000?
```

Currently discriminants -3 and -4 are not supported:

```
sage: E.heegner_index(-3)
Traceback (most recent call last):
...
ArithmeticError: Discriminant (=-3) must not be -3 or -4.
```

The curve 681b returns the true index, which is 3:

```
sage: E = EllipticCurve('681b')
sage: I = E.heegner_index(-8); I
3.0000?
```

In fact, whenever the returned index has a denominator of 2, the true index is got by multiplying the returned index by 2. Unfortunately, this is not an if and only if condition, i.e., sometimes the index must be multiplied by 2 even though the denominator is not 2.

This example demonstrates the `descent_second_limit` option, which can be used to fine tune the 2-descent used to compute the regulator of the twist:

```
sage: E = EllipticCurve([1,-1,0,-1228,-16267])
sage: E.heegner_index(-8)
Traceback (most recent call last):
...
RuntimeError: ...
```

However when we search higher, we find the points we need:

```
sage: E.heegner_index(-8, descent_second_limit=16, check_rank=False)  # long time
2.00000?
```

Two higher rank examples (of ranks 2 and 3):

```
sage: E = EllipticCurve('389a')
sage: E.heegner_index(-7)
+Infinity
sage: E = EllipticCurve('5077a')
sage: E.heegner_index(-7)
+Infinity
sage: E.heegner_index(-7, check_rank=False)
0.001?
sage: E.heegner_index(-7, check_rank=False).lower() == 0
True
```

sage.schemes.elliptic_curves.heegner.**heegner_index_bound**(*self*, *D=0*, *prec=5*, *max_height=None*)

Assume `self` has rank 0.

Return a list v of primes such that if an odd prime p divides the index of the Heegner point in the group of rational points modulo torsion, then p is in v.

If 0 is in the interval of the height of the Heegner point computed to the given prec, then this function returns $v = 0$. This does not mean that the Heegner point is torsion, just that it is very likely torsion.

If we obtain no information from a search up to `max_height`, e.g., if the Siksek et al. bound is bigger than `max_height`, then we return $v = -1$.

INPUT:

- `D (int)` – (default: 0) Heegner discriminant; if 0, use the first discriminant -4 that satisfies the Heegner hypothesis
- `verbose (bool)` – (default: `True`)
- `prec (int)` – (default: 5), use $prec \cdot \sqrt(N) + 20$ terms of L-series in computations, where N is the conductor.

- max_height (float) – should be = 21; bound on logarithmic naive height used in point searches. Make smaller to make this function faster, at the expense of possibly obtaining a worse answer. A good range is between 13 and 21.

OUTPUT:

- v – list or int (bad primes or 0 or -1)
- D – the discriminant that was used (this is useful if D was automatically selected).
- exact – either False, or the exact Heegner index (up to factors of 2)

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: E.heegner_index_bound()
([2], -7, 2)
```

sage.schemes.elliptic_curves.heegner.**heegner_point**(*N*, *D=None*, *c=1*)

Return a specific Heegner point of level N with given discriminant and conductor. If D is not specified, then the first valid Heegner discriminant is used. If c is not given, then $c = 1$ is used.

INPUT:

- N – level (positive integer)
- D – discriminant (optional: default first valid D)
- c – conductor (positive integer, default: 1)

EXAMPLES:

```
sage: heegner_point(389)
Heegner point 1/778*sqrt(-7) - 185/778 of discriminant -7 on X_0(389)
sage: heegner_point(389,-7)
Heegner point 1/778*sqrt(-7) - 185/778 of discriminant -7 on X_0(389)
sage: heegner_point(389,-7,5)
Heegner point 5/778*sqrt(-7) - 147/778 of discriminant -7 and conductor 5 on X_
↪0(389)
sage: heegner_point(389,-20)
Heegner point 1/778*sqrt(-20) - 165/389 of discriminant -20 on X_0(389)
```

sage.schemes.elliptic_curves.heegner.**heegner_point_height**(*self*, *D*, *prec=2*, *check_rank=True*)

Use the Gross-Zagier formula to compute the Neron-Tate canonical height over K of the Heegner point corresponding to D, as an interval (it is computed to some precision using L-functions).

If the curve has rank at least 2, then the returned height is the exact Sage integer 0.

INPUT:

- D (int) – fundamental discriminant (=/= -3, -4)
- prec (int) – (default: 2), use $prec \cdot \sqrt(N) + 20$ terms of L-series in computations, where N is the conductor.
- check_rank – whether to check if the rank is at least 2 by computing the Mordell-Weil rank directly.

OUTPUT: Interval that contains the height of the Heegner point.

EXAMPLES:

```
sage: E = EllipticCurve('11a')
sage: E.heegner_point_height(-7)
0.22227?
```

Some higher rank examples:

```
sage: E = EllipticCurve('389a')
sage: E.heegner_point_height(-7)
0
sage: E = EllipticCurve('5077a')
sage: E.heegner_point_height(-7)
0
sage: E.heegner_point_height(-7, check_rank=False)
0.0000?
```

sage.schemes.elliptic_curves.heegner.**heegner_points**(*N*, *D=None*, *c=None*)

Return all Heegner points of given level N. Can also restrict to Heegner points with specified discriminant D and optionally conductor c.

INPUT:

- N – level (positive integer)
- D – discriminant (negative integer)
- c – conductor (positive integer)

EXAMPLES:

```
sage: heegner_points(389, -7)
Set of all Heegner points on X_0(389) associated to QQ[sqrt(-7)]
sage: heegner_points(389, -7, 1)
All Heegner points of conductor 1 on X_0(389) associated to QQ[sqrt(-7)]
sage: heegner_points(389, -7, 5)
All Heegner points of conductor 5 on X_0(389) associated to QQ[sqrt(-7)]
```

sage.schemes.elliptic_curves.heegner.**heegner_sha_an**(*self*, *D*, *prec=53*)

Return the conjectural (analytic) order of Sha for E over the field $K = \mathbf{Q}(\sqrt{D})$.

INPUT:

- D – negative integer; the Heegner discriminant
- prec – integer (default: 53); bits of precision to compute analytic order of Sha

OUTPUT:

(floating point number) an approximation to the conjectural order of Sha.

Note: Often you'll want to do `proof.elliptic_curve(False)` when using this function, since often the twisted elliptic curves that come up have enormous conductor, and Sha is nontrivial, which makes provably finding the Mordell-Weil group using 2-descent difficult.

EXAMPLES:

An example where E has conductor 11:

```
sage: E = EllipticCurve('11a')
sage: E.heegner_sha_an(-7)                                    # long time
1.00000000000000
```

The cache works:

```
sage: E.heegner_sha_an(-7) is E.heegner_sha_an(-7)            # long time
True
```

Lower precision:

```
sage: E.heegner_sha_an(-7,10)                                 # long time
1.0
```

Checking that the cache works for any precision:

```
sage: E.heegner_sha_an(-7,10) is E.heegner_sha_an(-7,10)      # long time
True
```

Next we consider a rank 1 curve with nontrivial Sha over the quadratic imaginary field K; however, there is no Sha for E over $\mathbf{Q}$ or for the quadratic twist of E:

```
sage: E = EllipticCurve('37a')
sage: E.heegner_sha_an(-40)                                   # long time
4.00000000000000
sage: E.quadratic_twist(-40).sha().an()                       # long time
1
sage: E.sha().an()                                            # long time
1
```

A rank 2 curve:

```
sage: E = EllipticCurve('389a')                               # long time
sage: E.heegner_sha_an(-7)                                    # long time
1.00000000000000
```

If we remove the hypothesis that $E(K)$ has rank 1 in Conjecture 2.3 in [GZ1986] page 311, then that conjecture is false, as the following example shows:

```
sage: # long time
sage: E = EllipticCurve('65a')
sage: E.heegner_sha_an(-56)
1.00000000000000
sage: E.torsion_order()
2
sage: E.tamagawa_product()
1
sage: E.quadratic_twist(-56).rank()
2
```

sage.schemes.elliptic_curves.heegner.**is_inert**(D, p)

Return `True` if p is an inert prime in the field $\mathbf{Q}(\sqrt{D})$.

INPUT:

- D – fundamental discriminant
- p – prime integer

EXAMPLES:

```
sage: sage.schemes.elliptic_curves.heegner.is_inert(-7,3)
True
sage: sage.schemes.elliptic_curves.heegner.is_inert(-7,7)
False
sage: sage.schemes.elliptic_curves.heegner.is_inert(-7,11)
False
```

sage.schemes.elliptic_curves.heegner.**is_kolyvagin_conductor**(N, E, D, r, n, c)

Return `True` if c is a Kolyvagin conductor for level N, discriminant D, mod n, etc., i.e., c is divisible by exactly r prime factors, is coprime to ND, each prime dividing c is inert, and if E is not `None` then $n | \gcd(p+1, a_p(E))$ for each prime p dividing c.

INPUT:

- N – level (positive integer)
- E – elliptic curve or `None`
- D – negative fundamental discriminant
- r – number of prime factors (nonnegative integer) or `None`
- n – torsion order (i.e., do we get class in $(E(K_c)/nE(K_c))^{Gal(K_c/K)}$?)
- c – conductor (positive integer)

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.heegner import is_kolyvagin_conductor
sage: is_kolyvagin_conductor(389, None, -7, 1, None, 5)
True
sage: is_kolyvagin_conductor(389, None, -7, 1, None, 7)
False
sage: is_kolyvagin_conductor(389, None, -7, 1, None, 11)
False
sage: is_kolyvagin_conductor(389, EllipticCurve('389a'), -7, 1, 3, 5)
True
sage: is_kolyvagin_conductor(389, EllipticCurve('389a'), -7, 1, 11, 5)
False
```

sage.schemes.elliptic_curves.heegner.**is_ramified**(D, p)

Return `True` if p is a ramified prime in the field $\mathbf{Q}(\sqrt{D})$.

INPUT:

- D – fundamental discriminant
- p – prime integer

EXAMPLES:

```
sage: sage.schemes.elliptic_curves.heegner.is_ramified(-7,2)
False
sage: sage.schemes.elliptic_curves.heegner.is_ramified(-7,7)
True
sage: sage.schemes.elliptic_curves.heegner.is_ramified(-1,2)
True
```

sage.schemes.elliptic_curves.heegner.**is_split**(*D*, *p*)

Return `True` if p is a split prime in the field $\mathbf{Q}(\sqrt{D})$.

INPUT:

- D – fundamental discriminant
- p – prime integer

EXAMPLES:

```
sage: sage.schemes.elliptic_curves.heegner.is_split(-7,3)
False
sage: sage.schemes.elliptic_curves.heegner.is_split(-7,7)
False
sage: sage.schemes.elliptic_curves.heegner.is_split(-7,11)
True
```

sage.schemes.elliptic_curves.heegner.**kolyvagin_point**(*self*, *D*, *c=1*, *check=True*)

Return the Kolyvagin point on this curve associated to the quadratic imaginary field $K = \mathbf{Q}(\sqrt{D})$ and conductor c.

INPUT:

- D – a Heegner discriminant
- c – (default: 1) conductor, must be coprime to DN
- `check` – bool (default: `True`)

OUTPUT: The Kolyvagin point P of conductor c.

EXAMPLES:

```
sage: E = EllipticCurve('37a1')
sage: P = E.kolyvagin_point(-67); P
Kolyvagin point of discriminant -67 on elliptic curve of conductor 37
sage: P.numerical_approx()  # abs tol 1e-14
(6.00000000000000 : -15.0000000000000 : 1.00000000000000)
sage: P.index()
6
sage: g = E((0,-1,1)) # a generator
sage: E.regulator() == E.regulator_of_points([g])
True
sage: 6*g
(6 : -15 : 1)
```

sage.schemes.elliptic_curves.heegner.**kolyvagin_reduction_data**(*E*, *q*, *first_only=True*)

Given an elliptic curve of positive rank and a prime q, this function returns data about how to use Kolyvagin's q-torsion Heegner point Euler system to do computations with this curve. See the precise description of the output below.

INPUT:

- E – elliptic curve over $\mathbf{Q}$ of rank 1 or 2
- q – **an odd prime that does not divide the order of the** rational torsion subgroup of E
- **`first_only` – bool (default: `True`) whether two only return** the first prime that one can work modulo to get data about the Euler system

OUTPUT in the rank 1 case or when the default flag `first_only=True`:

- ℓ **– first good odd prime satisfying the Kolyvagin**
  condition that q divides gcd(a_{ell},ell+1)` and the reduction map is surjective to $E(\mathbf{F}_\ell)/qE(\mathbf{F}_\ell)$
- D **– discriminant of the first quadratic imaginary field**
  K that satisfies the Heegner hypothesis for E such that both ℓ is inert in K, and the twist E^D has analytic rank ≤ 1
- h_D – the class number of K
- the dimension of the Brandt module $B(\ell, N)$, where N is the conductor of E

OUTPUT in the rank 2 case:

- ℓ_1 – first prime (as above in the rank 1 case) where reduction map is surjective
- ℓ_2 – second prime (as above) where reduction map is surjective
- D **– discriminant of the first quadratic imaginary field**
  K that satisfies the Heegner hypothesis for E such that both ℓ_1 and ℓ_2 are simultaneously inert in K, and the twist E^D has analytic rank ≤ 1
- h_D – the class number of K
- the dimension of the Brandt module $B(\ell_1, N)$, where N is the conductor of E
- the dimension of the Brandt module $B(\ell_2, N)$

EXAMPLES:

Import this function:

```
sage: from sage.schemes.elliptic_curves.heegner import kolyvagin_reduction_data
```

A rank 1 example:

```
sage: kolyvagin_reduction_data(EllipticCurve('37a1'), 3)
(17, -7, 1, 52)
```

A rank 3 example:

```
sage: kolyvagin_reduction_data(EllipticCurve('5077a1'), 3)
(11, -47, 5, 4234)
sage: H = heegner_points(5077, -47)
sage: [c for c in H.kolyvagin_conductors(2, 10, EllipticCurve('5077a1'), 3)
....:    if c % 11]
[667, 943, 1189, 2461]
sage: factor(667)
23 * 29
```

A rank 4 example (the first Kolyvagin class that we could try to compute would be $P_{23\cdot 29\cdot 41}$, and would require working in a space of dimension 293060 (so prohibitive at present):

```
sage: E = elliptic_curves.rank(4)[0]
sage: kolyvagin_reduction_data(E,3)                # long time
(11, -71, 7, 293060)
sage: H = heegner_points(293060, -71)
sage: H.kolyvagin_conductors(1,4,E,3)
[11, 17, 23, 41]
```

The first rank 2 example:

```
sage: kolyvagin_reduction_data(EllipticCurve('389a'), 3)
(5, -7, 1, 130)
sage: kolyvagin_reduction_data(EllipticCurve('389a'), 3, first_only=False)
(5, 17, -7, 1, 130, 520)
```

A large $q = 7$:

```
sage: kolyvagin_reduction_data(EllipticCurve('1143c1'), 7, first_only=False)
(13, 83, -59, 3, 1536, 10496)
```

Additive reduction:

```
sage: kolyvagin_reduction_data(EllipticCurve('2350g1'), 5, first_only=False)
(19, 239, -311, 19, 6480, 85680)
```

sage.schemes.elliptic_curves.heegner.**make_monic**(f)

Return a monic integral polynomial g and an integer d such that if α is a root of g, then α/d is a root of f. In other words, $cf(x) = g(dx)$ for some scalar c.

INPUT:

- f – polynomial over the rational numbers

OUTPUT: A monic integral polynomial and an integer.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.heegner import make_monic
sage: R.<x> = QQ[]
sage: make_monic(3*x^3 + 14*x^2 - 7*x + 5)
(x^3 + 14*x^2 - 21*x + 45, 3)
```

In this example we verify that `make_monic` does what we claim it does:

```
sage: K.<a> = NumberField(x^3 + 17*x - 3)
sage: f = (a/7+2/3).minpoly(); f
x^3 - 2*x^2 + 247/147*x - 4967/9261
sage: g, d = make_monic(f); (g, d)
(x^3 - 42*x^2 + 741*x - 4967, 21)
sage: K.<b> = NumberField(g)
sage: (b/d).minpoly()
x^3 - 2*x^2 + 247/147*x - 4967/9261
```

sage.schemes.elliptic_curves.heegner.**nearby_rational_poly**(f, ***kwds*)

Return a polynomial whose coefficients are rational numbers close to the coefficients of f.

INPUT:

- f – polynomial with real floating point entries
- `**kwds` – passed on to `nearby_rational` method

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.heegner import nearby_rational_poly
sage: R.<x> = RR[]
sage: nearby_rational_poly(2.1*x^2 + 3.5*x - 1.2, max_error=10e-16)
21/10*X^2 + 7/2*X - 6/5
sage: nearby_rational_poly(2.1*x^2 + 3.5*x - 1.2, max_error=10e-17)
```

(continues on next page)

(continued from previous page)

```
4728779608739021/2251799813685248*X^2 + 7/2*X - 5404319552844595/4503599627370496
sage: RR(4728779608739021/2251799813685248  - 21/10)
8.88178419700125e-17
```

sage.schemes.elliptic_curves.heegner.**quadratic_order**(*D*, *c*, *names='a'*)

Return order of conductor c in quadratic field with fundamental discriminant D.

INPUT:

- D – fundamental discriminant
- c – conductor
- `names` – string (default: 'a')

OUTPUT:

- order R of conductor c in an imaginary quadratic field
- the element $c\sqrt{D}$ as an element of R

The generator for the field is named 'a' by default.

EXAMPLES:

```
sage: sage.schemes.elliptic_curves.heegner.quadratic_order(-7,3)
(Order of conductor 6 generated by 3*a in Number Field in a
 with defining polynomial x^2 + 7 with a = 2.645751311064591?*I,
 3*a)
sage: sage.schemes.elliptic_curves.heegner.quadratic_order(-7,3,'alpha')
(Order of conductor 6 generated by 3*alpha in Number Field in alpha
 with defining polynomial x^2 + 7 with alpha = 2.645751311064591?*I,
 3*alpha)
```

sage.schemes.elliptic_curves.heegner.**satisfies_heegner_hypothesis**(*self*, *D*)

Returns `True` precisely when D is a fundamental discriminant that satisfies the Heegner hypothesis for this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: E.satisfies_heegner_hypothesis(-7)
True
sage: E.satisfies_heegner_hypothesis(-11)
False
```

sage.schemes.elliptic_curves.heegner.**satisfies_weak_heegner_hypothesis**(*N*, *D*)

Check that D satisfies the weak Heegner hypothesis relative to N. This is all that is needed to define Heegner points.

The condition is that $D < 0$ is a fundamental discriminant and that each unramified prime dividing N splits in $K = \mathbf{Q}(\sqrt{D})$ and each ramified prime exactly divides N. We also do not require that $D < -4$.

INPUT:

- N – positive integer
- D – negative integer

EXAMPLES:

```
sage: s = sage.schemes.elliptic_curves.heegner.satisfies_weak_heegner_hypothesis
sage: s(37,-7)
True
sage: s(37,-37)
False
sage: s(37,-37*4)
True
sage: s(100,-4)
False
sage: [D for D in [-1,-2,..,-40] if s(37,D)]
[-3, -4, -7, -11, -40]
sage: [D for D in [-1,-2,..,-100] if s(37,D)]
[-3, -4, -7, -11, -40, -47, -67, -71, -83, -84, -95]
sage: EllipticCurve('37a').heegner_discriminants_list(10)
[-7, -11, -40, -47, -67, -71, -83, -84, -95, -104]
```

sage.schemes.elliptic_curves.heegner.**simplest_rational_poly**(*f*, *prec*)

Return a polynomial whose coefficients are as simple as possible rationals that are also close to the coefficients of f.

INPUT:

- *f* – polynomial with real floating point entries
- `prec` – positive integer

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.heegner import simplest_rational_poly
sage: R.<x> = RR[]
sage: simplest_rational_poly(2.1*x^2 + 3.5*x - 1.2, 53)
21/10*X^2 + 7/2*X - 6/5
```

18.24 p-adic L-functions of elliptic curves

To an elliptic curve E over the rational numbers and a prime p, one can associate a p-adic L-function; at least if E does not have additive reduction at p. This function is defined by interpolation of L-values of E at twists. Through the main conjecture of Iwasawa theory it should also be equal to a characteristic series of a certain Selmer group.

If E is ordinary, then it is an element of the Iwasawa algebra $\Lambda(\mathbf{Z}_p^\times) = \mathbf{Z}_p[\Delta][\![T]\!]$, where Δ is the group of $(p-1)$-st roots of unity in $\mathbf{Z}_p^\times$, and $T = [\gamma] - 1$ where $\gamma = 1 + p$ is a generator of $1 + p\mathbf{Z}_p$. (There is a slightly different description for $p = 2$.)

One can decompose this algebra as the direct product of the subalgebras corresponding to the characters of Δ, which are simply the powers τ^η ($0 \le \eta \le p-2$) of the Teichmueller character $\tau : \Delta \to \mathbf{Z}_p^\times$. Projecting the L-function into these components gives $p-1$ power series in T, each with coefficients in $\mathbf{Z}_p$.

If E is supersingular, the series will have coefficients in a quadratic extension of $\mathbf{Q}_p$, and the coefficients will be unbounded. In this case we have only implemented the series for $\eta = 0$. We have also implemented the p-adic L-series as formulated by Perrin-Riou [BP1993], which has coefficients in the Dieudonné module $D_pE = H^1_{dR}(E/\mathbf{Q}_p)$ of E. There is a different description by Pollack [Pol2003] which is not available here.

According to the p-adic version of the Birch and Swinnerton-Dyer conjecture [MTT1986], the order of vanishing of the L-function at the trivial character (i.e. of the series for $\eta = 0$ at $T = 0$) is just the rank of $E(\mathbf{Q})$, or this rank plus one if the reduction at p is split multiplicative.

See [SW2013] for more details.

AUTHORS:

- William Stein (2007-01-01): first version
- Chris Wuthrich (22/05/2007): changed minor issues and added supersingular things
- Chris Wuthrich (11/2008): added quadratic_twists
- David Loeffler (01/2011): added nontrivial Teichmueller components

class sage.schemes.elliptic_curves.padic_lseries.**pAdicLseries**(*E*, *p*, *implementation='eclib'*, *normalize='L_ratio'*)

Bases: SageObject

The p-adic L-series of an elliptic curve.

EXAMPLES:

An ordinary example:

```
sage: e = EllipticCurve('389a')
sage: L = e.padic_lseries(5)
sage: L.series(0)
Traceback (most recent call last):
...
ValueError: n (=0) must be a positive integer
sage: L.series(1)
O(T^1)
sage: L.series(2)
O(5^4) + O(5)*T + (4 + O(5))*T^2 + (2 + O(5))*T^3 + (3 + O(5))*T^4 + O(T^5)
sage: L.series(3, prec=10)
O(5^5) + O(5^2)*T + (4 + 4*5 + O(5^2))*T^2 + (2 + 4*5 + O(5^2))*T^3 + (3 + O(5^
↪2))*T^4 + (1 + O(5))*T^5 + O(5)*T^6 + (4 + O(5))*T^7 + (2 + O(5))*T^8 + O(5)*T^
↪9 + O(T^10)
sage: L.series(2,quadratic_twist=-3)
2 + 4*5 + 4*5^2 + O(5^4) + O(5)*T + (1 + O(5))*T^2 + (4 + O(5))*T^3 + O(5)*T^4 +␣
↪O(T^5)
```

A prime p such that E[p] is reducible:

```
sage: L = EllipticCurve('11a').padic_lseries(5)
sage: L.series(1)
5 + O(5^2) + O(T)
sage: L.series(2)
5 + 4*5^2 + O(5^3) + O(5^0)*T + O(5^0)*T^2 + O(5^0)*T^3 + O(5^0)*T^4 + O(T^5)
sage: L.series(3)
5 + 4*5^2 + 4*5^3 + O(5^4) + O(5)*T + O(5)*T^2 + O(5)*T^3 + O(5)*T^4 + O(T^5)
```

An example showing the calculation of nontrivial Teichmueller twists:

```
sage: E = EllipticCurve('11a1')
sage: lp = E.padic_lseries(7)
sage: lp.series(4,eta=1)
3 + 7^3 + 6*7^4 + 3*7^5 + O(7^6) + (2*7 + 7^2 + O(7^3))*T + (1 + 5*7^2 + O(7^
↪3))*T^2 + (4 + 4*7 + 4*7^2 + O(7^3))*T^3 + (4 + 3*7 + 7^2 + O(7^3))*T^4 + O(T^5)
sage: lp.series(4,eta=2)
5 + 6*7 + 4*7^2 + 2*7^3 + 3*7^4 + 2*7^5 + O(7^6) + (6 + 4*7 + 7^2 + O(7^3))*T +␣
↪(3 + 2*7^2 + O(7^3))*T^2 + (1 + 4*7 + 7^2 + O(7^3))*T^3 + (6 + 6*7 + 6*7^2 +␣
↪O(7^3))*T^4 + O(T^5)
```

(continues on next page)

(continued from previous page)

```
sage: lp.series(4,eta=3)
O(7^6) + (5 + 4*7 + 2*7^2 + O(7^3))*T + (6 + 5*7 + 2*7^2 + O(7^3))*T^2 + (5*7 +␣
↪O(7^3))*T^3 + (7 + 4*7^2 + O(7^3))*T^4 + O(T^5)
```

(Note that the last series vanishes at $T = 0$, which is consistent with

```
sage: E.quadratic_twist(-7).rank()
1
```

This proves that E has rank 1 over $\mathbf{Q}(\zeta_7)$.)

alpha (*prec=20*)

Return a p-adic root α of the polynomial $x^2 - a_p x + p$ with $ord_p(\alpha) < 1$. In the ordinary case this is just the unit root.

INPUT:

- `prec` – positive integer, the p-adic precision of the root.

EXAMPLES:

Consider the elliptic curve 37a:

```
sage: E = EllipticCurve('37a')
```

An ordinary prime:

```
sage: L = E.padic_lseries(5)
sage: alpha = L.alpha(10); alpha
3 + 2*5 + 4*5^2 + 2*5^3 + 5^4 + 4*5^5 + 2*5^7 + 5^8 + 5^9 + O(5^10)
sage: alpha^2 - E.ap(5)*alpha + 5
O(5^10)
```

A supersingular prime:

```
sage: L = E.padic_lseries(3)
sage: alpha = L.alpha(10); alpha
alpha + O(alpha^21)
sage: alpha^2 - E.ap(3)*alpha + 3
O(alpha^22)
```

A reducible prime:

```
sage: L = EllipticCurve('11a').padic_lseries(5)
sage: L.alpha(5)
1 + 4*5 + 3*5^2 + 2*5^3 + 4*5^4 + O(5^5)
```

elliptic_curve()

Return the elliptic curve to which this p-adic L-series is associated.

EXAMPLES:

```
sage: L = EllipticCurve('11a').padic_lseries(5)
sage: L.elliptic_curve()
Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational Field
```

measure(*a*, *n*, *prec*, *quadratic_twist=1*, *sign=1*)

Return the measure on $\mathbf{Z}_p^\times$ defined by

$$\mu_{E,\alpha}^{+}(a+p^n\mathbf{Z}_p) = \frac{1}{\alpha^n}\left[\frac{a}{p^n}\right]^{+} - \frac{1}{\alpha^{n+1}}\left[\frac{a}{p^{n-1}}\right]^{+}$$

where $[\cdot]^+$ is the modular symbol. This is used to define this p-adic L-function (at least when the reduction is good).

The optional argument `sign` allows the minus symbol $[\cdot]^-$ to be substituted for the plus symbol.

The optional argument `quadratic_twist` replaces E by the twist in the above formula, but the twisted modular symbol is computed using a sum over modular symbols of E rather than finding the modular symbols for the twist. Quadratic twists are only implemented if the sign is $+1$.

Note that the normalization is not correct at this stage: use `_quotient_of periods` and `_quotient_of periods_to_twist` to correct.

Note also that this function does not check if the condition on the `quadratic_twist=D` is satisfied. So the result will only be correct if for each prime ℓ dividing D, we have $ord_\ell(N) <= ord_\ell(D)$, where N is the conductor of the curve.

INPUT:

- `a` – an integer
- `n` – a non-negative integer
- `prec` – an integer
- `quadratic_twist` (default = 1) – a fundamental discriminant of a quadratic field, should be coprime to the conductor of E
- `sign` (default = 1) – an integer, which should be ± 1.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: L = E.padic_lseries(5)
sage: L.measure(1,2, prec=9)
2 + 3*5 + 4*5^3 + 2*5^4 + 3*5^5 + 3*5^6 + 4*5^7 + 4*5^8 + O(5^9)
sage: L.measure(1,2, quadratic_twist=8,prec=15)
O(5^15)
sage: L.measure(1,2, quadratic_twist=-4,prec=15)
4 + 4*5 + 4*5^2 + 3*5^3 + 2*5^4 + 5^5 + 3*5^6 + 5^8 + 2*5^9 + 3*5^12 + 2*5^13␣
↪+ 4*5^14 + O(5^15)

sage: E = EllipticCurve('11a1')
sage: a = E.quadratic_twist(-3).padic_lseries(5).measure(1,2,prec=15)
sage: b = E.padic_lseries(5).measure(1,2, quadratic_twist=-3,prec=15)
sage: a == b * E.padic_lseries(5)._quotient_of_periods_to_twist(-3)
True
```

modular_symbol (*r*, *sign=1*, *quadratic_twist=1*)

Return the modular symbol evaluated at r.

This is used to compute this p-adic L-series.

Note that the normalization is not correct at this stage: use `_quotient_of periods_to_twist` to correct.

Note also that this function does not check if the condition on the quadratic_twist=D is satisfied. So the result will only be correct if for each prime ℓ dividing D, we have $ord_\ell(N) <= ord_\ell(D)$, where N is the conductor of the curve.

INPUT:

- `r` – a cusp given as either a rational number or oo
- `sign` – +1 (default) or -1 (only implemented without twists)
- `quadratic_twist` – a fundamental discriminant of a quadratic field or +1 (default)

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: lp = E.padic_lseries(5)
sage: [lp.modular_symbol(r) for r in [0,1/5,oo,1/11]]
[1/5, 6/5, 0, 0]
sage: [lp.modular_symbol(r,sign=-1) for r in [0,1/3,oo,1/7]]
[0, 1/2, 0, -1/2]
sage: [lp.modular_symbol(r,quadratic_twist=-20) for r in [0,1/5,oo,1/11]]
[1, 1, 0, 1/2]

sage: E = EllipticCurve('20a1')
sage: Et = E.quadratic_twist(-4)
sage: lpt = Et.padic_lseries(5)
sage: eta = lpt._quotient_of_periods_to_twist(-4)
sage: lpt.modular_symbol(0) == lp.modular_symbol(0,quadratic_twist=-4) / eta
True
```

order_of_vanishing()

Return the order of vanishing of this p-adic L-series.

The output of this function is provably correct, due to a theorem of Kato [Kat2004].

Note: currently p must be a prime of good ordinary reduction.

REFERENCES:

- [MTT1986]
- [Kat2004]

EXAMPLES:

```
sage: L = EllipticCurve('11a').padic_lseries(3)
sage: L.order_of_vanishing()
0
sage: L = EllipticCurve('11a').padic_lseries(5)
sage: L.order_of_vanishing()
0
sage: L = EllipticCurve('37a').padic_lseries(5)
sage: L.order_of_vanishing()
1
sage: L = EllipticCurve('43a').padic_lseries(3)
sage: L.order_of_vanishing()
1
sage: L = EllipticCurve('37b').padic_lseries(3)
sage: L.order_of_vanishing()
0
sage: L = EllipticCurve('389a').padic_lseries(3)
sage: L.order_of_vanishing()
2
sage: L = EllipticCurve('389a').padic_lseries(5)
sage: L.order_of_vanishing()
```

(continues on next page)

(continued from previous page)

```
2
sage: L = EllipticCurve('5077a').padic_lseries(5, implementation = 'eclib')
sage: L.order_of_vanishing()
3
```

prime()

Return the prime p as in 'p-adic L-function'.

EXAMPLES:

```
sage: L = EllipticCurve('11a').padic_lseries(5)
sage: L.prime()
5
```

teichmuller(*prec*)

Return Teichmuller lifts to the given precision.

INPUT:

- `prec` – a positive integer.

OUTPUT:

- a list of p-adic numbers, the cached Teichmuller lifts

EXAMPLES:

```
sage: L = EllipticCurve('11a').padic_lseries(7)
sage: L.teichmuller(1)
[0, 1, 2, 3, 4, 5, 6]
sage: L.teichmuller(2)
[0, 1, 30, 31, 18, 19, 48]
```

class sage.schemes.elliptic_curves.padic_lseries.**pAdicLseriesOrdinary**(*E*, *p*, *implementation='eclib'*, *normalize='L_ratio'*)

Bases: *pAdicLseries*

is_ordinary()

Return `True` if the elliptic curve that this L-function is attached to is ordinary.

EXAMPLES:

```
sage: L = EllipticCurve('11a').padic_lseries(5)
sage: L.is_ordinary()
True
```

is_supersingular()

Return `True` if the elliptic curve that this L function is attached to is supersingular.

EXAMPLES:

```
sage: L = EllipticCurve('11a').padic_lseries(5)
sage: L.is_supersingular()
False
```

power_series (*n=2*, *quadratic_twist=1*, *prec=5*, *eta=0*)

Return the n-th approximation to the p-adic L-series, in the component corresponding to the η-th power of the Teichmueller character, as a power series in T (corresponding to $\gamma - 1$ with $\gamma = 1 + p$ as a generator of $1 + p\mathbf{Z}_p$). Each coefficient is a p-adic number whose precision is provably correct.

Here the normalization of the p-adic L-series is chosen such that $L_p(E,1) = (1 - 1/\alpha)^2 L(E,1)/\Omega_E$ where α is the unit root of the characteristic polynomial of Frobenius on $T_p E$ and Ω_E is the Néron period of E.

INPUT:

- `n` – (default: 2) a positive integer
- `quadratic_twist` – (default: +1) a fundamental discriminant of a quadratic field, coprime to the conductor of the curve
- `prec` – (default: 5) maximal number of terms of the series to compute; to compute as many as possible just give a very large number for `prec`; the result will still be correct.
- `eta` (default: 0) an integer (specifying the power of the Teichmueller character on the group of roots of unity in $\mathbf{Z}_p^\times$)

power_series() is identical to `series`.

EXAMPLES:

We compute some p-adic L-functions associated to the elliptic curve 11a:

```
sage: E = EllipticCurve('11a')
sage: p = 3
sage: E.is_ordinary(p)
True
sage: L = E.padic_lseries(p)
sage: L.series(3)
2 + 3 + 3^2 + 2*3^3 + O(3^5) + (1 + 3 + O(3^2))*T + (1 + 2*3 + O(3^2))*T^2 +␣
↪O(3)*T^3 + O(3)*T^4 + O(T^5)
```

Another example at a prime of bad reduction, where the p-adic L-function has an extra 0 (compared to the non p-adic L-function):

```
sage: E = EllipticCurve('11a')
sage: p = 11
sage: E.is_ordinary(p)
True
sage: L = E.padic_lseries(p)
sage: L.series(2)
O(11^4) + (10 + O(11))*T + (6 + O(11))*T^2 + (2 + O(11))*T^3 + (5 + O(11))*T^
↪4 + O(T^5)
```

We compute a p-adic L-function that vanishes to order 2:

```
sage: E = EllipticCurve('389a')
sage: p = 3
sage: E.is_ordinary(p)
True
sage: L = E.padic_lseries(p)
sage: L.series(1)
O(T^1)
sage: L.series(2)
O(3^4) + O(3)*T + (2 + O(3))*T^2 + O(T^3)
sage: L.series(3)
```

(continues on next page)

(continued from previous page)

```
O(3^5) + O(3^2)*T + (2 + 2*3 + O(3^2))*T^2 + (2 + O(3))*T^3 + (1 + O(3))*T^4␣
↪+ O(T^5)
```

Checks if the precision can be changed (Issue #5846):

```
sage: L.series(3,prec=4)
O(3^5) + O(3^2)*T + (2 + 2*3 + O(3^2))*T^2 + (2 + O(3))*T^3 + O(T^4)
sage: L.series(3,prec=6)
O(3^5) + O(3^2)*T + (2 + 2*3 + O(3^2))*T^2 + (2 + O(3))*T^3 + (1 + O(3))*T^4␣
↪+ (1 + O(3))*T^5 + O(T^6)
```

Rather than computing the p-adic L-function for the curve '15523a1', one can compute it as a quadratic_twist:

```
sage: E = EllipticCurve('43a1')
sage: lp = E.padic_lseries(3)
sage: lp.series(2,quadratic_twist=-19)
2 + 2*3 + 2*3^2 + O(3^4) + (1 + O(3))*T + (1 + O(3))*T^2 + O(T^3)
sage: E.quadratic_twist(-19).label()     # optional -- database_cremona_
↪ellcurve
'15523a1'
```

This proves that the rank of '15523a1' is zero, even if `mwrank` cannot determine this.

We calculate the L-series in the nontrivial Teichmueller components:

```
sage: L = EllipticCurve('110a1').padic_lseries(5, implementation="sage")
sage: for j in [0..3]: print(L.series(4, eta=j))
O(5^6) + (2 + 2*5 + 2*5^2 + O(5^3))*T + (5 + 5^2 + O(5^3))*T^2 + (4 + 4*5 +␣
↪2*5^2 + O(5^3))*T^3 + (1 + 5 + 3*5^2 + O(5^3))*T^4 + O(T^5)
4 + 3*5 + 2*5^2 + 3*5^3 + 5^4 + O(5^6) + (1 + 3*5 + 4*5^2 + O(5^3))*T + (3 +␣
↪4*5 + 3*5^2 + O(5^3))*T^2 + (3 + 3*5^2 + O(5^3))*T^3 + (1 + 2*5 + 2*5^2 +␣
↪O(5^3))*T^4 + O(T^5)
2 + O(5^6) + (1 + 5 + O(5^3))*T + (2 + 4*5 + 3*5^2 + O(5^3))*T^2 + (4 + 5 +␣
↪2*5^2 + O(5^3))*T^3 + (4 + O(5^3))*T^4 + O(T^5)
3 + 5 + 2*5^2 + 5^3 + 3*5^4 + 4*5^5 + O(5^6) + (1 + 2*5 + 4*5^2 + O(5^3))*T +␣
↪(1 + 4*5 + O(5^3))*T^2 + (3 + 2*5 + 2*5^2 + O(5^3))*T^3 + (5 + 5^2 + O(5^
↪3))*T^4 + O(T^5)
```

It should now also work with $p = 2$ (Issue #20798):

```
sage: E = EllipticCurve("53a1")
sage: lp = E.padic_lseries(2)
sage: lp.series(7)
O(2^8) + (1 + 2^2 + 2^3 + O(2^5))*T + (1 + 2^3 + O(2^4))*T^2 + (2^2 + 2^3 +␣
↪O(2^4))*T^3 + (2 + 2^2 + O(2^3))*T^4 + O(T^5)

sage: E = EllipticCurve("109a1")
sage: lp = E.padic_lseries(2)
sage: lp.series(6)
2^2 + 2^6 + O(2^7) + (2 + O(2^4))*T + O(2^3)*T^2 + (2^2 + O(2^3))*T^3 + (2 +␣
↪O(2^2))*T^4 + O(T^5)
```

Check that twists by odd Teichmuller characters are ok (Issue #32258):

```
sage: E = EllipticCurve("443c1")
sage: lp = E.padic_lseries(17, implementation="num")
sage: l8 = lp.series(2,eta=8,prec=3)
```

(continues on next page)

(continued from previous page)

```
sage: l8.list()[0] - 1/lp.alpha()
O(17^4)
sage: lp = E.padic_lseries(2, implementation="num")
sage: l1 = lp.series(8,eta=1,prec=3)
sage: l1.list()[0] - 4/lp.alpha()^2
O(2^9)
```

series (*n=2*, *quadratic_twist=1*, *prec=5*, *eta=0*)

Return the n-th approximation to the p-adic L-series, in the component corresponding to the η-th power of the Teichmueller character, as a power series in T (corresponding to $\gamma - 1$ with $\gamma = 1 + p$ as a generator of $1 + p\mathbf{Z}_p$). Each coefficient is a p-adic number whose precision is provably correct.

Here the normalization of the p-adic L-series is chosen such that $L_p(E, 1) = (1 - 1/\alpha)^2 L(E, 1)/\Omega_E$ where α is the unit root of the characteristic polynomial of Frobenius on $T_p E$ and Ω_E is the Néron period of E.

INPUT:

- `n` – (default: 2) a positive integer
- `quadratic_twist` – (default: +1) a fundamental discriminant of a quadratic field, coprime to the conductor of the curve
- `prec` – (default: 5) maximal number of terms of the series to compute; to compute as many as possible just give a very large number for `prec`; the result will still be correct.
- `eta` (default: 0) an integer (specifying the power of the Teichmueller character on the group of roots of unity in $\mathbf{Z}_p^\times$)

power_series() is identical to `series`.

EXAMPLES:

We compute some p-adic L-functions associated to the elliptic curve 11a:

```
sage: E = EllipticCurve('11a')
sage: p = 3
sage: E.is_ordinary(p)
True
sage: L = E.padic_lseries(p)
sage: L.series(3)
2 + 3 + 3^2 + 2*3^3 + O(3^5) + (1 + 3 + O(3^2))*T + (1 + 2*3 + O(3^2))*T^2 +␣
↪O(3)*T^3 + O(3)*T^4 + O(T^5)
```

Another example at a prime of bad reduction, where the p-adic L-function has an extra 0 (compared to the non p-adic L-function):

```
sage: E = EllipticCurve('11a')
sage: p = 11
sage: E.is_ordinary(p)
True
sage: L = E.padic_lseries(p)
sage: L.series(2)
O(11^4) + (10 + O(11))*T + (6 + O(11))*T^2 + (2 + O(11))*T^3 + (5 + O(11))*T^
↪4 + O(T^5)
```

We compute a p-adic L-function that vanishes to order 2:

```
sage: E = EllipticCurve('389a')
sage: p = 3
```

(continues on next page)

(continued from previous page)

```
sage: E.is_ordinary(p)
True
sage: L = E.padic_lseries(p)
sage: L.series(1)
O(T^1)
sage: L.series(2)
O(3^4) + O(3)*T + (2 + O(3))*T^2 + O(T^3)
sage: L.series(3)
O(3^5) + O(3^2)*T + (2 + 2*3 + O(3^2))*T^2 + (2 + O(3))*T^3 + (1 + O(3))*T^4␣
↪+ O(T^5)
```

Checks if the precision can be changed (Issue #5846):

```
sage: L.series(3,prec=4)
O(3^5) + O(3^2)*T + (2 + 2*3 + O(3^2))*T^2 + (2 + O(3))*T^3 + O(T^4)
sage: L.series(3,prec=6)
O(3^5) + O(3^2)*T + (2 + 2*3 + O(3^2))*T^2 + (2 + O(3))*T^3 + (1 + O(3))*T^4␣
↪+ (1 + O(3))*T^5 + O(T^6)
```

Rather than computing the p-adic L-function for the curve '15523a1', one can compute it as a quadratic_twist:

```
sage: E = EllipticCurve('43a1')
sage: lp = E.padic_lseries(3)
sage: lp.series(2,quadratic_twist=-19)
2 + 2*3 + 2*3^2 + O(3^4) + (1 + O(3))*T + (1 + O(3))*T^2 + O(T^3)
sage: E.quadratic_twist(-19).label()     # optional -- database_cremona_
↪ellcurve
'15523a1'
```

This proves that the rank of '15523a1' is zero, even if `mwrank` cannot determine this.

We calculate the L-series in the nontrivial Teichmueller components:

```
sage: L = EllipticCurve('110a1').padic_lseries(5, implementation="sage")
sage: for j in [0..3]: print(L.series(4, eta=j))
O(5^6) + (2 + 2*5 + 2*5^2 + O(5^3))*T + (5 + 5^2 + O(5^3))*T^2 + (4 + 4*5 +␣
↪2*5^2 + O(5^3))*T^3 + (1 + 5 + 3*5^2 + O(5^3))*T^4 + O(T^5)
4 + 3*5 + 2*5^2 + 3*5^3 + 5^4 + O(5^6) + (1 + 3*5 + 4*5^2 + O(5^3))*T + (3 +␣
↪4*5 + 3*5^2 + O(5^3))*T^2 + (3 + 3*5^2 + O(5^3))*T^3 + (1 + 2*5 + 2*5^2 +␣
↪O(5^3))*T^4 + O(T^5)
2 + O(5^6) + (1 + 5 + O(5^3))*T + (2 + 4*5 + 3*5^2 + O(5^3))*T^2 + (4 + 5 +␣
↪2*5^2 + O(5^3))*T^3 + (4 + O(5^3))*T^4 + O(T^5)
3 + 5 + 2*5^2 + 5^3 + 3*5^4 + 4*5^5 + O(5^6) + (1 + 2*5 + 4*5^2 + O(5^3))*T +␣
↪(1 + 4*5 + O(5^3))*T^2 + (3 + 2*5 + 2*5^2 + O(5^3))*T^3 + (5 + 5^2 + O(5^
↪3))*T^4 + O(T^5)
```

It should now also work with $p = 2$ (Issue #20798):

```
sage: E = EllipticCurve("53a1")
sage: lp = E.padic_lseries(2)
sage: lp.series(7)
O(2^8) + (1 + 2^2 + 2^3 + O(2^5))*T + (1 + 2^3 + O(2^4))*T^2 + (2^2 + 2^3 +␣
↪O(2^4))*T^3 + (2 + 2^2 + O(2^3))*T^4 + O(T^5)

sage: E = EllipticCurve("109a1")
sage: lp = E.padic_lseries(2)
sage: lp.series(6)
```

(continues on next page)

(continued from previous page)

```
2^2 + 2^6 + O(2^7) + (2 + O(2^4))*T + O(2^3)*T^2 + (2^2 + O(2^3))*T^3 + (2 +␣
↪O(2^2))*T^4 + O(T^5)
```

Check that twists by odd Teichmuller characters are ok (Issue #32258):

```
sage: E = EllipticCurve("443c1")
sage: lp = E.padic_lseries(17, implementation="num")
sage: l8 = lp.series(2,eta=8,prec=3)
sage: l8.list()[0] - 1/lp.alpha()
O(17^4)
sage: lp = E.padic_lseries(2, implementation="num")
sage: l1 = lp.series(8,eta=1,prec=3)
sage: l1.list()[0] - 4/lp.alpha()^2
O(2^9)
```

class sage.schemes.elliptic_curves.padic_lseries.**pAdicLseriesSupersingular**(*E*, *p*, *implementation='eclib'*, *normalize='L_ratio'*)

Bases: *pAdicLseries*

Dp_valued_height(*prec=20*)

Return the canonical p-adic height with values in the Dieudonné module $D_p(E)$.

It is defined to be

$$h_\eta \cdot \omega - h_\omega \cdot \eta$$

where h_η is made out of the sigma function of Bernardi and h_ω is log_E^2.

The answer `v` is given as `v[1]*omega + v[2]*eta`. The coordinates of `v` are dependent of the Weierstrass equation.

EXAMPLES:

```
sage: E = EllipticCurve('53a')
sage: L = E.padic_lseries(5)
sage: h = L.Dp_valued_height(7)
sage: h(E.gens()[0])
(3*5 + 5^2 + 2*5^3 + 3*5^4 + 4*5^5 + 5^6 + 5^7 + O(5^8), 5^2 + 4*5^4 + 2*5^7␣
↪+ 3*5^8 + O(5^9))
```

Dp_valued_regulator(*prec=20*, *v1=0*, *v2=0*)

Return the canonical p-adic regulator with values in the Dieudonné module $D_p(E)$ as defined by Perrin-Riou using the p-adic height with values in $D_p(E)$.

The result is written in the basis ω, $\varphi(\omega)$, and hence the coordinates of the result are independent of the chosen Weierstrass equation.

Note: The definition here is corrected with respect to Perrin-Riou's article [PR2003]. See [SW2013].

EXAMPLES:

```
sage: E = EllipticCurve('43a')
sage: L = E.padic_lseries(7)
sage: L.Dp_valued_regulator(7)
(5*7 + 6*7^2 + 4*7^3 + 4*7^4 + 7^5 + 4*7^7 + O(7^8), 4*7^2 + 2*7^3 + 3*7^4 +␣
↪7^5 + 6*7^6 + 4*7^7 + O(7^8))
```

Dp_valued_series (*n=3*, *quadratic_twist=1*, *prec=5*)

Return a vector of two components which are p-adic power series.

The answer v is such that

$$(1-\varphi)^{-2}\cdot L_p(E,T) = \mathtt{v[1]}\cdot\omega + \mathtt{v[2]}\cdot\varphi(\omega)$$

as an element of the Dieudonné module $D_p(E) = H^1_{dR}(E/\mathbf{Q}_p)$ where ω is the invariant differential and φ is the Frobenius on $D_p(E)$.

According to the p-adic Birch and Swinnerton-Dyer conjecture [BP1993] this function has a zero of order rank of $E(\mathbf{Q})$ and it's leading term is contains the order of the Tate-Shafarevich group, the Tamagawa numbers, the order of the torsion subgroup and the D_p-valued p-adic regulator.

INPUT:

- `n` – (default: 3) a positive integer
- `prec` – (default: 5) a positive integer

EXAMPLES:

```
sage: E = EllipticCurve('14a')
sage: L = E.padic_lseries(5)
sage: L.Dp_valued_series(4)  # long time (9s on sage.math, 2011)
(1 + 4*5 + O(5^2) + (4 + O(5))*T + (1 + O(5))*T^2 + (4 + O(5))*T^3 + (2 +␣
↪O(5))*T^4 + O(T^5), 5^2 + O(5^3) + O(5^2)*T + (4*5 + O(5^2))*T^2 + (2*5 +␣
↪O(5^2))*T^3 + (2 + 2*5 + O(5^2))*T^4 + O(T^5))
```

bernardi_sigma_function (*prec=20*)

Return the p-adic sigma function of Bernardi in terms of $z = log(t)$.

This is the same as `padic_sigma` with `E2 = 0`.

EXAMPLES:

```
sage: E = EllipticCurve('14a')
sage: L = E.padic_lseries(5)
sage: L.bernardi_sigma_function(prec=5) # Todo: some sort of consistency␣
↪check!?
z + 1/24*z^3 + 29/384*z^5 - 8399/322560*z^7 - 291743/92897280*z^9 + O(z^10)
```

frobenius (*prec=20*, *algorithm='mw'*)

Return a geometric Frobenius φ on the Dieudonné module $D_p(E)$ with respect to the basis ω, the invariant differential, and $\eta = x\omega$.

It satisfies $\varphi^2 - a_p/p\,\varphi + 1/p = 0$.

INPUT:

- `prec` – (default: 20) a positive integer
- `algorithm` – either 'mw' (default) for Monsky-Washnitzer or 'approx' for the algorithm described by Bernardi and Perrin-Riou (much slower and not fully tested)

EXAMPLES:

```
sage: E = EllipticCurve('14a')
sage: L = E.padic_lseries(5)
sage: phi = L.frobenius(5)
sage: phi
[                  2 + 5^2 + 5^4 + O(5^5)    3*5^-1 + 3 + 5 + 4*5^2 + 5^3 +␣
↪O(5^4)]
[      3 + 3*5^2 + 4*5^3 + 3*5^4 + O(5^5) 3 + 4*5 + 3*5^2 + 4*5^3 + 3*5^4 +␣
↪O(5^5)]
sage: -phi^2
[5^-1 + O(5^4)         O(5^4)]
[       O(5^5) 5^-1 + O(5^4)]
```

is_ordinary()

Return `True` if the elliptic curve that this L-function is attached to is ordinary.

EXAMPLES:

```
sage: L = EllipticCurve('11a').padic_lseries(19)
sage: L.is_ordinary()
False
```

is_supersingular()

Return `True` if the elliptic curve that this L function is attached to is supersingular.

EXAMPLES:

```
sage: L = EllipticCurve('11a').padic_lseries(19)
sage: L.is_supersingular()
True
```

power_series(*n=3*, *quadratic_twist=1*, *prec=5*, *eta=0*)

Return the n-th approximation to the p-adic L-series as a power series in T (corresponding to $\gamma - 1$ with $\gamma = 1 + p$ as a generator of $1 + p\mathbf{Z}_p$). Each coefficient is an element of a quadratic extension of the p-adic number whose precision is provably correct.

Here the normalization of the p-adic L-series is chosen such that $L_p(E, 1) = (1 - 1/\alpha)^2 L(E, 1)/\Omega_E$ where α is a root of the characteristic polynomial of Frobenius on T_pE and Ω_E is the Néron period of E.

INPUT:

- `n` – (default: 2) a positive integer
- `quadratic_twist` – (default: +1) a fundamental discriminant of a quadratic field, coprime to the conductor of the curve
- `prec` – (default: 5) maximal number of terms of the series to compute; to compute as many as possible just give a very large number for `prec`; the result will still be correct.
- `eta` (default: 0) an integer (specifying the power of the Teichmueller character on the group of roots of unity in $\mathbf{Z}_p^\times$)

OUTPUT:

a power series with coefficients in a quadratic ramified extension of the p-adic numbers generated by a root $alpha$ of the characteristic polynomial of Frobenius on T_pE.

ALIAS: power_series is identical to series.

EXAMPLES:

A supersingular example, where we must compute to higher precision to see anything:

```
sage: e = EllipticCurve('37a')
sage: L = e.padic_lseries(3); L
3-adic L-series of Elliptic Curve defined by y^2 + y = x^3 - x over Rational␣
↪Field
sage: L.series(2)
O(T^3)
sage: L.series(4)          # takes a long time (several seconds)
O(alpha) + (alpha^-2 + O(alpha^0))*T + (alpha^-2 + O(alpha^0))*T^2 + O(T^5)
sage: L.alpha(2).parent()
3-adic Eisenstein Extension Field in alpha defined by x^2 + 3*x + 3
```

An example where we only compute the leading term (Issue #15737):

```
sage: E = EllipticCurve("17a1")
sage: L = E.padic_lseries(3)
sage: L.series(4,prec=1)
alpha^-2 + alpha^-1 + 2 + 2*alpha + ... + O(alpha^38) + O(T)
```

It works also for $p = 2$:

```
sage: E = EllipticCurve("11a1")
sage: lp = E.padic_lseries(2)
sage: lp.series(10)
O(alpha^-3) + (alpha^-4 + O(alpha^-3))*T + (alpha^-4 + O(alpha^-3))*T^2 +␣
↪(alpha^-5 + alpha^-4 + O(alpha^-3))*T^3 + (alpha^-4 + O(alpha^-3))*T^4 +␣
↪O(T^5)
```

series (*n=3*, *quadratic_twist=1*, *prec=5*, *eta=0*)

Return the n-th approximation to the p-adic L-series as a power series in T (corresponding to $\gamma - 1$ with $\gamma = 1 + p$ as a generator of $1 + p\mathbf{Z}_p$). Each coefficient is an element of a quadratic extension of the p-adic number whose precision is provably correct.

Here the normalization of the p-adic L-series is chosen such that $L_p(E, 1) = (1 - 1/\alpha)^2 L(E, 1)/\Omega_E$ where α is a root of the characteristic polynomial of Frobenius on T_pE and Ω_E is the Néron period of E.

INPUT:

- `n` – (default: 2) a positive integer
- `quadratic_twist` – (default: +1) a fundamental discriminant of a quadratic field, coprime to the conductor of the curve
- `prec` – (default: 5) maximal number of terms of the series to compute; to compute as many as possible just give a very large number for `prec`; the result will still be correct.
- `eta` (default: 0) an integer (specifying the power of the Teichmueller character on the group of roots of unity in $\mathbf{Z}_p^\times$)

OUTPUT:

a power series with coefficients in a quadratic ramified extension of the p-adic numbers generated by a root $alpha$ of the characteristic polynomial of Frobenius on T_pE.

ALIAS: power_series is identical to series.

EXAMPLES:

A supersingular example, where we must compute to higher precision to see anything:

```
sage: e = EllipticCurve('37a')
sage: L = e.padic_lseries(3); L
3-adic L-series of Elliptic Curve defined by y^2 + y = x^3 - x over Rational␣
↪Field
sage: L.series(2)
O(T^3)
sage: L.series(4)          # takes a long time (several seconds)
O(alpha) + (alpha^-2 + O(alpha^0))*T + (alpha^-2 + O(alpha^0))*T^2 + O(T^5)
sage: L.alpha(2).parent()
3-adic Eisenstein Extension Field in alpha defined by x^2 + 3*x + 3
```

An example where we only compute the leading term (Issue #15737):

```
sage: E = EllipticCurve("17a1")
sage: L = E.padic_lseries(3)
sage: L.series(4,prec=1)
alpha^-2 + alpha^-1 + 2 + 2*alpha + ... + O(alpha^38) + O(T)
```

It works also for $p = 2$:

```
sage: E = EllipticCurve("11a1")
sage: lp = E.padic_lseries(2)
sage: lp.series(10)
O(alpha^-3) + (alpha^-4 + O(alpha^-3))*T + (alpha^-4 + O(alpha^-3))*T^2 +␣
↪(alpha^-5 + alpha^-4 + O(alpha^-3))*T^3 + (alpha^-4 + O(alpha^-3))*T^4 +␣
↪O(T^5)
```

CHAPTER

NINETEEN

TO BE SORTED

19.1 Descent on elliptic curves over Q with a 2-isogeny

sage.schemes.elliptic_curves.descent_two_isogeny.**test_els**(*a*, *b*, *c*, *d*, *e*)

Doctest function for cdef int everywhere_locally_soluble(mpz_t, mpz_t, mpz_t, mpz_t, mpz_t).

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.descent_two_isogeny import test_els
sage: for _ in range(1000):
....:     a,b,c,d,e = randint(1,1000), randint(1,1000), randint(1,1000),␣
↪randint(1,1000), randint(1,1000)
....:     if pari.Pol([a,b,c,d,e]).hyperellratpoints(1000, 1):
....:         try:
....:             if not test_els(a,b,c,d,e):
....:                 print("This never happened", a, b, c, d, e)
....:         except ValueError:
....:             continue
```

sage.schemes.elliptic_curves.descent_two_isogeny.**test_padic_square**(*a*, *p*)

Doctest function for cdef int padic_square(mpz_t, unsigned long).

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.descent_two_isogeny import test_padic_
↪square as ps
sage: for i in [1..300]:
....:     for p in prime_range(100):
....:         if Qp(p)(i).is_square() != bool(ps(i,p)):
....:             print(i, p)
```

sage.schemes.elliptic_curves.descent_two_isogeny.**test_qpls**(*a*, *b*, *c*, *d*, *e*, *p*)

Testing function for Qp_soluble.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.descent_two_isogeny import test_qpls as tq
sage: tq(1,2,3,4,5,7)
1
```

sage.schemes.elliptic_curves.descent_two_isogeny.**test_valuation**(*a*, *p*)

Doctest function for cdef long valuation(mpz_t, mpz_t).

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.descent_two_isogeny import test_valuation␣
↪as tv
sage: for i in [1..20]:
....:     print('{:>10} {} {} {}'.format(str(factor(i)), tv(i,2), tv(i,3), tv(i,
↪5)))
         1 0 0 0
         2 1 0 0
         3 0 1 0
       2^2 2 0 0
         5 0 0 1
     2 * 3 1 1 0
         7 0 0 0
       2^3 3 0 0
       3^2 0 2 0
     2 * 5 1 0 1
        11 0 0 0
   2^2 * 3 2 1 0
        13 0 0 0
     2 * 7 1 0 0
     3 * 5 0 1 1
       2^4 4 0 0
        17 0 0 0
   2 * 3^2 1 2 0
        19 0 0 0
   2^2 * 5 2 0 1
```

sage.schemes.elliptic_curves.descent_two_isogeny.**two_descent_by_two_isogeny**(*E*, *global_limit_small=10*, *global_limit_large=10*, *verbosity=0*, *selmer_only=0*, *proof=1*)

Given an elliptic curve E with a two-isogeny phi : E –> E' and dual isogeny phi', runs a two-isogeny descent on E, returning n1, n2, n1' and n2'. Here n1 is the number of quartic covers found with a rational point, and n2 is the number which are ELS.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.descent_two_isogeny import two_descent_by_
↪two_isogeny
sage: E = EllipticCurve('14a')
sage: n1, n2, n1_prime, n2_prime = two_descent_by_two_isogeny(E)
sage: log(n1,2) + log(n1_prime,2) - 2 # the rank
0
sage: E = EllipticCurve('65a')
sage: n1, n2, n1_prime, n2_prime = two_descent_by_two_isogeny(E)
sage: log(n1,2) + log(n1_prime,2) - 2 # the rank
1

sage: # needs sage.symbolic
sage: x,y = var('x,y')
sage: E = EllipticCurve(y^2 == x^3 + x^2 - 25*x + 39)
sage: n1, n2, n1_prime, n2_prime = two_descent_by_two_isogeny(E)
sage: log(n1,2) + log(n1_prime,2) - 2  # the rank
2
sage: E = EllipticCurve(y^2 + x*y + y == x^3 - 131*x + 558)
```

(continues on next page)

(continued from previous page)

```
sage: n1, n2, n1_prime, n2_prime = two_descent_by_two_isogeny(E)
sage: log(n1,2) + log(n1_prime,2) - 2  # the rank
3
```

Using the verbosity option:

```
sage: E = EllipticCurve('14a')
sage: two_descent_by_two_isogeny(E, verbosity=1)
2-isogeny
Results:
2 <= #E(Q)/phi'(E'(Q)) <= 2
2 <= #E'(Q)/phi(E(Q)) <= 2
#Sel^(phi')(E'/Q) = 2
#Sel^(phi)(E/Q) = 2
1 <= #Sha(E'/Q)[phi'] <= 1
1 <= #Sha(E/Q)[phi] <= 1
1 <= #Sha(E/Q)[2], #Sha(E'/Q)[2] <= 1
0 <= rank of E(Q) = rank of E'(Q) <= 0
(2, 2, 2, 2)
```

Handling curves whose discriminants involve larger than wordsize primes:

```
sage: E = EllipticCurve('14a')
sage: E = E.quadratic_twist(next_prime(10^20))
sage: E
Elliptic Curve defined by y^2 = x^3 + x^2 +␣
↪71666666666666667225666666666666666775672*x -␣
↪391925925925925926384240370370370370549019837037037037060249356 over Rational␣
↪Field
sage: E.discriminant().factor()
-1 * 2^18 * 7^3 * 10000000000000000039^6
sage: log(10000000000000000039.0, 2.0)
66.438...
sage: n1, n2, n1_prime, n2_prime = two_descent_by_two_isogeny(E)
sage: log(n1,2) + log(n1_prime,2) - 2 # the rank
0
```

sage.schemes.elliptic_curves.descent_two_isogeny.**two_descent_by_two_isogeny_work**(*c*, *d*, *global_limit_s*, *global_limit_l*, *verbosity=0*, *selmer_only=0*, *proof=1*)

Do all the work in doing a two-isogeny descent.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.descent_two_isogeny import two_descent_by_
↪two_isogeny_work
sage: n1, n2, n1_prime, n2_prime = two_descent_by_two_isogeny_work(13,128)
sage: log(n1,2) + log(n1_prime,2) - 2 # the rank
0
sage: n1, n2, n1_prime, n2_prime = two_descent_by_two_isogeny_work(1,-16)
sage: log(n1,2) + log(n1_prime,2) - 2 # the rank
```

(continues on next page)

(continued from previous page)

```
1
sage: n1, n2, n1_prime, n2_prime = two_descent_by_two_isogeny_work(10,8)
sage: log(n1,2) + log(n1_prime,2) - 2 # the rank
2
sage: n1, n2, n1_prime, n2_prime = two_descent_by_two_isogeny_work(85,320)
sage: log(n1,2) + log(n1_prime,2) - 2 # the rank
3
```

19.2 Elliptic curves with prescribed good reduction

Construction of elliptic curves with good reduction outside a finite set of primes

A theorem of Shafarevich states that, over a number field K, given any finite set S of primes of K, there are (up to isomorphism) only a finite set of elliptic curves defined over K with good reduction at all primes outside S. An explicit form of the theorem with an algorithm for finding this finite set was given in "Finding all elliptic curves with good reduction outside a given set of primes" by John Cremona and Mark Lingham, Experimental Mathematics 16 No.3 (2007), 303-312. The method requires computation of the class and unit groups of K as well as all the S-integral points on a collection of auxiliary elliptic curves defined over K.

This implementation (April 2009) is only for the case $K = \mathbf{Q}$, where in many cases the determination of the necessary sets of S-integral points is possible. The main user-level function is *EllipticCurves_with_good_reduction_outside_S()*, defined in constructor.py. Users should note carefully the following points:

(1) the number of auxiliary curves to be considered is exponential in the size of S (specifically, 2.6^s where $s = |S|$).

(2) For some of the auxiliary curves it is impossible at present to provably find all the S-integral points using the current algorithms, which rely on first finding a basis for their Mordell-Weil groups using 2-descent. A warning is output in cases where the set of points (and hence the final output) is not guaranteed to be complete. Using the `proof=False` flag suppresses these warnings.

EXAMPLES: We find all elliptic curves with good reduction outside 2, listing the label of each:

```
sage: [e.label() for e in EllipticCurves_with_good_reduction_outside_S([2])]  # long␣
↪time (5s on sage.math, 2013)
['32a1',
'32a2',
'32a3',
'32a4',
'64a1',
'64a2',
'64a3',
'64a4',
'128a1',
'128a2',
'128b1',
'128b2',
'128c1',
'128c2',
'128d1',
'128d2',
'256a1',
'256a2',
'256b1',
'256b2',
'256c1',
```

(continues on next page)

(continued from previous page)

```
'256c2',
'256d1',
'256d2']
```

Secondly we try the same with $S = 11$; note that warning messages are printed without `proof=False` (unless the optional database is installed: two of the auxiliary curves whose Mordell-Weil bases are required have conductors 13068 and 52272 so are in the database):

```
sage: [e.label() for e in EllipticCurves_with_good_reduction_outside_S([11],␣
↪proof=False)]  # long time (13s on sage.math, 2011)
['11a1', '11a2', '11a3', '121a1', '121a2', '121b1', '121b2', '121c1', '121c2', '121d1
↪', '121d2', '121d3']
```

AUTHORS:

- John Cremona (6 April 2009): initial version (over **Q** only).

sage.schemes.elliptic_curves.ell_egros.**curve_key**(*E1*)

Comparison key for elliptic curves over **Q**.

The key is a tuple:

- if the curve is in the database: (conductor, 0, label, number)
- otherwise: (conductor, 1, a_invariants)

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_egros import curve_key
sage: E = EllipticCurve_from_j(1728)
sage: curve_key(E)
(32, 0, 0, 2)
sage: E = EllipticCurve_from_j(1729)
sage: curve_key(E)
(2989441, 1, (1, 0, 0, -36, -1))
```

sage.schemes.elliptic_curves.ell_egros.**egros_from_j**(*j*, *S=[]*)

Given a rational j and a list of primes S, returns a list of elliptic curves over **Q** with j-invariant j and good reduction outside S, by checking all relevant quadratic twists.

INPUT:

- j – a rational number.
- S – list of primes (default: empty list).

Note: Primality of elements of S is not checked, and the output is undefined if S is not a list or contains non-primes.

OUTPUT:

A sorted list of all elliptic curves defined over **Q** with j-invariant equal to j and with good reduction at all primes outside the list `S`.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_egros import egros_from_j
sage: [e.label() for e in egros_from_j(0,[3])]
['27a1', '27a3', '243a1', '243a2', '243b1', '243b2']
```

(continues on next page)

(continued from previous page)

```
sage: [e.label() for e in egros_from_j(1728,[2])]
['32a1', '32a2', '64a1', '64a4', '256b1', '256b2', '256c1', '256c2']
sage: elist=egros_from_j(-4096/11,[11])
sage: [e.label() for e in elist]
['11a3', '121d1']
```

sage.schemes.elliptic_curves.ell_egros.**egros_from_j_0**(*S=[]*)

Given a list of primes S, returns a list of elliptic curves over **Q** with j-invariant 0 and good reduction outside S, by checking all relevant sextic twists.

INPUT:

- S – list of primes (default: empty list).

Note: Primality of elements of S is not checked, and the output is undefined if S is not a list or contains non-primes.

OUTPUT:

A sorted list of all elliptic curves defined over **Q** with j-invariant equal to 0 and with good reduction at all primes outside the list `S`.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_egros import egros_from_j_0
sage: egros_from_j_0([])
[]
sage: egros_from_j_0([2])
[]
sage: [e.label() for e in egros_from_j_0([3])]
['27a1', '27a3', '243a1', '243a2', '243b1', '243b2']
sage: len(egros_from_j_0([2,3,5]))  # long time (8s on sage.math, 2013)
432
```

sage.schemes.elliptic_curves.ell_egros.**egros_from_j_1728**(*S=[]*)

Given a list of primes S, returns a list of elliptic curves over **Q** with j-invariant 1728 and good reduction outside S, by checking all relevant quartic twists.

INPUT:

- S – list of primes (default: empty list).

Note: Primality of elements of S is not checked, and the output is undefined if S is not a list or contains non-primes.

OUTPUT:

A sorted list of all elliptic curves defined over **Q** with j-invariant equal to 1728 and with good reduction at all primes outside the list `S`.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_egros import egros_from_j_1728
sage: egros_from_j_1728([])
[]
sage: egros_from_j_1728([3])
[]
```

(continues on next page)

(continued from previous page)

```
sage: [e.cremona_label() for e in egros_from_j_1728([2])]
['32a1', '32a2', '64a1', '64a4', '256b1', '256b2', '256c1', '256c2']
```

sage.schemes.elliptic_curves.ell_egros.**egros_from_jlist**(*jlist*, *S=[]*)

Given a list of rational j and a list of primes S, returns a list of elliptic curves over **Q** with j-invariant in the list and good reduction outside S.

INPUT:

- j – list of rational numbers.
- S – list of primes (default: empty list).

Note: Primality of elements of S is not checked, and the output is undefined if S is not a list or contains non-primes.

OUTPUT:

A sorted list of all elliptic curves defined over **Q** with j-invariant in the list `jlist` and with good reduction at all primes outside the list `S`.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_egros import egros_get_j, egros_from_
↪jlist
sage: jlist=egros_get_j([3])
sage: elist=egros_from_jlist(jlist,[3])
sage: [e.label() for e in elist]
['27a1', '27a2', '27a3', '27a4', '243a1', '243a2', '243b1', '243b2']
sage: [e.ainvs() for e in elist]
[(0, 0, 1, 0, -7),
(0, 0, 1, -270, -1708),
(0, 0, 1, 0, 0),
(0, 0, 1, -30, 63),
(0, 0, 1, 0, -1),
(0, 0, 1, 0, 20),
(0, 0, 1, 0, 2),
(0, 0, 1, 0, -61)]
```

sage.schemes.elliptic_curves.ell_egros.**egros_get_j**(*S=[]*, *proof=None*, *verbose=False*)

Returns a list of rational j such that all elliptic curves defined over **Q** with good reduction outside S have j-invariant in the list, sorted by height.

INPUT:

- `S` – list of primes (default: empty list).
- `proof` – `True`/`False` (default `True`): the MW basis for auxiliary curves will be computed with this proof flag.
- `verbose` – `True`/`False` (default `False``): if `True`, some details of the computation will be output.

Note: Proof flag: The algorithm used requires determining all S-integral points on several auxiliary curves, which in turn requires the computation of their generators. This is not always possible (even in theory) using current knowledge.

The value of this flag is passed to the function which computes generators of various auxiliary elliptic curves, in order to find their S-integral points. Set to `False` if the default (`True`) causes warning messages, but note that

you can then not rely on the set of invariants returned being complete.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_egros import egros_get_j
sage: egros_get_j([])
[1728]
sage: egros_get_j([2])  # long time (3s on sage.math, 2013)
[128, 432, -864, 1728, 3375/2, -3456, 6912, 8000, 10976, -35937/4, 287496, -
↪784446336, -189613868625/128]
sage: egros_get_j([3])  # long time (3s on sage.math, 2013)
[0, -576, 1536, 1728, -5184, -13824, 21952/9, -41472, 140608/3, -12288000]
sage: jlist=egros_get_j([2,3]); len(jlist) # long time (30s)
83
```

sage.schemes.elliptic_curves.ell_egros.**is_possible_j**(*j*, *S=[]*)

Tests if the rational j is a possible j-invariant of an elliptic curve with good reduction outside S.

Note: The condition used is necessary but not sufficient unless S contains both 2 and 3.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_egros import is_possible_j
sage: is_possible_j(0,[])
False
sage: is_possible_j(1728,[])
True
sage: is_possible_j(-4096/11,[11])
True
```

19.3 Elliptic curves over padic fields

class sage.schemes.elliptic_curves.ell_padic_field.**EllipticCurve_padic_field**(*R*, *data*, *category=None*)

Bases: *EllipticCurve_field*, *HyperellipticCurve_padic_field*

Elliptic curve over a padic field.

EXAMPLES:

```
sage: Qp = pAdicField(17)
sage: E = EllipticCurve(Qp,[2,3]); E
Elliptic Curve defined by y^2  = x^3 + (2+O(17^20))*x + (3+O(17^20))
 over 17-adic Field with capped relative precision 20
sage: E == loads(dumps(E))
True
```

frobenius(*P=None*)

Return the Frobenius as a function on the group of points of this elliptic curve.

EXAMPLES:

```
sage: Qp = pAdicField(13)
sage: E = EllipticCurve(Qp,[1,1])
sage: type(E.frobenius())
<... 'function'>
sage: point = E(0,1)
sage: E.frobenius(point)
(0 : 1 + O(13^20) : 1 + O(13^20))
```

Check that Issue #29709 is fixed:

```
sage: Qp = pAdicField(13)
sage: E = EllipticCurve(Qp,[0,0,1,0,1])
sage: E.frobenius(E(1,1))
Traceback (most recent call last):
...
NotImplementedError: Curve must be in weierstrass normal form.
sage: E = EllipticCurve(Qp,[0,1,0,0,1])
sage: E.frobenius(E(0,1))
(0 : 1 + O(13^20) : 1 + O(13^20))
```

19.4 Denis Simon's PARI scripts

sage.schemes.elliptic_curves.gp_simon.**init**()

Function to initialize the gp process

sage.schemes.elliptic_curves.gp_simon.**simon_two_descent**(*E*, *verbose=0*, *lim1=None*, *lim3=None*, *limtriv=None*, *maxprob=20*, *limbigprime=30*, *known_points=[]*)

Interface to Simon's gp script for two-descent.

Note: Users should instead run E.simon_two_descent()

EXAMPLES:

```
sage: import sage.schemes.elliptic_curves.gp_simon
sage: E = EllipticCurve('389a1')
sage: sage.schemes.elliptic_curves.gp_simon.simon_two_descent(E)
(2, 2, [(5/4 : 5/8 : 1), (-3/4 : 7/8 : 1)])
```

19.5 Elliptic curves with congruent mod-5 representation

AUTHORS:

- Alice Silverberg and Karl Rubin – original PARI/GP version
- William Stein – Sage version

sage.schemes.elliptic_curves.mod5family.**mod5family**(*a*, *b*)

Formulas for computing the family of elliptic curves with congruent mod-5 representation.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.mod5family import mod5family
sage: mod5family(0,1)
Elliptic Curve defined by y^2 = x^3 + (t^30+30*t^29+435*t^28+4060*t^27+27405*t^
↪26+142506*t^25+593775*t^24+2035800*t^23+5852925*t^22+14307150*t^21+30045015*t^
↪20+54627300*t^19+86493225*t^18+119759850*t^17+145422675*t^16+155117520*t^
↪15+145422675*t^14+119759850*t^13+86493225*t^12+54627300*t^11+30045015*t^
↪10+14307150*t^9+5852925*t^8+2035800*t^7+593775*t^6+142506*t^5+27405*t^4+4060*t^
↪3+435*t^2+30*t+1) over Fraction Field of Univariate Polynomial Ring in t over␣
↪Rational Field
```

19.6 Morphism to bring a genus-one curve into Weierstrass form

You should use *EllipticCurve_from_cubic()* or `EllipticCurve_from_curve()` to construct the transformation starting with a cubic or with a genus one curve.

EXAMPLES:

```
sage: R.<u,v,w> = QQ[]
sage: f = EllipticCurve_from_cubic(u^3 + v^3 + w^3, [1,-1,0], morphism=True);  f
Scheme morphism:
  From: Projective Plane Curve over Rational Field defined by u^3 + v^3 + w^3
  To:   Elliptic Curve defined by y^2 - 9*y = x^3 - 27 over Rational Field
  Defn: Defined on coordinates by sending (u : v : w) to
        (-w : 3*u : 1/3*u + 1/3*v)

sage: finv = f.inverse();  finv
Scheme morphism:
  From: Elliptic Curve defined by y^2 - 9*y = x^3 - 27 over Rational Field
  To:   Projective Plane Curve over Rational Field defined by u^3 + v^3 + w^3
  Defn: Defined on coordinates by sending (x : y : z) to
        (1/3*y : -1/3*y + 3*z : -x)

sage: (u^3 + v^3 + w^3)(f.inverse().defining_polynomials()) * f.inverse().post_
↪rescaling()
-x^3 + y^2*z - 9*y*z^2 + 27*z^3

sage: E = finv.domain()
sage: E.defining_polynomial()(f.defining_polynomials()) * f.post_rescaling()
u^3 + v^3 + w^3

sage: f([1,-1,0])
(0 : 1 : 0)
sage: f([1,0,-1])
(3 : 9 : 1)
sage: f([0,1,-1])
(3 : 0 : 1)
```

class sage.schemes.elliptic_curves.weierstrass_transform.**WeierstrassTransformation**(*domain*, *codomain*, *defining_polynomials*, *post_multiplication*)

Bases: SchemeMorphism_polynomial

A morphism of a genus-one curve to/from the Weierstrass form.

INPUT:

- domain, codomain – two schemes, one of which is an elliptic curve.
- defining_polynomials – triplet of polynomials that define the transformation.
- post_multiplication – a polynomial to homogeneously rescale after substituting the defining polynomials.

EXAMPLES:

```
sage: P2.<u,v,w> = ProjectiveSpace(2,QQ)
sage: C = P2.subscheme(u^3 + v^3 + w^3)
sage: E = EllipticCurve([2, -1, -1/3, 1/3, -1/27])
sage: from sage.schemes.elliptic_curves.weierstrass_transform import␣
↪WeierstrassTransformation
sage: f = WeierstrassTransformation(C, E, [w, -v-w, -3*u-3*v], 1);  f
Scheme morphism:
  From: Closed subscheme of Projective Space of dimension 2 over Rational Field␣
↪defined by:
  u^3 + v^3 + w^3
  To:   Elliptic Curve defined by y^2 + 2*x*y - 1/3*y = x^3 - x^2 + 1/3*x - 1/27
        over Rational Field
  Defn: Defined on coordinates by sending (u : v : w) to
        (w : -v - w : -3*u - 3*v)

sage: f([-1, 1, 0])
(0 : 1 : 0)
sage: f([-1, 0, 1])
(1/3 : -1/3 : 1)
sage: f([ 0,-1, 1])
(1/3 : 0 : 1)

sage: A2.<a,b> = AffineSpace(2,QQ)
sage: C = A2.subscheme(a^3 + b^3 + 1)
sage: f = WeierstrassTransformation(C, E, [1, -b-1, -3*a-3*b], 1);  f
Scheme morphism:
  From: Closed subscheme of Affine Space of dimension 2 over Rational Field␣
↪defined by:
  a^3 + b^3 + 1
  To:   Elliptic Curve defined by y^2 + 2*x*y - 1/3*y
        = x^3 - x^2 + 1/3*x - 1/27 over Rational Field
```

(continues on next page)

(continued from previous page)

```
  Defn: Defined on coordinates by sending (a, b) to
        (1 : -b - 1 : -3*a - 3*b)
sage: f([-1,0])
(1/3 : -1/3 : 1)
sage: f([0,-1])
(1/3 : 0 : 1)
```

post_rescaling()

Return the homogeneous rescaling to apply after the coordinate substitution.

OUTPUT:

A polynomial. See the example below.

EXAMPLES:

```
sage: R.<a,b,c> = QQ[]
sage: cubic =  a^3+7*b^3+64*c^3
sage: P = [2,2,-1]
sage: f = EllipticCurve_from_cubic(cubic, P, morphism=True).inverse()
sage: f.post_rescaling()
-1/7
```

So here is what it does. If we just plug in the coordinate transformation, we get the defining polynomial up to scale. This method returns the overall rescaling of the equation to bring the result into the standard form:

```
sage: cubic(f.defining_polynomials())
7*x^3 - 7*y^2*z + 1806336*y*z^2 - 155373797376*z^3
sage: cubic(f.defining_polynomials()) * f.post_rescaling()
-x^3 + y^2*z - 258048*y*z^2 + 22196256768*z^3
```

sage.schemes.elliptic_curves.weierstrass_transform.**WeierstrassTransformationWithInverse**(*domain*, *codomain*, *defining_polynomials*, *post_multiplication*, *inverting_polynomials*, *inv_post_multiplication*)

Construct morphism of a genus-one curve to/from the Weierstrass form with its inverse.

EXAMPLES:

```
sage: R.<u,v,w> = QQ[]
sage: f = EllipticCurve_from_cubic(u^3 + v^3 + w^3, [1,-1,0], morphism=True);  f
Scheme morphism:
  From: Projective Plane Curve over Rational Field defined by u^3 + v^3 + w^3
  To:   Elliptic Curve defined by y^2 - 9*y = x^3 - 27 over Rational Field
  Defn: Defined on coordinates by sending (u : v : w) to
    (-w : 3*u : 1/3*u + 1/3*v)

Scheme morphism:
  From: Closed subscheme of Projective Space of dimension 2 over Rational Field␣
↪defined by:
  u^3 + v^3 + w^3
  To:   Elliptic Curve defined by y^2 + 2*x*y + 1/3*y
        = x^3 - x^2 - 1/3*x - 1/27 over Rational Field
  Defn: Defined on coordinates by sending (u : v : w) to
        (-w : -v + w : 3*u + 3*v)
```

class sage.schemes.elliptic_curves.weierstrass_transform.**WeierstrassTransformationWithInver**

Bases: *WeierstrassTransformation*

inverse()

Return the inverse.

OUTPUT:

A morphism in the opposite direction. This may be a rational inverse or an analytic inverse.

EXAMPLES:

```
sage: R.<u,v,w> = QQ[]
sage: f = EllipticCurve_from_cubic(u^3 + v^3 + w^3, [1,-1,0], morphism=True)
sage: f.inverse()
Scheme morphism:
  From: Elliptic Curve defined by y^2 - 9*y = x^3 - 27 over Rational Field
  To:   Projective Plane Curve over Rational Field defined by u^3 + v^3 + w^3
  Defn: Defined on coordinates by sending (x : y : z) to
  (1/3*y : -1/3*y + 3*z : -x)
```

CHAPTER

TWENTY

HYPERELLIPTIC CURVES

20.1 Hyperelliptic curve constructor

AUTHORS:

- David Kohel (2006): initial version
- Anna Somoza (2019-04): dynamic class creation

sage.schemes.hyperelliptic_curves.constructor.**HyperellipticCurve**(*f*, *h=0*, *names=None*, *PP=None*, *check_squarefree=True*)

Returns the hyperelliptic curve $y^2 + hy = f$, for univariate polynomials h and f. If h is not given, then it defaults to 0.

INPUT:

- `f` – univariate polynomial
- `h` – optional univariate polynomial
- `names` (default: `["x","y"]`) – names for the coordinate functions
- `check_squarefree` (default: `True`) – test if the input defines a hyperelliptic curve when f is homogenized to degree $2g + 2$ and h to degree $g + 1$ for some g.

Warning: When setting `check_squarefree=False` or using a base ring that is not a field, the output curves are not to be trusted. For example, the output of `is_singular` is always `False`, without this being properly tested in that case.

Note: The words "hyperelliptic curve" are normally only used for curves of genus at least two, but this class allows more general smooth double covers of the projective line (conics and elliptic curves), even though the class is not meant for those and some outputs may be incorrect.

EXAMPLES:

Basic examples:

```
sage: R.<x> = QQ[]
sage: HyperellipticCurve(x^5 + x + 1)
Hyperelliptic Curve over Rational Field defined by y^2 = x^5 + x + 1
sage: HyperellipticCurve(x^19 + x + 1, x - 2)
Hyperelliptic Curve over Rational Field defined by y^2 + (x - 2)*y = x^19 + x + 1

sage: k.<a> = GF(9); R.<x> = k[]                                                #␣
↪needs sage.rings.finite_rings
sage: HyperellipticCurve(x^3 + x - 1, x+a)                                      #␣
↪needs sage.rings.finite_rings
Hyperelliptic Curve over Finite Field in a of size 3^2
 defined by y^2 + (x + a)*y = x^3 + x + 2
```

Characteristic two:

```
sage: # needs sage.rings.finite_rings
sage: P.<x> = GF(8, 'a')[]
sage: HyperellipticCurve(x^7 + 1, x)
Hyperelliptic Curve over Finite Field in a of size 2^3
 defined by y^2 + x*y = x^7 + 1
sage: HyperellipticCurve(x^8 + x^7 + 1, x^4 + 1)
Hyperelliptic Curve over Finite Field in a of size 2^3
 defined by y^2 + (x^4 + 1)*y = x^8 + x^7 + 1
sage: HyperellipticCurve(x^8 + 1, x)
Traceback (most recent call last):
...
ValueError: not a hyperelliptic curve: highly singular at infinity
sage: HyperellipticCurve(x^8 + x^7 + 1, x^4)
Traceback (most recent call last):
...
ValueError: not a hyperelliptic curve: singularity in the provided affine patch

sage: F.<t> = PowerSeriesRing(FiniteField(2))
sage: P.<x> = PolynomialRing(FractionField(F))
sage: HyperellipticCurve(x^5 + t, x)
Hyperelliptic Curve over Laurent Series Ring in t over Finite Field of size 2
 defined by y^2 + x*y = x^5 + t
```

We can change the names of the variables in the output:

```
sage: k.<a> = GF(9); R.<x> = k[]                                                #␣
↪needs sage.rings.finite_rings
sage: HyperellipticCurve(x^3 + x - 1, x + a, names=['X','Y'])                   #␣
↪needs sage.rings.finite_rings
Hyperelliptic Curve over Finite Field in a of size 3^2
 defined by Y^2 + (X + a)*Y = X^3 + X + 2
```

This class also allows curves of genus zero or one, which are strictly speaking not hyperelliptic:

```
sage: P.<x> = QQ[]
sage: HyperellipticCurve(x^2 + 1)
Hyperelliptic Curve over Rational Field defined by y^2 = x^2 + 1
sage: HyperellipticCurve(x^4 - 1)
Hyperelliptic Curve over Rational Field defined by y^2 = x^4 - 1
sage: HyperellipticCurve(x^3 + 2*x + 2)
Hyperelliptic Curve over Rational Field defined by y^2 = x^3 + 2*x + 2
```

Double roots:

```
sage: P.<x> = GF(7)[]
sage: HyperellipticCurve((x^3-x+2)^2*(x^6-1))
Traceback (most recent call last):
...
ValueError: not a hyperelliptic curve: singularity in the provided affine patch

sage: HyperellipticCurve((x^3-x+2)^2*(x^6-1), check_squarefree=False)
Hyperelliptic Curve over Finite Field of size 7 defined by
 y^2 = x^12 + 5*x^10 + 4*x^9 + x^8 + 3*x^7 + 3*x^6 + 2*x^4 + 3*x^3 + 6*x^2 + 4*x␣
↪+ 3
```

The input for a (smooth) hyperelliptic curve of genus g should not contain polynomials of degree greater than $2g + 2$. In the following example, the hyperelliptic curve has genus 2 and there exists a model $y^2 = F$ of degree 6, so the model $y^2 + yh = f$ of degree 200 is not allowed.:

```
sage: P.<x> = QQ[]
sage: h = x^100
sage: F = x^6 + 1
sage: f = F - h^2/4
sage: HyperellipticCurve(f, h)
Traceback (most recent call last):
...
ValueError: not a hyperelliptic curve: highly singular at infinity

sage: HyperellipticCurve(F)
Hyperelliptic Curve over Rational Field defined by y^2 = x^6 + 1
```

An example with a singularity over an inseparable extension of the base field:

```
sage: F.<t> = GF(5)[]
sage: P.<x> = F[]
sage: HyperellipticCurve(x^5 + t)
Traceback (most recent call last):
...
ValueError: not a hyperelliptic curve: singularity in the provided affine patch
```

Input with integer coefficients creates objects with the integers as base ring, but only checks smoothness over $\mathbf{Q}$, not over Spec($\mathbf{Z}$). In other words, it is checked that the discriminant is non-zero, but it is not checked whether the discriminant is a unit in $\mathbf{Z}^*$.:

```
sage: P.<x> = ZZ[]
sage: HyperellipticCurve(3*x^7 + 6*x + 6)
Hyperelliptic Curve over Integer Ring defined by y^2 = 3*x^7 + 6*x + 6
```

20.2 Hyperelliptic curves over a general ring

EXAMPLES:

```
sage: P.<x> = GF(5)[]
sage: f = x^5 - 3*x^4 - 2*x^3 + 6*x^2 + 3*x - 1
sage: C = HyperellipticCurve(f); C
Hyperelliptic Curve over Finite Field of size 5
 defined by y^2 = x^5 + 2*x^4 + 3*x^3 + x^2 + 3*x + 4
```

```
sage: P.<x> = QQ[]
sage: f = 4*x^5 - 30*x^3 + 45*x - 22
sage: C = HyperellipticCurve(f); C
Hyperelliptic Curve over Rational Field defined by y^2 = 4*x^5 - 30*x^3 + 45*x - 22
sage: C.genus()
2

sage: D = C.affine_patch(0)
sage: D.defining_polynomials()[0].parent()
Multivariate Polynomial Ring in x1, x2 over Rational Field
```

class sage.schemes.hyperelliptic_curves.hyperelliptic_generic.**HyperellipticCurve_generic**(*Pl*
f,
h=
na
ge

Bases: ProjectivePlaneCurve

base_extend(*R*)

Returns this HyperellipticCurve over a new base ring R.

EXAMPLES:

```
sage: # needs sage.rings.padics
sage: R.<x> = QQ[]
sage: H = HyperellipticCurve(x^5 - 10*x + 9)
sage: K = Qp(3, 5)
sage: L.<a> = K.extension(x^30 - 3)
sage: HK = H.change_ring(K)
sage: HL = HK.change_ring(L); HL
Hyperelliptic Curve
 over 3-adic Eisenstein Extension Field in a defined by x^30 - 3
 defined by (1 + O(a^150))*y^2 = (1 + O(a^150))*x^5
  + (2 + 2*a^30 + a^60 + 2*a^90 + 2*a^120 + O(a^150))*x + a^60 + O(a^210)

sage: R.<x> = FiniteField(7)[]
sage: H = HyperellipticCurve(x^8 + x + 5)
sage: H.base_extend(FiniteField(7^2, 'a'))                                  #␣
↪needs sage.rings.finite_rings
Hyperelliptic Curve over Finite Field in a of size 7^2
 defined by y^2 = x^8 + x + 5
```

change_ring(*R*)

Returns this HyperellipticCurve over a new base ring R.

EXAMPLES:

```
sage: # needs sage.rings.padics
sage: R.<x> = QQ[]
sage: H = HyperellipticCurve(x^5 - 10*x + 9)
sage: K = Qp(3, 5)
sage: L.<a> = K.extension(x^30 - 3)
sage: HK = H.change_ring(K)
sage: HL = HK.change_ring(L); HL
Hyperelliptic Curve
 over 3-adic Eisenstein Extension Field in a defined by x^30 - 3
 defined by (1 + O(a^150))*y^2 = (1 + O(a^150))*x^5
```

(continues on next page)

(continued from previous page)

```
  + (2 + 2*a^30 + a^60 + 2*a^90 + 2*a^120 + O(a^150))*x + a^60 + O(a^210)

sage: R.<x> = FiniteField(7)[]
sage: H = HyperellipticCurve(x^8 + x + 5)
sage: H.base_extend(FiniteField(7^2, 'a'))                                  #␣
↪needs sage.rings.finite_rings
Hyperelliptic Curve over Finite Field in a of size 7^2
 defined by y^2 = x^8 + x + 5
```

genus()

has_odd_degree_model()

Return True if an odd degree model of self exists over the field of definition; False otherwise.

Use `odd_degree_model` to calculate an odd degree model.

EXAMPLES:

```
sage: x = QQ['x'].0
sage: HyperellipticCurve(x^5 + x).has_odd_degree_model()
True
sage: HyperellipticCurve(x^6 + x).has_odd_degree_model()
True
sage: HyperellipticCurve(x^6 + x + 1).has_odd_degree_model()
False
```

hyperelliptic_polynomials(*K=None*, *var='x'*)

EXAMPLES:

```
sage: R.<x> = QQ[]; C = HyperellipticCurve(x^3 + x - 1, x^3/5); C
Hyperelliptic Curve over Rational Field defined by y^2 + 1/5*x^3*y = x^3 + x -
↪ 1
sage: C.hyperelliptic_polynomials()
(x^3 + x - 1, 1/5*x^3)
```

invariant_differential()

Returns $dx/2y$, as an element of the Monsky-Washnitzer cohomology of self

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: C = HyperellipticCurve(x^5 - 4*x + 4)
sage: C.invariant_differential()
1 dx/2y
```

is_singular()

Returns False, because hyperelliptic curves are smooth projective curves, as checked on construction.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: H = HyperellipticCurve(x^5 + 1)
sage: H.is_singular()
False
```

A hyperelliptic curve with genus at least 2 always has a singularity at infinity when viewed as a *plane* projective curve. This can be seen in the following example.:

```
sage: R.<x> = QQ[]
sage: H = HyperellipticCurve(x^5 + 2)
sage: from sage.misc.verbose import set_verbose
sage: set_verbose(-1)
sage: H.is_singular()
False
sage: from sage.schemes.curves.projective_curve import ProjectivePlaneCurve
sage: ProjectivePlaneCurve.is_singular(H)
True
```

is_smooth()

Returns True, because hyperelliptic curves are smooth projective curves, as checked on construction.

EXAMPLES:

```
sage: R.<x> = GF(13)[]
sage: H = HyperellipticCurve(x^8 + 1)
sage: H.is_smooth()
True
```

A hyperelliptic curve with genus at least 2 always has a singularity at infinity when viewed as a *plane* projective curve. This can be seen in the following example.:

```
sage: # needs sage.rings.finite_rings
sage: R.<x> = GF(27, 'a')[]
sage: H = HyperellipticCurve(x^10 + 2)
sage: from sage.misc.verbose import set_verbose
sage: set_verbose(-1)
sage: H.is_smooth()
True
sage: from sage.schemes.curves.projective_curve import ProjectivePlaneCurve
sage: ProjectivePlaneCurve.is_smooth(H)
False
```

is_x_coord(x)

Return True if x is the x-coordinate of a point on this curve.

See also:

See also *lift_x()* to find the point(s) with a given x-coordinate. This function may be useful in cases where testing an element of the base field for being a square is faster than finding its square root.

INPUT:

- x – an element of the base ring of the curve

OUTPUT:

A bool stating whether or not x is a x-coordinate of a point on the curve

EXAMPLES:

When x is the x-coordinate of a rational point on the curve, we can request these:

```
sage: R.<x> = PolynomialRing(QQ)
sage: f = x^5 + x^3 + 1
sage: H = HyperellipticCurve(f)
sage: H.is_x_coord(0)
True
```

There are no rational points with x-coordinate 3:

```
sage: H.is_x_coord(3)
False
```

The function also handles the case when $h(x)$ is not zero:

```
sage: R.<x> = PolynomialRing(QQ)
sage: f = x^5 + x^3 + 1
sage: h = x + 1
sage: H = HyperellipticCurve(f, h)
sage: H.is_x_coord(1)
True
```

We can perform these operations over finite fields too:

```
sage: # needs sage.rings.finite_rings
sage: R.<x> = PolynomialRing(GF(163))
sage: f = x^7 + x + 1
sage: H = HyperellipticCurve(f)
sage: H.is_x_coord(13)
True
```

Including the case of characteristic two:

```
sage: # needs sage.rings.finite_rings
sage: F.<z4> = GF(2^4)
sage: R.<x> = PolynomialRing(F)
sage: f = x^7 + x^3 + 1
sage: h = x + 1
sage: H = HyperellipticCurve(f, h)
sage: H.is_x_coord(z4^3 + z4^2 + z4)
True
```

AUTHORS:

- Giacomo Pope (2024): adapted from *lift_x()*

jacobian()

lift_x(*x*, *all=False*)

Return one or all points with given x-coordinate.

This method is deterministic: It returns the same data each time when called again with the same x.

INPUT:

- `x` – an element of the base ring of the curve
- `all` (bool, default `False`) – if `True`, return a (possibly empty) list of all points; if `False`, return just one point, or raise a `ValueError` if there are none.

OUTPUT:

A point or list of up to two points on this curve.

See also:

is_x_coord()

AUTHORS:

- Giacomo Pope (2024): Allowed for the case of characteristic two

EXAMPLES:

When x is the x-coordinate of a rational point on the curve, we can request these:

```
sage: R.<x> = PolynomialRing(QQ)
sage: f = x^5 + x^3 + 1
sage: H = HyperellipticCurve(f)
sage: H.lift_x(0)
(0 : -1 : 1)
sage: H.lift_x(4, all=True)
[(4 : -33 : 1), (4 : 33 : 1)]
```

There are no rational points with x-coordinate 3:

```
sage: H.lift_x(3)
Traceback (most recent call last):
...
ValueError: No point with x-coordinate 3 on Hyperelliptic Curve over Rational␣
↪Field defined by y^2 = x^5 + x^3 + 1
```

An empty list is returned when there are no points and `all=True`:

```
sage: H.lift_x(3, all=True)
[]
```

The function also handles the case when $h(x)$ is not zero:

```
sage: R.<x> = PolynomialRing(QQ)
sage: f = x^5 + x^3 + 1
sage: h = x + 1
sage: H = HyperellipticCurve(f, h)
sage: H.lift_x(1)
(1 : -3 : 1)
```

We can perform these operations over finite fields too:

```
sage: # needs sage.rings.finite_rings
sage: R.<x> = PolynomialRing(GF(163))
sage: f = x^7 + x + 1
sage: H = HyperellipticCurve(f)
sage: H.lift_x(13)
(13 : 41 : 1)
```

Including the case of characteristic two:

```
sage: # needs sage.rings.finite_rings
sage: F.<z4> = GF(2^4)
sage: R.<x> = PolynomialRing(F)
sage: f = x^7 + x^3 + 1
sage: h = x + 1
sage: H = HyperellipticCurve(f, h)
sage: H.lift_x(z4^3 + z4^2 + z4, all=True)
[(z4^3 + z4^2 + z4 : z4^2 + z4 + 1 : 1), (z4^3 + z4^2 + z4 : z4^3 : 1)]
```

local_coord(*P*, *prec=20*, *name='t'*)

Calls the appropriate local_coordinates function

INPUT:

- P – a point on self
- prec – desired precision of the local coordinates
- name – generator of the power series ring (default: t)

OUTPUT:

$(x(t), y(t))$ such that $y(t)^2 = f(x(t))$, where t is the local parameter at P

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: H = HyperellipticCurve(x^5 - 23*x^3 + 18*x^2 + 40*x)
sage: H.local_coord(H(1 ,6), prec=5)
(1 + t + O(t^5), 6 + t - 7/2*t^2 - 1/2*t^3 - 25/48*t^4 + O(t^5))
sage: H.local_coord(H(4, 0), prec=7)
(4 + 1/360*t^2 - 191/23328000*t^4 + 7579/188956800000*t^6 + O(t^7), t + O(t^
↪7))
sage: H.local_coord(H(0, 1, 0), prec=5)
(t^-2 + 23*t^2 - 18*t^4 - 569*t^6 + O(t^7),
 t^-5 + 46*t^-1 - 36*t - 609*t^3 + 1656*t^5 + O(t^6))
```

AUTHOR:

- Jennifer Balakrishnan (2007-12)

local_coordinates_at_infinity (*prec=20*, *name='t'*)

For the genus g hyperelliptic curve $y^2 = f(x)$, return $(x(t), y(t))$ such that $(y(t))^2 = f(x(t))$, where $t = x^g/y$ is the local parameter at infinity

INPUT:

- prec – desired precision of the local coordinates
- name – generator of the power series ring (default: t)

OUTPUT:

$(x(t), y(t))$ such that $y(t)^2 = f(x(t))$ and $t = x^g/y$ is the local parameter at infinity

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: H = HyperellipticCurve(x^5 - 5*x^2 + 1)
sage: x, y = H.local_coordinates_at_infinity(10)
sage: x
t^-2 + 5*t^4 - t^8 - 50*t^10 + O(t^12)
sage: y
t^-5 + 10*t - 2*t^5 - 75*t^7 + 50*t^11 + O(t^12)
```

```
sage: R.<x> = QQ['x']
sage: H = HyperellipticCurve(x^3 - x + 1)
sage: x, y = H.local_coordinates_at_infinity(10)
sage: x
t^-2 + t^2 - t^4 - t^6 + 3*t^8 + O(t^12)
sage: y
t^-3 + t - t^3 - t^5 + 3*t^7 - 10*t^11 + O(t^12)
```

AUTHOR:

- Jennifer Balakrishnan (2007-12)

local_coordinates_at_nonweierstrass(*P*, *prec=20*, *name='t'*)

For a non-Weierstrass point $P = (a, b)$ on the hyperelliptic curve $y^2 = f(x)$, return $(x(t), y(t))$ such that $(y(t))^2 = f(x(t))$, where $t = x - a$ is the local parameter.

INPUT:

- `P = (a, b)` – a non-Weierstrass point on self
- `prec` – desired precision of the local coordinates
- `name` – gen of the power series ring (default: `t`)

OUTPUT:

$(x(t), y(t))$ such that $y(t)^2 = f(x(t))$ and $t = x - a$ is the local parameter at P

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: H = HyperellipticCurve(x^5 - 23*x^3 + 18*x^2 + 40*x)
sage: P = H(1, 6)
sage: x, y = H.local_coordinates_at_nonweierstrass(P, prec=5)
sage: x
1 + t + O(t^5)
sage: y
6 + t - 7/2*t^2 - 1/2*t^3 - 25/48*t^4 + O(t^5)
sage: Q = H(-2, 12)
sage: x, y = H.local_coordinates_at_nonweierstrass(Q, prec=5)
sage: x
-2 + t + O(t^5)
sage: y
12 - 19/2*t - 19/32*t^2 + 61/256*t^3 - 5965/24576*t^4 + O(t^5)
```

AUTHOR:

- Jennifer Balakrishnan (2007-12)

local_coordinates_at_weierstrass(*P*, *prec=20*, *name='t'*)

For a finite Weierstrass point on the hyperelliptic curve $y^2 = f(x)$, returns $(x(t), y(t))$ such that $(y(t))^2 = f(x(t))$, where $t = y$ is the local parameter.

INPUT:

- `P` – a finite Weierstrass point on self
- `prec` – desired precision of the local coordinates
- `name` – gen of the power series ring (default: t)

OUTPUT:

$(x(t), y(t))$ such that $y(t)^2 = f(x(t))$ and $t = y$ is the local parameter at P

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: H = HyperellipticCurve(x^5 - 23*x^3 + 18*x^2 + 40*x)
sage: A = H(4, 0)
sage: x, y = H.local_coordinates_at_weierstrass(A, prec=7)
sage: x
4 + 1/360*t^2 - 191/23328000*t^4 + 7579/188956800000*t^6 + O(t^7)
sage: y
t + O(t^7)
```

(continues on next page)

(continued from previous page)

```
sage: B = H(-5, 0)
sage: x, y = H.local_coordinates_at_weierstrass(B, prec=5)
sage: x
-5 + 1/1260*t^2 + 887/2000376000*t^4 + O(t^5)
sage: y
t + O(t^5)
```

AUTHOR:

- Jennifer Balakrishnan (2007-12)
  - Francis Clarke (2012-08-26)

monsky_washnitzer_gens()

odd_degree_model()

Return an odd degree model of self, or raise ValueError if one does not exist over the field of definition.

EXAMPLES:

```
sage: x = QQ['x'].gen()
sage: H = HyperellipticCurve((x^2 + 2)*(x^2 + 3)*(x^2 + 5)); H
Hyperelliptic Curve over Rational Field defined by y^2 = x^6 + 10*x^4 + 31*x^
↪2 + 30
sage: H.odd_degree_model()
Traceback (most recent call last):
...
ValueError: No odd degree model exists over field of definition

sage: K2 = QuadraticField(-2, 'a')                                          #␣
↪needs sage.rings.number_field
sage: Hp2 = H.change_ring(K2).odd_degree_model(); Hp2                       #␣
↪needs sage.rings.number_field
Hyperelliptic Curve over Number Field in a
 with defining polynomial x^2 + 2 with a = 1.4142135623730950?*I
 defined by y^2 = 6*a*x^5 - 29*x^4 - 20*x^2 + 6*a*x + 1

sage: K3 = QuadraticField(-3, 'b')                                          #␣
↪needs sage.rings.number_field
sage: Hp3 = H.change_ring(QuadraticField(-3, 'b')).odd_degree_model(); Hp3  #␣
↪needs sage.rings.number_field
Hyperelliptic Curve over Number Field in b
 with defining polynomial x^2 + 3 with b = 1.7320508075688780?*I
 defined by y^2 = -4*b*x^5 - 14*x^4 - 20*b*x^3 - 35*x^2 + 6*b*x + 1

Of course, ``Hp2`` and ``Hp3`` are isomorphic over the composite
extension.  One consequence of this is that odd degree models
reduced over "different" fields should have the same number of
points on their reductions.  43 and 67 split completely in the
compositum, so when we reduce we find:

sage: # needs sage.rings.number_field
sage: P2 = K2.factor(43)[0][0]
sage: P3 = K3.factor(43)[0][0]
sage: Hp2.change_ring(K2.residue_field(P2)).frobenius_polynomial()
x^4 - 16*x^3 + 134*x^2 - 688*x + 1849
sage: Hp3.change_ring(K3.residue_field(P3)).frobenius_polynomial()
```

(continues on next page)

(continued from previous page)

```
x^4 - 16*x^3 + 134*x^2 - 688*x + 1849

sage: H.change_ring(GF(43)).odd_degree_model().frobenius_polynomial()       #␣
↪needs sage.rings.finite_rings
x^4 - 16*x^3 + 134*x^2 - 688*x + 1849

sage: # needs sage.rings.number_field
sage: P2 = K2.factor(67)[0][0]
sage: P3 = K3.factor(67)[0][0]
sage: Hp2.change_ring(K2.residue_field(P2)).frobenius_polynomial()
x^4 - 8*x^3 + 150*x^2 - 536*x + 4489
sage: Hp3.change_ring(K3.residue_field(P3)).frobenius_polynomial()
x^4 - 8*x^3 + 150*x^2 - 536*x + 4489

sage: H.change_ring(GF(67)).odd_degree_model().frobenius_polynomial()       #␣
↪needs sage.rings.finite_rings
x^4 - 8*x^3 + 150*x^2 - 536*x + 4489
```

rational_points(***kwds*)

Find rational points on the hyperelliptic curve, all arguments are passed on to `sage.schemes.generic.algebraic_scheme.rational_points()`.

EXAMPLES:

For the LMFDB genus 2 curve 932.a.3728.1:

```
sage: R.<x> = PolynomialRing(QQ)
sage: C = HyperellipticCurve(R([0, -1, 1, 0, 1, -2, 1]), R([1]))
sage: C.rational_points(bound=8)
[(-1 : -3 : 1),
(-1 : 2 : 1),
(0 : -1 : 1),
(0 : 0 : 1),
(0 : 1 : 0),
(1/2 : -5/8 : 1),
(1/2 : -3/8 : 1),
(1 : -1 : 1),
(1 : 0 : 1)]
```

Check that Issue #29509 is fixed for the LMFDB genus 2 curve 169.a.169.1:

```
sage: C = HyperellipticCurve(R([0, 0, 0, 0, 1, 1]), R([1, 1, 0, 1]))
sage: C.rational_points(bound=10)
[(-1 : 0 : 1),
(-1 : 1 : 1),
(0 : -1 : 1),
(0 : 0 : 1),
(0 : 1 : 0)]
```

An example over a number field:

```
sage: R.<x> = PolynomialRing(QuadraticField(2))                             #␣
↪needs sage.rings.number_field
sage: C = HyperellipticCurve(R([1, 0, 0, 0, 0, 1]))                         #␣
↪needs sage.rings.number_field
sage: C.rational_points(bound=2)                                            #␣
↪needs sage.rings.number_field
```

(continues on next page)

(continued from previous page)

```
[(-1 : 0 : 1),
 (0 : -1 : 1),
 (0 : 1 : 0),
 (0 : 1 : 1),
 (1 : -a : 1),
 (1 : a : 1)]
```

sage.schemes.hyperelliptic_curves.hyperelliptic_generic.**is_HyperellipticCurve**(*C*)

EXAMPLES:

```
sage: from sage.schemes.hyperelliptic_curves.hyperelliptic_generic import is_
↪HyperellipticCurve
sage: R.<x> = QQ[]; C = HyperellipticCurve(x^3 + x - 1); C
Hyperelliptic Curve over Rational Field defined by y^2 = x^3 + x - 1
sage: is_HyperellipticCurve(C)
doctest:warning...
DeprecationWarning: The function is_HyperellipticCurve is deprecated; use
↪'isinstance(..., HyperellipticCurve_generic)' instead.
See https://github.com/sagemath/sage/issues/38022 for details.
True
```

20.3 Hyperelliptic curves over a finite field

EXAMPLES:

```
sage: K.<a> = GF(9, 'a')
sage: x = polygen(K)
sage: C = HyperellipticCurve(x^7 - x^5 - 2, x^2 + a)
sage: C._points_fast_sqrt()
[(0 : 1 : 0), (a + 1 : a : 1), (a + 1 : a + 1 : 1), (2 : a + 1 : 1),
 (2*a : 2*a + 2 : 1), (2*a : 2*a : 1), (1 : a + 1 : 1)]
```

AUTHORS:

- David Kohel (2006)
- Robert Bradshaw (2007)
- Alyson Deines, Marina Gresham, Gagan Sekhon, (2010)
- Daniel Krenn (2011)
- Jean-Pierre Flori, Jan Tuitman (2013)
- Kiran Kedlaya (2016)
- Dean Bisogno (2017): Fixed Hasse-Witt computation

class sage.schemes.hyperelliptic_curves.hyperelliptic_finite_field.**HyperellipticCurve_finit**

Bases: *HyperellipticCurve_generic*, ProjectivePlaneCurve_finite_field

Cartier_matrix()

INPUT:

- E : Hyperelliptic Curve of the form $y^2 = f(x)$ over a finite field, $\mathbf{F}_q$

OUTPUT:

- M: The matrix $M = (c_{pi-j})$, where c_i are the coefficients of $f(x)^{(p-1)/2} = \sum c_i x^i$

REFERENCES:

N. Yui. On the Jacobian varieties of hyperelliptic curves over fields of characteristic $p > 2$.

EXAMPLES:

```
sage: K.<x> = GF(9,'x')[]
sage: C = HyperellipticCurve(x^7 - 1, 0)
sage: C.Cartier_matrix()
[0 0 2]
[0 0 0]
[0 1 0]

sage: K.<x> = GF(49, 'x')[]
sage: C = HyperellipticCurve(x^5 + 1, 0)
sage: C.Cartier_matrix()
[0 3]
[0 0]

sage: P.<x> = GF(9, 'a')[]
sage: E = HyperellipticCurve(x^29 + 1, 0)
sage: E.Cartier_matrix()
[0 0 1 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 1 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[1 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 1 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 1 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 1 0 0]
```

Hasse_Witt()

INPUT:

- E : Hyperelliptic Curve of the form $y^2 = f(x)$ over a finite field, $\mathbf{F}_q$

OUTPUT:

- N : The matrix $N = MM^p \dots M^{p^{g-1}}$ where $M = c_{pi-j}$, and $f(x)^{(p-1)/2} = \sum c_i x^i$

Reference-N. Yui. On the Jacobian varieties of hyperelliptic curves over fields of characteristic $p > 2$.

EXAMPLES:

```
sage: K.<x> = GF(9, 'x')[]
sage: C = HyperellipticCurve(x^7 - 1, 0)
```

(continues on next page)

(continued from previous page)

```
sage: C.Hasse_Witt()
[0 0 0]
[0 0 0]
[0 0 0]

sage: K.<x> = GF(49, 'x')[]
sage: C = HyperellipticCurve(x^5 + 1, 0)
sage: C.Hasse_Witt()
[0 0]
[0 0]

sage: P.<x> = GF(9, 'a')[]
sage: E = HyperellipticCurve(x^29 + 1, 0)
sage: E.Hasse_Witt()
[0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 0 0 0 0 0]
```

a_number()

INPUT:

- `E`: Hyperelliptic Curve of the form $y^2 = f(x)$ over a finite field, $\mathbf{F}_q$

OUTPUT:

- `a` : a-number

EXAMPLES:

```
sage: K.<x> = GF(49, 'x')[]
sage: C = HyperellipticCurve(x^5 + 1, 0)
sage: C.a_number()
1

sage: K.<x> = GF(9, 'x')[]
sage: C = HyperellipticCurve(x^7 - 1, 0)
sage: C.a_number()
1

sage: P.<x> = GF(9, 'a')[]
sage: E = HyperellipticCurve(x^29 + 1, 0)
sage: E.a_number()
5
```

cardinality(*extension_degree=1*)

Count points on a single extension of the base field.

EXAMPLES:

```
sage: K = GF(101)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^9 + 3*t^5 + 5)
sage: H.cardinality()
106
sage: H.cardinality(15)
116096895536999256707640583100 0
sage: H.cardinality(100)
270481382942152609326719471080753083367793838278100277689020104911710151430673927943945601 43

sage: K = GF(37)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^9 + 3*t^5 + 5)
sage: H.cardinality()
40
sage: H.cardinality(2)
1408
sage: H.cardinality(3)
50116
```

The following example shows that Issue #20391 has been resolved:

```
sage: F=GF(23)
sage: x=polygen(F)
sage: C=HyperellipticCurve(x^8+1)
sage: C.cardinality()
24
```

cardinality_exhaustive(*extension_degree=1*, *algorithm=None*)

Count points on a single extension of the base field by enumerating over x and solving the resulting quadratic equation for y.

EXAMPLES:

```
sage: K.<a> = GF(9, 'a')
sage: x = polygen(K)
sage: C = HyperellipticCurve(x^7 - 1, x^2 + a)
sage: C.cardinality_exhaustive()
7

sage: K = GF(next_prime(1<<10))
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^7 + 3*t^5 + 5)
sage: H.cardinality_exhaustive()
1025

sage: P.<x> = PolynomialRing(GF(9,'a'))
sage: H = HyperellipticCurve(x^5+x^2+1)
sage: H.count_points(5)
[18, 78, 738, 6366, 60018]

sage: F.<a> = GF(4); P.<x> = F[]
sage: H = HyperellipticCurve(x^5+a*x^2+1, x+a+1)
sage: H.count_points(6)
[2, 24, 74, 256, 1082, 4272]
```

cardinality_hypellfrob(*extension_degree=1*, *algorithm=None*)

Count points on a single extension of the base field using the `hypellfrob` program.

EXAMPLES:

```
sage: K = GF(next_prime(1<<10))
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^7 + 3*t^5 + 5)
sage: H.cardinality_hypellfrob()
1025

sage: K = GF(49999)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^7 + 3*t^5 + 5)
sage: H.cardinality_hypellfrob()
50162
sage: H.cardinality_hypellfrob(3)
124992471088310
```

count_points(*n=1*)

Count points over finite fields.

INPUT:

- n – integer.

OUTPUT:

An integer. The number of points over $\mathbf{F}_q, \ldots, \mathbf{F}_{q^n}$ on a hyperelliptic curve over a finite field $\mathbf{F}_q$.

Warning: This is currently using exhaustive search for hyperelliptic curves over non-prime fields, which can be awfully slow.

EXAMPLES:

```
sage: P.<x> = PolynomialRing(GF(3))
sage: C = HyperellipticCurve(x^3+x^2+1)
sage: C.count_points(4)
[6, 12, 18, 96]
sage: C.base_extend(GF(9,'a')).count_points(2)
[12, 96]

sage: K = GF(2**31-1)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^5 + 3*t + 5)
sage: H.count_points() # long time, 2.4 sec on a Corei7
[2147464821]
sage: H.count_points(n=2) # long time, 30s on a Corei7
[2147464821, 4611686018988310237]

sage: K = GF(2**7-1)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^13 + 3*t^5 + 5)
sage: H.count_points(n=6)
[112, 16360, 2045356, 260199160, 33038302802, 4195868633548]

sage: P.<x> = PolynomialRing(GF(3))
sage: H = HyperellipticCurve(x^3+x^2+1)
sage: C1 = H.count_points(4); C1
[6, 12, 18, 96]
```

(continues on next page)

(continued from previous page)

```
sage: C2 = sage.schemes.generic.scheme.Scheme.count_points(H,4); C2 # long␣
↪time, 2s on a Corei7
[6, 12, 18, 96]
sage: C1 == C2 # long time, because we need C2 to be defined
True

sage: P.<x> = PolynomialRing(GF(9,'a'))
sage: H = HyperellipticCurve(x^5+x^2+1)
sage: H.count_points(5)
[18, 78, 738, 6366, 60018]

sage: F.<a> = GF(4); P.<x> = F[]
sage: H = HyperellipticCurve(x^5+a*x^2+1, x+a+1)
sage: H.count_points(6)
[2, 24, 74, 256, 1082, 4272]
```

This example shows that Issue #20391 is resolved:

```
sage: x = polygen(GF(4099))
sage: H = HyperellipticCurve(x^6 + x + 1)
sage: H.count_points(1)
[4106]
```

count_points_exhaustive (*n=1*, *naive=False*)

Count the number of points on the curve over the first n extensions of the base field by exhaustive search if n if smaller than g, the genus of the curve, and by computing the frobenius polynomial after performing exhaustive search on the first g extensions if $n > g$ (unless `naive == True`).

EXAMPLES:

```
sage: K = GF(5)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^9 + t^3 + 1)
sage: H.count_points_exhaustive(n=5)
[9, 27, 108, 675, 3069]
```

When $n > g$, the frobenius polynomial is computed from the numbers of points of the curve over the first g extension, so that computing the number of points on extensions of degree $n > g$ is not much more expensive than for $n == g$:

```
sage: H.count_points_exhaustive(n=15)
[9,
27,
108,
675,
3069,
16302,
78633,
389475,
1954044,
9768627,
48814533,
244072650,
1220693769,
6103414827,
30517927308]
```

This behavior can be disabled by passing `naive=True`:

```
sage: H.count_points_exhaustive(n=6, naive=True) # long time, 7s on a Corei7
[9, 27, 108, 675, 3069, 16302]
```

count_points_frobenius_polynomial (*n=1*, *f=None*)

Count the number of points on the curve over the first n extensions of the base field by computing the frobenius polynomial.

EXAMPLES:

```
sage: K = GF(49999)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^19 + t + 1)
```

The following computation takes a long time as the complete characteristic polynomial of the frobenius is computed:

```
sage: H.count_points_frobenius_polynomial(3) # long time, 20s on a Corei7␣
↪(when computed before the following test of course)
[49491, 2500024375, 124992509154249]
```

As the polynomial is cached, further computations of number of points are really fast:

```
sage: H.count_points_frobenius_polynomial(19) # long time, because of the␣
↪previous test
[49491,
2500024375,
124992509154249,
6249500007135192947,
312468751250758776051811,
15623125093747382662737313867,
781140631562281338861289572576257,
39056250437482500417107992413002794587,
1952773465623687539373429411200893147181079,
97636720507718753281169963459063147221761552935,
4881738388665429945305281187129778704058864736771824,
244082037694882831835318764490138139735446240036293092851,
12203857802706446708934102903106811520015567632046432103159713,
610180686277519628999996211052002771035439565767719719151141201339,
30508424133189703930370810556389262704405225546438978173388673620145499,
1525390698235352006814610157008906752699329454643826047826098161898351623931,
76268009521069364988723693240288328729528917832735078791261015331201838856825193,
↪
381332420804394718007119593832117614814724412806217255555871578364900658786872993991,
↪
190662397077989315056379725720120486231213267083935859751911720230901597698389839098903847]
```

count_points_hypellfrob (*n=1*, *N=None*, *algorithm=None*)

Count the number of points on the curve over the first n extensions of the base field using the `hypellfrob` program.

This only supports prime fields of large enough characteristic.

EXAMPLES:

```
sage: K = GF(49999)
sage: R.<t> = PolynomialRing(K)
```

(continues on next page)

(continued from previous page)

```
sage: H = HyperellipticCurve(t^21 + 3*t^5 + 5)
sage: H.count_points_hypellfrob()
[49804]
sage: H.count_points_hypellfrob(2)
[49804, 2499799038]

sage: K = GF(2**7-1)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^11 + 3*t^5 + 5)
sage: H.count_points_hypellfrob()
[127]
sage: H.count_points_hypellfrob(n=5)
[127, 16335, 2045701, 260134299, 33038098487]

sage: K = GF(2**7-1)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^13 + 3*t^5 + 5)
sage: H.count_points(n=6)
[112, 16360, 2045356, 260199160, 33038302802, 4195868633548]
```

The base field should be prime:

```
sage: K.<z> = GF(19**10)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^9 + (z+1)*t^5 + 1)
sage: H.count_points_hypellfrob()
Traceback (most recent call last):
...
ValueError: hypellfrob does not support non-prime fields
```

and the characteristic should be large enough:

```
sage: K = GF(7)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^9 + t^3 + 1)
sage: H.count_points_hypellfrob()
Traceback (most recent call last):
...
ValueError: p=7 should be greater than (2*g+1)(2*N-1)=27
```

count_points_matrix_traces(*n=1*, *M=None*, *N=None*)

Count the number of points on the curve over the first n extensions of the base field by computing traces of powers of the frobenius matrix. This requires less p-adic precision than computing the charpoly of the matrix when $n < g$ where g is the genus of the curve.

EXAMPLES:

```
sage: K = GF(49999)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^19 + t + 1)
sage: H.count_points_matrix_traces(3)
[49491, 2500024375, 124992509154249]
```

frobenius_matrix(*N=None*, *algorithm='hypellfrob'*)

Compute p-adic frobenius matrix to precision p^N. If N not supplied, a default value is selected, which is the minimum needed to recover the charpoly unambiguously.

Note: Currently only implemented using `hypellfrob`, which means it only works over the prime field $GF(p)$, and requires $p > (2g+1)(2N-1)$.

EXAMPLES:

```
sage: R.<t> = PolynomialRing(GF(37))
sage: H = HyperellipticCurve(t^5 + t + 2)
sage: H.frobenius_matrix()
[1258 + O(37^2)  925 + O(37^2)  132 + O(37^2)  587 + O(37^2)]
[1147 + O(37^2)  814 + O(37^2)  241 + O(37^2) 1011 + O(37^2)]
[1258 + O(37^2) 1184 + O(37^2) 1105 + O(37^2)  482 + O(37^2)]
[1073 + O(37^2)  999 + O(37^2)  772 + O(37^2)  929 + O(37^2)]
```

The `hypellfrob` program doesn't support non-prime fields:

```
sage: K.<z> = GF(37**3)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^9 + z*t^3 + 1)
sage: H.frobenius_matrix(algorithm='hypellfrob')
Traceback (most recent call last):
...
NotImplementedError: Computation of Frobenius matrix only implemented
for hyperelliptic curves defined over prime fields.
```

nor too small characteristic:

```
sage: K = GF(7)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^9 + t^3 + 1)
sage: H.frobenius_matrix(algorithm='hypellfrob')
Traceback (most recent call last):
...
ValueError: In the current implementation, p must be greater than (2g+1)(2N-
↪1) = 81
```

frobenius_matrix_hypellfrob(*N=None*)

Compute p-adic frobenius matrix to precision p^N. If N not supplied, a default value is selected, which is the minimum needed to recover the charpoly unambiguously.

Note: Implemented using `hypellfrob`, which means it only works over the prime field $GF(p)$, and requires $p > (2g+1)(2N-1)$.

EXAMPLES:

```
sage: R.<t> = PolynomialRing(GF(37))
sage: H = HyperellipticCurve(t^5 + t + 2)
sage: H.frobenius_matrix_hypellfrob()
[1258 + O(37^2)  925 + O(37^2)  132 + O(37^2)  587 + O(37^2)]
[1147 + O(37^2)  814 + O(37^2)  241 + O(37^2) 1011 + O(37^2)]
[1258 + O(37^2) 1184 + O(37^2) 1105 + O(37^2)  482 + O(37^2)]
[1073 + O(37^2)  999 + O(37^2)  772 + O(37^2)  929 + O(37^2)]
```

The `hypellfrob` program doesn't support non-prime fields:

```
sage: K.<z> = GF(37**3)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^9 + z*t^3 + 1)
sage: H.frobenius_matrix_hypellfrob()
Traceback (most recent call last):
...
NotImplementedError: Computation of Frobenius matrix only implemented
for hyperelliptic curves defined over prime fields.
```

nor too small characteristic:

```
sage: K = GF(7)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^9 + t^3 + 1)
sage: H.frobenius_matrix_hypellfrob()
Traceback (most recent call last):
...
ValueError: In the current implementation, p must be greater than (2g+1)(2N-
↪1) = 81
```

frobenius_polynomial()

Compute the charpoly of frobenius, as an element of $\mathbf{Z}[x]$.

EXAMPLES:

```
sage: R.<t> = PolynomialRing(GF(37))
sage: H = HyperellipticCurve(t^5 + t + 2)
sage: H.frobenius_polynomial()
x^4 + x^3 - 52*x^2 + 37*x + 1369
```

A quadratic twist:

```
sage: H = HyperellipticCurve(2*t^5 + 2*t + 4)
sage: H.frobenius_polynomial()
x^4 - x^3 - 52*x^2 - 37*x + 1369
```

Slightly larger example:

```
sage: K = GF(2003)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^7 + 487*t^5 + 9*t + 1)
sage: H.frobenius_polynomial()
x^6 - 14*x^5 + 1512*x^4 - 66290*x^3 + 3028536*x^2 - 56168126*x + 8036054027
```

Curves defined over a non-prime field of odd characteristic, or an odd prime field which is too small compared to the genus, are supported via PARI:

```
sage: K.<z> = GF(23**3)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^3 + z*t + 4)
sage: H.frobenius_polynomial()
x^2 - 15*x + 12167

sage: K.<z> = GF(3**3)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^5 + z*t + z**3)
sage: H.frobenius_polynomial()
x^4 - 3*x^3 + 10*x^2 - 81*x + 729
```

Over prime fields of odd characteristic, h may be non-zero:

```
sage: K = GF(101)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^5 + 27*t + 3, t)
sage: H.frobenius_polynomial()
x^4 + 2*x^3 - 58*x^2 + 202*x + 10201
```

Over prime fields of odd characteristic, f may have even degree:

```
sage: H = HyperellipticCurve(t^6 + 27*t + 3)
sage: H.frobenius_polynomial()
x^4 + 25*x^3 + 322*x^2 + 2525*x + 10201
```

In even characteristic, the naive algorithm could cover all cases because we can easily check for squareness in quotient rings of polynomial rings over finite fields but these rings unfortunately do not support iteration:

```
sage: K.<z> = GF(2**5)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^5 + z*t + z**3, t)
sage: H.frobenius_polynomial()
x^4 - x^3 + 16*x^2 - 32*x + 1024
```

frobenius_polynomial_cardinalities(*a=None*)

Compute the charpoly of frobenius, as an element of $\mathbf{Z}[x]$, by computing the number of points on the curve over g extensions of the base field where g is the genus of the curve.

Warning: This is highly inefficient when the base field or the genus of the curve are large.

EXAMPLES:

```
sage: R.<t> = PolynomialRing(GF(37))
sage: H = HyperellipticCurve(t^5 + t + 2)
sage: H.frobenius_polynomial_cardinalities()
x^4 + x^3 - 52*x^2 + 37*x + 1369
```

A quadratic twist:

```
sage: H = HyperellipticCurve(2*t^5 + 2*t + 4)
sage: H.frobenius_polynomial_cardinalities()
x^4 - x^3 - 52*x^2 - 37*x + 1369
```

Curve over a non-prime field:

```
sage: K.<z> = GF(7**2)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^5 + z*t + z^2)
sage: H.frobenius_polynomial_cardinalities()
x^4 + 8*x^3 + 70*x^2 + 392*x + 2401
```

This method may actually be useful when `hypellfrob` does not work:

```
sage: K = GF(7)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^9 + t^3 + 1)
```

(continues on next page)

(continued from previous page)

```
sage: H.frobenius_polynomial_matrix(algorithm='hypellfrob')
Traceback (most recent call last):
...
ValueError: In the current implementation, p must be greater than (2g+1)(2N-
↪1) = 81
sage: H.frobenius_polynomial_cardinalities()
x^8 - 5*x^7 + 7*x^6 + 36*x^5 - 180*x^4 + 252*x^3 + 343*x^2 - 1715*x + 2401
```

frobenius_polynomial_matrix(*M=None*, *algorithm='hypellfrob'*)

Compute the charpoly of frobenius, as an element of $\mathbf{Z}[x]$, by computing the charpoly of the frobenius matrix.

This is currently only supported when the base field is prime and large enough using the `hypellfrob` library.

EXAMPLES:

```
sage: R.<t> = PolynomialRing(GF(37))
sage: H = HyperellipticCurve(t^5 + t + 2)
sage: H.frobenius_polynomial_matrix()
x^4 + x^3 - 52*x^2 + 37*x + 1369
```

A quadratic twist:

```
sage: H = HyperellipticCurve(2*t^5 + 2*t + 4)
sage: H.frobenius_polynomial_matrix()
x^4 - x^3 - 52*x^2 - 37*x + 1369
```

Curves defined over larger prime fields:

```
sage: K = GF(49999)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^9 + t^5 + 1)
sage: H.frobenius_polynomial_matrix()
x^8 + 281*x^7 + 55939*x^6 + 14144175*x^5 + 3156455369*x^4 + 707194605825*x^3
+ 139841906155939*x^2 + 35122892542149719*x + 6249500014999800001
sage: H = HyperellipticCurve(t^15 + t^5 + 1)
sage: H.frobenius_polynomial_matrix()  # long time, 8s on a Corei7
x^14 - 76*x^13 + 220846*x^12 - 12984372*x^11 + 24374326657*x^10 -␣
↪1203243210304*x^9
+ 1770558798515792*x^8 - 7440151141521049 6*x^7 + 8852616936699108 4208*x^6
- 3007987702642212810304*x^5 + 3046608028331197124223343*x^4
- 8114583300876298313858 4372*x^3 + 6900747383855197890521127 9154*x^2
- 1187357507124810002849977200076*x + 7811406315622812543749475003 49999
```

frobenius_polynomial_pari()

Compute the charpoly of frobenius, as an element of $\mathbf{Z}[x]$, by calling the PARI function `hyperellcharpoly`.

EXAMPLES:

```
sage: R.<t> = PolynomialRing(GF(37))
sage: H = HyperellipticCurve(t^5 + t + 2)
sage: H.frobenius_polynomial_pari()
x^4 + x^3 - 52*x^2 + 37*x + 1369
```

A quadratic twist:

```
sage: H = HyperellipticCurve(2*t^5 + 2*t + 4)
sage: H.frobenius_polynomial_pari()
x^4 - x^3 - 52*x^2 - 37*x + 1369
```

Slightly larger example:

```
sage: K = GF(2003)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^7 + 487*t^5 + 9*t + 1)
sage: H.frobenius_polynomial_pari()
x^6 - 14*x^5 + 1512*x^4 - 66290*x^3 + 3028536*x^2 - 56168126*x + 8036054027
```

Curves defined over a non-prime field are supported as well:

```
sage: K.<a> = GF(7^2)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^5 + a*t + 1)
sage: H.frobenius_polynomial_pari()
x^4 + 4*x^3 + 84*x^2 + 196*x + 2401

sage: K.<z> = GF(23**3)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^3 + z*t + 4)
sage: H.frobenius_polynomial_pari()
x^2 - 15*x + 12167
```

Over prime fields of odd characteristic, h may be non-zero:

```
sage: K = GF(101)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^5 + 27*t + 3, t)
sage: H.frobenius_polynomial_pari()
x^4 + 2*x^3 - 58*x^2 + 202*x + 10201
```

p_rank()

INPUT:

- `E` : Hyperelliptic Curve of the form $y^2 = f(x)$ over a finite field, $\mathbf{F}_q$

OUTPUT:

- `pr` :p-rank

EXAMPLES:

```
sage: K.<x> = GF(49, 'x')[]
sage: C = HyperellipticCurve(x^5 + 1, 0)
```

(continues on next page)

(continued from previous page)

```
sage: C.p_rank()
0

sage: K.<x> = GF(9, 'x')[]
sage: C = HyperellipticCurve(x^7 - 1, 0)
sage: C.p_rank()
0

sage: P.<x> = GF(9, 'a')[]
sage: E = HyperellipticCurve(x^29 + 1, 0)
sage: E.p_rank()
0
```

points()

All the points on this hyperelliptic curve.

EXAMPLES:

```
sage: x = polygen(GF(7))
sage: C = HyperellipticCurve(x^7 - x^2 - 1)
sage: C.points()
[(0 : 1 : 0), (2 : 5 : 1), (2 : 2 : 1), (3 : 0 : 1), (4 : 6 : 1),
 (4 : 1 : 1), (5 : 0 : 1), (6 : 5 : 1), (6 : 2 : 1)]
```

```
sage: x = polygen(GF(121, 'a'))
sage: C = HyperellipticCurve(x^5 + x - 1, x^2 + 2)
sage: len(C.points())
122
```

Conics are allowed (the issue reported at Issue #11800 has been resolved):

```
sage: R.<x> = GF(7)[]
sage: H = HyperellipticCurve(3*x^2 + 5*x + 1)
sage: H.points()
[(0 : 6 : 1), (0 : 1 : 1), (1 : 4 : 1), (1 : 3 : 1), (2 : 4 : 1),
 (2 : 3 : 1), (3 : 6 : 1), (3 : 1 : 1)]
```

The method currently lists points on the plane projective model, that is the closure in $\mathbb{P}^2$ of the curve defined by $y^2 + hy = f$. This means that one point $(0 : 1 : 0)$ at infinity is returned if the degree of the curve is at least 4 and $\deg(f) > \deg(h) + 1$. This point is a singular point of the plane model. Later implementations may consider a smooth model instead since that would be a more relevant object. Then, for a curve whose only singularity is at $(0 : 1 : 0)$, the point at infinity would be replaced by a number of rational points of the smooth model. We illustrate this with an example of a genus 2 hyperelliptic curve:

```
sage: R.<x>=GF(11)[]
sage: H = HyperellipticCurve(x*(x+1)*(x+2)*(x+3)*(x+4)*(x+5))
sage: H.points()
[(0 : 1 : 0), (0 : 0 : 1), (1 : 7 : 1), (1 : 4 : 1), (5 : 7 : 1), (5 : 4 : 1),
 (6 : 0 : 1), (7 : 0 : 1), (8 : 0 : 1), (9 : 0 : 1), (10 : 0 : 1)]
```

The plane model of the genus 2 hyperelliptic curve in the above example is the curve in $\mathbb{P}^2$ defined by $y^2z^4 = g(x, z)$ where $g(x, z) = x(x + z)(x + 2z)(x + 3z)(x + 4z)(x + 5z)$. This model has one point at infinity $(0 : 1 : 0)$ which is also the only singular point of the plane model. In contrast, the hyperelliptic curve is smooth and imbeds via the equation $y^2 = g(x, z)$ into weighted projected space $\mathbb{P}(1, 3, 1)$. The latter model has two points at infinity: $(1 : 1 : 0)$ and $(1 : -1 : 0)$.

zeta_function()

Compute the zeta function of the hyperelliptic curve.

EXAMPLES:

```
sage: F = GF(2); R.<t> = F[]
sage: H = HyperellipticCurve(t^9 + t, t^4)
sage: H.zeta_function()
(16*x^8 + 8*x^7 + 8*x^6 + 4*x^5 + 6*x^4 + 2*x^3 + 2*x^2 + x + 1)/(2*x^2 - 3*x␣
↪+ 1)

sage: F.<a> = GF(4); R.<t> = F[]
sage: H = HyperellipticCurve(t^5 + t^3 + t^2 + t + 1, t^2 + t + 1)
sage: H.zeta_function()
(16*x^4 + 8*x^3 + x^2 + 2*x + 1)/(4*x^2 - 5*x + 1)

sage: F.<a> = GF(9); R.<t> = F[]
sage: H = HyperellipticCurve(t^5 + a*t)
sage: H.zeta_function()
(81*x^4 + 72*x^3 + 32*x^2 + 8*x + 1)/(9*x^2 - 10*x + 1)

sage: R.<t> = PolynomialRing(GF(37))
sage: H = HyperellipticCurve(t^5 + t + 2)
sage: H.zeta_function()
(1369*x^4 + 37*x^3 - 52*x^2 + x + 1)/(37*x^2 - 38*x + 1)
```

A quadratic twist:

```
sage: R.<t> = PolynomialRing(GF(37))
sage: H = HyperellipticCurve(2*t^5 + 2*t + 4)
sage: H.zeta_function()
(1369*x^4 - 37*x^3 - 52*x^2 - x + 1)/(37*x^2 - 38*x + 1)
```

20.4 Hyperelliptic curves over a p-adic field

class sage.schemes.hyperelliptic_curves.hyperelliptic_padic_field.**HyperellipticCurve_padic_**

Bases: *HyperellipticCurve_generic*, ProjectivePlaneCurve_field

P_to_S(P, S)

Given a finite Weierstrass point P and a point S in the same disc, computes the Coleman integrals $\{\int_P^S x^i dx/2y\}_{i=0}^{2g-1}$

INPUT:

- P: finite Weierstrass point
- S: point in disc of P

OUTPUT:

Coleman integrals $\{\int_P^S x^i dx/2y\}_{i=0}^{2g-1}$

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: H = HyperellipticCurve(x^3-10*x+9)
sage: K = Qp(5,4)
sage: HK = H.change_ring(K)
sage: P = HK(1,0)
sage: HJ = HK.curve_over_ram_extn(10)
sage: S = HK.get_boundary_point(HJ,P)
sage: HK.P_to_S(P, S)
(2*a + 4*a^3 + 2*a^11 + 4*a^13 + 2*a^17 + 2*a^19 + a^21 + 4*a^23 + a^25 + 2*a^
↪27 + 2*a^29 + 3*a^31 + 4*a^33 + O(a^35), a^-5 + 2*a + 2*a^3 + a^7 + 3*a^11␣
↪+ a^13 + 3*a^15 + 3*a^17 + 2*a^19 + 4*a^21 + 4*a^23 + 4*a^25 + 2*a^27 + a^
↪29 + a^31 + O(a^33))
```

AUTHOR:

- Jennifer Balakrishnan

S_to_Q(*S*, *Q*)

Given S a point on self over an extension field, computes the Coleman integrals $\{\int_S^Q x^i dx/2y\}_{i=0}^{2g-1}$

one should be able to feed `S,Q` into coleman_integral, but currently that segfaults

INPUT:

- S: a point with coordinates in an extension of $\mathbf{Q}_p$ (with unif. a)
- Q: a non-Weierstrass point defined over $\mathbf{Q}_p$

OUTPUT:

the Coleman integrals $\{\int_S^Q x^i dx/2y\}_{i=0}^{2g-1}$ in terms of a

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: H = HyperellipticCurve(x^3-10*x+9)
sage: K = Qp(5,6)
sage: HK = H.change_ring(K)
sage: J.<a> = K.extension(x^20-5)
sage: HJ  = H.change_ring(J)
sage: w = HK.invariant_differential()
sage: x,y = HK.monsky_washnitzer_gens()
sage: P = HK(1,0)
sage: Q = HK(0,3)
sage: S = HK.get_boundary_point(HJ,P)
sage: P_to_S = HK.P_to_S(P,S)
sage: S_to_Q = HJ.S_to_Q(S,Q)
sage: P_to_S + S_to_Q
(2*a^40 + a^80 + a^100 + O(a^105), a^20 + 2*a^40 + 4*a^60 + 2*a^80 + O(a^103))
sage: HK.coleman_integrals_on_basis(P,Q)
(2*5^2 + 5^4 + 5^5 + 3*5^6 + O(5^7), 5 + 2*5^2 + 4*5^3 + 2*5^4 + 5^6 + O(5^7))
```

AUTHOR:

- Jennifer Balakrishnan

coleman_integral(*w*, *P*, *Q*, *algorithm='None'*)

Return the Coleman integral $\int_P^Q w$.

INPUT:

- w differential (if one of P,Q is Weierstrass, w must be odd)

- P point on self
- Q point on self
- algorithm (optional) = None (uses Frobenius) or teichmuller (uses Teichmuller points)

OUTPUT:

the Coleman integral $\int_P^Q w$

EXAMPLES:

Example of Leprevost from Kiran Kedlaya The first two should be zero as $(P - Q) = 30(P - Q)$ in the Jacobian and $dx/2y$ and $xdx/2y$ are holomorphic.

```
sage: K = pAdicField(11, 6)
sage: x = polygen(K)
sage: C = HyperellipticCurve(x^5 + 33/16*x^4 + 3/4*x^3 + 3/8*x^2 - 1/4*x + 1/
↪16)
sage: P = C(-1, 1); P1 = C(-1, -1)
sage: Q = C(0, 1/4); Q1 = C(0, -1/4)
sage: x, y = C.monsky_washnitzer_gens()
sage: w = C.invariant_differential()
sage: w.coleman_integral(P, Q)
O(11^6)
sage: C.coleman_integral(x*w, P, Q)
O(11^6)
sage: C.coleman_integral(x^2*w, P, Q)
7*11 + 6*11^2 + 3*11^3 + 11^4 + 5*11^5 + O(11^6)
```

```
sage: p = 71; m = 4
sage: K = pAdicField(p, m)
sage: x = polygen(K)
sage: C = HyperellipticCurve(x^5 + 33/16*x^4 + 3/4*x^3 + 3/8*x^2 - 1/4*x + 1/
↪16)
sage: P = C(-1, 1); P1 = C(-1, -1)
sage: Q = C(0, 1/4); Q1 = C(0, -1/4)
sage: x, y = C.monsky_washnitzer_gens()
sage: w = C.invariant_differential()
sage: w.integrate(P, Q), (x*w).integrate(P, Q)
(O(71^4), O(71^4))
sage: R, R1 = C.lift_x(4, all=True)
sage: w.integrate(P, R)
50*71 + 3*71^2 + 43*71^3 + O(71^4)
sage: w.integrate(P, R) + w.integrate(P1, R1)
O(71^4)
```

A simple example, integrating dx:

```
sage: R.<x> = QQ['x']
sage: E= HyperellipticCurve(x^3-4*x+4)
sage: K = Qp(5,10)
sage: EK = E.change_ring(K)
sage: P = EK(2, 2)
sage: Q = EK.teichmuller(P)
sage: x, y = EK.monsky_washnitzer_gens()
sage: EK.coleman_integral(x.diff(), P, Q)
5 + 2*5^2 + 5^3 + 3*5^4 + 4*5^5 + 2*5^6 + 3*5^7 + 3*5^9 + O(5^10)
sage: Q[0] - P[0]
5 + 2*5^2 + 5^3 + 3*5^4 + 4*5^5 + 2*5^6 + 3*5^7 + 3*5^9 + O(5^10)
```

Yet another example:

```
sage: R.<x> = QQ['x']
sage: H = HyperellipticCurve(x*(x-1)*(x+9))
sage: K = Qp(7,10)
sage: HK = H.change_ring(K)
sage: import sage.schemes.hyperelliptic_curves.monsky_washnitzer as mw
sage: M_frob, forms = mw.matrix_of_frobenius_hyperelliptic(HK)
sage: w = HK.invariant_differential()
sage: x,y = HK.monsky_washnitzer_gens()
sage: f = forms[0]
sage: S = HK(9,36)
sage: Q = HK.teichmuller(S)
sage: P = HK(-1,4)
sage: b = x*w*w._coeff.parent()(f)
sage: HK.coleman_integral(b,P,Q)
7 + 7^2 + 4*7^3 + 5*7^4 + 3*7^5 + 7^6 + 5*7^7 + 3*7^8 + 4*7^9 + 4*7^10 + O(7^
↪11)
```

```
sage: R.<x> = QQ['x']
sage: H = HyperellipticCurve(x^3+1)
sage: K = Qp(5,8)
sage: HK = H.change_ring(K)
sage: w = HK.invariant_differential()
sage: P = HK(0,1)
sage: Q = HK(5, 1 + 3*5^3 + 2*5^4 + 2*5^5 + 3*5^7)
sage: x,y = HK.monsky_washnitzer_gens()
sage: (2*y*w).coleman_integral(P,Q)
5 + O(5^9)
sage: xloc,yloc,zloc = HK.local_analytic_interpolation(P,Q)
sage: I2 = (xloc.derivative()/(2*yloc)).integral()
sage: I2.polynomial()(1) - I2(0)
3*5 + 2*5^2 + 2*5^3 + 5^4 + 4*5^6 + 5^7 + O(5^9)
sage: HK.coleman_integral(w,P,Q)
3*5 + 2*5^2 + 2*5^3 + 5^4 + 4*5^6 + 5^7 + O(5^9)
```

Integrals involving Weierstrass points:

```
sage: R.<x> = QQ['x']
sage: H = HyperellipticCurve(x^3-10*x+9)
sage: K = Qp(5,8)
sage: HK = H.change_ring(K)
sage: S = HK(1,0)
sage: P = HK(0,3)
sage: negP = HK(0,-3)
sage: T = HK(0,1,0)
sage: w = HK.invariant_differential()
sage: x,y = HK.monsky_washnitzer_gens()
sage: HK.coleman_integral(w*x^3,S,T)
0
sage: HK.coleman_integral(w*x^3,T,S)
0
sage: HK.coleman_integral(w,S,P)
2*5^2 + 5^4 + 5^5 + 3*5^6 + 3*5^7 + 2*5^8 + O(5^9)
sage: HK.coleman_integral(w,T,P)
2*5^2 + 5^4 + 5^5 + 3*5^6 + 3*5^7 + 2*5^8 + O(5^9)
sage: HK.coleman_integral(w*x^3,T,P)
5^2 + 2*5^3 + 3*5^6 + 3*5^7 + O(5^8)
```

(continues on next page)

(continued from previous page)

```
sage: HK.coleman_integral(w*x^3,S,P)
5^2 + 2*5^3 + 3*5^6 + 3*5^7 + O(5^8)
sage: HK.coleman_integral(w, P, negP, algorithm='teichmuller')
5^2 + 4*5^3 + 2*5^4 + 2*5^5 + 3*5^6 + 2*5^7 + 4*5^8 + O(5^9)
sage: HK.coleman_integral(w, P, negP)
5^2 + 4*5^3 + 2*5^4 + 2*5^5 + 3*5^6 + 2*5^7 + 4*5^8 + O(5^9)
```

AUTHORS:

- Robert Bradshaw (2007-03)
- Kiran Kedlaya (2008-05)
- Jennifer Balakrishnan (2010-02)

coleman_integral_P_to_S(w, P, S)

Given a finite Weierstrass point P and a point S in the same disc, computes the Coleman integral $\int_P^S w$

INPUT:

- w: differential
- P: Weierstrass point
- S: point in the same disc of P (S is defined over an extension of $\mathbf{Q}_p$; coordinates of S are given in terms of uniformizer a)

OUTPUT:

Coleman integral $\int_P^S w$ in terms of a

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: H = HyperellipticCurve(x^3-10*x+9)
sage: K = Qp(5,4)
sage: HK = H.change_ring(K)
sage: P = HK(1,0)
sage: J.<a> = K.extension(x^10-5)
sage: HJ  = H.change_ring(J)
sage: S = HK.get_boundary_point(HJ,P)
sage: x,y = HK.monsky_washnitzer_gens()
sage: S[0]-P[0] == HK.coleman_integral_P_to_S(x.diff(),P,S)
True
sage: HK.coleman_integral_P_to_S(HK.invariant_differential(),P,S) == HK.P_to_
↪S(P,S)[0]
True
```

AUTHOR:

- Jennifer Balakrishnan

coleman_integral_S_to_Q(w, S, Q)

Compute the Coleman integral $\int_S^Q w$

one should be able to feed `S,Q` into coleman_integral, but currently that segfaults

INPUT:

- w: a differential
- S: a point with coordinates in an extension of $\mathbf{Q}_p$

- Q: a non-Weierstrass point defined over $\mathbf{Q}_p$

OUTPUT:

the Coleman integral $\int_S^Q w$

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: H = HyperellipticCurve(x^3-10*x+9)
sage: K = Qp(5,6)
sage: HK = H.change_ring(K)
sage: J.<a> = K.extension(x^20-5)
sage: HJ  = H.change_ring(J)
sage: x,y = HK.monsky_washnitzer_gens()
sage: P = HK(1,0)
sage: Q = HK(0,3)
sage: S = HK.get_boundary_point(HJ,P)
sage: P_to_S = HK.coleman_integral_P_to_S(y.diff(),P,S)
sage: S_to_Q = HJ.coleman_integral_S_to_Q(y.diff(),S,Q)
sage: P_to_S  + S_to_Q
3 + O(a^119)
sage: HK.coleman_integral(y.diff(),P,Q)
3 + O(5^6)
```

AUTHOR:

- Jennifer Balakrishnan

coleman_integral_from_weierstrass_via_boundary(*w*, *P*, *Q*, *d*)

Computes the Coleman integral $\int_P^Q w$ via a boundary point in the disc of P, defined over a degree d extension

INPUT:

- w: a differential
- P: a Weierstrass point
- Q: a non-Weierstrass point
- d: degree of extension where coordinates of boundary point lie

OUTPUT:

the Coleman integral $\int_P^Q w$, written in terms of the uniformizer a of the degree d extension

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: H = HyperellipticCurve(x^3-10*x+9)
sage: K = Qp(5,6)
sage: HK = H.change_ring(K)
sage: P = HK(1,0)
sage: Q = HK(0,3)
sage: x,y = HK.monsky_washnitzer_gens()
sage: HK.coleman_integral_from_weierstrass_via_boundary(y.diff(),P,Q,20)
3 + O(a^119)
sage: HK.coleman_integral(y.diff(),P,Q)
3 + O(5^6)
sage: w = HK.invariant_differential()
sage: HK.coleman_integral_from_weierstrass_via_boundary(w,P,Q,20)
2*a^40 + a^80 + a^100 + O(a^105)
```

(continues on next page)

(continued from previous page)

```
sage: HK.coleman_integral(w,P,Q)
2*5^2 + 5^4 + 5^5 + 3*5^6 + O(5^7)
```

AUTHOR:

- Jennifer Balakrishnan

coleman_integrals_on_basis(*P*, *Q*, *algorithm=None*)

Computes the Coleman integrals $\{\int_P^Q x^i dx/2y\}_{i=0}^{2g-1}$

INPUT:

- P point on self
- Q point on self
- algorithm (optional) = None (uses Frobenius) or teichmuller (uses Teichmuller points)

OUTPUT:

the Coleman integrals $\{\int_P^Q x^i dx/2y\}_{i=0}^{2g-1}$

EXAMPLES:

```
sage: K = pAdicField(11, 5)
sage: x = polygen(K)
sage: C = HyperellipticCurve(x^5 + 33/16*x^4 + 3/4*x^3 + 3/8*x^2 - 1/4*x + 1/
↪16)
sage: P = C.lift_x(2)
sage: Q = C.lift_x(3)
sage: C.coleman_integrals_on_basis(P, Q)
(9*11^2 + 7*11^3 + 5*11^4 + O(11^5), 11 + 3*11^2 + 7*11^3 + 11^4 + O(11^5),␣
↪10*11 + 11^2 + 5*11^3 + 5*11^4 + O(11^5), 3 + 9*11^2 + 6*11^3 + 11^4 + O(11^
↪5))
sage: C.coleman_integrals_on_basis(P, Q, algorithm='teichmuller')
(9*11^2 + 7*11^3 + 5*11^4 + O(11^5), 11 + 3*11^2 + 7*11^3 + 11^4 + O(11^5),␣
↪10*11 + 11^2 + 5*11^3 + 5*11^4 + O(11^5), 3 + 9*11^2 + 6*11^3 + 11^4 + O(11^
↪5))
```

```
sage: K = pAdicField(11,5)
sage: x = polygen(K)
sage: C = HyperellipticCurve(x^5 + 33/16*x^4 + 3/4*x^3 + 3/8*x^2 - 1/4*x + 1/
↪16)
sage: P = C(11^-2, 10*11^-5 + 10*11^-4 + 10*11^-3 + 2*11^-2 + 10*11^-1)
sage: Q = C(3*11^-2, 11^-5 + 11^-3 + 10*11^-2 + 7*11^-1)
sage: C.coleman_integrals_on_basis(P, Q)
(6*11^3 + 3*11^4 + 8*11^5 + 4*11^6 + 9*11^7 + O(11^8), 11 + 10*11^2 + 8*11^3␣
↪+ 7*11^4 + 5*11^5 + O(11^6), 6*11^-1 + 2 + 6*11 + 3*11^3 + O(11^4), 9*11^-3␣
↪+ 9*11^-2 + 9*11^-1 + 2*11 + O(11^2))
```

```
sage: R = C(0,1/4)
sage: a = C.coleman_integrals_on_basis(P,R)  # long time (7s on sage.math,␣
↪2011)
sage: b = C.coleman_integrals_on_basis(R,Q)  # long time (9s on sage.math,␣
↪2011)
sage: c = C.coleman_integrals_on_basis(P,Q)  # long time
sage: a+b == c  # long time
True
```

```
sage: R.<x> = QQ['x']
sage: H = HyperellipticCurve(x^3-10*x+9)
sage: K = Qp(5,8)
sage: HK = H.change_ring(K)
sage: S = HK(1,0)
sage: P = HK(0,3)
sage: T = HK(0,1,0)
sage: Q = HK.lift_x(5^-2)
sage: R = HK.lift_x(4*5^-2)
sage: HK.coleman_integrals_on_basis(S,P)
(2*5^2 + 5^4 + 5^5 + 3*5^6 + 3*5^7 + 2*5^8 + O(5^9), 5 + 2*5^2 + 4*5^3 + 2*5^
↪4 + 3*5^6 + 4*5^7 + 2*5^8 + O(5^9))
sage: HK.coleman_integrals_on_basis(T,P)
(2*5^2 + 5^4 + 5^5 + 3*5^6 + 3*5^7 + 2*5^8 + O(5^9), 5 + 2*5^2 + 4*5^3 + 2*5^
↪4 + 3*5^6 + 4*5^7 + 2*5^8 + O(5^9))
sage: HK.coleman_integrals_on_basis(P,S) == -HK.coleman_integrals_on_basis(S,
↪P)
True
sage: HK.coleman_integrals_on_basis(S,Q)
(5 + O(5^4), 4*5^-1 + 4 + 4*5 + 4*5^2 + O(5^3))
sage: HK.coleman_integrals_on_basis(Q,R)
(5 + 2*5^2 + 2*5^3 + 2*5^4 + 3*5^5 + 3*5^6 + 3*5^7 + 5^8 + O(5^9), 3*5^-1 +␣
↪2*5^4 + 5^5 + 2*5^6 + O(5^7))
sage: HK.coleman_integrals_on_basis(S,R) == HK.coleman_integrals_on_basis(S,
↪Q) + HK.coleman_integrals_on_basis(Q,R)
True
sage: HK.coleman_integrals_on_basis(T,T)
(0, 0)
sage: HK.coleman_integrals_on_basis(S,T)
(0, 0)
```

AUTHORS:

- Robert Bradshaw (2007-03): non-Weierstrass points
- Jennifer Balakrishnan and Robert Bradshaw (2010-02): Weierstrass points

coleman_integrals_on_basis_hyperelliptic(*P*, *Q*, *algorithm=None*)

Computes the Coleman integrals $\{\int_P^Q x^i dx/2y\}_{i=0}^{2g-1}$

INPUT:

- P point on self
- Q point on self
- algorithm (optional) = None (uses Frobenius) or teichmuller (uses Teichmuller points)

OUTPUT:

the Coleman integrals $\{\int_P^Q x^i dx/2y\}_{i=0}^{2g-1}$

EXAMPLES:

```
sage: K = pAdicField(11, 5)
sage: x = polygen(K)
sage: C = HyperellipticCurve(x^5 + 33/16*x^4 + 3/4*x^3 + 3/8*x^2 - 1/4*x + 1/
↪16)
sage: P = C.lift_x(2)
sage: Q = C.lift_x(3)
```

(continues on next page)

(continued from previous page)

```
sage: C.coleman_integrals_on_basis(P, Q)
(9*11^2 + 7*11^3 + 5*11^4 + O(11^5), 11 + 3*11^2 + 7*11^3 + 11^4 + O(11^5),␣
↪10*11 + 11^2 + 5*11^3 + 5*11^4 + O(11^5), 3 + 9*11^2 + 6*11^3 + 11^4 + O(11^
↪5))
sage: C.coleman_integrals_on_basis(P, Q, algorithm='teichmuller')
(9*11^2 + 7*11^3 + 5*11^4 + O(11^5), 11 + 3*11^2 + 7*11^3 + 11^4 + O(11^5),␣
↪10*11 + 11^2 + 5*11^3 + 5*11^4 + O(11^5), 3 + 9*11^2 + 6*11^3 + 11^4 + O(11^
↪5))
```

```
sage: K = pAdicField(11,5)
sage: x = polygen(K)
sage: C = HyperellipticCurve(x^5 + 33/16*x^4 + 3/4*x^3 + 3/8*x^2 - 1/4*x + 1/
↪16)
sage: P = C(11^-2, 10*11^-5 + 10*11^-4 + 10*11^-3 + 2*11^-2 + 10*11^-1)
sage: Q = C(3*11^-2, 11^-5 + 11^-3 + 10*11^-2 + 7*11^-1)
sage: C.coleman_integrals_on_basis(P, Q)
(6*11^3 + 3*11^4 + 8*11^5 + 4*11^6 + 9*11^7 + O(11^8), 11 + 10*11^2 + 8*11^3␣
↪+ 7*11^4 + 5*11^5 + O(11^6), 6*11^-1 + 2 + 6*11 + 3*11^3 + O(11^4), 9*11^-3␣
↪+ 9*11^-2 + 9*11^-1 + 2*11 + O(11^2))
```

```
sage: R = C(0,1/4)
sage: a = C.coleman_integrals_on_basis(P,R)  # long time (7s on sage.math,␣
↪2011)
sage: b = C.coleman_integrals_on_basis(R,Q)  # long time (9s on sage.math,␣
↪2011)
sage: c = C.coleman_integrals_on_basis(P,Q)  # long time
sage: a+b == c  # long time
True
```

```
sage: R.<x> = QQ['x']
sage: H = HyperellipticCurve(x^3-10*x+9)
sage: K = Qp(5,8)
sage: HK = H.change_ring(K)
sage: S = HK(1,0)
sage: P = HK(0,3)
sage: T = HK(0,1,0)
sage: Q = HK.lift_x(5^-2)
sage: R = HK.lift_x(4*5^-2)
sage: HK.coleman_integrals_on_basis(S,P)
(2*5^2 + 5^4 + 5^5 + 3*5^6 + 3*5^7 + 2*5^8 + O(5^9), 5 + 2*5^2 + 4*5^3 + 2*5^
↪4 + 3*5^6 + 4*5^7 + 2*5^8 + O(5^9))
sage: HK.coleman_integrals_on_basis(T,P)
(2*5^2 + 5^4 + 5^5 + 3*5^6 + 3*5^7 + 2*5^8 + O(5^9), 5 + 2*5^2 + 4*5^3 + 2*5^
↪4 + 3*5^6 + 4*5^7 + 2*5^8 + O(5^9))
sage: HK.coleman_integrals_on_basis(P,S) == -HK.coleman_integrals_on_basis(S,
↪P)
True
sage: HK.coleman_integrals_on_basis(S,Q)
(5 + O(5^4), 4*5^-1 + 4 + 4*5 + 4*5^2 + O(5^3))
sage: HK.coleman_integrals_on_basis(Q,R)
(5 + 2*5^2 + 2*5^3 + 2*5^4 + 3*5^5 + 3*5^6 + 3*5^7 + 5^8 + O(5^9), 3*5^-1 +␣
↪2*5^4 + 5^5 + 2*5^6 + O(5^7))
sage: HK.coleman_integrals_on_basis(S,R) == HK.coleman_integrals_on_basis(S,
↪Q) + HK.coleman_integrals_on_basis(Q,R)
True
sage: HK.coleman_integrals_on_basis(T,T)
```

(continues on next page)

(continued from previous page)

```
(0, 0)
sage: HK.coleman_integrals_on_basis(S,T)
(0, 0)
```

AUTHORS:

- Robert Bradshaw (2007-03): non-Weierstrass points
- Jennifer Balakrishnan and Robert Bradshaw (2010-02): Weierstrass points

curve_over_ram_extn(*deg*)

Return `self` over $\mathbf{Q}_p(p^{(}1/deg))$.

INPUT:

- deg: the degree of the ramified extension

OUTPUT:

`self` over $\mathbf{Q}_p(p^{(}1/deg))$

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: H = HyperellipticCurve(x^5-23*x^3+18*x^2+40*x)
sage: K = Qp(11,5)
sage: HK = H.change_ring(K)
sage: HL = HK.curve_over_ram_extn(2)
sage: HL
Hyperelliptic Curve over 11-adic Eisenstein Extension Field in a defined by x^
↪2 - 11 defined by (1 + O(a^10))*y^2 = (1 + O(a^10))*x^5 + (10 + 8*a^2 +␣
↪10*a^4 + 10*a^6 + 10*a^8 + O(a^10))*x^3 + (7 + a^2 + O(a^10))*x^2 + (7 +␣
↪3*a^2 + O(a^10))*x
```

AUTHOR:

- Jennifer Balakrishnan

find_char_zero_weier_point(*Q*)

Given Q a point on self in a Weierstrass disc, finds the center of the Weierstrass disc (if defined over self.base_ring())

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: H = HyperellipticCurve(x^3-10*x+9)
sage: K = Qp(5,8)
sage: HK = H.change_ring(K)
sage: P = HK.lift_x(1 + 2*5^2)
sage: Q = HK.lift_x(5^-2)
sage: S = HK(1,0)
sage: T = HK(0,1,0)
sage: HK.find_char_zero_weier_point(P)
(1 + O(5^8) : 0 : 1 + O(5^8))
sage: HK.find_char_zero_weier_point(Q)
(0 : 1 + O(5^8) : 0)
sage: HK.find_char_zero_weier_point(S)
(1 + O(5^8) : 0 : 1 + O(5^8))
sage: HK.find_char_zero_weier_point(T)
(0 : 1 + O(5^8) : 0)
```

AUTHOR:

- Jennifer Balakrishnan

frobenius(*P=None*)

Returns the p-th power lift of Frobenius of P

EXAMPLES:

```
sage: K = Qp(11, 5)
sage: R.<x> = K[]
sage: E = HyperellipticCurve(x^5 - 21*x - 20)
sage: P = E.lift_x(2)
sage: E.frobenius(P)
(2 + 10*11 + 5*11^2 + 11^3 + O(11^5) : 6 + 11 + 8*11^2 + 8*11^3 + 10*11^4 +␣
↪O(11^5) : 1 + O(11^5))
sage: Q = E.teichmuller(P); Q
(2 + 10*11 + 4*11^2 + 9*11^3 + 11^4 + O(11^5) : 6 + 11 + 4*11^2 + 9*11^3 +␣
↪4*11^4 + O(11^5) : 1 + O(11^5))
sage: E.frobenius(Q)
(2 + 10*11 + 4*11^2 + 9*11^3 + 11^4 + O(11^5) : 6 + 11 + 4*11^2 + 9*11^3 +␣
↪4*11^4 + O(11^5) : 1 + O(11^5))
```

```
sage: R.<x> = QQ[]
sage: H = HyperellipticCurve(x^5-23*x^3+18*x^2+40*x)
sage: Q = H(0,0)
sage: u,v = H.local_coord(Q,prec=100)
sage: K = Qp(11,5)
sage: L.<a> = K.extension(x^20-11)
sage: HL = H.change_ring(L)
sage: S = HL(u(a),v(a))
sage: HL.frobenius(S)
(8*a^22 + 10*a^42 + 4*a^44 + 2*a^46 + 9*a^48 + 8*a^50 + a^52 + 7*a^54 +
7*a^56 + 5*a^58 + 9*a^62 + 5*a^64 + a^66 + 6*a^68 + a^70 + 6*a^74 +
2*a^76 + 2*a^78 + 4*a^82 + 5*a^84 + 2*a^86 + 7*a^88 + a^90 + 6*a^92 +
a^96 + 5*a^98 + 2*a^102 + 2*a^106 + 6*a^108 + 8*a^110 + 3*a^112 +
a^114 + 8*a^116 + 10*a^118 + 3*a^120 + O(a^122) :
a^11 + 7*a^33 + 7*a^35 + 4*a^37 + 6*a^39 + 9*a^41 + 8*a^43 + 8*a^45 +
a^47 + 7*a^51 + 4*a^53 + 5*a^55 + a^57 + 7*a^59 + 5*a^61 + 9*a^63 +
4*a^65 + 10*a^69 + 3*a^71 + 2*a^73 + 9*a^75 + 10*a^77 + 6*a^79 +
10*a^81 + 7*a^85 + a^87 + 4*a^89 + 8*a^91 + a^93 + 8*a^95 + 2*a^97 +
7*a^99 + a^101 + 3*a^103 + 6*a^105 + 7*a^107 + 4*a^109 + O(a^111) :
1 + O(a^100))
```

AUTHORS:

- Robert Bradshaw and Jennifer Balakrishnan (2010-02)

get_boundary_point(*curve_over_extn*, *P*)

Given self over an extension field, find a point in the disc of P near the boundary

INPUT:

- curve_over_extn: self over a totally ramified extension
- P: Weierstrass point

OUTPUT:

a point in the disc of P near the boundary

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: H = HyperellipticCurve(x^3-10*x+9)
sage: K = Qp(3,6)
sage: HK = H.change_ring(K)
sage: P = HK(1,0)
sage: J.<a> = K.extension(x^30-3)
sage: HJ  = H.change_ring(J)
sage: S = HK.get_boundary_point(HJ,P)
sage: S
(1 + 2*a^2 + 2*a^6 + 2*a^18 + a^32 + a^34 + a^36 + 2*a^38 + 2*a^40 + a^42 +␣
↪2*a^44 + a^48 + 2*a^50 + 2*a^52 + a^54 + a^56 + 2*a^60 + 2*a^62 + a^70 +␣
↪2*a^72 + a^76 + 2*a^78 + a^82 + a^88 + a^96 + 2*a^98 + 2*a^102 + a^104 +␣
↪2*a^106 + a^108 + 2*a^110 + a^112 + 2*a^116 + a^126 + 2*a^130 + 2*a^132 + a^
↪144 + 2*a^148 + 2*a^150 + a^152 + 2*a^154 + a^162 + a^164 + a^166 + a^168 +␣
↪a^170 + a^176 + a^178 + O(a^180) : a + O(a^180) : 1 + O(a^180))
```

AUTHOR:

- Jennifer Balakrishnan

is_in_weierstrass_disc(P)

Checks if P is in a Weierstrass disc

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: H = HyperellipticCurve(x^3-10*x+9)
sage: K = Qp(5,8)
sage: HK = H.change_ring(K)
sage: P = HK(0,3)
sage: HK.is_in_weierstrass_disc(P)
False
sage: Q = HK(0,1,0)
sage: HK.is_in_weierstrass_disc(Q)
True
sage: S = HK(1,0)
sage: HK.is_in_weierstrass_disc(S)
True
sage: T = HK.lift_x(1+3*5^2); T
(1 + 3*5^2 + O(5^8) : 3*5 + 4*5^2 + 5^4 + 3*5^5 + 5^6 + O(5^7) : 1 + O(5^8))
sage: HK.is_in_weierstrass_disc(T)
True
```

AUTHOR:

- Jennifer Balakrishnan (2010-02)

is_same_disc(P, Q)

Checks if P, Q are in same residue disc

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: H = HyperellipticCurve(x^3-10*x+9)
sage: K = Qp(5,8)
sage: HK = H.change_ring(K)
sage: P = HK.lift_x(1 + 2*5^2)
sage: Q = HK.lift_x(5^-2)
sage: S = HK(1,0)
```

(continues on next page)

(continued from previous page)

```
sage: HK.is_same_disc(P,Q)
False
sage: HK.is_same_disc(P,S)
True
sage: HK.is_same_disc(Q,S)
False
```

is_weierstrass(*P*)

Checks if P is a Weierstrass point (i.e., fixed by the hyperelliptic involution)

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: H = HyperellipticCurve(x^3-10*x+9)
sage: K = Qp(5,8)
sage: HK = H.change_ring(K)
sage: P = HK(0,3)
sage: HK.is_weierstrass(P)
False
sage: Q = HK(0,1,0)
sage: HK.is_weierstrass(Q)
True
sage: S = HK(1,0)
sage: HK.is_weierstrass(S)
True
sage: T = HK.lift_x(1+3*5^2); T
(1 + 3*5^2 + O(5^8) : 3*5 + 4*5^2 + 5^4 + 3*5^5 + 5^6 + O(5^7) : 1 + O(5^8))
sage: HK.is_weierstrass(T)
False
```

AUTHOR:

- Jennifer Balakrishnan (2010-02)

local_analytic_interpolation(*P*, *Q*)

For points P, Q in the same residue disc, this constructs an interpolation from P to Q (in homogeneous coordinates) in a power series in the local parameter t, with precision equal to the p-adic precision of the underlying ring.

INPUT:

- P and Q points on self in the same residue disc

OUTPUT:

Returns a point $X(t) = (x(t) : y(t) : z(t))$ such that:

(1) $X(0) = P$ and $X(1) = Q$ if P, Q are not in the infinite disc

(2) $X(P[0]^g/P[1]) = P$ and $X(Q[0]^g/Q[1]) = Q$ if P, Q are in the infinite disc

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: H = HyperellipticCurve(x^3-10*x+9)
sage: K = Qp(5,8)
sage: HK = H.change_ring(K)
```

A non-Weierstrass disc:

```
sage: P = HK(0,3)
sage: Q = HK(5, 3 + 3*5^2 + 2*5^3 + 3*5^4 + 2*5^5 + 2*5^6 + 3*5^7 + O(5^8))
sage: x,y,z, = HK.local_analytic_interpolation(P,Q)
sage: x(0) == P[0], x(1) == Q[0], y(0) == P[1], y.polynomial()(1) == Q[1]
(True, True, True, True)
```

A finite Weierstrass disc:

```
sage: P = HK.lift_x(1 + 2*5^2)
sage: Q = HK.lift_x(1 + 3*5^2)
sage: x,y,z = HK.local_analytic_interpolation(P,Q)
sage: x(0) == P[0], x.polynomial()(1) == Q[0], y(0) == P[1], y(1) == Q[1]
(True, True, True, True)
```

The infinite disc:

```
sage: P = HK.lift_x(5^-2)
sage: Q = HK.lift_x(4*5^-2)
sage: x,y,z = HK.local_analytic_interpolation(P,Q)
sage: x = x/z
sage: y = y/z
sage: x(P[0]/P[1]) == P[0]
True
sage: x(Q[0]/Q[1]) == Q[0]
True
sage: y(P[0]/P[1]) == P[1]
True
sage: y(Q[0]/Q[1]) == Q[1]
True
```

An error if points are not in the same disc:

```
sage: x,y,z = HK.local_analytic_interpolation(P,HK(1,0))
Traceback (most recent call last):
...
ValueError: (5^-2 + O(5^6) : 4*5^-3 + 4*5^-2 + 4*5^-1 + 4 + 4*5 + 3*5^3 + 5^4␣
↪+ O(5^5) : 1 + O(5^8)) and (1 + O(5^8) : 0 : 1 + O(5^8)) are not in the␣
↪same residue disc
```

AUTHORS:

- Robert Bradshaw (2007-03)
- Jennifer Balakrishnan (2010-02)

newton_sqrt (*f*, *x0*, *prec*)

Takes the square root of the power series f by Newton's method

NOTE:

this function should eventually be moved to p-adic power series ring

INPUT:

- `f` – power series with coefficients in $\mathbf{Q}_p$ or an extension
- `x0` – seeds the Newton iteration
- `prec` – precision

OUTPUT: the square root of f

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: H = HyperellipticCurve(x^5-23*x^3+18*x^2+40*x)
sage: Q = H(0,0)
sage: u,v = H.local_coord(Q,prec=100)
sage: K = Qp(11,5)
sage: HK = H.change_ring(K)
sage: L.<a> = K.extension(x^20-11)
sage: HL = H.change_ring(L)
sage: S = HL(u(a),v(a))
sage: f = H.hyperelliptic_polynomials()[0]
sage: y = HK.newton_sqrt( f(u(a)^11), a^11,5)
sage: y^2 - f(u(a)^11)
O(a^122)
```

AUTHOR:

- Jennifer Balakrishnan

residue_disc(P)

Gives the residue disc of P

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: H = HyperellipticCurve(x^3-10*x+9)
sage: K = Qp(5,8)
sage: HK = H.change_ring(K)
sage: P = HK.lift_x(1 + 2*5^2)
sage: HK.residue_disc(P)
(1 : 0 : 1)
sage: Q = HK(0,3)
sage: HK.residue_disc(Q)
(0 : 3 : 1)
sage: S = HK.lift_x(5^-2)
sage: HK.residue_disc(S)
(0 : 1 : 0)
sage: T = HK(0,1,0)
sage: HK.residue_disc(T)
(0 : 1 : 0)
```

AUTHOR:

- Jennifer Balakrishnan

teichmuller(P)

Find a Teichm:uller point in the same residue class of P.

Because this lift of frobenius acts as $x \mapsto x^p$, take the Teichmuller lift of x and then find a matching y from that.

EXAMPLES:

```
sage: K = pAdicField(7, 5)
sage: E = EllipticCurve(K, [-31/3, -2501/108]) # 11a
sage: P = E(K(14/3), K(11/2))
sage: E.frobenius(P) == P
False
sage: TP = E.teichmuller(P); TP
```

(continues on next page)

(continued from previous page)

```
(0 : 2 + 3*7 + 3*7^2 + 3*7^4 + O(7^5) : 1 + O(7^5))
sage: E.frobenius(TP) == TP
True
sage: (TP[0] - P[0]).valuation() > 0, (TP[1] - P[1]).valuation() > 0
(True, True)
```

tiny_integrals(F, P, Q)

Evaluate the integrals of $f_i dx/2y$ from P to Q for each f_i in F by formally integrating a power series in a local parameter t

P and Q MUST be in the same residue disc for this result to make sense.

INPUT:

- F a list of functions f_i
- P a point on self
- Q a point on self (in the same residue disc as P)

OUTPUT:

The integrals $\int_P^Q f_i dx/2y$

EXAMPLES:

```
sage: K = pAdicField(17, 5)
sage: E = EllipticCurve(K, [-31/3, -2501/108]) # 11a
sage: P = E(K(14/3), K(11/2))
sage: TP = E.teichmuller(P);
sage: x,y = E.monsky_washnitzer_gens()
sage: E.tiny_integrals([1,x],P, TP) == E.tiny_integrals_on_basis(P,TP)
True
```

```
sage: K = pAdicField(11, 5)
sage: x = polygen(K)
sage: C = HyperellipticCurve(x^5 + 33/16*x^4 + 3/4*x^3 + 3/8*x^2 - 1/4*x + 1/
↪16)
sage: P = C.lift_x(11^(-2))
sage: Q = C.lift_x(3*11^(-2))
sage: C.tiny_integrals([1],P,Q)
(5*11^3 + 7*11^4 + 2*11^5 + 6*11^6 + 11^7 + O(11^8))
```

Note that this fails if the points are not in the same residue disc:

```
sage: S = C(0,1/4)
sage: C.tiny_integrals([1,x,x^2,x^3],P,S)
Traceback (most recent call last):
...
ValueError: (11^-2 + O(11^3) : 11^-5 + 8*11^-2 + O(11^0) : 1 + O(11^5)) and␣
↪(0 : 3 + 8*11 + 2*11^2 + 8*11^3 + 2*11^4 + O(11^5) : 1 + O(11^5)) are not␣
↪in the same residue disc
```

tiny_integrals_on_basis(P, Q)

Evaluate the integrals $\{\int_P^Q x^i dx/2y\}_{i=0}^{2g-1}$ by formally integrating a power series in a local parameter t. P and Q MUST be in the same residue disc for this result to make sense.

INPUT:

- P a point on self
- Q a point on self (in the same residue disc as P)

OUTPUT:

The integrals $\{\int_P^Q x^i dx/2y\}_{i=0}^{2g-1}$

EXAMPLES:

```
sage: K = pAdicField(17, 5)
sage: E = EllipticCurve(K, [-31/3, -2501/108]) # 11a
sage: P = E(K(14/3), K(11/2))
sage: TP = E.teichmuller(P);
sage: E.tiny_integrals_on_basis(P, TP)
(17 + 14*17^2 + 17^3 + 8*17^4 + O(17^5), 16*17 + 5*17^2 + 8*17^3 + 14*17^4 +␣
↪O(17^5))
```

```
sage: K = pAdicField(11, 5)
sage: x = polygen(K)
sage: C = HyperellipticCurve(x^5 + 33/16*x^4 + 3/4*x^3 + 3/8*x^2 - 1/4*x + 1/
↪16)
sage: P = C.lift_x(11^(-2))
sage: Q = C.lift_x(3*11^(-2))
sage: C.tiny_integrals_on_basis(P,Q)
(5*11^3 + 7*11^4 + 2*11^5 + 6*11^6 + 11^7 + O(11^8), 10*11 + 2*11^3 + 3*11^4␣
↪+ 5*11^5 + O(11^6), 5*11^-1 + 8 + 4*11 + 10*11^2 + 7*11^3 + O(11^4), 2*11^-
↪3 + 11^-2 + 11^-1 + 10 + 8*11 + O(11^2))
```

Note that this fails if the points are not in the same residue disc:

```
sage: S = C(0,1/4)
sage: C.tiny_integrals_on_basis(P,S)
Traceback (most recent call last):
...
ValueError: (11^-2 + O(11^3) : 11^-5 + 8*11^-2 + O(11^0) : 1 + O(11^5)) and␣
↪(0 : 3 + 8*11 + 2*11^2 + 8*11^3 + 2*11^4 + O(11^5) : 1 + O(11^5)) are not␣
↪in the same residue disc
```

weierstrass_points()

Return the Weierstrass points of self defined over self.base_ring(), that is, the point at infinity and those points in the support of the divisor of y

EXAMPLES:

```
sage: K = pAdicField(11, 5)
sage: x = polygen(K)
sage: C = HyperellipticCurve(x^5 + 33/16*x^4 + 3/4*x^3 + 3/8*x^2 - 1/4*x + 1/
↪16)
sage: C.weierstrass_points()
[(0 : 1 + O(11^5) : 0), (7 + 10*11 + 4*11^3 + O(11^5) : 0 : 1 + O(11^5))]
```

20.5 Hyperelliptic curves over the rationals

class sage.schemes.hyperelliptic_curves.hyperelliptic_rational_field.**HyperellipticCurve_rat**

Bases: *HyperellipticCurve_generic*, ProjectivePlaneCurve_field

lseries(*prec=53*)

Return the L-series of this hyperelliptic curve of genus 2.

EXAMPLES:

```
sage: x = polygen(QQ, 'x')
sage: C = HyperellipticCurve(x^2+x, x^3+x^2+1)
sage: C.lseries()
PARI L-function associated to Hyperelliptic Curve
over Rational Field defined by y^2 + (x^3 + x^2 + 1)*y = x^2 + x
```

matrix_of_frobenius(*p*, *prec=20*)

Compute the matrix of Frobenius on Monsky-Washnitzer cohomology using the p-adic field with precision `prec`.

This function is essentially a wrapper function of *sage.schemes.hyperelliptic_curves.monsky_washnitzer.matrix_of_frobenius_hyperelliptic()*.

INPUT:

- `p` (prime integer or pAdic ring / field) – if `p` is an integer, constructs a `pAdicField` with `p` to compute the matrix of Frobenius, otherwise uses the supplied pAdic ring or field.
- `prec` (optional) – if `p` is an prime integer, the p-adic precision of the coefficient ring constructed.

EXAMPLES:

```
sage: K = pAdicField(5, prec=3)
sage: R.<x> = QQ['x']
sage: H = HyperellipticCurve(x^5 - 2*x + 3)
sage: H.matrix_of_frobenius(K)
[            4*5 + O(5^3)       5 + 2*5^2 + O(5^3) 2 + 3*5 + 2*5^2 + O(5^3)  ␣
↪  2 + 5 + 5^2 + O(5^3)]
[      3*5 + 5^2 + O(5^3)             3*5 + O(5^3)             4*5 + O(5^3)  ␣
↪      2 + 5^2 + O(5^3)]
[    4*5 + 4*5^2 + O(5^3)     3*5 + 2*5^2 + O(5^3)       5 + 3*5^2 + O(5^3)  ␣
↪  2*5 + 2*5^2 + O(5^3)]
[            5^2 + O(5^3)       5 + 4*5^2 + O(5^3)     4*5 + 3*5^2 + O(5^3)  ␣
↪          2*5 + O(5^3)]
```

You can also pass directly a prime p with to construct a pAdic field with precision `prec`:

```
sage: H.matrix_of_frobenius(3, prec=2)
[        O(3^2)     3 + O(3^2)          O(3^2)          O(3^2)]
[    3 + O(3^2)         O(3^2)          O(3^2) 2 + 3 + O(3^2)]
[  2*3 + O(3^2)         O(3^2)          O(3^2)    3^-1 + O(3)]
[        O(3^2)         O(3^2)     3 + O(3^2)          O(3^2)]
```

20.6 Mestre's algorithm

This file contains functions that:

- create hyperelliptic curves from the Igusa-Clebsch invariants (over **Q** and finite fields)
- create Mestre's conic from the Igusa-Clebsch invariants

AUTHORS:

- Florian Bouyer
- Marco Streng

sage.schemes.hyperelliptic_curves.mestre.**HyperellipticCurve_from_invariants**(*i*, *reduced=True*, *precision=None*, *algorithm='default'*)

Returns a hyperelliptic curve with the given Igusa-Clebsch invariants up to scaling.

The output is a curve over the field in which the Igusa-Clebsch invariants are given. The output curve is unique up to isomorphism over the algebraic closure. If no such curve exists over the given field, then raise a ValueError.

INPUT:

- `i` – list or tuple of length 4 containing the four Igusa-Clebsch invariants: I2,I4,I6,I10.
- `reduced` – Boolean (default = True) If True, tries to reduce the polynomial defining the hyperelliptic curve using the function `reduce_polynomial()` (see the `reduce_polynomial()` documentation for more details).
- `precision` – integer (default = None) Which precision for real and complex numbers should the reduction use. This only affects the reduction, not the correctness. If None, the algorithm uses the default 53 bit precision.
- `algorithm` – `'default'` or `'magma'`. If set to `'magma'`, uses Magma to parameterize Mestre's conic (needs Magma to be installed).

OUTPUT:

A hyperelliptic curve object.

EXAMPLES:

Examples over the rationals:

```
sage: HyperellipticCurve_from_invariants([3840,414720,491028480,2437709561856])
Traceback (most recent call last):
...
NotImplementedError: Reduction of hyperelliptic curves not yet implemented.
See issues #14755 and #14756.

sage: HyperellipticCurve_from_invariants([3840,414720,491028480,2437709561856],␣
↪reduced=False)
Hyperelliptic Curve over Rational Field defined by
```

(continues on next page)

(continued from previous page)

```
y^2 = -46656*x^6 + 46656*x^5 - 19440*x^4 + 4320*x^3 - 540*x^2 + 4410*x - 1

sage: HyperellipticCurve_from_invariants([21, 225/64, 22941/512, 1])
Traceback (most recent call last):
...
NotImplementedError: Reduction of hyperelliptic curves not yet implemented.
See issues #14755 and #14756.
```

An example over a finite field:

```
sage: H = HyperellipticCurve_from_invariants([GF(13)(1), 3, 7, 5]); H
Hyperelliptic Curve over Finite Field of size 13 defined by ...
sage: H.igusa_clebsch_invariants()
(4, 9, 6, 11)
```

An example over a number field:

```
sage: K = QuadraticField(353, 'a')                                              #␣
↪needs sage.rings.number_field
sage: H = HyperellipticCurve_from_invariants([21, 225/64, 22941/512, 1],        #␣
↪needs sage.rings.number_field
....:                                         reduced=false)
sage: f = K['x'](H.hyperelliptic_polynomials()[0])                              #␣
↪needs sage.rings.number_field
```

If the Mestre Conic defined by the Igusa-Clebsch invariants has no rational points, then there exists no hyperelliptic curve over the base field with the given invariants.:

```
sage: HyperellipticCurve_from_invariants([1,2,3,4])
Traceback (most recent call last):
...
ValueError: No such curve exists over Rational Field as there are
no rational points on Projective Conic Curve over Rational Field defined by
-2572155000*u^2 - 317736000*u*v + 1250755459200*v^2 + 2501510918400*u*w
+ 39276887040*v*w + 2736219686912*w^2
```

Mestre's algorithm only works for generic curves of genus two, so another algorithm is needed for those curves with extra automorphism. See also Issue #12199:

```
sage: P.<x> = QQ[]
sage: C = HyperellipticCurve(x^6 + 1)
sage: i = C.igusa_clebsch_invariants()
sage: HyperellipticCurve_from_invariants(i)
Traceback (most recent call last):
...
TypeError: F (=0) must have degree 2
```

Igusa-Clebsch invariants also only work over fields of characteristic different from 2, 3, and 5, so another algorithm will be needed for fields of those characteristics. See also Issue #12200:

```
sage: P.<x> = GF(3)[]
sage: HyperellipticCurve(x^6 + x + 1).igusa_clebsch_invariants()
Traceback (most recent call last):
...
NotImplementedError: Invariants of binary sextics/genus 2 hyperelliptic curves
not implemented in characteristics 2, 3, and 5
```

(continues on next page)

(continued from previous page)

```
sage: HyperellipticCurve_from_invariants([GF(5)(1), 1, 0, 1])
Traceback (most recent call last):
...
ZeroDivisionError: inverse of Mod(0, 5) does not exist
```

ALGORITHM:

This is Mestre's algorithm [Mes1991]. Our implementation is based on the formulae on page 957 of [LY2001], cross-referenced with [Wam1999b] to correct typos.

First construct Mestre's conic using the *Mestre_conic()* function. Parametrize the conic if possible. Let f_1, f_2, f_3 be the three coordinates of the parametrization of the conic by the projective line, and change them into one variable by letting $F_i = f_i(t, 1)$. Note that each F_i has degree at most 2.

Then construct a sextic polynomial $f = \sum_{0<=i,j,k<=3} c_{ijk} * F_i * F_j * F_k$, where c_{ijk} are defined as rational functions in the invariants (see the source code for detailed formulae for c_{ijk}). The output is the hyperelliptic curve $y^2 = f$.

sage.schemes.hyperelliptic_curves.mestre.**Mestre_conic**(*i*, *xyz=False*, *names='u,v,w'*)

Return the conic equation from Mestre's algorithm given the Igusa-Clebsch invariants.

It has a rational point if and only if a hyperelliptic curve corresponding to the invariants exists.

INPUT:

- `i` – list or tuple of length 4 containing the four Igusa-Clebsch invariants: I2, I4, I6, I10
- `xyz` – Boolean (default: `False`) if `True`, the algorithm also returns three invariants x,`y`,`z` used in Mestre's algorithm
- `names` (default: `'u,v,w'`) – the variable names for the conic

OUTPUT:

A Conic object

EXAMPLES:

A standard example:

```
sage: Mestre_conic([1,2,3,4])
Projective Conic Curve over Rational Field defined by
-2572155000*u^2 - 317736000*u*v + 1250755459200*v^2 + 2501510918400*u*w
+ 39276887040*v*w + 2736219686912*w^2
```

Note that the algorithm works over number fields as well:

```
sage: x = polygen(ZZ, 'x')
sage: k = NumberField(x^2 - 41, 'a')                                    #␣
↪needs sage.rings.number_field
sage: a = k.an_element()                                                #␣
↪needs sage.rings.number_field
sage: Mestre_conic([1, 2 + a, a, 4 + a])                                #␣
↪needs sage.rings.number_field
Projective Conic Curve over Number Field in a with defining polynomial x^2 - 41
 defined by (-801900000*a + 343845000)*u^2 + (855360000*a + 15795864000)*u*v
  + (312292800000*a + 1284808579200)*v^2 + (624585600000*a + 2569617158400)*u*w
  + (15799910400*a + 234573143040)*v*w + (2034199306240*a + 16429854656512)*w^2
```

And over finite fields:

```
sage: Mestre_conic([GF(7)(10), GF(7)(1), GF(7)(2), GF(7)(3)])
Projective Conic Curve over Finite Field of size 7
defined by -2*u*v - v^2 - 2*u*w + 2*v*w - 3*w^2
```

An example with `xyz`:

```
sage: Mestre_conic([5,6,7,8], xyz=True)
(Projective Conic Curve over Rational Field
  defined by -415125000*u^2 + 608040000*u*v + 33065136000*v^2
             + 66130272000*u*w + 240829440*v*w + 10208835584*w^2,
 232/1125, -1072/16875, 14695616/2109375)
```

ALGORITHM:

The formulas are taken from pages 956 - 957 of [LY2001] and based on pages 321 and 332 of [Mes1991].

See the code or [LY2001] for the detailed formulae defining x, y, z and L.

20.7 Computation of Frobenius matrix on Monsky-Washnitzer cohomology

The most interesting functions to be exported here are *matrix_of_frobenius()* and *adjusted_prec()*.

Currently this code is limited to the case $p \geq 5$ (no $GF(p^n)$ for $n > 1$), and only handles the elliptic curve case (not more general hyperelliptic curves).

REFERENCES:

- [Ked2001]
- [Edix]

AUTHORS:

- David Harvey and Robert Bradshaw: initial code developed at the 2006 MSRI graduate workshop, working with Jennifer Balakrishnan and Liang Xiao
- David Harvey (2006-08): cleaned up, rewrote some chunks, lots more documentation, added Newton iteration method, added more complete 'trace trick', integrated better into Sage.
- David Harvey (2007-02): added algorithm with sqrt(p) complexity (removed in May 2007 due to better C++ implementation)
- Robert Bradshaw (2007-03): keep track of exact form in reduction algorithms
- Robert Bradshaw (2007-04): generalization to hyperelliptic curves
- Julian Rueth (2014-05-09): improved caching

class sage.schemes.hyperelliptic_curves.monsky_washnitzer.**MonskyWashnitzerDifferential**(*parent*, *val*, *offset=0*

Bases: ModuleElement

An element of the Monsky-Washnitzer ring of differentials, of the form $F dx/2y$.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: C = HyperellipticCurve(x^5 - 4*x + 4)
sage: x,y = C.monsky_washnitzer_gens()
sage: MW = C.invariant_differential().parent()
sage: MW(x)
x dx/2y
sage: MW(y)
y*1 dx/2y
sage: MW(x, 10)
y^10*x dx/2y
```

coeff()

Return A, where this element is $Adx/2y$.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: C = HyperellipticCurve(x^5 - 4*x + 4)
sage: x,y = C.monsky_washnitzer_gens()
sage: w = C.invariant_differential()
sage: w
1 dx/2y
sage: w.coeff()
1
sage: (x*y*w).coeff()
y*x
```

coeffs(*R=None*)

Used to obtain the raw coefficients of a differential, see *SpecialHyperellipticQuotientElement.coeffs()*

INPUT:

- R – An (optional) base ring in which to cast the coefficients

OUTPUT:

The raw coefficients of A where `self` is $Adx/2y$.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: C = HyperellipticCurve(x^5 - 4*x + 4)
sage: x,y = C.monsky_washnitzer_gens()
sage: w = C.invariant_differential()
sage: w.coeffs()
([(1, 0, 0, 0, 0)], 0)
sage: (x*w).coeffs()
([(0, 1, 0, 0, 0)], 0)
sage: (y*w).coeffs()
([(0, 0, 0, 0, 0), (1, 0, 0, 0, 0)], 0)
sage: (y^-2*w).coeffs()
([(1, 0, 0, 0, 0), (0, 0, 0, 0, 0), (0, 0, 0, 0, 0)], -2)
```

coleman_integral(P, Q)

Compute the definite integral of `self` from P to Q.

INPUT:

- P, Q – two points on the underlying curve

OUTPUT:

$\int_P^Q$ self

EXAMPLES:

```
sage: K = pAdicField(5,7)
sage: E = EllipticCurve(K,[-31/3,-2501/108]) #11a
sage: P = E(K(14/3), K(11/2))
sage: w = E.invariant_differential()
sage: w.coleman_integral(P,2*P)
O(5^6)

sage: Q = E([3,58332])
sage: w.coleman_integral(P,Q)
2*5 + 4*5^2 + 3*5^3 + 4*5^4 + 3*5^5 + O(5^6)
sage: w.coleman_integral(2*P,Q)
2*5 + 4*5^2 + 3*5^3 + 4*5^4 + 3*5^5 + O(5^6)
sage: (2*w).coleman_integral(P, Q) == 2*(w.coleman_integral(P, Q))
True
```

extract_pow_y(k)

Return the power of y in A where `self` is $Adx/2y$.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: C = HyperellipticCurve(x^5 - 3*x + 1)
sage: x,y = C.monsky_washnitzer_gens()
sage: A = y^5 - x*y^3
sage: A.extract_pow_y(5)
[1, 0, 0, 0, 0]
sage: (A * C.invariant_differential()).extract_pow_y(5)
[1, 0, 0, 0, 0]
```

integrate(P, Q)

Compute the definite integral of `self` from P to Q.

INPUT:

- P, Q – two points on the underlying curve

OUTPUT:

$\int_P^Q$ self

EXAMPLES:

```
sage: K = pAdicField(5,7)
sage: E = EllipticCurve(K,[-31/3,-2501/108]) #11a
sage: P = E(K(14/3), K(11/2))
sage: w = E.invariant_differential()
sage: w.coleman_integral(P,2*P)
O(5^6)

sage: Q = E([3,58332])
sage: w.coleman_integral(P,Q)
2*5 + 4*5^2 + 3*5^3 + 4*5^4 + 3*5^5 + O(5^6)
sage: w.coleman_integral(2*P,Q)
2*5 + 4*5^2 + 3*5^3 + 4*5^4 + 3*5^5 + O(5^6)
```

(continues on next page)

(continued from previous page)

```
sage: (2*w).coleman_integral(P, Q) == 2*(w.coleman_integral(P, Q))
True
```

max_pow_y()

Return the maximum power of y in A where `self` is $Adx/2y$.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: C = HyperellipticCurve(x^5 - 3*x + 1)
sage: x,y = C.monsky_washnitzer_gens()
sage: w = y^5 * C.invariant_differential()
sage: w.max_pow_y()
5
sage: w = (x^2*y^4 + y^5) * C.invariant_differential()
sage: w.max_pow_y()
5
```

min_pow_y()

Return the minimum power of y in A where `self` is $Adx/2y$.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: C = HyperellipticCurve(x^5 - 3*x + 1)
sage: x,y = C.monsky_washnitzer_gens()
sage: w = y^5 * C.invariant_differential()
sage: w.min_pow_y()
5
sage: w = (x^2*y^4 + y^5) * C.invariant_differential()
sage: w.min_pow_y()
4
```

reduce()

Use homology relations to find a and f such that this element is equal to $a + df$, where a is given in terms of the $x^i dx/2y$.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: C = HyperellipticCurve(x^5 - 4*x + 4)
sage: x,y = C.monsky_washnitzer_gens()
sage: w = (y*x).diff()
sage: w.reduce()
(y*x, 0 dx/2y)

sage: w = x^4 * C.invariant_differential()
sage: w.reduce()
(1/5*y*1, 4/5*1 dx/2y)

sage: w = sum(QQ.random_element() * x^i * y^j
....:         for i in [0..4] for j in [-3..3]) * C.invariant_differential()
sage: f, a = w.reduce()
sage: f.diff() + a - w
0 dx/2y
```

reduce_fast(*even_degree_only=False*)

Use homology relations to find a and f such that this element is equal to $a + df$, where a is given in terms of the $x^i dx/2y$.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^3 - 4*x + 4)
sage: x, y = E.monsky_washnitzer_gens()
sage: x.diff().reduce_fast()
(x, (0, 0))
sage: y.diff().reduce_fast()
(y*1, (0, 0))
sage: (y^-1).diff().reduce_fast()
((y^-1)*1, (0, 0))
sage: (y^-11).diff().reduce_fast()
((y^-11)*1, (0, 0))
sage: (x*y^2).diff().reduce_fast()
(y^2*x, (0, 0))
```

reduce_neg_y()

Use homology relations to eliminate negative powers of y.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: C = HyperellipticCurve(x^5 - 3*x + 1)
sage: x,y = C.monsky_washnitzer_gens()
sage: (y^-1).diff().reduce_neg_y()
((y^-1)*1, 0 dx/2y)
sage: (y^-5*x^2+y^-1*x).diff().reduce_neg_y()
((y^-1)*x + (y^-5)*x^2, 0 dx/2y)
```

reduce_neg_y_fast(*even_degree_only=False*)

Use homology relations to eliminate negative powers of y.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^5 - 3*x + 1)
sage: x, y = E.monsky_washnitzer_gens()
sage: (y^-1).diff().reduce_neg_y_fast()
((y^-1)*1, 0 dx/2y)
sage: (y^-5*x^2+y^-1*x).diff().reduce_neg_y_fast()
((y^-1)*x + (y^-5)*x^2, 0 dx/2y)
```

It leaves non-negative powers of y alone:

```
sage: y.diff()
(-3*1 + 5*x^4) dx/2y
sage: y.diff().reduce_neg_y_fast()
(0, (-3*1 + 5*x^4) dx/2y)
```

reduce_neg_y_faster(*even_degree_only=False*)

Use homology relations to eliminate negative powers of y.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: C = HyperellipticCurve(x^5 - 3*x + 1)
sage: x,y = C.monsky_washnitzer_gens()
sage: (y^-1).diff().reduce_neg_y()
((y^-1)*1, 0 dx/2y)
sage: (y^-5*x^2+y^-1*x).diff().reduce_neg_y_faster()
((y^-1)*x + (y^-5)*x^2, 0 dx/2y)
```

reduce_pos_y()

Use homology relations to eliminate positive powers of y.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: C = HyperellipticCurve(x^3-4*x+4)
sage: x,y = C.monsky_washnitzer_gens()
sage: (y^2).diff().reduce_pos_y()
(y^2*1, 0 dx/2y)
sage: (y^2*x).diff().reduce_pos_y()
(y^2*x, 0 dx/2y)
sage: (y^92*x).diff().reduce_pos_y()
(y^92*x, 0 dx/2y)
sage: w = (y^3 + x).diff()
sage: w += w.parent()(x)
sage: w.reduce_pos_y_fast()
(y^3*1 + x, x dx/2y)
```

reduce_pos_y_fast(*even_degree_only=False*)

Use homology relations to eliminate positive powers of y.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^3 - 4*x + 4)
sage: x, y = E.monsky_washnitzer_gens()
sage: y.diff().reduce_pos_y_fast()
(y*1, 0 dx/2y)
sage: (y^2).diff().reduce_pos_y_fast()
(y^2*1, 0 dx/2y)
sage: (y^2*x).diff().reduce_pos_y_fast()
(y^2*x, 0 dx/2y)
sage: (y^92*x).diff().reduce_pos_y_fast()
(y^92*x, 0 dx/2y)
sage: w = (y^3 + x).diff()
sage: w += w.parent()(x)
sage: w.reduce_pos_y_fast()
(y^3*1 + x, x dx/2y)
```

class sage.schemes.hyperelliptic_curves.monsky_washnitzer.**MonskyWashnitzerDifferentialRing**

Bases: UniqueRepresentation, Module

A ring of Monsky–Washnitzer differentials over base_ring.

Element

alias of *MonskyWashnitzerDifferential*

Q()

Return $Q(x)$ where the model of the underlying hyperelliptic curve of self is given by $y^2 = Q(x)$.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: C = HyperellipticCurve(x^5 - 4*x + 4)
sage: MW = C.invariant_differential().parent()
sage: MW.Q()
x^5 - 4*x + 4
```

base_extend(*R*)

Return a new differential ring which is `self` base-extended to R.

INPUT:

- `R` – ring

OUTPUT:

Self, base-extended to R.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: C = HyperellipticCurve(x^5 - 4*x + 4)
sage: MW = C.invariant_differential().parent()
sage: MW.base_ring()
SpecialHyperellipticQuotientRing K[x,y,y^-1] / (y^2 = x^5 - 4*x + 4)
 over Rational Field
sage: MW.base_extend(Qp(5,5)).base_ring()                                   #␣
↪needs sage.rings.padics
SpecialHyperellipticQuotientRing K[x,y,y^-1] / (y^2 = (1 + O(5^5))*x^5
            + (1 + 4*5 + 4*5^2 + 4*5^3 + 4*5^4 + O(5^5))*x + 4 + O(5^5))
 over 5-adic Field with capped relative precision 5
```

change_ring(*R*)

Return a new differential ring which is self with the coefficient ring changed to R.

INPUT:

- `R` – ring of coefficients

OUTPUT:

`self` with the coefficient ring changed to R.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: C = HyperellipticCurve(x^5 - 4*x + 4)
sage: MW = C.invariant_differential().parent()
sage: MW.base_ring()
SpecialHyperellipticQuotientRing K[x,y,y^-1] / (y^2 = x^5 - 4*x + 4)
 over Rational Field
sage: MW.change_ring(Qp(5,5)).base_ring()                                   #␣
↪needs sage.rings.padics
SpecialHyperellipticQuotientRing K[x,y,y^-1] / (y^2 = (1 + O(5^5))*x^5
            + (1 + 4*5 + 4*5^2 + 4*5^3 + 4*5^4 + O(5^5))*x + 4 + O(5^5))
 over 5-adic Field with capped relative precision 5
```

degree()

Return the degree of $Q(x)$, where the model of the underlying hyperelliptic curve of `self` is given by $y^2 = Q(x)$.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: C = HyperellipticCurve(x^5 - 4*x + 4)
sage: MW = C.invariant_differential().parent()
sage: MW.Q()
x^5 - 4*x + 4
sage: MW.degree()
5
```

dimension()

Return the dimension of `self`.

EXAMPLES:

```
sage: # needs sage.rings.padics
sage: R.<x> = QQ['x']
sage: C = HyperellipticCurve(x^5 - 4*x + 4)
sage: K = Qp(7,5)
sage: CK = C.change_ring(K)
sage: MW = CK.invariant_differential().parent()
sage: MW.dimension()
4
```

frob_Q(p)

Return and cache $Q(x^p)$, which is used in computing the image of y under a p-power lift of Frobenius to $A^\dagger$.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: C = HyperellipticCurve(x^5 - 4*x + 4)
sage: MW = C.invariant_differential().parent()
sage: MW.frob_Q(3)
-(60-48*y^2+12*y^4-y^6)*1 + (192-96*y^2+12*y^4)*x - (192-48*y^2)*x^2 + 60*x^3
sage: MW.Q()(MW.x_to_p(3))                                                  #␣
↪needs sage.rings.real_interval_field
-(60-48*y^2+12*y^4-y^6)*1 + (192-96*y^2+12*y^4)*x - (192-48*y^2)*x^2 + 60*x^3
sage: MW.frob_Q(11) is MW.frob_Q(11)
True
```

frob_basis_elements(*prec*, *p*)

Return the action of a p-power lift of Frobenius on the basis.

$$\{dx/2y, xdx/2y, ..., x^{d-2}dx/2y\},$$

where d is the degree of the underlying hyperelliptic curve.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: C = HyperellipticCurve(x^5 - 4*x + 4)
sage: prec = 1
sage: p = 5
sage: MW = C.invariant_differential().parent()
sage: MW.frob_basis_elements(prec, p)
[((92000*y^-14-74200*y^-12+32000*y^-10-8000*y^-8+1000*y^-6-50*y^-4)*1
  - (194400*y^-14-153600*y^-12+57600*y^-10-9600*y^-8+600*y^-6)*x
  + (204800*y^-14-153600*y^-12+38400*y^-10-3200*y^-8)*x^2
```

(continues on next page)

(continued from previous page)

```
  - (153600*y^-14-76800*y^-12+9600*y^-10)*x^3
  + (63950*y^-14-18550*y^-12+1600*y^-10-400*y^-8+50*y^-6+5*y^-4)*x^4) dx/2y,
 (-(1391200*y^-14-941400*y^-12+302000*y^-10-76800*y^-8+14400*y^-6-1320*y^-
↪4+30*y^-2)*1
  + (2168800*y^-14-1402400*y^-12+537600*y^-10-134400*y^-8+16800*y^-6-720*y^-
↪4)*x
  - (1596800*y^-14-1433600*y^-12+537600*y^-10-89600*y^-8+5600*y^-6)*x^2
  + (1433600*y^-14-1075200*y^-12+268800*y^-10-22400*y^-8)*x^3
  - (870200*y^-14-445350*y^-12+63350*y^-10-3200*y^-8+600*y^-6-30*y^-4-5*y^-
↪2)*x^4) dx/2y,
 ((19488000*y^-14-15763200*y^-12+4944400*y^-10-913800*y^-8+156800*y^-6-
↪22560*y^-4+1480*y^-2-10)*1
  - (28163200*y^-14-18669600*y^-12+5774400*y^-10-1433600*y^-8+268800*y^-6-
↪25440*y^-4+760*y^-2)*x
  + (15062400*y^-14-12940800*y^-12+5734400*y^-10-1433600*y^-8+179200*y^-6-
↪8480*y^-4)*x^2
  - (12121600*y^-14-11468800*y^-12+4300800*y^-10-716800*y^-8+44800*y^-6)*x^3
  + (9215200*y^-14-6952400*y^-12+1773950*y^-10-165750*y^-8+5600*y^-6-720*y^-
↪4+10*y^-2+5)*x^4) dx/2y,
 (-(225395200*y^-14-230640000*y^-12+91733600*y^-10-18347400*y^-8+2293600*y^-6-
↪280960*y^-4+31520*y^-2-1480-10*y^2)*1
  + (338048000*y^-14-277132800*y^-12+89928000*y^-10-17816000*y^-8+3225600*y^-
↪6-472320*y^-4+34560*y^-2-720)*x
  - (172902400*y^-14-141504000*y^-12+58976000*y^-10-17203200*y^-8+3225600*y^-
↪6-314880*y^-4+11520*y^-2)*x^2
  + (108736000*y^-14-109760000*y^-12+51609600*y^-10-12902400*y^-8+1612800*y^-
↪6-78720*y^-4)*x^3
  - (85347200*y^-14-82900000*y^-12+31251400*y^-10-5304150*y^-8+367350*y^-6-
↪8480*y^-4+760*y^-2+10-5*y^2)*x^4) dx/2y]
```

frob_invariant_differential (*prec*, *p*)

Kedlaya's algorithm allows us to calculate the action of Frobenius on the Monsky-Washnitzer cohomology. First we lift ϕ to $A^\dagger$ by setting

$$\phi(x) = x^p, \qquad \phi(y) = y^p\sqrt{1 + \frac{Q(x^p) - Q(x)^p}{Q(x)^p}}.$$

Pulling back the differential $dx/2y$, we get

$$\phi^*(dx/2y) = px^{p-1}y(\phi(y))^{-1}dx/2y = px^{p-1}y^{1-p}\sqrt{1 + \frac{Q(x^p) - Q(x)^p}{Q(x)^p}}dx/2y.$$

Use Newton's method to calculate the square root.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: C = HyperellipticCurve(x^5 - 4*x + 4)
sage: prec = 2
sage: p = 7
sage: MW = C.invariant_differential().parent()
sage: MW.frob_invariant_differential(prec, p)
((67894400*y^-20-81198880*y^-18+40140800*y^-16-10035200*y^-14+1254400*y^-12-
↪62720*y^-10)*1
 - (119503944*y^-20-116064242*y^-18+43753472*y^-16-7426048*y^-14+514304*y^-12-
↪12544*y^-10+1568*y^-8-70*y^-6-7*y^-4)*x
```

(continues on next page)

(continued from previous page)

```
 + (78905288*y^-20-61014016*y^-18+16859136*y^-16-2207744*y^-14+250880*y^-12-
↪37632*y^-10+3136*y^-8-70*y^-6)*x^2
 - (39452448*y^-20-26148752*y^-18+8085490*y^-16-2007040*y^-14+376320*y^-12-
↪37632*y^-10+1568*y^-8)*x^3
 + (21102144*y^-20-18120592*y^-18+8028160*y^-16-2007040*y^-14+250880*y^-12-
↪12544*y^-10)*x^4) dx/2y
```

helper_matrix()

We use this to solve for the linear combination of $x^i y^j$ needed to clear all terms with y^{j-1}.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: C = HyperellipticCurve(x^5 - 4*x + 4)
sage: MW = C.invariant_differential().parent()
sage: MW.helper_matrix()
[ 256/2101  320/2101  400/2101  500/2101  625/2101]
[-625/8404  -64/2101  -80/2101 -100/2101 -125/2101]
[-125/2101 -625/8404  -64/2101  -80/2101 -100/2101]
[-100/2101 -125/2101 -625/8404  -64/2101  -80/2101]
[ -80/2101 -100/2101 -125/2101 -625/8404  -64/2101]
```

invariant_differential()

Return $dx/2y$ as an element of `self`.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: C = HyperellipticCurve(x^5 - 4*x + 4)
sage: MW = C.invariant_differential().parent()
sage: MW.invariant_differential()
1 dx/2y
```

x_to_p(p)

Return and cache x^p, reduced via the relations coming from the defining polynomial of the hyperelliptic curve.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: C = HyperellipticCurve(x^5 - 4*x + 4)
sage: MW = C.invariant_differential().parent()
sage: MW.x_to_p(3)
x^3
sage: MW.x_to_p(5)
-(4-y^2)*1 + 4*x
sage: MW.x_to_p(101) is MW.x_to_p(101)
True
```

sage.schemes.hyperelliptic_curves.monsky_washnitzer.
MonskyWashnitzerDifferentialRing_class

alias of *MonskyWashnitzerDifferentialRing*

class sage.schemes.hyperelliptic_curves.monsky_washnitzer.**SpecialCubicQuotientRing**(*Q*, *laurent_series=False*)

Bases: UniqueRepresentation, Parent

Specialised class for representing the quotient ring $R[x,T]/(T-x^3-ax-b)$, where R is an arbitrary commutative base ring (in which 2 and 3 are invertible), a and b are elements of that ring.

Polynomials are represented internally in the form $p_0+p_1x+p_2x^2$ where the p_i are polynomials in T. Multiplication of polynomials always reduces high powers of x (i.e. beyond x^2) to powers of T.

Hopefully this ring is faster than a general quotient ring because it uses the special structure of this ring to speed multiplication (which is the dominant operation in the frobenius matrix calculation). I haven't actually tested this theory though…

Todo: Eventually we will want to run this in characteristic 3, so we need to: (a) Allow $Q(x)$ to contain an x^2 term, and (b) Remove the requirement that 3 be invertible. Currently this is used in the Toom-Cook algorithm to speed multiplication.

EXAMPLES:

```
sage: B.<t> = PolynomialRing(Integers(125))
sage: R = monsky_washnitzer.SpecialCubicQuotientRing(t^3 - t + B(1/4))
sage: R
SpecialCubicQuotientRing over Ring of integers modulo 125
with polynomial T = x^3 + 124*x + 94
sage: TestSuite(R).run()
```

Get generators:

```
sage: x, T = R.gens()
sage: x
(0) + (1)*x + (0)*x^2
sage: T
(T) + (0)*x + (0)*x^2
```

Coercions:

```
sage: R(7)
(7) + (0)*x + (0)*x^2
```

Create elements directly from polynomials:

```
sage: A = R.poly_ring()
sage: A
Univariate Polynomial Ring in T over Ring of integers modulo 125
sage: z = A.gen()
sage: R.create_element(z^2, z+1, 3)
(T^2) + (T + 1)*x + (3)*x^2
```

Some arithmetic:

```
sage: x^3
(T + 31) + (1)*x + (0)*x^2
sage: 3 * x**15 * T**2 + x - T
(3*T^7 + 90*T^6 + 110*T^5 + 20*T^4 + 58*T^3 + 26*T^2 + 124*T) +
(15*T^6 + 110*T^5 + 35*T^4 + 63*T^2 + 1)*x +
(30*T^5 + 40*T^4 + 8*T^3 + 38*T^2)*x^2
```

Retrieve coefficients (output is zero-padded):

```
sage: x^10
(3*T^2 + 61*T + 8) + (T^3 + 93*T^2 + 12*T + 40)*x + (3*T^2 + 61*T + 9)*x^2
sage: (x^10).coeffs()
[[8, 61, 3, 0], [40, 12, 93, 1], [9, 61, 3, 0]]
```

Todo: write an example checking multiplication of these polynomials against Sage's ordinary quotient ring arithmetic. I cannot seem to get the quotient ring stuff happening right now…

Element

alias of *SpecialCubicQuotientRingElement*

create_element(*check*, **args*)

Create the element $p_0 + p_1 * x + p_2 * x^2$, where the p_i are polynomials in T.

INPUT:

- p0, p1, p2 – coefficients; must be coercible into poly_ring()
- check – bool (default: True): whether to carry out coercion

EXAMPLES:

```
sage: B.<t> = PolynomialRing(Integers(125))
sage: R = monsky_washnitzer.SpecialCubicQuotientRing(t^3 - t + B(1/4))
sage: A, z = R.poly_ring().objgen()
sage: R.create_element(z^2, z+1, 3)  # indirect doctest
(T^2) + (T + 1)*x + (3)*x^2
```

gens()

Return a list [x, T] where x and T are the generators of the ring (as element *of this ring*).

Note: I have no idea if this is compatible with the usual Sage 'gens' interface.

EXAMPLES:

```
sage: B.<t> = PolynomialRing(Integers(125))
sage: R = monsky_washnitzer.SpecialCubicQuotientRing(t^3 - t + B(1/4))
sage: x, T = R.gens()
sage: x
(0) + (1)*x + (0)*x^2
sage: T
(T) + (0)*x + (0)*x^2
```

one()

Return the unit of self.

EXAMPLES:

```
sage: B.<t> = PolynomialRing(Integers(125))
sage: R = monsky_washnitzer.SpecialCubicQuotientRing(t^3 - t + B(1/4))
sage: R.one()
(1) + (0)*x + (0)*x^2
```

poly_ring()

Return the underlying polynomial ring in T.

EXAMPLES:

```
sage: B.<t> = PolynomialRing(Integers(125))
sage: R = monsky_washnitzer.SpecialCubicQuotientRing(t^3 - t + B(1/4))
sage: R.poly_ring()
Univariate Polynomial Ring in T over Ring of integers modulo 125
```

class sage.schemes.hyperelliptic_curves.monsky_washnitzer.**SpecialCubicQuotientRingElement**(

Bases: ModuleElement

An element of a *SpecialCubicQuotientRing*.

coeffs()

Return list of three lists of coefficients, corresponding to the x^0, x^1, x^2 coefficients.

The lists are zero padded to the same length. The list entries belong to the base ring.

EXAMPLES:

```
sage: B.<t> = PolynomialRing(Integers(125))
sage: R = monsky_washnitzer.SpecialCubicQuotientRing(t^3 - t + B(1/4))
sage: p = R.create_element(t, t^2 - 2, 3)
sage: p.coeffs()
[[0, 1, 0], [123, 0, 1], [3, 0, 0]]
```

scalar_multiply(*scalar*)

Multiply this element by a scalar, i.e. just multiply each coefficient of x^j by the scalar.

INPUT:

- scalar – either an element of base_ring, or an element of poly_ring.

EXAMPLES:

```
sage: B.<t> = PolynomialRing(Integers(125))
sage: R = monsky_washnitzer.SpecialCubicQuotientRing(t^3 - t + B(1/4))
sage: x, T = R.gens()
sage: f = R.create_element(2, t, t^2 - 3)
sage: f
(2) + (T)*x + (T^2 + 122)*x^2
sage: f.scalar_multiply(2)
(4) + (2*T)*x + (2*T^2 + 119)*x^2
sage: f.scalar_multiply(t)
(2*T) + (T^2)*x + (T^3 + 122*T)*x^2
```

shift(*n*)

Return this element multiplied by T^n.

EXAMPLES:

```
sage: B.<t> = PolynomialRing(Integers(125))
sage: R = monsky_washnitzer.SpecialCubicQuotientRing(t^3 - t + B(1/4))
sage: f = R.create_element(2, t, t^2 - 3)
sage: f
(2) + (T)*x + (T^2 + 122)*x^2
sage: f.shift(1)
(2*T) + (T^2)*x + (T^3 + 122*T)*x^2
sage: f.shift(2)
(2*T^2) + (T^3)*x + (T^4 + 122*T^2)*x^2
```

square()

Return the square of the element.

EXAMPLES:

```
sage: B.<t> = PolynomialRing(Integers(125))
sage: R = monsky_washnitzer.SpecialCubicQuotientRing(t^3 - t + B(1/4))
sage: x, T = R.gens()
```

```
sage: f = R.create_element(1 + 2*t + 3*t^2, 4 + 7*t + 9*t^2, 3 + 5*t + 11*t^2)
sage: f.square()
(73*T^5 + 16*T^4 + 38*T^3 + 39*T^2 + 70*T + 120)
+ (121*T^5 + 113*T^4 + 73*T^3 + 8*T^2 + 51*T + 61)*x
+ (18*T^4 + 60*T^3 + 22*T^2 + 108*T + 31)*x^2
```

class sage.schemes.hyperelliptic_curves.monsky_washnitzer.**SpecialHyperellipticQuotientEleme**

Bases: ModuleElement

Element in the Hyperelliptic quotient ring.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^5 - 36*x + 1)
sage: x,y = E.monsky_washnitzer_gens()
sage: MW = x.parent()
sage: MW(x + x**2 + y - 77)
-(77-y)*1 + x + x^2
```

change_ring(*R*)

Return the same element after changing the base ring to R.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^5 - 36*x + 1)
sage: x,y = E.monsky_washnitzer_gens()
sage: MW = x.parent()
sage: z = MW(x + x**2 + y - 77)
sage: z.change_ring(AA).parent()                                                #␣
↪needs sage.rings.number_field
```

(continues on next page)

(continued from previous page)

```
SpecialHyperellipticQuotientRing K[x,y,y^-1] / (y^2 = x^5 - 36*x + 1)
over Algebraic Real Field
```

coeffs (*R=None*)

Return the raw coefficients of this element.

INPUT:

- `R` – an (optional) base-ring in which to cast the coefficients

OUTPUT:

- `coeffs` – a list of coefficients of powers of x for each power of y
- `n` – an offset indicating the power of y of the first list element

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^5 - 3*x + 1)
sage: x,y = E.monsky_washnitzer_gens()
sage: x.coeffs()
([(0, 1, 0, 0, 0)], 0)
sage: y.coeffs()
([(0, 0, 0, 0, 0), (1, 0, 0, 0, 0)], 0)

sage: a = sum(n*x^n for n in range(5)); a
x + 2*x^2 + 3*x^3 + 4*x^4
sage: a.coeffs()
([(0, 1, 2, 3, 4)], 0)
sage: a.coeffs(Qp(7))                                                    #␣
↪needs sage.rings.padics
([(0, 1 + O(7^20), 2 + O(7^20), 3 + O(7^20), 4 + O(7^20))], 0)
sage: (a*y).coeffs()
([(0, 0, 0, 0, 0), (0, 1, 2, 3, 4)], 0)
sage: (a*y^-2).coeffs()
([(0, 1, 2, 3, 4), (0, 0, 0, 0, 0), (0, 0, 0, 0, 0)], -2)
```

Note that the coefficient list is transposed compared to how they are stored and printed:

```
sage: a*y^-2
(y^-2)*x + (2*y^-2)*x^2 + (3*y^-2)*x^3 + (4*y^-2)*x^4
```

A more complicated example:

```
sage: a = x^20*y^-3 - x^11*y^2; a
(y^-3-4*y^-1+6*y-4*y^3+y^5)*1 - (12*y^-3-36*y^-1+36*y+y^2-12*y^3-2*y^4+y^6)*x
 + (54*y^-3-108*y^-1+54*y+6*y^2-6*y^4)*x^2 - (108*y^-3-108*y^-1+9*y^2)*x^3
 + (81*y^-3)*x^4
sage: raw, offset = a.coeffs()
sage: a.min_pow_y()
-3
sage: offset
-3
sage: raw
[(1, -12, 54, -108, 81),
 (0, 0, 0, 0, 0),
 (-4, 36, -108, 108, 0),
```

(continues on next page)

(continued from previous page)

```
 (0, 0, 0, 0, 0),
 (6, -36, 54, 0, 0),
 (0, -1, 6, -9, 0),
 (-4, 12, 0, 0, 0),
 (0, 2, -6, 0, 0),
 (1, 0, 0, 0, 0),
 (0, -1, 0, 0, 0)]
sage: sum(c * x^i * y^(j+offset)
....:     for j, L in enumerate(raw) for i, c in enumerate(L)) == a
True
```

Can also be used to construct elements:

```
sage: a.parent()(raw, offset) == a
True
```

diff()

Return the differential of `self`.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^5 - 3*x + 1)
sage: x,y = E.monsky_washnitzer_gens()
sage: (x + 3*y).diff()
(-(9-2*y)*1 + 15*x^4) dx/2y
```

extract_pow_y(k)

Return the coefficients of y^k in `self` as a list.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^5 - 3*x + 1)
sage: x,y = E.monsky_washnitzer_gens()
sage: (x + 3*y + 9*x*y).extract_pow_y(1)
[3, 9, 0, 0, 0]
```

max_pow_y()

Return the maximal degree of `self` with respect to y.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^5 - 3*x + 1)
sage: x,y = E.monsky_washnitzer_gens()
sage: (x + 3*y).max_pow_y()
1
```

min_pow_y()

Return the minimal degree of `self` with respect to y.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^5 - 3*x + 1)
```

(continues on next page)

(continued from previous page)

```
sage: x,y = E.monsky_washnitzer_gens()
sage: (x + 3*y).min_pow_y()
0
```

truncate_neg(*n*)

Return `self` minus its terms of degree less than n wrt y.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^5 - 3*x + 1)
sage: x,y = E.monsky_washnitzer_gens()
sage: (x + 3*y + 7*x*2*y**4).truncate_neg(1)
3*y*1 + 14*y^4*x
```

class sage.schemes.hyperelliptic_curves.monsky_washnitzer.**SpecialHyperellipticQuotientRing**

Bases: `UniqueRepresentation`, `Parent`

The special hyperelliptic quotient ring.

Element

alias of *SpecialHyperellipticQuotientElement*

Q()

Return the defining polynomial of the underlying hyperelliptic curve.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^5-2*x+1)
sage: x,y = E.monsky_washnitzer_gens()
sage: x.parent().Q()
x^5 - 2*x + 1
```

base_extend(*R*)

Return the base extension of `self` to the ring `R` if possible.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^5 - 3*x + 1)
sage: x,y = E.monsky_washnitzer_gens()
sage: x.parent().base_extend(UniversalCyclotomicField())                    #␣
↪needs sage.libs.gap
SpecialHyperellipticQuotientRing K[x,y,y^-1] / (y^2 = x^5 - 3*x + 1)
over Universal Cyclotomic Field
sage: x.parent().base_extend(ZZ)
Traceback (most recent call last):
...
TypeError: no such base extension
```

change_ring(*R*)

Return the analog of `self` over the ring `R`.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^5 - 3*x + 1)
sage: x,y = E.monsky_washnitzer_gens()
sage: x.parent().change_ring(ZZ)
SpecialHyperellipticQuotientRing K[x,y,y^-1] / (y^2 = x^5 - 3*x + 1)
over Integer Ring
```

curve()

Return the underlying hyperelliptic curve.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^5 - 3*x + 1)
sage: x,y = E.monsky_washnitzer_gens()
sage: x.parent().curve()
Hyperelliptic Curve over Rational Field defined by y^2 = x^5 - 3*x + 1
```

degree()

Return the degree of the underlying hyperelliptic curve.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^5 - 3*x + 1)
sage: x,y = E.monsky_washnitzer_gens()
sage: x.parent().degree()
5
```

gens()

Return the generators of `self`

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^5 - 3*x + 1)
sage: x,y = E.monsky_washnitzer_gens()
sage: x.parent().gens()
(x, y*1)
```

is_field(*proof=True*)

Return `False` as `self` is not a field.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^5 - 3*x + 1)
sage: x,y = E.monsky_washnitzer_gens()
sage: x.parent().is_field()
False
```

monomial(*i*, *j*, *b=None*)

Return $by^j x^i$, computed quickly.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^5 - 3*x + 1)
sage: x,y = E.monsky_washnitzer_gens()
sage: x.parent().monomial(4,5)
y^5*x^4
```

monomial_diff_coeffs(*i*, *j*)

Compute coefficients of the basis representation of $d(x^i y^j)$.

The key here is that the formula for $d(x^i y^j)$ is messy in terms of i, but varies nicely with j.

$$d(x^i y^j) = y^{j-1}(2ix^{i-1}y^2 + j(A_i(x) + B_i(x)y^2))\frac{dx}{2y},$$

where A, B have degree at most $n-1$ for each i. Pre-compute A_i, B_i for each i the "hard" way, and the rest are easy.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^5 - 3*x + 1)
sage: x,y = E.monsky_washnitzer_gens()
sage: x.parent().monomial_diff_coeffs(2,3)
((0, -15, 36, 0, 0), (0, 19, 0, 0, 0))
```

monomial_diff_coeffs_matrices()

Compute tables of coefficients of the basis representation of $d(x^i y^j)$ for small i, j.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^5 - 3*x + 1)
sage: x,y = E.monsky_washnitzer_gens()
sage: x.parent().monomial_diff_coeffs_matrices()
(
[0 5 0 0 0]  [0 2 0 0 0]
[0 0 5 0 0]  [0 0 4 0 0]
[0 0 0 5 0]  [0 0 0 6 0]
[0 0 0 0 5]  [0 0 0 0 8]
[0 0 0 0 0], [0 0 0 0 0]
)
```

monsky_washnitzer()

Return the stored Monsky-Washnitzer differential ring.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^5 - 3*x + 1)
sage: x,y = E.monsky_washnitzer_gens()
sage: type(x.parent().monsky_washnitzer())
<class 'sage.schemes.hyperelliptic_curves.monsky_washnitzer.
↪MonskyWashnitzerDifferentialRing_with_category'>
```

one()

Return the unit of `self`.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^5 - 3*x + 1)
sage: x,y = E.monsky_washnitzer_gens()
sage: x.parent().one()
1
```

prime()

Return the stored prime number p.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^5 - 3*x + 1)
sage: x,y = E.monsky_washnitzer_gens()
sage: x.parent().prime() is None
True
```

x()

Return the generator x of `self`

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^5 - 3*x + 1)
sage: x,y = E.monsky_washnitzer_gens()
sage: x.parent().x()
x
```

y()

Return the generator y of `self`

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^5 - 3*x + 1)
sage: x,y = E.monsky_washnitzer_gens()
sage: x.parent().y()
y*1
```

zero()

Return the zero of `self`.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^5 - 3*x + 1)
sage: x,y = E.monsky_washnitzer_gens()
sage: x.parent().zero()
0
```

sage.schemes.hyperelliptic_curves.monsky_washnitzer.
SpecialHyperellipticQuotientRing_class

alias of *SpecialHyperellipticQuotientRing*

sage.schemes.hyperelliptic_curves.monsky_washnitzer.**adjusted_prec**(*p*, *prec*)

Compute how much precision is required in `matrix_of_frobenius` to get an answer correct to `prec` p-adic digits.

The issue is that the algorithm used in *matrix_of_frobenius()* sometimes performs divisions by p, so precision is lost during the algorithm.

The estimate returned by this function is based on Kedlaya's result (Lemmas 2 and 3 of [Ked2001]), which implies that if we start with M p-adic digits, the total precision loss is at most $1 + \lfloor \log_p(2M-3) \rfloor$ p-adic digits. (This estimate is somewhat less than the amount you would expect by naively counting the number of divisions by p.)

INPUT:

- `p` – a prime `p >= 5`
- `prec` – integer, desired output precision, `prec >= 1`

OUTPUT: adjusted precision (usually slightly more than `prec`)

EXAMPLES:

```
sage: from sage.schemes.hyperelliptic_curves.monsky_washnitzer import adjusted_
↪prec
sage: adjusted_prec(5,2)
3
```

sage.schemes.hyperelliptic_curves.monsky_washnitzer.**frobenius_expansion_by_newton**(*Q*, *p*, *M*)

Compute the action of Frobenius on dx/y and on xdx/y, using Newton's method (as suggested in Kedlaya's paper [Ked2001]).

(This function does *not* yet use the cohomology relations - that happens afterwards in the "reduction" step.)

More specifically, it finds F_0 and F_1 in the quotient ring $R[x,T]/(T-Q(x))$, such that

$$F(dx/y) = T^{-r} F0 dx/y, \text{ and } F(xdx/y) = T^{-r} F1 dx/y$$

where

$$r = ((2M-3)p-1)/2.$$

(Here T is $y^2 = z^{-2}$, and R is the coefficient ring of Q.)

F_0 and F_1 are computed in the *SpecialCubicQuotientRing* associated to Q, so all powers of x^j for $j \geq 3$ are reduced to powers of T.

INPUT:

- `Q` – cubic polynomial of the form $Q(x) = x^3 + ax + b$, whose coefficient ring is a $Z/(p^M)Z$-algebra
- `p` – residue characteristic of the p-adic field
- `M` – p-adic precision of the coefficient ring (this will be used to determine the number of Newton iterations)

OUTPUT:

- `F0, F1` – elements of `SpecialCubicQuotientRing(Q)`, as described above
- `r` – non-negative integer, as described above

EXAMPLES:

```
sage: from sage.schemes.hyperelliptic_curves.monsky_washnitzer import frobenius_
↪expansion_by_newton
sage: R.<x> = Integers(5^3)['x']
sage: Q = x^3 - x + R(1/4)
```

(continues on next page)

(continued from previous page)

```
sage: frobenius_expansion_by_newton(Q,5,3)
((25*T^5 + 75*T^3 + 100*T^2 + 100*T + 100) + (5*T^6 + 80*T^5 + 100*T^3
+ 25*T + 50)*x + (55*T^5 + 50*T^4 + 75*T^3 + 25*T^2 + 25*T + 25)*x^2,
(5*T^8 + 15*T^7 + 95*T^6 + 10*T^5 + 25*T^4 + 25*T^3 + 100*T^2 + 50)
+ (65*T^7 + 55*T^6 + 70*T^5 + 100*T^4 + 25*T^2 + 100*T)*x
+ (15*T^6 + 115*T^5 + 75*T^4 + 100*T^3 + 50*T^2 + 75*T + 75)*x^2, 7)
```

sage.schemes.hyperelliptic_curves.monsky_washnitzer.**frobenius_expansion_by_series**(*Q*, *p*, *M*)

Compute the action of Frobenius on dx/y and on xdx/y, using a series expansion.

(This function computes the same thing as frobenius_expansion_by_newton(), using a different method. Theoretically the Newton method should be asymptotically faster, when the precision gets large. However, in practice, this functions seems to be marginally faster for moderate precision, so I'm keeping it here until I figure out exactly why it is faster.)

(This function does *not* yet use the cohomology relations - that happens afterwards in the "reduction" step.)

More specifically, it finds F0 and F1 in the quotient ring $R[x,T]/(T-Q(x))$, such that $F(dx/y) = T^{-r}F0dx/y$, and $F(xdx/y) = T^{-r}F1dx/y$ where $r = ((2M-3)p-1)/2$. (Here T is $y^2 = z^{-2}$, and R is the coefficient ring of Q.)

F_0 and F_1 are computed in the *SpecialCubicQuotientRing* associated to Q, so all powers of x^j for $j \geq 3$ are reduced to powers of T.

It uses the sum

$$F0 = \sum_{k=0}^{M-2} \binom{-1/2}{k} px^{p-1}E^kT^{(M-2-k)p}$$

and

$$F1 = x^pF0,$$
$$where`E = Q(x^p) - Q(x)^p`.$$

INPUT:

- `Q` – cubic polynomial of the form $Q(x) = x^3 + ax + b$, whose coefficient ring is a $\mathbf{Z}/(p^M)\mathbf{Z}$ -algebra
- `p` – residue characteristic of the p-adic field
- `M` – p-adic precision of the coefficient ring (this will be used to determine the number of terms in the series)

OUTPUT:

- `F0`, `F1` – elements of `SpecialCubicQuotientRing(Q)`, as described above
- `r` – non-negative integer, as described above

EXAMPLES:

```
sage: from sage.schemes.hyperelliptic_curves.monsky_washnitzer import frobenius_
↪expansion_by_series
sage: R.<x> = Integers(5^3)['x']
sage: Q = x^3 - x + R(1/4)
sage: frobenius_expansion_by_series(Q,5,3)                                    #␣
↪needs sage.libs.pari
((25*T^5 + 75*T^3 + 100*T^2 + 100*T + 100) + (5*T^6 + 80*T^5 + 100*T^3
+ 25*T + 50)*x + (55*T^5 + 50*T^4 + 75*T^3 + 25*T^2 + 25*T + 25)*x^2,
```

(continues on next page)

(continued from previous page)

```
(5*T^8 + 15*T^7 + 95*T^6 + 10*T^5 + 25*T^4 + 25*T^3 + 100*T^2 + 50)
+ (65*T^7 + 55*T^6 + 70*T^5 + 100*T^4 + 25*T^2 + 100*T)*x
+ (15*T^6 + 115*T^5 + 75*T^4 + 100*T^3 + 50*T^2 + 75*T + 75)*x^2, 7)
```

sage.schemes.hyperelliptic_curves.monsky_washnitzer.**helper_matrix**(*Q*)

Compute the (constant) matrix used to calculate the linear combinations of the $d(x^i y^j)$ needed to eliminate the negative powers of y in the cohomology (i.e., in *reduce_negative()*).

INPUT:

- `Q` – cubic polynomial

EXAMPLES:

```
sage: t = polygen(QQ,'t')
sage: from sage.schemes.hyperelliptic_curves.monsky_washnitzer import helper_
↪matrix
sage: helper_matrix(t**3-4*t-691)
[     64/12891731  -16584/12891731 4297329/12891731]
[   6219/12891731     -32/12891731    8292/12891731]
[    -24/12891731    6219/12891731     -32/12891731]
```

sage.schemes.hyperelliptic_curves.monsky_washnitzer.**lift**(*x*)

Try to call `x.lift()`, presumably from the p-adics to $\mathbf{Z}$.

If this fails, it assumes the input is a power series, and tries to lift it to a power series over $\mathbf{Q}$.

This function is just a very kludgy solution to the problem of trying to make the reduction code (below) work over both $\mathbf{Z}_p$ and $\mathbf{Z}_p[[t]]$.

EXAMPLES:

```
sage: # needs sage.rings.padics
sage: from sage.schemes.hyperelliptic_curves.monsky_washnitzer import lift
sage: l = lift(Qp(13)(131)); l
131
sage: l.parent()
Integer Ring
sage: x = PowerSeriesRing(Qp(17),'x').gen()
sage: l = lift(4 + 5*x + 17*x**6); l
4 + 5*t + 17*t^6
sage: l.parent()
Power Series Ring in t over Rational Field
```

sage.schemes.hyperelliptic_curves.monsky_washnitzer.**matrix_of_frobenius**(*Q*, *p*, *M*, *trace=None*, *compute_exact_forms=False*)

Compute the matrix of Frobenius on Monsky-Washnitzer cohomology, with respect to the basis $(dx/y, xdx/y)$.

INPUT:

- `Q` – cubic polynomial $Q(x) = x^3 + ax + b$ defining an elliptic curve E by $y^2 = Q(x)$. The coefficient ring of Q should be a $\mathbf{Z}/(p^M)\mathbf{Z}$-algebra in which the matrix of frobenius will be constructed.
- `p` – prime >= 5 for which E has good reduction
- `M` – integer >= 2; p -adic precision of the coefficient ring

- `trace` – (optional) the trace of the matrix, if known in advance. This is easy to compute because it is just the a_p of the curve. If the trace is supplied, matrix_of_frobenius will use it to speed the computation (i.e. we know the determinant is p, so we have two conditions, so really only column of the matrix needs to be computed. it is actually a little more complicated than that, but that's the basic idea.) If trace=None, then both columns will be computed independently, and you can get a strong indication of correctness by verifying the trace afterwards.

> **Warning:** THE RESULT WILL NOT NECESSARILY BE CORRECT TO M p-ADIC DIGITS. If you want prec digits of precision, you need to use the function adjusted_prec(), and then you need to reduce the answer mod p^{prec} at the end.

OUTPUT:

2×2 matrix of Frobenius acting on Monsky-Washnitzer cohomology, with entries in the coefficient ring of `Q`.

EXAMPLES:

A simple example:

```
sage: p = 5
sage: prec = 3
sage: M = monsky_washnitzer.adjusted_prec(p, prec); M
4
sage: R.<x> = PolynomialRing(Integers(p**M))
sage: A = monsky_washnitzer.matrix_of_frobenius(x^3 - x + R(1/4), p, M)
sage: A
[340  62]
[ 70 533]
```

But the result is only accurate to `prec` digits:

```
sage: B = A.change_ring(Integers(p**prec))
sage: B
[90 62]
[70 33]
```

Check trace (123 = -2 mod 125) and determinant:

```
sage: B.det()
5
sage: B.trace()
123
sage: EllipticCurve([-1, 1/4]).ap(5)
-2
```

Try using the trace to speed up the calculation:

```
sage: A = monsky_washnitzer.matrix_of_frobenius(x^3 - x + R(1/4),
....:                                           p, M, -2)
sage: A
[ 90  62]
[320 533]
```

Hmmm… it looks different, but that's because the trace of our first answer was only -2 modulo 5^3, not -2 modulo 5^5. So the right answer is:

```
sage: A.change_ring(Integers(p**prec))
[90 62]
[70 33]
```

Check it works with only one digit of precision:

```
sage: p = 5
sage: prec = 1
sage: M = monsky_washnitzer.adjusted_prec(p, prec)
sage: R.<x> = PolynomialRing(Integers(p**M))
sage: A = monsky_washnitzer.matrix_of_frobenius(x^3 - x + R(1/4), p, M)
sage: A.change_ring(Integers(p))
[0 2]
[0 3]
```

Here is an example that is particularly badly conditioned for using the trace trick:

```
sage: # needs sage.libs.pari
sage: p = 11
sage: prec = 3
sage: M = monsky_washnitzer.adjusted_prec(p, prec)
sage: R.<x> = PolynomialRing(Integers(p**M))
sage: A = monsky_washnitzer.matrix_of_frobenius(x^3 + 7*x + 8, p, M)
sage: A.change_ring(Integers(p**prec))
[1144  176]
[ 847  185]
```

The problem here is that the top-right entry is divisible by 11, and the bottom-left entry is divisible by 11^2. So when you apply the trace trick, neither $F(dx/y)$ nor $F(xdx/y)$ is enough to compute the whole matrix to the desired precision, even if you try increasing the target precision by one. Nevertheless, `matrix_of_frobenius` knows how to get the right answer by evaluating $F((x+1)dx/y)$ instead:

```
sage: A = monsky_washnitzer.matrix_of_frobenius(x^3 + 7*x + 8, p, M, -2)
sage: A.change_ring(Integers(p**prec))
[1144  176]
[ 847  185]
```

The running time is about `O(p*prec**2)` (times some logarithmic factors), so it is feasible to run on fairly large primes, or precision (or both?!?!):

```
sage: # long time, needs sage.libs.pari
sage: p = 10007
sage: prec = 2
sage: M = monsky_washnitzer.adjusted_prec(p, prec)
sage: R.<x> = PolynomialRing(Integers(p**M))
sage: A = monsky_washnitzer.matrix_of_frobenius(x^3 - x + R(1/4), p, M)
sage: B = A.change_ring(Integers(p**prec)); B
[74311982 57996908]
[95877067 25828133]
sage: B.det()
10007
sage: B.trace()
66
sage: EllipticCurve([-1, 1/4]).ap(10007)
66
```

```
sage: # long time, needs sage.libs.pari
sage: p = 5
sage: prec = 300
sage: M = monsky_washnitzer.adjusted_prec(p, prec)
sage: R.<x> = PolynomialRing(Integers(p**M))
sage: A = monsky_washnitzer.matrix_of_frobenius(x^3 - x + R(1/4), p, M)
sage: B = A.change_ring(Integers(p**prec))
sage: B.det()
5
sage: -B.trace()
2
sage: EllipticCurve([-1, 1/4]).ap(5)
-2
```

Let us check consistency of the results for a range of precisions:

```
sage: # long time, needs sage.libs.pari
sage: p = 5
sage: max_prec = 60
sage: M = monsky_washnitzer.adjusted_prec(p, max_prec)
sage: R.<x> = PolynomialRing(Integers(p**M))
sage: A = monsky_washnitzer.matrix_of_frobenius(x^3 - x + R(1/4), p, M)
sage: A = A.change_ring(Integers(p**max_prec))
sage: result = []
sage: for prec in range(1, max_prec):
....:     M = monsky_washnitzer.adjusted_prec(p, prec)
....:     R.<x> = PolynomialRing(Integers(p^M),'x')
....:     B = monsky_washnitzer.matrix_of_frobenius(x^3 - x + R(1/4), p, M)
....:     B = B.change_ring(Integers(p**prec))
....:     result.append(B == A.change_ring(Integers(p**prec)))
sage: result == [True] * (max_prec - 1)
True
```

The remaining examples discuss what happens when you take the coefficient ring to be a power series ring; i.e. in effect you're looking at a family of curves.

The code does in fact work…

```
sage: # needs sage.libs.pari
sage: p = 11
sage: prec = 3
sage: M = monsky_washnitzer.adjusted_prec(p, prec)
sage: S.<t> = PowerSeriesRing(Integers(p**M), default_prec=4)
sage: a = 7 + t + 3*t^2
sage: b = 8 - 6*t + 17*t^2
sage: R.<x> = PolynomialRing(S)
sage: Q = x**3 + a*x + b
sage: A = monsky_washnitzer.matrix_of_frobenius(Q, p, M)             # long time
sage: B = A.change_ring(PowerSeriesRing(Integers(p**prec), 't',      # long time
....:                                   default_prec=4)); B
[1144 + 264*t + 841*t^2 + 1025*t^3 + O(t^4)  176 + 1052*t + 216*t^2 + 523*t^3 +␣
↪O(t^4)]
[   847 + 668*t + 81*t^2 + 424*t^3 + O(t^4)   185 + 341*t + 171*t^2 + 642*t^3 +␣
↪O(t^4)]
```

The trace trick should work for power series rings too, even in the badly-conditioned case. Unfortunately I do not know how to compute the trace in advance, so I am not sure exactly how this would help. Also, I suspect the running time will be dominated by the expansion, so the trace trick will not really speed things up anyway. Another

problem is that the determinant is not always p:

```
sage: B.det()                                                    # long time
11 + 484*t^2 + 451*t^3 + O(t^4)
```

However, it appears that the determinant always has the property that if you substitute t - 11t, you do get the constant series p (mod p**prec). Similarly for the trace. And since the parameter only really makes sense when it is divisible by p anyway, perhaps this is not a problem after all.

sage.schemes.hyperelliptic_curves.monsky_washnitzer.**matrix_of_frobenius_hyperelliptic**(*Q*, *p=Non*
prec=N
M=No

Compute the matrix of Frobenius on Monsky-Washnitzer cohomology, with respect to the basis $(dx/2y, xdx/2y, ...x^{d-2}dx/2y)$, where d is the degree of Q.

INPUT:

- `Q` – monic polynomial $Q(x)$
- `p` – prime ≥ 5 for which E has good reduction
- `prec` – (optional) p-adic precision of the coefficient ring
- `M` – (optional) adjusted p-adic precision of the coefficient ring

OUTPUT:

$(d-1)$ x $(d-1)$ matrix M of Frobenius on Monsky-Washnitzer cohomology, and list of differentials {f_i } such that

$$\phi^*(x^i dx/2y) = df_i + M[i] * vec(dx/2y, ..., x^{d-2}dx/2y)$$

EXAMPLES:

```
sage: # needs sage.rings.padics
sage: p = 5
sage: prec = 3
sage: R.<x> = QQ['x']
sage: A,f = monsky_washnitzer.matrix_of_frobenius_hyperelliptic(x^5 - 2*x + 3, p,␣
↪prec)
sage: A
[             4*5 + O(5^3)        5 + 2*5^2 + O(5^3) 2 + 3*5 + 2*5^2 + O(5^3)      2␣
↪+ 5 + 5^2 + O(5^3)]
[       3*5 + 5^2 + O(5^3)              3*5 + O(5^3)              4*5 + O(5^3)       ␣
↪  2 + 5^2 + O(5^3)]
[     4*5 + 4*5^2 + O(5^3)      3*5 + 2*5^2 + O(5^3)        5 + 3*5^2 + O(5^3)     ␣
↪2*5 + 2*5^2 + O(5^3)]
[             5^2 + O(5^3)        5 + 4*5^2 + O(5^3)      4*5 + 3*5^2 + O(5^3)       ␣
↪      2*5 + O(5^3)]
```

sage.schemes.hyperelliptic_curves.monsky_washnitzer.**reduce_all**(*Q*, *p*, *coeffs*, *offset*, *compute_exact_form=False*)

Apply cohomology relations to reduce all terms to a linear combination of dx/y and xdx/y.

INPUT:

- `Q` – cubic polynomial
- `coeffs` – list of length 3 lists. The i-th list [a, b, c] represents $y^{2(i-offset)}(a + bx + cx^2)dx/y$.

- `offset` – nonnegative integer

OUTPUT:

- `A`, `B` – pair such that the input differential is cohomologous to (A + Bx) dx/y.

Note: The algorithm operates in-place, so the data in coeffs is destroyed.

EXAMPLES:

```
sage: R.<x> = Integers(5^3)['x']
sage: Q = x^3 - x + R(1/4)
sage: coeffs = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
sage: coeffs = [[R.base_ring()(a) for a in row] for row in coeffs]
sage: monsky_washnitzer.reduce_all(Q, 5, coeffs, 1)
(21, 106)
```

sage.schemes.hyperelliptic_curves.monsky_washnitzer.**reduce_negative**(*Q*, *p*, *coeffs*, *offset*, *exact_form=None*)

Apply cohomology relations to incorporate negative powers of y into the y^0 term.

INPUT:

- `p` – prime
- `Q` – cubic polynomial
- `coeffs` – list of length 3 lists. The i-th list `[a, b, c]` represents $y^{2(i-offset)}(a + bx + cx^2)dx/y$.
- `offset` – nonnegative integer

OUTPUT:

The reduction is performed in-place. The output is placed in coeffs[offset]. Note that coeffs[i] will be meaningless for i offset after this function is finished.

EXAMPLES:

```
sage: R.<x> = Integers(5^3)['x']
sage: Q = x^3 - x + R(1/4)
sage: coeffs = [[10, 15, 20], [1, 2, 3], [4, 5, 6], [7, 8, 9]]
sage: coeffs = [[R.base_ring()(a) for a in row] for row in coeffs]
sage: monsky_washnitzer.reduce_negative(Q, 5, coeffs, 3)
sage: coeffs[3]
[28, 52, 9]
```

```
sage: R.<x> = Integers(7^3)['x']
sage: Q = x^3 - x + R(1/4)
sage: coeffs = [[7, 14, 21], [1, 2, 3], [4, 5, 6], [7, 8, 9]]
sage: coeffs = [[R.base_ring()(a) for a in row] for row in coeffs]
sage: monsky_washnitzer.reduce_negative(Q, 7, coeffs, 3)
sage: coeffs[3]
[245, 332, 9]
```

sage.schemes.hyperelliptic_curves.monsky_washnitzer.**reduce_positive**(*Q*, *p*, *coeffs*, *offset*, *exact_form=None*)

Apply cohomology relations to incorporate positive powers of y into the y^0 term.

INPUT:

- `Q` – cubic polynomial
- `coeffs` – list of length 3 lists. The i-th list [a, b, c] represents $y^{2(i-offset)}(a+bx+cx^2)dx/y$.
- `offset` – nonnegative integer

OUTPUT:

The reduction is performed in-place. The output is placed in coeffs[offset]. Note that coeffs[i] will be meaningless for i offset after this function is finished.

EXAMPLES:

```
sage: R.<x> = Integers(5^3)['x']
sage: Q = x^3 - x + R(1/4)
```

```
sage: coeffs = [[1, 2, 3], [10, 15, 20]]
sage: coeffs = [[R.base_ring()(a) for a in row] for row in coeffs]
sage: monsky_washnitzer.reduce_positive(Q, 5, coeffs, 0)
sage: coeffs[0]
[16, 102, 88]
```

```
sage: coeffs = [[9, 8, 7], [10, 15, 20]]
sage: coeffs = [[R.base_ring()(a) for a in row] for row in coeffs]
sage: monsky_washnitzer.reduce_positive(Q, 5, coeffs, 0)
sage: coeffs[0]
[24, 108, 92]
```

sage.schemes.hyperelliptic_curves.monsky_washnitzer.**reduce_zero**(*Q*, *coeffs*, *offset*, *exact_form=None*)

Apply cohomology relation to incorporate x^2y^0 term into x^0y^0 and x^1y^0 terms.

INPUT:

- `Q` – cubic polynomial
- `coeffs` – list of length 3 lists. The i-th list [a, b, c] represents $y^{2(i-offset)}(a+bx+cx^2)dx/y$.
- `offset` – nonnegative integer

OUTPUT:

The reduction is performed in-place. The output is placed in coeffs[offset]. This method completely ignores coeffs[i] for i != offset.

EXAMPLES:

```
sage: R.<x> = Integers(5^3)['x']
sage: Q = x^3 - x + R(1/4)
sage: coeffs = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
sage: coeffs = [[R.base_ring()(a) for a in row] for row in coeffs]
sage: monsky_washnitzer.reduce_zero(Q, coeffs, 1)
sage: coeffs[1]
[6, 5, 0]
```

sage.schemes.hyperelliptic_curves.monsky_washnitzer.**transpose_list**(*input*)

INPUT:

- `input` – a list of lists, each list of the same length

OUTPUT:

- `output` – a list of lists such that output[i][j] = input[j][i]

EXAMPLES:

```
sage: from sage.schemes.hyperelliptic_curves.monsky_washnitzer import transpose_
↪list
sage: L = [[1, 2], [3, 4], [5, 6]]
sage: transpose_list(L)
[[1, 3, 5], [2, 4, 6]]
```

20.8 Frobenius on Monsky-Washnitzer cohomology of a hyperelliptic curve

This module provides *hypellfrob()*, that is a wrapper for the `matrix()` function in `hypellfrob.cpp`.

`hypellfrob.cpp` is a C++ program for computing the zeta function of a hyperelliptic curve over a largish prime finite field, based on the method described in the paper [Harv2007]. More precisely, it computes the matrix of Frobenius on the Monsky-Washnitzer cohomology of the curve; the zeta function can be recovered via the characteristic polynomial of the matrix.

AUTHORS:

- David Harvey (2007-05): initial version
- David Harvey (2007-12): rewrote for `hypellfrob` version 2.0
- Alex J. Best (2018-02): added wrapper

`sage.schemes.hyperelliptic_curves.hypellfrob.`**`hypellfrob`**(p, N, Q)

Compute the matrix of Frobenius acting on the Monsky-Washnitzer cohomology of a hyperelliptic curve $y^2 = Q(x)$, with respect to the basis $x^i dx/y$, $0 \leq i < 2g$.

INPUT:

- `p` – a prime
- `Q` – a monic polynomial in $\mathbf{Z}[x]$ of odd degree; must have no multiple roots mod p.
- `N` – precision parameter; the output matrix will be correct modulo p^N

The prime p should satisfy $p > (2g+1)(2N-1)$, where $g = (\deg Q - 1)/2$ is the genus of the curve.

ALGORITHM:

Described in [Harv2007], Section 7. Running time is theoretically $\widetilde{O}(p^{1/2} N^{5/2} g^3)$.

EXAMPLES:

```
sage: from sage.schemes.hyperelliptic_curves.hypellfrob import hypellfrob
sage: R.<x> = PolynomialRing(ZZ)
sage: f = x^5 + 2*x^2 + x + 1; p = 101
sage: M = hypellfrob(p, 4, f); M
[ 91844754 + O(101^4)  38295665 + O(101^4)  44498269 + O(101^4)  11854028 + O(101^
↪4)]
[ 93514789 + O(101^4)  48987424 + O(101^4)  53287857 + O(101^4)  61431148 + O(101^
↪4)]
[ 77916046 + O(101^4)  60656459 + O(101^4) 101244586 + O(101^4)  56237448 + O(101^
↪4)]
```

(continues on next page)

(continued from previous page)

```
[ 58643832 + O(101^4)  81727988 + O(101^4)  85294589 + O(101^4)  70104432 + O(101^
↪4)]
sage: -M.trace()
7 + O(101^4)
sage: sum(legendre_symbol(f(i), p) for i in range(p))
7
sage: ZZ(M.det())
10201
sage: M = hypellfrob(p, 1, f); M
[     O(101)      O(101) 93 + O(101) 62 + O(101)]
[     O(101)      O(101) 55 + O(101) 19 + O(101)]
[     O(101)      O(101) 65 + O(101) 42 + O(101)]
[     O(101)      O(101) 89 + O(101) 29 + O(101)]
```

Todo: Remove the restriction on p. Probably by merging in Robert's code, which eventually needs a fast C++/NTL implementation.

sage.schemes.hyperelliptic_curves.hypellfrob.**interval_products**(*M0*, *M1*, *target*)

Given matrices $M(t)$ with entries linear in t over $\mathbf{Z}/N\mathbf{Z}$ and a list of integers $a_0 < b_0 \leq a_1 < b_1 \leq \cdots \leq a_n < b_n$, compute the matrices $\prod_{t=a_i+1}^{b_i} M(t)$ for $i = 0$ to n.

INPUT:

- `M0`, `M1` – matrices over $\mathbf{Z}/N\mathbf{Z}$, so that $M(t) = M_0 + M_1 t$
- `target` – a list of integers $a_0, b_0, \ldots, a_n, b_n$

ALGORITHM:

Described in [Harv2007], Theorem 10. Based on the work of Bostan-Gaudry-Schost [BGS2007].

EXAMPLES:

```
sage: from sage.schemes.hyperelliptic_curves.hypellfrob import interval_products
sage: interval_products(Matrix(Integers(9), 2,2, [1,0,1,0]),
....:   Matrix(Integers(9), 2, 2, [1, 1, 0, 2]),[0,2,2,4])
[
[7 8]  [5 4]
[5 1], [2 7]
]
sage: [prod(Matrix(Integers(9), 2, 2, [t + 1, t, 1, 2*t])
....:  for t in range(2*i + 1, 2*i + 1 + 2)) for i in range(2)]
[
[7 8]  [5 4]
[5 1], [2 7]
]
```

An example with larger modulus:

```
sage: interval_products(Matrix(Integers(3^8), 1, 1, [1]),
....:   Matrix(Integers(3^8), 1, 1, [1]), [2,4])
[[20]]
sage: [prod(Matrix(Integers(3^8), 1, 1, [t + 1]) for t in range(3,5))]
[[20]]
```

An even larger modulus:

```
sage: interval_products(Matrix(Integers(3^18), 1, 1, [1]),
....:   Matrix(Integers(3^18), 1, 1, [1]), [2,4])
[[20]]
sage: [prod(Matrix(Integers(3^18), 1, 1, [t + 1]) for t in range(3,5))]
[[20]]
```

20.9 Jacobian of a general hyperelliptic curve

class sage.schemes.hyperelliptic_curves.jacobian_generic.**HyperellipticJacobian_generic**(*C*, *category=*

Bases: Jacobian_generic

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: FF = FiniteField(2003)
sage: R.<x> = PolynomialRing(FF)
sage: f = x**5 + 1184*x**3 + 1846*x**2 + 956*x + 560
sage: C = HyperellipticCurve(f)
sage: J = C.jacobian()
sage: a = x**2 + 376*x + 245; b = 1015*x + 1368
sage: X = J(FF)
sage: D = X([a,b])
sage: D
(x^2 + 376*x + 245, y + 988*x + 635)
sage: J(0)
(1)
sage: D == J([a,b])
True
sage: D == D + J(0)
True
```

A more extended example, demonstrating arithmetic in J(QQ) and J(K) for a number field K/QQ.

```
sage: P.<x> = PolynomialRing(QQ)
sage: f = x^5 - x + 1; h = x
sage: C = HyperellipticCurve(f,h,'u,v'); C
Hyperelliptic Curve over Rational Field defined by v^2 + u*v = u^5 - u + 1
sage: PP = C.ambient_space(); PP
Projective Space of dimension 2 over Rational Field
sage: C.defining_polynomial()
-x0^5 + x0*x1*x2^3 + x1^2*x2^3 + x0*x2^4 - x2^5
sage: C(QQ)
Set of rational points of Hyperelliptic Curve over Rational Field
 defined by v^2 + u*v = u^5 - u + 1
sage: K.<t> = NumberField(x^2 - 2)                                          #␣
↪needs sage.rings.number_field
sage: C(K)                                                                  #␣
↪needs sage.rings.number_field
Set of rational points of Hyperelliptic Curve
 over Number Field in t with defining polynomial x^2 - 2
 defined by v^2 + u*v = u^5 - u + 1
```

(continues on next page)

(continued from previous page)

```
sage: P = C(QQ)(0,1,1); P
(0 : 1 : 1)
sage: P == C(0,1,1)
True
sage: C(0,1,1).parent()
Set of rational points of Hyperelliptic Curve over Rational Field
 defined by v^2 + u*v = u^5 - u + 1

sage: # needs sage.rings.number_field
sage: P1 = C(K)(P)
sage: P2 = C(K)([2, 4*t - 1, 1])
sage: P3 = C(K)([-1/2, 1/8*(7*t+2), 1])
sage: P1, P2, P3
((0 : 1 : 1), (2 : 4*t - 1 : 1), (-1/2 : 7/8*t + 1/4 : 1))

sage: J = C.jacobian(); J
Jacobian of Hyperelliptic Curve over Rational Field
defined by v^2 + u*v = u^5 - u + 1
sage: Q = J(QQ)(P); Q
(u, v - 1)
sage: for i in range(6): Q*i
(1)
(u, v - 1)
(u^2, v + u - 1)
(u^2, v + 1)
(u, v + 1)
(1)

sage: # needs sage.rings.number_field
sage: Q1 = J(K)(P1); print("%s -> %s"%( P1, Q1 ))
(0 : 1 : 1) -> (u, v - 1)
sage: Q2 = J(K)(P2); print("%s -> %s"%( P2, Q2 ))
(2 : 4*t - 1 : 1) -> (u - 2, v - 4*t + 1)
sage: Q3 = J(K)(P3); print("%s -> %s"%( P3, Q3 ))
(-1/2 : 7/8*t + 1/4 : 1) -> (u + 1/2, v - 7/8*t - 1/4)
sage: R.<x> = PolynomialRing(K)
sage: Q4 = J(K)([x^2 - t, R(1)])
sage: for i in range(4): Q4*i
(1)
(u^2 - t, v - 1)
(u^2 + (-3/4*t - 9/16)*u + 1/2*t + 1/4, v + (-1/32*t - 57/64)*u + 1/2*t + 9/16)
(u^2 + (1352416/247009*t - 1636930/247009)*u - 1156544/247009*t + 1900544/247009,
 v + (-2326345442/122763473*t + 3233153137/122763473)*u
                              + 2439343104/122763473*t - 3350862929/122763473)
sage: R2 = Q2*5; R2
(u^2 - 3789465233/116983808*u - 267915823/58491904,
 v + (-233827256513849/1789384327168*t + 1/2)*u - 15782925357447/894692163584*t)
sage: R3 = Q3*5; R3
(u^2 + 566330080839913890623/144264547989509096459 52*u
     - 265318141763956762312 73/288529095979018192919 04,
 v + (25315544032164561407086086819910 3/24504984201757336889038363781591 04*t + 1/
↪2)*u
   + 24277085050649026115135634317643 11/49009968403514673778076727563182 08*t)
sage: R4 = Q4*5; R4
(u^2 - 3789465233/116983808*u - 267915823/58491904,
 v + (233827256513849/1789384327168*t + 1/2)*u + 15782925357447/894692163584*t)
```

Thus we find the following identity:

```
sage: 5*Q2 + 5*Q4                                                          #␣
↪needs sage.rings.number_field
(1)
```

Moreover the following relation holds in the 5-torsion subgroup:

```
sage: Q2 + Q4 == 2*Q1                                                      #␣
↪needs sage.rings.number_field
True
```

dimension()

Return the dimension of this Jacobian.

OUTPUT:

Integer

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: k.<a> = GF(9); R.<x> = k[]
sage: HyperellipticCurve(x^3 + x - 1, x + a).jacobian().dimension()
1
sage: g = HyperellipticCurve(x^6 + x - 1, x + a).jacobian().dimension(); g
2
sage: type(g)
<... 'sage.rings.integer.Integer'>
```

geometric_endomorphism_algebra_is_field(*B=200*, *proof=False*)

Return whether the geometric endomorphism algebra is a field.

This implies that the Jacobian of the curve is geometrically simple. It is based on Algorithm 4.10 from [Lom2019]

INPUT:

- `B` – (default: 200) the bound which appears in the statement of the algorithm from [Lom2019]
- `proof` – (default: `False`) whether or not to insist on a provably correct answer. This is related to the warning in the docstring of this module: if this function returns `False`, then strictly speaking this has not been proven to be `False` until one has exhibited a non-trivial endomorphism, which these methods are not designed to carry out. If one is convinced that this method should return `True`, but it is returning `False`, then this can be exhibited by increasing B.

OUTPUT:

Boolean indicating whether or not the geometric endomorphism algebra is a field.

EXAMPLES:

This is LMFDB curve 262144.d.524288.2 which has QM. Although its Jacobian is geometrically simple, the geometric endomorphism algebra is not a field:

```
sage: R.<x> = QQ[]
sage: f = x^5 + x^4 + 4*x^3 + 8*x^2 + 5*x + 1
sage: C = HyperellipticCurve(f)
sage: J = C.jacobian()
sage: J.geometric_endomorphism_algebra_is_field()
False
```

This is LMFDB curve 50000.a.200000.1:

```
sage: f = 8*x^5 + 1
sage: C = HyperellipticCurve(f)
sage: J = C.jacobian()
sage: J.geometric_endomorphism_algebra_is_field()
True
```

geometric_endomorphism_ring_is_ZZ (*B=200*, *proof=False*)

Return whether the geometric endomorphism ring of `self` is the integer ring **Z**.

INPUT:

- `B` – (default: 200) the bound which appears in the statement of the algorithm from [Lom2019]
- `proof` – (default: `False`) whether or not to insist on a provably correct answer. This is related to the warning in the module docstring of $jacobian_endomorphisms.py$: if this function returns `False`, then strictly speaking this has not been proven to be `False` until one has exhibited a non-trivial endomorphism, which the methods in that module are not designed to carry out. If one is convinced that this method should return `True`, but it is returning `False`, then this can be exhibited by increasing B.

OUTPUT:

Boolean indicating whether or not the geometric endomorphism ring is isomorphic to the integer ring.

EXAMPLES:

This is LMFDB curve 603.a.603.2:

```
sage: R.<x> = QQ[]
sage: f = 4*x^5 + x^4 - 4*x^3 + 2*x^2 + 4*x + 1
sage: C = HyperellipticCurve(f)
sage: J = C.jacobian()
sage: J.geometric_endomorphism_ring_is_ZZ()
True
```

This is LMFDB curve 1152.a.147456.1 whose geometric endomorphism ring is isomorphic to the group of 2x2 matrices over **Q**:

```
sage: f = x^6 - 2*x^4 + 2*x^2 - 1
sage: C = HyperellipticCurve(f)
sage: J = C.jacobian()
sage: J.geometric_endomorphism_ring_is_ZZ()
False
```

This is LMFDB curve 20736.k.373248.1 whose geometric endomorphism ring is isomorphic to the group of 2x2 matrices over a CM field:

```
sage: f = x^6 + 8
sage: C = HyperellipticCurve(f)
sage: J = C.jacobian()
sage: J.geometric_endomorphism_ring_is_ZZ()
False
```

This is LMFDB curve 708.a.181248.1:

```
sage: R.<x> = QQ[]
sage: f = -3*x^6 - 16*x^5 + 36*x^4 + 194*x^3 - 164*x^2 - 392*x - 143
sage: C = HyperellipticCurve(f)
```

(continues on next page)

(continued from previous page)

```
sage: J = C.jacobian()
sage: J.geometric_endomorphism_ring_is_ZZ()
True
```

This is LMFDB curve 10609.a.10609.1 whose geometric endomorphism ring is an order in a real quadratic field:

```
sage: f = x^6 + 2*x^4 + 2*x^3 + 5*x^2 + 6*x + 1
sage: C = HyperellipticCurve(f)
sage: J = C.jacobian()
sage: J.geometric_endomorphism_ring_is_ZZ()
False
```

This is LMFDB curve 160000.c.800000.1 whose geometric endomorphism ring is an order in a CM field:

```
sage: f = x^5 - 1
sage: C = HyperellipticCurve(f)
sage: J = C.jacobian()
sage: J.geometric_endomorphism_ring_is_ZZ()
False
```

This is LMFDB curve 262144.d.524288.2 whose geometric endomorphism ring is an order in a quaternion algebra:

```
sage: f = x^5 + x^4 + 4*x^3 + 8*x^2 + 5*x + 1
sage: C = HyperellipticCurve(f)
sage: J = C.jacobian()
sage: J.geometric_endomorphism_ring_is_ZZ()
False
```

This is LMFDB curve 578.a.2312.1 whose geometric endomorphism ring is $\mathbf{Q} \times \mathbf{Q}$:

```
sage: f = 4*x^5 - 7*x^4 + 10*x^3 - 7*x^2 + 4*x
sage: C = HyperellipticCurve(f)
sage: J = C.jacobian()
sage: J.geometric_endomorphism_ring_is_ZZ()
False
```

point(*mumford*, *check=True*)

20.10 Jacobian of a hyperelliptic curve of genus 2

class sage.schemes.hyperelliptic_curves.jacobian_g2.**HyperellipticJacobian_g2**(*C*, *category=None*)

Bases: *HyperellipticJacobian_generic*

kummer_surface()

20.11 Rational point sets on a Jacobian

EXAMPLES:

```
sage: x = QQ['x'].0
sage: f = x^5 + x + 1
sage: C = HyperellipticCurve(f); C
Hyperelliptic Curve over Rational Field defined by y^2 = x^5 + x + 1
sage: C(QQ)
Set of rational points of Hyperelliptic Curve over Rational Field
 defined by y^2 = x^5 + x + 1
sage: P = C([0,1,1])
sage: J = C.jacobian(); J
Jacobian of Hyperelliptic Curve over Rational Field defined by y^2 = x^5 + x + 1
sage: Q = J(QQ)(P); Q
(x, y - 1)
sage: Q + Q
(x^2, y - 1/2*x - 1)
sage: Q*3
(x^2 - 1/64*x + 1/8, y + 255/512*x + 65/64)
```

```
sage: F.<a> = GF(3)
sage: R.<x> = F[]
sage: f = x^5 - 1
sage: C = HyperellipticCurve(f)
sage: J = C.jacobian()
sage: X = J(F)
sage: a = x^2 - x + 1; b = -x + 1; c = x - 1; d = 0
sage: D1 = X([a,b]); D1
(x^2 + 2*x + 1, y + x + 2)
sage: D2 = X([c,d]); D2
(x + 2, y)
sage: D1 + D2
(x^2 + 2*x + 2, y + 2*x + 1)
```

class sage.schemes.hyperelliptic_curves.jacobian_homset.**JacobianHomset_divisor_classes**(*Y*, *X*, ***kw*

Bases: SchemeHomset_points

base_extend(*R*)

curve()

value_ring()

Return S for a homset X(T) where T = Spec(S).

20.12 Jacobian 'morphism' as a class in the Picard group

This module implements the group operation in the Picard group of a hyperelliptic curve, represented as divisors in Mumford representation, using Cantor's algorithm.

A divisor on the hyperelliptic curve $y^2 + yh(x) = f(x)$ is stored in Mumford representation, that is, as two polynomials $u(x)$ and $v(x)$ such that:

- $u(x)$ is monic,
- $u(x)$ divides $f(x) - h(x)v(x) - v(x)^2$,
- $deg(v(x)) < deg(u(x)) \leq g$.

REFERENCES:

A readable introduction to divisors, the Picard group, Mumford representation, and Cantor's algorithm:

- J. Scholten, F. Vercauteren. An Introduction to Elliptic and Hyperelliptic Curve Cryptography and the NTRU Cryptosystem. To appear in B. Preneel (Ed.) State of the Art in Applied Cryptography - COSIC '03, Lecture Notes in Computer Science, Springer 2004.

A standard reference in the field of cryptography:

- R. Avanzi, H. Cohen, C. Doche, G. Frey, T. Lange, K. Nguyen, and F. Vercauteren, Handbook of Elliptic and Hyperelliptic Curve Cryptography. CRC Press, 2005.

EXAMPLES: The following curve is the reduction of a curve whose Jacobian has complex multiplication.

```
sage: x = GF(37)['x'].gen()
sage: H = HyperellipticCurve(x^5 + 12*x^4 + 13*x^3 + 15*x^2 + 33*x); H
Hyperelliptic Curve over Finite Field of size 37 defined
 by y^2 = x^5 + 12*x^4 + 13*x^3 + 15*x^2 + 33*x
```

At this time, Jacobians of hyperelliptic curves are handled differently than elliptic curves:

```
sage: J = H.jacobian(); J
Jacobian of Hyperelliptic Curve over Finite Field of size 37 defined
 by y^2 = x^5 + 12*x^4 + 13*x^3 + 15*x^2 + 33*x
sage: J = J(J.base_ring()); J
Set of rational points of Jacobian of Hyperelliptic Curve over Finite Field
 of size 37 defined by y^2 = x^5 + 12*x^4 + 13*x^3 + 15*x^2 + 33*x
```

Points on the Jacobian are represented by Mumford's polynomials. First we find a couple of points on the curve:

```
sage: P1 = H.lift_x(2); P1
(2 : 11 : 1)
sage: Q1 = H.lift_x(10); Q1
(10 : 18 : 1)
```

Observe that 2 and 10 are the roots of the polynomials in x, respectively:

```
sage: P = J(P1); P
(x + 35, y + 26)
sage: Q = J(Q1); Q
(x + 27, y + 19)
```

```
sage: P + Q
(x^2 + 25*x + 20, y + 13*x)
```

(continues on next page)

(continued from previous page)

```
sage: (x^2 + 25*x + 20).roots(multiplicities=False)
[10, 2]
```

Frobenius satisfies

$$x^4 + 12 * x^3 + 78 * x^2 + 444 * x + 1369$$

on the Jacobian of this reduction and the order of the Jacobian is $N = 1904$.

```
sage: 1904*P
(1)
sage: 34*P == 0
True
sage: 35*P == P
True
sage: 33*P == -P
True
```

```
sage: Q*1904
(1)
sage: Q*238 == 0
True
sage: Q*239 == Q
True
sage: Q*237 == -Q
True
```

class sage.schemes.hyperelliptic_curves.jacobian_morphism.**JacobianMorphism_divisor_class_f**

Bases: AdditiveGroupElement, SchemeMorphism

An element of a Jacobian defined over a field, i.e. in $J(K) = \mathrm{Pic}^0_K(C)$.

point_of_jacobian_of_curve()

Return the point in the Jacobian of the curve.

The Jacobian is the one attached to the projective curve corresponding to this hyperelliptic curve.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(GF(11))
sage: f = x^6 + x + 1
sage: H = HyperellipticCurve(f)
sage: J = H.jacobian()
sage: D = J(H.lift_x(1))
sage: D  # divisor in Mumford representation
(x + 10, y + 6)
sage: jacobian_order = sum(H.frobenius_polynomial())
sage: jacobian_order
234
sage: p = D.point_of_jacobian_of_curve(); p
[Place (1/x0, 1/x0^3*x1 + 1)
 + Place (x0 + 10, x1 + 6)]
sage: p  # Jacobian point represented by an effective divisor
```

(continues on next page)

(continued from previous page)

```
[Place (1/x0, 1/x0^3*x1 + 1)
 + Place (x0 + 10, x1 + 6)]
sage: p.order()
39
sage: 234*p == 0
True
sage: G = p.parent()
sage: G
Group of rational points of Jacobian over Finite Field of size 11 (Hess model)
sage: J = G.parent()
sage: J
Jacobian of Projective Plane Curve over Finite Field of size 11
 defined by x0^6 + x0^5*x1 + x1^6 - x0^4*x2^2 (Hess model)
sage: C = J.curve()
sage: C
Projective Plane Curve over Finite Field of size 11
 defined by x0^6 + x0^5*x1 + x1^6 - x0^4*x2^2
sage: C.affine_patch(0) == H.affine_patch(2)
True
```

scheme()

Return the scheme this morphism maps to; or, where this divisor lives.

Warning: Although a pointset is defined over a specific field, the scheme returned may be over a different (usually smaller) field. The example below demonstrates this: the pointset is determined over a number field of absolute degree 2 but the scheme returned is defined over the rationals.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: x = QQ['x'].gen()
sage: f = x^5 + x
sage: H = HyperellipticCurve(f)
sage: F.<a> = NumberField(x^2 - 2, 'a')
sage: J = H.jacobian()(F); J
Set of rational points of Jacobian of Hyperelliptic Curve
 over Number Field in a with defining polynomial x^2 - 2
 defined by y^2 = x^5 + x
sage: P = J(H.lift_x(F(1)))
sage: P.scheme()
Jacobian of Hyperelliptic Curve over Rational Field defined by y^2 = x^5 + x
```

sage.schemes.hyperelliptic_curves.jacobian_morphism.**cantor_composition**(*D1*, *D2*, *f*, *h*, *genus*)

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: F.<a> = GF(7^2, 'a')
sage: x = F['x'].gen()
sage: f = x^7 + x^2 + a
sage: H = HyperellipticCurve(f, 2*x); H
Hyperelliptic Curve over Finite Field in a of size 7^2
 defined by y^2 + 2*x*y = x^7 + x^2 + a
sage: J = H.jacobian()(F); J
```

(continues on next page)

(continued from previous page)

```
Set of rational points of Jacobian of Hyperelliptic Curve over
 Finite Field in a of size 7^2 defined by y^2 + 2*x*y = x^7 + x^2 + a
```

```
sage: Q = J(H.lift_x(F(1))); Q                                                  #␣
↪needs sage.rings.finite_rings
(x + 6, y + 5*a)
sage: 10*Q  # indirect doctest                                                  #␣
↪needs sage.rings.finite_rings
(x^3 + (3*a + 1)*x^2 + (2*a + 5)*x + a + 5, y + (3*a + 2)*x^2 + (6*a + 1)*x + a +␣
↪4)
sage: 7*8297*Q                                                                  #␣
↪needs sage.rings.finite_rings
(1)
```

```
sage: Q = J(H.lift_x(F(a+1))); Q                                                #␣
↪needs sage.rings.finite_rings
(x + 6*a + 6, y + 2)
sage: 7*8297*Q  # indirect doctest                                              #␣
↪needs sage.rings.finite_rings
(1)

A test over a prime field:

sage: # needs sage.rings.finite_rings
sage: F = GF(next_prime(10^30))
sage: x = F['x'].gen()
sage: f = x^7 + x^2 + 1
sage: H = HyperellipticCurve(f, 2*x); H
Hyperelliptic Curve over Finite Field of size 1000000000000000000000000000057
 defined by y^2 + 2*x*y = x^7 + x^2 + 1
sage: J = H.jacobian()(F); J
Set of rational points of Jacobian of Hyperelliptic Curve
 over Finite Field of size 1000000000000000000000000000057
 defined by y^2 + 2*x*y = x^7 + x^2 + 1
sage: Q = J(H.lift_x(F(1))); Q
(x + 1000000000000000000000000000056, y + 1000000000000000000000000000056)
sage: 10*Q  # indirect doctest
(x^3 + 150296037169838934997145567227*x^2
     + 377701248971234560956743242408*x + 509456150352486043408603286615,
 y + 514451014495791237681619598519*x^2
   + 875375621665039398768235387900*x + 861429240012590886251910326876)
sage: 7*8297*Q
(x^3 + 354109761395485675499198390 63*x^2
     + 262304042352264645458868899 60*x + 681571430588959705539385624700,
 y + 999722365017286747841221441793*x^2
   + 262703715994522725686603955650*x + 626219823403254233972118260890)
```

sage.schemes.hyperelliptic_curves.jacobian_morphism.**cantor_composition_simple**(*D1*, *D2*, *f*, *genus*)

Given D_1 and D_2 two reduced Mumford divisors on the Jacobian of the curve $y^2 = f(x)$, computes a representative $D_1 + D_2$.

Warning: The representative computed is NOT reduced! Use *cantor_reduction_simple()* to reduce it.

EXAMPLES:

```
sage: x = QQ['x'].gen()
sage: f = x^5 + x
sage: H = HyperellipticCurve(f); H
Hyperelliptic Curve over Rational Field defined by y^2 = x^5 + x
```

```
sage: F.<a> = NumberField(x^2 - 2, 'a')                                    #␣
↪needs sage.rings.number_field
sage: J = H.jacobian()(F); J                                               #␣
↪needs sage.rings.number_field
Set of rational points of Jacobian of Hyperelliptic Curve over
 Number Field in a with defining polynomial x^2 - 2 defined by y^2 = x^5 + x
```

```
sage: # needs sage.rings.number_field
sage: P = J(H.lift_x(F(1))); P
(x - 1, y + a)
sage: Q = J(H.lift_x(F(0))); Q
(x, y)
sage: 2*P + 2*Q # indirect doctest
(x^2 - 2*x + 1, y + 3/2*a*x - 1/2*a)
sage: 2*(P + Q) # indirect doctest
(x^2 - 2*x + 1, y + 3/2*a*x - 1/2*a)
sage: 3*P # indirect doctest
(x^2 - 25/32*x + 49/32, y + 45/256*a*x + 315/256*a)
```

sage.schemes.hyperelliptic_curves.jacobian_morphism.**cantor_reduction**(*a*, *b*, *f*, *h*, *genus*)

Return the unique reduced divisor linearly equivalent to (a, b) on the curve $y^2 + yh(x) = f(x)$.

See the docstring of *sage.schemes.hyperelliptic_curves.jacobian_morphism* for information about divisors, linear equivalence, and reduction.

EXAMPLES:

```
sage: x = QQ['x'].gen()
sage: f = x^5 - x
sage: H = HyperellipticCurve(f, x); H
Hyperelliptic Curve over Rational Field defined by y^2 + x*y = x^5 - x
sage: J = H.jacobian()(QQ); J
Set of rational points of Jacobian of Hyperelliptic Curve over
 Rational Field defined by y^2 + x*y = x^5 - x
```

The following point is 2-torsion:

```
sage: Q = J(H.lift_x(0)); Q
(x, y)
sage: 2*Q # indirect doctest
(1)
```

The next point is not 2-torsion:

```
sage: P = J(H.lift_x(-1)); P
(x + 1, y)
sage: 2 * J(H.lift_x(-1)) # indirect doctest
(x^2 + 2*x + 1, y + 4*x + 4)
sage: 3 * J(H.lift_x(-1)) # indirect doctest
(x^2 - 487*x - 324, y + 10755*x + 7146)
```

sage.schemes.hyperelliptic_curves.jacobian_morphism.**cantor_reduction_simple**(*a*, *b*, *f*, *genus*)

Return the unique reduced divisor linearly equivalent to (a, b) on the curve $y^2 = f(x)$.

See the docstring of *sage.schemes.hyperelliptic_curves.jacobian_morphism* for information about divisors, linear equivalence, and reduction.

EXAMPLES:

```
sage: x = QQ['x'].gen()
sage: f = x^5 - x
sage: H = HyperellipticCurve(f); H
Hyperelliptic Curve over Rational Field defined by y^2 = x^5 - x
sage: J = H.jacobian()(QQ); J
Set of rational points of Jacobian of Hyperelliptic Curve over Rational Field
 defined by y^2 = x^5 - x
```

The following point is 2-torsion:

```
sage: P = J(H.lift_x(-1)); P
(x + 1, y)
sage: 2 * P # indirect doctest
(1)
```

20.13 Hyperelliptic curves of genus 2 over a general ring

class sage.schemes.hyperelliptic_curves.hyperelliptic_g2.**HyperellipticCurve_g2**(*PP*, *f*, *h=None*, *names=None*, *genus=None*)

Bases: *HyperellipticCurve_generic*

absolute_igusa_invariants_kohel()

Return the three absolute Igusa invariants used by Kohel [KohECHIDNA].

See also:

sage.schemes.hyperelliptic_curves.invariants()

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: HyperellipticCurve(x^5 - 1).absolute_igusa_invariants_kohel()
(0, 0, 0)
sage: HyperellipticCurve(x^5 - x + 1, x^2).absolute_igusa_invariants_kohel()
```

(continues on next page)

(continued from previous page)

```
(-1030567/178769, 259686400/178769, 20806400/178769)
sage: HyperellipticCurve((x^5 - x + 1)(3*x + 1), (x^2)(3*x + 1)).absolute_
↪igusa_invariants_kohel()
(-1030567/178769, 259686400/178769, 20806400/178769)
```

absolute_igusa_invariants_wamelen()

Return the three absolute Igusa invariants used by van Wamelen [Wam1999].

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: HyperellipticCurve(x^5 - 1).absolute_igusa_invariants_wamelen()
(0, 0, 0)
sage: HyperellipticCurve((x^5 - 1)(x - 2), (x^2)(x - 2)).absolute_igusa_
↪invariants_wamelen()
(0, 0, 0)
```

clebsch_invariants()

Return the Clebsch invariants (A, B, C, D) of Mestre, p 317, [Mes1991].

See also:

sage.schemes.hyperelliptic_curves.invariants()

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f = x^5 - x^4 + 3
sage: HyperellipticCurve(f).clebsch_invariants()
(0, -2048/375, -4096/25, -4881645568/84375)
sage: HyperellipticCurve(f(2*x)).clebsch_invariants()
(0, -8388608/375, -1073741824/25, -5241627016305836032/84375)

sage: HyperellipticCurve(f, x).clebsch_invariants()
(-8/15, 17504/5625, -23162896/140625, -420832861216768/7119140625)
sage: HyperellipticCurve(f(2*x), 2*x).clebsch_invariants()
(-512/15, 71696384/5625, -6072014209024/140625, -4518658440020313317048320/
↪7119140625)
```

igusa_clebsch_invariants()

Return the Igusa-Clebsch invariants I_2, I_4, I_6, I_{10} of Igusa and Clebsch [IJ1960].

See also:

sage.schemes.hyperelliptic_curves.invariants()

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f = x^5 - x + 2
sage: HyperellipticCurve(f).igusa_clebsch_invariants()
(-640, -20480, 1310720, 52160364544)
sage: HyperellipticCurve(f(2*x)).igusa_clebsch_invariants()
(-40960, -83886080, 343597383680, 56006764965979488256)

sage: HyperellipticCurve(f, x).igusa_clebsch_invariants()
(-640, 17920, -1966656, 52409511936)
sage: HyperellipticCurve(f(2*x), 2*x).igusa_clebsch_invariants()
(-40960, 73400320, -515547070464, 562742849411104112640)
```

is_odd_degree()

Return `True` if the curve is an odd degree model.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f = x^5 - x^4 + 3
sage: HyperellipticCurve(f).is_odd_degree()
True
```

jacobian()

Return the Jacobian of the hyperelliptic curve.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f = x^5 - x^4 + 3
sage: HyperellipticCurve(f).jacobian()
Jacobian of Hyperelliptic Curve over Rational Field defined by y^2 = x^5 - x^
↪4 + 3
```

kummer_morphism()

Return the morphism of an odd degree hyperelliptic curve to the Kummer surface of its Jacobian.

This could be extended to an even degree model if a prescribed embedding in its Jacobian is fixed.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f = x^5 - x^4 + 3
sage: HyperellipticCurve(f).kummer_morphism()   # not tested
```

20.14 Compute invariants of quintics and sextics via 'Ueberschiebung'

Todo:

- Implement invariants in small positive characteristic.
- Cardona-Quer and additional invariants for classifying automorphism groups.

AUTHOR:

- Nick Alexander

sage.schemes.hyperelliptic_curves.invariants.**Ueberschiebung**(f, g, k)

Return the differential operator $(fg)_k$.

This is defined by Mestre on page 315 [Mes1991]:

$$(fg)_k = \frac{(m-k)!(n-k)!}{m!n!}\left(\frac{\partial f}{\partial x}\frac{\partial g}{\partial y} - \frac{\partial f}{\partial y}\frac{\partial g}{\partial x}\right)^k.$$

EXAMPLES:

```
sage: from sage.schemes.hyperelliptic_curves.invariants import Ueberschiebung as␣
↪ub
sage: R.<x, y> = QQ[]
sage: ub(x, y, 0)
x*y
sage: ub(x^5 + 1, x^5 + 1, 1)
0
sage: ub(x^5 + 5*x + 1, x^5 + 5*x + 1, 0)
x^10 + 10*x^6 + 2*x^5 + 25*x^2 + 10*x + 1
```

sage.schemes.hyperelliptic_curves.invariants.**absolute_igusa_invariants_kohel**(*f*)

Given a sextic form f, return the three absolute Igusa invariants used by Kohel [KohECHIDNA].

f may be homogeneous in two variables or inhomogeneous in one.

EXAMPLES:

```
sage: from sage.schemes.hyperelliptic_curves.invariants import absolute_igusa_
↪invariants_kohel
sage: R.<x> = QQ[]
sage: absolute_igusa_invariants_kohel(x^5 - 1)
(0, 0, 0)
sage: absolute_igusa_invariants_kohel(x^5 - x)
(100, -20000, -2000)
```

The following example can be checked against Kohel's database [KohECHIDNA]

```
sage: h = -x^5 + 3*x^4 + 2*x^3 - 6*x^2 - 3*x + 1
sage: i1, i2, i3 = absolute_igusa_invariants_kohel(h)
sage: list(map(factor, (i1, i2, i3)))
[2^2 * 3^5 * 5 * 31, 2^5 * 3^11 * 5, 2^4 * 3^9 * 31]
sage: list(map(factor, (150660, 28343520, 9762768)))
[2^2 * 3^5 * 5 * 31, 2^5 * 3^11 * 5, 2^4 * 3^9 * 31]
```

sage.schemes.hyperelliptic_curves.invariants.**absolute_igusa_invariants_wamelen**(*f*)

Given a sextic form f, return the three absolute Igusa invariants used by van Wamelen [Wam1999].

f may be homogeneous in two variables or inhomogeneous in one.

REFERENCES:

- [Wam1999]

EXAMPLES:

```
sage: from sage.schemes.hyperelliptic_curves.invariants import absolute_igusa_
↪invariants_wamelen
sage: R.<x> = QQ[]
sage: absolute_igusa_invariants_wamelen(x^5 - 1)
(0, 0, 0)
```

The following example can be checked against van Wamelen's paper:

```
sage: h = -x^5 + 3*x^4 + 2*x^3 - 6*x^2 - 3*x + 1
sage: i1, i2, i3 = absolute_igusa_invariants_wamelen(h)
sage: list(map(factor, (i1, i2, i3)))
[2^7 * 3^15, 2^5 * 3^11 * 5, 2^4 * 3^9 * 31]
```

sage.schemes.hyperelliptic_curves.invariants.**clebsch_invariants**(f)

Given a sextic form f, return the Clebsch invariants (A, B, C, D) of Mestre, p 317, [Mes1991].

f may be homogeneous in two variables or inhomogeneous in one.

EXAMPLES:

```
sage: from sage.schemes.hyperelliptic_curves.invariants import clebsch_invariants
sage: R.<x, y> = QQ[]
sage: clebsch_invariants(x^6 + y^6)
(2, 2/3, -2/9, 0)
sage: R.<x> = QQ[]
sage: clebsch_invariants(x^6 + x^5 + x^4 + x^2 + 2)
(62/15, 15434/5625, -236951/140625, 229930748/791015625)

sage: magma(x^6 + 1).ClebschInvariants()                    # optional - magma
[ 2, 2/3, -2/9, 0 ]
sage: magma(x^6 + x^5 + x^4 + x^2 + 2).ClebschInvariants()  # optional - magma
[ 62/15, 15434/5625, -236951/140625, 229930748/791015625 ]
```

sage.schemes.hyperelliptic_curves.invariants.**clebsch_to_igusa**(A, B, C, D)

Convert Clebsch invariants A, B, C, D to Igusa invariants I_2, I_4, I_6, I_{10}.

EXAMPLES:

```
sage: from sage.schemes.hyperelliptic_curves.invariants import clebsch_to_igusa,␣
↪igusa_to_clebsch
sage: clebsch_to_igusa(2, 3, 4, 5)
(-240, 17370, 231120, -103098906)
sage: igusa_to_clebsch(*clebsch_to_igusa(2, 3, 4, 5))
(2, 3, 4, 5)

sage: Cs = tuple(map(GF(31), (2, 3, 4, 5))); Cs
(2, 3, 4, 5)
sage: clebsch_to_igusa(*Cs)
(8, 10, 15, 26)
sage: igusa_to_clebsch(*clebsch_to_igusa(*Cs))
(2, 3, 4, 5)
```

sage.schemes.hyperelliptic_curves.invariants.**differential_operator**(f, g, k)

Return the differential operator $(fg)_k$ symbolically in the polynomial ring in `dfdx, dfdy, dgdx, dgdy`.

This is defined by Mestre on p 315 [Mes1991]:

$$(fg)_k = \frac{(m-k)!(n-k)!}{m!n!}\left(\frac{\partial f}{\partial x}\frac{\partial g}{\partial y} - \frac{\partial f}{\partial y}\frac{\partial g}{\partial x}\right)^k.$$

EXAMPLES:

```
sage: from sage.schemes.hyperelliptic_curves.invariants import differential_
↪operator
sage: R.<x, y> = QQ[]
sage: differential_operator(x, y, 0)
1
sage: differential_operator(x, y, 1)
-dfdy*dgdx + dfdx*dgdy
sage: differential_operator(x*y, x*y, 2)
1/4*dfdy^2*dgdx^2 - 1/2*dfdx*dfdy*dgdx*dgdy + 1/4*dfdx^2*dgdy^2
```

(continues on next page)

(continued from previous page)

```
sage: differential_operator(x^2*y, x*y^2, 2)
1/36*dfdy^2*dgdx^2 - 1/18*dfdx*dfdy*dgdx*dgdy + 1/36*dfdx^2*dgdy^2
sage: differential_operator(x^2*y, x*y^2, 4)
1/576*dfdy^4*dgdx^4 - 1/144*dfdx*dfdy^3*dgdx^3*dgdy + 1/96*dfdx^2*dfdy^2*dgdx^
↪2*dgdy^2
- 1/144*dfdx^3*dfdy*dgdx*dgdy^3 + 1/576*dfdx^4*dgdy^4
```

sage.schemes.hyperelliptic_curves.invariants.**diffsymb**(*U*, *f*, *g*)

Given a differential operator `U` in `dfdx, dfdy, dgdx, dgdy`, represented symbolically by `U`, apply it to `f, g`.

EXAMPLES:

```
sage: from sage.schemes.hyperelliptic_curves.invariants import diffsymb
sage: R.<x, y> = QQ[]
sage: S.<dfdx, dfdy, dgdx, dgdy> = QQ[]
sage: [ diffsymb(dd, x^2, y*0 + 1) for dd in S.gens() ]
[2*x, 0, 0, 0]
sage: [ diffsymb(dd, x*0 + 1, y^2) for dd in S.gens() ]
[0, 0, 0, 2*y]
sage: [ diffsymb(dd, x^2, y^2) for dd in S.gens() ]
[2*x*y^2, 0, 0, 2*x^2*y]

sage: diffsymb(dfdx + dfdy*dgdy, y*x^2, y^3)
2*x*y^4 + 3*x^2*y^2
```

sage.schemes.hyperelliptic_curves.invariants.**diffxy**(*f*, *x*, *xtimes*, *y*, *ytimes*)

Differentiate a polynomial `f`, `xtimes` with respect to `x`, and `` `ytimes `` with respect to `y`.

EXAMPLES:

```
sage: from sage.schemes.hyperelliptic_curves.invariants import diffxy
sage: R.<u, v> = QQ[]
sage: diffxy(u^2*v^3, u, 0, v, 0)
u^2*v^3
sage: diffxy(u^2*v^3, u, 2, v, 1)
6*v^2
sage: diffxy(u^2*v^3, u, 2, v, 2)
12*v
sage: diffxy(u^2*v^3 + u^4*v^4, u, 2, v, 2)
144*u^2*v^2 + 12*v
```

sage.schemes.hyperelliptic_curves.invariants.**igusa_clebsch_invariants**(*f*)

Given a sextic form f, return the Igusa-Clebsch invariants I_2, I_4, I_6, I_{10} of Igusa and Clebsch [IJ1960].

f may be homogeneous in two variables or inhomogeneous in one.

EXAMPLES:

```
sage: from sage.schemes.hyperelliptic_curves.invariants import igusa_clebsch_
↪invariants
sage: R.<x, y> = QQ[]
sage: igusa_clebsch_invariants(x^6 + y^6)
(-240, 1620, -119880, -46656)
sage: R.<x> = QQ[]
sage: igusa_clebsch_invariants(x^6 + x^5 + x^4 + x^2 + 2)
(-496, 6220, -955932, -1111784)
```

(continues on next page)

(continued from previous page)

```
sage: magma(x^6 + 1).IgusaClebschInvariants()                    # optional -␣
↪magma
[ -240, 1620, -119880, -46656 ]
sage: magma(x^6 + x^5 + x^4 + x^2 + 2).IgusaClebschInvariants()  # optional -␣
↪magma
[ -496, 6220, -955932, -1111784 ]
```

sage.schemes.hyperelliptic_curves.invariants.**igusa_to_clebsch**(*I2*, *I4*, *I6*, *I10*)

Convert Igusa invariants I_2, I_4, I_6, I_{10} to Clebsch invariants A, B, C, D.

EXAMPLES:

```
sage: from sage.schemes.hyperelliptic_curves.invariants import clebsch_to_igusa,␣
↪igusa_to_clebsch
sage: igusa_to_clebsch(-2400, 173700, 23112000, -10309890600)
(20, 342/5, 2512/5, 43381012/1125)
sage: clebsch_to_igusa(*igusa_to_clebsch(-2400, 173700, 23112000, -10309890600))
(-2400, 173700, 23112000, -10309890600)

sage: Is = tuple(map(GF(31), (-2400, 173700, 23112000, -10309890600))); Is
(18, 7, 12, 27)
sage: igusa_to_clebsch(*Is)
(20, 25, 25, 12)
sage: clebsch_to_igusa(*igusa_to_clebsch(*Is))
(18, 7, 12, 27)
```

sage.schemes.hyperelliptic_curves.invariants.**ubs**(*f*)

Given a sextic form f, return a dictionary of the invariants of Mestre, p 317 [Mes1991].

f may be homogeneous in two variables or inhomogeneous in one.

EXAMPLES:

```
sage: from sage.schemes.hyperelliptic_curves.invariants import ubs
sage: x = QQ['x'].0
sage: ubs(x^6 + 1)
{'A': 2,
 'B': 2/3,
 'C': -2/9,
 'D': 0,
 'Delta': -2/3*x^2*h^2,
 'f': x^6 + h^6,
 'i': 2*x^2*h^2,
 'y1': 0,
 'y2': 0,
 'y3': 0}

sage: R.<u, v> = QQ[]
sage: ubs(u^6 + v^6)
{'A': 2,
 'B': 2/3,
 'C': -2/9,
 'D': 0,
 'Delta': -2/3*u^2*v^2,
 'f': u^6 + v^6,
 'i': 2*u^2*v^2,
```

(continues on next page)

(continued from previous page)

```
 'y1': 0,
 'y2': 0,
 'y3': 0}

sage: R.<t> = GF(31)[]
sage: ubs(t^6 + 2*t^5 + t^2 + 3*t + 1)
{'A': 0,
 'B': -12,
 'C': -15,
 'D': -15,
 'Delta': -10*t^4 + 12*t^3*h + 7*t^2*h^2 - 5*t*h^3 + 2*h^4,
 'f': t^6 + 2*t^5*h + t^2*h^4 + 3*t*h^5 + h^6,
 'i': -4*t^4 + 10*t^3*h + 2*t^2*h^2 - 9*t*h^3 - 7*h^4,
 'y1': 4*t^2 - 10*t*h - 13*h^2,
 'y2': 6*t^2 - 4*t*h + 2*h^2,
 'y3': 4*t^2 - 4*t*h - 9*h^2}
```

20.15 Kummer surfaces over a general ring

class sage.schemes.hyperelliptic_curves.kummer_surface.**KummerSurface**(*J*)

Bases: AlgebraicScheme_subscheme_projective

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: f = x^5 + x + 1
sage: X = HyperellipticCurve(f)
sage: J = Jacobian(X)
sage: K = KummerSurface(J); K
Closed subscheme of Projective Space of dimension 3 over Rational Field defined␣
↪by:
X0^4 - 4*X0*X1^3 + 4*X0^2*X1*X2 - 4*X0*X1^2*X2 + 2*X0^2*X2^2 + X2^4 - 4*X0^3*X3 -␣
↪2*X0^2*X1*X3 - 2*X1*X2^2*X3 + X1^2*X3^2 - 4*X0*X2*X3^2
```

20.16 Conductor and reduction types for genus 2 curves

AUTHORS:

- Qing Liu and Henri Cohen (1994-1998): wrote genus2reduction C program
- William Stein (2006-03-05): wrote Sage interface to genus2reduction
- Jeroen Demeyer (2014-09-17): replace genus2reduction program by PARI library call (Issue #15808)

ACKNOWLEDGMENT: (From Liu's website:) Many thanks to Henri Cohen who started writing this program. After this program is available, many people pointed out to me (mathematical as well as programming) bugs : B. Poonen, E. Schaefer, C. Stahlke, M. Stoll, F. Villegas. So thanks to all of them. Thanks also go to Ph. Depouilly who help me to compile the program.

Also Liu has given me explicit permission to include genus2reduction with Sage and for people to modify the C source code however they want.

class sage.interfaces.genus2reduction.**Genus2reduction**

Bases: SageObject

Conductor and Reduction Types for Genus 2 Curves.

Use `R = genus2reduction(Q, P)` to obtain reduction information about the Jacobian of the projective smooth curve defined by $y^2 + Q(x)y = P(x)$. Type `R?` for further documentation and a description of how to interpret the local reduction data.

EXAMPLES:

```
sage: x = QQ['x'].0
sage: R = genus2reduction(x^3 - 2*x^2 - 2*x + 1, -5*x^5)
sage: R.conductor
1416875
sage: factor(R.conductor)
5^4 * 2267
```

The discriminant is always minimal:

```
sage: factor(R.minimal_disc)
2^3 * 5^5 * 2267
```

Printing R summarizes all the information computed about the curve

```
sage: R
Reduction data about this proper smooth genus 2 curve:
    y^2 + (x^3 - 2*x^2 - 2*x + 1)*y = -5*x^5
A Minimal Equation:
    y^2 ...
Minimal Discriminant: 56675000
Conductor: 1416875
Local Data:
    p=2
    (potential) stable reduction:  (II), j=1
    p=5
    (potential) stable reduction:  (I)
    reduction at p: [V] page 156, (3), f=4
    p=2267
    (potential) stable reduction:  (II), j=432
    reduction at p: [I{1-0-0}] page 170, (1), f=1
```

Here are some examples of curves with modular Jacobians:

```
sage: R = genus2reduction(x^3 + x + 1, -2*x^5 - 3*x^2 + 2*x - 2)
sage: factor(R.conductor)
23^2
sage: factor(genus2reduction(x^3 + 1, -x^5 - 3*x^4 + 2*x^2 + 2*x - 2).conductor)
29^2
sage: factor(genus2reduction(x^3 + x + 1, x^5 + 2*x^4 + 2*x^3 + x^2 - x - 1).
↪conductor)
5^6
```

EXAMPLES:

```
sage: genus2reduction(0, x^6 + 3*x^3 + 63)
Reduction data about this proper smooth genus 2 curve:
        y^2 = x^6 + 3*x^3 + 63
```

(continues on next page)

(continued from previous page)

```
A Minimal Equation:
        y^2 ...
Minimal Discriminant: -10628388316852992
Conductor: 2893401
Local Data:
        p=2
        (potential) stable reduction:  (V), j1+j2=0, j1*j2=0
        p=3
        (potential) stable reduction:  (I)
        reduction at p: [III{9}] page 184, (3)^2, f=10
        p=7
        (potential) stable reduction:  (V), j1+j2=0, j1*j2=0
        reduction at p: [I{0}-II-0] page 159, (1), f=2
```

In the above example, Liu remarks that in fact at $p = 2$, the reduction is [II-II-0] page 163, (1), $f = 8$. So the conductor of J(C) is actually $2 \cdot 2893401 = 5786802$.

A MODULAR CURVE:

Consider the modular curve $X_1(13)$ defined by an equation

$$y^2 + (x^3 - x^2 - 1)y = x^2 - x.$$

We have:

```
sage: genus2reduction(x^3-x^2-1, x^2 - x)
Reduction data about this proper smooth genus 2 curve:
        y^2 + (x^3 - x^2 - 1)*y = x^2 - x
A Minimal Equation:
        y^2 ...
Minimal Discriminant: -169
Conductor: 169
Local Data:
        p=13
        (potential) stable reduction:  (V), j1+j2=0, j1*j2=0
        reduction at p: [I{0}-II-0] page 159, (1), f=2
```

So the curve has good reduction at 2. At $p = 13$, the stable reduction is union of two elliptic curves, and both of them have 0 as modular invariant. The reduction at 13 is of type [I_0-II-0] (see Namikawa-Ueno, page 159). It is an elliptic curve with a cusp. The group of connected components of the Neron model of $J(C)$ is trivial, and the exponent of the conductor of $J(C)$ at 13 is $f = 2$. The conductor of $J(C)$ is 13^2. (Note: It is a theorem of Conrad-Edixhoven-Stein that the component group of $J(X_1(p))$ is trivial for all primes p.)

class sage.interfaces.genus2reduction.**ReductionData**(*pari_result*, *P*, *Q*, *Pmin*, *Qmin*, *minimal_disc*, *local_data*, *conductor*)

Bases: SageObject

Reduction data for a genus 2 curve.

How to read `local_data` attribute, i.e., if this class is R, then the following is the meaning of `R.local_data[p]`.

For each prime number p dividing the discriminant of $y^2 + Q(x)y = P(x)$, there are two lines.

The first line contains information about the stable reduction after field extension. Here are the meanings of the symbols of stable reduction:

(I) The stable reduction is smooth (i.e. the curve has potentially good reduction).

(II) The stable reduction is an elliptic curve E with an ordinary double point. j mod p is the modular invariant of E.

(III) The stable reduction is a projective line with two ordinary double points.

(IV) The stable reduction is two projective lines crossing transversally at three points.

(V) The stable reduction is the union of two elliptic curves E_1 and E_2 intersecting transversally at one point. Let j_1, j_2 be their modular invariants, then $j_1 + j_2$ and $j_1 j_2$ are computed (they are numbers mod p).

(VI) The stable reduction is the union of an elliptic curve E and a projective line which has an ordinary double point. These two components intersect transversally at one point. j mod p is the modular invariant of E.

(VII) The stable reduction is as above, but the two components are both singular.

In the cases (I) and (V), the Jacobian $J(C)$ has potentially good reduction. In the cases (III), (IV) and (VII), $J(C)$ has potentially multiplicative reduction. In the two remaining cases, the (potential) semi-abelian reduction of $J(C)$ is extension of an elliptic curve (with modular invariant j mod p) by a torus.

The second line contains three data concerning the reduction at p without any field extension.

1. The first symbol describes the REDUCTION AT p of C. We use the symbols of Namikawa-Ueno for the type of the reduction (Namikawa, Ueno:"The complete classification of fibers in pencils of curves of genus two", Manuscripta Math., vol. 9, (1973), pages 143-186.) The reduction symbol is followed by the corresponding page number (or just an indication) in the above article. The lower index is printed by , for instance, [I2-II-5] means [I_2-II-5]. Note that if K and K' are Kodaira symbols for singular fibers of elliptic curves, [K-K'-m] and [K'-K-m] are the same type. Finally, [K-K'-1] (not the same as [K-K'-1]) is [K'-K-alpha] in the notation of Namikawa-Ueno. The figure [2I_0-m] in Namikawa-Ueno, page 159 must be denoted by [2I_0-(m+1)].
2. The second datum is the GROUP OF CONNECTED COMPONENTS (over an ALGEBRAIC CLOSURE (!) of $\mathbf{F}_p$) of the Neron model of J(C). The symbol (n) means the cyclic group with n elements. When n=0, (0) is the trivial group (1). `Hn` is isomorphic to (2)x(2) if n is even and to (4) otherwise.

   Note - The set of rational points of Φ can be computed using Theorem 1.17 in S. Bosch and Q. Liu "Rational points of the group of components of a Neron model", Manuscripta Math. 98 (1999), 275-293.
3. Finally, f is the exponent of the conductor of $J(C)$ at p.

Warning: Be careful regarding the formula:

$$\text{valuation of the naive minimal discriminant} = f + n - 1 + 11c(X).$$

(Q. Liu : "Conducteur et discriminant minimal de courbes de genre 2", Compositio Math. 94 (1994) 51-79, Theoreme 2) is valid only if the residual field is algebraically closed as stated in the paper. So this equality does not hold in general over $\mathbf{Q}_p$. The fact is that the minimal discriminant may change after unramified extension. One can show however that, at worst, the change will stabilize after a quadratic unramified extension (Q. Liu : "Modèles entiers de courbes hyperelliptiques sur un corps de valuation discrète", Trans. AMS 348 (1996), 4577-4610, Section 7.2, Proposition 4).

`sage.interfaces.genus2reduction.`**`divisors_to_string`**(*divs*)

Convert a list of numbers (representing the orders of cyclic groups in the factorization of a finite abelian group) to a string according to the format shown in the examples.

INPUT:

- `divs` – a (possibly empty) list of numbers

OUTPUT: a string representation of these numbers

EXAMPLES:

```
sage: from sage.interfaces.genus2reduction import divisors_to_string
sage: print(divisors_to_string([]))
(1)
sage: print(divisors_to_string([5]))
(5)
sage: print(divisors_to_string([5]*6))
(5)^6
sage: print(divisors_to_string([2,3,4]))
(2)x(3)x(4)
sage: print(divisors_to_string([6,2,2]))
(6)x(2)^2
```

CHAPTER

TWENTYONE

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

i

sage.interfaces.genus2reduction, 737

s

sage.schemes.elliptic_curves.cm, 480
sage.schemes.elliptic_curves.constructor, 1
sage.schemes.elliptic_curves.descent_two_isogeny, 627
sage.schemes.elliptic_curves.ec_database, 361
sage.schemes.elliptic_curves.ell_curve_isogeny, 193
sage.schemes.elliptic_curves.ell_egros, 630
sage.schemes.elliptic_curves.ell_field, 91
sage.schemes.elliptic_curves.ell_finite_field, 121
sage.schemes.elliptic_curves.ell_generic, 59
sage.schemes.elliptic_curves.ell_local_data, 493
sage.schemes.elliptic_curves.ell_modular_symbols, 537
sage.schemes.elliptic_curves.ell_number_field, 362
sage.schemes.elliptic_curves.ell_padic_field, 634
sage.schemes.elliptic_curves.ell_point, 21
sage.schemes.elliptic_curves.ell_rational_field, 269
sage.schemes.elliptic_curves.ell_tate_curve, 503
sage.schemes.elliptic_curves.ell_torsion, 437
sage.schemes.elliptic_curves.ell_wp, 507
sage.schemes.elliptic_curves.formal_group, 153
sage.schemes.elliptic_curves.gal_reps, 441
sage.schemes.elliptic_curves.gal_reps_number_field, 452
sage.schemes.elliptic_curves.gp_simon, 635
sage.schemes.elliptic_curves.heegner, 557
sage.schemes.elliptic_curves.height, 414
sage.schemes.elliptic_curves.hom, 159
sage.schemes.elliptic_curves.hom_composite, 173
sage.schemes.elliptic_curves.hom_frobenius, 229
sage.schemes.elliptic_curves.hom_scalar, 223
sage.schemes.elliptic_curves.hom_sum, 181
sage.schemes.elliptic_curves.hom_velusqrt, 215
sage.schemes.elliptic_curves.isogeny_class, 463
sage.schemes.elliptic_curves.isogeny_small_degree, 235
sage.schemes.elliptic_curves.jacobian, 17
sage.schemes.elliptic_curves.kodaira_symbol, 501
sage.schemes.elliptic_curves.lseries_ell, 548
sage.schemes.elliptic_curves.mod5family, 635
sage.schemes.elliptic_curves.mod_poly, 267
sage.schemes.elliptic_curves.mod_sym_num, 542
sage.schemes.elliptic_curves.modular_parametrization, 535
sage.schemes.elliptic_curves.padic_lseries, 612

sage.schemes.elliptic_curves.period_lattice, 511
sage.schemes.elliptic_curves.period_lattice_region, 530
sage.schemes.elliptic_curves.Qcurves, 489
sage.schemes.elliptic_curves.saturation, 432
sage.schemes.elliptic_curves.sha_tate, 470
sage.schemes.elliptic_curves.weierstrass_morphism, 187
sage.schemes.elliptic_curves.weierstrass_transform, 636
sage.schemes.hyperelliptic_curves.constructor, 641
sage.schemes.hyperelliptic_curves.hypellfrob, 717
sage.schemes.hyperelliptic_curves.hyperelliptic_finite_field, 653
sage.schemes.hyperelliptic_curves.hyperelliptic_g2, 730
sage.schemes.hyperelliptic_curves.hyperelliptic_generic, 643
sage.schemes.hyperelliptic_curves.hyperelliptic_padic_field, 667
sage.schemes.hyperelliptic_curves.hyperelliptic_rational_field, 684
sage.schemes.hyperelliptic_curves.invariants, 732
sage.schemes.hyperelliptic_curves.jacobian_g2, 723
sage.schemes.hyperelliptic_curves.jacobian_generic, 719
sage.schemes.hyperelliptic_curves.jacobian_homset, 724
sage.schemes.hyperelliptic_curves.jacobian_morphism, 725
sage.schemes.hyperelliptic_curves.kummer_surface, 737
sage.schemes.hyperelliptic_curves.mestre, 685
sage.schemes.hyperelliptic_curves.monsky_washnitzer, 688

INDEX

A

a1() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 60
a2() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 60
a3() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 60
a4() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 60
a6() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 60
a_invariants() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 60
a_number() (*sage.schemes.hyperelliptic_curves.hyperelliptic_finite_field.HyperellipticCurve_finite_field method*), 655
abelian_group() (*sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field method*), 122
abelian_variety() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 274
absolute_degree() (*sage.schemes.elliptic_curves.heegner.RingClassField method*), 596
absolute_igusa_invariants_kohel() (*in module sage.schemes.hyperelliptic_curves.invariants*), 733
absolute_igusa_invariants_kohel() (*sage.schemes.hyperelliptic_curves.hyperelliptic_g2.HyperellipticCurve_g2 method*), 730
absolute_igusa_invariants_wamelen() (*in module sage.schemes.hyperelliptic_curves.invariants*), 733
absolute_igusa_invariants_wamelen() (*sage.schemes.hyperelliptic_curves.hyperelliptic_g2.HyperellipticCurve_g2 method*), 731
add_reductions() (*sage.schemes.elliptic_curves.saturation.EllipticCurveSaturator method*), 433
additive_order() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_field method*), 24
additive_order() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_finite_field method*), 40
additive_order() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_number_field method*), 44
adjusted_prec() (*in module sage.schemes.hyperelliptic_curves.monsky_washnitzer*), 707
ainvs() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 61
all_values_for_one_denominator() (*sage.schemes.elliptic_curves.mod_sym_num.ModularSymbolNumerical method*), 544
alpha() (*sage.schemes.elliptic_curves.heegner.GaloisAutomorphismQuadraticForm method*), 560
alpha() (*sage.schemes.elliptic_curves.height.EllipticCurveCanonicalHeight method*), 417
alpha() (*sage.schemes.elliptic_curves.padic_lseries.pAdicLseries method*), 614
an() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 275
an() (*sage.schemes.elliptic_curves.sha_tate.Sha method*), 472
an_numerical() (*sage.schemes.elliptic_curves.sha_tate.Sha method*), 473
an_padic() (*sage.schemes.elliptic_curves.sha_tate.Sha method*), 474
analytic_rank() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 275
analytic_rank_upper_bound() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 276
anlist() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 279
antilogarithm() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 279

ap() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 280
aplist() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 280
approximative_value() (*sage.schemes.elliptic_curves.mod_sym_num.ModularSymbolNumerical method*), 545
archimedean_local_height() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_number_field method*), 45
are_projectively_equivalent() (*in module sage.schemes.elliptic_curves.constructor*), 14
as_morphism() (*sage.schemes.elliptic_curves.hom.EllipticCurveHom method*), 159
at1() (*sage.schemes.elliptic_curves.lseries_ell.Lseries_ell method*), 549
ate_pairing() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_field method*), 24
atkin_lehner_act() (*sage.schemes.elliptic_curves.heegner.HeegnerPointOnX0N method*), 572
automorphisms() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 61

B

B() (*sage.schemes.elliptic_curves.height.EllipticCurveCanonicalHeight method*), 414
b2() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 62
b4() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 62
b6() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 62
b8() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 63
b_invariants() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 63
bad_reduction_type() (*sage.schemes.elliptic_curves.ell_local_data.EllipticCurveLocalData method*), 495
base_extend() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 63
base_extend() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 364
base_extend() (*sage.schemes.hyperelliptic_curves.hyperelliptic_generic.HyperellipticCurve_generic method*), 644
base_extend() (*sage.schemes.hyperelliptic_curves.jacobian_homset.JacobianHomset_divisor_classes method*), 724
base_extend() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.MonskyWashnitzerDifferentialRing method*), 694
base_extend() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialHyperellipticQuotientRing method*), 704
base_field() (*sage.schemes.elliptic_curves.ell_field.EllipticCurve_field method*), 91
base_field() (*sage.schemes.elliptic_curves.heegner.GaloisGroup method*), 561
base_field() (*sage.schemes.elliptic_curves.height.EllipticCurveCanonicalHeight method*), 417
base_ring() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 64
base_ring() (*sage.schemes.elliptic_curves.ell_modular_symbols.ModularSymbol method*), 538
baseWI (*class in sage.schemes.elliptic_curves.weierstrass_morphism*), 191
basis() (*sage.schemes.elliptic_curves.period_lattice.PeriodLattice_ell method*), 513
basis_matrix() (*sage.schemes.elliptic_curves.period_lattice.PeriodLattice_ell method*), 514
bernardi_sigma_function() (*sage.schemes.elliptic_curves.padic_lseries.pAdicLseriesSupersingular method*), 623
beta() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlgEmbedding method*), 590
betas() (*sage.schemes.elliptic_curves.heegner.HeegnerPoints_level_disc_cond method*), 578
Billerey_B_bound() (*in module sage.schemes.elliptic_curves.gal_reps_number_field*), 453
Billerey_B_l() (*in module sage.schemes.elliptic_curves.gal_reps_number_field*), 454
Billerey_P_l() (*in module sage.schemes.elliptic_curves.gal_reps_number_field*), 454
Billerey_R_bound() (*in module sage.schemes.elliptic_curves.gal_reps_number_field*), 454
Billerey_R_q() (*in module sage.schemes.elliptic_curves.gal_reps_number_field*), 455
border() (*sage.schemes.elliptic_curves.period_lattice_region.PeriodicRegion method*), 530
bound() (*sage.schemes.elliptic_curves.sha_tate.Sha method*), 476
bound_kato() (*sage.schemes.elliptic_curves.sha_tate.Sha method*), 476
bound_kolyvagin() (*sage.schemes.elliptic_curves.sha_tate.Sha method*), 477
brandt_module() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlg method*), 580

C

c4() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 64
c6() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 64
c_invariants() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 64
cantor_composition() (*in module sage.schemes.hyperelliptic_curves.jacobian_morphism*), 727
cantor_composition_simple() (*in module sage.schemes.hyperelliptic_curves.jacobian_morphism*), 728
cantor_reduction() (*in module sage.schemes.hyperelliptic_curves.jacobian_morphism*), 729
cantor_reduction_simple() (*in module sage.schemes.hyperelliptic_curves.jacobian_morphism*), 730
cardinality() (*sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field method*), 123
cardinality() (*sage.schemes.elliptic_curves.heegner.GaloisGroup method*), 562
cardinality() (*sage.schemes.hyperelliptic_curves.hyperelliptic_finite_field.HyperellipticCurve_finite_field method*), 655
cardinality_bsgs() (*sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field method*), 125
cardinality_exhaustive() (*sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field method*), 126
cardinality_exhaustive() (*sage.schemes.hyperelliptic_curves.hyperelliptic_finite_field.HyperellipticCurve_finite_field method*), 656
cardinality_hypellfrob() (*sage.schemes.hyperelliptic_curves.hyperelliptic_finite_field.HyperellipticCurve_finite_field method*), 656
cardinality_pari() (*sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field method*), 126
Cartier_matrix() (*sage.schemes.hyperelliptic_curves.hyperelliptic_finite_field.HyperellipticCurve_finite_field method*), 653
change_ring() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 65
change_ring() (*sage.schemes.hyperelliptic_curves.hyperelliptic_generic.HyperellipticCurve_generic method*), 644
change_ring() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.MonskyWashnitzerDifferentialRing method*), 694
change_ring() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialHyperellipticQuotientElement method*), 701
change_ring() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialHyperellipticQuotientRing method*), 704
change_weierstrass_model() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 65
characteristic_polynomial() (*sage.schemes.elliptic_curves.hom.EllipticCurveHom method*), 159
check_prime() (*in module sage.schemes.elliptic_curves.ell_local_data*), 500
chord_and_tangent() (*in module sage.schemes.elliptic_curves.constructor*), 14
class_number() (*in module sage.schemes.elliptic_curves.heegner*), 600
classical_modular_polynomial() (*in module sage.schemes.elliptic_curves.mod_poly*), 267
clear_cache() (*sage.schemes.elliptic_curves.mod_sym_num.ModularSymbolNumerical method*), 546
clebsch_invariants() (*in module sage.schemes.hyperelliptic_curves.invariants*), 733
clebsch_invariants() (*sage.schemes.hyperelliptic_curves.hyperelliptic_g2.HyperellipticCurve_g2 method*), 731
clebsch_to_igusa() (*in module sage.schemes.hyperelliptic_curves.invariants*), 734
cm_discriminant() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 364
cm_discriminant() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 280
cm_j_invariants() (*in module sage.schemes.elliptic_curves.cm*), 481
cm_j_invariants_and_orders() (*in module sage.schemes.elliptic_curves.cm*), 482
cm_orders() (*in module sage.schemes.elliptic_curves.cm*), 483
codomain() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlgEmbedding method*), 590
coeff() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.MonskyWashnitzerDifferential method*), 689
coefficients_from_j() (*in module sage.schemes.elliptic_curves.constructor*), 15
coefficients_from_Weierstrass_polynomial() (*in module sage.schemes.elliptic_curves.constructor*), 15
coeffs() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.MonskyWashnitzerDifferential*

method), 689
coeffs() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialCubicQuotientRingElement method*), 700
coeffs() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialHyperellipticQuotientElement method*), 702
coleman_integral() (*sage.schemes.hyperelliptic_curves.hyperelliptic_padic_field.HyperellipticCurve_padic_field method*), 668
coleman_integral() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.MonskyWashnitzerDifferential method*), 689
coleman_integral_from_weierstrass_via_boundary() (*sage.schemes.hyperelliptic_curves.hyperelliptic_padic_field.HyperellipticCurve_padic_field method*), 672
coleman_integral_P_to_S() (*sage.schemes.hyperelliptic_curves.hyperelliptic_padic_field.HyperellipticCurve_padic_field method*), 671
coleman_integral_S_to_Q() (*sage.schemes.hyperelliptic_curves.hyperelliptic_padic_field.HyperellipticCurve_padic_field method*), 671
coleman_integrals_on_basis() (*sage.schemes.hyperelliptic_curves.hyperelliptic_padic_field.HyperellipticCurve_padic_field method*), 673
coleman_integrals_on_basis_hyperelliptic() (*sage.schemes.hyperelliptic_curves.hyperelliptic_padic_field.HyperellipticCurve_padic_field method*), 674
compare_via_evaluation() (*in module sage.schemes.elliptic_curves.hom*), 169
complex_area() (*sage.schemes.elliptic_curves.period_lattice.PeriodLattice_ell method*), 514
complex_conjugation() (*sage.schemes.elliptic_curves.heegner.GaloisGroup method*), 562
complex_intersection_is_empty() (*sage.schemes.elliptic_curves.height.EllipticCurveCanonicalHeight method*), 417
compute_codomain_formula() (*in module sage.schemes.elliptic_curves.ell_curve_isogeny*), 202
compute_codomain_kohel() (*in module sage.schemes.elliptic_curves.ell_curve_isogeny*), 203
compute_intermediate_curves() (*in module sage.schemes.elliptic_curves.ell_curve_isogeny*), 204
compute_isogeny_bmss() (*in module sage.schemes.elliptic_curves.ell_curve_isogeny*), 205
compute_isogeny_kernel_polynomial() (*in module sage.schemes.elliptic_curves.ell_curve_isogeny*), 205
compute_isogeny_stark() (*in module sage.schemes.elliptic_curves.ell_curve_isogeny*), 206
compute_model() (*in module sage.schemes.elliptic_curves.ell_field*), 118
compute_sequence_of_maps() (*in module sage.schemes.elliptic_curves.ell_curve_isogeny*), 207
compute_trace_generic() (*in module sage.schemes.elliptic_curves.hom*), 170
compute_vw_kohel_even_deg1() (*in module sage.schemes.elliptic_curves.ell_curve_isogeny*), 208
compute_vw_kohel_even_deg3() (*in module sage.schemes.elliptic_curves.ell_curve_isogeny*), 209
compute_vw_kohel_odd() (*in module sage.schemes.elliptic_curves.ell_curve_isogeny*), 210
compute_wp_fast() (*in module sage.schemes.elliptic_curves.ell_wp*), 508
compute_wp_pari() (*in module sage.schemes.elliptic_curves.ell_wp*), 508
compute_wp_quadratic() (*in module sage.schemes.elliptic_curves.ell_wp*), 508
conductor() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 365
conductor() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 281
conductor() (*sage.schemes.elliptic_curves.heegner.HeegnerPoint method*), 564
conductor() (*sage.schemes.elliptic_curves.heegner.HeegnerPoints_level_disc_cond method*), 578
conductor() (*sage.schemes.elliptic_curves.heegner.KolyvaginCohomologyClass method*), 591
conductor() (*sage.schemes.elliptic_curves.heegner.RingClassField method*), 597
conductor_valuation() (*sage.schemes.elliptic_curves.ell_local_data.EllipticCurveLocalData method*), 496
congruence_number() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 281
conjugacy_test() (*in module sage.schemes.elliptic_curves.Qcurves*), 490
conjugate() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlgEmbedding method*), 590
conjugates_over_K() (*sage.schemes.elliptic_curves.heegner.HeegnerPointOnEllipticCurve*

method), 566
contract() (*sage.schemes.elliptic_curves.period_lattice_region.PeriodicRegion method*), 531
coordinates() (*sage.schemes.elliptic_curves.period_lattice.PeriodLattice_ell method*), 515
copy() (*sage.schemes.elliptic_curves.isogeny_class.IsogenyClass_EC_NumberField method*), 466
copy() (*sage.schemes.elliptic_curves.isogeny_class.IsogenyClass_EC_Rational method*), 466
count_points() (*sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field method*), 126
count_points() (*sage.schemes.hyperelliptic_curves.hyperelliptic_finite_field.HyperellipticCurve_finite_field method*), 657
count_points_exhaustive() (*sage.schemes.hyperelliptic_curves.hyperelliptic_finite_field.HyperellipticCurve_finite_field method*), 658
count_points_frobenius_polynomial() (*sage.schemes.hyperelliptic_curves.hyperelliptic_finite_field.HyperellipticCurve_finite_field method*), 659
count_points_hypellfrob() (*sage.schemes.hyperelliptic_curves.hyperelliptic_finite_field.HyperellipticCurve_finite_field method*), 659
count_points_matrix_traces() (*sage.schemes.hyperelliptic_curves.hyperelliptic_finite_field.HyperellipticCurve_finite_field method*), 660
CPS_height_bound() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 270
create_element() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialCubicQuotientRing method*), 699
create_key_and_extra_args() (*sage.schemes.elliptic_curves.constructor.EllipticCurveFactory method*), 4
create_object() (*sage.schemes.elliptic_curves.constructor.EllipticCurveFactory method*), 5
cremona_curves() (*in module sage.schemes.elliptic_curves.ell_rational_field*), 359
cremona_label() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 282
cremona_optimal_curves() (*in module sage.schemes.elliptic_curves.ell_rational_field*), 359
curve() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint method*), 22
curve() (*sage.schemes.elliptic_curves.ell_tate_curve.TateCurve method*), 504
curve() (*sage.schemes.elliptic_curves.ell_torsion.EllipticCurveTorsionSubgroup method*), 439
curve() (*sage.schemes.elliptic_curves.formal_group.EllipticCurveFormalGroup method*), 153
curve() (*sage.schemes.elliptic_curves.heegner.HeegnerPointOnEllipticCurve method*), 566
curve() (*sage.schemes.elliptic_curves.heegner.KolyvaginPoint method*), 593
curve() (*sage.schemes.elliptic_curves.height.EllipticCurveCanonicalHeight method*), 418
curve() (*sage.schemes.elliptic_curves.modular_parametrization.ModularParameterization method*), 535
curve() (*sage.schemes.elliptic_curves.period_lattice.PeriodLattice_ell method*), 516
curve() (*sage.schemes.hyperelliptic_curves.jacobian_homset.JacobianHomset_divisor_classes method*), 724
curve() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialHyperellipticQuotientRing method*), 705
curve_key() (*in module sage.schemes.elliptic_curves.ell_egros*), 631
curve_over_ram_extn() (*sage.schemes.hyperelliptic_curves.hyperelliptic_padic_field.HyperellipticCurve_padic_field method*), 676
curves_with_j_0() (*in module sage.schemes.elliptic_curves.ell_finite_field*), 146
curves_with_j_0_char2() (*in module sage.schemes.elliptic_curves.ell_finite_field*), 146
curves_with_j_0_char3() (*in module sage.schemes.elliptic_curves.ell_finite_field*), 147
curves_with_j_1728() (*in module sage.schemes.elliptic_curves.ell_finite_field*), 149
cyclic_subideal_p1() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlg method*), 580

D

data (*sage.schemes.elliptic_curves.period_lattice_region.PeriodicRegion attribute*), 531
database_attributes() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 283
database_curve() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 283
DE() (*sage.schemes.elliptic_curves.height.EllipticCurveCanonicalHeight method*), 414
deg_one_primes_iter() (*in module sage.schemes.elliptic_curves.gal_reps_num-*

ber_field), 460
degree() (*sage.schemes.elliptic_curves.hom_scalar.EllipticCurveHom_scalar method*), 225
degree() (*sage.schemes.elliptic_curves.hom_sum.EllipticCurveHom_sum method*), 182
degree() (*sage.schemes.elliptic_curves.hom.EllipticCurveHom method*), 160
degree() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.MonskyWashnitzerDifferentialRing method*), 694
degree() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialHyperellipticQuotientRing method*), 705
degree_over_H() (*sage.schemes.elliptic_curves.heegner.RingClassField method*), 597
degree_over_K() (*sage.schemes.elliptic_curves.heegner.RingClassField method*), 598
degree_over_Q() (*sage.schemes.elliptic_curves.heegner.RingClassField method*), 598
deriv_at1() (*sage.schemes.elliptic_curves.lseries_ell.Lseries_ell method*), 550
descend_to() (*sage.schemes.elliptic_curves.ell_field.EllipticCurve_field method*), 91
diff() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialHyperellipticQuotientElement method*), 703
differential() (*sage.schemes.elliptic_curves.formal_group.EllipticCurveFormalGroup method*), 153
differential_operator() (*in module sage.schemes.hyperelliptic_curves.invariants*), 734
diffsymb() (*in module sage.schemes.hyperelliptic_curves.invariants*), 735
diffxy() (*in module sage.schemes.hyperelliptic_curves.invariants*), 735
dimension() (*sage.schemes.hyperelliptic_curves.jacobian_generic.HyperellipticJacobian_generic method*), 721
dimension() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.MonskyWashnitzerDifferentialRing method*), 695
discrete_log() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_finite_field method*), 41
discriminant() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 65
discriminant() (*sage.schemes.elliptic_curves.heegner.HeegnerPoint method*), 564
discriminant() (*sage.schemes.elliptic_curves.heegner.HeegnerPoints_level_disc method*), 576
discriminant_of_K() (*sage.schemes.elliptic_curves.heegner.RingClassField method*), 598
discriminant_valuation() (*sage.schemes.elliptic_curves.ell_local_data.EllipticCurveLocalData method*), 496
discriminants() (*sage.schemes.elliptic_curves.heegner.HeegnerPoints_level method*), 575
discriminants_with_bounded_class_number() (*in module sage.schemes.elliptic_curves.cm*), 483
division_field() (*sage.schemes.elliptic_curves.ell_field.EllipticCurve_field method*), 92
division_points() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_field method*), 27
division_polynomial() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 66
division_polynomial_0() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 67
divisors_to_string() (*in module sage.interfaces.genus2reduction*), 740
dokchitser() (*sage.schemes.elliptic_curves.lseries_ell.Lseries_ell method*), 551
domain() (*sage.schemes.elliptic_curves.heegner.GaloisAutomorphism method*), 558
domain() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlgEmbedding method*), 590
domain_conductor() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlgEmbedding method*), 590
domain_gen() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlgEmbedding method*), 591
Dp_valued_height() (*sage.schemes.elliptic_curves.padic_lseries.pAdicLseriesSupersingular method*), 622
Dp_valued_regulator() (*sage.schemes.elliptic_curves.padic_lseries.pAdicLseriesSupersingular method*), 622
Dp_valued_series() (*sage.schemes.elliptic_curves.padic_lseries.pAdicLseriesSupersingular method*), 623
ds() (*sage.schemes.elliptic_curves.period_lattice_region.PeriodicRegion method*), 531
dual() (*sage.schemes.elliptic_curves.ell_curve_isogeny.EllipticCurveIsogeny method*), 199
dual() (*sage.schemes.elliptic_curves.hom_composite.EllipticCurveHom_composite method*), 175
dual() (*sage.schemes.elliptic_curves.hom_frobenius.El-*

lipticCurveHom_frobenius method), 231
dual() (*sage.schemes.elliptic_curves.hom_scalar.EllipticCurveHom_scalar method*), 225
dual() (*sage.schemes.elliptic_curves.hom_sum.EllipticCurveHom_sum method*), 182
dual() (*sage.schemes.elliptic_curves.hom_velusqrt.EllipticCurveHom_velusqrt method*), 217
dual() (*sage.schemes.elliptic_curves.hom.EllipticCurveHom method*), 161
dual() (*sage.schemes.elliptic_curves.weierstrass_morphism.WeierstrassIsomorphism method*), 188

E

E2() (*sage.schemes.elliptic_curves.ell_tate_curve.TateCurve method*), 503
e_log_RC() (*sage.schemes.elliptic_curves.period_lattice.PeriodLattice_ell method*), 516
e_p() (*sage.schemes.elliptic_curves.height.EllipticCurveCanonicalHeight method*), 418
egros_from_j() (*in module sage.schemes.elliptic_curves.ell_egros*), 631
egros_from_j_0() (*in module sage.schemes.elliptic_curves.ell_egros*), 632
egros_from_j_1728() (*in module sage.schemes.elliptic_curves.ell_egros*), 632
egros_from_jlist() (*in module sage.schemes.elliptic_curves.ell_egros*), 633
egros_get_j() (*in module sage.schemes.elliptic_curves.ell_egros*), 633
ei() (*sage.schemes.elliptic_curves.period_lattice.PeriodLattice_ell method*), 517
Element (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.MonskyWashnitzerDifferentialRing attribute*), 693
Element (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialCubicQuotientRing attribute*), 699
Element (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialHyperellipticQuotientRing attribute*), 704
ell() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlg method*), 580
ell_heegner_discriminants() (*in module sage.schemes.elliptic_curves.heegner*), 600
ell_heegner_discriminants_list() (*in module sage.schemes.elliptic_curves.heegner*), 600
ell_heegner_point() (*in module sage.schemes.elliptic_curves.heegner*), 600
elliptic_curve() (*sage.schemes.elliptic_curves.ell_modular_symbols.ModularSymbol method*), 538
elliptic_curve() (*sage.schemes.elliptic_curves.gal_reps_number_field.GaloisRepresentation method*), 457
elliptic_curve() (*sage.schemes.elliptic_curves.gal_reps.GaloisRepresentation method*), 442
elliptic_curve() (*sage.schemes.elliptic_curves.lseries_ell.Lseries_ell method*), 552
elliptic_curve() (*sage.schemes.elliptic_curves.mod_sym_num.ModularSymbolNumerical method*), 546
elliptic_curve() (*sage.schemes.elliptic_curves.padic_lseries.pAdicLseries method*), 614
elliptic_curve_congruence_graph() (*in module sage.schemes.elliptic_curves.ell_rational_field*), 360
elliptic_exponential() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 283
elliptic_exponential() (*sage.schemes.elliptic_curves.period_lattice.PeriodLattice_ell method*), 518
elliptic_logarithm() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_number_field method*), 46
elliptic_logarithm() (*sage.schemes.elliptic_curves.period_lattice.PeriodLattice_ell method*), 520
EllipticCurve_field (*class in sage.schemes.elliptic_curves.ell_field*), 91
EllipticCurve_finite_field (*class in sage.schemes.elliptic_curves.ell_finite_field*), 121
EllipticCurve_from_c4c6() (*in module sage.schemes.elliptic_curves.constructor*), 6
EllipticCurve_from_cubic() (*in module sage.schemes.elliptic_curves.constructor*), 6
EllipticCurve_from_j() (*in module sage.schemes.elliptic_curves.constructor*), 11
EllipticCurve_from_Weierstrass_polynomial() (*in module sage.schemes.elliptic_curves.constructor*), 5
EllipticCurve_generic (*class in sage.schemes.elliptic_curves.ell_generic*), 59
EllipticCurve_number_field (*class in sage.schemes.elliptic_curves.ell_number_field*), 363
EllipticCurve_padic_field (*class in sage.schemes.elliptic_curves.ell_padic_field*), 634
EllipticCurve_rational_field (*class in sage.schemes.elliptic_curves.ell_rational_field*), 269
EllipticCurve_with_order() (*in module sage.schemes.elliptic_curves.ell_finite_field*), 144

EllipticCurveCanonicalHeight (*class in sage.schemes.elliptic_curves.height*), 414
EllipticCurveFactory (*class in sage.schemes.elliptic_curves.constructor*), 1
EllipticCurveFormalGroup (*class in sage.schemes.elliptic_curves.formal_group*), 153
EllipticCurveHom (*class in sage.schemes.elliptic_curves.hom*), 159
EllipticCurveHom_composite (*class in sage.schemes.elliptic_curves.hom_composite*), 174
EllipticCurveHom_frobenius (*class in sage.schemes.elliptic_curves.hom_frobenius*), 230
EllipticCurveHom_scalar (*class in sage.schemes.elliptic_curves.hom_scalar*), 225
EllipticCurveHom_sum (*class in sage.schemes.elliptic_curves.hom_sum*), 181
EllipticCurveHom_velusqrt (*class in sage.schemes.elliptic_curves.hom_velusqrt*), 217
EllipticCurveIsogeny (*class in sage.schemes.elliptic_curves.ell_curve_isogeny*), 194
EllipticCurveLocalData (*class in sage.schemes.elliptic_curves.ell_local_data*), 494
EllipticCurvePoint (*class in sage.schemes.elliptic_curves.ell_point*), 22
EllipticCurvePoint_field (*class in sage.schemes.elliptic_curves.ell_point*), 23
EllipticCurvePoint_finite_field (*class in sage.schemes.elliptic_curves.ell_point*), 40
EllipticCurvePoint_number_field (*class in sage.schemes.elliptic_curves.ell_point*), 44
EllipticCurves (*class in sage.schemes.elliptic_curves.ec_database*), 361
EllipticCurves_with_good_reduction_outside_S() (*in module sage.schemes.elliptic_curves.constructor*), 12
EllipticCurveSaturator (*class in sage.schemes.elliptic_curves.saturation*), 432
EllipticCurveTorsionSubgroup (*class in sage.schemes.elliptic_curves.ell_torsion*), 437
endomorphism_discriminant_from_class_number() (*sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field method*), 127
endomorphism_ring_is_commutative() (*sage.schemes.elliptic_curves.ell_field.EllipticCurve_field method*), 96
eps() (*in module sage.schemes.elliptic_curves.height*), 430
eval_modular_form() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 285
expand() (*sage.schemes.elliptic_curves.period_lattice_region.PeriodicRegion method*), 532
extended_agm_iteration() (*in module sage.schemes.elliptic_curves.period_lattice*), 528
extract_pow_y() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.MonskyWashnitzerDifferential method*), 690
extract_pow_y() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialHyperellipticQuotientElement method*), 703

F

factors() (*sage.schemes.elliptic_curves.hom_composite.EllipticCurveHom_composite method*), 176
faltings_height() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 285
FastEllipticPolynomial (*class in sage.schemes.elliptic_curves.hom_velusqrt*), 220
field() (*sage.schemes.elliptic_curves.heegner.GaloisGroup method*), 563
fill_isogeny_matrix() (*in module sage.schemes.elliptic_curves.ell_curve_isogeny*), 210
fill_ss_j_dict() (*in module sage.schemes.elliptic_curves.ell_finite_field*), 149
find_char_zero_weier_point() (*sage.schemes.hyperelliptic_curves.hyperelliptic_padic_field.HyperellipticCurve_padic_field method*), 676
find_post_isomorphism() (*in module sage.schemes.elliptic_curves.hom*), 171
finite_endpoints() (*sage.schemes.elliptic_curves.height.UnionOfIntervals method*), 428
fk_intervals() (*sage.schemes.elliptic_curves.height.EllipticCurveCanonicalHeight method*), 419
formal() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 69
formal() (*sage.schemes.elliptic_curves.hom_composite.EllipticCurveHom_composite method*), 176
formal() (*sage.schemes.elliptic_curves.hom.EllipticCurveHom method*), 161
formal_group() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 70
Fricke_module() (*in module sage.schemes.elliptic_curves.isogeny_small_degree*), 235

Fricke_polynomial() (*in module sage.schemes.elliptic_curves.isogeny_small_degree*), 235
frob_basis_elements() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.MonskyWashnitzerDifferentialRing method*), 695
frob_invariant_differential() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.MonskyWashnitzerDifferentialRing method*), 696
frob_Q() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.MonskyWashnitzerDifferentialRing method*), 695
frobenius() (*sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field method*), 128
frobenius() (*sage.schemes.elliptic_curves.ell_padic_field.EllipticCurve_padic_field method*), 634
frobenius() (*sage.schemes.elliptic_curves.padic_lseries.pAdicLseriesSupersingular method*), 623
frobenius() (*sage.schemes.hyperelliptic_curves.hyperelliptic_padic_field.HyperellipticCurve_padic_field method*), 677
frobenius_discriminant() (*sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field method*), 128
frobenius_endomorphism() (*sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field method*), 128
frobenius_expansion_by_newton() (*in module sage.schemes.hyperelliptic_curves.monsky_washnitzer*), 708
frobenius_expansion_by_series() (*in module sage.schemes.hyperelliptic_curves.monsky_washnitzer*), 709
Frobenius_filter() (*in module sage.schemes.elliptic_curves.gal_reps_number_field*), 456
frobenius_isogeny() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 70
frobenius_matrix() (*sage.schemes.hyperelliptic_curves.hyperelliptic_finite_field.HyperellipticCurve_finite_field method*), 660
frobenius_matrix_hypellfrob() (*sage.schemes.hyperelliptic_curves.hyperelliptic_finite_field.HyperellipticCurve_finite_field method*), 661
frobenius_order() (*sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field method*), 129
frobenius_polynomial() (*sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field method*), 129
frobenius_polynomial() (*sage.schemes.hyperelliptic_curves.hyperelliptic_finite_field.HyperellipticCurve_finite_field method*), 662
frobenius_polynomial_cardinalities() (*sage.schemes.hyperelliptic_curves.hyperelliptic_finite_field.HyperellipticCurve_finite_field method*), 663
frobenius_polynomial_matrix() (*sage.schemes.hyperelliptic_curves.hyperelliptic_finite_field.HyperellipticCurve_finite_field method*), 664
frobenius_polynomial_pari() (*sage.schemes.hyperelliptic_curves.hyperelliptic_finite_field.HyperellipticCurve_finite_field method*), 664
from_factors() (*sage.schemes.elliptic_curves.hom_composite.EllipticCurveHom_composite class method*), 176
full (*sage.schemes.elliptic_curves.period_lattice_region.PeriodicRegion attribute*), 532
full_p_saturation() (*sage.schemes.elliptic_curves.saturation.EllipticCurveSaturator method*), 434

G

galois_group() (*sage.schemes.elliptic_curves.heegner.RingClassField method*), 598
galois_group_over_hilbert_class_field() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlg method*), 580
galois_group_over_quadratic_field() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlg method*), 581
galois_orbit_over_K() (*sage.schemes.elliptic_curves.heegner.HeegnerPointOnX0N method*), 573
galois_representation() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 365
galois_representation() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 286
GaloisAutomorphism (*class in sage.schemes.elliptic_curves.heegner*), 558
GaloisAutomorphismComplexConjugation (*class in sage.schemes.elliptic_curves.heegner*), 559
GaloisAutomorphismQuadraticForm (*class in sage.schemes.elliptic_curves.heegner*), 559
GaloisGroup (*class in sage.schemes.elliptic_curves.heegner*), 561
GaloisRepresentation (*class in sage.schemes.elliptic_curves.gal_reps*), 442

GaloisRepresentation (*class in sage.schemes.elliptic_curves.gal_reps_number_field*), 457
gen() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 70
gens() (*sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field method*), 129
gens() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 71
gens() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 366
gens() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 287
gens() (*sage.schemes.elliptic_curves.period_lattice.PeriodLattice_ell method*), 523
gens() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialCubicQuotientRing method*), 699
gens() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialHyperellipticQuotientRing method*), 705
gens_certain() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 288
gens_quadratic() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 367
genus() (*sage.schemes.elliptic_curves.ell_field.EllipticCurve_field method*), 96
genus() (*sage.schemes.hyperelliptic_curves.hyperelliptic_generic.HyperellipticCurve_generic method*), 645
Genus2reduction (*class in sage.interfaces.genus2reduction*), 737
geometric_endomorphism_algebra_is_field() (*sage.schemes.hyperelliptic_curves.jacobian_generic.HyperellipticJacobian_generic method*), 721
geometric_endomorphism_ring_is_ZZ() (*sage.schemes.hyperelliptic_curves.jacobian_generic.HyperellipticJacobian_generic method*), 722
get_boundary_point() (*sage.schemes.hyperelliptic_curves.hyperelliptic_padic_field.HyperellipticCurve_padic_field method*), 677
global_integral_model() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 368
global_integral_model() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 288
global_minimal_model() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 369
global_minimality_class() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 371
graph() (*sage.schemes.elliptic_curves.isogeny_class.IsogenyClass_EC method*), 463
group_law() (*sage.schemes.elliptic_curves.formal_group.EllipticCurveFormalGroup method*), 154

H

has_additive_reduction() (*sage.schemes.elliptic_curves.ell_local_data.EllipticCurveLocalData method*), 496
has_additive_reduction() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 372
has_bad_reduction() (*sage.schemes.elliptic_curves.ell_local_data.EllipticCurveLocalData method*), 496
has_bad_reduction() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 372
has_cm() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 373
has_cm() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 288
has_finite_order() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_field method*), 29
has_finite_order() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_finite_field method*), 41
has_finite_order() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_number_field method*), 48
has_global_minimal_model() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 373
has_good_reduction() (*sage.schemes.elliptic_curves.ell_local_data.EllipticCurveLocalData method*), 497
has_good_reduction() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 373
has_good_reduction() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_number_field method*), 49
has_good_reduction_outside_S() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*),

289
has_infinite_order() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_field method*), 29
has_infinite_order() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_number_field method*), 49
has_multiplicative_reduction() (*sage.schemes.elliptic_curves.ell_local_data.EllipticCurveLocalData method*), 497
has_multiplicative_reduction() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 374
has_nonsplit_multiplicative_reduction() (*sage.schemes.elliptic_curves.ell_local_data.EllipticCurveLocalData method*), 497
has_nonsplit_multiplicative_reduction() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 374
has_odd_degree_model() (*sage.schemes.hyperelliptic_curves.hyperelliptic_generic.HyperellipticCurve_generic method*), 645
has_order() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_field method*), 30
has_rational_cm() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 375
has_rational_cm() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 289
has_split_multiplicative_reduction() (*sage.schemes.elliptic_curves.ell_local_data.EllipticCurveLocalData method*), 498
has_split_multiplicative_reduction() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 376
hasse_invariant() (*sage.schemes.elliptic_curves.ell_field.EllipticCurve_field method*), 97
Hasse_Witt() (*sage.schemes.hyperelliptic_curves.hyperelliptic_finite_field.HyperellipticCurve_finite_field method*), 654
heegner_conductors() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlg method*), 581
heegner_discriminants() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 290
heegner_discriminants() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlg method*), 581
heegner_discriminants_list() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 291
heegner_divisor() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlg method*), 582
heegner_index() (*in module sage.schemes.elliptic_curves.heegner*), 601
heegner_index() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 291
heegner_index_bound() (*in module sage.schemes.elliptic_curves.heegner*), 603
heegner_index_bound() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 292
heegner_point() (*in module sage.schemes.elliptic_curves.heegner*), 604
heegner_point() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 293
heegner_point() (*sage.schemes.elliptic_curves.heegner.KolyvaginCohomologyClass method*), 592
heegner_point() (*sage.schemes.elliptic_curves.heegner.KolyvaginPoint method*), 593
heegner_point_height() (*in module sage.schemes.elliptic_curves.heegner*), 604
heegner_point_height() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 294
heegner_point_on_X0N() (*sage.schemes.elliptic_curves.heegner.HeegnerPointOnEllipticCurve method*), 567
heegner_points() (*in module sage.schemes.elliptic_curves.heegner*), 605
heegner_sha_an() (*in module sage.schemes.elliptic_curves.heegner*), 605
heegner_sha_an() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 295
HeegnerPoint (*class in sage.schemes.elliptic_curves.heegner*), 564
HeegnerPointOnEllipticCurve (*class in sage.schemes.elliptic_curves.heegner*), 566
HeegnerPointOnX0N (*class in sage.schemes.elliptic_curves.heegner*), 572
HeegnerPoints (*class in sage.schemes.elliptic_curves.heegner*), 574
HeegnerPoints_level (*class in sage.schemes.elliptic_curves.heegner*), 575
HeegnerPoints_level_disc (*class in sage.schemes.elliptic_curves.heegner*), 576

HeegnerPoints_level_disc_cond (*class in sage.schemes.elliptic_curves.heegner*), 577
HeegnerQuatAlg (*class in sage.schemes.elliptic_curves.heegner*), 579
HeegnerQuatAlgEmbedding (*class in sage.schemes.elliptic_curves.heegner*), 589
height() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_number_field method*), 50
height() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 296
height_above_floor() (*sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field method*), 131
height_function() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 376
height_pairing_matrix() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 377
helper_matrix() (*in module sage.schemes.hyperelliptic_curves.monsky_washnitzer*), 710
helper_matrix() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.MonskyWashnitzerDifferentialRing method*), 697
hilbert_class_polynomial() (*in module sage.schemes.elliptic_curves.cm*), 484
hypellfrob() (*in module sage.schemes.hyperelliptic_curves.hypellfrob*), 717
hyperelliptic_polynomials() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 71
hyperelliptic_polynomials() (*sage.schemes.hyperelliptic_curves.hyperelliptic_generic.HyperellipticCurve_generic method*), 645
HyperellipticCurve() (*in module sage.schemes.hyperelliptic_curves.constructor*), 641
HyperellipticCurve_finite_field (*class in sage.schemes.hyperelliptic_curves.hyperelliptic_finite_field*), 653
HyperellipticCurve_from_invariants() (*in module sage.schemes.hyperelliptic_curves.mestre*), 685
HyperellipticCurve_g2 (*class in sage.schemes.hyperelliptic_curves.hyperelliptic_g2*), 730
HyperellipticCurve_generic (*class in sage.schemes.hyperelliptic_curves.hyperelliptic_generic*), 644
HyperellipticCurve_padic_field (*class in sage.schemes.hyperelliptic_curves.hyperelliptic_padic_field*), 667
HyperellipticCurve_rational_field (*class in sage.schemes.hyperelliptic_curves.hyperelliptic_rational_field*), 684
HyperellipticJacobian_g2 (*class in sage.schemes.hyperelliptic_curves.jacobian_g2*), 723
HyperellipticJacobian_generic (*class in sage.schemes.hyperelliptic_curves.jacobian_generic*), 719

I

ideal() (*sage.schemes.elliptic_curves.heegner.GaloisAutomorphismQuadraticForm method*), 560
identity_morphism() (*in module sage.schemes.elliptic_curves.weierstrass_morphism*), 191
identity_morphism() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 71
igusa_clebsch_invariants() (*in module sage.schemes.hyperelliptic_curves.invariants*), 735
igusa_clebsch_invariants() (*sage.schemes.hyperelliptic_curves.hyperelliptic_g2.HyperellipticCurve_g2 method*), 731
igusa_to_clebsch() (*in module sage.schemes.hyperelliptic_curves.invariants*), 736
image_classes() (*sage.schemes.elliptic_curves.gal_reps.GaloisRepresentation method*), 442
image_type() (*sage.schemes.elliptic_curves.gal_reps.GaloisRepresentation method*), 444
index() (*sage.schemes.elliptic_curves.heegner.KolyvaginPoint method*), 593
index() (*sage.schemes.elliptic_curves.isogeny_class.IsogenyClass_EC method*), 463
inf_max_abs() (*in module sage.schemes.elliptic_curves.height*), 430
init() (*in module sage.schemes.elliptic_curves.gp_simon*), 635
innermost_point() (*sage.schemes.elliptic_curves.period_lattice_region.PeriodicRegion method*), 532
inseparable_degree() (*sage.schemes.elliptic_curves.ell_curve_isogeny.EllipticCurveIsogeny method*), 201
inseparable_degree() (*sage.schemes.elliptic_curves.hom_composite.EllipticCurveHom_composite method*), 177
inseparable_degree() (*sage.schemes.elliptic_curves.hom_frobenius.EllipticCurveHom_frobenius method*), 232
inseparable_degree() (*sage.schemes.elliptic_curves.hom_scalar.EllipticCurveHom_scalar method*), 226

inseparable_degree() (*sage.schemes.elliptic_curves.hom_sum.EllipticCurveHom_sum method*), 183
inseparable_degree() (*sage.schemes.elliptic_curves.hom_velusqrt.EllipticCurveHom_velusqrt method*), 218
inseparable_degree() (*sage.schemes.elliptic_curves.hom.EllipticCurveHom method*), 161
inseparable_degree() (*sage.schemes.elliptic_curves.weierstrass_morphism.WeierstrassIsomorphism method*), 188
integral_model() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 378
integral_model() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 296
integral_points() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 296
integral_points_with_bounded_mw_coeffs() (*in module sage.schemes.elliptic_curves.ell_rational_field*), 360
integral_short_weierstrass_model() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 298
integral_x_coords_in_interval() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 298
integrate() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.MonskyWashnitzerDifferential method*), 690
intersection() (*sage.schemes.elliptic_curves.height.UnionOfIntervals class method*), 428
interval_products() (*in module sage.schemes.hyperelliptic_curves.hypellfrob*), 718
intervals() (*sage.schemes.elliptic_curves.height.UnionOfIntervals method*), 428
invariant_differential() (*sage.schemes.hyperelliptic_curves.hyperelliptic_generic.HyperellipticCurve_generic method*), 645
invariant_differential() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.MonskyWashnitzerDifferentialRing method*), 697
inverse() (*sage.schemes.elliptic_curves.formal_group.EllipticCurveFormalGroup method*), 154
inverse() (*sage.schemes.elliptic_curves.weierstrass_transform.WeierstrassTransformationWithInverse_class method*), 639
is_cm_j_invariant() (*in module sage.schemes.elliptic_curves.cm*), 486
is_crystalline() (*sage.schemes.elliptic_curves.gal_reps.GaloisRepresentation method*), 446
is_divisible_by() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_field method*), 31
is_EllipticCurve() (*in module sage.schemes.elliptic_curves.ell_generic*), 90
is_empty() (*sage.schemes.elliptic_curves.height.UnionOfIntervals method*), 429
is_empty() (*sage.schemes.elliptic_curves.period_lattice_region.PeriodicRegion method*), 533
is_field() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialHyperellipticQuotientRing method*), 705
is_finite_order() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_field method*), 32
is_global_integral_model() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 381
is_global_integral_model() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 298
is_global_minimal_model() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 381
is_good() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 299
is_HCP() (*in module sage.schemes.elliptic_curves.cm*), 485
is_HyperellipticCurve() (*in module sage.schemes.hyperelliptic_curves.hyperelliptic_generic*), 653
is_identity() (*sage.schemes.elliptic_curves.weierstrass_morphism.baseWI method*), 191
is_identity() (*sage.schemes.elliptic_curves.weierstrass_morphism.WeierstrassIsomorphism method*), 188
is_in_weierstrass_disc() (*sage.schemes.hyperelliptic_curves.hyperelliptic_padic_field.HyperellipticCurve_padic_field method*), 678
is_inert() (*in module sage.schemes.elliptic_curves.heegner*), 606
is_injective() (*sage.schemes.elliptic_curves.hom.EllipticCurveHom method*), 162
is_integral() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*),

299
is_irreducible() (*sage.schemes.elliptic_curves.gal_reps.GaloisRepresentation method*), 447
is_isogenous() (*sage.schemes.elliptic_curves.ell_field.EllipticCurve_field method*), 97
is_isogenous() (*sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field method*), 131
is_isogenous() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 382
is_isogenous() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 299
is_isomorphic() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 71
is_j_supersingular() (*in module sage.schemes.elliptic_curves.ell_finite_field*), 149
is_kernel_polynomial() (*in module sage.schemes.elliptic_curves.isogeny_small_degree*), 236
is_kolyvagin() (*sage.schemes.elliptic_curves.heegner.GaloisGroup method*), 563
is_kolyvagin_conductor() (*in module sage.schemes.elliptic_curves.heegner*), 607
is_local_integral_model() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 383
is_local_integral_model() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 300
is_minimal() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 300
is_normalized() (*sage.schemes.elliptic_curves.hom.EllipticCurveHom method*), 163
is_odd_degree() (*sage.schemes.hyperelliptic_curves.hyperelliptic_g2.HyperellipticCurve_g2 method*), 731
is_on_curve() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 72
is_on_identity_component() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_number_field method*), 54
is_ordinary() (*sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field method*), 133
is_ordinary() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 300
is_ordinary() (*sage.schemes.elliptic_curves.gal_reps.GaloisRepresentation method*), 447
is_ordinary() (*sage.schemes.elliptic_curves.padic_lseries.pAdicLseriesOrdinary method*), 617
is_ordinary() (*sage.schemes.elliptic_curves.padic_lseries.pAdicLseriesSupersingular method*), 624
is_p_integral() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 301
is_p_minimal() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 301
is_possible_j() (*in module sage.schemes.elliptic_curves.ell_egros*), 634
is_potentially_crystalline() (*sage.schemes.elliptic_curves.gal_reps.GaloisRepresentation method*), 447
is_potentially_semistable() (*sage.schemes.elliptic_curves.gal_reps.GaloisRepresentation method*), 448
is_Q_curve() (*in module sage.schemes.elliptic_curves.Qcurves*), 490
is_Q_curve() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 379
is_quadratic_twist() (*sage.schemes.elliptic_curves.ell_field.EllipticCurve_field method*), 98
is_quartic_twist() (*sage.schemes.elliptic_curves.ell_field.EllipticCurve_field method*), 100
is_quasi_unipotent() (*sage.schemes.elliptic_curves.gal_reps.GaloisRepresentation method*), 448
is_ramified() (*in module sage.schemes.elliptic_curves.heegner*), 607
is_real() (*sage.schemes.elliptic_curves.period_lattice.PeriodLattice_ell method*), 523
is_rectangular() (*sage.schemes.elliptic_curves.period_lattice.PeriodLattice_ell method*), 524
is_reducible() (*sage.schemes.elliptic_curves.gal_reps.GaloisRepresentation method*), 448
is_same_disc() (*sage.schemes.hyperelliptic_curves.hyperelliptic_padic_field.HyperellipticCurve_padic_field method*), 678
is_semistable() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 301
is_semistable() (*sage.schemes.ellip-*

tic_curves.gal_reps.GaloisRepresentation method), 449
is_separable() (*sage.schemes.elliptic_curves.hom.EllipticCurveHom method*), 164
is_sextic_twist() (*sage.schemes.elliptic_curves.ell_field.EllipticCurve_field method*), 100
is_singular() (*sage.schemes.hyperelliptic_curves.hyperelliptic_generic.HyperellipticCurve_generic method*), 645
is_smooth() (*sage.schemes.hyperelliptic_curves.hyperelliptic_generic.HyperellipticCurve_generic method*), 646
is_split() (*in module sage.schemes.elliptic_curves.heegner*), 607
is_split() (*sage.schemes.elliptic_curves.ell_tate_curve.TateCurve method*), 504
is_subfield() (*sage.schemes.elliptic_curves.heegner.RingClassField method*), 599
is_supersingular() (*sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field method*), 133
is_supersingular() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 302
is_supersingular() (*sage.schemes.elliptic_curves.padic_lseries.pAdicLseriesOrdinary method*), 617
is_supersingular() (*sage.schemes.elliptic_curves.padic_lseries.pAdicLseriesSupersingular method*), 624
is_surjective() (*sage.schemes.elliptic_curves.gal_reps_number_field.GaloisRepresentation method*), 457
is_surjective() (*sage.schemes.elliptic_curves.gal_reps.GaloisRepresentation method*), 449
is_surjective() (*sage.schemes.elliptic_curves.hom.EllipticCurveHom method*), 165
is_unipotent() (*sage.schemes.elliptic_curves.gal_reps.GaloisRepresentation method*), 450
is_unramified() (*sage.schemes.elliptic_curves.gal_reps.GaloisRepresentation method*), 451
is_weierstrass() (*sage.schemes.hyperelliptic_curves.hyperelliptic_padic_field.HyperellipticCurve_padic_field method*), 679
is_x_coord() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 72
is_x_coord() (*sage.schemes.hyperelliptic_curves.hyperelliptic_generic.HyperellipticCurve_generic method*), 646
is_zero() (*sage.schemes.elliptic_curves.hom.EllipticCurveHom method*), 166
isogenies() (*sage.schemes.elliptic_curves.isogeny_class.IsogenyClass_EC method*), 464
isogenies_2() (*in module sage.schemes.elliptic_curves.isogeny_small_degree*), 241
isogenies_3() (*in module sage.schemes.elliptic_curves.isogeny_small_degree*), 242
isogenies_5_0() (*in module sage.schemes.elliptic_curves.isogeny_small_degree*), 243
isogenies_5_1728() (*in module sage.schemes.elliptic_curves.isogeny_small_degree*), 244
isogenies_7_0() (*in module sage.schemes.elliptic_curves.isogeny_small_degree*), 246
isogenies_7_1728() (*in module sage.schemes.elliptic_curves.isogeny_small_degree*), 248
isogenies_13_0() (*in module sage.schemes.elliptic_curves.isogeny_small_degree*), 238
isogenies_13_1728() (*in module sage.schemes.elliptic_curves.isogeny_small_degree*), 240
isogenies_prime_degree() (*in module sage.schemes.elliptic_curves.isogeny_small_degree*), 249
isogenies_prime_degree() (*sage.schemes.elliptic_curves.ell_field.EllipticCurve_field method*), 101
isogenies_prime_degree() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 384
isogenies_prime_degree() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 302
isogenies_prime_degree_general() (*in module sage.schemes.elliptic_curves.isogeny_small_degree*), 252
isogenies_prime_degree_genus_0() (*in module sage.schemes.elliptic_curves.isogeny_small_degree*), 256
isogenies_prime_degree_genus_plus_0() (*in module sage.schemes.elliptic_curves.isogeny_small_degree*), 257
isogenies_prime_degree_genus_plus_0_j0() (*in module sage.schemes.elliptic_curves.isogeny_small_degree*), 260
isogenies_prime_degree_genus_plus_0_j1728() (*in module sage.schemes.elliptic_curves.isogeny_small_degree*), 262
isogenies_sporadic_Q() (*in module*

sage.schemes.elliptic_curves.isogeny_small_degree), 264
isogeny() (*sage.schemes.elliptic_curves.ell_field.EllipticCurve_field method*), 106
isogeny_bound() (*sage.schemes.elliptic_curves.gal_reps_number_field.GaloisRepresentation method*), 458
isogeny_class() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 385
isogeny_class() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 303
isogeny_codomain() (*sage.schemes.elliptic_curves.ell_field.EllipticCurve_field method*), 110
isogeny_codomain_from_kernel() (*in module sage.schemes.elliptic_curves.ell_curve_isogeny*), 211
isogeny_degree() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 390
isogeny_degree() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 306
isogeny_degrees_cm() (*in module sage.schemes.elliptic_curves.isogeny_class*), 467
isogeny_ell_graph() (*sage.schemes.elliptic_curves.ell_field.EllipticCurve_field method*), 110
isogeny_graph() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 307
IsogenyClass_EC (*class in sage.schemes.elliptic_curves.isogeny_class*), 463
IsogenyClass_EC_NumberField (*class in sage.schemes.elliptic_curves.isogeny_class*), 466
IsogenyClass_EC_Rational (*class in sage.schemes.elliptic_curves.isogeny_class*), 466
isomorphism() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 73
isomorphism_to() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 74
isomorphisms() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 75

J

j_invariant() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 76
Jacobian() (*in module sage.schemes.elliptic_curves.jacobian*), 17
jacobian() (*sage.schemes.hyperelliptic_curves.hyperelliptic_g2.HyperellipticCurve_g2 method*), 732
jacobian() (*sage.schemes.hyperelliptic_curves.hyperelliptic_generic.HyperellipticCurve_generic method*), 647
Jacobian_of_curve() (*in module sage.schemes.elliptic_curves.jacobian*), 18
Jacobian_of_equation() (*in module sage.schemes.elliptic_curves.jacobian*), 18
JacobianHomset_divisor_classes (*class in sage.schemes.hyperelliptic_curves.jacobian_homset*), 724
JacobianMorphism_divisor_class_field (*class in sage.schemes.hyperelliptic_curves.jacobian_morphism*), 726
join() (*sage.schemes.elliptic_curves.height.UnionOfIntervals static method*), 429

K

kernel_polynomial() (*sage.schemes.elliptic_curves.ell_curve_isogeny.EllipticCurveIsogeny method*), 201
kernel_polynomial() (*sage.schemes.elliptic_curves.hom_composite.EllipticCurveHom_composite method*), 177
kernel_polynomial() (*sage.schemes.elliptic_curves.hom_frobenius.EllipticCurveHom_frobenius method*), 232
kernel_polynomial() (*sage.schemes.elliptic_curves.hom_scalar.EllipticCurveHom_scalar method*), 226
kernel_polynomial() (*sage.schemes.elliptic_curves.hom_sum.EllipticCurveHom_sum method*), 184
kernel_polynomial() (*sage.schemes.elliptic_curves.hom_velusqrt.EllipticCurveHom_velusqrt method*), 218
kernel_polynomial() (*sage.schemes.elliptic_curves.hom.EllipticCurveHom method*), 166
kernel_polynomial() (*sage.schemes.elliptic_curves.weierstrass_morphism.WeierstrassIsomorphism method*), 189
kernel_polynomial_from_divisor() (*sage.schemes.elliptic_curves.ell_field.EllipticCurve_field method*), 112
kernel_polynomial_from_point() (*sage.schemes.elliptic_curves.ell_field.EllipticCurve_field method*), 113
kodaira_symbol() (*sage.schemes.elliptic_curves.ell_local_data.EllipticCurveLocal-*

Data method), 498
kodaira_symbol() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 391
kodaira_symbol() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 308
kodaira_type() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 308
kodaira_type_old() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 309
KodairaSymbol() (*in module sage.schemes.elliptic_curves.kodaira_symbol*), 502
KodairaSymbol_class (*class in sage.schemes.elliptic_curves.kodaira_symbol*), 502
kolyvagin_cohomology_class() (*sage.schemes.elliptic_curves.heegner.HeegnerPointOnEllipticCurve method*), 567
kolyvagin_cohomology_class() (*sage.schemes.elliptic_curves.heegner.KolyvaginPoint method*), 593
kolyvagin_conductors() (*sage.schemes.elliptic_curves.heegner.HeegnerPoints_level_disc method*), 576
kolyvagin_cyclic_subideals() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlg method*), 582
kolyvagin_generator() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlg method*), 583
kolyvagin_generators() (*sage.schemes.elliptic_curves.heegner.GaloisGroup method*), 563
kolyvagin_generators() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlg method*), 583
kolyvagin_point() (*in module sage.schemes.elliptic_curves.heegner*), 608
kolyvagin_point() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 309
kolyvagin_point() (*sage.schemes.elliptic_curves.heegner.HeegnerPointOnEllipticCurve method*), 567
kolyvagin_point() (*sage.schemes.elliptic_curves.heegner.KolyvaginCohomologyClass method*), 592
kolyvagin_point_on_curve() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlg method*), 584
kolyvagin_reduction_data() (*in module sage.schemes.elliptic_curves.heegner*), 608
kolyvagin_sigma_operator() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlg method*), 584
KolyvaginCohomologyClass (*class in sage.schemes.elliptic_curves.heegner*), 591
KolyvaginCohomologyClassEn (*class in sage.schemes.elliptic_curves.heegner*), 592
KolyvaginPoint (*class in sage.schemes.elliptic_curves.heegner*), 592
kummer_morphism() (*sage.schemes.hyperelliptic_curves.hyperelliptic_g2.HyperellipticCurve_g2 method*), 732
kummer_surface() (*sage.schemes.hyperelliptic_curves.jacobian_g2.HyperellipticJacobian_g2 method*), 723
KummerSurface (*class in sage.schemes.hyperelliptic_curves.kummer_surface*), 737

L

L1_vanishes() (*sage.schemes.elliptic_curves.lseries_ell.Lseries_ell method*), 548
L_invariant() (*sage.schemes.elliptic_curves.ell_tate_curve.TateCurve method*), 503
L_ratio() (*sage.schemes.elliptic_curves.lseries_ell.Lseries_ell method*), 548
label() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 309
Lambda() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 270
largest_disc_with_class_number() (*in module sage.schemes.elliptic_curves.cm*), 487
largest_fundamental_disc_with_class_number() (*in module sage.schemes.elliptic_curves.cm*), 488
left_orders() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlg method*), 586
level() (*sage.schemes.elliptic_curves.heegner.HeegnerPoint method*), 564
level() (*sage.schemes.elliptic_curves.heegner.HeegnerPoints method*), 575
level() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlg method*), 586
lift() (*in module sage.schemes.hyperelliptic_curves.monsky_washnitzer*), 710
lift() (*sage.schemes.elliptic_curves.ell_tate_curve.TateCurve method*), 504
lift_of_hilbert_class_field_galois_group() (*sage.schemes.elliptic_curves.heegner.GaloisGroup method*), 563

lift_x() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 76
lift_x() (*sage.schemes.hyperelliptic_curves.hyperelliptic_generic.HyperellipticCurve_generic method*), 647
lll_reduce() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 391
lmfdb_page() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 310
local_analytic_interpolation() (*sage.schemes.hyperelliptic_curves.hyperelliptic_padic_field.HyperellipticCurve_padic_field method*), 679
local_coord() (*sage.schemes.hyperelliptic_curves.hyperelliptic_generic.HyperellipticCurve_generic method*), 648
local_coordinates_at_infinity() (*sage.schemes.hyperelliptic_curves.hyperelliptic_generic.HyperellipticCurve_generic method*), 649
local_coordinates_at_nonweierstrass() (*sage.schemes.hyperelliptic_curves.hyperelliptic_generic.HyperellipticCurve_generic method*), 649
local_coordinates_at_weierstrass() (*sage.schemes.hyperelliptic_curves.hyperelliptic_generic.HyperellipticCurve_generic method*), 650
local_data() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 393
local_integral_model() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 395
local_integral_model() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 310
local_minimal_model() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 395
log() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_finite_field method*), 41
log() (*sage.schemes.elliptic_curves.formal_group.EllipticCurveFormalGroup method*), 155
lseries() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 310
lseries() (*sage.schemes.hyperelliptic_curves.hyperelliptic_rational_field.HyperellipticCurve_rational_field method*), 684
Lseries_ell (*class in sage.schemes.elliptic_curves.lseries_ell*), 548
lseries_gross_zagier() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 311

M

make_monic() (*in module sage.schemes.elliptic_curves.heegner*), 610
manin_constant() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 311
manin_symbol() (*sage.schemes.elliptic_curves.mod_sym_num.ModularSymbolNumerical method*), 546
map_to_complex_numbers() (*sage.schemes.elliptic_curves.heegner.HeegnerPointOnEllipticCurve method*), 567
map_to_complex_numbers() (*sage.schemes.elliptic_curves.modular_parametrization.ModularParameterization method*), 535
map_to_curve() (*sage.schemes.elliptic_curves.heegner.HeegnerPointOnX0N method*), 573
matrix() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlgEmbedding method*), 591
matrix() (*sage.schemes.elliptic_curves.isogeny_class.IsogenyClass_EC method*), 464
matrix_of_frobenius() (*in module sage.schemes.hyperelliptic_curves.monsky_washnitzer*), 710
matrix_of_frobenius() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 312
matrix_of_frobenius() (*sage.schemes.hyperelliptic_curves.hyperelliptic_rational_field.HyperellipticCurve_rational_field method*), 684
matrix_of_frobenius_hyperelliptic() (*in module sage.schemes.hyperelliptic_curves.monsky_washnitzer*), 714
matrix_on_subgroup() (*sage.schemes.elliptic_curves.hom.EllipticCurveHom method*), 166
max_pow_y() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.MonskyWashnitzerDifferential method*), 691
max_pow_y() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialHyperellipticQuotientElement method*), 703
ME() (*sage.schemes.elliptic_curves.height.EllipticCurveCanonicalHeight method*), 415
measure() (*sage.schemes.elliptic_curves.padic_lseries.pAdicLseries method*), 614
Mestre_conic() (*in module sage.schemes.hyperelliptic_curves.mestre*), 687

min() (*sage.schemes.elliptic_curves.height.EllipticCurveCanonicalHeight method*), 420
min_gr() (*sage.schemes.elliptic_curves.height.EllipticCurveCanonicalHeight method*), 421
min_on_disk() (*in module sage.schemes.elliptic_curves.height*), 431
min_pow_y() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.MonskyWashnitzerDifferential method*), 691
min_pow_y() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialHyperellipticQuotientElement method*), 703
minimal_discriminant_ideal() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 396
minimal_model() (*sage.schemes.elliptic_curves.ell_local_data.EllipticCurveLocalData method*), 498
minimal_model() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 313
minimal_quadratic_twist() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 313
mod() (*sage.schemes.elliptic_curves.heegner.KolyvaginPoint method*), 594
mod5family() (*in module sage.schemes.elliptic_curves.mod5family*), 635
mod5family() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 314
modp_dual_elliptic_curve_factor() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlg method*), 586
modp_splitting_data() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlg method*), 586
modp_splitting_map() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlg method*), 587
modular_degree() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 314
modular_form() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 316
modular_parametrization() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 316
modular_symbol() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 317
modular_symbol() (*sage.schemes.elliptic_curves.padic_lseries.pAdicLseries method*), 615
modular_symbol_numerical() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 319
modular_symbol_space() (*in module sage.schemes.elliptic_curves.ell_modular_symbols*), 541
modular_symbol_space() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 320
ModularParameterization (*class in sage.schemes.elliptic_curves.modular_parametrization*), 535
ModularSymbol (*class in sage.schemes.elliptic_curves.ell_modular_symbols*), 538
ModularSymbolECLIB (*class in sage.schemes.elliptic_curves.ell_modular_symbols*), 538
ModularSymbolNumerical (*class in sage.schemes.elliptic_curves.mod_sym_num*), 544
ModularSymbolSage (*class in sage.schemes.elliptic_curves.ell_modular_symbols*), 540
module
    sage.interfaces.genus2reduction, 737
    sage.schemes.elliptic_curves.cm, 480
    sage.schemes.elliptic_curves.constructor, 1
    sage.schemes.elliptic_curves.descent_two_isogeny, 627
    sage.schemes.elliptic_curves.ec_database, 361
    sage.schemes.elliptic_curves.ell_curve_isogeny, 193
    sage.schemes.elliptic_curves.ell_egros, 630
    sage.schemes.elliptic_curves.ell_field, 91
    sage.schemes.elliptic_curves.ell_finite_field, 121
    sage.schemes.elliptic_curves.ell_generic, 59
    sage.schemes.elliptic_curves.ell_local_data, 493
    sage.schemes.elliptic_curves.ell_modular_symbols, 537
    sage.schemes.elliptic_curves.ell_number_field, 362
    sage.schemes.elliptic_curves.ell_padic_field, 634
    sage.schemes.elliptic_curves.ell_point, 21

sage.schemes.elliptic_curves.ell_rational_field, 269
sage.schemes.elliptic_curves.ell_tate_curve, 503
sage.schemes.elliptic_curves.ell_torsion, 437
sage.schemes.elliptic_curves.ell_wp, 507
sage.schemes.elliptic_curves.formal_group, 153
sage.schemes.elliptic_curves.gal_reps, 441
sage.schemes.elliptic_curves.gal_reps_number_field, 452
sage.schemes.elliptic_curves.gp_simon, 635
sage.schemes.elliptic_curves.heegner, 557
sage.schemes.elliptic_curves.height, 414
sage.schemes.elliptic_curves.hom, 159
sage.schemes.elliptic_curves.hom_composite, 173
sage.schemes.elliptic_curves.hom_frobenius, 229
sage.schemes.elliptic_curves.hom_scalar, 223
sage.schemes.elliptic_curves.hom_sum, 181
sage.schemes.elliptic_curves.hom_velusqrt, 215
sage.schemes.elliptic_curves.isogeny_class, 463
sage.schemes.elliptic_curves.isogeny_small_degree, 235
sage.schemes.elliptic_curves.jacobian, 17
sage.schemes.elliptic_curves.kodaira_symbol, 501
sage.schemes.elliptic_curves.lseries_ell, 548
sage.schemes.elliptic_curves.mod5family, 635
sage.schemes.elliptic_curves.mod_poly, 267
sage.schemes.elliptic_curves.mod_sym_num, 542
sage.schemes.elliptic_curves.modular_parametrization, 535
sage.schemes.elliptic_curves.padic_lseries, 612
sage.schemes.elliptic_curves.period_lattice, 511
sage.schemes.elliptic_curves.period_lattice_region, 530
sage.schemes.elliptic_curves.Qcurves, 489
sage.schemes.elliptic_curves.saturation, 432
sage.schemes.elliptic_curves.sha_tate, 470
sage.schemes.elliptic_curves.weierstrass_morphism, 187
sage.schemes.elliptic_curves.weierstrass_transform, 636
sage.schemes.hyperelliptic_curves.constructor, 641
sage.schemes.hyperelliptic_curves.hypellfrob, 717
sage.schemes.hyperelliptic_curves.hyperelliptic_finite_field, 653
sage.schemes.hyperelliptic_curves.hyperelliptic_g2, 730
sage.schemes.hyperelliptic_curves.hyperelliptic_generic, 643
sage.schemes.hyperelliptic_curves.hyperelliptic_padic_field, 667
sage.schemes.hyperelliptic_curves.hyperelliptic_rational_field, 684
sage.schemes.hyperelliptic_curves.invariants, 732
sage.schemes.hyperelliptic_curves.jacobian_g2, 723
sage.schemes.hyperelliptic_curves.jacobian_generic, 719
sage.schemes.hyperelliptic_curves.jacobian_homset, 724
sage.schemes.hyperelliptic_curves.jacobian_morphism, 725
sage.schemes.hyperelliptic_curves.kummer_surface, 737
sage.schemes.hyperelliptic_curves.mestre, 685
sage.schemes.hyperelliptic_curves.monsky_washnitzer,

688
monomial() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialHyperellipticQuotientRing method*), 705
monomial_diff_coeffs() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialHyperellipticQuotientRing method*), 706
monomial_diff_coeffs_matrices() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialHyperellipticQuotientRing method*), 706
monsky_washnitzer() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialHyperellipticQuotientRing method*), 706
monsky_washnitzer_gens() (*sage.schemes.hyperelliptic_curves.hyperelliptic_generic.HyperellipticCurve_generic method*), 651
MonskyWashnitzerDifferential (*class in sage.schemes.hyperelliptic_curves.monsky_washnitzer*), 688
MonskyWashnitzerDifferentialRing (*class in sage.schemes.hyperelliptic_curves.monsky_washnitzer*), 693
MonskyWashnitzerDifferentialRing_class (*in module sage.schemes.hyperelliptic_curves.monsky_washnitzer*), 697
montgomery_model() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 79
mult_by_n() (*sage.schemes.elliptic_curves.formal_group.EllipticCurveFormalGroup method*), 155
multiplication_by_m() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 82
multiplication_by_m_isogeny() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 83
multiplication_by_p_isogeny() (*sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field method*), 134
mwrank() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 321
mwrank_curve() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 322

N

n() (*sage.schemes.elliptic_curves.heegner.KolyvaginCohomologyClass method*), 592
nearby_rational_poly() (*in module sage.schemes.elliptic_curves.heegner*), 610
negation_morphism() (*in module sage.schemes.elliptic_curves.weierstrass_morphism*), 192
newform() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 322
newton_sqrt() (*sage.schemes.hyperelliptic_curves.hyperelliptic_padic_field.HyperellipticCurve_padic_field method*), 680
ngens() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 322
non_archimedean_local_height() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_number_field method*), 55
non_minimal_primes() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 396
non_surjective() (*sage.schemes.elliptic_curves.gal_reps_number_field.GaloisRepresentation method*), 459
non_surjective() (*sage.schemes.elliptic_curves.gal_reps.GaloisRepresentation method*), 451
nonneg_region() (*in module sage.schemes.elliptic_curves.height*), 431
normalise_periods() (*in module sage.schemes.elliptic_curves.period_lattice*), 529
normalised_basis() (*sage.schemes.elliptic_curves.period_lattice.PeriodLattice_ell method*), 524
Np() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 271
numerical_approx() (*sage.schemes.elliptic_curves.heegner.HeegnerPointOnEllipticCurve method*), 568
numerical_approx() (*sage.schemes.elliptic_curves.heegner.KolyvaginPoint method*), 594

O

odd_degree_model() (*sage.schemes.hyperelliptic_curves.hyperelliptic_generic.HyperellipticCurve_generic method*), 651
omega() (*sage.schemes.elliptic_curves.period_lattice.PeriodLattice_ell method*), 525
one() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialCubicQuotientRing method*), 699
one() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialHyperellipticQuotientRing method*), 706
optimal_curve() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 323
optimal_embeddings() (*sage.schemes.ellip-*

tic_curves.heegner.HeegnerQuatAlg method), 588
order() (*sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field method*), 134
order() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_field method*), 32
order() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_finite_field method*), 42
order() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_number_field method*), 56
order() (*sage.schemes.elliptic_curves.heegner.GaloisAutomorphismComplexConjugation method*), 559
order() (*sage.schemes.elliptic_curves.heegner.GaloisAutomorphismQuadraticForm method*), 560
order() (*sage.schemes.elliptic_curves.weierstrass_morphism.WeierstrassIsomorphism method*), 189
order_of_vanishing() (*sage.schemes.elliptic_curves.padic_lseries.pAdicLseries method*), 616
OrderClassNumber() (*in module sage.schemes.elliptic_curves.cm*), 481
ordinary_primes() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 323
original_curve() (*sage.schemes.elliptic_curves.ell_tate_curve.TateCurve method*), 505

P

p1_element() (*sage.schemes.elliptic_curves.heegner.GaloisAutomorphismQuadraticForm method*), 561
p_primary_bound() (*sage.schemes.elliptic_curves.sha_tate.Sha method*), 478
p_primary_order() (*sage.schemes.elliptic_curves.sha_tate.Sha method*), 479
p_projections() (*in module sage.schemes.elliptic_curves.saturation*), 436
p_rank() (*sage.schemes.hyperelliptic_curves.hyperelliptic_finite_field.HyperellipticCurve_finite_field method*), 665
p_saturation() (*sage.schemes.elliptic_curves.saturation.EllipticCurveSaturator method*), 434
P_to_S() (*sage.schemes.hyperelliptic_curves.hyperelliptic_padic_field.HyperellipticCurve_padic_field method*), 667
padic_E2() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 324
padic_elliptic_logarithm() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_finite_field method*), 43
padic_elliptic_logarithm() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_number_field method*), 56
padic_height() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 326
padic_height() (*sage.schemes.elliptic_curves.ell_tate_curve.TateCurve method*), 505
padic_height_pairing_matrix() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 328
padic_height_via_multiply() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 329
padic_lseries() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 330
padic_regulator() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 332
padic_regulator() (*sage.schemes.elliptic_curves.ell_tate_curve.TateCurve method*), 505
padic_sigma() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 333
padic_sigma_truncated() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 335
pAdicLseries (*class in sage.schemes.elliptic_curves.padic_lseries*), 613
pAdicLseriesOrdinary (*class in sage.schemes.elliptic_curves.padic_lseries*), 617
pAdicLseriesSupersingular (*class in sage.schemes.elliptic_curves.padic_lseries*), 622
parameter() (*sage.schemes.elliptic_curves.ell_tate_curve.TateCurve method*), 506
parametrisation_onto_original_curve() (*sage.schemes.elliptic_curves.ell_tate_curve.TateCurve method*), 506
parametrisation_onto_tate_curve() (*sage.schemes.elliptic_curves.ell_tate_curve.TateCurve method*), 506
parent() (*sage.schemes.elliptic_curves.heegner.GaloisAutomorphism method*), 559
pari_curve() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 83

pari_curve() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 336
pari_mincurve() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 337
period_lattice() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 397
period_lattice() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 337
PeriodicRegion (*class in sage.schemes.elliptic_curves.period_lattice_region*), 530
PeriodLattice (*class in sage.schemes.elliptic_curves.period_lattice*), 512
PeriodLattice_ell (*class in sage.schemes.elliptic_curves.period_lattice*), 512
plot() (*sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field method*), 136
plot() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 84
plot() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_field method*), 32
plot() (*sage.schemes.elliptic_curves.heegner.HeegnerPointOnX0N method*), 574
plot() (*sage.schemes.elliptic_curves.heegner.HeegnerPoints_level_disc_cond method*), 578
plot() (*sage.schemes.elliptic_curves.heegner.KolyvaginPoint method*), 595
plot() (*sage.schemes.elliptic_curves.period_lattice_region.PeriodicRegion method*), 533
point() (*sage.schemes.hyperelliptic_curves.jacobian_generic.HyperellipticJacobian_generic method*), 723
point_exact() (*sage.schemes.elliptic_curves.heegner.HeegnerPointOnEllipticCurve method*), 569
point_exact() (*sage.schemes.elliptic_curves.heegner.KolyvaginPoint method*), 595
point_of_jacobian_of_curve() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_field method*), 33
point_of_jacobian_of_curve() (*sage.schemes.hyperelliptic_curves.jacobian_morphism.JacobianMorphism_divisor_class_field method*), 726
point_of_order() (*in module sage.schemes.elliptic_curves.ell_field*), 119
point_search() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 337
points() (*sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field method*), 136
points() (*sage.schemes.elliptic_curves.ell_torsion.EllipticCurveTorsionSubgroup method*), 439
points() (*sage.schemes.elliptic_curves.heegner.HeegnerPoints_level_disc_cond method*), 578
points() (*sage.schemes.hyperelliptic_curves.hyperelliptic_finite_field.HyperellipticCurve_finite_field method*), 666
pollack_stevens_modular_symbol() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 338
poly_ring() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialCubicQuotientRing method*), 699
possible_isogeny_degrees() (*in module sage.schemes.elliptic_curves.isogeny_class*), 468
post_rescaling() (*sage.schemes.elliptic_curves.weierstrass_transform.WeierstrassTransformation method*), 638
power_series() (*sage.schemes.elliptic_curves.modular_parametrization.ModularParameterization method*), 536
power_series() (*sage.schemes.elliptic_curves.padic_lseries.pAdicLseriesOrdinary method*), 617
power_series() (*sage.schemes.elliptic_curves.padic_lseries.pAdicLseriesSupersingular method*), 624
prime() (*sage.schemes.elliptic_curves.ell_local_data.EllipticCurveLocalData method*), 499
prime() (*sage.schemes.elliptic_curves.ell_tate_curve.TateCurve method*), 507
prime() (*sage.schemes.elliptic_curves.padic_lseries.pAdicLseries method*), 617
prime() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialHyperellipticQuotientRing method*), 707
projective_point() (*in module sage.schemes.elliptic_curves.constructor*), 15
prove_BSD() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 339
Psi() (*in module sage.schemes.elliptic_curves.isogeny_small_degree*), 235
psi() (*sage.schemes.elliptic_curves.height.EllipticCurveCanonicalHeight method*), 422
Psi2() (*in module sage.schemes.elliptic_curves.isogeny_small_degree*), 236

Q

Q() (*sage.schemes.hyperelliptic_curves.monsky_wash-*

nitzer.MonskyWashnitzerDifferentialRing method), 693
Q() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialHyperellipticQuotientRing method*), 704
q_eigenform() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 342
q_expansion() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 342
qf_matrix() (*sage.schemes.elliptic_curves.isogeny_class.IsogenyClass_EC method*), 465
quadratic_field() (*sage.schemes.elliptic_curves.heegner.HeegnerPoint method*), 565
quadratic_field() (*sage.schemes.elliptic_curves.heegner.HeegnerPoints_level_disc method*), 577
quadratic_field() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlg method*), 588
quadratic_field() (*sage.schemes.elliptic_curves.heegner.RingClassField method*), 599
quadratic_form() (*sage.schemes.elliptic_curves.heegner.GaloisAutomorphismQuadraticForm method*), 561
quadratic_form() (*sage.schemes.elliptic_curves.heegner.HeegnerPointOnEllipticCurve method*), 570
quadratic_form() (*sage.schemes.elliptic_curves.heegner.HeegnerPointOnX0N method*), 574
quadratic_order() (*in module sage.schemes.elliptic_curves.heegner*), 611
quadratic_order() (*sage.schemes.elliptic_curves.heegner.HeegnerPoint method*), 565
quadratic_twist() (*sage.schemes.elliptic_curves.ell_field.EllipticCurve_field method*), 115
quadratic_twist() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 342
quartic_twist() (*sage.schemes.elliptic_curves.ell_field.EllipticCurve_field method*), 116
quaternion_algebra() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlg method*), 588

R

ramified_primes() (*sage.schemes.elliptic_curves.heegner.RingClassField method*), 599
random_element() (*sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field method*), 137
random_point() (*sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field method*), 138
rank() (*sage.schemes.elliptic_curves.ec_database.EllipticCurves method*), 361
rank() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 398
rank() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 343
rank_bound() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 344
rank_bounds() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 399
rat_term_CIF() (*in module sage.schemes.elliptic_curves.height*), 432
rational_kolyvagin_divisor() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlg method*), 588
rational_maps() (*sage.schemes.elliptic_curves.ell_curve_isogeny.EllipticCurveIsogeny method*), 201
rational_maps() (*sage.schemes.elliptic_curves.hom_composite.EllipticCurveHom_composite method*), 177
rational_maps() (*sage.schemes.elliptic_curves.hom_frobenius.EllipticCurveHom_frobenius method*), 232
rational_maps() (*sage.schemes.elliptic_curves.hom_scalar.EllipticCurveHom_scalar method*), 226
rational_maps() (*sage.schemes.elliptic_curves.hom_sum.EllipticCurveHom_sum method*), 184
rational_maps() (*sage.schemes.elliptic_curves.hom_velusqrt.EllipticCurveHom_velusqrt method*), 218
rational_maps() (*sage.schemes.elliptic_curves.hom.EllipticCurveHom method*), 168
rational_maps() (*sage.schemes.elliptic_curves.weierstrass_morphism.WeierstrassIsomorphism method*), 189
rational_points() (*sage.schemes.ellip-*

tic_curves.ell_finite_field.EllipticCurve_finite_field method), 139
rational_points() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 400
rational_points() (*sage.schemes.hyperelliptic_curves.hyperelliptic_generic.HyperellipticCurve_generic method*), 652
real_components() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 401
real_components() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 344
real_intersection_is_empty() (*sage.schemes.elliptic_curves.height.EllipticCurveCanonicalHeight method*), 423
real_period() (*sage.schemes.elliptic_curves.period_lattice.PeriodLattice_ell method*), 526
reduce() (*sage.schemes.elliptic_curves.period_lattice.PeriodLattice_ell method*), 527
reduce() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.MonskyWashnitzerDifferential method*), 691
reduce_all() (*in module sage.schemes.hyperelliptic_curves.monsky_washnitzer*), 714
reduce_fast() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.MonskyWashnitzerDifferential method*), 691
reduce_mod() (*sage.schemes.elliptic_curves.heegner.HeegnerPoints_level method*), 575
reduce_mod_q() (*in module sage.schemes.elliptic_curves.saturation*), 436
reduce_neg_y() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.MonskyWashnitzerDifferential method*), 692
reduce_neg_y_fast() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.MonskyWashnitzerDifferential method*), 692
reduce_neg_y_faster() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.MonskyWashnitzerDifferential method*), 692
reduce_negative() (*in module sage.schemes.hyperelliptic_curves.monsky_washnitzer*), 715
reduce_pos_y() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.MonskyWashnitzerDifferential method*), 693
reduce_pos_y_fast() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.MonskyWashnitzerDifferential method*), 693
reduce_positive() (*in module sage.schemes.hyperelliptic_curves.monsky_washnitzer*), 715
reduce_tau() (*in module sage.schemes.elliptic_curves.period_lattice*), 529
reduce_zero() (*in module sage.schemes.hyperelliptic_curves.monsky_washnitzer*), 716
reduced_quadratic_form() (*sage.schemes.elliptic_curves.heegner.HeegnerPointOnX0N method*), 574
reducible_primes() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 401
reducible_primes() (*sage.schemes.elliptic_curves.gal_reps_number_field.GaloisRepresentation method*), 460
reducible_primes() (*sage.schemes.elliptic_curves.gal_reps.GaloisRepresentation method*), 452
reducible_primes_Billerey() (*in module sage.schemes.elliptic_curves.gal_reps_number_field*), 461
reducible_primes_naive() (*in module sage.schemes.elliptic_curves.gal_reps_number_field*), 462
reduction() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 402
reduction() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_number_field method*), 58
reduction() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 345
ReductionData (*class in sage.interfaces.genus2reduction*), 739
refine() (*sage.schemes.elliptic_curves.period_lattice_region.PeriodicRegion method*), 533
regulator() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 345
regulator_of_points() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 402
reorder() (*sage.schemes.elliptic_curves.isogeny_class.IsogenyClass_EC method*), 465
residue_disc() (*sage.schemes.hyperelliptic_curves.hyperelliptic_padic_field.HyperellipticCurve_padic_field method*), 681
right_ideals() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlg method*), 589
ring_class_field() (*sage.schemes.elliptic_curves.heegner.HeegnerPoint method*), 565
ring_class_field() (*sage.schemes.elliptic_curves.heegner.HeegnerPoints_level_disc_cond method*), 579
RingClassField (*class in sage.schemes.elliptic_curves.heegner*), 596

root_number() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 346
rst_transform() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 86

S

S() (*sage.schemes.elliptic_curves.height.EllipticCurveCanonicalHeight method*), 415
S_integral_points() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 271
S_to_Q() (*sage.schemes.hyperelliptic_curves.hyperelliptic_padic_field.HyperellipticCurve_padic_field method*), 668
sage.interfaces.genus2reduction module, 737
sage.schemes.elliptic_curves.cm module, 480
sage.schemes.elliptic_curves.constructor module, 1
sage.schemes.elliptic_curves.descent_two_isogeny module, 627
sage.schemes.elliptic_curves.ec_database module, 361
sage.schemes.elliptic_curves.ell_curve_isogeny module, 193
sage.schemes.elliptic_curves.ell_egros module, 630
sage.schemes.elliptic_curves.ell_field module, 91
sage.schemes.elliptic_curves.ell_finite_field module, 121
sage.schemes.elliptic_curves.ell_generic module, 59
sage.schemes.elliptic_curves.ell_local_data module, 493
sage.schemes.elliptic_curves.ell_modular_symbols module, 537
sage.schemes.elliptic_curves.ell_number_field module, 362
sage.schemes.elliptic_curves.ell_padic_field module, 634
sage.schemes.elliptic_curves.ell_point module, 21
sage.schemes.elliptic_curves.ell_rational_field module, 269
sage.schemes.elliptic_curves.ell_tate_curve module, 503
sage.schemes.elliptic_curves.ell_torsion module, 437
sage.schemes.elliptic_curves.ell_wp module, 507
sage.schemes.elliptic_curves.formal_group module, 153
sage.schemes.elliptic_curves.gal_reps module, 441
sage.schemes.elliptic_curves.gal_reps_number_field module, 452
sage.schemes.elliptic_curves.gp_simon module, 635
sage.schemes.elliptic_curves.heegner module, 557
sage.schemes.elliptic_curves.height module, 414
sage.schemes.elliptic_curves.hom module, 159
sage.schemes.elliptic_curves.hom_composite module, 173
sage.schemes.elliptic_curves.hom_frobenius module, 229
sage.schemes.elliptic_curves.hom_scalar module, 223
sage.schemes.elliptic_curves.hom_sum module, 181
sage.schemes.elliptic_curves.hom_velusqrt module, 215
sage.schemes.elliptic_curves.isogeny_class module, 463
sage.schemes.elliptic_curves.isogeny_small_degree module, 235
sage.schemes.elliptic_curves.jacobian module, 17
sage.schemes.elliptic_curves.kodaira_symbol

module, 501
sage.schemes.elliptic_curves.lseries_ell module, 548
sage.schemes.elliptic_curves.mod5family module, 635
sage.schemes.elliptic_curves.mod_poly module, 267
sage.schemes.elliptic_curves.mod_sym_num module, 542
sage.schemes.elliptic_curves.modular_parametrization module, 535
sage.schemes.elliptic_curves.padic_lseries module, 612
sage.schemes.elliptic_curves.period_lattice module, 511
sage.schemes.elliptic_curves.period_lattice_region module, 530
sage.schemes.elliptic_curves.Qcurves module, 489
sage.schemes.elliptic_curves.saturation module, 432
sage.schemes.elliptic_curves.sha_tate module, 470
sage.schemes.elliptic_curves.weierstrass_morphism module, 187
sage.schemes.elliptic_curves.weierstrass_transform module, 636
sage.schemes.hyperelliptic_curves.constructor module, 641
sage.schemes.hyperelliptic_curves.hypellfrob module, 717
sage.schemes.hyperelliptic_curves.hyperelliptic_finite_field module, 653
sage.schemes.hyperelliptic_curves.hyperelliptic_g2 module, 730
sage.schemes.hyperelliptic_curves.hyperelliptic_generic module, 643
sage.schemes.hyperelliptic_curves.hyperelliptic_padic_field module, 667
sage.schemes.hyperelliptic_curves.hyperelliptic_rational_field module, 684
sage.schemes.hyperelliptic_curves.invariants module, 732
sage.schemes.hyperelliptic_curves.jacobian_g2 module, 723
sage.schemes.hyperelliptic_curves.jacobian_generic module, 719
sage.schemes.hyperelliptic_curves.jacobian_homset module, 724
sage.schemes.hyperelliptic_curves.jacobian_morphism module, 725
sage.schemes.hyperelliptic_curves.kummer_surface module, 737
sage.schemes.hyperelliptic_curves.mestre module, 685
sage.schemes.hyperelliptic_curves.monsky_washnitzer module, 688
satisfies_heegner_hypothesis() (*in module sage.schemes.elliptic_curves.heegner*), 611
satisfies_heegner_hypothesis() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 346
satisfies_heegner_hypothesis() (*sage.schemes.elliptic_curves.heegner.HeegnerQuatAlg method*), 589
satisfies_kolyvagin_hypothesis() (*sage.schemes.elliptic_curves.heegner.HeegnerPointOnEllipticCurve method*), 570
satisfies_kolyvagin_hypothesis() (*sage.schemes.elliptic_curves.heegner.HeegnerPoints_level_disc_cond method*), 579
satisfies_kolyvagin_hypothesis() (*sage.schemes.elliptic_curves.heegner.KolyvaginPoint method*), 596
satisfies_weak_heegner_hypothesis() (*in module sage.schemes.elliptic_curves.heegner*), 611
saturation() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 404
saturation() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*),

347
scalar_multiplication() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 86
scalar_multiply() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialCubicQuotientRingElement method*), 700
scale_curve() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 86
scaling_factor() (*sage.schemes.elliptic_curves.ell_curve_isogeny.EllipticCurveIsogeny method*), 202
scaling_factor() (*sage.schemes.elliptic_curves.hom_composite.EllipticCurveHom_composite method*), 178
scaling_factor() (*sage.schemes.elliptic_curves.hom_frobenius.EllipticCurveHom_frobenius method*), 232
scaling_factor() (*sage.schemes.elliptic_curves.hom_scalar.EllipticCurveHom_scalar method*), 227
scaling_factor() (*sage.schemes.elliptic_curves.hom_sum.EllipticCurveHom_sum method*), 184
scaling_factor() (*sage.schemes.elliptic_curves.hom_velusqrt.EllipticCurveHom_velusqrt method*), 219
scaling_factor() (*sage.schemes.elliptic_curves.hom.EllipticCurveHom method*), 168
scaling_factor() (*sage.schemes.elliptic_curves.weierstrass_morphism.WeierstrassIsomorphism method*), 190
scheme() (*sage.schemes.hyperelliptic_curves.jacobian_morphism.JacobianMorphism_divisor_class_field method*), 727
selmer_rank() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 348
separable_degree() (*sage.schemes.elliptic_curves.hom.EllipticCurveHom method*), 168
series() (*sage.schemes.elliptic_curves.padic_lseries.pAdicLseriesOrdinary method*), 620
series() (*sage.schemes.elliptic_curves.padic_lseries.pAdicLseriesSupersingular method*), 625
set_order() (*sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field method*), 139
set_order() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_field method*), 33
sextic_twist() (*sage.schemes.elliptic_curves.ell_field.EllipticCurve_field method*), 116
Sha (*class in sage.schemes.elliptic_curves.sha_tate*), 471
sha() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 348
shift() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialCubicQuotientRingElement method*), 700
short_weierstrass_model() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 87
sigma() (*sage.schemes.elliptic_curves.formal_group.EllipticCurveFormalGroup method*), 156
sigma() (*sage.schemes.elliptic_curves.period_lattice.PeriodLattice_ell method*), 527
sign() (*sage.schemes.elliptic_curves.ell_modular_symbols.ModularSymbol method*), 538
silverman_height_bound() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 348
simon_two_descent() (*in module sage.schemes.elliptic_curves.gp_simon*), 635
simon_two_descent() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 406
simon_two_descent() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 349
simplest_rational_poly() (*in module sage.schemes.elliptic_curves.heegner*), 612
Sn() (*sage.schemes.elliptic_curves.height.EllipticCurveCanonicalHeight method*), 416
solve_linear_differential_system() (*in module sage.schemes.elliptic_curves.ell_wp*), 509
special_supersingular_curve() (*in module sage.schemes.elliptic_curves.ell_finite_field*), 150
SpecialCubicQuotientRing (*class in sage.schemes.hyperelliptic_curves.monsky_washnitzer*), 697
SpecialCubicQuotientRingElement (*class in sage.schemes.hyperelliptic_curves.monsky_washnitzer*), 700
SpecialHyperellipticQuotientElement (*class in sage.schemes.hyperelliptic_curves.monsky_washnitzer*), 701
SpecialHyperellipticQuotientRing (*class in sage.schemes.hyperelliptic_curves.monsky_washnitzer*), 704
SpecialHyperellipticQuotientRing_class (*in module sage.schemes.hyperelliptic_curves.monsky_washnitzer*), 707
square() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialCubicQuotientRingEle-*

ment method), 701
Step4Test() (*in module sage.schemes.elliptic_curves.Qcurves*), 489
summands() (*sage.schemes.elliptic_curves.hom_sum.EllipticCurveHom_sum method*), 185
supersingular_j_polynomial() (*in module sage.schemes.elliptic_curves.ell_finite_field*), 151
supersingular_primes() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 351
sympow() (*sage.schemes.elliptic_curves.lseries_ell.Lseries_ell method*), 552
sympow_derivs() (*sage.schemes.elliptic_curves.lseries_ell.Lseries_ell method*), 553

T

tamagawa_exponent() (*sage.schemes.elliptic_curves.ell_local_data.EllipticCurveLocalData method*), 500
tamagawa_exponent() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 408
tamagawa_exponent() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 352
tamagawa_number() (*sage.schemes.elliptic_curves.ell_local_data.EllipticCurveLocalData method*), 500
tamagawa_number() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 408
tamagawa_number() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 352
tamagawa_number_old() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 352
tamagawa_numbers() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 409
tamagawa_product() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 409
tamagawa_product() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 353
tamagawa_product_bsd() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 410
tangent_at_smooth_point() (*in module sage.schemes.elliptic_curves.constructor*), 16
tate_curve() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 353
tate_pairing() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_field method*), 35
TateCurve (*class in sage.schemes.elliptic_curves.ell_tate_curve*), 503
tau() (*sage.schemes.elliptic_curves.heegner.HeegnerPointOnEllipticCurve method*), 570
tau() (*sage.schemes.elliptic_curves.heegner.HeegnerPointOnX0N method*), 574
tau() (*sage.schemes.elliptic_curves.height.EllipticCurveCanonicalHeight method*), 424
tau() (*sage.schemes.elliptic_curves.period_lattice.PeriodLattice_ell method*), 528
taylor_series() (*sage.schemes.elliptic_curves.lseries_ell.Lseries_ell method*), 554
teichmuller() (*sage.schemes.elliptic_curves.padic_lseries.pAdicLseries method*), 617
teichmuller() (*sage.schemes.hyperelliptic_curves.hyperelliptic_padic_field.HyperellipticCurve_padic_field method*), 681
test_els() (*in module sage.schemes.elliptic_curves.descent_two_isogeny*), 627
test_mu() (*sage.schemes.elliptic_curves.height.EllipticCurveCanonicalHeight method*), 424
test_padic_square() (*in module sage.schemes.elliptic_curves.descent_two_isogeny*), 627
test_qpls() (*in module sage.schemes.elliptic_curves.descent_two_isogeny*), 627
test_valuation() (*in module sage.schemes.elliptic_curves.descent_two_isogeny*), 627
three_selmer_rank() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 354
tiny_integrals() (*sage.schemes.hyperelliptic_curves.hyperelliptic_padic_field.HyperellipticCurve_padic_field method*), 682
tiny_integrals_on_basis() (*sage.schemes.hyperelliptic_curves.hyperelliptic_padic_field.HyperellipticCurve_padic_field method*), 682
to_isogeny_chain() (*sage.schemes.elliptic_curves.hom_sum.EllipticCurveHom_sum method*), 185
torsion_basis() (*sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field method*), 141
torsion_bound() (*in module sage.schemes.elliptic_curves.ell_torsion*), 439
torsion_order() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 410
torsion_order() (*sage.schemes.ellip-*

tic_curves.ell_rational_field.EllipticCurve_rational_field method), 354
torsion_points() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 411
torsion_points() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 354
torsion_polynomial() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 88
torsion_subgroup() (*sage.schemes.elliptic_curves.ell_number_field.EllipticCurve_number_field method*), 413
torsion_subgroup() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 356
trace() (*sage.schemes.elliptic_curves.hom.EllipticCurveHom method*), 169
trace_of_frobenius() (*sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field method*), 142
trace_to_real_numerical() (*sage.schemes.elliptic_curves.heegner.KolyvaginPoint method*), 596
transportable_symbol() (*sage.schemes.elliptic_curves.mod_sym_num.ModularSymbolNumerical method*), 547
transpose_list() (*in module sage.schemes.hyperelliptic_curves.monsky_washnitzer*), 716
truncate_neg() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialHyperellipticQuotientElement method*), 704
tuple() (*sage.schemes.elliptic_curves.weierstrass_morphism.baseWI method*), 191
twist_values() (*sage.schemes.elliptic_curves.lseries_ell.Lseries_ell method*), 554
twist_zeros() (*sage.schemes.elliptic_curves.lseries_ell.Lseries_ell method*), 555
twists() (*sage.schemes.elliptic_curves.ell_finite_field.EllipticCurve_finite_field method*), 142
two_descent() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 356
two_descent_by_two_isogeny() (*in module sage.schemes.elliptic_curves.descent_two_isogeny*), 628
two_descent_by_two_isogeny_work() (*in module sage.schemes.elliptic_curves.descent_two_isogeny*), 629
two_descent_simon() (*sage.schemes.elliptic_curves.ell_rational_field.EllipticCurve_rational_field method*), 357
two_division_polynomial() (*sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic method*), 89
two_selmer_bound() (*sage.schemes.elliptic_curves.sha_tate.Sha method*), 480
two_torsion_part() (*in module sage.schemes.elliptic_curves.ell_curve_isogeny*), 211
two_torsion_rank() (*sage.schemes.elliptic_curves.ell_field.EllipticCurve_field method*), 117

U

ubs() (*in module sage.schemes.hyperelliptic_curves.invariants*), 736
Ueberschiebung() (*in module sage.schemes.hyperelliptic_curves.invariants*), 732
unfill_isogeny_matrix() (*in module sage.schemes.elliptic_curves.ell_curve_isogeny*), 212
union() (*sage.schemes.elliptic_curves.height.UnionOfIntervals class method*), 429
UnionOfIntervals (*class in sage.schemes.elliptic_curves.height*), 427

V

value_ring() (*sage.schemes.hyperelliptic_curves.jacobian_homset.JacobianHomset_divisor_classes method*), 724
values_along_line() (*sage.schemes.elliptic_curves.lseries_ell.Lseries_ell method*), 555
verify() (*sage.schemes.elliptic_curves.period_lattice_region.PeriodicRegion method*), 534

W

w() (*sage.schemes.elliptic_curves.formal_group.EllipticCurveFormalGroup method*), 157
w1 (*sage.schemes.elliptic_curves.period_lattice_region.PeriodicRegion attribute*), 534
w2 (*sage.schemes.elliptic_curves.period_lattice_region.PeriodicRegion attribute*), 534
weierstrass_p() (*in module sage.schemes.elliptic_curves.ell_wp*), 509
weierstrass_p() (*sage.schemes.elliptic_curves.ell_field.EllipticCurve_field method*), 117
weierstrass_points() (*sage.schemes.hyperelliptic_curves.hyperelliptic_padic_field.HyperellipticCurve_padic_field method*), 683
WeierstrassIsomorphism (*class in sage.schemes.elliptic_curves.weierstrass_morphism*), 187

WeierstrassTransformation (*class in sage.schemes.elliptic_curves.weierstrass_transform*), 636
WeierstrassTransformationWithInverse() (*in module sage.schemes.elliptic_curves.weierstrass_transform*), 638
WeierstrassTransformationWithInverse_class (*class in sage.schemes.elliptic_curves.weierstrass_transform*), 639
weil_pairing() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_field method*), 37
wp_c() (*sage.schemes.elliptic_curves.height.EllipticCurveCanonicalHeight method*), 426
wp_intervals() (*sage.schemes.elliptic_curves.height.EllipticCurveCanonicalHeight method*), 426
wp_on_grid() (*sage.schemes.elliptic_curves.height.EllipticCurveCanonicalHeight method*), 427

X

x() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_field method*), 39
x() (*sage.schemes.elliptic_curves.formal_group.EllipticCurveFormalGroup method*), 157
x() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialHyperellipticQuotientRing method*), 707
x_poly_exact() (*sage.schemes.elliptic_curves.heegner.HeegnerPointOnEllipticCurve method*), 571
x_rational_map() (*sage.schemes.elliptic_curves.ell_curve_isogeny.EllipticCurveIsogeny method*), 202
x_rational_map() (*sage.schemes.elliptic_curves.hom_composite.EllipticCurveHom_composite method*), 178
x_rational_map() (*sage.schemes.elliptic_curves.hom_frobenius.EllipticCurveHom_frobenius method*), 233
x_rational_map() (*sage.schemes.elliptic_curves.hom_scalar.EllipticCurveHom_scalar method*), 227
x_rational_map() (*sage.schemes.elliptic_curves.hom_sum.EllipticCurveHom_sum method*), 186
x_rational_map() (*sage.schemes.elliptic_curves.hom_velusqrt.EllipticCurveHom_velusqrt method*), 219
x_rational_map() (*sage.schemes.elliptic_curves.hom.EllipticCurveHom method*), 169
x_rational_map() (*sage.schemes.elliptic_curves.weierstrass_morphism.WeierstrassIsomorphism method*), 190
x_to_p() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.MonskyWashnitzerDifferentialRing method*), 697
xy() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_field method*), 39

Y

y() (*sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_field method*), 39
y() (*sage.schemes.elliptic_curves.formal_group.EllipticCurveFormalGroup method*), 158
y() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialHyperellipticQuotientRing method*), 707

Z

zero() (*sage.schemes.hyperelliptic_curves.monsky_washnitzer.SpecialHyperellipticQuotientRing method*), 707
zero_sums() (*sage.schemes.elliptic_curves.lseries_ell.Lseries_ell method*), 556
zeros() (*sage.schemes.elliptic_curves.lseries_ell.Lseries_ell method*), 556
zeros_in_interval() (*sage.schemes.elliptic_curves.lseries_ell.Lseries_ell method*), 556
zeta_function() (*sage.schemes.hyperelliptic_curves.hyperelliptic_finite_field.HyperellipticCurve_finite_field method*), 666