Sage Reference Manual: Elliptic and Plane Curves

Release 6.3

The Sage Development Team

August 11, 2014

CONTENTS

CHAPTER

ONE

PLANE CURVE CONSTRUCTORS

AUTHORS:

- William Stein (2005-11-13)
- David Kohel (2006-01)

sage.schemes.plane_curves.constructor.**Curve**(F)

Return the plane or space curve defined by F, where F can be either a multivariate polynomial, a list or tuple of polynomials, or an algebraic scheme.

If F is in two variables the curve is affine, and if it is homogenous in 3 variables, then the curve is projective.

EXAMPLE: A projective plane curve

```
sage: x,y,z = QQ['x,y,z'].gens()
sage: C = Curve(x^3 + y^3 + z^3); C
Projective Curve over Rational Field defined by x^3 + y^3 + z^3
sage: C.genus()
1
```

EXAMPLE: Affine plane curves

```
sage: x,y = GF(7)['x,y'].gens()
sage: C = Curve(y^2 + x^3 + x^10); C
Affine Curve over Finite Field of size 7 defined by x^10 + x^3 + y^2
sage: C.genus()
0
sage: x, y = QQ['x,y'].gens()
sage: Curve(x^3 + y^3 + 1)
Affine Curve over Rational Field defined by x^3 + y^3 + 1
```

EXAMPLE: A projective space curve

```
sage: x,y,z,w = QQ['x,y,z,w'].gens()
sage: C = Curve([x^3 + y^3 - z^3 - w^3, x^5 - y*z^4]); C
Projective Space Curve over Rational Field defined by x^3 + y^3 - z^3 - w^3, x^5 - y*z^4
sage: C.genus()
13
```

EXAMPLE: An affine space curve

```
sage: x,y,z = QQ['x,y,z'].gens()
sage: C = Curve([y^2 + x^3 + x^10 + z^7,  x^2 + y^2]); C
Affine Space Curve over Rational Field defined by x^10 + z^7 + x^3 + y^2, x^2 + y^2
sage: C.genus()
47
```

EXAMPLE: We can also make non-reduced non-irreducible curves.

```
sage: x,y,z = QQ['x,y,z'].gens()
sage: Curve((x-y)*(x+y))
Projective Conic Curve over Rational Field defined by x^2 - y^2
sage: Curve((x-y)^2*(x+y)^2)
Projective Curve over Rational Field defined by x^4 - 2*x^2*y^2 + y^4
```

EXAMPLE: A union of curves is a curve.

```
sage: x,y,z = QQ['x,y,z'].gens()
sage: C = Curve(x^3 + y^3 + z^3)
sage: D = Curve(x^4 + y^4 + z^4)
sage: C.union(D)
Projective Curve over Rational Field defined by
x^7 + x^4*y^3 + x^3*y^4 + y^7 + x^4*z^3 + y^4*z^3 + x^3*z^4 + y^3*z^4 + z^7
```

The intersection is not a curve, though it is a scheme.

```
sage: X = C.intersection(D); X
Closed subscheme of Projective Space of dimension 2 over Rational Field defined by:
 x^3 + y^3 + z^3,
 x^4 + y^4 + z^4
```

Note that the intersection has dimension 0.

```
sage: X.dimension()
0
sage: I = X.defining_ideal(); I
Ideal (x^3 + y^3 + z^3, x^4 + y^4 + z^4) of Multivariate Polynomial Ring in x, y, z over Rationa
```

EXAMPLE: In three variables, the defining equation must be homogeneous.

If the parent polynomial ring is in three variables, then the defining ideal must be homogeneous.

```
sage: x,y,z = QQ['x,y,z'].gens()
sage: Curve(x^2+y^2)
Projective Conic Curve over Rational Field defined by x^2 + y^2
sage: Curve(x^2+y^2+z)
Traceback (most recent call last):
...
TypeError: x^2 + y^2 + z is not a homogeneous polynomial!
```

The defining polynomial must always be nonzero:

```
sage: P1.<x,y> = ProjectiveSpace(1,GF(5))
sage: Curve(0*x)
Traceback (most recent call last):
...
ValueError: defining polynomial of curve must be nonzero
```

CHAPTER

TWO

AFFINE PLANE CURVES OVER A GENERAL RING

AUTHORS:

- William Stein (2005-11-13)
- David Joyner (2005-11-13)
- David Kohel (2006-01)

class sage.schemes.plane_curves.affine_curve.**AffineCurve_finite_field**(*A*, *f*)
Bases: sage.schemes.plane_curves.affine_curve.AffineCurve_generic

rational_points(*algorithm='enum'*)
Return sorted list of all rational points on this curve.

Use *very* naive point enumeration to find all rational points on this curve over a finite field.

EXAMPLE:

```
sage: A, (x,y) = AffineSpace(2,GF(9,'a')).objgens()
sage: C = Curve(x^2 + y^2 - 1)
sage: C
Affine Curve over Finite Field in a of size 3^2 defined by x0^2 + x1^2 - 1
sage: C.rational_points()
[(0, 1), (0, 2), (1, 0), (2, 0), (a + 1, a + 1), (a + 1, 2*a + 2), (2*a + 2, a + 1), (2*a +
```

class sage.schemes.plane_curves.affine_curve.**AffineCurve_generic**(*A*, *f*)
Bases: sage.schemes.plane_curves.curve.Curve_generic

divisor_of_function(*r*)
Return the divisor of a function on a curve.

INPUT: r is a rational function on X

OUTPUT:

- `list` - The divisor of r represented as a list of coefficients and points. (TODO: This will change to a more structural output in the future.)

EXAMPLES:

```
sage: F = GF(5)
sage: P2 = AffineSpace(2, F, names = 'xy')
sage: R = P2.coordinate_ring()
sage: x, y = R.gens()
sage: f = y^2 - x^9 - x
sage: C = Curve(f)
```

```
sage: K = FractionField(R)
sage: r = 1/x
sage: C.divisor_of_function(r)      # todo: not implemented (broken)
      [[-1, (0, 0, 1)]]
sage: r = 1/x^3
sage: C.divisor_of_function(r)      # todo: not implemented (broken)
      [[-3, (0, 0, 1)]]
```

local_coordinates(*pt*, *n*)

Return local coordinates to precision n at the given point.

Behaviour is flaky - some choices of n are worst that others.

INPUT:

- `pt` - an F-rational point on X which is not a point of ramification for the projection (x,y) - x.
- `n` - the number of terms desired

OUTPUT: x = x0 + t y = y0 + power series in t

EXAMPLES:

```
sage: F = GF(5)
sage: pt = (2,3)
sage: R = PolynomialRing(F,2, names = ['x','y'])
sage: x,y = R.gens()
sage: f = y^2-x^9-x
sage: C = Curve(f)
sage: C.local_coordinates(pt, 9)
[t + 2, -2*t^12 - 2*t^11 + 2*t^9 + t^8 - 2*t^7 - 2*t^6 - 2*t^4 + t^3 - 2*t^2 - 2]
```

plot(**args*, ***kwds*)

Plot the real points on this affine plane curve.

INPUT:

- `self` - an affine plane curve
- `*args` - optional tuples (variable, minimum, maximum) for plotting dimensions
- `**kwds` - optional keyword arguments passed on to `implicit_plot`

EXAMPLES:

A cuspidal curve:

```
sage: R.<x, y> = QQ[]
sage: C = Curve(x^3 - y^2)
sage: C.plot()
```

A 5-nodal curve of degree 11. This example also illustrates some of the optional arguments:

```
sage: R.<x, y> = ZZ[]
sage: C = Curve(32*x^2 - 2097152*y^11 + 1441792*y^9 - 360448*y^7 + 39424*y^5 - 1760*y^3 + 22
sage: C.plot((x, -1, 1), (y, -1, 1), plot_points=400)
```

A line over $\mathbf{RR}$:

```
sage: R.<x, y> = RR[]
sage: C = Curve(R(y - sqrt(2)*x))
sage: C.plot()
```

class sage.schemes.plane_curves.affine_curve.**AffineCurve_prime_finite_field**(*A*, *f*)

Bases: sage.schemes.plane_curves.affine_curve.AffineCurve_finite_field

rational_points(*algorithm='enum'*)

Return sorted list of all rational points on this curve.

INPUT:

- `algorithm` - string:
  - `'enum'` - straightforward enumeration
  - `'bn'` - via Singular's Brill-Noether package.
  - `'all'` - use all implemented algorithms and verify that they give the same answer, then return it

---

Note: The Brill-Noether package does not always work. When it fails a RuntimeError exception is raised.

EXAMPLE:

```
sage: x, y = (GF(5)['x,y']).gens()
sage: f = y^2 - x^9 - x
sage: C = Curve(f); C
Affine Curve over Finite Field of size 5 defined by -x^9 + y^2 - x
sage: C.rational_points(algorithm='bn')
[(0, 0), (2, 2), (2, 3), (3, 1), (3, 4)]
sage: C = Curve(x - y + 1)
sage: C.rational_points()
[(0, 1), (1, 2), (2, 3), (3, 4), (4, 0)]
```

We compare Brill-Noether and enumeration:

```
sage: x, y = (GF(17)['x,y']).gens()
sage: C = Curve(x^2 + y^5 + x*y - 19)
sage: v = C.rational_points(algorithm='bn')
sage: w = C.rational_points(algorithm='enum')
sage: len(v)
20
sage: v == w
True
```

riemann_roch_basis(*D*)

Interfaces with Singular's BrillNoether command.

INPUT:

- `self` - a plane curve defined by a polynomial eqn f(x,y) = 0 over a prime finite field F = GF(p) in 2 variables x,y representing a curve X: f(x,y) = 0 having n F-rational points (see the Sage function places_on_curve)
- `D` - an n-tuple of integers $(d1, ..., dn)$ representing the divisor $Div = d1 * P1 + ... + dn * Pn$, where $X(F) = \{P1, ..., Pn\}$. *The ordering is that dictated by places_on_curve.*

OUTPUT: basis of L(Div)

EXAMPLE:

```
sage: R = PolynomialRing(GF(5),2,names = ["x","y"])
sage: x, y = R.gens()
sage: f = y^2 - x^9 - x
sage: C = Curve(f)
sage: D = [6,0,0,0,0,0]
```

```
sage: C.riemann_roch_basis(D)
[1, (y^2*z^4 - x*z^5)/x^6, (y^2*z^5 - x*z^6)/x^7, (y^2*z^6 - x*z^7)/x^8]
```

class sage.schemes.plane_curves.affine_curve.**AffineSpaceCurve_generic**(A, X)

Bases: sage.schemes.plane_curves.curve.Curve_generic, sage.schemes.generic.algebraic_scheme.AlgebraicScheme_subscheme_affine

CHAPTER

THREE

PROJECTIVE PLANE CURVES OVER A GENERAL RING

AUTHORS:

- William Stein (2005-11-13)
- David Joyner (2005-11-13)
- David Kohel (2006-01)
- Moritz Minzlaff (2010-11)

sage.schemes.plane_curves.projective_curve.**Hasse_bounds**(*q*, *genus=1*)

Return the Hasse-Weil bounds for the cardinality of a nonsingular curve defined over $\mathbf{F}_q$ of given `genus`.

INPUT:

- `q` (int) – a prime power
- `genus` (int, default 1) – a non-negative integer,

OUTPUT:

(tuple) The Hasse bounds (lb,ub) for the cardinality of a curve of genus `genus` defined over $\mathbf{F}_q$.

EXAMPLES:

```
sage: Hasse_bounds(2)
(1, 5)
sage: Hasse_bounds(next_prime(10^30))
(999999999999998000000000000058, 1000000000000002000000000000058)
```

class sage.schemes.plane_curves.projective_curve.**ProjectiveCurve_finite_field**(*A*, *f*)

Bases: sage.schemes.plane_curves.projective_curve.ProjectiveCurve_generic

rational_points(*algorithm='enum'*, *sort=True*)

Return the rational points on this curve computed via enumeration.

INPUT:

- `algorithm` (string, default: 'enum') – the algorithm to use. Currently this is ignored.
- `sort` (boolean, default `True`) – whether the output points should be sorted. If False, the order of the output is non-deterministic.

OUTPUT:

A list of all the rational points on the curve defined over its base field, possibly sorted.

Note: This is a slow Python-level implementation.

EXAMPLES:

```
sage: F = GF(7)
sage: P2.<X,Y,Z> = ProjectiveSpace(F,2)
sage: C = Curve(X^3+Y^3-Z^3)
sage: C.rational_points()
[(0 : 1 : 1), (0 : 2 : 1), (0 : 4 : 1), (1 : 0 : 1), (2 : 0 : 1), (3 : 1 : 0), (4 : 0 : 1),

sage: F = GF(1237)
sage: P2.<X,Y,Z> = ProjectiveSpace(F,2)
sage: C = Curve(X^7+7*Y^6*Z+Z^4*X^2*Y*89)
sage: len(C.rational_points())
1237

sage: F = GF(2^6,'a')
sage: P2.<X,Y,Z> = ProjectiveSpace(F,2)
sage: C = Curve(X^5+11*X*Y*Z^3 + X^2*Y^3 - 13*Y^2*Z^3)
sage: len(C.rational_points())
104

sage: R.<x,y,z> = GF(2)[]
sage: f = x^3*y + y^3*z + x*z^3
sage: C = Curve(f); pts = C.rational_points()
sage: pts
[(0 : 0 : 1), (0 : 1 : 0), (1 : 0 : 0)]
```

rational_points_iterator()

Return a generator object for the rational points on this curve.

INPUT:

- `self` – a projective curve

OUTPUT:

A generator of all the rational points on the curve defined over its base field.

EXAMPLE:

```
sage: F = GF(37)
sage: P2.<X,Y,Z> = ProjectiveSpace(F,2)
sage: C = Curve(X^7+Y*X*Z^5*55+Y^7*12)
sage: len(list(C.rational_points_iterator()))
37

sage: F = GF(2)
sage: P2.<X,Y,Z> = ProjectiveSpace(F,2)
sage: C = Curve(X*Y*Z)
sage: a = C.rational_points_iterator()
sage: a.next()
(1 : 0 : 0)
sage: a.next()
(0 : 1 : 0)
sage: a.next()
(1 : 1 : 0)
sage: a.next()
(0 : 0 : 1)
sage: a.next()
```

```
(1 : 0 : 1)
sage: a.next()
(0 : 1 : 1)
sage: a.next()
Traceback (most recent call last):
...
StopIteration

sage: F = GF(3^2,'a')
sage: P2.<X,Y,Z> = ProjectiveSpace(F,2)
sage: C = Curve(X^3+5*Y^2*Z-33*X*Y*X)
sage: b = C.rational_points_iterator()
sage: b.next()
(0 : 1 : 0)
sage: b.next()
(0 : 0 : 1)
sage: b.next()
(2*a + 2 : a : 1)
sage: b.next()
(2 : a + 1 : 1)
sage: b.next()
(a + 1 : 2*a + 1 : 1)
sage: b.next()
(1 : 2 : 1)
sage: b.next()
(2*a + 2 : 2*a : 1)
sage: b.next()
(2 : 2*a + 2 : 1)
sage: b.next()
(a + 1 : a + 2 : 1)
sage: b.next()
(1 : 1 : 1)
sage: b.next()
Traceback (most recent call last):
...
StopIteration
```

class sage.schemes.plane_curves.projective_curve.**ProjectiveCurve_generic**(A, f)

Bases: sage.schemes.plane_curves.curve.Curve_generic_projective

arithmetic_genus()

Return the arithmetic genus of this curve.

This is the arithmetic genus $g_a(C)$ as defined in Hartshorne. If the curve has degree d then this is simply $(d-1)(d-2)/2$. It need *not* equal the geometric genus (the genus of the normalization of the curve).

EXAMPLE:

```
sage: x,y,z = PolynomialRing(GF(5), 3, 'xyz').gens()
sage: C = Curve(y^2*z^7 - x^9 - x*z^8); C
Projective Curve over Finite Field of size 5 defined by -x^9 + y^2*z^7 - x*z^8
sage: C.arithmetic_genus()
28
sage: C.genus()
4
```

divisor_of_function(r)

Return the divisor of a function on a curve.

INPUT: r is a rational function on X

OUTPUT:

- `list` - The divisor of r represented as a list of coefficients and points. (TODO: This will change to a more structural output in the future.)

EXAMPLES:

```
sage: FF = FiniteField(5)
sage: P2 = ProjectiveSpace(2, FF, names = ['x','y','z'])
sage: R = P2.coordinate_ring()
sage: x, y, z = R.gens()
sage: f = y^2*z^7 - x^9 - x*z^8
sage: C = Curve(f)
sage: K = FractionField(R)
sage: r = 1/x
sage: C.divisor_of_function(r)      # todo: not implemented  !!!!
[[-1, (0, 0, 1)]]
sage: r = 1/x^3
sage: C.divisor_of_function(r)      # todo: not implemented  !!!!
[[-3, (0, 0, 1)]]
```

is_singular(*C*)

Returns whether the curve is singular or not.

EXAMPLES:

Over $\mathbf{Q}$:

```
sage: F = QQ
sage: P2.<X,Y,Z> = ProjectiveSpace(F,2)
sage: C = Curve(X^3-Y^2*Z)
sage: C.is_singular()
True
```

Over a finite field:

```
sage: F = GF(19)
sage: P2.<X,Y,Z> = ProjectiveSpace(F,2)
sage: C = Curve(X^3+Y^3+Z^3)
sage: C.is_singular()
False
sage: D = Curve(X^4-X*Z^3)
sage: D.is_singular()
True
sage: E = Curve(X^5+19*Y^5+Z^5)
sage: E.is_singular()
True
sage: E = Curve(X^5+9*Y^5+Z^5)
sage: E.is_singular()
False
```

Over $\mathbf{C}$:

```
sage: F = CC
sage: P2.<X,Y,Z> = ProjectiveSpace(F,2)
sage: C = Curve(X)
sage: C.is_singular()
False
sage: D = Curve(Y^2*Z-X^3)
sage: D.is_singular()
```

```
True
sage: E = Curve(Y^2*Z-X^3+Z^3)
sage: E.is_singular()
False
```

Showing that ticket #12187 is fixed:

```
sage: F.<X,Y,Z> = GF(2)[]
sage: G = Curve(X^2+Y*Z)
sage: G.is_singular()
False
```

local_coordinates(*pt*, *n*)

Return local coordinates to precision n at the given point.

Behaviour is flaky - some choices of n are worst that others.

INPUT:

- `pt` - an F-rational point on X which is not a point of ramification for the projection (x,y) - x.
- `n` - the number of terms desired

OUTPUT: x = x0 + t y = y0 + power series in t

EXAMPLES:

```
sage: FF = FiniteField(5)
sage: P2 = ProjectiveSpace(2, FF, names = ['x','y','z'])
sage: x, y, z = P2.coordinate_ring().gens()
sage: C = Curve(y^2*z^7-x^9-x*z^8)
sage: pt = C([2,3,1])
sage: C.local_coordinates(pt,9)      # todo: not implemented  !!!!
      [2 + t, 3 + 3*t^2 + t^3 + 3*t^4 + 3*t^6 + 3*t^7 + t^8 + 2*t^9 + 3*t^11 + 3*t^12]
```

plot(**args*, ***kwds*)

Plot the real points of an affine patch of this projective plane curve.

INPUT:

- `self` - an affine plane curve
- `patch` - (optional) the affine patch to be plotted; if not specified, the patch corresponding to the last projective coordinate being nonzero
- `*args` - optional tuples (variable, minimum, maximum) for plotting dimensions
- `**kwds` - optional keyword arguments passed on to `implicit_plot`

EXAMPLES:

A cuspidal curve:

```
sage: R.<x, y, z> = QQ[]
sage: C = Curve(x^3 - y^2*z)
sage: C.plot()
```

The other affine patches of the same curve:

```
sage: C.plot(patch=0)
sage: C.plot(patch=1)
```

An elliptic curve:

```
sage: E = EllipticCurve('101a')
sage: C = Curve(E)
sage: C.plot()
sage: C.plot(patch=0)
sage: C.plot(patch=1)
```

A hyperelliptic curve:

```
sage: P.<x> = QQ[]
sage: f = 4*x^5 - 30*x^3 + 45*x - 22
sage: C = HyperellipticCurve(f)
sage: C.plot()
sage: C.plot(patch=0)
sage: C.plot(patch=1)
```

class sage.schemes.plane_curves.projective_curve.**ProjectiveCurve_prime_finite_field**(*A*, *f*)

Bases: sage.schemes.plane_curves.projective_curve.ProjectiveCurve_finite_field

rational_points(*algorithm='enum'*, *sort=True*)

INPUT:

- algorithm - string:
- 'enum' - straightforward enumeration
- 'bn' - via Singular's brnoeth package.

EXAMPLE:

```
sage: x, y, z = PolynomialRing(GF(5), 3, 'xyz').gens()
sage: f = y^2*z^7 - x^9 - x*z^8
sage: C = Curve(f); C
Projective Curve over Finite Field of size 5 defined by
-x^9 + y^2*z^7 - x*z^8
sage: C.rational_points()
[(0 : 0 : 1), (0 : 1 : 0), (2 : 2 : 1), (2 : 3 : 1),
 (3 : 1 : 1), (3 : 4 : 1)]
sage: C = Curve(x - y + z)
sage: C.rational_points()
[(0 : 1 : 1), (1 : 1 : 0), (1 : 2 : 1), (2 : 3 : 1),
 (3 : 4 : 1), (4 : 0 : 1)]
sage: C = Curve(x*z+z^2)
sage: C.rational_points('all')
[(0 : 1 : 0), (1 : 0 : 0), (1 : 1 : 0), (2 : 1 : 0),
 (3 : 1 : 0), (4 : 0 : 1), (4 : 1 : 0), (4 : 1 : 1),
 (4 : 2 : 1), (4 : 3 : 1), (4 : 4 : 1)]
```

Note: The Brill-Noether package does not always work (i.e., the 'bn' algorithm. When it fails a RuntimeError exception is raised.

---

riemann_roch_basis(*D*)

Return a basis for the Riemann-Roch space corresponding to D.

This uses Singular's Brill-Noether implementation.

INPUT:

- D - a divisor

OUTPUT:

A list of function field elements that form a basis of the Riemann-Roch space

EXAMPLE:

```
sage: R.<x,y,z> = GF(2)[]
sage: f = x^3*y + y^3*z + x*z^3
sage: C = Curve(f); pts = C.rational_points()
sage: D = C.divisor([ (4, pts[0]), (4, pts[2]) ])
sage: C.riemann_roch_basis(D)
[x/y, 1, z/y, z^2/y^2, z/x, z^2/(x*y)]

sage: R.<x,y,z> = GF(5)[]
sage: f = x^7 + y^7 + z^7
sage: C = Curve(f); pts = C.rational_points()
sage: D = C.divisor([ (3, pts[0]), (-1,pts[1]), (10, pts[5]) ])
sage: C.riemann_roch_basis(D)
[(-2*x + y)/(x + y), (-x + z)/(x + y)]
```

Note: Currently this only works over prime field and divisors supported on rational points.

---

class sage.schemes.plane_curves.projective_curve.**ProjectiveSpaceCurve_generic**(*A*, *X*)

Bases: sage.schemes.plane_curves.curve.Curve_generic_projective

CHAPTER

FOUR

PLANE CONIC CONSTRUCTOR

AUTHORS:

- Marco Streng (2010-07-20)
- Nick Alexander (2008-01-08)

sage.schemes.plane_conics.constructor.**Conic**(*base_field*, *F=None*, *names=None*, *unique=True*)

Return the plane projective conic curve defined by `F` over `base_field`.

The input form `Conic(F, names=None)` is also accepted, in which case the fraction field of the base ring of `F` is used as base field.

INPUT:

- `base_field` – The base field of the conic.
- `names` – a list, tuple, or comma separated string of three variable names specifying the names of the coordinate functions of the ambient space $\mathbf{P}^3$. If not specified or read off from `F`, then this defaults to `'x,y,z'`.
- `F` – a polynomial, list, matrix, ternary quadratic form, or list or tuple of 5 points in the plane.

  If `F` is a polynomial or quadratic form, then the output is the curve in the projective plane defined by `F = 0`.

  If `F` is a polynomial, then it must be a polynomial of degree at most 2 in 2 variables, or a homogeneous polynomial in of degree 2 in 3 variables.

  If `F` is a matrix, then the output is the zero locus of $(x, y, z)F(x, y, z)^t$.

  If `F` is a list of coefficients, then it has length 3 or 6 and gives the coefficients of the monomials x^2, y^2, z^2 or all 6 monomials $x^2, xy, xz, y^2, yz, z^2$ in lexicographic order.

  If `F` is a list of 5 points in the plane, then the output is a conic through those points.
- `unique` – Used only if `F` is a list of points in the plane. If the conic through the points is not unique, then raise `ValueError` if and only if `unique` is True

OUTPUT:

A plane projective conic curve defined by `F` over a field.

EXAMPLES:

Conic curves given by polynomials

```
sage: X,Y,Z = QQ['X,Y,Z'].gens()
sage: Conic(X^2 - X*Y + Y^2 - Z^2)
Projective Conic Curve over Rational Field defined by X^2 - X*Y + Y^2 - Z^2
```

```
sage: x,y = GF(7)['x,y'].gens()
sage: Conic(x^2 - x + 2*y^2 - 3, 'U,V,W')
Projective Conic Curve over Finite Field of size 7 defined by U^2 + 2*V^2 - U*W - 3*W^2
```

Conic curves given by matrices

```
sage: Conic(matrix(QQ, [[1, 2, 0], [4, 0, 0], [7, 0, 9]]), 'x,y,z')
Projective Conic Curve over Rational Field defined by x^2 + 6*x*y + 7*x*z + 9*z^2

sage: x,y,z = GF(11)['x,y,z'].gens()
sage: C = Conic(x^2+y^2-2*z^2); C
Projective Conic Curve over Finite Field of size 11 defined by x^2 + y^2 - 2*z^2
sage: Conic(C.symmetric_matrix(), 'x,y,z')
Projective Conic Curve over Finite Field of size 11 defined by x^2 + y^2 - 2*z^2
```

Conics given by coefficients

```
sage: Conic(QQ, [1,2,3])
Projective Conic Curve over Rational Field defined by x^2 + 2*y^2 + 3*z^2
sage: Conic(GF(7), [1,2,3,4,5,6], 'X')
Projective Conic Curve over Finite Field of size 7 defined by X0^2 + 2*X0*X1 - 3*X1^2 + 3*X0*X2
```

The conic through a set of points

```
sage: C = Conic(QQ, [[10,2],[3,4],[-7,6],[7,8],[9,10]]); C
Projective Conic Curve over Rational Field defined by x^2 + 13/4*x*y - 17/4*y^2 - 35/2*x*z + 91/
sage: C.rational_point()
(10 : 2 : 1)
sage: C.point([3,4])
(3 : 4 : 1)

sage: a=AffineSpace(GF(13),2)
sage: Conic([a([x,x^2]) for x in range(5)])
Projective Conic Curve over Finite Field of size 13 defined by x^2 - y*z
```

CHAPTER

FIVE

PROJECTIVE PLANE CONICS OVER A FIELD

AUTHORS:

- Marco Streng (2010-07-20)
- Nick Alexander (2008-01-08)

class sage.schemes.plane_conics.con_field.**ProjectiveConic_field**(A, f)

Bases: sage.schemes.plane_curves.projective_curve.ProjectiveCurve_generic

Create a projective plane conic curve over a field. See Conic for full documentation.

EXAMPLES:

```
sage: K = FractionField(PolynomialRing(QQ, 't'))
sage: P.<X, Y, Z> = K[]
sage: Conic(X^2 + Y^2 - Z^2)
Projective Conic Curve over Fraction Field of Univariate Polynomial Ring in t over Rational Fiel
```

TESTS:

```
sage: K = FractionField(PolynomialRing(QQ, 't'))
sage: Conic([K(1), 1, -1])._test_pickling()
```

base_extend(S)

Returns the conic over S given by the same equation as self.

EXAMPLES:

```
sage: c = Conic([1, 1, 1]); c
Projective Conic Curve over Rational Field defined by x^2 + y^2 + z^2
sage: c.has_rational_point()
False
sage: d = c.base_extend(QuadraticField(-1, 'i')); d
Projective Conic Curve over Number Field in i with defining polynomial x^2 + 1 defined by x^
sage: d.rational_point(algorithm = 'rnfisnorm')
(i : 1 : 0)
```

cache_point(p)

Replace the point in the cache of self by p for use by self.rational_point() and self.parametrization().

EXAMPLES:

```
sage: c = Conic([1, -1, 1])
sage: c.point([15, 17, 8])
(15/8 : 17/8 : 1)
sage: c.rational_point()
(15/8 : 17/8 : 1)
sage: c.cache_point(c.rational_point(read_cache = False))
sage: c.rational_point()
(-1 : 1 : 0)
```

coefficients()

Gives a the 6 coefficients of the conic `self` in lexicographic order.

EXAMPLES:

```
sage: Conic(QQ, [1,2,3,4,5,6]).coefficients()
[1, 2, 3, 4, 5, 6]

sage: P.<x,y,z> = GF(13)[]
sage: a = Conic(x^2+5*x*y+y^2+z^2).coefficients(); a
[1, 5, 0, 1, 0, 1]
sage: Conic(a)
Projective Conic Curve over Finite Field of size 13 defined by x^2 + 5*x*y + y^2 + z^2
```

derivative_matrix()

Gives the derivative of the defining polynomial of the conic `self`, which is a linear map, as a 3×3 matrix.

EXAMPLES:

In characteristic different from 2, the derivative matrix is twice the symmetric matrix:

```
sage: c = Conic(QQ, [1,1,1,1,1,0])
sage: c.symmetric_matrix()
[  1 1/2 1/2]
[1/2   1 1/2]
[1/2 1/2   0]
sage: c.derivative_matrix()
[2 1 1]
[1 2 1]
[1 1 0]
```

An example in characteristic 2:

```
sage: P.<t> = GF(2)[]
sage: c = Conic([t, 1, t^2, 1, 1, 0]); c
Projective Conic Curve over Fraction Field of Univariate Polynomial Ring in t over Finite Fi
sage: c.is_smooth()
True
sage: c.derivative_matrix()
[  0   1 t^2]
[  1   0   1]
[t^2   1   0]
```

determinant()

Returns the determinant of the symmetric matrix that defines the conic `self`.

This is defined only if the base field has characteristic different from 2.

EXAMPLES:

```
sage: C = Conic([1,2,3,4,5,6])
sage: C.determinant()
```

```
41/4
sage: C.symmetric_matrix().determinant()
41/4
```

Determinants are only defined in characteristic different from 2:

```
sage: C = Conic(GF(2), [1, 1, 1, 1, 1, 0])
sage: C.is_smooth()
True
sage: C.determinant()
Traceback (most recent call last):
...
ValueError: The conic self (= Projective Conic Curve over Finite Field of size 2 defined by
```

diagonal_matrix()

Returns a diagonal matrix D and a matrix T such that $T^t A T = D$ holds, where $(x, y, z)A(x, y, z)^t$ is the defining polynomial of the conic `self`.

EXAMPLES:

```
sage: c = Conic(QQ, [1,2,3,4,5,6])
sage: d, t = c.diagonal_matrix(); d, t
(
[    1     0     0]  [   1   -1 -7/6]
[    0     3     0]  [   0    1 -1/3]
[    0     0 41/12], [   0    0    1]
)
sage: t.transpose()*c.symmetric_matrix()*t
[    1     0     0]
[    0     3     0]
[    0     0 41/12]
```

Diagonal matrices are only defined in characteristic different from 2:

```
sage: c = Conic(GF(4, 'a'), [0, 1, 1, 1, 1, 1])
sage: c.is_smooth()
True
sage: c.diagonal_matrix()
Traceback (most recent call last):
...
ValueError: The conic self (= Projective Conic Curve over Finite Field in a of size 2^2 defi
```

diagonalization(*names=None*)

Returns a diagonal conic C, an isomorphism of schemes $M : C$ -> `self` and the inverse N of M.

EXAMPLES:

```
sage: Conic(GF(5), [1,0,1,1,0,1]).diagonalization()
(Projective Conic Curve over Finite Field of size 5 defined by x^2 + y^2 + 2*z^2,
 Scheme morphism:
  From: Projective Conic Curve over Finite Field of size 5 defined by x^2 + y^2 + 2*z^2
  To:   Projective Conic Curve over Finite Field of size 5 defined by x^2 + y^2 + x*z + z^2
  Defn: Defined on coordinates by sending (x : y : z) to
        (x + 2*z : y : z),
 Scheme morphism:
  From: Projective Conic Curve over Finite Field of size 5 defined by x^2 + y^2 + x*z + z^2
  To:   Projective Conic Curve over Finite Field of size 5 defined by x^2 + y^2 + 2*z^2
  Defn: Defined on coordinates by sending (x : y : z) to
        (x - 2*z : y : z))
```

The diagonalization is only defined in characteristic different from 2:

```
sage: Conic(GF(2), [1,1,1,1,1,0]).diagonalization()
Traceback (most recent call last):
...
ValueError: The conic self (= Projective Conic Curve over Finite Field of size 2 defined by
```

gens()

Returns the generators of the coordinate ring of `self`.

EXAMPLES:

```
sage: P.<x,y,z> = QQ[]
sage: c = Conic(x^2+y^2+z^2)
sage: c.gens()
(xbar, ybar, zbar)
sage: c.defining_polynomial()(c.gens())
0
```

The function `gens()` is required for the following construction:

```
sage: C.<a,b,c> = Conic(GF(3), [1, 1, 1])
sage: C
Projective Conic Curve over Finite Field of size 3 defined by a^2 + b^2 + c^2
```

has_rational_point(*point=False*, *algorithm='default'*, *read_cache=True*)

Returns True if and only if the conic `self` has a point over its base field B.

If `point` is True, then returns a second output, which is a rational point if one exists.

Points are cached whenever they are found. Cached information is used if and only if `read_cache` is True.

ALGORITHM:

The parameter `algorithm` specifies the algorithm to be used:

- `'default'` – If the base field is real or complex, use an elementary native Sage implementation.
- `'magma'` (requires Magma to be installed) – delegates the task to the Magma computer algebra system.

EXAMPLES:

sage: Conic(RR, [1, 1, 1]).has_rational_point() False sage: Conic(CC, [1, 1, 1]).has_rational_point() True

sage: Conic(RR, [1, 2, -3]).has_rational_point(point = True) (True, (1.73205080756888 : 0.000000000000000 : 1.00000000000000))

Conics over polynomial rings can not be solved yet without Magma:

```
sage: R.<t> = QQ[]
sage: C = Conic([-2,t^2+1,t^2-1])
sage: C.has_rational_point()
Traceback (most recent call last):
...
NotImplementedError: has_rational_point not implemented for conics over base field Fraction
```

But they can be solved with Magma:

```
sage: C.has_rational_point(algorithm='magma') # optional - magma
True
sage: C.has_rational_point(algorithm='magma', point=True) # optional - magma
```

```
(True, (t : 1 : 1))

sage: D = Conic([t,1,t^2])
sage: D.has_rational_point(algorithm='magma') # optional - magma
False
```

TESTS:

One of the following fields comes with an embedding into the complex numbers, one does not. Check that they are both handled correctly by the Magma interface.

```
sage: K.<i> = QuadraticField(-1)
sage: K.coerce_embedding()
Generic morphism:
  From: Number Field in i with defining polynomial x^2 + 1
  To:   Complex Lazy Field
  Defn: i -> 1*I
sage: Conic(K, [1,1,1]).rational_point(algorithm='magma') # optional - magma
(-i : 1 : 0)

sage: x = QQ['x'].gen()
sage: L.<i> = NumberField(x^2+1, embedding=None)
sage: Conic(L, [1,1,1]).rational_point(algorithm='magma') # optional - magma
(-i : 1 : 0)
sage: L == K
False
```

has_singular_point (*point=False*)

Return True if and only if the conic `self` has a rational singular point.

If `point` is True, then also return a rational singular point (or `None` if no such point exists).

EXAMPLES:

```
sage: c = Conic(QQ, [1,0,1]); c
Projective Conic Curve over Rational Field defined by x^2 + z^2
sage: c.has_singular_point(point = True)
(True, (0 : 1 : 0))

sage: P.<x,y,z> = GF(7)[]
sage: e = Conic((x+y+z)*(x-y+2*z)); e
Projective Conic Curve over Finite Field of size 7 defined by x^2 - y^2 + 3*x*z + y*z + 2*z^
sage: e.has_singular_point(point = True)
(True, (2 : 4 : 1))

sage: Conic([1, 1, -1]).has_singular_point()
False
sage: Conic([1, 1, -1]).has_singular_point(point = True)
(False, None)
```

`has_singular_point` is not implemented over all fields of characteristic 2. It is implemented over finite fields.

```
sage: F.<a> = FiniteField(8)
sage: Conic([a, a+1, 1]).has_singular_point(point = True)
(True, (a + 1 : 0 : 1))

sage: P.<t> = GF(2)[]
sage: C = Conic(P, [t,t,1]); C
Projective Conic Curve over Fraction Field of Univariate Polynomial Ring in t over Finite Fi
```

```
sage: C.has_singular_point(point = False)
Traceback (most recent call last):
...
NotImplementedError: Sorry, find singular point on conics not implemented over all fields of
```

hom(*x*, *Y=None*)

Return the scheme morphism from `self` to `Y` defined by `x`. Here `x` can be a matrix or a sequence of polynomials. If `Y` is omitted, then a natural image is found if possible.

EXAMPLES:

Here are a few Morphisms given by matrices. In the first example, `Y` is omitted, in the second example, `Y` is specified.

```
sage: c = Conic([-1, 1, 1])
sage: h = c.hom(Matrix([[1,1,0],[0,1,0],[0,0,1]])); h
Scheme morphism:
  From: Projective Conic Curve over Rational Field defined by -x^2 + y^2 + z^2
  To:   Projective Conic Curve over Rational Field defined by -x^2 + 2*x*y + z^2
  Defn: Defined on coordinates by sending (x : y : z) to
        (x + y : y : z)
sage: h([-1, 1, 0])
(0 : 1 : 0)

sage: c = Conic([-1, 1, 1])
sage: d = Conic([4, 1, -1])
sage: c.hom(Matrix([[0, 0, 1/2], [0, 1, 0], [1, 0, 0]]), d)
Scheme morphism:
  From: Projective Conic Curve over Rational Field defined by -x^2 + y^2 + z^2
  To:   Projective Conic Curve over Rational Field defined by 4*x^2 + y^2 - z^2
  Defn: Defined on coordinates by sending (x : y : z) to
        (1/2*z : y : x)
```

`ValueError` is raised if the wrong codomain `Y` is specified:

```
sage: c = Conic([-1, 1, 1])
sage: c.hom(Matrix([[0, 0, 1/2], [0, 1, 0], [1, 0, 0]]), c)
Traceback (most recent call last):
...
ValueError: The matrix x (= [  0   0 1/2]
[  0   1   0]
[  1   0   0]) does not define a map from self (= Projective Conic Curve over Rational Field
```

is_diagonal()

Return True if and only if the conic has the form $a * x^2 + b * y^2 + c * z^2$.

EXAMPLES:

```
sage: c=Conic([1,1,0,1,0,1]); c
Projective Conic Curve over Rational Field defined by x^2 + x*y + y^2 + z^2
sage: d,t = c.diagonal_matrix()
sage: c.is_diagonal()
False
sage: c.diagonalization()[0].is_diagonal()
True
```

is_smooth()

Returns True if and only if `self` is smooth.

EXAMPLES:

```
sage: Conic([1,-1,0]).is_smooth()
False
sage: Conic(GF(2),[1,1,1,1,1,0]).is_smooth()
True
```

matrix()

Returns a matrix M such that $(x,y,z)M(x,y,z)^t$ is the defining equation of `self`.

The matrix M is upper triangular if the base field has characteristic 2 and symmetric otherwise.

EXAMPLES:

```
sage: R.<x, y, z> = QQ[]
sage: C = Conic(x^2 + x*y + y^2 + z^2)
sage: C.matrix()
[  1 1/2   0]
[1/2   1   0]
[  0   0   1]

sage: R.<x, y, z> = GF(2)[]
sage: C = Conic(x^2 + x*y + y^2 + x*z + z^2)
sage: C.matrix()
[1 1 1]
[0 1 0]
[0 0 1]
```

parametrization(*point=None*, *morphism=True*)

Return a parametrization f of `self` together with the inverse of f.

If `point` is specified, then that point is used for the parametrization. Otherwise, use `self.rational_point()` to find a point.

If `morphism` is True, then f is returned in the form of a Scheme morphism. Otherwise, it is a tuple of polynomials that gives the parametrization.

EXAMPLES:

An example over a finite field

```
sage: c = Conic(GF(2), [1,1,1,1,1,0])
sage: c.parametrization()
(Scheme morphism:
  From: Projective Space of dimension 1 over Finite Field of size 2
  To:   Projective Conic Curve over Finite Field of size 2 defined by x^2 + x*y
+ y^2 + x*z + y*z
  Defn: Defined on coordinates by sending (x : y) to
        (x*y + y^2 : x^2 + x*y : x^2 + x*y + y^2),
 Scheme morphism:
  From: Projective Conic Curve over Finite Field of size 2 defined by x^2 + x*y
+ y^2 + x*z + y*z
  To:   Projective Space of dimension 1 over Finite Field of size 2
  Defn: Defined on coordinates by sending (x : y : z) to
        (y : x))
```

An example with `morphism = False`

```
sage: R.<x,y,z> = QQ[]
sage: C = Curve(7*x^2 + 2*y*z + z^2)
sage: (p, i) = C.parametrization(morphism = False); (p, i)
([-2*x*y, 7*x^2 + y^2, -2*y^2], [-1/2*x, -1/2*z])
sage: C.defining_polynomial()(p)
```

```
0
sage: i[0](p) / i[1](p)
x/y
```

A ValueError is raised if self has no rational point

```
sage: C = Conic(x^2 + y^2 + 7*z^2)
sage: C.parametrization()
Traceback (most recent call last):
...
ValueError: Conic Projective Conic Curve over Rational Field defined by x^2 + y^2 + 7*z^2 ha
```

A ValueError is raised if self is not smooth

```
sage: C = Conic(x^2 + y^2)
sage: C.parametrization()
Traceback (most recent call last):
...
ValueError: The conic self (=Projective Conic Curve over Rational Field defined by x^2 + y^2
```

point (*v*, *check=True*)

Constructs a point on self corresponding to the input v.

If check is True, then checks if v defines a valid point on self.

If no rational point on self is known yet, then also caches the point for use by self.rational_point() and self.parametrization().

EXAMPLES

```
sage: c = Conic([1, -1, 1])
sage: c.point([15, 17, 8])
(15/8 : 17/8 : 1)
sage: c.rational_point()
(15/8 : 17/8 : 1)
sage: d = Conic([1, -1, 1])
sage: d.rational_point()
(-1 : 1 : 0)
```

random_rational_point (**args1*, ***args2*)

Return a random rational point of the conic self.

ALGORITHM:

1. Compute a parametrization f of self using self.parametrization().
2. Computes a random point $(x : y)$ on the projective line.
3. Output $f(x : y)$.

The coordinates x and y are computed using B.random_element, where B is the base field of self and additional arguments to random_rational_point are passed to random_element.

If the base field is a finite field, then the output is uniformly distributed over the points of self.

EXAMPLES

```
sage: c = Conic(GF(2), [1,1,1,1,1,0])
sage: [c.random_rational_point() for i in range(10)] # output is random
[(1 : 0 : 1), (1 : 0 : 1), (1 : 0 : 1), (0 : 1 : 1), (1 : 0 : 1), (0 : 0 : 1), (1 : 0 : 1),

sage: d = Conic(QQ, [1, 1, -1])
sage: d.random_rational_point(den_bound = 1, num_bound = 5) # output is random
```

```
(-24/25 : 7/25 : 1)

sage: Conic(QQ, [1, 1, 1]).random_rational_point()
Traceback (most recent call last):
...
ValueError: Conic Projective Conic Curve over Rational Field defined by x^2 + y^2 + z^2 has
```

rational_point (*algorithm='default'*, *read_cache=True*)

Return a point on `self` defined over the base field.

Raises `ValueError` if no rational point exists.

See `self.has_rational_point` for the algorithm used and for the use of the parameters `algorithm` and `read_cache`.

EXAMPLES:

Examples over **Q**

```
sage: R.<x,y,z> = QQ[]
sage: C = Conic(7*x^2 + 2*y*z + z^2)
sage: C.rational_point()
(0 : 1 : 0)

sage: C = Conic(x^2 + 2*y^2 + z^2)
sage: C.rational_point()
Traceback (most recent call last):
...
ValueError: Conic Projective Conic Curve over Rational Field defined by x^2 + 2*y^2 + z^2 ha

sage: C = Conic(x^2 + y^2 + 7*z^2)
sage: C.rational_point(algorithm = 'rnfisnorm')
Traceback (most recent call last):
...
ValueError: Conic Projective Conic Curve over Rational Field defined by x^2 + y^2 + 7*z^2 ha
```

Examples over number fields

```
sage: P.<x> = QQ[]
sage: L.<b> = NumberField(x^3-5)
sage: C = Conic(L, [3, 2, -5])
sage: p = C.rational_point(algorithm = 'rnfisnorm')
sage: p                                          # output is random
(60*b^2 - 196*b + 161 : -120*b^2 - 6*b + 361 : 1)
sage: C.defining_polynomial()(list(p))
0

sage: K.<i> = QuadraticField(-1)
sage: D = Conic(K, [3, 2, 5])
sage: D.rational_point(algorithm = 'rnfisnorm') # output is random
(-3 : 4*i : 1)

sage: L.<s> = QuadraticField(2)
sage: Conic(QQ, [1, 1, -3]).has_rational_point()
False
sage: E = Conic(L, [1, 1, -3])
sage: E.rational_point()                         # output is random
(-1 : -s : 1)
```

Currently Magma is better at solving conics over number fields than Sage, so it helps to use the algorithm

'magma' if Magma is installed:

```
sage: q = C.rational_point(algorithm = 'magma', read_cache=False) # optional - magma
sage: q                        # output is random, optional - magma
(-1 : -1 : 1)
sage: C.defining_polynomial()(list(p))          # optional - magma
0
sage: len(str(p)) / len(str(q)) > 2             # optional - magma
True

sage: D.rational_point(algorithm = 'magma', read_cache=False) # random, optional - magma
(1 : 2*i : 1)

sage: E.rational_point(algorithm='magma', read_cache=False) # random, optional - magma
(-s : 1 : 1)

sage: F = Conic([L.gen(), 30, -20])
sage: q = F.rational_point(algorithm='magma')    # optional - magma
sage: q                        # output is random, optional - magma
(-10/7*s + 40/7 : 5/7*s - 6/7 : 1)
sage: p = F.rational_point(read_cache=False)
sage: p                        # output is random
(788210*s - 1114700 : -171135*s + 242022 : 1)
sage: len(str(p)) > len(str(q))                 # optional - magma
True

sage: Conic([L.gen(), 30, -21]).has_rational_point(algorithm='magma') # optional - magma
False
```

Examples over finite fields

```
sage: F.<a> = FiniteField(7^20)
sage: C = Conic([1, a, -5]); C
Projective Conic Curve over Finite Field in a of size 7^20 defined by x^2 + (a)*y^2 + 2*z^2
sage: C.rational_point()  # output is random
(4*a^19 + 5*a^18 + 4*a^17 + a^16 + 6*a^15 + 3*a^13 + 6*a^11 + a^9 + 3*a^8 + 2*a^7 + 4*a^6 +
```

Examples over $\mathbf{R}$ and $\mathbf{C}$

```
sage: Conic(CC, [1, 2, 3]).rational_point()
(0 : 1.22474487139159*I : 1)

sage: Conic(RR, [1, 1, 1]).rational_point()
Traceback (most recent call last):
...
ValueError: Conic Projective Conic Curve over Real Field with 53 bits of precision defined b
```

singular_point()

Returns a singular rational point of `self`

EXAMPLES:

```
sage: Conic(GF(2), [1,1,1,1,1,1]).singular_point()
(1 : 1 : 1)
```

`ValueError` is raised if the conic has no rational singular point

```
sage: Conic(QQ, [1,1,1,1,1,1]).singular_point()
Traceback (most recent call last):
...
ValueError: The conic self (= Projective Conic Curve over Rational Field defined by x^2 + x*
```

symmetric_matrix()

The symmetric matrix M such that $(xyz)M(xyz)^t$ is the defining equation of `self`.

EXAMPLES

```
sage: R.<x, y, z> = QQ[]
sage: C = Conic(x^2 + x*y/2 + y^2 + z^2)
sage: C.symmetric_matrix()
[  1 1/4   0]
[1/4   1   0]
[  0   0   1]

sage: C = Conic(x^2 + 2*x*y + y^2 + 3*x*z + z^2)
sage: v = vector([x, y, z])
sage: v * C.symmetric_matrix() * v
x^2 + 2*x*y + y^2 + 3*x*z + z^2
```

upper_triangular_matrix()

The upper-triangular matrix M such that $(xyz)M(xyz)^t$ is the defining equation of `self`.

EXAMPLES:

```
sage: R.<x, y, z> = QQ[]
sage: C = Conic(x^2 + x*y + y^2 + z^2)
sage: C.upper_triangular_matrix()
[1 1 0]
[0 1 0]
[0 0 1]

sage: C = Conic(x^2 + 2*x*y + y^2 + 3*x*z + z^2)
sage: v = vector([x, y, z])
sage: v * C.upper_triangular_matrix() * v
x^2 + 2*x*y + y^2 + 3*x*z + z^2
```

variable_names()

Returns the variable names of the defining polynomial of `self`.

EXAMPLES:

```
sage: c=Conic([1,1,0,1,0,1], 'x,y,z')
sage: c.variable_names()
('x', 'y', 'z')
sage: c.variable_name()
'x'
```

The function `variable_names()` is required for the following construction:

```
sage: C.<p,q,r> = Conic(QQ, [1, 1, 1])
sage: C
Projective Conic Curve over Rational Field defined by p^2 + q^2 + r^2
```

CHAPTER

SIX

PROJECTIVE PLANE CONICS OVER A NUMBER FIELD

AUTHORS:

- Marco Streng (2010-07-20)

class sage.schemes.plane_conics.con_number_field.**ProjectiveConic_number_field**(*A*, *f*)

Bases: sage.schemes.plane_conics.con_field.ProjectiveConic_field

Create a projective plane conic curve over a number field. See Conic for full documentation.

EXAMPLES:

```
sage: K.<a> = NumberField(x^3 - 2, 'a')
sage: P.<X, Y, Z> = K[]
sage: Conic(X^2 + Y^2 - a*Z^2)
Projective Conic Curve over Number Field in a with defining polynomial x^3 - 2 defined by X^2 +
```

TESTS:

```
sage: K.<a> = NumberField(x^3 - 3, 'a')
sage: Conic([a, 1, -1])._test_pickling()
```

has_rational_point(*point=False*, *obstruction=False*, *algorithm='default'*, *read_cache=True*)

Returns True if and only if self has a point defined over its base field B.

If point and obstruction are both False (default), then the output is a boolean out saying whether self has a rational point.

If point or obstruction is True, then the output is a pair (out, S), where out is as above and:

- if point is True and self has a rational point, then S is a rational point,
- if obstruction is True, self has no rational point, then S is a prime or infinite place of B such that no rational point exists over the completion at S.

Points and obstructions are cached whenever they are found. Cached information is used for the output if available, but only if read_cache is True.

ALGORITHM:

The parameter algorithm specifies the algorithm to be used:

- 'rnfisnorm' – Use PARI's rnfisnorm (cannot be combined with obstruction = True)
- 'local' – Check if a local solution exists for all primes and infinite places of B and apply the Hasse principle. (Cannot be combined with point = True.)

- 'default' – Use algorithm 'rnfisnorm' first. Then, if no point exists and obstructions are requested, use algorithm 'local' to find an obstruction.
- 'magma' (requires Magma to be installed) – delegates the task to the Magma computer algebra system.

EXAMPLES:

An example over $\mathbf{Q}$

```
sage: C = Conic(QQ, [1, 113922743, -310146482690273725409])
sage: C.has_rational_point(point = True)
(True, (-76842858034579/5424 : -5316144401/5424 : 1))
sage: C.has_rational_point(algorithm = 'local', read_cache = False)
True
```

Examples over number fields

```
sage: K.<i> = QuadraticField(-1)
sage: C = Conic(K, [1, 3, -5])
sage: C.has_rational_point(point = True, obstruction = True)
(False, Fractional ideal (-i - 2))
sage: C.has_rational_point(algorithm = "rnfisnorm")
False
sage: C.has_rational_point(algorithm = "rnfisnorm", obstruction = True, read_cache=False)
Traceback (most recent call last):
...
ValueError: Algorithm rnfisnorm cannot be combined with obstruction = True in has_rational_p

sage: P.<x> = QQ[]
sage: L.<b> = NumberField(x^3-5)
sage: C = Conic(L, [1, 2, -3])
sage: C.has_rational_point(point = True, algorithm = 'rnfisnorm')
(True, (5/3 : -1/3 : 1))

sage: K.<a> = NumberField(x^4+2)
sage: Conic(QQ, [4,5,6]).has_rational_point()
False
sage: Conic(K, [4,5,6]).has_rational_point()
True
sage: Conic(K, [4,5,6]).has_rational_point(algorithm='magma', read_cache=False) # optional -
True
```

TESTS:

Create a bunch of conics over number fields and check whether has_rational_point runs without errors for algorithms 'rnfisnorm' and 'local'. Check if all points returned are valid. If Magma is available, then also check if the output agrees with Magma.

```
sage: P.<X> = QQ[]
sage: Q = P.fraction_field()
sage: c = [1, X/2, 1/X]
sage: l = Sequence(cartesian_product_iterator([c for i in range(3)]))
sage: l = l + [[X, 1, 1, 1, 1, 1]] + [[X, 1/5, 1, 1, 2, 1]]
sage: K.<a> = QuadraticField(-23)
sage: L.<b> = QuadraticField(19)
sage: M.<c> = NumberField(X^3+3*X+1)
sage: m = [[Q(b)(F.gen()) for b in a] for a in l for F in [K, L, M]]
sage: d = []
sage: c = []
sage: c = [Conic(a) for a in m if a != [0,0,0]]
```

```
sage: d = [C.has_rational_point(algorithm = 'rnfisnorm', point = True) for C in c] # long ti
sage: all([c[k].defining_polynomial()(Sequence(d[k][1])) == 0 for k in range(len(d)) if d[k]
True
sage: [C.has_rational_point(algorithm='local', read_cache=False) for C in c] == [o[0] for o
True
sage: [C.has_rational_point(algorithm = 'magma', read_cache=False) for C in c] == [o[0] for
True
```

Create a bunch of conics that are known to have rational points already over $\mathbf{Q}$ and check if points are found by `has_rational_point`.

```
sage: l = Sequence(cartesian_product_iterator([[-1, 0, 1] for i in range(3)]))
sage: K.<a> = QuadraticField(-23)
sage: L.<b> = QuadraticField(19)
sage: M.<c> = NumberField(x^5+3*x+1)
sage: m = [[F(b) for b in a] for a in l for F in [K, L, M]]
sage: c = [Conic(a) for a in m if a != [0,0,0] and a != [1,1,1] and a != [-1,-1,-1]]
sage: assert all([C.has_rational_point(algorithm = 'rnfisnorm') for C in c])
sage: assert all([C.defining_polynomial()(Sequence(C.has_rational_point(point = True)[1])) =
sage: assert all([C.has_rational_point(algorithm='local', read_cache=False) for C in c]) # l
```

is_locally_solvable(*p*)

Returns `True` if and only if `self` has a solution over the completion of the base field B of `self` at `p`. Here `p` is a finite prime or infinite place of B.

EXAMPLES:

```
sage: P.<x> = QQ[]
sage: K.<a> = NumberField(x^3 + 5)
sage: C = Conic(K, [1, 2, 3 - a])
sage: [p1, p2] = K.places()
sage: C.is_locally_solvable(p1)
False

sage: C.is_locally_solvable(p2)
True

sage: O = K.maximal_order()
sage: f = (2*O).factor()
sage: C.is_locally_solvable(f[0][0])
True

sage: C.is_locally_solvable(f[1][0])
False
```

local_obstructions(*finite=True*, *infinite=True*, *read_cache=True*)

Returns the sequence of finite primes and/or infinite places such that `self` is locally solvable at those primes and places.

If the base field is $\mathbf{Q}$, then the infinite place is denoted -1.

The parameters `finite` and `infinite` (both True by default) are used to specify whether to look at finite and/or infinite places. Note that `finite = True` involves factorization of the determinant of `self`, hence may be slow.

Local obstructions are cached. The parameter `read_cache` specifies whether to look at the cache before computing anything.

EXAMPLES

```
sage: K.<i> = QuadraticField(-1)
sage: Conic(K, [1, 2, 3]).local_obstructions()
[]

sage: L.<a> = QuadraticField(5)
sage: Conic(L, [1, 2, 3]).local_obstructions()
[Ring morphism:
  From: Number Field in a with defining polynomial x^2 - 5
  To:   Algebraic Real Field
  Defn: a |--> -2.236067977499790?, Ring morphism:
  From: Number Field in a with defining polynomial x^2 - 5
  To:   Algebraic Real Field
  Defn: a |--> 2.236067977499790?]
```

CHAPTER

SEVEN

PROJECTIVE PLANE CONICS OVER $\mathbb{Q}$

AUTHORS:

- Marco Streng (2010-07-20)
- Nick Alexander (2008-01-08)

class sage.schemes.plane_conics.con_rational_field.**ProjectiveConic_rational_field**(*A*, *f*)

Bases: sage.schemes.plane_conics.con_number_field.ProjectiveConic_number_field

Create a projective plane conic curve over $\mathbf{Q}$. See `Conic` for full documentation.

EXAMPLES:

```
sage: P.<X, Y, Z> = QQ[]
sage: Conic(X^2 + Y^2 - 3*Z^2)
Projective Conic Curve over Rational Field defined by X^2 + Y^2 - 3*Z^2
```

TESTS:

```
sage: Conic([2, 1, -1])._test_pickling()
```

has_rational_point(*point=False*, *obstruction=False*, *algorithm='default'*, *read_cache=True*)

Returns True if and only if `self` has a point defined over $\mathbf{Q}$.

If `point` and `obstruction` are both False (default), then the output is a boolean `out` saying whether `self` has a rational point.

If `point` or `obstruction` is True, then the output is a pair `(out, S)`, where `out` is as above and the following holds:

- if `point` is True and `self` has a rational point, then `S` is a rational point,
- if `obstruction` is True and `self` has no rational point, then `S` is a prime such that no rational point exists over the completion at `S` or -1 if no point exists over $\mathbf{R}$.

Points and obstructions are cached, whenever they are found. Cached information is used if and only if `read_cache` is True.

ALGORITHM:

The parameter `algorithm` specifies the algorithm to be used:

- `'qfsolve'` – Use Denis Simon's GP script `qfsolve` (see `sage.quadratic_forms.qfsolve.qfsolve`)
- `'rnfisnorm'` – Use PARI's function rnfisnorm (cannot be combined with `obstruction = True`)

- 'local' – Check if a local solution exists for all primes and infinite places of $\mathbf{Q}$ and apply the Hasse principle (cannot be combined with point = True)
- 'default' – Use 'qfsolve'
- 'magma' (requires Magma to be installed) – delegates the task to the Magma computer algebra system.

EXAMPLES:

```
sage: C = Conic(QQ, [1, 2, -3])
sage: C.has_rational_point(point = True)
(True, (1 : 1 : 1))
sage: D = Conic(QQ, [1, 3, -5])
sage: D.has_rational_point(point = True)
(False, 3)
sage: P.<X,Y,Z> = QQ[]
sage: E = Curve(X^2 + Y^2 + Z^2); E
Projective Conic Curve over Rational Field defined by X^2 + Y^2 + Z^2
sage: E.has_rational_point(obstruction = True)
(False, -1)
```

The following would not terminate quickly with algorithm = 'rnfisnorm'

```
sage: C = Conic(QQ, [1, 113922743, -310146482690273725409])
sage: C.has_rational_point(point = True)
(True, (-76842858034579/5424 : -5316144401/5424 : 1))
sage: C.has_rational_point(algorithm = 'local', read_cache = False)
True
sage: C.has_rational_point(point=True, algorithm='magma', read_cache=False) # optional - mag
(True, (30106379962113/7913 : 12747947692/7913 : 1))
```

TESTS:

Create a bunch of conics over $\mathbf{Q}$, check if has_rational_point runs without errors and returns consistent answers for all algorithms. Check if all points returned are valid.

```
sage: l = Sequence(cartesian_product_iterator([[-1, 0, 1] for i in range(6)]))
sage: c = [Conic(QQ, a) for a in l if a != [0,0,0] and a != (0,0,0,0,0,0)]
sage: d = []
sage: d = [[C]+[C.has_rational_point(algorithm = algorithm, read_cache = False, obstruction
sage: assert all([e[1][0] == e[2][0] and e[1][0] == e[3][0] for e in d])
sage: assert all([e[0].defining_polynomial()(Sequence(e[i][1])) == 0 for e in d for i in [2,
```

is_locally_solvable(*p*)

Returns True if and only if self has a solution over the p-adic numbers. Here p is a prime number or equals -1, infinity, or $\mathbf{R}$ to denote the infinite place.

EXAMPLES:

```
sage: C = Conic(QQ, [1,2,3])
sage: C.is_locally_solvable(-1)
False
sage: C.is_locally_solvable(2)
False
sage: C.is_locally_solvable(3)
True
sage: C.is_locally_solvable(QQ.hom(RR))
False
sage: D = Conic(QQ, [1, 2, -3])
sage: D.is_locally_solvable(infinity)
True
```

```
sage: D.is_locally_solvable(RR)
True
```

local_obstructions (*finite=True*, *infinite=True*, *read_cache=True*)

Returns the sequence of finite primes and/or infinite places such that self is locally solvable at those primes and places.

The infinite place is denoted -1.

The parameters `finite` and `infinite` (both True by default) are used to specify whether to look at finite and/or infinite places. Note that `finite = True` involves factorization of the determinant of `self`, hence may be slow.

Local obstructions are cached. The parameter `read_cache` specifies whether to look at the cache before computing anything.

EXAMPLES

```
sage: Conic(QQ, [1, 1, 1]).local_obstructions()
[2, -1]
sage: Conic(QQ, [1, 2, -3]).local_obstructions()
[]
sage: Conic(QQ, [1, 2, 3, 4, 5, 6]).local_obstructions()
[41, -1]
```

parametrization (*point=None*, *morphism=True*)

Return a parametrization f of `self` together with the inverse of f.

If `point` is specified, then that point is used for the parametrization. Otherwise, use `self.rational_point()` to find a point.

If `morphism` is True, then f is returned in the form of a Scheme morphism. Otherwise, it is a tuple of polynomials that gives the parametrization.

ALGORITHM:

Uses Denis Simon's GP script `qfparam`. See `sage.quadratic_forms.qfsolve.qfparam`.

EXAMPLES

```
sage: c = Conic([1,1,-1])
sage: c.parametrization()
(Scheme morphism:
  From: Projective Space of dimension 1 over Rational Field
  To:   Projective Conic Curve over Rational Field defined by x^2 + y^2 - z^2
  Defn: Defined on coordinates by sending (x : y) to
        (2*x*y : -x^2 + y^2 : x^2 + y^2),
 Scheme morphism:
  From: Projective Conic Curve over Rational Field defined by x^2 + y^2 - z^2
  To:   Projective Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (x : y : z) to
        (1/2*x : 1/2*y + 1/2*z))
```

An example with `morphism = False`

```
sage: R.<x,y,z> = QQ[]
sage: C = Curve(7*x^2 + 2*y*z + z^2)
sage: (p, i) = C.parametrization(morphism = False); (p, i)
([-2*x*y, 7*x^2 + y^2, -2*y^2], [-1/2*x, -1/2*z])
sage: C.defining_polynomial()(p)
0
```

```
sage: i[0](p) / i[1](p)
x/y
```

A `ValueError` is raised if `self` has no rational point

```
sage: C = Conic(x^2 + 2*y^2 + z^2)
sage: C.parametrization()
Traceback (most recent call last):
...
ValueError: Conic Projective Conic Curve over Rational Field defined by x^2 + 2*y^2 + z^2 ha
```

A `ValueError` is raised if `self` is not smooth

```
sage: C = Conic(x^2 + y^2)
sage: C.parametrization()
Traceback (most recent call last):
...
ValueError: The conic self (=Projective Conic Curve over Rational Field defined by x^2 + y^2
```

CHAPTER

EIGHT

PROJECTIVE PLANE CONICS OVER FINITE FIELDS

AUTHORS:

- Marco Streng (2010-07-20)

class sage.schemes.plane_conics.con_finite_field.**ProjectiveConic_finite_field**(*A*, *f*)

Bases: sage.schemes.plane_conics.con_field.ProjectiveConic_field, sage.schemes.plane_curves.projective_curve.ProjectiveCurve_finite_field

Create a projective plane conic curve over a finite field. See `Conic` for full documentation.

EXAMPLES:

```
sage: K.<a> = FiniteField(9, 'a')
sage: P.<X, Y, Z> = K[]
sage: Conic(X^2 + Y^2 - a*Z^2)
Projective Conic Curve over Finite Field in a of size 3^2 defined by X^2 + Y^2 + (-a)*Z^2
```

TESTS:

```
sage: K.<a> = FiniteField(4, 'a')
sage: Conic([a, 1, -1])._test_pickling()
```

count_points(*n*)

If the base field B of $self$ is finite of order q, then returns the number of points over $\mathbf{F}_q, ..., \mathbf{F}_{q^n}$.

EXAMPLES:

```
sage: P.<x,y,z> = GF(3)[]
sage: c = Curve(x^2+y^2+z^2); c
Projective Conic Curve over Finite Field of size 3 defined by x^2 + y^2 + z^2
sage: c.count_points(4)
[4, 10, 28, 82]
```

has_rational_point(*point=False*, *read_cache=True*, *algorithm='default'*)

Always returns `True` because self has a point defined over its finite base field B.

If `point` is True, then returns a second output S, which is a rational point if one exists.

Points are cached. If `read_cache` is True, then cached information is used for the output if available. If no cached point is available or `read_cache` is False, then random y-coordinates are tried if `self` is smooth and a singular point is returned otherwise.

EXAMPLES:

```
sage: Conic(FiniteField(37), [1, 2, 3, 4, 5, 6]).has_rational_point()
True

sage: C = Conic(FiniteField(2), [1, 1, 1, 1, 1, 0]); C
Projective Conic Curve over Finite Field of size 2 defined by x^2 + x*y + y^2 + x*z + y*z
sage: C.has_rational_point(point = True)  # output is random
(True, (0 : 0 : 1))

sage: p = next_prime(10^50)
sage: F = FiniteField(p)
sage: C = Conic(F, [1, 2, 3]); C
Projective Conic Curve over Finite Field of size 10000000000000000000000000000000000000000000000
sage: C.has_rational_point(point = True)  # output is random
(True,
 (14971942941468509742682168602989039212496867586852 : 752354657080177928927622020881747410

sage: F.<a> = FiniteField(7^20)
sage: C = Conic([1, a, -5]); C
Projective Conic Curve over Finite Field in a of size 7^20 defined by x^2 + (a)*y^2 + 2*z^2
sage: C.has_rational_point(point = True)  # output is random
(True,
 (a^18 + 2*a^17 + 4*a^16 + 6*a^13 + a^12 + 6*a^11 + 3*a^10 + 4*a^9 + 2*a^8 + 4*a^7 + a^6 + 4
```

TESTS:

```
sage: l = Sequence(cartesian_product_iterator([[0, 1] for i in range(6)]))
sage: bigF = GF(next_prime(2^100))
sage: bigF2 = GF(next_prime(2^50)^2, 'b')
sage: m = [[F(b) for b in a] for a in l for F in [GF(2), GF(4, 'a'), GF(5), GF(9, 'a'), bigF
sage: m += [[F.random_element() for i in range(6)] for j in range(20) for F in [GF(5), bigF]
sage: c = [Conic(a) for a in m if a != [0,0,0,0,0,0]]
sage: assert all([C.has_rational_point() for C in c])
sage: r = randrange(0, 5)
sage: assert all([C.defining_polynomial()(Sequence(C.has_rational_point(point = True)[1])) =
```

CHAPTER

NINE

PROJECTIVE PLANE CONICS OVER PRIME FINITE FIELDS

AUTHORS:

- Marco Streng (2010-07-20)

class sage.schemes.plane_conics.con_prime_finite_field.**ProjectiveConic_prime_finite_field**(*A*, *f*)

Bases: sage.schemes.plane_conics.con_finite_field.ProjectiveConic_finite_field, sage.schemes.plane_curves.projective_curve.ProjectiveCurve_prime_finite_field

Create a projective plane conic curve over a prime finite field. See Conic for full documentation.

EXAMPLES:

```
sage: P.<X, Y, Z> = FiniteField(5)[]
sage: Conic(X^2 + Y^2 - 2*Z^2)
Projective Conic Curve over Finite Field of size 5 defined by X^2 + Y^2 - 2*Z^2
```

TESTS:

```
sage: Conic([FiniteField(7)(1), 1, -1])._test_pickling()
```

CHAPTER

TEN

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: sage.structure.factory.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()`.

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 schemes 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 Fi
sage: E.j_invariant()
2988.97297297297
```

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

```
sage: x, y = var('x,y')
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 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 trac ticket #6657

```
sage: EllipticCurve(GF(144169),j=1728)
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 trac ticket #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
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 trac ticket #13100):

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

TESTS:

```
sage: R = ZZ['u', 'v']
sage: EllipticCurve(R, [1,1])
Elliptic Curve defined by y^2 = x^3 + x + 1 over Multivariate Polynomial Ring in u, v
over Integer Ring
```

We create a curve and a point over QQbar (see #6879):

```
sage: E = EllipticCurve(QQbar,[0,1])
sage: E(0)
(0 : 1 : 0)
sage: E.base_field()
Algebraic Field

sage: E = EllipticCurve(RR,[1,2]); E; E.base_field()
Elliptic Curve defined by y^2 = x^3 + 1.00000000000000*x + 2.00000000000000 over Real Field with
Real Field with 53 bits of precision
sage: EllipticCurve(CC,[3,4]); E; E.base_field()
Elliptic Curve defined by y^2 = x^3 + 3.00000000000000*x + 4.00000000000000 over Complex Field w
Elliptic Curve defined by y^2 = x^3 + 1.00000000000000*x + 2.00000000000000 over Real Field with
Real Field with 53 bits of precision
sage: E = EllipticCurve(QQbar,[5,6]); E; E.base_field()
Elliptic Curve defined by y^2 = x^3 + 5*x + 6 over Algebraic Field
Algebraic Field
```

See trac ticket #6657

```
sage: EllipticCurve(3,j=1728)
Traceback (most recent call last):
...
ValueError: First parameter (if present) must be a ring when j is specified

sage: EllipticCurve(GF(5),j=3/5)
Traceback (most recent call last):
...
ValueError: First parameter must be a ring containing 3/5
```

If the universe of the coefficients is a general field, the object constructed has type EllipticCurve_field. Otherwise it is EllipticCurve_generic. See trac ticket #9816

```
sage: E = EllipticCurve([QQbar(1),3]); E
Elliptic Curve defined by y^2 = x^3 + x + 3 over Algebraic Field
sage: type(E)
<class 'sage.schemes.elliptic_curves.ell_field.EllipticCurve_field_with_category'>

sage: E = EllipticCurve([RR(1),3]); E
Elliptic Curve defined by y^2 = x^3 + 1.00000000000000*x + 3.00000000000000 over Real Field with
sage: type(E)
<class 'sage.schemes.elliptic_curves.ell_field.EllipticCurve_field_with_category'>

sage: E = EllipticCurve([i,i]); E
Elliptic Curve defined by y^2 = x^3 + I*x + I over Symbolic Ring
sage: type(E)
<class 'sage.schemes.elliptic_curves.ell_field.EllipticCurve_field_with_category'>
sage: E.category()
Category of schemes over Symbolic Ring
sage: SR in Fields()
True

sage: F = FractionField(PolynomialRing(QQ,'t'))
sage: t = F.gen()
sage: E = EllipticCurve([t,0]); E
Elliptic Curve defined by y^2 = x^3 + t*x over Fraction Field of Univariate Polynomial Ring in t
sage: type(E)
<class 'sage.schemes.elliptic_curves.ell_field.EllipticCurve_field_with_category'>
sage: E.category()
Category of schemes over Fraction Field of Univariate Polynomial Ring in t over Rational Field
```

See trac ticket #12517:

```
sage: E = EllipticCurve([1..5])
sage: EllipticCurve(E.a_invariants())
Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5 over Rational Field
```

See trac ticket #11773:

```
sage: E = EllipticCurve()
Traceback (most recent call last):
...
TypeError: invalid input to EllipticCurve constructor
```

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],
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_categ
```

Note: Keyword arguments are currently only passed to the constructor for elliptic curves over $\mathbf{Q}$; 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
```

TESTS:

```
sage: from sage.schemes.elliptic_curves.constructor import EllipticCurve_from_Weierstrass_polync
sage: EllipticCurve_from_Weierstrass_polynomial(-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*, *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 `Jacobian()` instead. This will construct the same elliptic curve but you don't 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.
- `P` – a 3-tuple (x, y, z) defining a projective point on the curve $F = 0$. Need not be a flex, but see caveat on output.
- `morphism` – boolean (default: `True`). Whether to return the morphism or just the elliptic curve.

OUTPUT:

An elliptic curve in long Weierstrass form isomorphic to the curve $F = 0$.

If `morphism=True` is passed, then a birational equivalence between F and the Weierstrass curve is returned. If the point happens to be a flex, then this is an isomorphism.

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 + 2*x*y + 1/3*y = x^3 - x^2 - 1/3*x - 1/27 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 + 2*x*y + 20*y = x^3 - x^2 - 20*x - 400/3 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 equivalence to and from the Weierstrass form. We start with an example where `P` is a flex and the equivalence is an isomorphism:

```
sage: f = EllipticCurve_from_cubic(cubic, P, morphism=True)
sage: f
Scheme morphism:
  From: Closed subscheme of Projective Space of dimension 2 over Rational Field defined by:
        a^3 + b^3 + 60*c^3
  To:   Elliptic Curve defined by y^2 + 2*x*y + 20*y = x^3 - x^2 - 20*x - 400/3
        over Rational Field
  Defn: Defined on coordinates by sending (a : b : c) to
        (-c : -b + c : 1/20*a + 1/20*b)

sage: finv = f.inverse();  finv
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 + x^2*z + 2*x*y*z + y^2*z + 20*x*z^2 + 20*y*z^2 + 400/3*z^3

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

If the point is not a flex then the cubic can not be transformed to a Weierstrass equation by a linear transformation. The general birational transformation is quadratic:

```
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 - 722*x*y - 21870000*y = x^3
+ 23579*x^2 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: Closed subscheme of Projective Space of dimension 2 over Rational Field defined by:
        a^3 + 7*b^3 + 64*c^3
  To:   Elliptic Curve defined by y^2 - 722*x*y - 21870000*y =
        x^3 + 23579*x^2 over Rational Field
```

```
  Defn: Defined on coordinates by sending (a : b : c) to
        (-5/112896*a^2 - 17/40320*a*b - 1/1280*b^2 - 29/35280*a*c
         - 13/5040*b*c - 4/2205*c^2 :
         -4055/112896*a^2 - 4787/40320*a*b - 91/1280*b^2 - 7769/35280*a*c
         - 1993/5040*b*c - 724/2205*c^2 :
         1/4572288000*a^2 + 1/326592000*a*b + 1/93312000*b^2 + 1/142884000*a*c
         + 1/20412000*b*c + 1/17860500*c^2)

sage: finv = f.inverse();  finv
Scheme morphism:
  From: Elliptic Curve defined by y^2 - 722*x*y - 21870000*y =
        x^3 + 23579*x^2 over Rational Field
  To:   Closed subscheme of Projective Space of dimension 2 over Rational Field defined by:
        a^3 + 7*b^3 + 64*c^3
  Defn: Defined on coordinates by sending (x : y : z) to
        (2*x^2 + 227700*x*z - 900*y*z :
         2*x^2 - 32940*x*z + 540*y*z :
         -x^2 - 56520*x*z - 180*y*z)

sage: cubic(finv.defining_polynomials()) * finv.post_rescaling()
-x^3 - 23579*x^2*z - 722*x*y*z + y^2*z - 21870000*y*z^2

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

TESTS:

```
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: EllipticCurve_from_cubic(cubic, [1,-1,1], morphism=False)
Elliptic Curve defined by y^2 - 882*x*y - 2560000*y = x^3 - 127281*x^2 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 $\mathbf{Q}$, the curve returned is a minimal twist, i.e. has minimal conductor. If j is not in $\mathbf{Q}$ 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
'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 $\mathbf{Q}$ 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.**EllipticCurve_from_plane_curve**(*C*, *P*)

Deprecated way to construct an elliptic curve.

Use `Jacobian()` instead.

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: C = Curve(x^3+y^3+z^3)
sage: P = C(1,-1,0)
sage: E = EllipticCurve_from_plane_curve(C,P); E  # long time (3s on sage.math, 2013)
doctest:...: DeprecationWarning: use Jacobian(C) instead
See http://trac.sagemath.org/3416 for details.
Elliptic Curve defined by y^2 = x^3 - 27/4 over Rational Field
```

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

Returns 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` - 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 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,
...
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
```

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

```
sage: elist = EllipticCurves_with_good_reduction_outside_S([11],proof=False)  # long time (14s c
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: elist = EllipticCurves_with_good_reduction_outside_S([2,3]) # long time (26s on sage.math,
sage: len(elist) # long time
752
sage: max([e.conductor() for e in elist]) # long time
62208
sage: [N.factor() for N in Set([e.conductor() for e in elist])] # long time
[2^7,
2^8,
2^3 * 3^4,
2^2 * 3^3,
2^8 * 3^4,
2^4 * 3^4,
2^3 * 3,
2^7 * 3,
2^3 * 3^5,
3^3,
2^8 * 3,
2^5 * 3^4,
2^4 * 3,
2 * 3^4,
2^2 * 3^2,
2^6 * 3^4,
2^6,
2^7 * 3^2,
2^4 * 3^5,
2^4 * 3^3,
2 * 3^3,
2^6 * 3^3,
2^6 * 3,
2^5,
2^2 * 3^4,
2^3 * 3^2,
2^5 * 3,
2^7 * 3^4,
2^2 * 3^5,
```

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

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:

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*)

Use the chord and tangent method to get another point on a cubic.

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:

Another point satisfying the equation F.

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]

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, -183, -314]
```

TESTS:

```
sage: F(p2)
0
sage: map(type, p2)
[<type 'sage.rings.rational.Rational'>,
 <type 'sage.rings.rational.Rational'>,
 <type 'sage.rings.rational.Rational'>]
```

See trac ticket #16068:

```
sage: F = x**3 - 4*x**2*y - 65*x*y**2 + 3*x*y*z - 76*y*z**2
sage: chord_and_tangent(F, [0, 1, 0])
[0, 0, -1]
```

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

Return the coefficients $(a_1, a_2, a_3, a_4, a_5)$ for a cubic in Weierstrass form.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.constructor import coefficients_from_Weierstrass_polynom
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)
sage: projective_point([F(4), F(8), F(2)])
[4, 8, 2]
```

CHAPTER

ELEVEN

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:

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*v^7*w + u*v^4*w^4 + u*v*w^7 :
         v^9 + 3/2*v^6*w^3 - 3/2*v^3*w^6 - w^9 :
         -v^6*w^3 - v^3*w^6)
```

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 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*b^7*c + 12960000*a*b^4*c^4 + 777600000*a*b*c^7 :
         216000*b^9 + 19440000*b^6*c^3 - 1166400000*b^3*c^6 - 46656000000*c^9 :
         -216000*b^6*c^3 - 12960000*b^3*c^6)
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()
(-100776960000000000) * c^3 * b^3 * (a^3 + b^3 + 60*c^3) * (b^6 + 60*b^3*c^3 + 3600*c^6)^3
```

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
```

TESTS:

```
sage: from sage.schemes.elliptic_curves.jacobian import Jacobian_of_equation
sage: Jacobian_of_equation(f, variables=[a,b,c])
Elliptic Curve defined by y^2 = x^3 - 24300 over Rational Field
```

CHAPTER

TWELVE

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, in Magma, 'Weierstrass Model' means a model with $a1 = a2 = a3 = 0$, 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 = 97; a=1; b=3
sage: R, u = PolynomialRing(GF(p), 'u').objgen()
sage: S, v = PolynomialRing(R, 'v').objgen()
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 o
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

class sage.schemes.elliptic_curves.ell_generic.**EllipticCurve_generic**(*K*, *ainvs*)

Bases: sage.misc.fast_methods.WithEqualityById, sage.schemes.plane_curves.projective_curve

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

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

a1()
Returns the a_1 invariant of this elliptic curve.

EXAMPLES:

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

a2()
Returns the a_2 invariant of this elliptic curve.

EXAMPLES:

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

a3()
Returns the a_3 invariant of this elliptic curve.

EXAMPLES:

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

a4()
Returns the a_4 invariant of this elliptic curve.

EXAMPLES:

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

a6()
Returns 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])
sage: 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)

sage: E = EllipticCurve([1,0,0,0,1])
sage: E.a_invariants()[0] = 100000000
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
```

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])
sage: 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)

sage: E = EllipticCurve([1,0,0,0,1])
sage: E.a_invariants()[0] = 100000000
Traceback (most recent call last):
...
TypeError: 'tuple' object does not support item assignment
```

automorphisms(*field=None*)

Return the set of isomorphisms from self to itself (as a 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();
[Generic endomorphism of Abelian group of points on Elliptic Curve defined by y^2 + y = x^3
Via:  (u,r,s,t) = (-1, 0, 0, -1), Generic endomorphism of Abelian group of points on Ellipti
Via:  (u,r,s,t) = (1, 0, 0, 0)]
```

We can also find automorphisms defined over extension fields:

```
sage: K.<a> = NumberField(x^2+3) # adjoin roots of unity
sage: E.automorphisms(K)
[Generic endomorphism of Abelian group of points on Elliptic Curve defined by y^2 + y = x^3
Via:  (u,r,s,t) = (-1, 0, 0, -1),
...
Generic endomorphism of Abelian group of points on Elliptic Curve defined by y^2 + y = x^3 c
Via:  (u,r,s,t) = (1, 0, 0, 0)]

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

b2()

Returns the b_2 invariant of this elliptic curve.

EXAMPLES:

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

b4()

Returns the b_4 invariant of this elliptic curve.

EXAMPLES:

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

b6()

Returns the b_6 invariant of this elliptic curve.

EXAMPLES:

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

b8()

Returns the b_8 invariant of this elliptic curve.

EXAMPLES:

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

b_invariants()

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

OUTPUT:

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

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
Elliptic Curve defined by y^2 = x^3 + x + 1 over Finite Field in a of size 5^3
```

base_ring()

Returns the base ring of the elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve(GF(49, 'a'), [3,5])
sage: E.base_ring()
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()

Returns the c_4 invariant of this elliptic curve.

EXAMPLES:

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

c6()

Returns the c_6 invariant of this elliptic curve.

EXAMPLES:

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

c_invariants()

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

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: F2=GF(5^2,'a'); a=F2.gen()
sage: F4=GF(5^4,'b'); b=F4.gen()
sage: h=F2.hom([a.charpoly().roots(ring=F4,multiplicities=False)[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
```

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 - 58/1176
sage: F1.is_isomorphic(F2)
True
```

discriminant()

Returns the discriminant of this elliptic curve.

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*)

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

INPUT:

- `m` - positive integer.
- `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 ok. 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. Similarly 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: when x is None, a bivariate polynomial over the base ring of the curve is returned, which includes a factor $2 * y + a1 * x + a3$ which has 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.

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)]
[<type 'sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint'>,
 <type 'sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular'>,
 <type '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*
```

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*)

Returns 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 [MazurTate1991], 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:

`multiple_x_numerator()` `multiple_x_denominator()` `division_polynomial()`

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 [MazurTate1991]); 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 [MazurTate1991]. 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

REFERENCES:

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 -
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
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 - 50
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

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
```

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: 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:
...       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 : 23 : 1), (9 : -33 : 1)], [(2 : 2 : 1), (2 : -5 : 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()

The formal group associated to this elliptic curve.

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 Fie
```

formal_group()

The formal group associated to this elliptic curve.

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 Fie
```

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()

Returns 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)
```

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

Returns 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*)

Returns 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*)

Returns 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()

TEST:

```
sage: E=EllipticCurve('5077a1')
sage: [x for x in srange(-10,10) if E.is_x_coord (x)]
[-3, -2, -1, 0, 1, 2, 3, 4, 8]

sage: F=GF(32,'a')
sage: E=EllipticCurve(F,[1,0,0,0,1])
sage: set([P[0] for P in E.points() if P!=E(0)]) == set([x for x in F if E.is_x_coord(x)])
True
```

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
Generic morphism:
From: Abelian group of points on Elliptic Curve defined by y^2 + y = x^3 - x over Rational F
To:   Abelian group of points on Elliptic Curve defined by y^2  = x^3 - 16*x + 16 over Ratio
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: K.<a> = NumberField(x^3-7)
sage: E.isomorphism_to(E.change_ring(K))
Generic morphism:
  From: Abelian group of points on Elliptic Curve defined by y^2 + y = x^3 - x over Rational
  To:   Abelian group of points on Elliptic Curve defined by y^2 + y = x^3 + (-1)*x over Num
  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);
[Generic endomorphism of Abelian group of points on Elliptic Curve defined by y^2 + y = x^3
  Via:  (u,r,s,t) = (-1, 0, 0, -1),
 Generic endomorphism of Abelian group of points on Elliptic Curve defined by y^2 + y = x^3
  Via:  (u,r,s,t) = (1, 0, 0, 0)]
```

We can also find isomorphisms defined over extension fields:

```
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'))
[Generic morphism:
From: Abelian group of points on Elliptic Curve defined by y^2 = x^3 + x + 1 over Finite Fie
To:   Abelian group of points on Elliptic Curve defined by y^2 = x^3 + x + 6 over Finite Fie
Via:  (u,r,s,t) = (a + 3, 0, 0, 0), Generic morphism:
From: Abelian group of points on Elliptic Curve defined by y^2 = x^3 + x + 1 over Finite Fie
To:   Abelian group of points on Elliptic Curve defined by y^2 = x^3 + x + 6 over Finite Fie
Via:  (u,r,s,t) = (6*a + 4, 0, 0, 0)]
```

j_invariant()

Returns the j-invariant of this elliptic curve.

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*)

Returns one or all points with given x-coordinate.

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.

Note: 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 : 0 : 1)
sage: E.lift_x(2)
(2 : 2 : 1)
sage: E.lift_x(1/4, all=True)
[(1/4 : -3/8 : 1), (1/4 : -5/8 : 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
```

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

```
sage: E.change_ring(RR).lift_x(3, all=True)
[(3.00000000000000 : 4.42442890089805 : 1.00000000000000), (3.00000000000000 : -5.4244289008
```

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)
```

We can perform these operations over finite fields too:

```
sage: E = E.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 : 11 : 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 o
```

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:

```
sage: E = EllipticCurve('37a');
sage: E.lift_x(pAdicField(17, 5)(6))
(6 + O(17^5) : 2 + 16*17 + 16*17^2 + 16*17^3 + 16*17^4 + O(17^5) : 1 + O(17^5))
sage: K.<t> = PowerSeriesRing(QQ, 't', 5)
sage: E.lift_x(1+t)
(1 + t : 2*t - t^2 + 5*t^3 - 21*t^4 + O(t^5) : 1)
sage: K.<a> = GF(16)
sage: E = E.change_ring(K)
sage: E.lift_x(a^3)
(a^3 : a^3 + a : 1)
```

AUTHOR:

- Robert Bradshaw (2007-04-24)

TEST:

```
sage: E = EllipticCurve('37a').short_weierstrass_model().change_ring(GF(17))
sage: E.lift_x(3, all=True)
[]
sage: E.lift_x(7, all=True)
[(7 : 3 : 1), (7 : 14 : 1)]
```

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 + 9
```

Grab only the x-coordinate (less work):

```
sage: mx = E.multiplication_by_m(2, x_only=True); mx
(x^4 + 2*x^2 - 24*x + 1)/(4*x^3 - 4*x + 12)
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
```

TESTS:

Verify for this fairly random looking curve and point that multiplication by m returns the right result for the first 10 integers:

```
sage: E = EllipticCurve([23,-105])
sage: P = E([129/4, 1479/8])
sage: for n in [1..10]:
....:     f = E.multiplication_by_m(n)
....:     Q = n*P
....:     assert Q == E(eval(f,P))
....:     f = E.multiplication_by_m(-n)
....:     Q = -n*P
....:     assert Q == E(eval(f,P))
```

The following test shows that trac ticket #4364 is indeed fixed:

```
sage: p = next_prime(2^30-41)
sage: a = GF(p)(1)
sage: b = GF(p)(1)
sage: E = EllipticCurve([a, b])
sage: P = E.random_point()
sage: my_eval = lambda f,P: [fi(P[0],P[1]) for fi in f]
sage: f = E.multiplication_by_m(2)
sage: assert(E(eval(f,P)) == 2*P)
```

multiplication_by_m_isogeny(*m*)

Return the `EllipticCurveIsogeny` object associated to the multiplication-by-m map on self. The resulting isogeny will have the associated rational maps (i.e. those returned by $self.multiplication_by_m()$) 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 trac tickets #7368 and #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 self.

EXAMPLES:

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

pari_curve()

Return the PARI curve corresponding to this elliptic curve.

The result is cached.

EXAMPLES:

```
sage: E = EllipticCurve([RR(0), RR(0), RR(1), RR(-1), RR(0)])
sage: e = E.pari_curve()
sage: type(e)
<type 'sage.libs.pari.gen.gen'>
sage: e.type()
't_VEC'
sage: e.disc()
37.0000000000000
```

Over a finite field:

```
sage: EllipticCurve(GF(41),[2,5]).pari_curve()
[Mod(0, 41), Mod(0, 41), Mod(0, 41), Mod(2, 41), Mod(5, 41), Mod(0, 41), Mod(4, 41), Mod(20,
```

Over a p-adic field:

```
sage: Qp = pAdicField(5, prec=3)
sage: E = EllipticCurve(Qp,[3, 4])
sage: E.pari_curve()
[O(5^3), O(5^3), O(5^3), 3 + O(5^3), 4 + O(5^3), O(5^3), 1 + 5 + O(5^3), 1 + 3*5 + O(5^3), 1
sage: E.j_invariant()
3*5^-1 + O(5)
```

The j-invariant must have negative p-adic valuation:

```
sage: E = EllipticCurve(Qp,[1, 1])
sage: E.j_invariant() # the j-invariant is a p-adic integer
2 + 4*5^2 + O(5^3)
sage: E.pari_curve()
Traceback (most recent call last):
...
PariError: valuation of j must be negative in p-adic ellinit
```

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

Draw a graph of this elliptic curve.

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))
sage: E = EllipticCurve('37a')
sage: plot(E)
sage: plot(E, xmin=25,xmax=26)
```

With #12766 we added the components keyword:

```
sage: E.real_components()
2
sage: E.plot(components='bounded')
sage: E.plot(components='unbounded')
```

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

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

rst_transform(*r*, *s*, *t*)

Returns 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^
```

scale_curve(*u*)

Returns 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 Fra
```

short_weierstrass_model(*complete_cube=True*)

Returns a short Weierstrass model for self.

INPUT:

- `complete_cube` - bool (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: print 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: print 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: print F
Elliptic Curve defined by y^2  = x^3 + 9*x^2 + 88*x + 464 over Rational Field
sage: print 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 trac #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
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*)

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

INPUT:

- `m` - positive integer.
- `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 ok. 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. Similarly 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: when x is None, a bivariate polynomial over the base ring of the curve is returned, which includes a factor $2 * y + a1 * x + a3$ which has 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.

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)]
[<type 'sage.rings.polynomial.polynomial_rational_flint.Polynomial_rational_flint'>,
 <type 'sage.rings.polynomial.multi_polynomial_libsingular.MPolynomial_libsingular'>,
 <type '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*
```

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*)

Returns 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 ok. 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])
sage: E.two_division_polynomial()
x^3 + 2*x^2 + 2
sage: E.two_division_polynomial().roots()
[(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)
True
sage: is_EllipticCurve(0)
False
```

CHAPTER

THIRTEEN

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`**(*K*, *ainvs*)

Bases: `sage.schemes.elliptic_curves.ell_generic.EllipticCurve_generic`

Construct an elliptic curve from Weierstrass a-coefficients.

INPUT:

- `K` – a ring
- `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:

```
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: E = EllipticCurve(GF(7),[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 Finite Field of size 7
```

Constructor from $[a_4, a_6]$ sets $a_1 = a_2 = a_3 = 0$:

```
sage: EllipticCurve([4,5]).ainvs()
(0, 0, 0, 4, 5)
```

The base ring need not be a field:

```
sage: EllipticCurve(IntegerModRing(91),[1,2,3,4,5])
Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 4*x + 5 over Ring of integers modulo 9
```

`base_field`()

Returns the base ring of the elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve(GF(49, 'a'), [3,5])
sage: E.base_ring()
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: F.<b> = QuadraticField(23)
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
sage: all([Ei.change_ring(G).is_isomorphic(E) for Ei in EF])
True

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
Elliptic Curve defined by y^2 = x^3 + 7*b*x over Number Field in b with defining polynomial
sage: all([Ei.change_ring(L).is_isomorphic(E) for Ei in EK])
True

sage: K.<a> = QuadraticField(17)
sage: E = EllipticCurve(j = 2*a)
sage: E.descend_to(QQ)
[]
```

TESTS:

Check that trac ticket #16456 is fixed:

```
sage: K.<a> = NumberField(x^3-2)
sage: E = EllipticCurve('11a1').quadratic_twist(2)
sage: EK = E.change_ring(K)
sage: EK2 = EK.change_weierstrass_model((a,a,a,a+1))
sage: EK2.descend_to(QQ)
[Elliptic Curve defined by y^2 = x^3 + x^2 - 41*x - 199 over Rational Field]

sage: k.<i> = QuadraticField(-1)
sage: E = EllipticCurve(k,[0,0,0,1,0])
sage: E.descend_to(QQ)
[Elliptic Curve defined by y^2 = x^3 + x over Rational Field,
Elliptic Curve defined by y^2 = x^3 - 4*x over Rational Field]
```

hasse_invariant()

Returns 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 returned.

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: 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*)

Returns 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 Fie
sage: E2 = EllipticCurve(CC, [2,7]); E2
Elliptic Curve defined by y^2 = x^3 + 2.00000000000000*x + 7.00000000000000 over Complex Fie
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
sage: E2 = EllipticCurve(CC, [23,4]); E2
Elliptic Curve defined by y^2 = x^3 + 23.0000000000000*x + 4.00000000000000 over Complex Fie
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 curves 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 $\mathbb{Q}$, 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))    #trac 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: 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: 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: 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)
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*)

Generic code, valid for all fields, for arbitrary prime l not equal to the characteristic.

INPUT:

- `l` – either None, a prime or a list of primes.
- `max_l` – a bound on the primes to be tested (ignored unless l is None).

OUTPUT:

(list) All l-isogenies for the given l with domain self.

METHOD:

Calls the generic function `isogenies_prime_degree()`. This requires that certain operations have been implemented over the base field, such as root-finding for univariate polynomials.

EXAMPLES:

```
sage: F = QQbar
sage: E = EllipticCurve(F, [1,18]); E
Elliptic Curve defined by y^2 = x^3 + x + 18 over Algebraic Field
sage: E.isogenies_prime_degree()
Traceback (most recent call last):
...
NotImplementedError: This code could be implemented for QQbar, but has not been yet.

sage: F = CC
sage: E = EllipticCurve(F, [1,18]); E
Elliptic Curve defined by y^2 = x^3 + 1.00000000000000*x + 18.0000000000000 over Complex Fie
sage: E.isogenies_prime_degree(11)
Traceback (most recent call last):
...
NotImplementedError: This code could be implemented for general complex fields, but has not
```

Examples over finite fields:

```
sage: E = EllipticCurve(GF(next_prime(1000000)), [7,8])
sage: E.isogenies_prime_degree()
[Isogeny of degree 2 from Elliptic Curve defined by y^2 = x^3 + 7*x + 8 over Finite Field of
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
sage: E.isogenies_prime_degree(3)
[]
sage: E.isogenies_prime_degree(5)
[]
sage: E.isogenies_prime_degree(7)
[]
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 o
Isogeny of degree 13 from Elliptic Curve defined by y^2 = x^3 + 7*x + 8 over Finite Field of

sage: E.isogenies_prime_degree([2, 3, 5, 7, 13])
[Isogeny of degree 2 from Elliptic Curve defined by y^2 = x^3 + 7*x + 8 over Finite Field of
sage: E.isogenies_prime_degree([2, 4])
Traceback (most recent call last):
...
ValueError: 4 is not prime.
sage: E.isogenies_prime_degree(4)
Traceback (most recent call last):
...
```

```
ValueError: 4 is not prime.
sage: E.isogenies_prime_degree(11)
[]
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 siz

sage: E = EllipticCurve(GF(13^4, 'a'),[2,8])
sage: E.isogenies_prime_degree(2)
[Isogeny of degree 2 from Elliptic Curve defined by y^2 = x^3 + 2*x + 8 over Finite Field in

sage: E.isogenies_prime_degree(3)
[Isogeny of degree 3 from Elliptic Curve defined by y^2 = x^3 + 2*x + 8 over Finite Field in
```

Example to show that separable isogenies of degree equal to the characteristic are now implemented:

```
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 i
```

Examples over number fields (other than QQ):

```
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 + (-629407744
Isogeny of degree 2 from Elliptic Curve defined by y^2 = x^3 + (-150528000)*x + (-629407744(
Isogeny of degree 2 from Elliptic Curve defined by y^2 = x^3 + (-150528000)*x + (-629407744(

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 defin
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 Nu
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 Nu
```

isogeny (*kernel*, *codomain=None*, *degree=None*, *model=None*, *check=True*)

Returns an elliptic curve isogeny from self.

The isogeny can be determined in two ways, either by a polynomial or a set of torsion points. The methods used are:

- Velu's Formulas: Velu's original formulas for computing isogenies. This algorithm is selected by giving as the `kernel` parameter a point or a list of points which generate a finite subgroup.
- Kohel's Formulas: Kohel's original formulas for computing isogenies. This algorithm is selected by giving as the `kernel` parameter a polynomial (or a coefficient list (little endian)) which will define the kernel of the isogeny.

INPUT:

- **`E` - an elliptic curve, the domain of the isogeny to** initialize.
- **`kernel` - a kernel, either a point in `E`, a list of points** in `E`, a univariate kernel polynomial or `None`. If initiating from a domain/codomain, this must be set to None. Validity of input is *not* fully checked.
- **`codomain` - an elliptic curve (default:None). If `kernel` is** None, then this must be the codomain of a separable normalized isogeny, furthermore, `degree` must be the degree of the isogeny from

E to codomain. If kernel is not None, then this must be isomorphic to the codomain of the normalized separable isogeny defined by kernel, in this case, the isogeny is post composed with an isomorphism so that this parameter is the codomain.

- **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 kernel polynomial with the two torsion polynomial of E.
- **model - a string (default:None). Only supported variable is** "minimal", in which case if ''E'' is a curve over the rationals, then the codomain is set to be the unique global minimum model.
- **check (default: True) does some partial checks that the** input is valid (e.g., that the points defined by the kernel polynomial are torsion); however, invalid input can in some cases still pass, since that the points define a group is not checked.

OUTPUT:

An isogeny between elliptic curves. This is a morphism of curves.

EXAMPLES:

```
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 Rati

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
Isogeny of degree 21 from Elliptic Curve defined by y^2 = x^3 + x + 1 over Finite Field of s
sage: phi(E.random_point()) # all points defined over GF(19) are in the kernel
(0 : 1 : 0)


# not all polynomials define a finite subgroup trac #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 does not define a finite subgroup of the elliptic curve.
```

An example in which we construct an invalid morphism, which illustrates that the check for correctness of the input is not sufficient. (See trac 11578.):

```
sage: R.<x> = QQ[]
sage: K.<a> = NumberField(x^2-x-1)
sage: E = EllipticCurve(K, [-13392, -1080432])
sage: R.<x> = K[]
sage: phi = E.isogeny( (x-564)*(x - 396/5*a + 348/5) )
sage: phi.codomain().conductor().norm().factor()
5^2 * 11^2 * 3271 * 15806939 * 4169267639351
sage: phi.domain().conductor().norm().factor()
```

```
11^2
```

isogeny_codomain(*kernel*, *degree=None*)

Returns 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 a either a univariate polynomial or a coefficient list.)
- **degree - an integer, (default:None) optionally specified degree** of the kernel.

OUTPUT:

An elliptic curve, the codomain of the separable normalized isogeny from 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 Fie
```

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: 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: 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) ove
sage: E.is_isomorphic(Et)
False
sage: E.j_invariant()==Et.j_invariant()
True

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 Finit
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: 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 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 0.

EXAMPLES:

```
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())
sage: E.base_extend(K).two_torsion_rank()
1
sage: E.reduction(53).two_torsion_rank()
2

sage: E = EllipticCurve('14a1')
sage: E.two_torsion_rank()
1
sage: K.<alpha>=QQ.extension(E.division_polynomial(2).monic().factor()[1][0])
sage: E.base_extend(K).two_torsion_rank()
2

sage: EllipticCurve('15a1').two_torsion_rank()
2
```

weierstrass_p(*prec=20*, *algorithm=None*)

Computes the Weierstrass $\wp$-function of the elliptic curve.

INPUT:

- `mprec` - precision
- **`algorithm` - string (default:`None`) an algorithm identifier** indicating using the `pari`, `fast` or `quadratic` algorithm. If the algorithm is `None`, then this function determines 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
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
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
```

CHAPTER

FOURTEEN

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

class sage.schemes.elliptic_curves.ell_rational_field.**EllipticCurve_rational_field**(*ainvs*, ***kwds*)

Bases: sage.schemes.elliptic_curves.ell_number_field.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 [CPS] paper is given for number fields. It's only the implementation in Sage that restricts to the rational field.

Lambda(*s*, *prec*)

Returns 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 and 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*)

Computes all S-integral points (up to sign) on this elliptic curve.

INPUT:

- S - list of primes
- mw_base - list of EllipticCurvePoint generating the Mordell-Weil group of E (default: 'auto' - calls gens())
- both_signs - True/False (default False): if True the output contains both P and -P, otherwise only one of each pair.
- verbose - True/False (default False): if True, some details of the computation are output.
- proof - True/False (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)); P2=E.point((-1,3)); 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 6.68597129142710e234 1.31952866480763 3.31908110593519e9 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
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
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
k5,k6,k7 0.321154513240... 1.55246328915... 0.161999172489...
initial bound 2.6227097483365...e117
bound_list [58, 58, 58]
bound_list [8, 9, 9]
bound_list [8, 7, 7]
bound_list [8, 7, 7]
starting search of points using coefficient bound  8
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,
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]
Total number of S-integral points: 43
[(-3 : 0 : 1), (-26/9 : 28/27 : 1), (-8159/2916 : 233461/157464 : 1), (-2759/1024 : 60819/32
```

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), (4 : 18 : 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])] #
[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
```

An example from [PZGH]:

```
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 #4805 was fixed:

```
sage: EllipticCurve([1,1,1,-301,-1821]).S_integral_points([13,2])
[(-13 : 16 : 1),
(-9 : 20 : 1),
(-7 : 4 : 1),
(21 : 30 : 1),
(23 : 52 : 1),
(63 : 452 : 1),
(71 : 548 : 1),
(87 : 756 : 1),
```

```
(2711 : 139828 : 1),
(7323 : 623052 : 1),
(17687 : 2343476 : 1)]
```

REFERENCES:

- [PZGH] Petho A., Zimmer H.G., Gebel J. and Herrmann E., Computing all S-integral points on elliptic curves Math. Proc. Camb. Phil. Soc. (1999), 127, 383-402
- 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)

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. If leading_coefficient is `True` (only implemented for PARI), return a tuple $(rank, lead)$ where $lead$ is the value of the first non-zero derivative of the L-function of the elliptic curve.

INPUT:

- algorithm -
  - 'pari' (default) - use the PARI library function.
  - 'sympow' -use Watkins's program sympow
  - 'rubinstein' - use Rubinstein's L-function C++ program lcalc.
  - 'magma' - use MAGMA
  - 'all' - compute with all other free algorithms, check that the answers agree, and return the common answer.

Note: If the curve is loaded from the large Cremona database, then the modular degree is taken from the database.

Of the three above, probably Rubinstein's is the most efficient (in some limited testing I've done).

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='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:

```
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...)
```

TESTS:

When the input is horrendous, some of the algorithms just bomb out with a RuntimeError:

```
sage: EllipticCurve([1234567,89101112]).analytic_rank(algorithm='rubinstein')
Traceback (most recent call last):
...
RuntimeError: unable to compute analytic rank using rubinstein algorithm ('unable to convert
sage: EllipticCurve([1234567,89101112]).analytic_rank(algorithm='sympow')
Traceback (most recent call last):
...
RuntimeError: failed to compute analytic rank
```

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*)

Returns 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 (int or None) – parameter controlling the attempted conversion of real numbers to rationals. If None, 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()
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])
<type 'sage.rings.integer.Integer'>
sage: type(e.aplist(13, python_ints=True)[0])
<type 'int'>
```

cm_discriminant()

Returns the associated quadratic discriminant if this elliptic curve has Complex Multiplication.

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()
-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'*)

Returns the conductor of the elliptic curve.

INPUT:

- `algorithm` - str, (default: "pari")
  - `"pari"` - use the PARI C-library 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.

EXAMPLE:

```
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 compliment 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 $ord_p(c_f/m_f) \leq ord_p(N)/2$.

TESTS:

```
sage: E = EllipticCurve('11a')
sage: E.congruence_number()
1
```

cremona_label(*space=False*)

Return the Cremona label associated to (the minimal model) of this curve, if it is known. If not, raise a RuntimeError 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):
...
RuntimeError: Cremona label not known for Elliptic Curve defined by y^2 + y = x^3 - 79*x + 3
```

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 `RuntimeError` 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):
...
RuntimeError: no database entry for Elliptic Curve defined by y^2 + 8*x*y + 21*y = x^3 + 13*
```

database_curve()

Return the curve in the elliptic curve database isomorphic to this curve, if possible. Otherwise raise a RuntimeError exception.

Since trac ticket #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*)

Computes the elliptic exponential of a complex number with respect to the elliptic curve.

INPUT:

- `z` (complex) – 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 : 1.0000000000000000000000
sage: z = E([1,-2]).elliptic_logarithm(precision=201)
sage: E.elliptic_exponential(z)
(1.0000000000000000000000000000000000000000000000000000000000000000 : -2.0000000000000000000000000

sage: E = EllipticCurve('389a')
sage: Q = E([3,5])
sage: E.elliptic_exponential(Q.elliptic_logarithm())
(3.0000000000000000000000000000 : 5.0000000000000000000000000000 : 1.0000000000000000000000
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 : 1.0000000000000000000000
```

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)]]
[-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: 2 * E.elliptic_exponential(z)
(-1.52184235874404 - 0.0581413944316544*I : 0.948655866506124 - 0.0381469928565030*I : 1.000
sage: E.elliptic_exponential(2 * z)
(-1.52184235874404 - 0.0581413944316562*I : 0.948655866506128 - 0.0381469928565034*I : 1.000
```

eval_modular_form(*points*, *prec*)

Evaluate the modular form of this elliptic curve at points in CC

INPUT:

- `points` - a list of points in the half-plane of convergence
- `prec` - precision

OUTPUT: A list of values L(E,s) for s in points

Note: Better examples are welcome.

EXAMPLES:

```
sage: E=EllipticCurve('37a1')
sage: E.eval_modular_form([1.5+I,2.0+I,2.5+I],0.000001)
[0, 0, 0]
```

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
sage: rho.is_irreducible(7)
True
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.

> **Warning:** If the program fails to give a provably correct result, it prints a warning message, but does not raise an exception. Use `gens_certain()` to find out if this warning message was printed.

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
- `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.

OUTPUT:

- **`generators` - list of generators for the Mordell-Weil** group modulo torsion

IMPLEMENTATION: Uses Cremona's mwrank C library.

EXAMPLES:

```
sage: E = EllipticCurve('389a')
sage: E.gens()                  # random output
[(-1 : 1 : 1), (0 : 0 : 1)]
```

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)]
```

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()
Returns True iff this elliptic curve has Complex Multiplication.

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*=[])
Tests if this elliptic curve has good reduction outside S.

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.

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
```

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.

EXAMPLE:

```
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([0, 0, 1, -34874, -2506691])
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)
1.00000?
```

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: P.point_exact()
(0 : 0 : 1)
```

```
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
```

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.

EXAMPLE:

```
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?
```

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 [Gross-Zagier, 1986, page 311], then that conjecture is false, as the following example shows:

```
sage: E = EllipticCurve('65a')                                  # long time
sage: E.heegner_sha_an(-56)                                     # long time
1.00000000000000
sage: E.torsion_order()                                         # long time
2
sage: E.tamagawa_product()                                      # long time
1
sage: E.quadratic_twist(-56).rank()                             # long time
2
```

height (*precision=None*)

Returns 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*)

Computes all integral points (up to sign) on this elliptic curve.

INPUT:

- `mw_base` - list of EllipticCurvePoint generating the Mordell-Weil group of E (default: 'auto' - calls self.gens())
- `both_signs` - True/False (default False): if True the output contains both P and -P, otherwise only one of each pair.
- `verbose` - True/False (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 : 0 : 1), (-2 : 3 : 1), (-1 : 3 : 1), (0 : 2 : 1), (1 : 0 : 1), (2 : 0 : 1), (3 : 3 : 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 eigenvalue of height pairing matrix:  0.63792081458500...
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
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 :
```

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
```

TESTS:

The bug reported on trac #4525 is now fixed:

```
sage: EllipticCurve('91b1').integral_points()
[(-1 : 3 : 1), (1 : 0 : 1), (3 : 4 : 1)]
```

```
sage: [len(e.integral_points(both_signs=False)) for e in cremona_curves([11..100])]  # long
[2, 0, 2, 3, 2, 1, 3, 0, 2, 4, 2, 4, 3, 0, 0, 1, 2, 1, 2, 0, 2, 1, 0, 1, 3, 3, 1, 1, 4, 2, 3
```

The bug reported at #4897 is now fixed:

```
sage: [P[0] for P in EllipticCurve([0,0,0,-468,2592]).integral_points()]
[-24, -18, -14, -6, -3, 4, 6, 18, 21, 24, 36, 46, 102, 168, 186, 381, 1476, 2034, 67246]
```

Note: This function uses the algorithm given in [Co1].

REFERENCES:

•[Co1] Cohen H., Number Theory Vol I: Tools and Diophantine Equations GTM 239, Springer 2007

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_weierstrass_model()

Return a model of the form $y^2 = x^3 + ax + b$ for this curve with $a, b \in \mathbf{Z}$.

Note that this function is deprecated, and that you should use integral_short_weierstrass_model instead as this will be disappearing in the near future.

EXAMPLES:

```
sage: E = EllipticCurve('17a1')
sage: E.integral_weierstrass_model() #random
doctest:...: DeprecationWarning: integral_weierstrass_model is deprecated, use integral_shor
Elliptic Curve defined by y^2  = x^3 - 11*x - 890 over Rational Field
```

integral_x_coords_in_interval(*xmin*, *xmax*)

Returns 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: xlist = list(xset); xlist.sort(); xlist
[-3, -2, -1, 0, 1, 2, 3, 4, 8, 11, 14, 21, 37, 52, 93]
```

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()

Returns 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_irreducible(*p*)

Return True if the mod p representation is irreducible.

Note that this function is deprecated, and that you should use galois_representation().is_irreducible(p) instead as this will be disappearing in the near future.

EXAMPLES:

```
sage: EllipticCurve('20a1').is_irreducible(7) #random
doctest:...: DeprecationWarning: is_irreducible is deprecated, use galois_representation().i
True
```

is_isogenous(*other*, *proof=True*, *maxp=200*)

Returns 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. If True, this test is followed by a rigorous test (which may be more time-consuming).
- maxp (int, 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`.

METHOD:

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 $a_1, a_2, a_3 \in \{0, \pm 1\}$.

TO DO: 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*)

Returns 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 primeOUTPUT: True - if curve is p-minimal

- False - if curve isn't 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_reducible(*p*)

Return True if the mod-p representation attached to E is reducible.

Note that this function is deprecated, and that you should use galois_representation().is_reducible(p) instead as this will be disappearing in the near future.

EXAMPLES:

```
sage: EllipticCurve('20a1').is_reducible(3) #random
doctest:...: DeprecationWarning: is_reducible is deprecated, use galois_representation().is_
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]
```

is_surjective(*p*, *A=1000*)
Returns true if the mod p representation is surjective

Note that this function is deprecated, and that you should use galois_representation().is_surjective(p) instead as this will be disappearing in the near future.

EXAMPLES:

```
sage: EllipticCurve('20a1').is_surjective(7) #random
doctest:...: DeprecationWarning: is_surjective is deprecated, use galois_representation().is
True
```

isogenies_prime_degree(*l=None*)
Returns 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 $\mathbf{Q}$.

EXAMPLES:

```
sage: E = EllipticCurve([45,32])
sage: E.isogenies_prime_degree()
[]
sage: E = EllipticCurve(j = -262537412640768000)
sage: E.isogenies_prime_degree()
[Isogeny of degree 163 from Elliptic Curve defined by y^2 + y = x^3 - 2174420*x + 1234136692
sage: E1 = E.quadratic_twist(6584935282)
sage: E1.isogenies_prime_degree()
[Isogeny of degree 163 from Elliptic Curve defined by y^2 = x^3 - 942858359570317979813760800

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 Ratio
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 Ratio
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*)

Returns the $\mathbf{Q}$-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
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 Fiel
```

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 Fie
```

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()
184330929667120636533304 96
```

```
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 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*)

Returns the minimal degree of an isogeny between self and other.

INPUT:

- `other` – another elliptic curve.

OUTPUT:

(int) 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')
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.

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 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)
```

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*)

Returns 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() # imaginary parts approx. 0
(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 RuntimeError 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):
...
RuntimeError: Cremona label not known for Elliptic Curve defined by y^2 + y = x^3 - 79*x + 3
```

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()

Returns 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()
Complex L-series of the Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
```

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 Neron 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 databases; 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 #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 elliptic curve.

INPUT:

- `p` - prime (= 5) 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 +
```

```
[    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*7^11 + 6*7^12 + 5
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
```

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()

Determines a quadratic twist with minimal conductor. Returns 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
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 trac ticket #14060):

```
sage: E = cremona_optimal_curves([2*3*5*7*11]).next()
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
```

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)
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
```

Note: If you just want the q-expansion, use `q_expansion()`.

---

modular_parametrization()

Returns 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 +
sage: z = 0.1 + 0.2j
sage: phi(z)
(8.20822465478531 - 13.1562816054682*I : -8.79855099049364 + 69.4006129342200*I : 1.0000000
```

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 : 1.0000000
```

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 + 2
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
```

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*, *use_eclib=False*, *normalize='L_ratio'*)

Return the modular symbol associated to this elliptic curve, with given sign and base ring. This is the map that sends r/s to a fixed multiple of the integral of $2\pi i f(z)dz$ from ∞ to r/s, normalized so that all values of this map take values in $\mathbf{Q}$.

The normalization is such that for sign +1, the value at the cusp 0 is equal to the quotient of $L(E,1)$ by the least positive period of 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.

INPUT:

- `sign` - 1 (default) or -1
- `use_eclib` - (default: False); if True the computation is done with John Cremona's implementation of modular symbols in `eclib`
- `normalize` - (default: 'L_ratio'); either 'L_ratio', 'period', or 'none'; For 'L_ratio', the modular symbol tries to normalized correctly as explained above by comparing it to `L_ratio` for the curve

and some small twists. The normalization 'period' is only available if `use_eclib=False`. It 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 if `use_eclib=False`, though evaluation of it after computing it won't be any faster.

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
sage: M(1/2)
0
sage: M(1/5)
1

sage: E=EllipticCurve('121b1')
sage: M=E.modular_symbol()
Warning : Could not normalize the modular symbols, maybe all further results will be multipl
sage: M(1/7)
-1/2

sage: E=EllipticCurve('11a1')
sage: E.modular_symbol()(0)
1/5
sage: E=EllipticCurve('11a2')
sage: E.modular_symbol()(0)
1
sage: E=EllipticCurve('11a3')
sage: E.modular_symbol()(0)
1/25

sage: E=EllipticCurve('11a2')
sage: E.modular_symbol(use_eclib=True, normalize='L_ratio')(0)
1
sage: E.modular_symbol(use_eclib=True, normalize='none')(0)
2/5
sage: E.modular_symbol(use_eclib=True, normalize='period')(0)
Traceback (most recent call last):
...
ValueError: no normalization 'period' known for modular symbols using John Cremona's eclib
sage: E.modular_symbol(use_eclib=False, normalize='L_ratio')(0)
1
sage: E.modular_symbol(use_eclib=False, normalize='none')(0)
1
sage: E.modular_symbol(use_eclib=False, normalize='period')(0)
1

sage: E=EllipticCurve('11a3')
sage: E.modular_symbol(use_eclib=True, normalize='L_ratio')(0)
1/25
sage: E.modular_symbol(use_eclib=True, normalize='none')(0)
2/5
sage: E.modular_symbol(use_eclib=True, normalize='period')(0)
Traceback (most recent call last):
...
ValueError: no normalization 'period' known for modular symbols using John Cremona's eclib
sage: E.modular_symbol(use_eclib=False, normalize='L_ratio')(0)
1/25
```

```
sage: E.modular_symbol(use_eclib=False, normalize='none')(0)
1
sage: E.modular_symbol(use_eclib=False, normalize='period')(0)
1/25
```

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(
sage: f.modular_symbol_space(-1)
Modular Symbols subspace of dimension 1 of Modular Symbols space of dimension 2 for Gamma_0(
sage: f.modular_symbol_space(0, bound=3)
Modular Symbols subspace of dimension 2 of Modular Symbols space of dimension 5 for Gamma_0(
```

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:-25625626798
Regulator = 95.980371987964
```

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.mwrank.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 :meth:'.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
```

TO DO: This example should not cause a run-time error.

```
sage: E=EllipticCurve([0,0,0,877,0])
sage: # E.ngens()  ######## causes run-time error

sage: print E.mwrank('-v0 -b12 -l')
Curve [0,0,0,877,0] :   Rank = 1
Generator 1 is [29604565304828237474403861024284371796799791624792913256602210:-25625626798
Regulator = 95.980...
```

non_surjective(*A=1000*)

Returns a list of primes p for which the Galois representation mod p is not surjective.

Note that this function is deprecated, and that you should use galois_representation().non_surjective() instead as this will be disappearing in the near future.

EXAMPLES:

```
sage: EllipticCurve('20a1').non_surjective() #random
doctest:...: DeprecationWarning: non_surjective is deprecated, use galois_representation().n
[2,3]
```

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):

```
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*
```

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*
```

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*
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*
```

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*)

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 semi-stable reduction
- `prec` - integer = 1, 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 [Mazur-Tate-Teitelbaum].

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 +
```

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   # long time
```

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
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.

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.

TODO: - remove restriction that curve must be in minimal Weierstrass form. This is currently required for E.gens().

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
[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)
```

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 +
[      4 + 4*5^2 + 2*5^3 + O(5^4)   3 + 4*5 + 3*5^2 + 5^3 + O(5^4)                 2 + 4*5 +
[      3*5 + 4*5^2 + 5^3 + O(5^4)                 2 + 4*5 + O(5^4)     1 + 3*5 + 5^2 + 5^3 +
```

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, 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 [Mazur-Tate-Teitelbaum].

AUTHORS:

- David Harvey (2008-01): based on the padic_height() function, using the algorithm of"Computing p-adic heights via point multiplication"

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 +
```

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   # long time
```

padic_lseries(*p*, *normalize='L_ratio'*, *use_eclib=True*)

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
- use_eclib - bool (default:True); whether or not to use John Cremona's eclib for the computation of modular symbols
- normalize - 'L_ratio' (default), 'period' or 'none'; this is describes the way the modular symbols are normalized. See modular_symbol for more details.

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,use_eclib=False)
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
```

padic_regulator(*p*, *prec=20*, *height=None*, *check_hypotheses=True*)

Computes the cyclotomic p-adic regulator of this curve.

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.

TODO: - remove restriction that curve must be in minimal Weierstrass form. This is currently required for E.gens().

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 +
```

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   # long time
```

A case where the generator belongs to the formal group already (trac #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
```

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, 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, 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), which is basically an optimised version of the algorithm from "p-adic Heights and Log Convergence" (Mazur, Stein, Tate).

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 +
```

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 +
```

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)
```

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
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 +
sage: f(2*t)/2 -g
O(t^11)
```

Test that it returns consistent results over a range of precision:

```
sage: max_N = 30   # get up to at least p^2          # long time
sage: E = EllipticCurve([1, 1, 1, 1, 1])             # long time
sage: p = 5                                          # long time
sage: E2 = E.padic_E2(5, max_N)                      # long time
sage: max_sigma = E.padic_sigma(p, max_N, E2=E2)     # long time
sage: for N in range(3, max_N):                      # long time
...       sigma = E.padic_sigma(p, N, E2=E2)         # long time
...       assert sigma == max_sigma
```

padic_sigma_truncated(*p*, *N=20*, *lamb=0*, *E2=None*, *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, 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, 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" (David Harvey, to appear in LMS JCM), which is basically an optimised version of the algorithm from "p-adic Heights and Log Convergence" (Mazur, Stein, Tate), and "Computing p-adic heights via point multiplication" (David Harvey, still draft form).

Running time is soft-$O(N^2\lambda^{-1}\log p)$, plus whatever time is necessary to compute E_2.

AUTHOR:

- 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)
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 +
```

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 +
```

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):                            # long time
```

```
...          correct = E.padic_sigma(5, N + 3*lamb, E2=E2)            # long time
...          compare = E.padic_sigma_truncated(5, N=N, lamb=lamb, E2=E2)    # long time
...          assert compare == correct                                # long time
```

pari_curve(*prec=None*, *factor=1*)

Return the PARI curve corresponding to this elliptic curve.

INPUT:

- `prec` - The precision of quantities calculated for the returned curve, in bits. If None, defaults to factor multiplied by the precision of the largest cached curve (or a small default precision depending on the curve if none yet computed).
- `factor` - The factor by which to increase the precision over the maximum previously computed precision. Only used if prec (which gives an explicit precision) is None.

EXAMPLES:

```
sage: E = EllipticCurve([0, 0, 1, -1, 0])
sage: e = E.pari_curve()
sage: type(e)
<type 'sage.libs.pari.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(prec=100)
sage: 100 in E._pari_curve
True
sage: e.type()
't_VEC'
sage: e[:5]
[0, 0, 0, 1/3, 2/3]
```

This shows that the bug uncovered by trac:3954 is fixed:

```
sage: 100 in E._pari_curve
True

sage: E = EllipticCurve('37a1')
sage: _ = E.__dict__.pop('_pari_curve')  # clear cached data
sage: Epari = E.pari_curve()
sage: Epari[14].python().prec()
64
sage: [a.precision() for a in Epari]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 4, 4, 4, 4, 4]  # 32-bit
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 3, 3, 3, 3]  # 64-bit
sage: Epari = E.pari_curve(factor=2)
sage: Epari[14].python().prec()
128
```

This shows that the bug uncovered by trac‘4715‘ is fixed:

```
sage: Ep = EllipticCurve('903b3').pari_curve()
```

When the curve coefficients are large, a larger precision is required (see trac ticket #13163):

```
sage: E = EllipticCurve([438269645756479469160344233878810649 7, 28, 3992, 16777216, 298])
sage: E.pari_curve(prec=64)
Traceback (most recent call last):
...
PariError: precision too low in ellinit
sage: E.pari_curve()  # automatically choose the right precision
[438269645756479469160344233878810649 7, 28, 3992, 16777216, 298, ...]
sage: E.minimal_model()
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 - 768642393408379739067598116922917190 76
```

pari_mincurve(*prec=None*, *factor=1*)

Return the PARI curve corresponding to a minimal model for this elliptic curve.

INPUT:

- `prec` - The precision of quantities calculated for the returned curve, in bits. If None, defaults to factor multiplied by the precision of the largest cached curve (or the default real precision if none yet computed).
- `factor` - The factor by which to increase the precision over the maximum previously computed precision. Only used if prec (which gives an explicit precision) is None.

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]
```

period_lattice(*embedding=None*)

Returns 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 $\mathbf{Q}$ into $\mathbf{R}$).

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 (bool)` - (default: False)

  If True, report on each point as found together with linear relations between the points found and the saturation process.

If False, just return the result.

- `rank_bound (bool)` - (default: None)

  If provided, stop searching for points 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 library.

EXAMPLES:

```
sage: E=EllipticCurve('389a1')
sage: E.point_search(5, verbose=False)
[(-1 : 1 : 1), (-3/4 : 7/8 : 1)]
```

Increasing the height_limit takes longer, but finds no more points:

```
sage: E.point_search(10, verbose=False)
[(-1 : 1 : 1), (-3/4 : 7/8 : 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), (-3/4 : 7/8 : 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)]
```

prove_BSD(*E*, *verbosity=0*, *two_desc='mwrank'*, *proof=None*, *secs_hi=5*, *return_BSD=False*)

Attempts 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'`, `'simon'`, `'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:

EXAMPLES:

```
sage: EllipticCurve('11a').prove_BSD(verbosity=2)
p = 2: True by 2-descent...
True for p not in {2, 5} by Kolyvagin.
True for p=5 by Mazur
[]

sage: EllipticCurve('14a').prove_BSD(verbosity=2)
p = 2: True by 2-descent
True for p not in {2, 3} by Kolyvagin.
Remaining primes:
p = 3: reducible, not surjective, good ordinary, divides a Tamagawa number
    (no bounds found)
    ord_p(#Sha_an) = 0
[3]
sage: EllipticCurve('14a').prove_BSD(two_desc='simon')
[3]
```

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.
True for p=3 by Mazur
[]

sage: E = EllipticCurve('37a')
sage: E.rank()
1
sage: E._EllipticCurve_rational_field__rank
{True: 1}
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
```

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 Fie
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.
Remaining primes:
p = 5: reducible, not surjective, good ordinary, divides a Tamagawa number
    (no bounds found)
    ord_p(#Sha_an) = 0
[5]
```

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
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
[7]
```

```
sage: B.heegner_indexes
{-23: 2}
```

TESTS:

This was fixed by trac #8184 and #7575:

```
sage: EllipticCurve('438e1').prove_BSD(verbosity=1)
p = 2: True by 2-descent...
True for p not in {2} by Kolyvagin.
[]

sage: E = EllipticCurve('960d1')
sage: E.prove_BSD(verbosity=1)  # long time (4s on sage.math, 2011)
p = 2: True by 2-descent
True for p not in {2} by Kolyvagin.
[]
```

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
```

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
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=False*, *verbose=False*, *only_use_mwrank=True*, *algorithm='mwrank_lib'*, *proof=None*)

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 (bool)` - (default: False), if True, try to look up the regulator in the Cremona database.
- `verbose` - (default: None), if specified changes the verbosity of mwrank computations. algorithm -
- – `'mwrank_shell'` - call mwrank shell command
- – `'mwrank_lib'` - call mwrank c library
- `only_use_mwrank` - (default: True) if False try using analytic rank methods first.
- `proof` - bool or None (default: None, see proof.elliptic_curve or sage.structure.proof). Note that results obtained from databases are considered proof = True

OUTPUT:

- `rank (int)` - the rank of the elliptic curve.

IMPLEMENTATION: Uses L-functions, mwrank, 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 behavio
4
sage: EllipticCurve([0, 0, 1, -79, 342]).rank(proof=False)
5
sage: EllipticCurve([0, 0, 1, -79, 342]).simon_two_descent()[0]  # long time (7s on sage.mat
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:

```
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.
```

rank_bound()

Upper bound on the rank of the curve, computed using 2-descent. In many cases, this is the actual rank of the curve. If the curve has no 2-torsion it is the same as the 2-selmer rank.

EXAMPLE: 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.rank_bound()
0
```

It gives 0 instead of 2, because it knows Sha is nontrivial. In contrast, for the curve 571A, also with rank 0 and Sha of order 4, we get a worse bound:

```
sage: E = EllipticCurve([0, -1, 1, -929, -10595])
sage: E.rank_bound()
2
sage: E.rank(only_use_mwrank=False)    # uses L-function
0
```

real_components()

Returns 1 if there is 1 real component and 2 if there are 2.

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
```

reducible_primes()

Returns a list of reducible primes.

Note that this function is deprecated, and that you should use galois_representation().reducible_primes() instead as this will be disappearing in the near future.

EXAMPLES:

```
sage: EllipticCurve('20a1').reducible_primes() #random
doctest:...: DeprecationWarning: reducible_primes is deprecated, use galois_representation()
[2,3]
```

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 GF(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(*use_database=True*, *proof=None*, *precision=None*, *descent_second_limit=12*, *verbose=False*)

Returns the regulator of this curve, which must be defined over Q.

INPUT:

- `use_database` - bool (default: False), if True, try to look up the generators in the Cremona database.
- `proof` - bool or None (default: None, see proof.[tab] or sage.structure.proof). Note that results from databases are considered proof = True
- `precision` - int or None (default: None): the precision in bits of the result (default real precision if None)
- `descent_second_limit` - (default: 12)- used in 2-descent
- `verbose` - whether to print mwrank's verbose output

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.mat
14.790527570131...
```

root_number(*p=None*)

Returns 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, default (None); 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())
<type '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=0*, *odd_primes_only=False*)

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: 0), saturation is performed for all primes up to max_prime. If max_prime==0, perform saturation at *all* primes, i.e., compute the true saturation.
- `odd_primes_only (bool)` - only do saturation at odd 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 `mwrank` package. With `max_prime=0`, we call `mwrank` with successively larger prime bounds until the full saturation is provably found. The results of saturation at the previous primes is stored in each case, so this should be reasonably fast.

EXAMPLES:

```
sage: E=EllipticCurve('37a1')
sage: P=E(0,0)
sage: Q=5*P; Q
(1/4 : -5/8 : 1)
sage: E.saturation([Q])
([(0 : 0 : 1)], 5, 0.0511114082399688)
```

TESTS:

See trac ticket #10590. This example would loop forever at default precision:

```
sage: E = EllipticCurve([1, 0, 1, -977842, -372252745])
sage: P = E([-19212812585867619458571882166754266082232352862627 3/3369955684303192766951066
sage: P.height()
113.302910926080
sage: E.saturation([P])
([(-19212812585867619458571882166754266082232352862627 3/3369955684303192766951066021742834 79
sage: (Q,), ind, reg = E.saturation([2*P])  # needs higher precision, handled by eclib
sage: 2*Q == 2*P
True
sage: ind
2
sage: reg
113.302910926080
```

See trac ticket #10840. This used to cause eclib to crash since the curve is non-minimal at 2:

```
sage: E = EllipticCurve([0,0,0,-13711473216,0])
sage: P = E([-19992,16313472])
sage: Q = E([-24108,-17791704])
sage: R = E([-97104,-20391840])
sage: S = E([-113288,-9969344])
sage: E.saturation([P,Q,R,S])
([(-19992 : 16313472 : 1), (-24108 : -17791704 : 1), (-97104 : -20391840 : 1), (-113288 : -9
```

selmer_rank()

The rank of the 2-Selmer group of the curve.

EXAMPLE: 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
```

Here the Selmer rank is equal to the 2-torsion rank (=1) plus the 2-rank of Sha (=2), and the rank itself is zero:

```
sage: E.rank()
0
```

In contrast, for the curve 571A, also with rank 0 and Sha of order 4, we get a worse bound:

```
sage: E = EllipticCurve([0, -1, 1, -929, -10595])
sage: E.selmer_rank()
2
sage: E.rank_bound()
```

```
2
```

To establish that the rank is in fact 0 in this case, we would need to carry out a higher descent:

```
sage: E.three_selmer_rank() # optional: magma
0
```

Or use the L-function to compute the analytic rank:

```
sage: E.rank(only_use_mwrank=False)
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 Fie
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` –
  - 'default' (default) – compute using a Python implementation in Sage
  - 'mwrank' – use a C++ implementation in the mwrank library

NOTES:

- 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.

INPUT:

- `self` – an elliptic curve E over $\mathbf{Q}$
- `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()
(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, [(7 : 11 : 1), (-1 : 20 : 1), (0 : 18 : 1), (3 : 11 : 1), (-3 : 23 : 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
(7, 7)
sage: E = EllipticCurve([0, 0, 1, -23737, 960366])
sage: r, s, G = E.simon_two_descent(); r,s
(8, 8)
```

Example from :trac: 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 :trac: 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 trac ticket #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 $C_2 imes C_2$.

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 #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()

Returns the product of the Tamagawa numbers.

EXAMPLES:

```
sage: E = EllipticCurve('54a')
sage: E.tamagawa_product ()
3
```

tate_curve(p)

Creates the Tate Curve over the p-adics associated to this elliptic curves.

This Tate curve 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 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 + 6
```

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 + 6
```

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 secor
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())
<type 'sage.rings.integer.Integer'>
sage: e = EllipticCurve([1,2,3,4,5])
sage: e.torsion_order()
1
sage: type(e.torsion_order())
<type 'sage.rings.integer.Integer'>
```

torsion_points(*algorithm='pari'*)

Returns the torsion points of this elliptic curve as a sorted list.

INPUT:

- `algorithm` - string:
  - "pari" - (default) use the PARI library

– "doud" - use Doud's algorithm

– "lutz_nagell" - use the Lutz-Nagell theorem

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)]

sage: E=EllipticCurve([-1386747,368636886])
sage: T=E.torsion_subgroup(); T
Torsion Subgroup isomorphic to Z/2 + Z/8 associated to the Elliptic Curve defined by y^2  =
sage: T == E.torsion_subgroup(algorithm="doud")
True
sage: T == E.torsion_subgroup(algorithm="lutz_nagell")
True
sage: E.torsion_points()
[(-1293 : 0 : 1),
(-933 : -29160 : 1),
(-933 : 29160 : 1),
(-285 : -27216 : 1),
(-285 : 27216 : 1),
(0 : 1 : 0),
(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)]
```

torsion_subgroup(*algorithm='pari'*)

Returns the torsion subgroup of this elliptic curve.

INPUT:

- `algorithm` - string:
- `"pari"` - (default) use the PARI library
- `"doud"` - use Doud's algorithm
- `"lutz_nagell"` - use the Lutz-Nagell theorem

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
sage: EllipticCurve('37b').torsion_subgroup()
Torsion Subgroup isomorphic to Z/3 associated to the Elliptic Curve defined by y^2 + y = x^3
```

```
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/2 + Z/8 associated to the Elliptic
Curve defined by y^2 = x^3 - 1386747*x + 368636886 over
Rational Field
sage: G.0
(1227 : 22680 : 1)
sage: G.1
(282 : 0 : 1)
sage: list(G)
[(0 : 1 : 0), (1227 : 22680 : 1), (2307 : -97200 : 1), (8787 : 816480 : 1), (1011 : 0 : 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:

Returns `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.

INPUT:

- `self` – an elliptic curve E over $\mathbf{Q}$
- `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()
(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, [(7 : 11 : 1), (-1 : 20 : 1), (0 : 18 : 1), (3 : 11 : 1), (-3 : 23 : 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
(7, 7)
```

```
sage: E = EllipticCurve([0, 0, 1, -23737, 960366])
sage: r, s, G = E.simon_two_descent(); r,s
(8, 8)
```

Example from :trac: 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 :trac: 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 trac ticket #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', '990k
```

sage.schemes.elliptic_curves.ell_rational_field.**integral_points_with_bounded_mw_coeffs**(*E*, *mw_*
N)

Returns 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.

CHAPTER

FIFTEEN

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 : 1/27*a - 4/27 : 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
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/
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: P.point_exact()
(0 : 0 : 1)

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: P.point_exact()
(141/49 : -162/343 : 1)
```

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); z = P.point_exact(); z
(0 : 1 : 1)
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: sage.structure.sage_object.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: s = E.heegner_point(-7,5).ring_class_field().galois_group().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: s = E.heegner_point(-7,5).ring_class_field().galois_group().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: sage.schemes.elliptic_curves.heegner.GaloisAutomorphism

The complex conjugation automorphism of a ring class field.

EXAMPLES:

```
sage: conj = heegner_point(37,-7,5).ring_class_field().galois_group().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
```

TESTS:

```
sage: type(conj)
<class 'sage.schemes.elliptic_curves.heegner.GaloisAutomorphismComplexConjugation'>
sage: loads(dumps(conj)) == conj
True
```

order()
EXAMPLES:

```
sage: conj = heegner_point(37,-7,5).ring_class_field().galois_group().complex_conjugation()
sage: conj.order()
2
```

class sage.schemes.elliptic_curves.heegner.**GaloisAutomorphismQuadraticForm**(*parent*, *quadratic_form*, *alpha=None*)
Bases: sage.schemes.elliptic_curves.heegner.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 b
[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 b
[1, -1/2*sqrt_minus_52, 1/2*sqrt_minus_52 + 1, 1/2*sqrt_minus_52 - 1, 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_mi
```

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()
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 $\mathrm{Gal}(K_c/K_1)$.

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
sage: s.quadratic_form()
x^2 + 45*y^2
```

class sage.schemes.elliptic_curves.heegner.**GaloisGroup**(*field*, *base=Rational Field*)

Bases: sage.structure.sage_object.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
```

TESTS:

```
sage: G = heegner_point(37,-7).ring_class_field().galois_group()
sage: loads(dumps(G)) == G
True
sage: type(G)
<class 'sage.schemes.elliptic_curves.heegner.GaloisGroup'>
```

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
sage: G.cardinality()
6
```

```
sage: G.base_field()
Number Field in sqrt_minus_7 with defining polynomial x^2 + 7
sage: G = Kc.galois_group(Kc); G
Galois group of Ring class field extension of QQ[sqrt(-7)] of conductor 5 over Ring class fi
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()
Complex conjugation automorphism of Ring class field extension of QQ[sqrt(-7)] of conductor
```

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^
sage: G.cardinality()
12
sage: K5.quadratic_field().class_number()
2
```

class sage.schemes.elliptic_curves.heegner.**HeegnerPoint**(N, D, c)

Bases: sage.structure.sage_object.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


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
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 in Number Field in sqrt_minus_7 with defining polynomial x^2 + 7
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 in Number Field in sqrt_minus_40 with defining polynomial x^2 + 40
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: sage.schemes.elliptic_curves.heegner.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
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
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: PP = P.numerical_approx() # approximately (0 : 0 : 1)
sage: all([c.abs() < 1e-15 for c in PP.xy()])
True
```

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
```

```
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 : 0.0000000000000000000000
```

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: all([c.abs()< 1e-15 for c in P.numerical_approx().xy()])
True
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()
(-6.6...e-16 + 1.41421356237310*I : 1.00000000000000 - 1.41421356237309*I : 1.0000000000000
```

A rank 2 curve, where all Heegner points of conductor 1 are 0:

```
sage: E = EllipticCurve('389a'); E.rank()
2
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)
(0.675507556926806 + 0.344749649302635*I : -0.377142931401887 + 0.843366227137146*I : 1.0000
sage: P.numerical_approx()
(0.6755075569268... + 0.3447496493026...*I : -0.3771429314018... + 0.8433662271371...*I : 1.
sage: E.heegner_point(-7, 11).numerical_approx()
(0.1795583794118... + 0.02035501750912...*I : -0.5573941377055... + 0.2738940831635...*I : 1
sage: E.heegner_point(-7, 13).numerical_approx()
(1.034302915374... - 3.302744319777...*I : 1.323937875767... + 6.908264226850...*I : 1.00000
```

We find (probably) the definining 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` – book (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(100, 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
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

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 +
```

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(75)
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(300)
x^10 + 282527/52441*x^9 + 27049007420/2750058481*x^8 - 22058564794/2750058481*x^7 - 14005423
```

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(300)
x^6 + 1108754853727159228/7235104880325254 7*x^5 + 888755055511840481 68/19534783176878187 69*x
```

class sage.schemes.elliptic_curves.heegner.**HeegnerPointOnX0N**(*N*, *D*, *c=1*, *f=None*, *check=True*)

Bases: sage.schemes.elliptic_curves.heegner.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
sage: x.quadratic_form()
37*x^2 + 11*x*y + 2*y^2
sage: x.quadratic_order()
Order in Number Field in sqrt_minus_7 with defining polynomial x^2 + 7
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 $\gcd(Q, N/Q) = 1$, returns 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), Heeg
```

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)
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)
```

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 imagqinary 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: sage.structure.sage_object.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)
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: sage.schemes.elliptic_curves.heegner.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.

INPUTS:

- 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: sage.schemes.elliptic_curves.heegner.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
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
```

class sage.schemes.elliptic_curves.heegner.**HeegnerPoints_level_disc_cond**(*N*, *D*, *c=1*)

Bases: sage.schemes.elliptic_curves.heegner.HeegnerPoints_level, sage.schemes.elliptic_curves.heegner.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
sage: H.ring_class_field()
Ring class field extension of QQ[sqrt(-7)] of conductor 5

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')
sage: heegner_points(53,-7,15).plot(pointsize=50, rgbcolor='purple')
```

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), ...,
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 - 925/155

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 - 1409/15
```

The result is cached and is a tuple (since it is immutable):

```
sage: H.points() is H.points()
True
sage: type(H.points())
<type '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: sage.structure.sage_object.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).iteritems())
[((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 clas
```

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
```

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:

- 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:

- 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:

- 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:

- list of 2-tuples

EXAMPLES:

```
sage: N = 37; D = -7; ell = 17; c=5
sage: H = heegner_points(N).reduce_mod(ell)
sage: B = H.brandt_module(); I = B.right_ideals()[32]
sage: f = H.optimal_embeddings(D, 1, I.left_order())[0]
sage: g = H.kolyvagin_generators(f.domain().number_field(), c)
sage: alpha_quaternion = f(g[0]); alpha_quaternion
1 - 5/128*i - 77/192*j + 137/384*k
sage: H.kolyvagin_cyclic_subideals(I, 5, alpha_quaternion)
[(Fractional ideal (2 + 874/3*j + 128356/3*k, 2*i + 932/3*j + 198806/3*k, 2560/3*j + 33280/3
```

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: B = H.brandt_module(); I = B.right_ideals()[32]
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: B = H.brandt_module(); I = B.right_ideals()[32]
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 chose 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))) for b in V.basis()]  # long time
[0, 0]
sage: [b.dot_product(k118.element().change_ring(GF(3))) for b in V.basis()]  # long time
[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))) for b in V.basis()]  # long time
[1, 0]
sage: [b.dot_product(k104.element().change_ring(GF(3))) for b in V.basis()]  # long time
[2, 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()
[32, 51]
sage: k32 = H.kolyvagin_sigma_operator(D, c, 32); k32
(63, 68, 47, 47, 31, 52, 37, 0, 0, 47, 3, 31, 47, 7, 21, 26, 19, 10, 0, 0, 11, 28, 41, 2, 47
sage: k51 = H.kolyvagin_sigma_operator(D, c, 51); k51
(5, 13, 0, 0, 14, 0, 21, 0, 0, 0, 29, 0, 0, 45, 0, 6, 0, 40, 0, 61, 0, 0, 40, 32, 0, 9, 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(k32.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
sage: k                                                  # long time
(494, 472, 1923, 1067, ..., 102, 926)
sage: [b.dot_product(k.element().change_ring(GF(3))) for b in V.basis()]    # long time (but
[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
 Order of Quaternion Algebra (-1, -3) with base ring Rational Field with basis (1/2 + 1/2*j
```

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:

- 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
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 disriminant
- 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
```

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)
(14, 16, 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()]
[1, 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:

- bool

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: sage.structure.sage_object.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)[0]; 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)[0].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 +
```

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)[0]
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 in Number Field in a with defining polynomial x^2 + 7
```

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)[0]; 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: sage.structure.sage_object.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()
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: sage.schemes.elliptic_curves.heegner.KolyvaginCohomologyClass

EXAMPLES:

class sage.schemes.elliptic_curves.heegner.**KolyvaginPoint**(*heegner_point*)

Bases: sage.schemes.elliptic_curves.heegner.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)
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
```

TESTS:

```
sage: y = EllipticCurve('37a1').heegner_point(-7)
sage: type(y)
<class 'sage.schemes.elliptic_curves.heegner.HeegnerPointOnEllipticCurve'>
sage: loads(dumps(y)) == y
True
```

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 [SteinToward], 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)
```

REFERENCES:

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()
```

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(); PP
(6.00000000000000 ... : -15.0000000000000 ... : 1.00000000000000)
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)
(1.61355529131985731272300 : -2.18446840788880232891870 : 1.00000000000000000000000)
```

class sage.schemes.elliptic_curves.heegner.**RingClassField**(*D*, *c*, *check=True*)

Bases: sage.structure.sage_object.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
```

TESTS:

```
sage: K_c = heegner_point(37,-7).ring_class_field()
sage: type(K_c)
<class 'sage.schemes.elliptic_curves.heegner.RingClassField'>
sage: loads(dumps(K_c)) == K_c
True
```

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 trac ticket #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())
<type '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
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 f
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
```

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.

EXAMPLE:

```
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: P.point_exact()
(0 : 0 : 1)
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([0, 0, 1, -34874, -2506691])
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)
1.00000?
```

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, optional, 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.

EXAMPLE:

```
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 [Gross-Zagier, 1986, page 311], then that conjecture is false, as the following example shows:

```
sage: E = EllipticCurve('65a')                                  # long time
sage: E.heegner_sha_an(-56)                                     # long time
1.00000000000000
sage: E.torsion_order()                                         # long time
2
sage: E.tamagawa_product()                                      # long time
1
sage: E.quadratic_twist(-56).rank()                             # long time
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*)
Returns 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() # imaginary parts approx. 0
(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 $P_{23\cdot29\cdot41}$, 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*)

make_monic returns a monic integral polynomial g and an integer d such that if α is a root of g then a root of f is α/d.

INPUT:

- f – polynomial over the rational numbers

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: sage.schemes.elliptic_curves.heegner.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 = sage.schemes.elliptic_curves.heegner.make_monic(f)
sage: g
x^3 - 18522*x^2 + 144110421*x - 426000323007
sage: d
9261
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: R.<x> = RR[]
sage: sage.schemes.elliptic_curves.heegner.nearby_rational_poly(2.1*x^2 + 3.5*x - 1.2, max_error
21/10*X^2 + 7/2*X - 6/5
sage: sage.schemes.elliptic_curves.heegner.nearby_rational_poly(2.1*x^2 + 3.5*x - 1.2, max_error
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 in Number Field in a with defining polynomial x^2 + 7, 3*a)
sage: sage.schemes.elliptic_curves.heegner.quadratic_order(-7,3,'alpha')
(Order in Number Field in alpha with defining polynomial x^2 + 7, 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: R.<x> = RR[]
sage: sage.schemes.elliptic_curves.heegner.simplest_rational_poly(2.1*x^2 + 3.5*x - 1.2, 53)
21/10*X^2 + 7/2*X - 6/5
```

CHAPTER

SIXTEEN

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	1	234446
5	1	19047851
6	1	5187563742
7	1	382623908456
8	1	457532830151317

AUTHOR: - William Stein (2007-10-07): initial version

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**

rank(*rank*, *tors=0*, *n=10*, *labels=False*)

Return a list of at most n non-isogenous 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()
((0, 0, 1, -23737, 960366), 457532830151317)
```

CHAPTER

SEVENTEEN

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.

EXAMPLE:

```
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
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 de
```

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: sage.schemes.elliptic_curves.ell_field.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
```

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 poly
```

Check that non-torsion points are remembered when extending the base field (see trac ticket #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)]
```

conductor()

Returns 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, 21796814856932765568243810*a - 134364590724198567128296995, 12
sage: E.conductor()
Fractional ideal (1)
```

An example which used to fail (see trac ticket #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 trac ticket #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)]
```

division_field(*p*, *names*, *map=False*, ***kwds*)

Given an elliptic curve over a number field F and a prime number p, construct the field $F(E[p])$.

INPUT:

- p – a prime number (an element of **Z**)
- names – a variable name for the number field
- map – (default: False) also return an embedding of the base_field() into the resulting field.
- kwds – additional keywords passed to sage.rings.number_field.splitting_field.splitting_fiel

OUTPUT:

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

Warning: This takes a very long time when the degree of the division field is large (e.g. when p is large or when the Galois representation is surjective). The simplify flag also has a big influence on the running time: sometimes simplify=False is faster, sometimes simplify=True (the default) 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: 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 + 2407
```

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

```
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: 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
```

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

```
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+12600
sage: K.<b> = E.division_field(3, simplify_all=True); K
Number Field in b with defining polynomial x^12 - 10*x^10 + 55*x^8 - 60*x^6 + 75*x^4 + 1350*
```

Some higher-degree examples:

```
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
sage: K.<b> = E.division_field(3); K  # long time (3s on sage.math, 2014)
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 (2s on sage.math, 2014)
Torsion Subgroup isomorphic to Z/5 + Z/5 associated to the Elliptic Curve defined by y^2 + y

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 +
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 (4s on sage.math, 2014)
Number Field in b with defining polynomial x^48 ...
sage: K.<b> = E.division_field(7); K  # long time (8s on sage.math, 2014)
Number Field in b with defining polynomial x^72 ...
```

Over a number field:

```
sage: R.<x> = PolynomialRing(QQ)
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
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 ...
```

AUTHORS:

- Jeroen Demeyer (2014-01-06): trac ticket #11905, use `splitting_field` method, moved from `gal_reps.py`, make it work over number fields.

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: 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
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, don't 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: K.<a> = NumberField(x^2 + 23, 'a')
sage: E = EllipticCurve(K, '37')
sage: E == loads(dumps(E))
True
sage: E.gens()
[(0 : 0 : 1), (1/8*a + 5/8 : -3/16*a - 7/16 : 1)]
```

It can happen that no points are found if the height bounds used in the search are too small (see trac ticket #10745):

```
sage: K.<y> = NumberField(x^4 + x^2 - 7)
sage: E = EllipticCurve(K, [1, 0, 5*y^2 + 16, 0, 0])
sage: E.gens(lim1=1, lim3=1)
[]
sage: E.rank(), E.gens()  # long time (about 3 s)
(1, [(-369/25*y^3 + 539/25*y^2 - 1178/25*y + 1718/25 : -27193/125*y^3 + 39683/125*y^2 - 8681
```

Here is a curve of rank 2, yet the list contains many points:

```
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),
(-2*t + 8 : -8*t + 32 : 1),
(1/2*t + 3/2 : -1/2*t - 7/2 : 1),
(-1/8*t - 7/8 : -1/16*t - 23/16 : 1),
(1/8*t - 7/8 : -1/16*t + 23/16 : 1),
(t + 3 : -2*t - 10 : 1),
(2*t + 8 : -8*t - 32 : 1),
(1/2*t + 1/2 : -1/2*t - 1/2 : 1),
(-1/2*t + 3/2 : -1/2*t + 7/2 : 1),
(t + 7 : -4*t - 20 : 1),
(-t + 7 : -4*t + 20 : 1),
(-t + 3 : -2*t + 10 : 1)]
sage: E.rank()
2
```

Test that points of finite order are not included (see trac ticket #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:

Uses Denis Simon's PARI/GP scripts from http://www.math.unicaen.fr/~simon/.

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
```

trac ticket #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
```

trac ticket #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
```

trac ticket #12151:

```
sage: K.<v> = NumberField(x^2 + 161*x - 150)
sage: E = EllipticCurve([25105/216*v - 3839/36, 634768555/7776*v - 98002625/1296, 634768555/
sage: E.global_integral_model()
Elliptic Curve defined by y^2 + (-502639783*v+465618899)*x*y + (-660360421146348939946086 0*v
```

trac ticket #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 - 40
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
```

global_minimal_model(*proof=None*)

Returns a model of self that is integral, minimal at all primes.

Note: This is only implemented for class number 1. In general, such a model may or may not exist.

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.

OUTPUT:

A global integral and minimal model.

EXAMPLES:

```
sage: K.<a> = NumberField(x^2-38)
sage: E = EllipticCurve([0,0,0, 21796814856932765568243810*a - 13436459072419856712829695,

sage: E2 = E.global_minimal_model()
sage: E2 # random (the global minimal model is not unique)
Elliptic Curve defined by y^2 + a*x*y + (a+1)*y = x^3 + (a+1)*x^2 + (36825852020052204680631

sage: E2.local_data()
[]
```

See trac ticket #11347:

```
sage: K.<g> = NumberField(x^2 - x - 1)
sage: E = EllipticCurve(K,[0,0,0,-1/48,161/864]).integral_model().global_minimal_model(); E
Elliptic Curve defined by y^2 + x*y + y = x^3 + x^2 over Number Field in g with defining pol
sage: [(p.norm(), e) for p, e in E.conductor().factor()]
[(9, 1), (5, 1)]
sage: [(p.norm(), e) for p, e in E.discriminant().factor()]
[(-5, 2), (9, 1)]
```

See trac ticket #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 + (v+w+1)*y = x^3 + ((6*w-10)*v+16*w+20) over Number Field in
```

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)]
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_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_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.

EXAMPLE:

```
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 +
```

height_pairing_matrix(*points=None*, *precision=None*)

Returns the height pairing matrix of the given points.

INPUT:

- points - either a list of points, which must be on this curve, or (default) None, in which case self.gens() will be used.
- precision - number of bits of precision of result (default: None, for default RealField precision)

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 -0.63486715783715592064475
[ -1.3095767070865761992624519454   2.7173593928122930896610589220   1.09981843056672921397
[-0.63486715783715592064475542573   1.0998184305667292139777571432  0.66820516565192793503
```

```
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
```

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
```

trac ticket #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
```

trac ticket #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
```

trac ticket #12151:

```
sage: K.<v> = NumberField(x^2 + 161*x - 150)
sage: E = EllipticCurve([25105/216*v - 3839/36, 634768555/7776*v - 98002625/1296, 634768555/
sage: E.global_integral_model()
Elliptic Curve defined by y^2 + (-502639783*v+465618899)*x*y + (-660360421146348939946086
```

trac ticket #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 - 40
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
```

is_global_integral_model()
Return true iff 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_isogenous(*other*, *proof=True*, *maxnorm=100*)
Returns 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 `maxp`. If `True`, the function returns False when the previous condition does not hold, and if it does hold we attempt to see if the curves are indeed isogenous. However, this has not been fully implemented (see examples below), so we may not be able to determine whether or not the curves are 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 (2s on sage.math, 2014)
Traceback (most recent call last):
...
NotImplementedError: Curves appear to be isogenous (same conductor, isogenous modulo all pri
sage: E1.is_isogenous(E2, proof=False)
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)
9
```

TESTS:

Check that trac ticket #15890 is fixed:

```
sage: K.<s> = QuadraticField(229)
sage: c4 = 2173 - 235*(1 - s)/2
sage: c6 = -124369 + 15988*(1 - s)/2
sage: c4c = 2173 - 235*(1 + s)/2
sage: c6c = -124369 + 15988*(1 + s)/2
sage: E = EllipticCurve_from_c4c6(c4, c6)
sage: Ec = EllipticCurve_from_c4c6(c4c, c6c)
sage: E.is_isogenous(Ec)
True
```

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)
False
sage: Emin = E.local_integral_model(P1,P2)
sage: Emin.is_local_integral_model(P1,P2)
True
```

isogeny_degree(*other*)

Returns 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.

Warning: Not all isogenies over number fields are yet implemented. Currently the code only works if there is a chain of isogenies from `self` to `other` of degrees 2, 3, 5, 7 and 13.

EXAMPLES:

```
sage: x = QQ['x'].0
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.isogeny_degree(E2)
2
sage: E2.isogeny_degree(E2)
1
sage: E5 = EllipticCurve('14a5').change_ring(F)
sage: E1.isogeny_degree(E5)
6
```

kodaira_symbol(*P*, *proof=None*)

Returns 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), Fractic
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*)

Returns 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 `qflllgram()` function when applied to a gram matrix which is not positive definite.

EXAMPLES:

Some examples over $\mathbf{Q}$:

```
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,-120039822036992245303534619191166796374,5042249924849106700
sage: xi = [2005024558054813068,            -4690836759490453344,            4700156326649800
sage: points = [E.lift_x(x) for x in xi]
sage: newpoints, U = E.lll_reduce(points)  # long time (35s on sage.math, 2011)
sage: [P[0] for P in newpoints]            # long time
[6823803569166584943, 5949539878899294213, 2005024558054813068, 5864879778877955778, 2395526
```

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 trac ticket #9411):

```
sage: K.<a> = QuadraticField(-23, 'a')
sage: E = EllipticCurve(K, '37')
sage: E.lll_reduce(E.gens())
(
                                          [ 1 -1]
[(0 : 0 : 1), (-2 : -1/2*a - 1/2 : 1)], [ 0  1]
)

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/
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 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 $\mathbf{Q}$. If "pari", use the PARI C-library `ellglobalred` implementation of Tate's algorithm over $\mathbf{Q}$. If "generic", use the general number field implementation.
- `globally` – whether the local algorithm uses global generators for the prime ideals. Default is False, which won't 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 (i - 2):
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 i
Minimal discriminant valuation: 1
Conductor exponent: 1
Kodaira Symbol: I1
Tamagawa Number: 1,
Local data at Fractional ideal (-3*i - 2):
Reduction type: bad split multiplicative
Local minimal model: Elliptic Curve defined by y^2 + (i+1)*x*y + y = x^3 over Number Field i
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 i
Minimal discriminant valuation: 0
Conductor exponent: 0
Kodaira Symbol: I0
Tamagawa Number: 1
```

An example raised in trac ticket #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
```

local_minimal_model(*P*, *proof=None*, *algorithm='pari'*)

Returns 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 $\mathbf{Q}$. If "pari", use the PARI C-library `ellglobalred` implementation of Tate's algorithm over $\mathbf{Q}$. 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)
```

period_lattice(*embedding*)

Returns 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 $\mathbf{R}$ or $\mathbf{C}$.

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
From: Number Field in a with defining polynomial x^3 - 2
To:   Algebraic Field
Defn: a |--> -0.6299605249474365? - 1.091123635971722?*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
From: Number Field in a with defining polynomial x^3 - 2
To:   Algebraic Field
Defn: a |--> -0.6299605249474365? + 1.091123635971722?*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
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, -0.14934463314391922099
```

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, don't 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: 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*)

Returns 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, don't 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: 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/.

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 +
```

```
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 ic
```

regulator_of_points (*points=[ ], precision=None*)

Returns 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)

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)
0.15246017794314375162432475704945582324372707748663081784028098004605322568356246360411481
```

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

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
```

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, don't 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 http://www.math.unicaen.fr/~simon/.

EXAMPLES:

```
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), (1/8*a + 5/8 : -3/16*a - 7/16 : 1)])
sage: E.simon_two_descent(lim1=3, lim3=20, limtriv=5, maxprob=7, limbigprime=10)
(2, 2, [(-1 : 0 : 1), (-1/8*a + 5/8 : -3/16*a - 9/16 : 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

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),
#S(E/K)[2]    = 2
#E(K)/2E(K)   = 2
#Ш(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)]]);
 elliptic curve: Y^2 = x^3 + Mod(1, y^2 + 7)*x + Mod(y, y^2 + 7)
  A = 0
  B = Mod(1, y^2 + 7)
  C = Mod(y, y^2 + 7)
  Computing L(S,2)
  L(S,2) = [Mod(Mod(-1, y^2 + 7)*x^2 + Mod(-1/2*y + 1/2, y^2 + 7)*x + 1, x^3 + Mod(1, y^2 +
  Computing the Selmer group
  #LS2gen = 2
   LS2gen = [Mod(Mod(-5, y^2 + 7)*x^2 + Mod(-3*y, y^2 + 7)*x + Mod(8, y^2 + 7), x^3 + Mod(1,
  Search for trivial points on the curve
 Trivial points on the curve = [[Mod(1/2*y + 3/2, y^2 + 7), Mod(-y - 2, y^2 + 7)], [1, 1, 0]
  zc = Mod(Mod(-5, y^2 + 7)*x^2 + Mod(-3*y, y^2 + 7)*x + Mod(8, y^2 + 7), x^3 + Mod(1, y^2 +
```

```
  Hilbert symbol (Mod(2, y^2 + 7),Mod(-5, y^2 + 7)) =
  zc = Mod(Mod(1, y^2 + 7)*x^2 + Mod(1/2*y - 1/2, y^2 + 7)*x + Mod(-1, y^2 + 7), x^3 + Mod(1
  Hilbert symbol (Mod(-2*y + 2, y^2 + 7),Mod(1, y^2 + 7)) =
  sol of quadratic equation = [1, 0, 1]~
  zc*z1^2 = Mod(Mod(2*y - 2, y^2 + 7)*x + Mod(2*y + 10, y^2 + 7), x^3 + Mod(1, y^2 + 7)*x +
  quartic: (-1/2*y + 1/2)*Y^2 = x^4 + (-3*y - 15)*x^2 + (-8*y - 16)*x + (-11/2*y - 15/2)
  reduced: Y^2 = (-1/2*y + 1/2)*x^4 - 4*x^3 + (-3*y + 3)*x^2 + (2*y - 2)*x + (1/2*y + 3/2)
  not ELS at [2, [0, 1]~, 1, 1, [1, 1]~]
  zc = Mod(Mod(1, y^2 + 7)*x^2 + Mod(1/2*y + 1/2, y^2 + 7)*x + Mod(-1, y^2 + 7), x^3 + Mod(1
  comes from the trivial point [Mod(1/2*y + 3/2, y^2 + 7), Mod(-y - 2, y^2 + 7)]
  m1 = 1
  m2 = 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)]]
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 trac ticket #15483 is fixed:

```
sage: K.<s> = QuadraticField(229)
sage: c4 = 2173 - 235*(1 - s)/2
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, [(0 : 0 : 1), (1/2*zeta43_0^2 + 3/2*zeta43_0 - 2 : -zeta43_0^2 - 4*zeta43_0 + 3 : 1)]
```

tamagawa_exponent(*P*, *proof=None*)

Returns 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*)

Returns 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: 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_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 [Ta2] and [Do] for details.

Note: This definition is slightly different 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 $\mathbb{Q}$ (trac ticket #9413):

```
sage: E = EllipticCurve('30a')
sage: E.tamagawa_product_bsd()
6
```

REFERENCES:

- [Ta2] Tate, John, On the conjectures of Birch and Swinnerton-Dyer and a geometric analog. Seminaire Bourbaki, Vol. 9, Exp. No. 306.
- [Do] Dokchitser, Tim and Vladimir, On the Birch-Swinnerton-Dyer quotients modulo squares, Annals of Math., 2010.

torsion_order()

Returns 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: 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()

Returns 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: 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)
[(16 : 60 : 1),
 (5 : 5 : 1),
 (5 : -6 : 1),
 (16 : -61 : 1),
 (t : 1/11*t^3 + 6/11*t^2 + 19/11*t + 48/11 : 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),
 (-9/121*t^3 - 21/121*t^2 - 127/121*t - 377/121 : -7/121*t^3 + 24/121*t^2 + 197/121*t + 16/1
 (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
 (10/121*t^3 + 49/121*t^2 + 168/121*t + 73/121 : 32/121*t^3 + 60/121*t^2 - 261/121*t - 807/1
 (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),
 (14/121*t^3 - 15/121*t^2 + 90/121*t + 232/121 : 16/121*t^3 - 69/121*t^2 + 293/121*t - 46/12
 (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 + 708/12
 (-26/121*t^3 + 20/121*t^2 - 219/121*t - 995/121 : 15/121*t^3 + 156/121*t^2 - 232/121*t + 27
 (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),
 (-26/121*t^3 + 20/121*t^2 - 219/121*t - 995/121 : -15/121*t^3 - 156/121*t^2 + 232/121*t - 2
 (-5/121*t^3 + 36/121*t^2 - 84/121*t + 24/121 : -34/121*t^3 + 27/121*t^2 - 305/121*t - 829/1
 (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),
 (14/121*t^3 - 15/121*t^2 + 90/121*t + 232/121 : -16/121*t^3 + 69/121*t^2 - 293/121*t - 75/1
 (t : -1/11*t^3 - 6/11*t^2 - 19/11*t - 59/11 : 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 + 686/
 (5/121*t^3 - 14/121*t^2 - 158/121*t - 453/121 : 49/121*t^3 + 129/121*t^2 + 315/121*t + 86/1
 (-9/121*t^3 - 21/121*t^2 - 127/121*t - 377/121 : 7/121*t^3 - 24/121*t^2 - 197/121*t - 137/1
 (-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),
 (0 : 1 : 0)]

sage: E = EllipticCurve('15a1')
sage: K.<t> = NumberField(x^2 + 2*x + 10)
sage: EK = E.base_extend(K)
sage: EK.torsion_points()
[(-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),
 (0 : 1 : 0),
 (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 ()
 [(-2 : -3*i : 1), (-2 : 3*i : 1), (0 : -i : 1), (0 : i : 1), (0 : 1 : 0), (1 : 0 : 1)]
```

torsion_subgroup()

Returns the torsion subgroup of this elliptic curve.

OUTPUT:

(EllipticCurveTorsionSubgroup) The EllipticCurveTorsionSubgroup associated to this elliptic curve.

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
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
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

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

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
```

CHAPTER

EIGHTEEN

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

class sage.schemes.elliptic_curves.ell_finite_field.**EllipticCurve_finite_field**(*K*, *ainvs*)

Bases: sage.schemes.elliptic_curves.ell_field.EllipticCurve_field, sage.schemes.hyperelliptic_curves.hyperelliptic_finite_field.HyperellipticCurve_finite_

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: 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 schemes 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(*debug=False*)

Returns the abelian group structure of the group of points on this elliptic curve.

Warning: The algorithm is definitely *not* intended for use with *large* finite fields! The factorization of the orders of elements must be feasible. Also, baby-step-giant-step methods are used which have space and time requirements which are $O(\sqrt{q})$.

Also, the algorithm uses random points on the curve and hence the generators are likely to differ from one run to another; but the group is cached so the generators will not change in any one run of Sage.

INPUT:

- debug - (default: False): if True, print debugging messages

OUTPUT:

- an abelian group
- tuple of images of each of the generators of the abelian group as points on this curve

AUTHORS:

- John Cremona

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 Cu

sage: E=EllipticCurve(GF(41),[2,5])
sage: E.abelian_group()
Additive abelian group isomorphic to Z/2 + Z/22 ...

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: 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
```

Of course, there are plenty of points if we extend the field:

```
sage: E.cardinality(extension_degree=100)
1267650600228231653296516890625
```

This tests the patch for trac #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 trac #3926 has been fixed:

```
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 Fract
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='pari'*, *extension_degree=1*)

Return the number of points on this elliptic curve.

INPUT:

- `algorithm` – string (default: `'pari'`), used only for point counting over prime fields:
  - `'pari'` – use the baby-step giant-step or Schoof-Elkies-Atkin methods as implemented in the PARI C-library function `ellap`
  - `'bsgs'` – use the baby-step giant-step method as implemented in Sage, with the Cremona-Sutherland version of Mestre's trick
  - `'all'` – compute cardinality with both `'pari'` and `'bsgs'`; return result if they agree or raise a `RuntimeError` 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.

Over prime fields, one of the above algorithms is used. Over non-prime fields, the serious point counting is done on a standard curve with the same j-invariant over the field $\mathbf{F}_p(j)$, then lifted to the base field, and finally account is taken of twists.

For $j = 0$ and $j = 1728$ special formulas are used instead.

EXAMPLES:

```
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: l = [1, 1, 0, 2, 0]
sage: EllipticCurve(k, l).cardinality()
38
```

An even bigger extension (which we check against Magma):

```
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: E = EllipticCurve(GF(3^100, 'a'), [1,2,3,4,5])
sage: E.cardinality() is E.cardinality()
True
sage: E = EllipticCurve(GF(11^2, 'a'), [3,3])
sage: E.cardinality()
128
sage: EllipticCurve(GF(11^100, 'a'), [3,3]).cardinality()
13780612339822270184118337172089636776264331200038467184683526694179151034106556517649784650
```

TESTS:

```
sage: EllipticCurve(GF(10009), [1,2,3,4,5]).cardinality(algorithm='foobar')
Traceback (most recent call last):
...
ValueError: Algorithm is not known
```

If the cardinality has already been computed, then the `algorithm` keyword is ignored:

```
sage: E = EllipticCurve(GF(10007), [1,2,3,4,5])
sage: E.cardinality(algorithm='pari')
10076
sage: E.cardinality(algorithm='foobar')
10076
```

cardinality_bsgs (*verbose=False*)

Return the cardinality of self over the base field. Will be called by user function cardinality only when necessary, i.e. when the j_invariant is not in the prime 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. 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 over the (prime) base field using PARI.

The result is not cached.

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
```

TESTS:

```
sage: K.<a>=GF(3^20)
sage: E=EllipticCurve(K,[1,0,0,1,a])
sage: E.cardinality_pari()
Traceback (most recent call last):
...
ValueError: cardinality_pari() only works over prime fields.
sage: E.cardinality()
3486794310
```

count_points(*n=1*)

Returns 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: 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]
```

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: E=EllipticCurve(GF(25,'a'),[0,0,0,0,1])
sage: E.frobenius()
-5
```

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 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: E=EllipticCurve(GF(25,'a'),[0,0,0,0,1])
sage: R=E.frobenius_order()
sage: R
Order 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: E=EllipticCurve(GF(25,'a'),[0,0,0,0,1])
sage: E.frobenius_polynomial().factor()
(x + 5)^2
```

gens()

Returns a tuple of length up to 2 of points which generate the abelian group of points on this elliptic curve. See abelian_group() for limitations.

The algorithm uses random points on the curve, and hence the generators are likely to differ from one run to another; but they are cached so will be consistent in any one run of Sage.

AUTHORS:

- John Cremona

EXAMPLES:

```
sage: E=EllipticCurve(GF(11),[2,5])
sage: E.gens()                              # random output
((0 : 7 : 1),)
sage: EllipticCurve(GF(41),[2,5]).gens()  # random output
((21 : 1 : 1), (8 : 0 : 1))
sage: F.<a>=GF(3^6,'a')
sage: E=EllipticCurve([a,a+1])
sage: pts=E.gens()
sage: len(pts)
1
sage: pts[0].order()==E.cardinality()
True
```

is_isogenous(*other*, *field=None*, *proof=True*)

Returns 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: 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: 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: 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: 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: 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: E1 = EllipticCurve(GF(13^2,'a'),[2,7]); E1 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 documentaion 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 documentaion 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
```

TESTS:

```
sage: from sage.schemes.elliptic_curves.ell_finite_field import supersingular_j_polynomial,
sage: F = GF(103)
sage: ssjlist = [F(1728)] + supersingular_j_polynomial(103).roots(multiplicities=False)
sage: Set([j for j in F if is_j_supersingular(j)]) == Set(ssjlist)
True
```

order(*algorithm='pari', extension_degree=1*)

Return the number of points on this elliptic curve.

INPUT:

- `algorithm` – string (default: `'pari'`), used only for point counting over prime fields:
  - `'pari'` – use the baby-step giant-step or Schoof-Elkies-Atkin methods as implemented in the PARI C-library function `ellap`

– 'bsgs' – use the baby-step giant-step method as implemented in Sage, with the Cremona-Sutherland version of Mestre's trick

– 'all' – compute cardinality with both 'pari' and 'bsgs'; return result if they agree or raise a RuntimeError 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.

Over prime fields, one of the above algorithms is used. Over non-prime fields, the serious point counting is done on a standard curve with the same j-invariant over the field $\mathbf{F}_p(j)$, then lifted to the base field, and finally account is taken of twists.

For $j = 0$ and $j = 1728$ special formulas are used instead.

EXAMPLES:

```
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: l = [1, 1, 0, 2, 0]
sage: EllipticCurve(k, l).cardinality()
38
```

An even bigger extension (which we check against Magma):

```
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: E = EllipticCurve(GF(3^100, 'a'), [1,2,3,4,5])
sage: E.cardinality() is E.cardinality()
True
sage: E = EllipticCurve(GF(11^2, 'a'), [3,3])
sage: E.cardinality()
128
sage: EllipticCurve(GF(11^100, 'a'), [3,3]).cardinality()
13780612339822270184118337172089636776264331200038467184683526694179151034106556517649784650
```

TESTS:

```
sage: EllipticCurve(GF(10009), [1,2,3,4,5]).cardinality(algorithm='foobar')
Traceback (most recent call last):
```

```
...
ValueError: Algorithm is not known
```

If the cardinality has already been computed, then the `algorithm` keyword is ignored:

```
sage: E = EllipticCurve(GF(10007), [1,2,3,4,5])
sage: E.cardinality(algorithm='pari')
10076
sage: E.cardinality(algorithm='foobar')
10076
```

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))
```

points()

All the 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: a_sub_p = E.change_ring(QQ).ap(p); a_sub_p
2

sage: len(E.points())
4
sage: p + 1 - a_sub_p
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: (p + 1)**2 - a_sub_p**2
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
```

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()

Returns a random point on this elliptic curve.

If q is small, finds all points and returns one at random. Otherwise, returns the point at infinity with

probability $1/(q+1)$ where the base field has cardinality q, and then picks random x-coordinates from the base field until one gives a rational point.

EXAMPLES:

```
sage: k = GF(next_prime(7^5))
sage: E = EllipticCurve(k,[2,4])
sage: P = E.random_element(); P
(16740 : 12486 : 1)
sage: type(P)
<class 'sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_finite_field'>
sage: P in E
True

sage: k.<a> = GF(7^5)
sage: E = EllipticCurve(k,[2,4])
sage: P = E.random_element(); P
(2*a^4 + 3*a^2 + 4*a : 3*a^4 + 6*a^2 + 5 : 1)
sage: type(P)
<class 'sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_finite_field'>
sage: P in E
True

sage: k.<a> = GF(2^5)
sage: E = EllipticCurve(k,[a^2,a,1,a+1,1])
sage: P = E.random_element(); P
(a^4 + a^2 + 1 : a^3 + a : 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: len(set(E.random_element() for _ in range(100)))
16
sage: E.cardinality()
16
```

TESTS:

See trac #8311:

```
sage: E = EllipticCurve(GF(3), [0,0,0,2,2])
sage: E.random_element()
(0 : 1 : 0)
sage: E.cardinality()
1

sage: E = EllipticCurve(GF(2), [0,0,1,1,1])
sage: E.random_point()
(0 : 1 : 0)
sage: E.cardinality()
1

sage: F.<a> = GF(4)
sage: E = EllipticCurve(F, [0, 0, 1, 0, a])
sage: E.random_point()
(0 : 1 : 0)
sage: E.cardinality()
```

```
1
```

random_point()

Returns a random point on this elliptic curve.

If q is small, finds all points and returns one at random. Otherwise, returns the point at infinity with probability $1/(q+1)$ where the base field has cardinality q, and then picks random x-coordinates from the base field until one gives a rational point.

EXAMPLES:

```
sage: k = GF(next_prime(7^5))
sage: E = EllipticCurve(k,[2,4])
sage: P = E.random_element(); P
(16740 : 12486 : 1)
sage: type(P)
<class 'sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_finite_field'>
sage: P in E
True

sage: k.<a> = GF(7^5)
sage: E = EllipticCurve(k,[2,4])
sage: P = E.random_element(); P
(2*a^4 + 3*a^2 + 4*a : 3*a^4 + 6*a^2 + 5 : 1)
sage: type(P)
<class 'sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_finite_field'>
sage: P in E
True

sage: k.<a> = GF(2^5)
sage: E = EllipticCurve(k,[a^2,a,1,a+1,1])
sage: P = E.random_element(); P
(a^4 + a^2 + 1 : a^3 + a : 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: len(set(E.random_element() for _ in range(100)))
16
sage: E.cardinality()
16
```

TESTS:

See trac #8311:

```
sage: E = EllipticCurve(GF(3), [0,0,0,2,2])
sage: E.random_element()
(0 : 1 : 0)
sage: E.cardinality()
1

sage: E = EllipticCurve(GF(2), [0,0,1,1,1])
sage: E.random_point()
(0 : 1 : 0)
sage: E.cardinality()
1
```

```
sage: F.<a> = GF(4)
sage: E = EllipticCurve(F, [0, 0, 1, 0, a])
sage: E.random_point()
(0 : 1 : 0)
sage: E.cardinality()
1
```

rational_points()

All the 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: a_sub_p = E.change_ring(QQ).ap(p); a_sub_p
2

sage: len(E.points())
4
sage: p + 1 - a_sub_p
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: (p + 1)**2 - a_sub_p**2
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
```

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*, *num_checks=8*)

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.
- `num_checks` - Integer (default: 8) number of times to check whether value*(a random point on this curve) is equal to the identity.

OUTPUT:

None

EXAMPLES:

This example illustrates basic usage.

```
sage: E = EllipticCurve(GF(7), [0, 1]) # This curve has order 6
sage: E.set_order(6)
sage: E.order()
6
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 = 1093849038073734274511112390766805569936207598951683748994586394495953116150735016
sage: q = 6864797660130609714981900799081393217269435300143305409394463459185543183397655394
sage: E = EllipticCurve([F(A), F(B)])
sage: E.set_order(q)
sage: G = E.random_point()
sage: E.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 6
sage: E.set_order("hi")
Traceback (most recent call last):
...
ValueError: Value hi illegal (not an integer in the Hasse range)
sage: E.set_order(3.14159)
Traceback (most recent call last):
...
ValueError: Value 3.14159000000000 illegal (not an integer in the Hasse range)
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(7), [0, 1]) # This curve has order 6
sage: E.set_order(11)
Traceback (most recent call last):
...
ValueError: Value 11 illegal (multiple of random point not the identity)
```

However, set_order can be fooled, though it's not likely in "real cases of interest". For instance, the order can be set to a multiple of the actual order:

```
sage: E = EllipticCurve(GF(7), [0, 1]) # This curve has order 6
sage: E.set_order(12)  # 12 just fits in the Hasse range
sage: E.order()
```

```
12
```

Or, the order can be set incorrectly along with num_checks set too small:

```
sage: E = EllipticCurve(GF(7), [0, 1]) # This curve has order 6
sage: E.set_order(4, num_checks=0)
WARNING: No checking done in set_order
sage: E.order()
4
```

The value of num_checks must be an integer. Negative values are interpreted as zero, which means don't do any checking:

```
sage: E = EllipticCurve(GF(7), [0, 1]) # This curve has order 6
sage: E.set_order(4, num_checks=-12)
WARNING: No checking done in set_order
sage: E.order()
4
```

NOTES:

The implementation is based on the fact that orders of elliptic curves are cached in the (pseudo-private) _order slot.

AUTHORS:

- Mariah Lenox (2011-02-16)

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])
sage: E.trace_of_frobenius()
802
```

The following shows that the issue from trac #2849 is fixed:

```
sage: E=EllipticCurve(GF(3^5,'a'),[-1,-1])
sage: E.trace_of_frobenius()
-27
```

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 $\mathrm{GF}(p)$, we check that $(p+1)P = 0$ for several random points P, returning False if any fail: supersingular curves over $\mathrm{GF}(p)$ 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, supersingula
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]), (

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.**supersingular_j_polynomial**(p)

Return a polynomial whose roots are the supersingular j-invariants in characteristic p, other than 0, 1728.

INPUT:

- p (integer) – a prime number.

ALGORITHM:

First compute H(X) whose roots are the Legendre λ-invariants of supersingular curves (Silverman V.4.1(b)) in charactersitic 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.

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]
```

TESTS:

```
sage: supersingular_j_polynomial(6)
Traceback (most recent call last):
...
ValueError: p (=6) should be a prime number
```

CHAPTER

NINETEEN

POINTS ON ELLIPTIC CURVES

The base class `EllipticCurvePoint_field`, derived from `AdditiveGroupElement`, provides support for points on elliptic curves defined over general fields. The derived classes `EllipticCurvePoint_number_field` and `EllipticCurvePoint_finite_field` provide further support for point on curves defined over number fields (including the rational field $\mathbf{Q}$) and over finite fields.

The class `EllipticCurvePoint`, which is based on `SchemeMorphism_point_projective_ring`, currently has little extra functionality.

EXAMPLES:

An example over $\mathbf{Q}$:

```
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: 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: 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: 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 1520944668 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: sage.schemes.projective.projective_point.SchemeMorphism_point_projective_ring

A point on an elliptic curve.

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

Bases: sage.schemes.projective.projective_point.SchemeMorphism_point_abelian_variety_fiel

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: K.<i>=NumberField(x^2+1)
sage: E=EllipticCurve(K,[0,1,0,-160,308])
sage: P=E(26,-120)
sage: Q=E(2+12*i,-36+48*i)
sage: P.order() == Q.order() == 4  # long time (3s)
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))
```

TESTS:

```
sage: loads(S.dumps()) == S
True
sage: E = EllipticCurve('37a')
sage: P = E(0,0); P
(0 : 0 : 1)
sage: loads(P.dumps()) == P
True
sage: T = 100*P
sage: loads(T.dumps()) == T
True
```

Test pickling an elliptic curve that has known points on it:

```
sage: e = EllipticCurve([0, 0, 1, -1, 0]); g = e.gens(); loads(dumps(e)) == e
True
```

Test that the refactoring from trac ticket #14711 did preserve the behaviour of domain and codomain:

```
sage: E=EllipticCurve(QQ,[1,1])
sage: P=E(0,1)
sage: P.domain()
```

```
Spectrum of Rational Field
sage: K.<a>=NumberField(x^2-3,'a')
sage: P=E.base_extend(K)(1,a)
sage: P.domain()
Spectrum of Number Field in a with defining polynomial x^2 - 3
sage: P.codomain()
Elliptic Curve defined by y^2 = x^3 + x + 1 over Number Field in a with defining polynomial x^2
sage: P.codomain() == P.curve()
True
```

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()

EXAMPLE:

```
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: 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: 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: 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 [HSV] end of section 3.1):

```
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: 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.xy()[0]),phi(P.xy()[1]))
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 fi
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 +
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: 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):
...
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: 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)
```

NOTES:

First defined in the paper of [HSV], 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.

REFERENCES:

AUTHORS:

- Mariah Lenox (2011-03-08)

curve()

Return the curve that this point is on.

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
```

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: k.<a> = GF(25)
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)
[(3 : 3*a + 4 : 1)]
```

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

```
sage: P.division_points(2)
[(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)]
[(3 : 3*a + 4 : 1), (3 : 3*a + 4 : 1)]
```

Some other checks:

```
sage: P.division_points(3)
[]
sage: P.division_points(4)
[(0 : 3*a + 2 : 1), (1 : 0 : 1)]
sage: P.division_points(5)
[(1 : 1 : 1)]
```

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

```
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: E(0).division_points(3)
[(0 : 1 : 0), (5 : -10 : 1), (5 : 9 : 1)]
sage: EK(0).division_points(3)
[(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/
```

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])
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
```

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
```

TESTS:

This shows that the bug reported at trac ticket #10076 is fixed:

```
sage: K = QuadraticField(8,'a')
sage: E = EllipticCurve([K(0),0,0,-1,0])
sage: P = E([-1,0])
sage: P.is_divisible_by(2)
False
```

```
sage: P.division_points(2)
[]
```

Note that it is not sufficient to test that `self.division_points(m,poly_only=True)` has roots:

```
sage: P.division_points(2, poly_only=True).roots()
[(1/2*a - 1, 1), (-1/2*a - 1, 1)]

sage: tor = E.torsion_points(); len(tor)
8
sage: [T.order() for T in tor]
[2, 4, 4, 2, 1, 2, 4, 4]
sage: all([T.is_divisible_by(3) for T in tor])
True
sage: Set([T for T in tor if T.is_divisible_by(2)])
{(0 : 1 : 0), (1 : 0 : 1)}
sage: Set([2*T for T in tor])
{(0 : 1 : 0), (1 : 0 : 1)}
```

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()`

EXAMPLE:

```
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))
```

scheme()

Return the scheme of this point, i.e., the curve it is on. This is synonymous with `curve()` which is perhaps more intuitive.

EXAMPLES:

```
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: K.<a>=NumberField(x^2-3,'a')
sage: P=E.base_extend(K)(1,a)
sage: P.scheme()
Elliptic Curve defined by y^2 = x^3 + x + 1 over Number Field in a with defining polynomial
```

set_order(*value*)

Set the value of self._order to value.

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

INPUT:

- `value` - positive Integer

OUTPUT:

`None`

EXAMPLES:

This example illustrates basic usage.

```
sage: E = EllipticCurve(GF(7), [0, 1]) # This curve has order 6
sage: G = E(5, 0)
sage: G.set_order(2)
sage: 2*G
(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 = 109384903807373427451111239076680556993620759895168374899458639449595311615073501
sage: q = 686479766013060971498190079908139321726943530014330540939446345918554318339765539
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
```

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

```
sage: E = EllipticCurve(GF(7), [0, 1]) # This curve has order 6
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 6
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, though it's not likely in "real cases of interest". For instance, the order can be set to a multiple the actual order:

```
sage: E = EllipticCurve(GF(7), [0, 1]) # This curve has order 6
sage: G = E(5, 0)   # G has order 2
sage: G.set_order(8)
sage: G.order()
8
```

NOTES:

The implementation is based of the fact that orders of elliptic curve points are cached in the (pseudo-private) _order slot.

AUTHORS:

- Mariah Lenox (2011-02-16)

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: 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);
sage: # multiply by cofactor so Q has order n:
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: 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: 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.xy()[0]),phi(P.xy()[1]))
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 fi
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
```

NOTES:

This function uses Miller's algorithm, followed by a naive exponentiation. It does not do anything fancy. In the case that there is an issue with Q being on one of the lines generated in the $r * P$ calculation, Q is offset by a random point R and P.tate_pairing(Q+R,n,k)/P.tate_pairing(R,n,k) is returned.

AUTHORS:

- Mariah Lenox (2011-03-07)

weil_pairing(Q, n)

Compute the Weil pairing of self and Q using Miller's algorithm.

INPUT:

- Q – a point on self.curve().
- n – an integer n such that $nP = nQ = (0 : 1 : 0)$ where P = self.

OUTPUT:

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

EXAMPLES:

```
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.xy()[0]),phi(P.xy()[1]))
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^
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)
Traceback (most recent call last):
...
ValueError: points must both be n-torsion
```

A larger example (see trac ticket #4964):

```
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
```

An example over a number field:

```
sage: P,Q = EllipticCurve('11a1').change_ring(CyclotomicField(5)).torsion_subgroup().gens()
sage: P,Q = (P.element(), Q.element())  # long time
sage: (P.order(),Q.order())  # long time
(5, 5)
sage: P.weil_pairing(Q,5)  # long time
zeta5^2
sage: Q.weil_pairing(P,5)  # long time
zeta5^3
```

ALGORITHM:

Implemented using Proposition 8 in [Mil04]. The value 1 is returned for linearly dependent input points. This condition is caught via a DivisionByZeroError, since the use of a discrete logarithm test for linear dependence, is much too slow for large n.

REFERENCES:

AUTHOR:

- David Hansen (2009-01-25)

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
```

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

Bases: sage.schemes.elliptic_curves.ell_point.EllipticCurvePoint_field

Class for elliptic curve points over finite fields.

additive_order()

Return the order of this point on the elliptic curve.

ALGORITHM:

Use generic functions from sage.groups.generic. If the group order is known, use order_from_multiple(), otherwise use order_from_bounds() with the Hasse bounds for the base field. In the latter case, we might find that we have a generator for the group, in which case it is cached.

We do not cause the group order to be calculated when not known, since this function is used in determining the group order via computation of several random points and their orders. The exceptions to this are (1) when the base field is a prime field and efficient SEA-based methods are available for the cardinality, and (2) when finding the group order is possible quickly, currently only implemented for curves with $j = 0$ or $j = 1728$ (see trac ticket #15567).

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

AUTHOR:

- John Cremona, 2008-02-10, adapted 2008-04-05 to use generic functions.

EXAMPLES:

```
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
```

In the next example, the cardinality of E will be computed (using SEA) and cached:

```
sage: p=next_prime(2^150)
sage: E=EllipticCurve(GF(p),[1,1])
sage: P=E(831623307675610677632782670796608848711856078, 42295786042873366706573292533588638
sage: P.order()
1427247692705959881058262545272474300628281448
sage: P.order()==E.cardinality()
True
```

In the next example, the cardinality of E will be computed and cached since $j(E) = 0$:

```
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 = E.random_point()
sage: P.order() # random
16777251
sage: E._order  # as cached
1125904604468004
```

Similarly when $j(E) = 1728$:

```
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
sage: E._order  # as cached
1125902679258240
```

discrete_log(*Q*, *ord=None*)

Returns discrete log of Q with respect to P =self.

INPUT:

- Q (point) – another point on the same curve as self.
- ord (integer or None (default)) – the order of self.

OUTPUT:

(integer) – The discrete log of Q with respect to P, which is an integer m with $0 \leq m < o(P)$ such that $mP = Q$, if one exists. A ValueError is raised if there is no solution.

Note: The order of self is computed if not supplied.

AUTHOR:

- John Cremona. Adapted to use generic functions 2008-04-05.

EXAMPLE:

```
sage: F = GF(3^6,'a')
sage: a = F.gen()
sage: E = EllipticCurve([0,1,1,a,a])
sage: E.cardinality()
762
sage: A = E.abelian_group()
sage: P = A.gen(0).element()
sage: Q = 400*P
sage: P.discrete_log(Q)
400
```

order()

Return the order of this point on the elliptic curve.

ALGORITHM:

Use generic functions from sage.groups.generic. If the group order is known, use order_from_multiple(), otherwise use order_from_bounds() with the Hasse bounds for the base field. In the latter case, we might find that we have a generator for the group, in which case it is cached.

We do not cause the group order to be calculated when not known, since this function is used in determining the group order via computation of several random points and their orders. The exceptions to this are (1) when the base field is a prime field and efficient SEA-based methods are available for the cardinality, and (2) when finding the group order is possible quickly, currently only implemented for curves with $j = 0$ or $j = 1728$ (see trac ticket #15567).

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

AUTHOR:

- John Cremona, 2008-02-10, adapted 2008-04-05 to use generic functions.

EXAMPLES:

```
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
```

In the next example, the cardinality of E will be computed (using SEA) and cached:

```
sage: p=next_prime(2^150)
sage: E=EllipticCurve(GF(p),[1,1])
sage: P=E(831623307675610677632782670796608848711856078, 42295786042873366706573292533588638
sage: P.order()
1427247692705959881058262545272474300628281448
sage: P.order()==E.cardinality()
True
```

In the next example, the cardinality of E will be computed and cached since $j(E) = 0$:

```
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 = E.random_point()
sage: P.order() # random
16777251
sage: E._order  # as cached
1125904604468004
```

Similarly when $j(E) = 1728$:

```
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
```

```
sage: E._order  # as cached
1125902679258240
```

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

Bases: sage.schemes.elliptic_curves.ell_point.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
```

TESTS:

```
sage: loads(S.dumps()) == S
True
sage: P = E(0,0); P
(0 : 0 : 1)
sage: loads(P.dumps()) == P
True
sage: T = 100*P
sage: loads(T.dumps()) == T
True
```

Test pickling an elliptic curve that has known points on it:

```
sage: e = EllipticCurve([0, 0, 1, -1, 0]); g = e.gens(); loads(dumps(e)) == e
True
```

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: 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 poly
sage: P = E.lift_x(2+a); P
(a + 2 : 2*a + 1 : 1)
sage: P.archimedean_local_height(K.places(prec=170)[0]) / 2
0.45754773287523276736211210741423654346576029814695

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

sage: Q = E.lift_x(-9/4); Q
(-9/4 : -27/8*i : 1)
```

```
sage: Q.archimedean_local_height(K.places()[0]) / 2
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: 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
```

TESTS:

See trac ticket #12509:

```
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.archimedean_local_height()
-0.22066079554682784892183362746930
```

archimedian_local_height(**args*, ***kwds*)
Deprecated: Use `archimedean_local_height()` instead. See trac ticket #13951 for details.

elliptic_logarithm(*embedding=None*, *precision=100*, *algorithm='pari'*)
Returns the elliptic logarithm of this elliptic curve point.

An embedding of the base field into $\mathbf{R}$ or $\mathbf{C}$ (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 $\mathbf{R}$ or $\mathbf{C}$
- `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 `ellpointtoz{}`, or 'sage' for a native implementation. Ignored for complex embeddings.

ALGORITHM:

See [Co2] Cohen H., A Course in Computational Algebraic Number Theory GTM 138, Springer 1996 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 [Co2].
- 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 trac ticket #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)
0.00025638725886520225353198932528666427411683880083463700150051421280096109363 73
sage: P.elliptic_logarithm(precision=257)
0.00025638725886520225353198932528666427411683880083463700150051421280096109363730
```

Examples over number fields:

```
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, 1.90726488

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*
sage: [[L.elliptic_logarithm(P) for P in pts] for L in Ls]
[[0.250819591818930 - 0.411963479992219*I, -0.290994550611374 - 1.37239400324105*I, -0.69347

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])
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: 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 (i - 2),
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
```

TESTS:

An example showing that trac ticket #8498 is fixed:

```
sage: E = EllipticCurve('11a1')
sage: K.<t> = NumberField(x^2+47)
sage: EK = E.base_extend(K)
sage: T = EK(5,5)
sage: P = EK(-2, -1/2*t - 1/2)
sage: p = K.ideal(11)
sage: T.has_good_reduction(p)
False
sage: P.has_good_reduction(p)
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 $\mathbf{Q}$, 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 [Cre] and double that of [SilBook].

See Wikipedia article Néron-Tate height

REFERENCES:

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()
+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 Fi
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)
sage: E = EllipticCurve([a, 4]); E
Elliptic Curve defined by y^2 = x^3 + a*x + 4 over Number Field in a with defining polynomia
sage: P = E((0,2))
sage: P.height()
0.810463096585925
sage: P.height(precision=100)
0.81046309658592536863991810577
sage: P.height(precision=200)
0.81046309658592536863991810576865158896130286417155832378086
sage: (2*P).height() / P.height()
4.00000000000000
sage: (100*P).height() / P.height()
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)
sage: EK = E.change_ring(K)
sage: PK = EK([0,0])
sage: PK.height()
0.0511114082399688
sage: PK.height(normalised=False)
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'))
sage: PK = EK([-2,3,1])
sage: PK.height()
1.36857250535393

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 : 1)
sage: Q.height()
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)
sage: ED = E.quadratic_twist(-7)
sage: Q = E.isomorphism_to(ED.change_ring(K))(P); Q
(0 : -7/2*a - 1/2 : 1)
sage: Q.height()
0.0511114082399688
sage: Q.height(precision=100)
0.051111408239968840235886099757
```

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

```
sage: E = EllipticCurve([1, -1, 1, -20637587012466263707737269 78, 32838647793306133075103747
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)
25.860317067546190743868840740735110323098872903844416215577171041783572512955113057088981 32

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

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

```
sage: E = EllipticCurve([-55804723294461149528055058045934980800 00,-157339733785368110382973
sage: xP = 204885147732879546487576840131729064308289385547094673627174585676211859152978311
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 trac ticket #12509 is fixed (precision issues):

```
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)
4.000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
```

This shows that the bug reported at trac ticket #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'))
sage: P2 = F([2,5])
sage: P2.height()
1.06248137652528
```

is_on_identity_component(*embedding=None*)

Returns 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. $\mathbf{Q}$)
- `embedding` – an embedding from the base_field of the point's curve into $\mathbf{R}$ or $\mathbf{C}$; if `None` (the default) it uses the first embedding of the base_field into $\mathbf{R}$ if any, else the first embedding into $\mathbf{C}$.

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 range(-3,5)]
[False, False, False, False, False, True, True, True]
```

An example over a field with two real embeddings:

```
sage: L.<a> = QuadraticField(2)
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: [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: 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 polyn
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

sage: Q = E.lift_x(-9/4); Q
(-9/4 : -27/8*i : 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()
1/2*log(16)
```

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: 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)
```

TESTS:

```
sage: Q.non_archimedean_local_height(prec=100)
1.3862943611198906188344642429
sage: (3*Q).non_archimedean_local_height()
1/2*log(75923153929839865104)

sage: F.<a> = NumberField(x^4 + 2*x^3 + 19*x^2 + 18*x + 288)
sage: F.ring_of_integers().gens()
[1, 5/6*a^3 + 1/6*a, 1/6*a^3 + 1/6*a^2, a^3]
sage: F.class_number()
12
sage: E = EllipticCurve('37a').change_ring(F)
sage: P = E((-a^2/6 - a/6 - 1, a)); P
(-1/6*a^2 - 1/6*a - 1 : a : 1)
sage: P[0].is_integral()
True
sage: P.non_archimedean_local_height()
0
```

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

```
sage: E = EllipticCurve([0,17])
sage: P = E(2,5)
sage: P.non_archimedean_local_height(2)
-2/3*log(2)
```

nonarchimedian_local_height(**args*, ***kwds*)

Deprecated: Use `non_archimedean_local_height()` instead. See trac ticket #13951 for details.

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
```

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 trac ticket #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)
0
sage: P = E(0,0)
sage: P.padic_elliptic_logarithm(3)
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()
660257522
sage: P = E(-11/9,28/27)
sage: [(2*P).padic_elliptic_logarithm(p)/P.padic_elliptic_logarithm(p) for p in prime_range(
[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
sage: [(3*P).padic_elliptic_logarithm(p)/P.padic_elliptic_logarithm(p) for p in prime_range(
[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(
[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 trac ticket #4741:

```
sage: E = EllipticCurve('794a1')
sage: P = E(-1,2)
sage: P.padic_elliptic_logarithm(2) # default precision=20
```

```
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)
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)
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 +
```

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: 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: 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

TWENTY

CANONICAL HEIGHTS FOR ELLIPTIC CURVES OVER NUMBER FIELDS

Also, rigorous lower bounds for the canonical height of non-torsion points, implementing the algorithms in [CS] (over **Q**) and [TT], which also refer to [CPS].

AUTHORS:

- Robert Bradshaw (2010): initial version
- John Cremona (2014): added many docstrings and doctests

REFERENCES:

class sage.schemes.elliptic_curves.height.**EllipticCurveCanonicalHeight**(*E*)

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 R
```

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 R
```

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 [TT], section 5.

EXAMPLES:

Example 10.2 from [TT]:

```
sage: K.<i>=QuadraticField(-1)
sage: E = EllipticCurve([0,1-i,i,-i,0])
sage: H = E.height_function()
```

In [TT] the value is given as 0.772:

```
sage: RealField(12)( H.B(5, 0.01) )
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 [TT], section 4.

EXAMPLES:

```
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 [TT], 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)
sage: E = EllipticCurve([0,0,0,1+5*i,3+i])
sage: H = E.height_function()
sage: H.ME()
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 [TT] sectoin 6.1.

EXAMPLES:

An example over $\mathbf{Q}$:

```
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: 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)
([0.0781194447253472, 0.0823423732016403] U [0.917657626798360, 0.921880555274653])
```

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 [TT] (Lemma 6.1).

EXAMPLES:

An example over $\mathbf{Q}$:

```
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 [0.2040
```

An example over a number field:

```
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 [0.1903
```

alpha (*v*, *tol=0.01*)

Return the constant α_v associated to the embedding `v`.

INPUT:

- v – an embedding of the base field into $\mathbf{R}$ or $\mathbf{C}$

OUTPUT:

The constant α_v. In the notation of [CPS] (2006) and [TT] (section 3.2), $\alpha_v^3 = \epsilon_v$. The result is cached since it only depends on the curve.

EXAMPLES:

Example 1 from [CPS] (2006):

```
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 [CPS]:

```
sage: 3*alpha.log()
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 [TT], 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 [TT], Proposition 7.8.

EXAMPLES:

```
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 (~6s)
False
sage: H.complex_intersection_is_empty([H.B(n,0.02) for n in [1,2,3,4]],v)
True
sage: H.complex_intersection_is_empty([H.B(n,0.03) for n in [1,2,3,4]],v) # long time (4s)
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)
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: 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: good
...
sage: H.e_p(p1)
20815
sage: E.local_data(p2)
Local data at Fractional ideal (-40*i + 139):
Reduction type: bad split multiplicative
...
sage: H.e_p(p2)
20920
```

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 instad of CIF:

```
sage: fk, err = E.height_function().fk_intervals(N=10, domain=CDF)
sage: fk(z)
-1.82543539306 - 2.49336319993*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 [CS] (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)
0.00112632873099
```

Example 10.1 from [TT] (where a lower bound of 0.18 was given):

```
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 (8.1s)
0.162104944331
```

Example 10.2 from [TT]:

```
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 (4s)
0.0150437964347
```

In this example the point $P = (0, 0)$ has height 0.023 so our lower bound is quite good:

```
sage: P = E((0,0))
sage: P.height()
0.0230242154471211
```

Example 10.3 from [TT] (where the same bound of 0.0625 is given):

```
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 (7s)
0.0625
```

More examples over $\mathbf{Q}$:

```
sage: E = EllipticCurve('37a')
sage: h = E.height_function()
sage: h.min(.01, 5)
0.0398731805749
sage: E.gen(0).height()
0.0511114082399688
```

After base change the lower bound can decrease:

```
sage: K.<a> = QuadraticField(-5)
sage: E.change_ring(K).height_function().min(0.5, 10) # long time (8s)
0.0441941738242
```

```
sage: E = EllipticCurve('389a')
sage: h = E.height_function()
sage: h.min(0.1, 5)
0.0573127527003
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 [CS] (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)
1.98684388147
```

Example 10.1 from [TT] (where a lower bound of 0.18 was given):

```
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 (8.1s)
0.162104944331
```

Example 10.2 from [TT]:

```
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)
0.0150437964347
```

In this example the point $P = (0, 0)$ has height 0.023 so our lower bound is quite good:

```
sage: P = E((0,0))
sage: P.has_good_reduction()
True
sage: P.height()
0.0230242154471211
```

Example 10.3 from [TT] (where the same bound of 0.25 is given):

```
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 (7.2s)
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 $\mathbf{Q}$:

```
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)
1.53151606047462
sage: L(P) / L.real_period()
0.615014189772115
sage: H = E.height_function()
sage: H.psi(10/9,v)
0.615014189772115
```

An example over a number field:

```
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.xy()[0])
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 [TT], Proposition 6.2.

EXAMPLES:

An example over $\mathbf{Q}$:

```
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: 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 (3.3s)
True
sage: H.real_intersection_is_empty([H.B(n,0.08) for n in srange(1,5)],v)
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 bounf do 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`.

EXAMPLE:

```
sage: K.<a> = NumberField(x^3-2)
sage: E = EllipticCurve([0,0,0,0,a])
sage: H = E.height_function()
```

This curve does have a point of good reduction whose canonical point is approximately 1.68:

```
sage: P = E.gens()[0]
sage: P.height()
1.68038085233673
sage: P.has_good_reduction()
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)
B_1(0.100000000000000) = 1.51580969677387
B_2(0.100000000000000) = 0.932072561526720
True
sage: H.test_mu(0.2, 5)
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()
4.09924487233530
sage: P.division_points(2)
[]
sage: P.division_points(3)
[]
```

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: 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? - 2.4933631999285?*I

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*)

A class representing a finite union of closed intervals in $\mathbf{R}$ 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 trac ticket #13125; see trac ticket #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(range(10)).intervals()
[(0, 1), (2, 3), (4, 5), (6, 7), (8, 9)]
sage: UnionOfIntervals([-infinity, pi, 17, infinity]).intervals()
[(-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 polynomaials
- 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
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 minumum value.

EXAMPLE:

```
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])
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
```

CHAPTER

TWENTYONE

TORSION SUBGROUPS OF ELLIPTIC CURVES OVER NUMBER FIELDS (INCLUDING $\mathbb{Q}$)

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*, *algorithm=None*)

Bases: sage.groups.additive_abelian.additive_abelian_wrapper.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
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 Elliptic Curve defined by y^2 + 1
sage: G.gens()
()
sage: e = EllipticCurve([0, 33076156654533652066609946884,0,\
```

```
347897536144342179642120321790729023127716119338758604800,\
114112815436927429551902303280680424778815462104985764887003237028585178135281664000])
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), (7 : 0 : 1), (0 : 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
sage: [E(t) for t in T]
[(0 : 1 : 0)]
```

Examples over other Number Fields:

```
sage: E=EllipticCurve('11a1')
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 + (

sage: E=EllipticCurve('11a1')
sage: K.<i>=NumberField(x^2+1)
sage: EK=E.change_ring(K)
sage: T = EK.torsion_subgroup()
sage: T.ngens()
1
sage: T.gen(0)
(5 : -6 : 1)
```

Note: this class is normally constructed indirectly as follows:

```
sage: T = EK.torsion_subgroup(); T
Torsion Subgroup isomorphic to Z/5 associated to the Elliptic Curve defined by y^2 + y = x^3 + (
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: E=EllipticCurve('11a1')
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: 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), (-i : 0 : 1), (0 : 0 : 1), (i : 0 : 1)]
```

CHAPTER

TWENTYTWO

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 [Sil2] IV.9.4 or [Ta]).

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: 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*
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 po
```

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: 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
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: 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
```

REFERENCES:

- [Sil2] Silverman, Joseph H., Advanced topics in the arithmetic of elliptic curves. Graduate Texts in Mathematics, 151. Springer-Verlag, New York, 1994.
- [Ta] Tate, John, Algorithm for determining the type of a singular fiber in an elliptic pencil. Modular functions of one variable, IV, pp. 33–52. Lecture Notes in Math., Vol. 476, Springer, Berlin, 1975.

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: `sage.structure.sage_object.SageObject`

The class for the local reduction data of an elliptic curve.

Currently supported are elliptic curves defined over $\mathbf{Q}$, 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 $\mathbf{Q}$.
- `P` – a prime ideal of the field, or a prime integer if the field is $\mathbf{Q}$.
- `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 $\mathbf{Q}$. If "pari", use the PARI C-library `ellglobalred` implementation of Tate's algorithm over $\mathbf{Q}$. 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: 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: 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: 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: 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: 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: 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: 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: 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 polynomia
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

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
```

trac ticket #14476:

```
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,
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/
```

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 $\mathbf{Q}$).
- P – an element of K or a (fractional) ideal of K.

OUTPUT:

- If K is $\mathbf{Q}$: the prime integer equal to or which generates P.
- If K is not $\mathbf{Q}$: 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,ZZ.ideal(31))
31
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)]
```

CHAPTER

TWENTYTHREE

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 $\mathrm{I_m}$, II, III, IV, $\mathrm{I_m^*}$, 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*)

Returns 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: sage.structure.sage_object.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.

CHAPTER

TWENTYFOUR

ISOMORPHISMS BETWEEN WEIERSTRASS MODELS OF ELLIPTIC CURVES

AUTHORS:

- Robert Bradshaw (2007): initial version
- John Cremona (Jan 2008): isomorphisms, automorphisms and twists in all characteristics

class sage.schemes.elliptic_curves.weierstrass_morphism.**WeierstrassIsomorphism**(*E=None*, *urst=None*, *F=None*)

Bases: sage.schemes.elliptic_curves.weierstrass_morphism.baseWI, sage.categories.morphism.Morphism

Class representing a Weierstrass isomorphism between two elliptic curves.

class sage.schemes.elliptic_curves.weierstrass_morphism.**baseWI**(*u=1*, *r=0*, *s=0*, *t=0*)

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[]; baseWI(u,r,s,t)
(u, r, s, t)
```

is_identity()

Returns 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()

Returns the parameters u, r, s, t as a tuple.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.weierstrass_morphism import *
sage: u,r,s,t=baseWI(2,3,4,5).tuple()
sage: w=baseWI(2,3,4,5)
sage: u,r,s,t=w.tuple()
sage: u
2
```

sage.schemes.elliptic_curves.weierstrass_morphism.**isomorphisms**(*E*, *F*, *JustOne=False*)

Returns one or all isomorphisms between two elliptic curves.

INPUT:

- E, F (EllipticCurve) – Two elliptic curves.
- JustOne (bool) If True, returns one isomorphism, or None if the curves are not isomorphic. If False, returns a (possibly empty) list of isomorphisms.

OUTPUT:

Either None, or a 4-tuple (u, r, s, t) representing an isomorphism, or a list of these.

Note: This function is not intended for users, who should use the interface provided by ell_generic.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.weierstrass_morphism import *
sage: isomorphisms(EllipticCurve_from_j(0),EllipticCurve('27a3'))
[(-1, 0, 0, -1), (1, 0, 0, 0)]
sage: isomorphisms(EllipticCurve_from_j(0),EllipticCurve('27a3'),JustOne=True)
(1, 0, 0, 0)
sage: isomorphisms(EllipticCurve_from_j(0),EllipticCurve('27a1'))
[]
sage: isomorphisms(EllipticCurve_from_j(0),EllipticCurve('27a1'),JustOne=True)
```

CHAPTER

TWENTYFIVE

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
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
```

The functions directly accessible from an elliptic curve `E` over a field are `isogeny` and `isogeny_codomain`.

The most useful functions that apply to isogenies are

- `codomain`
- `degree`
- `domain`
- `dual`
- `rational_maps`
- `kernel_polynomial`

Warning: Only cyclic, separable isogenies are implemented (except for [2]). Some algorithms may need the isogeny to be normalized.

AUTHORS:

- Daniel Shumow <shumow@gmail.com>: 2009-04-19: initial version
- Chris Wuthrich : 7/09: changes: add check of input, not the full list is needed. 10/09: eliminating some bugs.

class sage.schemes.elliptic_curves.ell_curve_isogeny.**EllipticCurveIsogeny**(*E*, *kernel*, *codomain=None*, *degree=None*, *model=None*, *check=True*)

Bases: sage.categories.morphism.Morphism

Class Implementing Isogenies of Elliptic Curves

This class implements cyclic, separable, normalized isogenies of elliptic curves.

Several different algorithms for computing isogenies are available. These include:

- Velu's Formulas: Velu's original formulas for computing isogenies. This algorithm is selected by giving as the `kernel` parameter a list of points which generate 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 (little endian)) which will define the kernel of the isogeny.

INPUT:

- **E - an elliptic curve, the domain of the isogeny to** initialize.
- **kernel - a kernel, either a point in E, a list of points** in `E`, a monic kernel polynomial, or `None`. If initializing from a domain/codomain, this must be set to None.
- **codomain - an elliptic curve (default:None). If kernel** is `None`, then this must be the codomain of a cyclic, separable, normalized isogeny, furthermore, `degree` must be the degree of the isogeny from `E` to `codomain`. If `kernel` is not `None`, then this must be isomorphic to the codomain of the cyclic normalized separable isogeny defined by `kernel`, in this case, the isogeny is post composed with an isomorphism so that this parameter is the codomain.
- **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 kernel polynomial with the two torsion polynomial of `E`.
- **model - a string (default:None). Only supported variable is** `minimal`, in which case if `E` is a curve over the rationals, then the codomain is set to be the unique global minimum model.
- `check` (default: `True`) checks if the input is valid to define an isogeny

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
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: 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
sage: phi_ker_poly = phi_v.kernel_polynomial()
sage: phi_ker_poly
x + 1
sage: ker_poly_list = phi_ker_poly.list()
sage: phi_k = EllipticCurveIsogeny(E, ker_poly_list)
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()
True
sage: phi_k.degree()
3
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)))
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 that has kernel equal to the full 2 torsion:

```
sage: E = EllipticCurve(GF(3), [0,0,0,1,1])
sage: ker_list = E.division_polynomial(2).list()
sage: phi = EllipticCurveIsogeny(E, ker_list); phi
Isogeny of degree 4 from Elliptic Curve defined by y^2 = x^3 + x + 1 over Finite Field of size 3
sage: phi(E(0))
(0 : 1 : 0)
sage: phi(E((0,1)))
(1 : 0 : 1)
sage: phi(E((0,2)))
(1 : 0 : 1)
sage: phi(E((1,0)))
(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])
sage: 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 ov
sage: phi_k.degree()
1
sage: phi_k.rational_maps()
(x, y)
sage: phi_k.codomain() == E
True
sage: phi_k.kernel_polynomial()
1
sage: P = E.random_point(); P == phi_k(P)
True
```

Velu 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)
sage: phi
Isogeny of degree 2 from Elliptic Curve defined by y^2 = x^3 + 3*x + 4 over Rational Field to El
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: ker_poly_list = phi_ker_poly.list()
sage: phi_k = EllipticCurveIsogeny(E, ker_poly_list); phi_k
Isogeny of degree 2 from Elliptic Curve defined by y^2 = x^3 + 3*x + 4 over Rational Field to El
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
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: ker_poly_list = ker_poly.list()
sage: phi_k = EllipticCurveIsogeny(E, ker_poly_list); phi_k
Isogeny of degree 5 from Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational
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: E = EllipticCurve('11a1')
sage: P_list = E.torsion_points()
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) ov
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: ker_poly_list = ker_poly.list()
sage: phi_k = EllipticCurveIsogeny(EK, ker_poly_list)
sage: phi_k
Isogeny of degree 5 from Elliptic Curve defined by y^2 + y = x^3 + (-1)*x^2 + (-10)*x + (-20) ov
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)
sage: phi_s
Isogeny of degree 5 from Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rational
sage: phi_s == phi
True
sage: phi_s.rational_maps() == phi.rational_maps()
True
```

However only cyclic normalized isogenies can be constructed this way. So it won't find the isogeny [3]:

```
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
```

Also the presumed isogeny between the domain and codomain must be normalized:

```
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: phi.dual()
Isogeny of degree 5 from Elliptic Curve defined by y^2 + y = x^3 - x^2 - 7820*x - 263580 over Ra
sage: phi.dual().is_normalized()
False
```

Here an example of a construction of a endomorphisms with cyclic kernel on a CM-curve:

```
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.set_post_isomorphism(isom)
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
```

Domain and codomain tests (see trac ticket #12880):

```
sage: E = EllipticCurve(QQ, [0,0,0,1,0])
sage: phi = EllipticCurveIsogeny(E,  E(0,0))
sage: phi.domain() == E
True
sage: phi.codomain()
Elliptic Curve defined by y^2 = x^3 - 4*x over Rational Field

sage: E = EllipticCurve(GF(31), [1,0,0,1,2])
sage: phi = EllipticCurveIsogeny(E, [17, 1])
sage: phi.domain()
Elliptic Curve defined by y^2 + x*y = x^3 + x + 2 over Finite Field of size 31
sage: phi.codomain()
Elliptic Curve defined by y^2 + x*y = x^3 + 24*x + 6 over Finite Field of size 31
```

degree()

Returns the degree of this isogeny.

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
```

dual()

Computes and returns the dual isogeny of this isogeny. If $\varphi\colon E \to E_2$ is the given isogeny, then the dual is by definition the unique isogeny $\hat{\varphi}\colon E_2 \to E$ such that the compositions $\hat{\varphi} \circ \varphi$ and $\varphi \circ \hat{\varphi}$ are the multiplication $[n]$ by the degree of φ on E and E_2 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()
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
```

Test (for trac ticket 7096):

```
sage: E = EllipticCurve('11a1')
sage: phi = E.isogeny(E(5,5))
sage: phi.dual().dual() == phi
True
```

```
sage: k = GF(103)
sage: E = EllipticCurve(k,[11,11])
sage: phi = E.isogeny(E(4,4))
sage: phi
Isogeny of degree 5 from Elliptic Curve defined by y^2 = x^3 + 11*x + 11 over Finite Field o
sage: from sage.schemes.elliptic_curves.weierstrass_morphism import WeierstrassIsomorphism
sage: phi.set_post_isomorphism(WeierstrassIsomorphism(phi.codomain(),(5,0,1,2)))
sage: phi.dual().dual() == phi
True

sage: E = EllipticCurve(GF(103),[1,0,0,1,-1])
sage: phi = E.isogeny(E(60,85))
sage: phi.dual()
Isogeny of degree 7 from Elliptic Curve defined by y^2 + x*y = x^3 + 84*x + 34 over Finite F
```

formal (*prec=20*)

Computes the formal isogeny 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)
```

get_post_isomorphism()

Returns the post-isomorphism of this isogeny. If there has been no post-isomorphism set, this returns None.

EXAMPLES:

```
sage: E = EllipticCurve(j=GF(31)(0))
sage: R.<x> = GF(31)[]
sage: phi = EllipticCurveIsogeny(E, x+18)
sage: phi.get_post_isomorphism()
sage: from sage.schemes.elliptic_curves.weierstrass_morphism import WeierstrassIsomorphism
sage: isom = WeierstrassIsomorphism(phi.codomain(), (6,8,10,12))
sage: phi.set_post_isomorphism(isom)
sage: isom == phi.get_post_isomorphism()
True

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: phi2 = EllipticCurveIsogeny(E, None, E2, 2)
sage: phi2.get_post_isomorphism()
Generic morphism:
From: Abelian group of points on Elliptic Curve defined by y^2 = x^3 + 65*x + 69 over Finite
To:   Abelian group of points on Elliptic Curve defined by y^2 + x*y + 77*y = x^3 + 49*x + 2
Via:  (u,r,s,t) = (1, 7, 42, 80)
```

get_pre_isomorphism()

Returns the pre-isomorphism of this isogeny. If there has been no pre-isomorphism set, this returns `None`.

EXAMPLES:

```
sage: E = EllipticCurve(GF(31), [1,1,0,1,-1])
sage: R.<x> = GF(31)[]
sage: f = x^3 + 9*x^2 + x + 30
sage: phi = EllipticCurveIsogeny(E, f)
sage: phi.get_post_isomorphism()
sage: Epr = E.short_weierstrass_model()
sage: isom = Epr.isomorphism_to(E)
sage: phi.set_pre_isomorphism(isom)
sage: isom == phi.get_pre_isomorphism()
True

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: phi2 = EllipticCurveIsogeny(E, None, E2, 2)
sage: phi2.get_pre_isomorphism()
Generic morphism:
  From: Abelian group of points on Elliptic Curve defined by y^2 + x*y + y = x^3 + x over Fi
  To:   Abelian group of points on Elliptic Curve defined by y^2 = x^3 + 62*x + 74 over Fini
  Via:  (u,r,s,t) = (1, 76, 41, 3)
```

is_injective()

Method inherited from the morphism class. Returns `True` if and only if this isogeny has trivial kernel.

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
```

is_normalized(*via_formal=True*, *check_by_pullback=True*)

Returns `True` if this isogeny is normalized. An isogeny $\varphi\colon E \to E_2$ between two given Weierstrass equations is said to be normalized if the constant c is 1 in $\varphi * (\omega_2) = c \cdot \omega$, where ω and $omega_2$ are the invariant differentials on E and E_2 corresponding to the given equation.

INPUT:

- **via_formal - (default: True) If True it simply checks if** the leading term of the formal series is 1. Otherwise it uses a deprecated algorithm involving the second optional argument.
- `check_by_pullback` - (default:`True`) Deprecated.

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.set_post_isomorphism(isom)
sage: phi.is_normalized()
False
sage: isom = WeierstrassIsomorphism(phi.codomain(), (5, 0, 0, 0))
sage: phi.set_post_isomorphism(isom)
sage: phi.is_normalized()
True
sage: isom = WeierstrassIsomorphism(phi.codomain(), (1, 1, 1, 1))
sage: phi.set_post_isomorphism(isom)
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))
sage: phi.is_normalized()
True
sage: phi.set_post_isomorphism(isom)
sage: phi.is_normalized()
False
sage: isom = WeierstrassIsomorphism(phi.codomain(), (1/alpha, 0, 0, 0))
sage: phi.set_post_isomorphism(isom)
sage: phi.is_normalized()
True
sage: isom = WeierstrassIsomorphism(phi.codomain(), (1, 1, 1, 1))
sage: phi.set_post_isomorphism(isom)
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.set_post_isomorphism(isom)
```

```
sage: phi.is_normalized()
False
sage: isom = WeierstrassIsomorphism(phi.codomain(), (1/2, 0, 0, 0))
sage: phi.set_post_isomorphism(isom)
sage: phi.is_normalized()
True
sage: isom = WeierstrassIsomorphism(phi.codomain(), (1, 1, 1, 1))
sage: phi.set_post_isomorphism(isom)
sage: phi.is_normalized()
True
```

is_separable()

This function returns a bool indicating whether or not this isogeny is separable.

This function always returns `True` as currently this class only implements separable isogenies.

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
```

is_surjective()

For elliptic curve isogenies, always returns `True` (as a non-constant map of algebraic curves must be surjective).

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: R.<x> = QQ[]
sage: f = x^2 + x - 29/5
sage: phi = EllipticCurveIsogeny(E, f)
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()

Member function inherited from morphism class.

EXAMPLES:

```
sage: E = EllipticCurve(j=GF(7)(0))
sage: phi = EllipticCurveIsogeny(E, [ E((0,1)), E((0,-1))])
sage: phi.is_zero()
```

```
Traceback (most recent call last):
...
NotImplementedError
```

kernel_polynomial()
Returns 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
```

n()
Numerical Approximation inherited from Map (through morphism), nonsensical for isogenies.

EXAMPLES:

```
sage: E = EllipticCurve(j=GF(7)(0))
sage: phi = EllipticCurveIsogeny(E, [ E((0,1)), E((0,-1))])
sage: phi.n()
Traceback (most recent call last):
...
NotImplementedError: Numerical approximations do not make sense for Elliptic Curve Isogenies
```

post_compose(*left*)
Member function inherited from morphism class.

EXAMPLES:

```
sage: E = EllipticCurve(j=GF(7)(0))
sage: phi = EllipticCurveIsogeny(E, [ E((0,1)), E((0,-1))])
sage: phi.post_compose(phi)
Traceback (most recent call last):
...
NotImplementedError
```

pre_compose(*right*)
Member function inherited from morphism class.

EXAMPLES:

```
sage: E = EllipticCurve(j=GF(7)(0))
sage: phi = EllipticCurveIsogeny(E, [ E((0,1)), E((0,-1))])
sage: phi.pre_compose(phi)
```

```
Traceback (most recent call last):
...
NotImplementedError
```

rational_maps()

This function returns this isogeny as a pair of rational maps.

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)
```

set_post_isomorphism(*postWI*)

Modifies this isogeny object to post compose with the given Weierstrass isomorphism.

EXAMPLES:

```
sage: E = EllipticCurve(j=GF(31)(0))
sage: R.<x> = GF(31)[]
sage: phi = EllipticCurveIsogeny(E, x+18)
sage: from sage.schemes.elliptic_curves.weierstrass_morphism import WeierstrassIsomorphism
sage: phi.set_post_isomorphism(WeierstrassIsomorphism(phi.codomain(), (6,8,10,12)))
sage: phi
Isogeny of degree 3 from Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field of size 3

sage: E = EllipticCurve(j=GF(47)(0))
sage: f = E.torsion_polynomial(3)/3
sage: phi = EllipticCurveIsogeny(E, f)
sage: E2 = phi.codomain()
sage: post_isom = E2.isomorphism_to(E)
sage: phi.set_post_isomorphism(post_isom)
sage: phi.rational_maps() == E.multiplication_by_m(3)
False
sage: phi.switch_sign()
sage: phi.rational_maps() == E.multiplication_by_m(3)
True
```

Example over a number field:

```
sage: R.<x> = QQ[]
sage: K.<a> = NumberField(x^2 + 2)
sage: E = EllipticCurve(j=K(1728))
sage: ker_list = E.torsion_points()
sage: phi = EllipticCurveIsogeny(E, ker_list)
sage: from sage.schemes.elliptic_curves.weierstrass_morphism import WeierstrassIsomorphism
sage: post_isom = WeierstrassIsomorphism(phi.codomain(), (a,2,3,5))
sage: phi
Isogeny of degree 4 from Elliptic Curve defined by y^2 = x^3 + x over Number Field in a with
```

set_pre_isomorphism(*preWI*)

Modifies this isogeny object to pre compose with the given Weierstrass isomorphism.

EXAMPLES:

```
sage: E = EllipticCurve(GF(31), [1,1,0,1,-1])
sage: R.<x> = GF(31)[]
sage: f = x^3 + 9*x^2 + x + 30
sage: phi = EllipticCurveIsogeny(E, f)
sage: Epr = E.short_weierstrass_model()
sage: isom = Epr.isomorphism_to(E)
sage: phi.set_pre_isomorphism(isom)
sage: phi.rational_maps()
((-6*x^4 - 3*x^3 + 12*x^2 + 10*x - 1)/(x^3 + x - 12),
 (3*x^7 + x^6*y - 14*x^6 - 3*x^5 + 5*x^4*y + 7*x^4 + 8*x^3*y - 8*x^3 - 5*x^2*y + 5*x^2 - 14*
sage: phi(Epr((0,22)))
(13 : 21 : 1)
sage: phi(Epr((3,7)))
(14 : 17 : 1)

sage: E = EllipticCurve(GF(29), [0,0,0,1,0])
sage: R.<x> = GF(29)[]
sage: f = x^2 + 5
sage: phi = EllipticCurveIsogeny(E, f)
sage: phi
Isogeny of degree 5 from Elliptic Curve defined by y^2 = x^3 + x over Finite Field of size 2
sage: from sage.schemes.elliptic_curves.weierstrass_morphism import WeierstrassIsomorphism
sage: inv_isom = WeierstrassIsomorphism(E, (1,-2,5,10))
sage: Epr = inv_isom.codomain().codomain()
sage: isom = Epr.isomorphism_to(E)
sage: phi.set_pre_isomorphism(isom); phi
Isogeny of degree 5 from Elliptic Curve defined by y^2 + 10*x*y + 20*y = x^3 + 27*x^2 + 6 ov
sage: phi(Epr((12,1)))
(26 : 0 : 1)
sage: phi(Epr((2,9)))
(0 : 0 : 1)
sage: phi(Epr((21,12)))
(3 : 0 : 1)
sage: phi.rational_maps()[0]
(x^5 - 10*x^4 - 6*x^3 - 7*x^2 - x + 3)/(x^4 - 8*x^3 + 5*x^2 - 14*x - 6)

sage: E = EllipticCurve('11a1')
sage: R.<x> = QQ[]
sage: f = x^2 - 21*x + 80
sage: phi = EllipticCurveIsogeny(E, f); phi
Isogeny of degree 5 from Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 over Rati
sage: from sage.schemes.elliptic_curves.weierstrass_morphism import WeierstrassIsomorphism
sage: Epr = E.short_weierstrass_model()
sage: isom = Epr.isomorphism_to(E)
sage: phi.set_pre_isomorphism(isom)
sage: phi
Isogeny of degree 5 from Elliptic Curve defined by y^2 = x^3 - 13392*x - 1080432 over Ration
sage: phi(Epr((168,1188)))
(0 : 1 : 0)
```

switch_sign()

This function composes the isogeny with $[-1]$ (flipping the coefficient between +/-1 on the y coordinate rational map).

EXAMPLES:

```
sage: E = EllipticCurve(GF(23), [0,0,0,1,0])
sage: f = E.torsion_polynomial(3)/3
sage: phi = EllipticCurveIsogeny(E, f, E)
```

```
sage: phi.rational_maps() == E.multiplication_by_m(3)
False
sage: phi.switch_sign()
sage: phi.rational_maps() == E.multiplication_by_m(3)
True

sage: E = EllipticCurve(GF(17), [-2, 3, -5, 7, -11])
sage: R.<x> = GF(17)[]
sage: f = x+6
sage: phi = EllipticCurveIsogeny(E, f)
sage: phi
Isogeny of degree 2 from Elliptic Curve defined by y^2 + 15*x*y + 12*y = x^3 + 3*x^2 + 7*x +
sage: phi.rational_maps()
((x^2 + 6*x + 4)/(x + 6), (x^2*y - 5*x*y + 8*x - 2*y)/(x^2 - 5*x + 2))
sage: phi.switch_sign()
sage: phi
Isogeny of degree 2 from Elliptic Curve defined by y^2 + 15*x*y + 12*y = x^3 + 3*x^2 + 7*x +
sage: phi.rational_maps()
((x^2 + 6*x + 4)/(x + 6),
 (2*x^3 - x^2*y - 5*x^2 + 5*x*y - 4*x + 2*y + 7)/(x^2 - 5*x + 2))

sage: E = EllipticCurve('11a1')
sage: R.<x> = QQ[]
sage: f = x^2 - 21*x + 80
sage: phi = EllipticCurveIsogeny(E, f)
sage: (xmap1, ymap1) = phi.rational_maps()
sage: phi.switch_sign()
sage: (xmap2, ymap2) = phi.rational_maps()
sage: xmap1 == xmap2
True
sage: ymap1 == -ymap2 - E.a1()*xmap2 - E.a3()
True

sage: K.<a> = NumberField(x^2 + 1)
sage: E = EllipticCurve(K, [0,0,0,1,0])
sage: R.<x> = K[]
sage: phi = EllipticCurveIsogeny(E, x-a)
sage: phi.rational_maps()
((x^2 + (-a)*x - 2)/(x + (-a)), (x^2*y + (-2*a)*x*y + y)/(x^2 + (-2*a)*x - 1))
sage: phi.switch_sign()
sage: phi.rational_maps()
((x^2 + (-a)*x - 2)/(x + (-a)), (-x^2*y + (2*a)*x*y - y)/(x^2 + (-2*a)*x - 1))
```

sage.schemes.elliptic_curves.ell_curve_isogeny.**compute_codomain_formula**(*E*, *v*, *w*)

Given parameters v and w (as in Velu / Kohel / etc formulas) computes the codomain curve.

EXAMPLES:

This formula is used by every Isogeny Instantiation:

```
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)
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.schemes.elliptic_curves.ell_curve_isogeny.**compute_codomain_kohel**(*E*, *kernel*, *degree*)

This function computes the codomain from the kernel polynomial as per Kohel's formulas.

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], 2)
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, 7)
Elliptic Curve defined by y^2 + 18*x*y + 16*y = x^3 + 17*x^2 + 18*x + 18 over Finite Field of si
sage: E = EllipticCurve(GF(19), [1,2,3,4,5])
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 + 12,4)
Elliptic Curve defined by y^2 + x*y + 3*y = x^3 + 2*x^2 + 3*x + 15 over Finite Field of size 19
```

NOTES:

This function uses the formulas of Section 2.4 of [K96].

REFERENCES:

•[K96] Kohel, "Endomorphism Rings of Elliptic Curves over Finite Fields"

sage.schemes.elliptic_curves.ell_curve_isogeny.**compute_intermediate_curves**(*E1*, *E2*)

Computes isomorphism from `E1` to an intermediate domain and an isomorphism from an intermediate codomain to `E2`.

Intermediate domain and intermediate codomain, are in short Weierstrass form.

This is used so we can compute $\wp$ functions from the short Weierstrass model more easily.

The underlying field must be of characteristic not equal to 2,3.

INPUT:

•`E1` - an elliptic curve

•`E2` - an elliptic curve

OUTPUT:

tuple – (pre_isomorphism, post_isomorphism, intermediate_domain, intermediate_codomain):

- `intermediate_domain`: a short Weierstrass model isomorphic to `E1`
- `intermediate_codomain`: a short Weierstrass model isomorphic to `E2`
- `pre_isomorphism`: normalized isomorphism from `E1` to intermediate_domain
- `post_isomorphism`: 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,
 Generic morphism:
  From: Abelian group of points on Elliptic Curve defined by y^2 + x*y + y = x^3 + x over Finite
  To:   Abelian group of points on Elliptic Curve defined by y^2 = x^3 + 62*x + 74 over Finite F
  Via:  (u,r,s,t) = (1, 76, 41, 3),
 Generic morphism:
  From: Abelian group of points on Elliptic Curve defined by y^2 = x^3 + 65*x + 69 over Finite F
  To:   Abelian group of points on Elliptic Curve defined by y^2 + x*y + 77*y = x^3 + 49*x + 28
  Via:  (u,r,s,t) = (1, 7, 42, 80))

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
 Generic endomorphism of Abelian group of points on Elliptic Curve defined by y^2 = x^3 + x over
  Via:  (u,r,s,t) = (1, 0, 0, 0),
 Generic endomorphism of Abelian group of points on Elliptic Curve defined by y^2 = x^3 + 16*x o
  Via:  (u,r,s,t) = (1, 0, 0, 0))
```

sage.schemes.elliptic_curves.ell_curve_isogeny.**compute_isogeny_kernel_polynomial**(*E1*, *E2*, *ell*, *algorithm='starks*

Computes the kernel polynomial of the degree `ell` isogeny between `E1` and `E2`. There must be a degree `ell`, cyclic, separable, normalized isogeny from `E1` to `E2`.

INPUT:

- `E1` - an elliptic curve in short Weierstrass form.
- `E2` - an elliptic curve in short Weierstrass form.
- `ell` - the degree of the isogeny from `E1` to `E2`.
- `algorithm` - currently only `starks` (default) is implemented.

OUTPUT:

polynomial – over the field of definition of `E1`, `E2`, that is the kernel polynomial of the isogeny from `E1` to `E2`.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_curve_isogeny import compute_isogeny_kernel_polynomi

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: 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_starks**(*E1*, *E2*, *ell*)

Computes the degree `ell` isogeny between `E1` and `E2` via Stark's algorithm. There must be a degree `ell`, separable, normalized cyclic isogeny from `E1` to `E2`.

INPUT:

- `E1` - an elliptic curve in short Weierstrass form.
- `E2` - an elliptic curve in short Weierstrass form.
- `ell` - the degree of the isogeny from E1 to E2.

OUTPUT:

polynomial – over the field of definition of `E1`, `E2`, that is the kernel polynomial of the isogeny from `E1` to `E2`.

ALGORITHM:

This function uses Starks Algorithm as presented in section 6.2 of [BMSS].

Note: As published there, the algorithm is incorrect, and a correct version (with slightly different notation) can be found in [M09]. The algorithm originates in [S72]

REFERENCES:

- [BMSS] Boston, Morain, Salvy, Schost, "Fast Algorithms for Isogenies."
- [M09] Moody, "The Diffie-Hellman Problem and Generalization of Verheul's Theorem"
- [S72] Stark, "Class-numbers of complex quadratic fields."

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.ell_curve_isogeny import compute_isogeny_starks, compute

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_starks(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_starks(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_starks(E, E2, 2)
x
```

sage.schemes.elliptic_curves.ell_curve_isogeny.**compute_sequence_of_maps**(*E1*, *E2*, *ell*)

Given domain `E1` and codomain `E2` such that there is a degree `ell` separable normalized isogeny from `E1` to `E2`, returns pre/post isomorphism, as well as intermediate domain and codomain, and kernel polynomial.

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()
sage: compute_sequence_of_maps(E, E2, 5)
(Generic morphism:
  From: Abelian group of points on Elliptic Curve defined by y^2 + y = x^3 - x^2 - 10*x - 20 ove
  To:   Abelian group of points on Elliptic Curve defined by y^2 = x^3 - 31/3*x - 2501/108 over
  Via:  (u,r,s,t) = (1, 1/3, 0, -1/2),
 Generic morphism:
  From: Abelian group of points on Elliptic Curve defined by y^2 = x^3 - 23461/3*x - 28748141/10
  To:   Abelian group of points on Elliptic Curve defined by y^2 + y = x^3 - x^2 - 7820*x - 2635
  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: 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)
(Generic endomorphism of Abelian group of points on Elliptic Curve defined by y^2 = x^3 + x over
  Via:  (u,r,s,t) = (1, 0, 0, 0),
 Generic endomorphism of Abelian group of points on Elliptic Curve defined by y^2 = x^3 + 16*x c
  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
 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)
(Generic morphism:
  From: Abelian group of points on Elliptic Curve defined by y^2 + x*y + y = x^3 + x over Finite
  To:   Abelian group of points on Elliptic Curve defined by y^2 = x^3 + 52*x + 31 over Finite F
  Via:  (u,r,s,t) = (1, 8, 48, 44),
 Generic morphism:
  From: Abelian group of points on Elliptic Curve defined by y^2 = x^3 + 41*x + 66 over Finite F
  To:   Abelian group of points on Elliptic Curve defined by y^2 + x*y + 9*y = x^3 + 83*x + 6 ov
  Via:  (u,r,s,t) = (1, 89, 49, 53),
 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*)

The formula for computing v and w using Kohel's formulas for isogenies of degree 2.

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])
sage: 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
sage: from sage.schemes.elliptic_curves.ell_curve_isogeny import compute_vw_kohel_even_deg1
sage: a1,a2,a3,a4,a6 = E.ainvs()
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*)

The formula for computing v and w using Kohel's formulas for isogenies of degree 3.

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)
sage: 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
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*)

This function computes the v and w according to Kohel's formulas.

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)
sage: phi
Isogeny of degree 7 from Elliptic Curve defined by y^2 + 18*x*y + 16*y = x^3 + 17*x^2 + 15*x + 1
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*)

Returns 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 is 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 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*, *degree=None*)

This function computes the isogeny codomain given a kernel.

INPUT:

- E - The domain elliptic curve.

- **kernel - Either a list of points in the kernel of the isogeny, or a** kernel polynomial (specified as a either a univariate polynomial or a coefficient list.)
- **degree - an integer, (default:None) optionally specified degree** of the kernel.

OUTPUT:

(elliptic curve) the codomain of the separable normalized isogeny from 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], degree=3)
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], degree
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: 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.**isogeny_determine_algorithm**(*E*, *kernel*, *codomain*, *degree*, *model*)

Helper function that allows the various isogeny functions to infer the algorithm type from the parameters passed in.

If `kernel` is a list of points on the EllipticCurve E, then we assume the algorithm to use is Velu.

If `kernel` is a list of coefficients or a univariate polynomial we try to use the Kohel's algorithms.

EXAMPLES:

This helper function will be implicitly called by the following examples:

```
sage: R.<x> = GF(5)[]
sage: E = EllipticCurve(GF(5), [0,0,0,1,0])
sage: phi = EllipticCurveIsogeny(E, x+3)
sage: phi2 = EllipticCurveIsogeny(E, [GF(5)(3),GF(5)(1)])
sage: phi == phi2
True
sage: phi3 = EllipticCurveIsogeny(E,  E((2,0)) )
sage: phi3 == phi2
True
sage: from sage.schemes.elliptic_curves.ell_curve_isogeny import isogeny_determine_algorithm
sage: isogeny_determine_algorithm(E, x+3, None, None, None)
'kohel'
sage: isogeny_determine_algorithm(E, [3, 1], None, None, None)
'kohel'
sage: isogeny_determine_algorithm(E, E((2,0)), None, None, None)
'velu'
```

sage.schemes.elliptic_curves.ell_curve_isogeny.**split_kernel_polynomial**(*El*, *ker_poly*, *ell*)

Internal helper function for `compute_isogeny_kernel_polynomial`.

Given a full kernel polynomial (where two torsion x-coordinates are roots of multiplicity 1, and all other roots have multiplicity 2.) of degree $\ell - 1$, returns the maximum separable divisor. (i.e. the kernel polynomial with roots of multiplicity at most 1).

EXAMPLES:

The following example implicitly exercises this function:

```
sage: E = EllipticCurve(GF(37), [0,0,0,1,8])
sage: R.<x> = GF(37)[]
sage: f = (x + 10) * (x + 12) * (x + 16)
sage: phi = EllipticCurveIsogeny(E, f)
sage: E2 = phi.codomain()
sage: from sage.schemes.elliptic_curves.ell_curve_isogeny import compute_isogeny_starks
sage: from sage.schemes.elliptic_curves.ell_curve_isogeny import split_kernel_polynomial
sage: ker_poly = compute_isogeny_starks(E, E2, 7); ker_poly
x^6 + 2*x^5 + 20*x^4 + 11*x^3 + 36*x^2 + 35*x + 16
sage: split_kernel_polynomial(E, ker_poly, 7)
x^3 + x^2 + 28*x + 33
```

sage.schemes.elliptic_curves.ell_curve_isogeny.**two_torsion_part**(*E*, *poly_ring*, *psi*, *degree*)

Returns the greatest common divisor 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, R, x+13, 2)
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 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_isc
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]
[0 3 0 2 0 0]
sage: unfill_isogeny_matrix(M1) == M
True
```

CHAPTER

TWENTYSIX

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].

REFERENCES:

sage.schemes.elliptic_curves.isogeny_small_degree.**Fricke_module**(*l*)

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 $P(t)/t$.

TESTS:

```
sage: from sage.schemes.elliptic_curves.isogeny_small_degree import Fricke_module
sage: Fricke_module(2)
(t^3 + 48*t^2 + 768*t + 4096)/t
sage: Fricke_module(3)
(t^4 + 36*t^3 + 270*t^2 + 756*t + 729)/t
sage: Fricke_module(5)
(t^6 + 30*t^5 + 315*t^4 + 1300*t^3 + 1575*t^2 + 750*t + 125)/t
sage: Fricke_module(7)
(t^8 + 28*t^7 + 322*t^6 + 1904*t^5 + 5915*t^4 + 8624*t^3 + 4018*t^2 + 748*t + 49)/t
sage: Fricke_module(13)
(t^14 + 26*t^13 + 325*t^12 + 2548*t^11 + 13832*t^10 + 54340*t^9 + 157118*t^8 + 333580*t^7 + 5093
```

sage.schemes.elliptic_curves.isogeny_small_degree.**Fricke_polynomial**(*l*)

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 $\mathbf{Z}[t]$.

TESTS:

```
sage: from sage.schemes.elliptic_curves.isogeny_small_degree import Fricke_polynomial
sage: Fricke_polynomial(2)
t^3 + 48*t^2 + 768*t + 4096
sage: Fricke_polynomial(3)
t^4 + 36*t^3 + 270*t^2 + 756*t + 729
sage: Fricke_polynomial(5)
t^6 + 30*t^5 + 315*t^4 + 1300*t^3 + 1575*t^2 + 750*t + 125
sage: Fricke_polynomial(7)
t^8 + 28*t^7 + 322*t^6 + 1904*t^5 + 5915*t^4 + 8624*t^3 + 4018*t^2 + 748*t + 49
sage: Fricke_polynomial(13)
t^14 + 26*t^13 + 325*t^12 + 2548*t^11 + 13832*t^10 + 54340*t^9 + 157118*t^8 + 333580*t^7 + 50936
```

sage.schemes.elliptic_curves.isogeny_small_degree.**Psi**(*l*, *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.

TESTS:

```
sage: from sage.schemes.elliptic_curves.isogeny_small_degree import Fricke_module, Psi
sage: assert Psi(2, use_stored=True) == Psi(2, use_stored=False)
sage: assert Psi(3, use_stored=True) == Psi(3, use_stored=False)
sage: assert Psi(5, use_stored=True) == Psi(5, use_stored=False)
sage: assert Psi(7, use_stored=True) == Psi(7, use_stored=False)
sage: assert Psi(13, use_stored=True) == Psi(13, use_stored=False) # not tested (very long time)
```

sage.schemes.elliptic_curves.isogeny_small_degree.**Psi2**(*l*)
Returns 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.

TESTS:

```
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 -
```

sage.schemes.elliptic_curves.isogeny_small_degree.**isogenies_13_0**(*E*)
Returns list of all 13-isogenies from E when the j-invariant is 0.

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: 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
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 d
Isogeny of degree 13 from Elliptic Curve defined by y^2 = x^3 + r over Number Field in r with de
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 -
```

An example of endomorphisms over a finite field:

```
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 siz
Isogeny of degree 13 from Elliptic Curve defined by y^2 = x^3 + 1 over Finite Field in a of size
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
```

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
```

```
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)
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, -739946459/2385716286l049856*a^11 - 2591641747/1062017577504*a^8 + 16583647773233/424
```

sage.schemes.elliptic_curves.isogeny_small_degree.**isogenies_13_1728**(*E*)

Returns list of all 13-isogenies from E when the j-invariant is 1728.

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 $\mathbf{Q}$, the codomain is a minimal model.

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)
sage: E = EllipticCurve([0,0,0,i,0]); E.ainvs()
(0, 0, 0, i, 0)
sage: isogenies_13_1728(E)
[Isogeny of degree 13 from Elliptic Curve defined by y^2 = x^3 + i*x over Number Field in i with
Isogeny of degree 13 from Elliptic Curve defined by y^2 = x^3 + i*x over Number Field in i with

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)
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 8
Isogeny of degree 13 from 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
```

```
sage: x = polygen(QQ)
sage: f = x^12 + 1092*x^10 - 432432*x^8 + 6641024*x^6 - 282896640*x^4 - 149879808*x^2 - 34936012
sage: K.<a> = NumberField(f)
sage: E = EllipticCurve(K, [1,0])
sage: [phi.codomain().ainvs() for phi in isogenies_13_1728(E)]
[(0,
0,
0,
11090413835/2094372703969862 4*a^10 + 32280103535965/55849938772529664*a^8 - 355655987835845/1551
214217013065/82065216155553792*a^11 + 1217882637605/427423000810176*a^9 - 214645003230565/189965
(0,
0,
0,
11090413835/2094372703969862 4*a^10 + 32280103535965/55849938772529664*a^8 - 355655987835845/1551
-214217013065/82065216155553792*a^11 - 1217882637605/427423000810176*a^9 + 214645003230565/18996
```

sage.schemes.elliptic_curves.isogeny_small_degree.**isogenies_2**(*E*)

Returns a list of all 2-isogenies with domain `E`.

INPUT:

- `E` – an elliptic curve.

OUTPUT:

(list) 2-isogenies with domain `E`. In general these are normalised, but over $\mathbf{Q}$ the codomain is a minimal model.

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
Elliptic Curve defined by y^2 = x^3 + 9*x + 8 over Algebraic Field
sage: isogenies_2(E) # not implemented
```

sage.schemes.elliptic_curves.isogeny_small_degree.**isogenies_3**(*E*)

Returns a list of all 3-isogenies with domain `E`.

INPUT:

- `E` – an elliptic curve.

OUTPUT:

(list) 3-isogenies with domain `E`. In general these are normalised, but over $\mathbf{Q}$ the codomain is a minimal model.

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])
```

```
sage: [phi.codomain().ainvs() for phi in isogenies_3(E)]
[(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)]

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*)

Returns a list of all the 5-isogenies with domain `E` when the j-invariant is 0.

OUTPUT:

(list) 5-isogenies with codomain E. In general these are normalised, but over $\mathbf{Q}$ the codomain is a minimal model.

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])
sage: isogenies_5_0(E)
[Isogeny of degree 5 from Elliptic Curve defined by y^2 = x^3 + 10 over Finite Field in a of siz

sage: K.<a> = NumberField(x**6-320*x**3-320)
sage: E = EllipticCurve(K,[0,0,1,0,0])
sage: isogenies_5_0(E)
[Isogeny of degree 5 from Elliptic Curve defined by y^2 + y = x^3 over Number Field in a with de
Isogeny of degree 5 from Elliptic Curve defined by y^2 + y = x^3 over Number Field in a with def
```

sage.schemes.elliptic_curves.isogeny_small_degree.**isogenies_5_1728**(*E*)

Returns a list of 5-isogenies with domain `E` when the j-invariant is 1728.

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 $\mathbf{Q}$, the codomain is a minimal model.

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 1
Isogeny of degree 5 from Elliptic Curve defined by y^2 = x^3 + 11*x over Finite Field of size 13
```

An example of endomorphisms of degree 5:

```
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 de
Isogeny of degree 5 from Elliptic Curve defined by y^2 = x^3 + x over Number Field in i with def
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
```

An example of 5-isogenies over a number field:

```
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])
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 de
Isogeny of degree 5 from Elliptic Curve defined by y^2 = x^3 + x over Number Field in a with def
```

sage.schemes.elliptic_curves.isogeny_small_degree.**isogenies_7_0**(*E*)

Returns list of all 7-isogenies from E when the j-invariant is 0.

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 $\mathbf{Q}$, the codomain is a minimal model.

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)
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 Number Field in r with de
Isogeny of degree 7 from Elliptic Curve defined by y^2 = x^3 + 1 over Number Field in r with def

sage: E = EllipticCurve(GF(13^2,'a'),[0,-3])
sage: isogenies_7_0(E)
```

```
[Isogeny of degree 7 from Elliptic Curve defined by y^2 = x^3 + 10 over Finite Field in a of siz
```

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
```

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'))
sage: isogenies_7_0(E)
[Isogeny of degree 7 from Elliptic Curve defined by y^2 + y = x^3 + (-7) over Number Field in r
Isogeny of degree 7 from Elliptic Curve defined by y^2 + y = x^3 + (-7) over Number Field in r w

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, -5/294*a^5 - 300/7*a^2, -55/2*a^3 - 1133),
(0, 0, 0, -295/1176*a^5 - 5385/14*a^2, 55/2*a^3 + 40447)]
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*)

Returns list of all 7-isogenies from E when the j-invariant is 1728.

OUTPUT:

(list) 7-isogenies with codomain E. In general these are normalised; but over **Q** the codomain is a minimal model.

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])
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 t
Isogeny of degree 7 from Elliptic Curve defined by y^2 = x^3 + x over Finite Field of size 47 to
```

An example in characteristic 53 (for which an earlier implementation did not work):

```
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*
```

```
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().a_invariants() for phi in isogs]
[(0,
0,
0,
35/636*a^6 + 55/12*a^4 - 79135/636*a^2 + 1127/212,
155/636*a^7 + 245/12*a^5 - 313355/636*a^3 - 3577/636*a),
(0,
0,
0,
35/636*a^6 + 55/12*a^4 - 79135/636*a^2 + 1127/212,
-155/636*a^7 - 245/12*a^5 + 313355/636*a^3 + 3577/636*a)]
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*)

Returns a list of `l` -isogenies with domain `E`.

INPUT:

- `E` – an elliptic curve.
- `l` – a prime.

OUTPUT:

(list) a list of all isogenies of degree l. If the characteristic is l then only separable isogenies are constructed.

EXAMPLES:

```
sage: from sage.schemes.elliptic_curves.isogeny_small_degree import isogenies_prime_degree
sage: E = EllipticCurve_from_j(GF(2^6,'a')(1))
sage: isogenies_prime_degree(E, 7)
[Isogeny of degree 7 from Elliptic Curve defined by y^2 + x*y = x^3 + 1 over Finite Field in a c
sage: E = EllipticCurve_from_j(GF(3^12,'a')(2))
sage: isogenies_prime_degree(E, 17)
[Isogeny of degree 17 from Elliptic Curve defined by y^2 = x^3 + 2*x^2 + 2 over Finite Field in
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 Fi
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 Fi
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 Rationa
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 Rati
```

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
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 Fi
[Isogeny of degree 5 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 4*x + 4 over Finite Fi
[Isogeny of degree 7 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 6*x + 6 over Finite Fi
[Isogeny of degree 11 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 10*x + 10 over Finite
[Isogeny of degree 13 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 12*x + 12 over Finite
[Isogeny of degree 17 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 16*x + 16 over Finite
[Isogeny of degree 19 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 18*x + 18 over Finite
[Isogeny of degree 23 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 22*x + 22 over Finite
[Isogeny of degree 29 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 28*x + 28 over Finite
[Isogeny of degree 31 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 30*x + 30 over Finite
[Isogeny of degree 37 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 36*x + 36 over Finite
[Isogeny of degree 41 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 40*x + 40 over Finite
[Isogeny of degree 43 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 42*x + 42 over Finite
[Isogeny of degree 47 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 46*x + 46 over Finite
```

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) # not tested (very long time: 32s)
[]
```

sage.schemes.elliptic_curves.isogeny_small_degree.**isogenies_prime_degree_general**(*E*, *l*)

Returns a list of `l` -isogenies with domain `E`.

INPUT:

- `E` – an elliptic curve.
- `l` – a prime.

OUTPUT:

(list) a list of all isogenies of degree l.

ALGORITHM:

This algorithm factors the `l`-division polynomial, then combines its factors to otain kernels. See [KT2013], Chapter 3.

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_gener
sage: E = EllipticCurve_from_j(GF(2^6,'a')(1))
sage: isogenies_prime_degree_general(E, 7)
[Isogeny of degree 7 from Elliptic Curve defined by y^2 + x*y = x^3 + 1 over Finite Field in a c
sage: E = EllipticCurve_from_j(GF(3^12,'a')(2))
```

```
sage: isogenies_prime_degree_general(E, 17)
[Isogeny of degree 17 from Elliptic Curve defined by y^2 = x^3 + 2*x^2 + 2 over Finite Field in
sage: E = EllipticCurve('50a1')
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 Fi
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 Fi
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 Rationa
sage: E = EllipticCurve([0, -1, 0, -6288, 211072])
sage: isogenies_prime_degree_general(E, 37)  # long time (10s)
[Isogeny of degree 37 from Elliptic Curve defined by y^2 = x^3 - x^2 - 6288*x + 211072 over Rati

sage: E = EllipticCurve([-3440, 77658])
sage: isogenies_prime_degree_general(E, 43)  # long time (16s)
[Isogeny of degree 43 from Elliptic Curve defined by y^2 = x^3 - 3440*x + 77658 over Rational Fi
```

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_gener
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 Fi
[Isogeny of degree 5 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 4*x + 4 over Finite Fi
[Isogeny of degree 7 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 6*x + 6 over Finite Fi
[Isogeny of degree 11 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 10*x + 10 over Finite
[Isogeny of degree 13 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 12*x + 12 over Finite
[Isogeny of degree 17 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 16*x + 16 over Finite
[Isogeny of degree 19 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 18*x + 18 over Finite
[Isogeny of degree 23 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 22*x + 22 over Finite
[Isogeny of degree 29 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 28*x + 28 over Finite
[Isogeny of degree 31 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 30*x + 30 over Finite
[Isogeny of degree 37 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 36*x + 36 over Finite
[Isogeny of degree 41 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 40*x + 40 over Finite
[Isogeny of degree 43 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 42*x + 42 over Finite
[Isogeny of degree 47 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + 46*x + 46 over Finite
```

sage.schemes.elliptic_curves.isogeny_small_degree.**isogenies_prime_degree_genus_0**(*E*, *l=None*)

Returns list of `l` -isogenies with domain `E`.

INPUT:

- `E` – an elliptic curve.
- `l` – either None or 2, 3, 5, 7, or 13.

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
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 Ra

sage: E = EllipticCurve('50a1')
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 Fi
Isogeny of degree 5 from Elliptic Curve defined by y^2 + x*y + y = x^3 - x - 2 over Rational Fie
```

sage.schemes.elliptic_curves.isogeny_small_degree.**isogenies_prime_degree_genus_plus_0**(*E*, *l=Non*

Returns 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.

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

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

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 R

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 Rationa

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 5/14073748

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+51401307

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 from Elliptic Curve defined by y^2 = x^3 + (414634512218543303467795660800

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

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 (4.3s)
[Isogeny of degree 47 from Elliptic Curve defined by y^2 = x^3 + (454562028554080355857852049849

sage: K = QuadraticField(-66827,'a')
sage: a = K.gen()
sage: E = EllipticCurve_from_j(-98669236224000*a + 4401720074240000)
sage: isogenies_prime_degree_genus_plus_0(E, 59)   # long time (25s, 2012)
[Isogeny of degree 59 from Elliptic Curve defined by y^2 = x^3 + (260588614678214476229797478400

sage: E = EllipticCurve_from_j(GF(13)(5))
sage: isogenies_prime_degree_genus_plus_0(E, 71) # long time
[Isogeny of degree 71 from Elliptic Curve defined by y^2 = x^3 + x + 4 over Finite Field of size

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 o
Isogeny of degree 17 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + x over Finite Field of
Isogeny of degree 29 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + x over Finite Field of
Isogeny of degree 29 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + x over Finite Field of
Isogeny of degree 41 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + x over Finite Field of
Isogeny of degree 41 from Elliptic Curve defined by y^2 + y = x^3 + x^2 + x over Finite Field of
```

sage.schemes.elliptic_curves.isogeny_small_degree.**isogenies_prime_degree_genus_plus_0_j0**(*E*, *l*)

Returns 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.

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

sage: u = polygen(QQ)
sage: K.<a> = NumberField(u^4+228*u^3+486*u^2-540*u+225)
sage: E = EllipticCurve(K,[0,-121/5*a^3-20691/5*a^2-29403/5*a+3267])
sage: isogenies_prime_degree_genus_plus_0_j0(E,11)
[Isogeny of degree 11 from Elliptic Curve defined by y^2 = x^3 + (-121/5*a^3-20691/5*a^2-29403/5

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 siz
```

sage.schemes.elliptic_curves.isogeny_small_degree.**isogenies_prime_degree_genus_plus_0_j1728**

Returns 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.

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

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+17086861/24
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+13066735/44
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 from Elliptic Curve defined by y^2 = x^3 + (-2*i-1)*x over Number Field in
Isogeny of degree 17 from Elliptic Curve defined by y^2 = x^3 + (-2*i-1)*x over Number Field in
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*)

Returns list of `l` -isogenies with domain `E` (defined over $\mathbf{Q}$).

Returns 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
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

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 R
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 R
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 Rationa
sage: isogenies_sporadic_Q(E)
[Isogeny of degree 19 from Elliptic Curve defined by y^2 + y = x^3 - 1862*x - 30956 over Rationa

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 Rati

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 + 486169

sage: E = EllipticCurve([-3440, 77658])
sage: E.conductor()
118336
```

```
sage: isogenies_sporadic_Q(E,43)
[Isogeny of degree 43 from Elliptic Curve defined by y^2 = x^3 - 3440*x + 77658 over Rational Fi

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

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
```

CHAPTER

TWENTYSEVEN

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.

EXAMPLE:

```
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:

- [BMSS] Boston, Morain, Salvy, Schost, "Fast Algorithms for Isogenies."

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 trac ticket #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 [BMSS].

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 coeffients 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 [BMSS].

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*)

Computes 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 aproximating the Weierstrass $\wp$-function to precision prec.

ALGORITHM: This function uses the algorithm described in section 3.2 of [BMSS].

REFERENCES: [BMSS] Boston, Morain, Salvy, Schost, "Fast Algorithms for Isogenies."

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

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*)

Computes the Weierstrass $\wp$-function on an elliptic curve.

INPUT:

- `E` – an elliptic curve
- `prec` – precision
- `algorithm` – string (default:`None`) an algorithm identifier indicating the `pari`, `fast` or `quadratic` algorithm. If the algorithm is `None`, then this function determines 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
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 c
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 characteri
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 un
```

TESTS:

```
sage: E.weierstrass_p(prec=4, algorithm='foo')
Traceback (most recent call last):
...
ValueError: unknown algorithm for computing the Weierstrass p-function
```

CHAPTER

TWENTYEIGHT

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: K.<a> = NumberField(x^3-2)
sage: E = EllipticCurve([0,1,0,a,a])
```

First we try a real embedding:

```
sage: emb = K.embeddings(RealField())[0]
sage: L = E.period_lattice(emb); L
Period lattice associated to Elliptic Curve defined by y^2 = x^3 + x^2 + a*x + a over Number Field in
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()
(3.81452977217855, 1.90726488608927 + 1.34047785962440*I)
sage: L.is_real()
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 function `normalised_basis()` instead:

```
sage: L.normalised_basis()
(1.90726488608927 - 1.34047785962440*I, -1.90726488608927 - 1.34047785962440*I)
```

Next a complex embedding:

```
sage: emb = K.embeddings(ComplexField())[0]
sage: L = E.period_lattice(emb); L
Period lattice associated to Elliptic Curve defined by y^2 = x^3 + x^2 + a*x + a over Number Field in
```

```
From: Number Field in a with defining polynomial x^3 - 2
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: 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 #8415 (caused by a PARI bug fixed in v2.3.5) is OK:

```
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
  From: Number Field in a with defining polynomial x^2 + 7
  To:   Algebraic Field
  Defn: a |--> -2.645751311064591?*I
```

REFERENCES:

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*)

Bases: sage.modules.free_module.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: sage.schemes.elliptic_curves.period_lattice.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 function `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 trac #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 trac #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: 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: emb = K.embeddings(ComplexField())[0]
sage: L = E.period_lattice(emb)
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 None) – if True and the embedding is real, use the normalised basis (see `normalised_basis()`) instead of the default.

OUTPUT:

A 2x2 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: 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  1.340477859624402021]
```

See #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: 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*)

Returns 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 :meth:.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')
(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: 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])
sage: L.curve() is E
True
```

e_log_RC(*xP*, *yP*, *prec=None*, *reduce=True*)

Return the elliptic logarithm of a real or complex point.

- 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 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 [CT] 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: 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: 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)
3.76298229503967
sage: xP, yP = ER.lift_x(0).xy()
sage: L.e_log_RC(xP, yP)
2.69842609082114
```

Elliptic logs of complex points:

```
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)
1.03355715602040 - 0.867257428417356*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()
[-1.107159871688768?, 0.2695944364054446?, 0.8375654352833230?]

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.ei()
[0.?e-17 - 1.122462048309373?*I, 0.?e-17 + 1.122462048309373?*I, -1]
```

sage: L = E.period_lattice(K.embeddings(ComplexField())[0]) sage: L.ei() [-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 $\mathbf{R}$ or $\mathbf{C}$ 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
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 + 1.00000000
sage: L.elliptic_exponential(z,to_curve=False)
(1.41666666666667, -1.00000000000000)
sage: z = L(P,prec=201); z
1.17044757240089592298992188482371493504472561677451007994189
sage: L.elliptic_exponential(z)
(1.00000000000000000000000000000000000000000000000000000000000000000000000 : -2.00000000000000000000000
```

Examples over number fields:

```
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: 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 : 1.00000000
sage: [embs[0](c) for c in Q]
[-1.62996052494744 - 1.09112363597172*I, 1.79370052598410 - 1.37472963699860*I, 1.0000000000
sage: Li[1].elliptic_exponential(zi[1])
(-1.62996052494744 + 1.09112363597172*I : 1.79370052598410 + 1.37472963699860*I : 1.00000000
sage: [embs[1](c) for c in Q]
[-1.62996052494744 + 1.09112363597172*I, 1.79370052598410 + 1.37472963699860*I, 1.0000000000
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 #8820 is fixed:

```
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 : 1.0000
sage: L.elliptic_exponential(CDF(.1,.1), to_curve=False)
(0.0000142854026029... - 49.9960001066650*I, 250.020141250950 + 250.019855549131*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: 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 + (-1.00
```

Very small z are handled properly (see #8820):

```
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()
True
sage: L.elliptic_exponential((RealField(420)(1e-100))).is_zero()
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 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 [CT] 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 + 0.98586885077582410221120384 91*I
sage: Q=E([3,5])
sage: Q.is_on_identity_component()
True
sage: L.elliptic_logarithm(Q, prec=96)
1.9311282715425594424885852 20
```

Note that this is actually the inverse of the Weierstrass isomorphism:

```
sage: L.elliptic_exponential(_)
(3.0000000000000000000000000... : 5.0000000000000000000000000... : 1.0000000000000000000000
```

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])
sage: L = E.period_lattice()
sage: L.ei()
[5334.003952567705? - 1.964393150436?e-6*I, 5334.003952567705? + 1.964393150436?e-6*I, -1066
sage: L.elliptic_logarithm(P,prec=100)
0.27656204014107061464076203097
```

Some complex examples, taken from the paper by Cremona and Thongjunthug:

```
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
0.70448375537782208460499649302 - 0.79246725643650979858266018068*I
sage: L.coordinates(z)
(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
0.57002153834710752778063503023 + 0.46476340520469798857457031393*I
sage: L.coordinates(z)
(0.46247636364807931766105406092, -0.20502411273191295799239604171)
```

The elliptic logs of the 2-torsion points are half-periods:

```
sage: L.elliptic_logarithm(E(e1,0),prec=100)
0.64607575874356525952487867052 + 0.22379609053909448304176885364*I
sage: L.elliptic_logarithm(E(e2,0),prec=100)
0.71330686725892253793705940192 - 0.40481924028150941053684639367*I
sage: L.elliptic_logarithm(E(e3,0),prec=100)
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))
(1.0000000000000000000000000000, 0.0000000000000000000000000000)
sage: L.coordinates(2*L.elliptic_logarithm(E(e2,0),prec=100))
(1.0000000000000000000000000000, 1.0000000000000000000000000000)
sage: L.coordinates(2*L.elliptic_logarithm(E(e3,0),prec=100))
(0.0000000000000000000000000000, 1.0000000000000000000000000000)

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)
```

TESTS (see #10026 and #11767):

```
sage: K.<w> = QuadraticField(2)
sage: E = EllipticCurve([ 0, -1, 1, -3*w -4, 3*w + 4 ])
sage: T = E.simon_two_descent(lim1=20,lim3=5,limtriv=20)
sage: P,Q = T[2]
sage: embs = K.embeddings(CC)
sage: Lambda = E.period_lattice(embs[0])
sage: Lambda.elliptic_logarithm(P,100)
4.7100131126199672766973600998
sage: R.<x> = QQ[]
sage: K.<a> = NumberField(x^2 + x + 5)
sage: E = EllipticCurve(K, [0,0,1,-3,-5])
sage: P = E([0,a])
sage: Lambda = P.curve().period_lattice(K.embeddings(ComplexField(600))[0])
sage: Lambda.elliptic_logarithm(P, prec=600)
-0.84224816648773939337501800838169399080058886406950618703387318384524623354805847756170640
sage: K.<a> = QuadraticField(-5)
```

```
sage: E = EllipticCurve([1,1,a,a,0])
sage: P = E(0,0)
sage: L = P.curve().period_lattice(K.embeddings(ComplexField())[0])
sage: L.elliptic_logarithm(P, prec=500)
1.17058357737548897849026170185581196033579563441850967539191867385734983296504066660506637
sage: L.elliptic_logarithm(P, prec=1000)
1.17058357737548897849026170185581196033579563441850967539191867385734983296504066660506637
```

is_real()

Return True if this period lattice is real.

EXAMPLES:

```
sage: f = EllipticCurve('11a')
sage: f.period_lattice().is_real()
True

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: K.<a> = NumberField(x^3-2)
sage: E = EllipticCurve([0,1,0,a,a])
sage: [E.period_lattice(emb).is_real() for emb in K.embeddings(CC)]
[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 function basis() instead.

EXAMPLES:

```
sage: E = EllipticCurve('37a')
sage: E.period_lattice().normalised_basis()
(2.99345864623196, -2.45138938198679*I)

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: 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*)

Returns the real or complex volume of this period lattice.

INPUT:

- `prec` (int or `None''(default)) -- real precision in bits (default real precision if ''None`)

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.

Note: If the curve is defined over $\mathbf{Q}$ and is given by a *minimal* Weierstrass equation, then this is the correct period in the BSD conjecture, i.e., it is the least real period * 2 when the period lattice is rectangular. More generally the product of this quantity over all embeddings appears in the generalised BSD formula.

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 nonsense. The following works though:

```
sage: F = E.minimal_model()
sage: F.period_lattice().omega()
0.972218771420113
```

```
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):

```
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
```

real_period(*prec=None*, *algorithm='sage'*)

Returns the real period of this period lattice.

INPUT:

- `prec` (int 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: 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*)

Returns 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: 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: 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: 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)
```

TESTS:

```
sage: extended_agm_iteration(1,2,3)
Traceback (most recent call last):
...
ValueError: values must be real or complex numbers
```

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` (complex) – 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: 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: 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])
```

CHAPTER

TWENTYNINE

REGIONS IN FUNDAMENTAL DOMAINS OF PERIOD LATTICES

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 [TT].

AUTHORS:

- Robert Bradshaw (2010): initial version
- John Cremona (2014): added some docstrings and doctests

REFERENCES:

class sage.schemes.elliptic_curves.period_lattice_region.**PeriodicRegion**
Bases: object

EXAMPLE:

```
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()
sage: data = np.zeros((4, 4))
sage: data[1,1] = True
sage: S = PeriodicRegion(CDF(2), CDF(2*I+1), data)
sage: S.plot()
```

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()
sage: S.contract().plot()
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()
sage: S.expand().plot()
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()
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()

Returns 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())
```

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*)

Plots 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)
```

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 $\mathbb{C}$.

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()<0.5
sage: S.verify(condition)
False
sage: condition = lambda z: z.real().abs()<1
sage: S.verify(condition)
True
```

w1

w2

CHAPTER

THIRTY

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: sage.structure.sage_object.SageObject

The formal group associated to an elliptic curve.

curve()
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*)
Returns 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

DETAILS: 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 [Silverman AEC1].

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)
```

AUTHOR:

- David Harvey (2006-09-10): factored out of log

group_law(*prec=10*)

The formal group law.

INPUT:

- `prec` - integer (default 10)

OUTPUT: a power series with given precision in R[['t1','t2']], where the curve is defined over R.

DETAILS: Return the formal power series

$$F(t_1, t_2) = t_1 + t_2 - a_1 t_1 t_2 - \cdots$$

to precision $O(t1, t2)^{prec}$ of page 115 of [Silverman AEC1].

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
```

TESTS:

```
sage: R.<x,y,z> = GF(7)[[]]
sage: F(x, ehat.inverse()(x))
0 + O(x, y, z)^7
sage: F(x, y) == F(y, x)
True
sage: F(x, F(y, z)) == F(F(x, y), z)
True
```

Let's ensure caching with changed precision is working:

```
sage: e.formal_group().group_law(4)
t1 + t2 + O(t1, t2)^4
```

Test for trac ticket #9646:

```
sage: P.<a1, a2, a3, a4, a6> = PolynomialRing(ZZ, 5)
sage: E = EllipticCurve(list(P.gens()))
sage: F = E.formal().group_law(prec=5)
sage: t1, t2 = F.parent().gens()
sage: F(t1, 0)
t1 + O(t1, t2)^5
sage: F(0, t2)
t2 + O(t1, t2)^5
sage: F.coefficients()[t1*t2^2]
-a2
```

inverse (*prec=20*)

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

DETAILS: Return the formal power series

$$i(t) = -t + a_1 t^2 + \cdots$$

to precision $O(t^{prec})$ of page 114 of [Silverman AEC1].

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*)

Returns 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*)

The formal 'multiplication by n' endomorphism $[n]$.

INPUT:

•`prec` - integer (default 10)

OUTPUT: a power series with given precision

DETAILS: Return the formal power series

$$[n](t) = nt + \cdots$$

to precision $O(t^{prec})$ of Proposition 2.3 of [Silverman AEC1].

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 + 13333332
```

TESTS:

```
sage: F = EllipticCurve(GF(17), [1, 1]).formal_group()
sage: F.mult_by_n(10, 50)  # long time (13s on sage.math, 2011)
10*t + 5*t^5 + 7*t^7 + 13*t^9 + t^11 + 16*t^13 + 13*t^15 + 9*t^17 + 16*t^19 + 15*t^23 + 15*t
```

```
sage: F = EllipticCurve(GF(101), [1, 1]).formal_group()
sage: F.mult_by_n(100, 20)
100*t + O(t^20)

sage: P.<a1, a2, a3, a4, a6> = PolynomialRing(ZZ, 5)
sage: E = EllipticCurve(list(P.gens()))
sage: E.formal().mult_by_n(2,prec=5)
2*t - a1*t^2 - 2*a2*t^3 + (a1*a2 - 7*a3)*t^4 + O(t^5)

sage: E = EllipticCurve(QQ, [1,2,3,4,6])
sage: E.formal().mult_by_n(2,prec=5)
2*t - t^2 - 4*t^3 - 19*t^4 + O(t^5)
```

sigma(*prec=10*)

EXAMPLE:

```
sage: E = EllipticCurve('14a')
sage: F = E.formal_group()
sage: F.sigma(5)
t + 1/2*t^2 + (1/2*c + 1/3)*t^3 + (3/4*c + 3/4)*t^4 + O(t^5)
```

w(*prec=20*)

The formal group power series w.

INPUT:

- `prec` - integer (default 20)

OUTPUT: a power series with given precision

DETAILS: 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 [Silverman AEC1]. This is the formal expansion of $w = -1/y$ about the formal parameter $t = -x/y$ at
$infty$.

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.

AUTHOR:

- 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
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
```

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

DETAILS: 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 [Silverman AEC1].

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

DETAILS: 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 [Silverman AEC1].

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)
```

CHAPTER

THIRTYONE

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 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 [Sil2] for more details.

REFERENCES :

- **[Sil2] Silverman Joseph, Advanced Topics in the Arithmetic of Elliptic Curves,** GTM 151, Springer 1994.

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: sage.structure.sage_object.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 ov
sage: eq == loads(dumps(eq))
True
```

REFERENCES :

•[Sil2] Silverman Joseph, Advanced Topics in the Arithmetic of Elliptic Curves, GTM 151, Springer 1994.

E2 (*prec=20*)

Returns 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^
```

L_invariant (*prec=20*)

Returns 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 [MTT]. See [Col] for a detailed discussion.

INPUT:

- `prec` - the p-adic precision, default is 20.

REFERENCES:

- [MTT] B. Mazur, J. Tate, and J. Teitelbaum, On p-adic analogues of the conjectures of Birch and Swinnerton-Dyer, Inventiones mathematicae 84, (1986), 1-48.
- [Col] Pierre Colmez, Invariant $\mathcal{L}$ et derivees de valeurs propores de Frobenius, preprint, 2004.

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*)

Returns 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 ()

Returns 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^
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^
```

original_curve()

Returns 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*)

Returns 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(5^8)
```

Check that it is a quadratic function:

```
sage: h(3*P)-3^2*h(P)
O(5^8)
```

padic_regulator(*prec=20*)

Computes the canonical p-adic regulator on the extended Mordell-Weil group as in [MTT] (with the correction of [Wer] and sign convention in [SW].) 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.

REFERENCES:

- [MTT] B. Mazur, J. Tate, and J. Teitelbaum, On p-adic analogues of the conjectures of Birch and Swinnerton-Dyer, Inventiones mathematicae 84, (1986), 1-48.
- [Wer] Annette Werner, Local heights on abelian varieties and rigid analytic unifomization, Doc. Math. 3 (1998), 301-319.
- [SW] William Stein and Christian Wuthrich, Computations About Tate-Shafarevich Groups using Iwasawa theory, preprint 2009.

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 +
```

parameter(*prec=20*)

Returns 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=20*)

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 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(5^20))
```

Here is how one gets a 4-torsion point on E over $\mathbf{Q}_5$:

```
sage: R = Qp(5,10)
sage: i = R(-1).sqrt()
sage: T = eq.parametrisation_onto_original_curve(i); T
(2 + 3*5 + 4*5^2 + 2*5^3 + 5^4 + 4*5^5 + 2*5^7 + 5^8 + 5^9 + O(5^10) :
3*5 + 5^2 + 5^4 + 3*5^5 + 3*5^7 + 2*5^8 + 4*5^9 + O(5^10) : 1 + O(5^20))
sage: 4*T
(0 : 1 + O(5^20) : 0)
```

parametrisation_onto_tate_curve(*u*, *prec=20*)

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 20.

EXAMPLES:

```
sage: eq = EllipticCurve('130a1').tate_curve(5)
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(5^20))
```

prime()

Returns 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
```

CHAPTER

THIRTYTWO

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 ($0 \leq \eta \leq p-2$) 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 [BP], which has coefficients in the Dieudonne module $D_pE = H^1_{dR}(E/\mathbf{Q}_p)$ of E. There is a different description by Pollack [Po] which is not available here.

According to the p-adic version of the Birch and Swinnerton-Dyer conjecture [MTT], 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 [SW] for more details.

REFERENCES:

- [MTT] B. Mazur, J. Tate, and J. Teitelbaum, On p-adic analogues of the conjectures of Birch and Swinnerton-Dyer, Inventiones mathematicae 84, (1986), 1-48.
- [BP] Dominique Bernardi and Bernadette Perrin-Riou, Variante p-adique de la conjecture de Birch et Swinnerton-Dyer (le cas supersingulier), C. R. Acad. Sci. Paris, Ser I. Math, 317 (1993), no 3, 227-232.
- [Po] Robert Pollack, On the p-adic L-function of a modular form at supersingular prime, Duke Math. J. 118 (2003), no 3, 523-558.
- [SW] William Stein and Christian Wuthrich, Computations About Tate-Shafarevich Groups using Iwasawa theory, preprint 2009.

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*, *use_eclib=True*, *normalize='L_ratio'*)

Bases: sage.structure.sage_object.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(
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)
6 + 2*7^3 + 5*7^4 + O(7^6) + (4*7 + 2*7^2 + O(7^3))*T + (2 + 3*7^2 + O(7^3))*T^2 + (1 + 2*7 + 2*
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
sage: lp.series(4,eta=3)
O(7^6) + (3 + 2*7 + 5*7^2 + O(7^3))*T + (5 + 4*7 + 5*7^2 + O(7^3))*T^2 + (3*7 + 7^2 + O(7^3))*T^
```

(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 $\mathbf{Q}(\zeta_7)$.)

the load-dumps test:

```
sage: lp = EllipticCurve('11a').padic_lseries(5)
sage: lp == loads(dumps(lp))
True
```

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
(1 + O(3^10))*alpha
sage: alpha^2 - E.ap(3)*alpha + 3
(O(3^10))*alpha^2 + (O(3^11))*alpha + (O(3^11))
```

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 then finding the modular symbols for the twist. Quadratic twists are only implemented if the sign is $+1$.

Note that the normalisation 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

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 normalisation 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, 0, -1]
sage: [lp.modular_symbol(r,quadratic_twist=-20) for r in [0,1/5,oo,1/11]]
[2, 2, 0, 1]

sage: lpt = E.quadratic_twist(-3).padic_lseries(5)
sage: et = E.padic_lseries(5)._quotient_of_periods_to_twist(-3)
sage: lpt.modular_symbol(0) == lp.modular_symbol(0,quadratic_twist=-3)/et
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 [Ka].

NOTE: currently p must be a prime of good ordinary reduction.

REFERENCES:

- [MTT] B. Mazur, J. Tate, and J. Teitelbaum, On p-adic analogues of the conjectures of Birch and Swinnerton-Dyer, Inventiones mathematicae 84, (1986), 1-48.
- [Ka] Kayuza Kato, p-adic Hodge theory and values of zeta functions of modular forms, Cohomologies p-adiques et applications arithmetiques III, Asterisque vol 295, SMF, Paris, 2004.

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()
2
sage: L = EllipticCurve('5077a').padic_lseries(5, use_eclib=True)
sage: L.order_of_vanishing()
3
```

prime()

Returns 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*, *use_eclib=True*, *normalize='L_ratio'*)

Bases: sage.schemes.elliptic_curves.padic_lseries.pAdicLseries

INPUT:

- E - an elliptic curve
- p - a prime of good reduction
- use_eclib - bool (default:True); whether or not to use John Cremona's eclib for the computation of modular symbols
- normalize - 'L_ratio' (default), 'period' or 'none'; this is describes the way the modular symbols are normalized. See modular_symbol of an elliptic curve over Q for more details.

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: Lp = E.padic_lseries(3)
sage: Lp.series(2,prec=3)
2 + 3 + 3^2 + 2*3^3 + O(3^4) + (1 + O(3))*T + (1 + O(3))*T^2 + O(T^3)
```

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*)

Returns 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 Neron 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 $\mathbf{Z}_p^\times$)

ALIAS: 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)
```

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)
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 (:trac: 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^
```

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` can not determine this.

We calculate the L-series in the nontrivial Teichmueller components:

```
sage: L = EllipticCurve('110a1').padic_lseries(5)
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))*
3 + 2*5 + 2*5^3 + 3*5^4 + O(5^6) + (2 + 5 + 4*5^2 + O(5^3))*T + (1 + 4*5 + 2*5^2 + O(5^3))*T
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))*
1 + 3*5 + 4*5^2 + 2*5^3 + 5^4 + 4*5^5 + O(5^6) + (2 + 4*5 + 3*5^2 + O(5^3))*T + (2 + 3*5 + 5
```

series (*n=2*, *quadratic_twist=1*, *prec=5*, *eta=0*)

Returns 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_pE and Ω_E is the Neron 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 $\mathbf{Z}_p^\times$)

ALIAS: 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)
```

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)
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 (:trac: 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^
```

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` can not determine this.

We calculate the L-series in the nontrivial Teichmueller components:

```
sage: L = EllipticCurve('110a1').padic_lseries(5)
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))*
3 + 2*5 + 2*5^3 + 3*5^4 + O(5^6) + (2 + 5 + 4*5^2 + O(5^3))*T + (1 + 4*5 + 2*5^2 + O(5^3))*T
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))*
1 + 3*5 + 4*5^2 + 2*5^3 + 5^4 + 4*5^5 + O(5^6) + (2 + 4*5 + 3*5^2 + O(5^3))*T + (2 + 3*5 + 5
```

class sage.schemes.elliptic_curves.padic_lseries.**pAdicLseriesSupersingular**(*E*, *p*, *use_eclib=True*, *normalize='L_ratio'*)

Bases: sage.schemes.elliptic_curves.padic_lseries.pAdicLseries

INPUT:

- `E` - an elliptic curve
- `p` - a prime of good reduction
- `use_eclib` - bool (default:True); whether or not to use John Cremona's `eclib` for the computation of modular symbols
- `normalize` - `'L_ratio'` (default), `'period'` or `'none'`; this is describes the way the modular symbols are normalized. See `modular_symbol` of an elliptic curve over Q for more details.

EXAMPLES:

```
sage: E = EllipticCurve('11a1')
sage: Lp = E.padic_lseries(3)
sage: Lp.series(2,prec=3)
2 + 3 + 3^2 + 2*3^3 + O(3^4) + (1 + O(3))*T + (1 + O(3))*T^2 + O(T^3)
```

Dp_valued_height (*prec=20*)

Returns the canonical p-adic height with values in the Dieudonne 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*)

Returns the canonical p-adic regulator with values in the Dieudonne 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 [PR]. See [SW].

REFERENCES:

- [PR] Perrin Riou, Arithmetique des courbes elliptiques a reduction supersinguliere en p, Experiment. Math. 12 (2003), no. 2, 155-186.
- [SW] William Stein and Christian Wuthrich, Computations About Tate-Shafarevich Groups using Iwasawa theory, preprint 2009.

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
```

Dp_valued_series (*n=3*, *quadratic_twist=1*, *prec=5*)

Returns a vector of two components which are p-adic power series. The answer v is such that

$$(1-\varphi)^{-2} \cdot L_p(E,T) = \texttt{v[1]} \cdot\omega + \texttt{v[2]} \cdot\varphi(\omega)$$

as an element of the Dieudonne 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 [BP] 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

REFERENCE:

- [BP] Dominique Bernardi and Bernadette Perrin-Riou, Variante p-adique de la conjecture de Birch et Swinnerton-Dyer (le cas supersingulier), C. R. Acad. Sci. Paris, Ser I. Math, 317 (1993), no 3, 227-232.

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 + 4*5^3 + O(5^4)) + (4 + O(5))*T + (1 + O(5))*T^2 + (4 + O(5))*T^3 + (2 + O(5))*T^4
```

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'*)

This returns a geometric Frobenius φ on the Diedonne 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-Washintzer 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 probably 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_pE and Ω_E is the Neron 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 $\mathbf{Z}_p^\times$)

ALIAS: power_series is identical to series.

EXAMPLES: A superingular 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(3))*alpha + (O(3^2)) + ((O(3^-1))*alpha + (2*3^-1 + O(3^0)))*T + ((O(3^-1))*alpha + (2*3^
sage: L.alpha(2).parent()
Univariate Quotient Polynomial Ring in alpha over 3-adic Field with capped
relative precision 2 with modulus (1 + O(3^2))*x^2 + (3 + O(3^3))*x + (3 + O(3^3))
```

An example where we only compute the leading term (:trac: 15737):

```
sage: E = EllipticCurve("17a1")
sage: L = E.padic_lseries(3)
sage: L.series(4,prec=1)
(O(3^18))*alpha^2 + (2*3^-1 + 1 + 3 + 3^2 + 3^3 + ... + 3^18 + O(3^19))*alpha + (2*3^-1 + 1
```

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 probably 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_pE and Ω_E is the Neron 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 $\mathbf{Z}_p^\times$)

ALIAS: power_series is identical to series.

EXAMPLES: A superingular 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(3))*alpha + (O(3^2)) + ((O(3^-1))*alpha + (2*3^-1 + O(3^0)))*T + ((O(3^-1))*alpha + (2*3^
sage: L.alpha(2).parent()
Univariate Quotient Polynomial Ring in alpha over 3-adic Field with capped
relative precision 2 with modulus (1 + O(3^2))*x^2 + (3 + O(3^3))*x + (3 + O(3^3))
```

An example where we only compute the leading term (:trac: 15737):

```
sage: E = EllipticCurve("17a1")
sage: L = E.padic_lseries(3)
sage: L.series(4,prec=1)
(O(3^18))*alpha^2 + (2*3^-1 + 1 + 3 + 3^2 + 3^3 + ... + 3^18 + O(3^19))*alpha + (2*3^-1 + 1
```

CHAPTER

THIRTYTHREE

MODULAR SYMBOLS

To an elliptic curves E over the rational numbers one can associate a space - or better two spaces - of modular symbols of level N, equal to the conductor of E; because E is known to be modular.

There are two implementations of modular symbols, one within `sage` and the other as part of Cremona's `eclib`. One can choose here which one is used.

The normalisation of our modular symbols attached to E can be chosen, too. For instance one can make it depended on E rather than on its isogeny class. This is useful for p-adic L-functions.

For more details on modular symbols consult the following

REFERENCES:

- [MTT] B. Mazur, J. Tate, and J. Teitelbaum, On p-adic analogues of the conjectures of Birch and Swinnerton-Dyer, Inventiones mathematicae 84, (1986), 1-48.
- [Cre] John Cremona, Algorithms for modular elliptic curves, Cambridge University Press, 1997.
- [SW] William Stein and Christian Wuthrich, Computations About Tate-Shafarevich Groups using Iwasawa theory, preprint 2009.

AUTHORS:

- William Stein (2007): first version
- Chris Wuthrich (2008): add scaling and reference to eclib

class `sage.schemes.elliptic_curves.ell_modular_symbols.`**`ModularSymbol`**
Bases: `sage.structure.sage_object.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 from r to ∞.

This is as defined in [MTT], but normalized to depend on the curve and not only its isogeny class as in [SW].

See the documentation of `E.modular_symbol()` in Elliptic curves over the rational numbers for help.

REFERENCES:

•[MTT] B. Mazur, J. Tate, and J. Teitelbaum, On p-adic analogues of the conjectures of Birch and Swinnerton-Dyer, Inventiones mathematicae 84, (1986), 1-48.

•[SW] William Stein and Christian Wuthrich, Computations About Tate-Shafarevich Groups using Iwasawa theory, preprint 2009.

`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)
sage: m.sign()
-1
```

class sage.schemes.elliptic_curves.ell_modular_symbols.**ModularSymbolECLIB**(*E*, *sign*, *normalize='L_ratio'*)

Bases: sage.schemes.elliptic_curves.ell_modular_symbols.ModularSymbol

Modular symbols attached to E using `eclib`.

INPUT:

- `E` - an elliptic curve
- `sign` - an integer, -1 or 1
- `normalize` - either 'L_ratio' (default) 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. 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.

EXAMPLES:

```
sage: import sage.schemes.elliptic_curves.ell_modular_symbols
sage: E=EllipticCurve('11a1')
sage: M=sage.schemes.elliptic_curves.ell_modular_symbols.ModularSymbolECLIB(E,+1)
sage: M
Modular symbol with sign 1 over Rational Field attached to Elliptic Curve defined by y^2 + y = x
sage: M(0)
1/5
sage: E=EllipticCurve('11a2')
sage: M=sage.schemes.elliptic_curves.ell_modular_symbols.ModularSymbolECLIB(E,+1)
sage: M(0)
1
```

This is a rank 1 case with vanishing positive twists. The modular symbol can not be adjusted:

```
sage: E=EllipticCurve('121b1')
sage: M=sage.schemes.elliptic_curves.ell_modular_symbols.ModularSymbolECLIB(E,+1)
Warning : Could not normalize the modular symbols, maybe all further results will be multiplied
sage: M(0)
0
sage: M(1/7)
-2

sage: M = EllipticCurve('121d1').modular_symbol(use_eclib=True)
sage: M(0)
2
sage: M = EllipticCurve('121d1').modular_symbol(use_eclib=True,normalize='none')
sage: M(0)
8

sage: E = EllipticCurve('15a1')
sage: [C.modular_symbol(use_eclib=True,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(use_eclib=True,normalize='none')(0) for C in E.isogeny_class()]
[1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4, 1/4]
```

Currently, the interface for negative modular symbols in eclib is not yet written:

```
sage: E.modular_symbol(use_eclib=True,sign=-1)
Traceback (most recent call last):
...
NotImplementedError: Despite that eclib has now -1 modular symbols the interface to them is not
```

TESTS (for trac 10236):

```
sage: E = EllipticCurve('11a1')
sage: m = E.modular_symbol(use_eclib=True)
sage: m(1/7)
7/10
sage: m(0)
1/5
```

class sage.schemes.elliptic_curves.ell_modular_symbols.**ModularSymbolSage**(*E*, *sign*, *normalize='L_ratio'*)

Bases: sage.schemes.elliptic_curves.ell_modular_symbols.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: import sage.schemes.elliptic_curves.ell_modular_symbols
sage: M=sage.schemes.elliptic_curves.ell_modular_symbols.ModularSymbolSage(E,+1)
sage: M
Modular symbol with sign 1 over Rational Field attached to Elliptic Curve defined by y^2 + y = x
sage: M(0)
1/5
sage: E=EllipticCurve('11a2')
sage: M=sage.schemes.elliptic_curves.ell_modular_symbols.ModularSymbolSage(E,+1)
sage: M(0)
1
sage: M=sage.schemes.elliptic_curves.ell_modular_symbols.ModularSymbolSage(E,-1)
sage: M(1/3)
1
```

This is a rank 1 case with vanishing positive twists. The modular symbol is adjusted by -2:

```
sage: E=EllipticCurve('121b1')
sage: M=sage.schemes.elliptic_curves.ell_modular_symbols.ModularSymbolSage(E,-1,normalize='L_rat
sage: M(1/3)
2
sage: M._scaling
-2

sage: M = EllipticCurve('121d1').modular_symbol(use_eclib=False)
sage: M(0)
2
sage: M = EllipticCurve('121d1').modular_symbol(use_eclib=False,normalize='none')
sage: M(0)
1

sage: E = EllipticCurve('15a1')
sage: [C.modular_symbol(use_eclib=False, 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(use_eclib=False, 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(use_eclib=False, 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 elliptic curves.

INPUT:

- `E` - an elliptic curve over $\mathbf{Q}$
- `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: import sage.schemes.elliptic_curves.ell_modular_symbols
sage: E=EllipticCurve('11a1')
sage: M=sage.schemes.elliptic_curves.ell_modular_symbols.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
```

CHAPTER

THIRTYFOUR

MODULAR PARAMETRIZATION OF ELLIPTIC CURVES OVER $\mathbb{Q}$

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.

EXMAPLES:

```
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 -
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
```

AUTHORS:

- chris wuthrich (02/10) - moved from ell_rational_field.py.

class sage.schemes.elliptic_curves.modular_parametrization.**ModularParameterization**(E)

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 -
```

curve()

Returns 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 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: E = EllipticCurve('37a'); phi = E.modular_parametrization()
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*)

Computes and returns the power series of this modular parametrization.

The curve must be a 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 + 2
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
```

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)
sage: q=f.parent().gen()
sage: f/q == (X.derivative()/(2*Y+a1*X+a3))
True
```

CHAPTER

THIRTYFIVE

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 $\mathrm{Aut}(E[p]) = GL_2(\mathbb{F}_p)$.

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 $GL_2(\mathbb{F}_p)$:

```
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:

AUTHORS:

- chris wuthrich (02/10) - moved from ell_rational_field.py.

class sage.schemes.elliptic_curves.gal_reps.**GaloisRepresentation**(E)
Bases: sage.structure.sage_object.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
```

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 $PGL_2(\mathbb{F}_p)$.

Let M be a matrix in $GL_2(\mathbb{F}_p)$, then define $u(M) = \mathrm{tr}(M)^2/\det(M)$, which only depends on the conjugacy class of M in $PGL_2(\mathbb{F}_p)$. Hence this defines a map $u : PGL_2(\mathbb{F}_p) \to \mathbb{F}_p$, which is almost a bijection between conjugacy classes of the source and $\mathbb{F}_p$ (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` (optional, 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, 0.1

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
```

REMARKS:

Conjugacy classes of subgroups of $PGL_2(\mathbb{F}_5)$

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)

Returns 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 do
'The image is a... group of order 18.'
sage: EllipticCurve([1,-1,0,-107,-379]).galois_representation().image_type(7)        # the do
'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 trac ticket #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 trac ticket #14752:

```
sage: EllipticCurve([0, 0, 0, -1129345880,-86028258620304]).galois_representation().image_ty
'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)

Returns 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*)

Returns 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 $\mathrm{Gal}(\bar{\mathbf{Q}}/\mathbf{Q})$ 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*)

Returns 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*)

Returns 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*)

Returns true if the Galois representation to $GL_2(\mathbf{Z}_p)$ is quasi-unipotent at $\ell \neq p$, i.e. if there is a fintie 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:

•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*)

Returns 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 $Aut(E[p]) = GL_2(\mathbb{F}_p)$.

False if it is not, or None if we were unable to determine whether it is or not.

INPUT:

- p - int (a prime number)
- A - int (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*)

Returns 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:

- 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):
...
ValueError: unipotent is not defined for l = p, use semistable instead.
```

is_unramified(*p*, *ell*)

Returns 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:

•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*)

Returns 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()
[]

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()

Returns 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]
```

CHAPTER

THIRTYSIX

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 (cyrillic Sha), 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, allow 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, 3, 5]
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()
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 Fiel
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

class sage.schemes.elliptic_curves.sha_tate.**Sha**(E)
Bases: sage.structure.sage_object.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([])
sage: S = E.sha()
sage: S.bound_kato()
[2, 3]
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
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)
sage: S.an_padic(5,prec=4)  # long time
1 + O(5^3)
```

an(*use_database=False*, *descent_second_limit=12*)
Returns 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, g
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
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 trac ticket #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
1.00000000000000
```

A rank 5 curve:

```
sage: EllipticCurve([0, 0, 1, -79, 342]).sha().an_numerical(prec=10, proof=False)  # long ti
1.0
```

See trac ticket #1115:

```
sage: sha=EllipticCurve('37a1').sha()
sage: [sha.an_numerical(prec) for prec in xrange(40,100,10)]  # long time (3s on sage.math,
[1.0000000000,
1.0000000000000,
1.0000000000000000,
1.0000000000000000000,
1.0000000000000000000000,
1.0000000000000000000000000]
```

an_padic(*p*, *prec=0*, *use_twists=True*)

Returns the conjectural order of $Sha(E/\mathbf{Q})$, according to the p-adic analogue of the Birch and Swinnerton-Dyer conjecture as formulated in [MTT] and [BP].

REFERENCES:

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 o
...                                                                  # 32-bit
OverflowError: Python int too large to convert to C long     # 32-bit
7^2 + O(7^24) # 64-bit
sage: EllipticCurve('300b2').sha().an_padic(3)    # 9 elements in sha, long time (2s on sage.
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 (trac ticket #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
2 + 7 + O(7^6)
sage: EllipticCurve([-19,34]).sha().an_padic(5)  # see :trac: `6455`, long time (4s on sage.
1 + O(5)
```

Test for :trac: 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 divisor of Sha is in this list.

EXAMPLES:

```
sage: EllipticCurve('37a').sha().bound()
([2], 1)
```

bound_kato()

Returns a list p of primes such that the theorems of Kato's [Ka] and others (e.g., as explained in a paper/thesis of Grigor Grigorov [Gri]) 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 (Kato): Suppose $L(E,1) \neq 0$ and $p \neq 2, 3$ is a prime such that

- E does not have additive reduction at p,
- the mod-p representation is surjective.

Then $ord_p(\#Sha(E))$ divides $ord_p(L(E,1) \cdot \#E(\mathbf{Q})^2_{tor}/(\Omega_E \cdot \prod c_q))$.

EXAMPLES:

```
sage: E = EllipticCurve([0, -1, 1, -10, -20])    # 11A  = X_0(11)
sage: E.sha().bound_kato()
[2, 3, 5]
sage: E = EllipticCurve([0, -1, 1, 0, 0])        # X_1(11)
sage: E.sha().bound_kato()
[2, 3, 5]
sage: E = EllipticCurve([1,1,1,-352,-2689])      # 66B3
sage: E.sha().bound_kato()
[2, 3]
```

For the following curve one really has that 25 divides the order of Sha (by Grigorov-Stein paper [GS]):

```
sage: E = EllipticCurve([1, -1, 0, -332311, -73733731])   # 1058D1
sage: E.sha().bound_kato()                  # long time (about 1 second)
[2, 3, 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, 3, 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
```

REFERENCES:

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)
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)

Returns a provable 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.

INPUT: p – a prime > 2

OUTPUT: integer – power of p that bounds the order of $Sha(E)(p)$ from above

The result is a proven upper bound on the order of $Sha(E)(p)$. So in particular it proves it finiteness even if the rank of the curve is larger than 1. Note also that this bound is sharp if one assumes the main conjecture of Iwasawa theory of elliptic curves (and this is known in certain cases).

Currently the algorithm is only implemented when certain conditions are verified.

- The mod p Galois representation must be surjective.
- The reduction at p is not allowed to be additive.
- If the reduction at p is non-split multiplicative, then the rank has to be 0.
- If $p = 3$ then the reduction at 3 must be good ordinary or split multiplicative and the rank must be 0.

EXAMPLES:

```
sage: e = EllipticCurve('11a3')
sage: e.sha().p_primary_bound(3)
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('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)
...                                                                 # 32-bit
OverflowError: Python int too large to convert to C long            # 32-bit
0                                                                   # 64-bit
```

Some checks for trac ticket #6406:

```
sage: e.sha().p_primary_bound(7)
Traceback (most recent call last):
...
ValueError: The mod-p Galois representation is not surjective. Current knowledge about Euler

sage: e.sha().an_padic(7)  # long time (depends on "e.sha().p_primary_bound(3)" above)
...                                                                 # 32-bit
OverflowError: Python int too large to convert to C long            # 32-bit
7^2 + O(7^24)                                                        # 64-bit

sage: e = EllipticCurve('11a3')
sage: e.sha().p_primary_bound(5)
Traceback (most recent call last):
...
ValueError: The mod-p Galois representation is not surjective. Current knowledge about Euler
sage: e.sha().an_padic(5)
1 + O(5^22)
```

two_selmer_bound()

This returns 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
```

CHAPTER

THIRTYSEVEN

MISCELLANEOUS P-ADIC FUNCTIONS

p-adic functions from ell_rational_field.py, moved here to reduce crowding in that file.

sage.schemes.elliptic_curves.padics.**matrix_of_frobenius**(*self*, *p*, *prec=20*, *check=False*, *check_hypotheses=True*, *algorithm='auto'*)

Returns the matrix of Frobenius on the Monsky Washnitzer cohomology of the elliptic curve.

INPUT:

- `p` - prime (= 5) 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
[    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*7^11 + 6*7^12 + 5*7^1
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.schemes.elliptic_curves.padics.**padic_E2**(*self*, *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):

```
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
```

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
```

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
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
```

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)
```

sage.schemes.elliptic_curves.padics.**padic_height**(*self*, *p*, *prec=20*, *sigma=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 semi-stable reduction
- `prec` - integer = 1, 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 [Mazur-Tate-Teitelbaum].

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*
```

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   # long time
```

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
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
```

sage.schemes.elliptic_curves.padics.**padic_height_pairing_matrix**(*self*, *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.

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.

TODO: - remove restriction that curve must be in minimal Weierstrass form. This is currently required for E.gens().

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 +
[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
```

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*5^2 + 2*5^3 + O(5^4)   3 + 4*5 + 3*5^2 + 5^3 + O(5^4)                 2 + 4*5 + O(5
[      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
```

sage.schemes.elliptic_curves.padics.**padic_height_via_multiply**(*self*, *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, 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 [Mazur-Tate-Teitelbaum].

AUTHORS:

- David Harvey (2008-01): based on the padic_height() function, using the algorithm of"Computing p-adic heights via point multiplication"

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*
```

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   # long time
```

sage.schemes.elliptic_curves.padics.**padic_lseries**(*self*, *p*, *normalize='L_ratio'*, *use_eclib=True*)

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
- use_eclib - bool (default:True); whether or not to use John Cremona's eclib for the computation of modular symbols

- `normalize` - 'L_ratio' (default), 'period' or 'none'; this is describes the way the modular symbols are normalized. See modular_symbol for more details.

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,use_eclib=False)
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
```

sage.schemes.elliptic_curves.padics.**padic_regulator**(*self*, *p*, *prec=20*, *height=None*, *check_hypotheses=True*)

Computes the cyclotomic p-adic regulator of this curve.

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.

TODO: - remove restriction that curve must be in minimal Weierstrass form. This is currently required for E.gens().

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(5
```

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   # long time
```

A case where the generator belongs to the formal group already (trac #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
```

sage.schemes.elliptic_curves.padics.**padic_sigma**(*self*, *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, 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, 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), which is basically an optimised version of the algorithm from "p-adic Heights and Log Convergence" (Mazur, Stein, Tate).

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^
```

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 +
```

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^
```

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*
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
sage: f(2*t)/2 -g
O(t^11)
```

Test that it returns consistent results over a range of precision:

```
sage: max_N = 30   # get up to at least p^2          # long time
sage: E = EllipticCurve([1, 1, 1, 1, 1])             # long time
sage: p = 5                                          # long time
sage: E2 = E.padic_E2(5, max_N)                      # long time
sage: max_sigma = E.padic_sigma(p, max_N, E2=E2)     # long time
sage: for N in range(3, max_N):                      # long time
...       sigma = E.padic_sigma(p, N, E2=E2)         # long time
...       assert sigma == max_sigma
```

sage.schemes.elliptic_curves.padics.**padic_sigma_truncated**(*self*, *p*, *N=20*, *lamb=0*, *E2=None*, *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, 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, 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" (David Harvey, to appear in LMS JCM), which is basically an optimised version of the algorithm from "p-adic Heights and Log Convergence" (Mazur, Stein, Tate), and "Computing p-adic heights via point multiplication" (David Harvey, still draft form).

Running time is soft-$O(N^2\lambda^{-1}\log p)$, plus whatever time is necessary to compute E_2.

AUTHOR:

- 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)
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^
```

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
```

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):                                          # long time
...           correct = E.padic_sigma(5, N + 3*lamb, E2=E2)               # long time
...           compare = E.padic_sigma_truncated(5, N=N, lamb=lamb, E2=E2)     # long time
...           assert compare == correct                                   # long time
```

CHAPTER

THIRTYEIGHT

COMPLEX MULTIPLICATION FOR ELLIPTIC CURVES

This module implements the functions

- `hilbert_class_polynomial`
- `cm_j_invariants`
- `cm_orders`
- `discriminants_with_bounded_class_number`
- `cm_j_invariants_and_orders`
- `largest_fundamental_disc_with_class_number`

AUTHORS:

- Robert Bradshaw
- John Cremona
- William Stein

`sage.schemes.elliptic_curves.cm.`**`cm_j_invariants`**(*K*, *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.

EXAMPLE:

```
sage: cm_j_invariants(QQ)
[-262537412640768000, -147197952000, -884736000, -12288000, -884736, -32768, -3375, 0, 1728, 800
```

Over imaginary quadratic fields there are no more than over QQ:

```
sage: cm_j_invariants(QuadraticField(-1, 'i'))
[-262537412640768000, -147197952000, -884736000, -12288000, -884736, -32768, -3375, 0, 1728, 800
```

Over real quadratic fields there may be more, for example:

```
sage: len(cm_j_invariants(QuadraticField(5, 'a')))
31
```

Over number fields K of many higher degrees this also works:

```
sage: K.<a> = NumberField(x^3 - 2)
sage: cm_j_invariants(K)
[-12288000, 54000, 0, 287496, 1728, 16581375, -3375, 8000, -32768, -884736, -884736000, -1471979
sage: K.<a> = NumberField(x^4 - 2)
sage: len(cm_j_invariants(K))
23
```

sage.schemes.elliptic_curves.cm.**cm_j_invariants_and_orders**(*K*, *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.

EXAMPLE:

```
sage: cm_j_invariants_and_orders(QQ)
[(-3, 3, -12288000), (-3, 2, 54000), (-3, 1, 0), (-4, 2, 287496), (-4, 1, 1728), (-7, 2, 1658137
```

Over an imaginary quadratic field there are no more than over QQ:

```
sage: cm_j_invariants_and_orders(QuadraticField(-1, 'i'))
[(-3, 3, -12288000), (-3, 2, 54000), (-3, 1, 0), (-4, 2, 287496), (-4, 1, 1728), (-7, 2, 1658137
```

Over real quadratic fields there may be more:

```
sage: v = cm_j_invariants_and_orders(QuadraticField(5,'a')); len(v)
31
sage: [(D,f) for D,f,j in v if j not in QQ]
[(-3, 5), (-3, 5), (-4, 5), (-4, 5), (-15, 2), (-15, 2), (-15, 1), (-15, 1), (-20, 1), (-20, 1),
```

Over number fields K of many higher degrees this also works:

```
sage: K.<a> = NumberField(x^3 - 2)
sage: cm_j_invariants_and_orders(K)
[(-3, 3, -12288000), (-3, 2, 54000), (-3, 1, 0), (-4, 2, 287496), (-4, 1, 1728), (-7, 2, 1658137
```

sage.schemes.elliptic_curves.cm.**cm_orders**(*h*, *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)

EXAMPLES:

```
sage: cm_orders(0)
[]
sage: v = cm_orders(1); v
[(-3, 3), (-3, 2), (-3, 1), (-4, 2), (-4, 1), (-7, 2), (-7, 1), (-8, 1), (-11, 1), (-19, 1), (-4
sage: type(v[0][0]), type(v[0][1])
(<type 'sage.rings.integer.Integer'>, <type 'sage.rings.integer.Integer'>)
sage: v = cm_orders(2); v
 [(-3, 7), (-3, 5), (-3, 4), (-4, 5), (-4, 4), (-4, 3), (-7, 4), (-8, 3), (-8, 2), (-11, 3), (-1
sage: len(v)
29
sage: set([hilbert_class_polynomial(D*f^2).degree() for D,f in v])
set([2])
```

Any degree up to 100 is implemented, but may be prohibitively slow:

```
sage: cm_orders(3)
[(-3, 9), (-3, 6), (-11, 2), (-19, 2), (-23, 2), (-23, 1), (-31, 2), (-31, 1), (-43, 2), (-59, 1
sage: len(cm_orders(4))
84
```

sage.schemes.elliptic_curves.cm.**discriminants_with_bounded_class_number**(*hmax*, *B=None*, *proof=None*)

Return dictionary with keys class numbers $h \leq hmax$ and values the list of all pairs (D, f), with $D < 0$ a fundamental discriminant such that Df^2 has class number h. If the optional bound B is given, return only those pairs with fundamental $|D| \leq B$, though f can still be arbitrarily large.

INPUT:

- `hmax` – integer
- B – integer or None; if None returns all pairs
- `proof` – this code calls the PARI function `qfbclassno`, so it could give wrong answers when `proof``==``False`. The default is whatever `proof.number_field()` is. If `proof==False` and $B$ is `None`, at least the number of discriminants is correct, since it is double checked with Watkins's table.

OUTPUT:

- dictionary

In case B is not given, we use Mark Watkins's: "Class numbers of imaginary quadratic fields" to compute a B that captures all h up to $hmax$ (only available for $hmax \leq 100$).

EXAMPLES:

```
sage: v = sage.schemes.elliptic_curves.cm.discriminants_with_bounded_class_number(3)
sage: v.keys()
[1, 2, 3]
sage: v[1]
[(-3, 3), (-3, 2), (-3, 1), (-4, 2), (-4, 1), (-7, 2), (-7, 1), (-8, 1), (-11, 1), (-19, 1), (-4
sage: v[2]
[(-3, 7), (-3, 5), (-3, 4), (-4, 5), (-4, 4), (-4, 3), (-7, 4), (-8, 3), (-8, 2), (-11, 3), (-15
sage: v[3]
[(-3, 9), (-3, 6), (-11, 2), (-19, 2), (-23, 2), (-23, 1), (-31, 2), (-31, 1), (-43, 2), (-59, 1
sage: v = sage.schemes.elliptic_curves.cm.discriminants_with_bounded_class_number(8, proof=False
sage: [len(v[h]) for h in v.keys()]
[13, 29, 25, 84, 29, 101, 38, 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, 3), (-3, 2), (-3, 1), (-4, 2), (-4, 1), (-7, 2), (-7, 1), (-8, 1), (-11, 1), (-19, 1),
```

sage.schemes.elliptic_curves.cm.**hilbert_class_polynomial**(*D*, *algorithm=None*)

Returns the Hilbert class polynomial for discriminant D.

INPUT:

- `D` (int) – a negative integer congruent to 0 or 1 modulo 4.
- `algorithm` (string, default None) – if "sage" then use the Sage implementation; if "magma" then call Magma (if available).

OUTPUT:

(integer polynomial) The Hilbert class polynomial for the discriminant D.

ALGORITHM:

- If `algorithm` = "sage": Use complex approximations to the roots.
- If `algorithm` = "magma": Call the appropriate Magma function.

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: 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 - 78982425159364679040000000
sage: hilbert_class_polynomial(-37*4, algorithm="magma") # optional - magma
x^2 - 39660183801072000*x - 78982425159364679040000000
sage: hilbert_class_polynomial(-163)
x + 262537412640768000
sage: hilbert_class_polynomial(-163, algorithm="magma") # optional - magma
x + 262537412640768000
```

sage.schemes.elliptic_curves.cm.**largest_fundamental_disc_with_class_number**(*h*)

Return largest absolute value of any fundamental discriminant with class number h, and the number of fundamental discriminants with that class number. This is known for h up to 100, by work of Mark Watkins.

INPUT:

- h – integer

EXAMPLES:

```
sage: sage.schemes.elliptic_curves.cm.largest_fundamental_disc_with_class_number(0)
(0, 0)
sage: sage.schemes.elliptic_curves.cm.largest_fundamental_disc_with_class_number(1)
(163, 9)
sage: sage.schemes.elliptic_curves.cm.largest_fundamental_disc_with_class_number(2)
(427, 18)
```

```
sage: sage.schemes.elliptic_curves.cm.largest_fundamental_disc_with_class_number(10)
(13843, 87)
sage: sage.schemes.elliptic_curves.cm.largest_fundamental_disc_with_class_number(100)
(1856563, 1736)
sage: sage.schemes.elliptic_curves.cm.largest_fundamental_disc_with_class_number(101)
Traceback (most recent call last):
...
NotImplementedError: largest discriminant not known for class number 101
```

CHAPTER

THIRTYNINE

HYPERELLIPTIC CURVES

39.1 Hyperelliptic curve constructor

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[]
sage: HyperellipticCurve(x^3 + x - 1, x+a)
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: 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

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
```

We can change the names of the variables in the output:

```
sage: k.<a> = GF(9); R.<x> = k[]
sage: HyperellipticCurve(x^3 + x - 1, x+a, names=['X','Y'])
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
```

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
```

TESTS:

Check that f can be a constant (see trac ticket #15516):

```
sage: R.<u> = PolynomialRing(Rationals())
sage: HyperellipticCurve(-12, u^4 + 7)
Hyperelliptic Curve over Rational Field defined by y^2 + (x^4 + 7)*y = -12
```

39.2 Hyperelliptic curves over a finite field

AUTHORS:

- David Kohel (2006)
- Robert Bradshaw (2007)
- Alyson Deines, Marina Gresham, Gagan Sekhon, (2010)
- Daniel Krenn (2011)
- Jean-Pierre Flori, Jan Tuitman (2013)

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
```

class sage.schemes.hyperelliptic_curves.hyperelliptic_finite_field.**HyperellipticCurve_finite_**

Bases: sage.schemes.hyperelliptic_curves.hyperelliptic_generic.HyperellipticCurve_generic

Cartier_matrix()

INPUT:

•E : Hyperelliptic Curve of the form $y^2 = f(x)$ over a finite field, $\mathbb{F}_q$

OUTPUT:

•M: The matrix $M = (c_{pi-j})$, where c_i are the coeffients of $f(x)^{(p-1)/2} = \sum c_i x^i$

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)
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 0 1 0]
```

TESTS:

```
sage: K.<x>=GF(2,'x')[]
sage: C=HyperellipticCurve(x^7-1,x)
sage: C.Cartier_matrix()
Traceback (most recent call last):
...
ValueError: p must be odd

sage: K.<x>=GF(5,'x')[]
sage: C=HyperellipticCurve(x^7-1,4)
sage: C.Cartier_matrix()
Traceback (most recent call last):
...
ValueError: E must be of the form y^2 = f(x)

sage: K.<x>=GF(5,'x')[]
```

```
sage: C=HyperellipticCurve(x^8-1,0)
sage: C.Cartier_matrix()
Traceback (most recent call last):
...
ValueError: In this implementation the degree of f must be odd

sage: K.<x>=GF(5,'x')[]
sage: C=HyperellipticCurve(x^5+1,0,check_squarefree=False)
sage: C.Cartier_matrix()
Traceback (most recent call last):
...
ValueError: curve is not smooth
```

Hasse_Witt()

INPUT:

- E : Hyperelliptic Curve of the form $y^2 = f(x)$ over a finite field, $\mathbb{F}_q$

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)
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, $\mathbb{F}_q$

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
```

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 prgoram.

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]
sage: C2 = sage.schemes.generic.scheme.Scheme.count_points(H,4); C2 # long time, 2s on a Cor
[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]
```

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 bef
[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,
3813324208043947180071195938321176148147244128062172555558715783649006587868272993991,
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)
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 only works over the prime field `GF(p)`, and must have `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 curv
```

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: Currently only implemented using hypellfrob, which means only works over the prime field `GF(p)`, and must have `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 curv
```

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 ZZ[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 are supported as well, but a naive algorithm is used; especially when `g = 1`, fast point counting on elliptic curves should be used:

```
sage: K.<z> = GF(23**3)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^3 + z*t + 4)
sage: H.frobenius_polynomial() # long time, 4s on a Corei7
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, when `h` is non-zero, this naive algorithm is currently used as well, whereas we should rather use another defining equation:

```
sage: K = GF(101)
sage: R.<t> = PolynomialRing(K)
sage: H = HyperellipticCurve(t^5 + 27*t + 3, t)
sage: H.frobenius_polynomial() # long time, 3s on a Corei7
x^4 + 2*x^3 - 58*x^2 + 202*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 `\ZZ[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)
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 `\ZZ[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 + 139841906155
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 + 177055
```

This *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_polynomial_matrix(algorithm='hypellfrob')
Traceback (most recent call last):
...
NotImplementedError: Computation of Frobenius matrix only implemented for hyperelliptic curv
```

p_rank()

INPUT:

- E : Hyperelliptic Curve of the form $y^2 = f(x)$ over a finite field, $\mathbb{F}_q$

OUTPUT:

- pr :p-rank

EXAMPLES:

```
sage: K.<x>=GF(49,'x')[]
sage: C=HyperellipticCurve(x^5+1,0)
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),
```

```
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 #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),
```

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),
```

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()

Gives 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)
```

39.3 Hyperelliptic curves over a general ring

EXAMPLE:

```
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
```

EXAMPLE:

```
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 x0, x1 over Rational Field
```

class sage.schemes.hyperelliptic_curves.hyperelliptic_generic.**HyperellipticCurve_generic**(*PP*, *f*, *h=No*
name
genus

Bases: sage.schemes.plane_curves.projective_curve.ProjectiveCurve_generic

base_extend(*R*)

Returns this HyperellipticCurve over a new base ring R.

EXAMPLES:

```
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 Eisenstein Extension of 3-adic Field with capped relative precision

sage: R.<x> = FiniteField(7)[]
sage: H = HyperellipticCurve(x^8 + x + 5)
sage: H.base_extend(FiniteField(7^2, 'a'))
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: 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 Eisenstein Extension of 3-adic Field with capped relative precision

sage: R.<x> = FiniteField(7)[]
sage: H = HyperellipticCurve(x^8 + x + 5)
sage: H.base_extend(FiniteField(7^2, 'a'))
Hyperelliptic Curve over Finite Field in a of size 7^2 defined by y^2 = x^8 + x + 5
```

genus()

x.__init__(...) initializes x; see help(type(x)) for signature

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: set_verbose(None)
sage: H.is_singular()
False
sage: from sage.schemes.plane_curves.projective_curve import ProjectiveCurve_generic
```

```
sage: ProjectiveCurve_generic.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: R.<x> = GF(27, 'a')[]
sage: H = HyperellipticCurve(x^10+2)
sage: set_verbose(None)
sage: H.is_smooth()
True
sage: from sage.schemes.plane_curves.projective_curve import ProjectiveCurve_generic
sage: ProjectiveCurve_generic.is_smooth(H)
False
```

jacobian()

x.__init__(...) initializes x; see help(type(x)) for signature

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

x.__init__(...) initializes x; see help(type(x)) for signature

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: gen 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), returns (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: gen 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), returns (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) 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()

x.__init__(...) initializes x; see help(type(x)) for signature

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')
sage: Hp2 = H.change_ring(K2).odd_degree_model(); Hp2
Hyperelliptic Curve over Number Field in a with defining polynomial x^2 + 2 defined by y^2 =

sage: K3 = QuadraticField(-3, 'b')
sage: Hp3 = H.change_ring(QuadraticField(-3, 'b')).odd_degree_model(); Hp3
Hyperelliptic Curve over Number Field in b with defining polynomial x^2 + 3 defined by y^2 =

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: 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()
x^4 - 16*x^3 + 134*x^2 - 688*x + 1849
sage: H.change_ring(GF(43)).odd_degree_model().frobenius_polynomial()
x^4 - 16*x^3 + 134*x^2 - 688*x + 1849

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()
x^4 - 8*x^3 + 150*x^2 - 536*x + 4489
```

TESTS:: sage: HyperellipticCurve(x^5 + 1, 1).odd_degree_model() Traceback (most recent call last): ... NotImplementedError: odd_degree_model only implemented for curves in Weierstrass form

sage: HyperellipticCurve(x^5 + 1, names="U, V").odd_degree_model() Hyperelliptic Curve over Rational Field defined by V^2 = U^5 + 1

sage.schemes.hyperelliptic_curves.hyperelliptic_generic.**is_HyperellipticCurve**(*C*)

EXAMPLES:

```
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: sage.schemes.hyperelliptic_curves.hyperelliptic_generic.is_HyperellipticCurve(C)
True
```

39.4 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.

EXAMPLE:

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 trac #14755 and
sage: HyperellipticCurve_from_invariants([3840,414720,491028480,2437709561856],reduced = False)
Hyperelliptic Curve over Rational Field defined by y^2 = -46656*x^6 + 46656*x^5 - 19440*x^4 + 43
sage: HyperellipticCurve_from_invariants([21, 225/64, 22941/512, 1])
Traceback (most recent call last):
...
NotImplementedError: Reduction of hyperelliptic curves not yet implemented. See trac #14755 and
```

An example over a finite field:

```
sage: HyperellipticCurve_from_invariants([GF(13)(1),3,7,5])
Hyperelliptic Curve over Finite Field of size 13 defined by y^2 = 8*x^5 + 5*x^4 + 5*x^2 + 9*x +
```

An example over a number field:

```
sage: K = QuadraticField(353, 'a')
sage: H = HyperellipticCurve_from_invariants([21, 225/64, 22941/512, 1], reduced = false)
sage: f = K['x'](H.hyperelliptic_polynomials()[0])
```

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 Projecti
```

Mestre's algorithm only works for generic curves of genus two, so another algorithm is needed for those curves with extra automorphism. See also trac ticket #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 charateristic different from 2, 3, and 5, so another algorithm will be needed for fields of those characteristics. See also trac ticket #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 i
sage: HyperellipticCurve_from_invariants([GF(5)(1),1,0,1])
Traceback (most recent call last):
...
ZeroDivisionError: Inverse does not exist.
```

ALGORITHM:

This is Mestre's algorithm [M1991]. Our implementation is based on the formulae on page 957 of [LY2001],

cross-referenced with [W1999] 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 $y^2 = f$.

REFERENCES:

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 + 12507554
```

Note that the algorithm works over number fields as well:

```
sage: k = NumberField(x^2-41,'a')
sage: a = k.an_element()
sage: Mestre_conic([1,2+a,a,4+a])
Projective Conic Curve over Number Field in a with defining polynomial x^2 - 41 defined by (-801
```

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
```

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 + 33065136
```

ALGORITHM:

The formulas are taken from pages 956 - 957 of [LY2001] and based on pages 321 and 332 of [M1991].

See the code or [LY2001] for the detailed formulae defining x, y, z and L.

39.5 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:

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=0*, *offset=0*)

Bases: sage.structure.element.ModuleElement

Create an element of the Monsky-Washnitzer ring of differentials, of the form $F dx/2y$.

INPUT:

- `parent` – Monsky-Washnitzer differential ring (instance of class `MonskyWashnitzerDifferentialRing`
- `val` – element of the base ring, or list of coefficients
- `offset` – if non-zero, shift val by y^{offset} (default 0)

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: sage.schemes.hyperelliptic_curves.monsky_washnitzer.MonskyWashnitzerDifferential(MW, x)
x dx/2y
sage: sage.schemes.hyperelliptic_curves.monsky_washnitzer.MonskyWashnitzerDifferential(MW, y)
y*1 dx/2y
sage: sage.schemes.hyperelliptic_curves.monsky_washnitzer.MonskyWashnitzerDifferential(MW, x, 10
y^10*x dx/2y
```

coeff()

Returns A, where this element is $A dx/2y$.

EXAMPLES:

```
sage: R.<x> = QQ['x']
sage: C = HyperellipticCurve(x^5-4*x+4)
sage: x,y = C.monsky_washnitzer_gens()
sage: w = 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*)

Computes 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.lift_x(3)
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)

Returns 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)

Computes 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.lift_x(3)
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
```

max_pow_y()

Returns 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()

Returns 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.invariar
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**(*ba*

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.modules.module.Module

A ring of Monsky–Washnitzer differentials over `base_ring`.

Q()

Returns $Q(x)$ 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()
SpecialHyperellipticQuotientRing K[x,y,y^-1] / (y^2 = (1 + O(5^5))*x^5 + (1 + 4*5 + 4*5^2 +
```

change_ring(*R*)

Returns 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()
SpecialHyperellipticQuotientRing K[x,y,y^-1] / (y^2 = (1 + O(5^5))*x^5 + (1 + 4*5 + 4*5^2 +
```

degree()

Returns 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()

Returns the dimension of self.

EXAMPLES:

```
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)

Returns and caches $Q(x^p)$, 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))
-(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)

Returns 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
```

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

$$\phi(x) = x^p$$
$$\phi(y) = y^p\sqrt{1 + \frac{Q(x^p) - Q(x)^p}{Q(x)^p}}.$$

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 -
```

helper_matrix()

We use this to solve for the linear combination of x^iy^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()

Returns $dx/2y$ 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*)

Returns and caches 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=F*

Bases: sage.rings.ring.CommutativeAlgebra

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
```

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, z = R.poly_ring().objgen()
sage: A
Univariate Polynomial Ring in T over Ring of integers modulo 125
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*
```

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 can't seem to get the quotient ring stuff happening right now...

---

create_element(*p0*, *p1*, *p2*, *check=True*)

Creates 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)
(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
```

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**(*parent*, *p0*, *p1*, *p2*, *check*)

Bases: sage.structure.element.CommutativeAlgebraElement

An element of a SpecialCubicQuotientRing.

coeffs()

Returns 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*)

Multiplies 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*)

Returns 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()

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*
```

class sage.schemes.hyperelliptic_curves.monsky_washnitzer.**SpecialHyperellipticQuotientElement**

Bases: sage.structure.element.CommutativeAlgebraElement

Elements 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)  # indirect doctest
-(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()
SpecialHyperellipticQuotientRing K[x,y,y^-1] / (y^2 = x^5 - 36*x + 1) over Algebraic Real Fi
```

coeffs(*R=None*)

Returns 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))
([(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
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),
 (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` w.r.t. 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` w.r.t. 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).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**(*Q*, *R=*, *in-*, *ve*

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.rings.ring.CommutativeAlgebra

Initialization.

TESTS:

Check that caching works:

```
sage: R.<x> = QQ['x']
sage: E = HyperellipticCurve(x^5-3*x+1)
sage: from sage.schemes.hyperelliptic_curves.monsky_washnitzer import SpecialHyperellipticQuotie
sage: SpecialHyperellipticQuotientRing(E) is SpecialHyperellipticQuotientRing(E)
True
```

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())
SpecialHyperellipticQuotientRing K[x,y,y^-1] / (y^2 = x^5 - 3*x + 1) over Universal Cyclotom
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*)

Returns $by^j x^i$, 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*)

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.

monomial_diff_coeffs_matrices()

x.__init__(...) initializes x; see help(type(x)) for signature

monsky_washnitzer()

x.__init__(...) initializes x; see help(type(x)) for signature

prime()

x.__init__(...) initializes x; see help(type(x)) for signature

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
```

sage.schemes.hyperelliptic_curves.monsky_washnitzer.**SpecialHyperellipticQuotientRing_class**

alias of SpecialHyperellipticQuotientRing

sage.schemes.hyperelliptic_curves.monsky_washnitzer.**adjusted_prec**(*p*, *prec*)

Computes 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 = 5
- `prec` – integer, desired output precision, = 1

OUTPUT: adjusted precision (usually slightly more than prec)

sage.schemes.hyperelliptic_curves.monsky_washnitzer.**frobenius_expansion_by_newton**(*Q*, *p*, *M*)

Computes 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_new
sage: R.<x> = Integers(5^3)['x']
sage: Q = x^3 - x + R(1/4)
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*)

Computes 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

$$F0 = \sum_{k=0}^{M-2} \binom{-1/2}{k} p x^{p-1} E^k T^{(M-2-k)p}$$

and

$$F1 = x^p F0,$$
$$where`E = Q(x^p) - Q(x)^p`.$$

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_ser
sage: R.<x> = Integers(5^3)['x']
sage: Q = x^3 - x + R(1/4)
sage: frobenius_expansion_by_series(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.**helper_matrix**(*Q*)

Computes 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*)

Tries to call x.lift(), presumably from the p-adics to ZZ.

If this fails, it assumes the input is a power series, and tries to lift it to a power series over QQ.

This function is just a very kludgy solution to the problem of trying to make the reduction code (below) work over both Zp and Zp[[t]].

EXAMPLES:

```
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*)

Computes 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:

2x2 matrix of frobenius 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)
sage: M
5
sage: R.<x> = PolynomialRing(Integers(p**M))
sage: A = monsky_washnitzer.matrix_of_frobenius(x^3 - x + R(1/4), p, M)
sage: A
[3090  187]
[2945  408]
```

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
[2715  187]
[1445  408]
```

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: 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: 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(            # long time
...                                 x^3 - x + R(1/4), p, M)      # long time
sage: B = A.change_ring(Integers(p**prec)); B               # long time
[74311982 57996908]
[95877067 25828133]
sage: B.det()                                               # long time
10007
sage: B.trace()                                             # long time
66
sage: EllipticCurve([-1, 1/4]).ap(10007)                    # long time
66

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(            # long time
...                                 x^3 - x + R(1/4), p, M)      # long time
sage: B = A.change_ring(Integers(p**prec))                  # long time
sage: B.det()                                               # long time
5
sage: -B.trace()                                            # long time
2
sage: EllipticCurve([-1, 1/4]).ap(5)                        # long time
-2
```

Let us check consistency of the results for a range of precisions:

```
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)          # long time
sage: A = A.change_ring(Integers(p**max_prec))              # long time
sage: result = []                                           # long time
sage: for prec in range(1, max_prec):                       # long time
...       M = monsky_washnitzer.adjusted_prec(p, prec)      # long time
...       R.<x> = PolynomialRing(Integers(p^M),'x')         # long time
...       B = monsky_washnitzer.matrix_of_frobenius(        # long time
...                           x^3 - x + R(1/4), p, M)       # long time
...       B = B.change_ring(Integers(p**prec))              # long time
...       result.append(B == A.change_ring(                 # long time
...                                   Integers(p**prec)))   # long time
sage: result == [True] * (max_prec - 1)                     # long time
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: 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', default_prec=4))        # long t
sage: B                                                     # long time
[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 don't know how to compute the trace in advance, so I'm not sure exactly how this would help. Also, I suspect the running time will be dominated by the expansion, so the trace trick won't 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 isn't a problem after all.

sage.schemes.hyperelliptic_curves.monsky_washnitzer.**matrix_of_frobenius_hyperelliptic**(*Q*, *p=Nor*
prec=
M=No

Computes 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 coefficint 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: 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(
[      3*5 + 5^2 + O(5^3)             3*5 + O(5^3)             4*5 + O(5^3)         2 + 5^2 + O(
[    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(
[            5^2 + O(5^3)       5 + 4*5^2 + O(5^3)     4*5 + 3*5^2 + O(5^3)             2*5 + O(
```

sage.schemes.hyperelliptic_curves.monsky_washnitzer.**reduce_all**(*Q*, *p*, *coeffs*, *offset*, *compute_exact_form=False*)

Applies 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.

EXAMPLE:

```
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*)

Applies 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.

EXAMPLE:

```
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*)

Applies 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.

EXAMPLE:

```
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*)

Applies 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.

EXAMPLE:

```
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]]
```

39.6 Jacobian of a Hyperelliptic curve of Genus 2

class sage.schemes.hyperelliptic_curves.jacobian_g2.**HyperellipticJacobian_g2**(*C*)

Bases: sage.schemes.hyperelliptic_curves.jacobian_generic.HyperellipticJacobian_generic

TESTS:

```
sage: from sage.schemes.jacobians.abstract_jacobian import Jacobian_generic
sage: P2.<x, y, z> = ProjectiveSpace(QQ, 2)
sage: C = Curve(x^3 + y^3 + z^3)
sage: J = Jacobian_generic(C); J
Jacobian of Projective Curve over Rational Field defined by x^3 + y^3 + z^3
sage: type(J)
<class 'sage.schemes.jacobians.abstract_jacobian.Jacobian_generic_with_category'>
```

Note: this is an abstract parent, so we skip element tests:

```
sage: TestSuite(J).run(skip =["_test_an_element",                                    "_tes
```

```
sage: Jacobian_generic(ZZ)
Traceback (most recent call last):
...
TypeError: Argument (=Integer Ring) must be a scheme.
sage: Jacobian_generic(P2)
Traceback (most recent call last):
...
ValueError: C (=Projective Space of dimension 2 over Rational Field) must have dimension 1.
sage: P2.<x, y, z> = ProjectiveSpace(Zmod(6), 2)
sage: C = Curve(x + y + z)
sage: Jacobian_generic(C)
Traceback (most recent call last):
...
TypeError: C (=Projective Curve over Ring of integers modulo 6 defined by x + y + z) must be def
```

kummer_surface()

x.__init__(...) initializes x; see help(type(x)) for signature

39.7 Jacobian of a General Hyperelliptic Curve

class sage.schemes.hyperelliptic_curves.jacobian_generic.**HyperellipticJacobian_generic**(*C*)

Bases: sage.schemes.jacobians.abstract_jacobian.Jacobian_generic

EXAMPLES:

```
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
```

An 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')
sage: C
Hyperelliptic Curve over Rational Field defined by v^2 + u*v = u^5 - u + 1
sage: PP = C.ambient_space()
sage: 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
sage: K.<t> = NumberField(x^2-2)
sage: C(K)
Set of rational points of Hyperelliptic Curve over Number Field in t with defining polynomial x^
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
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()
sage: 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: 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 + (-23263454
sage: R2 = Q2*5; R2
(u^2 - 3789465233/116983808*u - 267915823/58491904, v + (-233827256513849/1789384327168*t + 1/2)
```

```
sage: R3 = Q3*5; R3
(u^2 + 566330080839991389062 3/144264547989509096459 52*u - 265318141763956762312 73/288529095979012
sage: R4 = Q4*5; R4
(u^2 - 3789465233/116983808*u - 267915823/58491904, v + (233827256513849/1789384327168*t + 1/2)*
sage: # Thus we find the following identity:
sage: 5*Q2 + 5*Q4
(1)
sage: # Moreover the following relation holds in the 5-torsion subgroup:
sage: Q2 + Q4 == 2*Q1
True
```

dimension()

Return the dimension of this Jacobian.

OUTPUT: Integer

EXAMPLES:

```
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)
<type 'sage.rings.integer.Integer'>
```

point(*mumford*, *check=True*)

x.__init__(...) initializes x; see help(type(x)) for signature

39.8 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
```

```
sage: b = -x +1
sage: c = x-1
sage: d = 0
sage: D1 = X([a,b])
sage: D1
(x^2 + 2*x + 1, y + x + 2)
sage: D2 = X([c,d])
sage: 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*, ***kwds*)

Bases: sage.schemes.generic.homset.SchemeHomset_points

base_extend(*R*)
x.__init__(...) initializes x; see help(type(x)) for signature

curve()
x.__init__(...) initializes x; see help(type(x)) for signature

value_ring()
Returns S for a homset X(T) where T = Spec(S).

39.9 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)
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_fiel**

Bases: sage.structure.element.AdditiveGroupElement, sage.schemes.generic.morphism.SchemeMo

An element of a Jacobian defined over a field, i.e. in $J(K) = \mathrm{Pic}^0_K(C)$.

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: 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: 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
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
(x + 6, y + 2*a + 2)
sage: 10*Q # indirect doctest
(x^3 + (3*a + 1)*x^2 + (2*a + 5)*x + a + 5, y + (4*a + 5)*x^2 + (a + 1)*x + 6*a + 3)
sage: 7*8297*Q
(1)

sage: Q = J(H.lift_x(F(a+1))); Q
(x + 6*a + 6, y + 2*a)
sage: 7*8297*Q # indirect doctest
(1)

A test over a prime field:

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
sage: J = H.jacobian()(F); J
verbose 0 (...: multi_polynomial_ideal.py, dimension) Warning: falling back to very slow toy imp
Set of rational points of Jacobian of Hyperelliptic Curve over
```

```
Finite Field of size 10000000000000000000000000057 defined
by y^2 + 2*x*y = x^7 + x^2 + 1
sage: Q = J(H.lift_x(F(1))); Q
(x + 10000000000000000000000000056, y + 10000000000000000000000000056)
sage: 10*Q # indirect doctest
(x^3 + 150296037169838934997145567227*x^2 + 377701248971234560956743242408*x + 5094561503524860
sage: 7*8297*Q
(x^3 + 354109761395485675499198390 63*x^2 + 262304042352264645458868899 60*x + 6815714305889597055
```

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')
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))); 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 - 1)
sage: 2 * J(H.lift_x(-1)) # indirect doctest
(x^2 + 2*x + 1, y - 3*x - 4)
sage: 3 * J(H.lift_x(-1)) # indirect doctest
(x^2 - 487*x - 324, y - 10754*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)
```

39.10 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

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: sage.structure.sage_object.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
```

This means that only the odd part of the conductor is known.

```
sage: R.prime_to_2_conductor_only
True
```

The discriminant is always minimal away from 2, but possibly not at 2.

```
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 (away from 2):
    y^2 = x^6 - 240*x^4 - 2550*x^3 - 11400*x^2 - 24100*x - 19855
Minimal Discriminant (away from 2):  56675000
Conductor (away from 2): 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
```

EXAMPLE:

```
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
A Minimal Equation (away from 2):
        y^2 = x^6 + 3*x^3 + 63
Minimal Discriminant (away from 2):  10628388316852992
Conductor (away from 2): 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 (away from 2):
        y^2 = x^6 + 58*x^5 + 1401*x^4 + 18038*x^3 + 130546*x^2 + 503516*x + 808561
Minimal Discriminant (away from 2):  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.)

console()

x.__init__(...) initializes x; see help(type(x)) for signature

raw(Q, P)

Return the raw output of running the `genus2reduction` program on the hyperelliptic curve $y^2 + Q(x)y = P(x)$ as a string.

INPUT:

- `Q` - something coercible to a univariate polynomial over Q.
- `P` - something coercible to a univariate polynomial over Q.

OUTPUT:

- `string` - raw output
- `Q` - what Q was actually input to auxiliary genus2reduction program
- `P` - what P was actually input to auxiliary genus2reduction program

EXAMPLES:

```
sage: x = QQ['x'].0
sage: print genus2reduction.raw(x^3 - 2*x^2 - 2*x + 1, -5*x^5)[0]
a minimal equation over Z[1/2] is :
y^2 = x^6-240*x^4-2550*x^3-11400*x^2-24100*x-19855

factorization of the minimal (away from 2) discriminant :
[2,3;5,5;2267,1]

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

the prime to 2 part of the conductor is 1416875
in factorized form : [2,0;5,4;2267,1]
```

Verify that we fix trac 5573:

```
sage: genus2reduction(x^3 + x^2 + x,-2*x^5 + 3*x^4 - x^3 - x^2 - 6*x - 2)
Reduction data about this proper smooth genus 2 curve:
y^2 + (x^3 + x^2 + x)*y = -2*x^5 + 3*x^4 - x^3 - x^2 - 6*x - 2
...
```

class `sage.interfaces.genus2reduction.`**`Genus2reduction_expect`**(*server=None*, *server_tmpdir=None*, *logfile=None*)

Bases: `sage.interfaces.expect.Expect`

class `sage.interfaces.genus2reduction.`**`ReductionData`**(*raw*, *P*, *Q*, *minimal_equation*, *minimal_disc*, *local_data*, *conductor*, *prime_to_2_conductor_only*)

Bases: `sage.structure.sage_object.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 indiction) 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 : "Modeles entiers de courbes hyperelliptiques sur un corps de valuation discrete", Trans. AMS 348 (1996), 4577-4610, Section 7.2, Proposition 4).

`sage.interfaces.genus2reduction.`**`genus2reduction_console`**`()`

x.__init__(...) initializes x; see help(type(x)) for signature

CHAPTER

FORTY

INDICES AND TABLES

- Index
- Module Index
- Search Page

BIBLIOGRAPHY

[WpJacobianVariety] http://en.wikipedia.org/wiki/Jacobian_variety

[MazurTate1991] Mazur, B., & Tate, J. (1991). The p-adic sigma function. Duke Mathematical Journal, 62(3), 663-688.

[Cha] B. Cha. Vanishing of some cohomology goups and bounds for the Shafarevich-Tate groups of elliptic curves. J. Number Theory, 111:154- 178, 2005.

[Jetchev] D. Jetchev. Global divisibility of Heegner points and Tamagawa numbers. Compos. Math. 144 (2008), no. 4, 811–826.

[Kato] K. Kato. p-adic Hodge theory and values of zeta functions of modular forms. Astérisque, (295):ix, 117-290, 2004.

[Kolyvagin] V. A. Kolyvagin. On the structure of Shafarevich-Tate groups. Algebraic geometry, 94–121, Lecture Notes in Math., 1479, Springer, Berlin, 1991.

[LumStein] A. Lum, W. Stein. Verification of the Birch and Swinnerton-Dyer Conjecture for Elliptic Curves with Complex Multiplication (unpublished)

[Mazur] B. Mazur. Modular curves and the Eisenstein ideal. Inst. Hautes Études Sci. Publ. Math. No. 47 (1977), 33–186 (1978).

[Rubin] K. Rubin. The "main conjectures" of Iwasawa theory for imaginary quadratic fields. Invent. Math. 103 (1991), no. 1, 25–68.

[SteinWuthrich] W. Stein and C. Wuthrich. Computations about Tate-Shafarevich groups using Iwasawa theory. http://wstein.org/papers/shark, February 2008.

[SteinEtAl] G. Grigorov, A. Jorza, S. Patrikis, W. Stein, C. Tarniţă. Computational verification of the Birch and Swinnerton-Dyer conjecture for individual elliptic curves. Math. Comp. 78 (2009), no. 268, 2397–2425.

[SteinToward] Stein, "Toward a Generalization of the Gross-Zagier Conjecture", Int Math Res Notices (2011), doi:10.1093/imrn/rnq075

[HSV] Hess, Smart, Vercauteren, "The Eta Pairing Revisited", IEEE Trans. Information Theory, 52(10): 4595-4602, 2006.

[Mil04] Victor S. Miller, "The Weil pairing, and its efficient calculation", J. Cryptol., 17(4):235-261, 2004

[Cre] John Cremona, Algorithms for Modular Elliptic Curves. Cambridge University Press, 1997.

[Sil1988] Joseph H. Silverman, Computing heights on elliptic curves. Mathematics of Computation, Vol. 51, No. 183 (Jul., 1988), pp. 339-358.

[SilBook] Joseph H. Silverman, The Arithmetic of Elliptic Curves. Second edition. Graduate Texts in Mathematics, 106. Springer, 2009.

[CS] J.E.Cremona, and S. Siksek, Computing a Lower Bound for the Canonical Height on Elliptic Curves over $\mathbf{Q}$, ANTS VII Proceedings: F.Hess, S.Pauli and M.Pohst (eds.), ANTS VII, Lecture Notes in Computer Science 4076 (2006), pages 275-286.

[TT] T. Thongjunthug, Computing a lower bound for the canonical height on elliptic curves over number fields, Math. Comp. 79 (2010), pages 2431-2449.

[CPS] J.E. Cremona, M. Prickett and S. Siksek, Height Difference Bounds For Elliptic Curves over Number Fields, Journal of Number Theory 116(1) (2006), pages 42-68.

[CW2005] 10. (a) Cremona and M. Watkins. Computing isogenies of elliptic curves. preprint, 2005.

[KT2013] K. Tsukazaki, Explicit Isogenies of Elliptic Curves, PhD thesis, University of Warwick, 2013.

[CT] J. E. Cremona and T. Thongjunthug, The Complex AGM, periods of elliptic curves over CC and complex elliptic logarithms. Journal of Number Theory Volume 133, Issue 8, August 2013, pages 2813-2841.

[T] T. Thongjunthug, Computing a lower bound for the canonical height on elliptic curves over number fields, Math. Comp. 79 (2010), pages 2431-2449.

[Se1] Jean-Pierre Serre, Propriétés galoisiennes des points d'ordre fini des courbes elliptiques. Invent. Math. 15 (1972), no. 4, 259–331.

[Se2] Jean-Pierre Serre, Sur les représentations modulaires de degré 2 de $\mathrm{Gal}(\bar{\mathbf{Q}}/\mathbf{Q})$. Duke Math. J. 54 (1987), no. 1, 179–230.

[Co] Alina Carmen Cojocaru, On the surjectivity of the Galois representations associated to non-CM elliptic curves. With an appendix by Ernst Kani. Canad. Math. Bull. 48 (2005), no. 1, 16–31.

[MTT] B. Mazur, J. Tate, and J. Teitelbaum, On p-adic analogues of the conjectures of Birch and Swinnerton-Dyer, Inventiones mathematicae 84, (1986), 1-48.

[BP] Dominique Bernardi and Bernadette Perrin-Riou, Variante p-adique de la conjecture de Birch et Swinnerton-Dyer (le cas supersingulier), C. R. Acad. Sci. Paris, Ser I. Math, 317 (1993), no 3, 227-232.

[SW] William Stein and Christian Wuthrich, Algorithms for the Arithmetic of Elliptic Curves using Iwasawa Theory Mathematics of Computation 82 (2013), 1757-1792.

[Ka] Kayuza Kato, p-adic Hodge theory and values of zeta functions of modular forms, Cohomologies p-adiques et applications arithmétiques III, Astérisque vol 295, SMF, Paris, 2004.

[Gri]

[GS]

[LY2001] K. Lauter and T. Yang, "Computing genus 2 curves from invariants on the Hilbert moduli space", Journal of Number Theory 131 (2011), pages 936 - 958

[M1991] J.-F. Mestre, "Construction de courbes de genre 2 a partir de leurs modules", in Effective methods in algebraic geometry (Castiglioncello, 1990), volume 94 of Progr. Math., pages 313 - 334

[W1999] P. van Wamelen, Pari-GP code, section "thecubic" https://www.math.lsu.edu/~wamelen/Genus2/FindCurve/igusa2curve.gp

[Ked2001] Kedlaya, K., "Counting points on hyperelliptic curves using Monsky-Washnitzer cohomology", J. Ramanujan Math. Soc. 16 (2001) no 4, 323-338

[Edix] Edixhoven, B., "Point counting after Kedlaya", EIDMA-Stieltjes graduate course, Lieden (lecture notes?).

PYTHON MODULE INDEX

i

s

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z