CONTENTS

1 Curve constructor 1
2 Base class of curves 5
3 Affine curves 13
4 Projective curves 43
5 Rational points of curves 77
6 Closed points of integral curves 85
7 Jacobians of curves 93
8 Plane conics 95
9 Plane quartics 125
10 Riemann surfaces 127
11 Indices and Tables 153
Python Module Index 155
Index 157
CHAPTER ONE

CURVE CONSTRUCTOR

Curves are constructed through the curve constructor, after an ambient space is defined either explicitly or implicitly.

EXAMPLES:

```
 sage: A.<x,y> = AffineSpace(QQ, 2)
 sage: Curve([y - x^2], A)
 Affine Plane Curve over Rational Field defined by -x^2 + y
```

```
 sage: P.<x,y,z> = ProjectiveSpace(GF(5), 2)
 sage: Curve(y^2*z^7 - x^9 - x*z^8)
 Projective Plane Curve over Finite Field of size 5 defined by -x^9 + y^2*z^7 - x*z^8
```

AUTHORS:

• William Stein (2005-11-13)
• David Kohel (2006-01)
• Grayson Jorgenson (2016-06)

`sage.schemes.curves.constructor.Curve(F, A=None)`

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 no ambient space is passed in for $A$, and if $F$ is not an algebraic scheme, a new ambient space is constructed.

Also not specifying an ambient space will cause the curve to be defined in either affine or projective space based on properties of $F$. In particular, if $F$ contains a nonhomogeneous polynomial, the curve is affine, and if $F$ consists of homogeneous polynomials, then the curve is projective.

INPUT:

• $F$ – a multivariate polynomial, or a list or tuple of polynomials, or an algebraic scheme.

• $A$ – (default: None) an ambient space in which to create the curve.

EXAMPLES: A projective plane curve.

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

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

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 Curve over Rational Field defined by x^3 + y^3 - z^3 - w^3, x^5 - y*z^4
sage: C.genus()
13
```

An affine space 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 Plane 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
```

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 Plane Curve over Rational Field defined by x^4 - 2*x^2*y^2 + y^4
```

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 Plane 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.
If only a polynomial in three variables is given, then it must be homogeneous such that a projective curve is constructed.

An ambient space can be specified to construct a space curve in an affine or a projective space.

The defining polynomial must be nonzero unless the ambient space itself is of dimension 1.
CHAPTER TWO

BASE CLASS OF CURVES

This module defines the base class of curves in Sage.

Curves in Sage are reduced subschemes of dimension 1 of an ambient space. The ambient space is either an affine space or a projective space.

EXAMPLES:

```sage
A.<x,y,z> = AffineSpace(QQ, 3)
sage: C = Curve([x - y, z - 2])
sage: C
Affine Curve over Rational Field defined by x - y, z - 2
sage: C.dimension()
1
```

AUTHORS:

• William Stein (2005)

class `sage.schemes.curves.curve.Curve_generic(A, polynomials)`

Bases: `AlgebraicScheme_subscheme`

Generic curve class.

EXAMPLES:

```sage
sage: A.<x,y,z> = AffineSpace(QQ, 3)
sage: C = Curve([x - y, z - 2])
sage: loads(C.dumps()) == C
True
```

`change_ring(R)`

Return a new curve which is this curve coerced to R.

INPUT:

• R – ring or embedding

OUTPUT: a new curve which is this curve coerced to R

EXAMPLES:

```sage
sage: P.<x,y,z,w> = ProjectiveSpace(QQ, 3)
sage: C = Curve([x^2 - y^2, z*y - 4/5*w^2], P)
sage: C.change_ring(QuadraticField(-1))  # optional - sage.rings.number_field
```

(continues on next page)
Projective Curve over Number Field in a with defining polynomial \( x^2 + 1 \)
with \( a = 1*I \) defined by \( x^2 - y^2, y*z - 4/5*w^2 \)

\[
\begin{align*}
\text{sage: } & \text{R.<a> = QQ[]} \\
\text{sage: } & \text{K.<b> = NumberField(a^3 + a^2 - 1)} \\
& \text{optional - sage.rings.number_field} \\
\text{sage: } & \text{A.<x,y> = AffineSpace(K, 2)} \\
& \text{optional - sage.rings.number_field} \\
\text{sage: } & \text{C = Curve([K.0*x^2 - x + y^3 - 11], A)} \\
& \text{optional - sage.rings.number_field} \\
\text{sage: } & \text{L = K.embeddings(QQbar)} \\
& \text{optional - sage.rings.number_field} \\
\text{sage: } & \text{setVerbose(-1) \# suppress warnings for slow computation} \\
\text{sage: } & \text{C.change_ring(L[0])} \\
& \text{optional - sage.rings.number_field} \\
\end{align*}
\]

Affine Plane Curve over Algebraic Field defined
by \( y^3 + (-0.8774388331233464? - 0.744861766619745?*I)*x^2 - x - 11 \)

\[
\begin{align*}
\text{sage: } & \text{P.<x,y,z> = ProjectiveSpace(QQ, 2)} \\
\text{sage: } & \text{C = P.curve([y*x - 18*x^2 + 17*z^2])} \\
\text{sage: } & \text{C.change_ring(GF(17))} \\
& \text{optional - sage.rings.finite_rings} \\
\end{align*}
\]

Projective Plane Curve over Finite Field of size 17 defined by \( -x^2 + x*y \)

\[
\text{defining_polynomial()}
\]

Return the defining polynomial of the curve.

**EXAMPLES:**

\[
\begin{align*}
\text{sage: } & \text{x,y,z = PolynomialRing(QQ, 3, names='x,y,z').gens()} \\
\text{sage: } & \text{C = Curve(y^2*z - x^3 - 17*x*z^2 + y*z^2)} \\
\text{sage: } & \text{C.defining_polynomial()} \\
& \text{-x^3 + y^2*z - 17*x*z^2 + y*z^2}
\end{align*}
\]

\[
\text{dimension()}
\]

Return the dimension of the curve.

Curves have dimension one by definition.

**EXAMPLES:**

\[
\begin{align*}
\text{sage: } & \text{x = polygen(QQ)} \\
\text{sage: } & \text{C = HyperellipticCurve(x^7 + x^4 + x)} \\
\text{sage: } & \text{C.dimension()} \\
& 1 \\
\text{sage: } & \text{from sage.schemes.projective.projective_subscheme import AlgebraicScheme_} \\
& \text{subscheme_projective} \\
\text{sage: } & \text{AlgebraicScheme_subscheme_projective.dimension(C)} \\
& 1 \\
\end{align*}
\]

\[
\text{divisor(v, base_ring=None, check=True, reduce=True)}
\]

Return the divisor specified by \( v \).
**Warning:** The coefficients of the divisor must be in the base ring and the terms must be reduced. If you set `check=False` and/or `reduce=False` it is your responsibility to pass a valid object `v`.

EXAMPLES:

```sage
sage: x,y,z = PolynomialRing(QQ, 3, names='x,y,z').gens()
sage: C = Curve(y^2*z - x^3 - 17*x*z^2 + y*z^2)
```

`divisor_group(base_ring=None)`

Return the divisor group of the curve.

**INPUT:**

- `base_ring` – the base ring of the divisor group. Usually, this is \( \mathbb{Z} \) (default) or \( \mathbb{Q} \).

**OUTPUT:** the divisor group of the curve

**EXAMPLES:**

```sage
sage: x,y,z = PolynomialRing(QQ, 3, names='x,y,z').gens()
sage: C = Curve(y^2*z - x^3 - 17*x*z^2 + y*z^2)
sage: Cp = Curve(y^2*z - x^3 - 17*x*z^2 + y*z^2)
sage: C.divisor_group() is Cp.divisor_group()
True
```

`genus()`

Return the geometric genus of the curve.

**EXAMPLES:**

```sage
sage: x,y,z = PolynomialRing(QQ, 3, names='x,y,z').gens()
sage: C = Curve(y^2*z - x^3 - 17*x*z^2 + y*z^2)
sage: C.genus()
1
```

`geometric_genus()`

Return the geometric genus of the curve.

This is by definition the genus of the normalization of the projective closure of the curve over the algebraic closure of the base field; the base field must be a prime field.

**Note:** This calls Singular's genus command.

**EXAMPLES:**

Examples of projective curves.

```sage
sage: P2 = ProjectiveSpace(2, GF(5), names=["x", "y", "z"])
# optional - sage.rings.finite_rings
sage: x, y, z = P2.coordinate_ring().gens()
# optional - sage.rings.finite_rings
sage: C = Curve(y^2*z - x^3 - 17*x*z^2 + y*z^2)
# optional - sage.rings.finite_rings
sage: C.geometric_genus()
```

(continues on next page)
Examples of affine curves.

```
sage: x, y = PolynomialRing(GF(5), 2, 'xy').gens()
#...
sage: C = Curve(y^2 - x^3)
#...
sage: C.geometric_genus()
#...
1
```

```
sage: C = Curve(x^10 + y^7 + 1)
#...
sage: C.geometric_genus()
#...
3
```

**intersection_points**(*C, F=None*)

Return the points in the intersection of this curve and the curve *C*.

If the intersection of these two curves has dimension greater than zero, and if the base ring of this curve is not a finite field, then an error is returned.

**INPUT:**

- *C* – a curve in the same ambient space as this curve
- *F* – (default: None); field over which to compute the intersection points; if not specified, the base ring of this curve is used

**OUTPUT:** a list of points in the ambient space of this curve

**EXAMPLES:**

```
sage: R.<a> = QQ[]
sage: K.<b> = NumberField(a^2 + a + 1)
#...
sage: P.<x,y,z,w> = ProjectiveSpace(QQ, 3)
#...
```

sage: C = Curve([y^2 - w*z, w^3 - y^3], P)  # optional - sage.rings.number_field
sage: D = Curve([x^y - w*z, z^3 - y^3], P)  # optional - sage.rings.number_field
sage: C.intersection_points(D, F=K)  # optional - sage.rings.number_field
[(-b - 1 : -b - 1 : b : 1), (b : b : -b - 1 : 1),
 (1 : 0 : 0 : 0), (1 : 1 : 1 : 1)]

sage: A.<x,y> = AffineSpace(GF(7), 2)  # optional - sage.rings.finite_rings
sage: C = Curve([y^3 - x^3], A)  # optional - sage.rings.finite_rings
sage: D = Curve([-x*y^3 + y^4 - 2*x^3 + 2*x^2*y], A)  # optional - sage.rings.finite_rings
sage: C.intersection_points(D)  # optional - sage.rings.finite_rings
[(0, 0), (1, 1), (2, 2), (3, 3), (4, 4), (5, 5), (5, 6), (6, 6)]

sage: A.<x,y> = AffineSpace(QQ, 2)
sage: C = Curve([y^3 - x^3], A)
sage: D = Curve([-x*y^3 + y^4 - 2*x^3 + 2*x^2*y], A)
sage: C.intersection_points(D)
Traceback (most recent call last):
...
NotImplementedError: the intersection must have dimension zero or (=Rational Field) must be a finite field

intersects_at(C, P)

Return whether the point P is or is not in the intersection of this curve with the curve C.

INPUT:

  • C -- a curve in the same ambient space as this curve.
  • P -- a point in the ambient space of this curve.

EXAMPLES:

sage: P.<x,y,z,w> = ProjectiveSpace(QQ, 3)
sage: C = Curve([x^2 - z^2, y^3 - w*x^2], P)
sage: D = Curve([w^2 - 2*x*y + z^2, y^2 - w^2], P)
sage: Q1 = P([1,1,-1,1])
sage: C.intersects_at(D, Q1)
True
sage: Q2 = P([0,0,1,-1])
sage: C.intersects_at(D, Q2)
False
sage: D = Curve([y^2 + 7*x^2 + 8], A)  # optional - sage.rings.finite_rings

sage: Q1 = A([9,6])  # optional - sage.rings.finite_rings

sage: C.intersects_at(D, Q1)  # optional - sage.rings.finite_rings
True

sage: Q2 = A([3,7])  # optional - sage.rings.finite_rings

sage: C.intersects_at(D, Q2)  # optional - sage.rings.finite_rings
False

is_singular(P=None)

Return whether P is a singular point of this curve, or if no point is passed, whether this curve is singular or not.

This just uses the is_smooth function for algebraic subschemes.

INPUT:

- P – (default: None) a point on this curve

OUTPUT:

A boolean. If a point P is provided, and if P lies on this curve, returns True if P is a singular point of this curve, and False otherwise. If no point is provided, returns True or False depending on whether this curve is or is not singular, respectively.

EXAMPLES:

sage: P.<x,y,z,w> = ProjectiveSpace(QQ, 3)
sage: C = P.curve([y^2 - x^2 - z^2, z - w])
sage: C.is_singular()
False

sage: A.<x,y,z> = AffineSpace(GF(11), 3)  # optional - sage.rings.finite_rings

sage: C = A.curve([y^3 - z^5, x^5 - y + 1])  # optional - sage.rings.finite_rings

sage: Q = A([7,0,0])  # optional - sage.rings.finite_rings

sage: C.is_singular(Q)  # optional - sage.rings.finite_rings
True

singular_points(F=None)

Return the set of singular points of this curve.

INPUT:

- F – (default: None) field over which to find the singular points; if not given, the base ring of this curve is used

OUTPUT: a list of points in the ambient space of this curve

EXAMPLES:
**sage:** \texttt{A.<x,y,z> = AffineSpace(QQ, 3)}

**sage:** \texttt{C = Curve([y^2 - x^5, x - z], A)}

**sage:** \texttt{C.singular_points()}

\[(0, 0, 0)]

**sage:** \texttt{R.<a> = QQ[]}

**sage:** \texttt{K.<b> = NumberField(a^8 - a^4 + 1)}  
\[\text{# optional - sage.rings.number_field}\]

**sage:** \texttt{P.<x,y,z> = ProjectiveSpace(QQ, 2)}  
\[\text{# optional - sage.rings.number_field}\]

**sage:** \texttt{C = Curve([359/12*x*y^2*z^2 + 2*y*z^4 + 187/12*y^3*z^2 + x*z^4 \ldots:} + 67/3*x^2*y*z^2 + 117/4*y^5 + 9*x^5 + 6*x^3*z^2 \ldots:} + 393/4*x^3*y^4 + 145*x^2*y^3 + 115*x^3*y^2 + 49*x^4*y], P)

**sage:** \texttt{sorted(C.singular_points(K), key=str)}  
\[\text{# optional - sage.rings.number_field}\]

\[(-1/2*b^5 - 1/2*b^3 + 1/2*b - 1 : 1 : 0),\]
\[(-2/3*b^4 + 1/3 : 0 : 1),\]
\[(-b^6 : b^6 : 1),\]
\[(1/2*b^5 + 1/2*b^3 - 1/2*b - 1 : 1 : 0),\]
\[(2/3*b^4 - 1/3 : 0 : 1),\]
\[(b^6 : -b^6 : 1)\]

**singular_subscheme()**

Return the subscheme of singular points of this curve.

**OUTPUT:**

- a subscheme in the ambient space of this curve.

**EXAMPLES:**

**sage:** \texttt{A.<x,y> = AffineSpace(CC, 2)}

**sage:** \texttt{C = Curve([y^4 - 2*x^5 - x^2*y], A)}

**sage:** \texttt{C.singular_subscheme()}

Closed subscheme of Affine Space of dimension 2 over Complex Field with 53 bits of precision defined by:

\[(-2.00000000000000)*x^5 + y^4 - x^2*y,\]
\[-10.0000000000000)*x^4 + (-2.00000000000000)*x*y,\]
\[4.00000000000000*y^3 - x^2\]

**sage:** \texttt{P.<x,y,z,w> = ProjectiveSpace(QQ, 3)}

**sage:** \texttt{C = Curve([y^8 - x^2*z*w^5, w^2 - 2*y^2 - x*z], P)}

**sage:** \texttt{C.singular_subscheme()}

Closed subscheme of Projective Space of dimension 3 over Rational Field defined by:

\[y^8 - x^2*z*w^5,\]
\[-2*y^2 - x*z + w^2,\]
\[-x^3*y^2*z^4 + 3*x^2*y*z^3*w^2 - 3*x*y^2*z^2*w^2 + 8*x*y*z^2*w^4 + 9*x^2*y^2*w^5 + y^2*z*w^6,\]
\[x^2*z^2*w^5,\]
\[-5*x^2*z^2*w^4 - 4*x*z*w^6,\]
\[x^4*y^2*z^3 - 3*x^3*y^2*z^2*w^2 + 3*x^2*y^2*z*w^4 - 4*x^2*y^2*w^5 - x*y*w^6,\]
\[-2*x^3*y*z^2*w + 6*x^2*y*z^2*w^3 - 20*x^2*y*z*w^4\]
union(other)

Return the union of self and other.

EXAMPLES:

```
sage: x,y,z = PolynomialRing(QQ, 3, names='x,y,z').gens()
sage: C1 = Curve(z - x)
sage: C2 = Curve(y - x)
sage: C1.union(C2).defining_polynomial()
x^2 - x*y - x*z + y*z
```
Affine curves in Sage are curves in an affine space or an affine plane.

EXAMPLES:

We can construct curves in either an affine plane:

```
sage: A.<x,y> = AffineSpace(QQ, 2)
sage: C = Curve([y - x^2], A); C
Affine Plane Curve over Rational Field defined by -x^2 + y
```

or in higher dimensional affine space:

```
sage: A.<x,y,z,w> = AffineSpace(QQ, 4)
sage: C = Curve([y - x^2, z - w^3, w - y^4], A); C
Affine Curve over Rational Field defined by -x^2 + y, -w^3 + z, -y^4 + w
```

### 3.1 Integral affine curves over finite fields

If the curve is defined over a finite field and integral, that is reduced and irreducible, its function field is tightly coupled with the curve so that advanced computations based on Sage’s global function field machinery are available.

EXAMPLES:

```
sage: k.<a> = GF(2)  # optional - sage.rings.finite_rings
sage: A.<x,y,z> = AffineSpace(k, 3)  # optional - sage.rings.finite_rings
sage: C = Curve([x^2 + x - y^3, y^4 - y - z^3], A)  # optional - sage.rings.finite_rings
sage: C.genus()  # optional - sage.rings.finite_rings
10
sage: C.function_field()  # optional - sage.rings.finite_rings
Function field in z defined by z^9 + x^8 + x^6 + x^5 + x^4 + x^3 + x
```

Closed points of arbitrary degree can be computed:

```
sage: C.closed_points()  # long time  # optional - sage.rings.finite_rings
(continues on next page)
```
The places at infinity correspond to the extra closed points of the curve’s projective closure:

\[
\text{sage: } \text{C.places_at_infinity()}
\]

\[
\text{[Place (1/x, 1/x*z)]}
\]

It is easy to transit to and from the function field of the curve:

\[
\text{sage: } \text{fx} = \text{C}(x)
\]

\[
\text{sage: } \text{fy} = \text{C}(y)
\]

\[
\text{sage: } \text{fx}^2 + \text{fx} - \text{fy}^3
\]

\[
0
\]

\[
\text{sage: } \text{fx}.\text{divisor()}
\]

\[
-9*\text{Place (1/x, 1/x*z)} + 9*\text{Place (x, z)}
\]

\[
\text{sage: } p, = \text{fx}.\text{zeros()}
\]

\[
\text{sage: } \text{C}.\text{place_to_closed_point}(p)
\]

\[
\text{Point (x, y, z)}
\]

\[
\text{sage: } \_\text{rational_point()}
\]

\[
\text{Point (x, y, z)}
\]

\[
\text{sage: } \_\text{closed_point()}
\]

\[
\text{Point (x, y, z)}
\]

\[
\text{sage: } \_\text{place()}
\]

\[
\text{Place (x, z)}
\]
3.2 Integral affine curves over \( \mathbb{Q} \)

An integral curve over \( \mathbb{Q} \) is equipped also with the function field. Unlike over finite fields, it is not possible to enumerate closed points.

EXAMPLES:

```sage
sage: A.\langle x,y \rangle = AffineSpace(QQ, 2)
sage: C = Curve(x^2 + y^2 - 1)
sage: p = C(0,1)
sage: p
(0, 1)
sage: p.closed_point()
Point (x, y - 1)
sage: pl = _.place()
sage: C.parametric_representation(pl)
(s + ..., 1 - 1/2*s^2 - 1/8*s^4 - 1/16*s^6 + ...)
sage: sx, sy = _
sage: sx = sx.polynomial(10); sx
s
sage: sy = sy.polynomial(10); sy
-7/256*s^10 - 5/128*s^8 - 1/16*s^6 - 1/8*s^4 - 1/2*s^2 + 1
``` 

AUTHORS:

- William Stein (2005-11-13)
- David Joyner (2005-11-13)
- David Kohel (2006-01)
- Grayson Jorgenson (2016-08)
- Kwankyu Lee (2019-05): added integral affine curves

**class** `sage.schemes.curves.affine_curve.AffineCurve(A, X)`
Bases: `Curve_generic`, `AlgebraicScheme_subscheme_affine`

Affine curves.

EXAMPLES:

```sage
sage: R.<v> = QQ[]
sage: K.<u> = NumberField(v^2 + 3)  # optional - sage.rings.number_field
sage: A.<x,y,z> = AffineSpace(K, 3)  # optional - sage.rings.number_field
sage: C = Curve([z - u*x^2, y^2], A); C
```
optional - sage.rings.number_field
Affine Curve over Number Field in u with defining polynomial v^2 + 3 defined by (-u)*x^2 + z, y^2

sage: A.<x,y,z> = AffineSpace(GF(7), 3)
optional - sage.rings.finite_rings
sage: C = Curve([x^2 - z, z - 8*x], A); C
optional - sage.rings.finite_rings
Affine Curve over Finite Field of size 7 defined by x^2 - z, -x + z

projective_closure(i=0, PP=None)
Return the projective closure of this affine curve.

INPUT:

- i – (default: 0) the index of the affine coordinate chart of the projective space that the affine ambient space of this curve embeds into.
- PP – (default: None) ambient projective space to compute the projective closure in. This is constructed if it is not given.

OUTPUT: A curve in projective space.

EXAMPLES:

sage: A.<x,y,z> = AffineSpace(QQ, 3)
sage: C = Curve([y-x^2,z-x^3], A)
sage: C.projective_closure()
Projective Curve over Rational Field defined by x1^2 - x0*x2, x1*x2 - x0*x3, x2^2 - x1*x3

sage: A.<x,y,z> = AffineSpace(QQ, 3)
sage: C = Curve([y - x^2, z - x^3], A)
sage: C.projective_closure()
Projective Curve over Rational Field defined by x1^2 - x0*x2, x1*x2 - x0*x3, x2^2 - x1*x3

sage: A.<x,y> = AffineSpace(CC, 2)
sage: C = Curve(y - x^3 + x - 1, A)
sage: C.projective_closure(1)
Projective Plane Curve over Complex Field with 53 bits of precision defined by x0^3 - x0*x1^2 + x1^3 - x1^2*x2

sage: A.<x,y> = AffineSpace(QQ, 2)
sage: P.<u,v,w> = ProjectiveSpace(QQ, 2)
sage: C = Curve([y - x^2], A)
sage: C.projective_closure(1, P).ambient_space() == P
True

class sage.schemes.curves.affine_curve.AffineCurve_field(A, X)

Bases: AffineCurve, AlgebraicScheme_subscheme_affine_field

Affine curves over fields.
\textbf{\texttt{blowup}(P=None)}

Return the blow up of this affine curve at the point \(P\).

The blow up is described by affine charts. This curve must be irreducible.

\textbf{INPUT:}

\begin{itemize}
  \item \(P\) – (default: None) a point on this curve at which to blow up; if \texttt{None}, then \(P\) is taken to be the origin.
\end{itemize}

\textbf{OUTPUT: A tuple of}

\begin{itemize}
  \item a tuple of curves in affine space of the same dimension as the ambient space of this curve, which define the blow up in each affine chart.
  \item a tuple of tuples such that the jth element of the ith tuple is the transition map from the ith affine patch to the jth affine patch.
  \item a tuple consisting of the restrictions of the projection map from the blow up back to the original curve, restricted to each affine patch. There the ith element will be the projection from the ith affine patch.
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}
sage: A.<x,y> = AffineSpace(QQ, 2)
sage: C = Curve([y^2 - x^3], A)
sage: C.blowup()
((Affine Plane Curve over Rational Field defined by s1^2 - x,
  Affine Plane Curve over Rational Field defined by y*s0^3 - 1),
 ([Scheme endomorphism of Affine Plane Curve over Rational Field
defined by s1^2 - x
  Defn: Defined on coordinates by sending (x, s1) to (x, s1),
  Scheme morphism:
  From: Affine Plane Curve over Rational Field defined by s1^2 - x
  To:  Affine Plane Curve over Rational Field defined by y*s0^3 - 1
  Defn: Defined on coordinates by sending (x, s1) to (x*s1, 1/s1)],
  [Scheme morphism:
  From: Affine Plane Curve over Rational Field defined by y*s0^3 - 1
  To:  Affine Plane Curve over Rational Field defined by y*s0^3 - 1
  Defn: Defined on coordinates by sending (y, s0) to (y*s0, 1/s0),
  Scheme endomorphism of Affine Plane Curve over Rational Field
defined by y*s0^3 - 1
  Defn: Defined on coordinates by sending (y, s0) to (y, s0)],)
((Scheme morphism:
  From: Affine Plane Curve over Rational Field defined by s1^2 - x
  To:  Affine Plane Curve over Rational Field defined by -x^3 + y^2
  Defn: Defined on coordinates by sending (x, s1) to (x, x*s1),
  Scheme morphism:
  From: Affine Plane Curve over Rational Field defined by -x^3 + y^2
  To:  Affine Plane Curve over Rational Field defined by -x^3 + y^2
  Defn: Defined on coordinates by sending (x, s1) to (x*s1, 1/s1)))
\end{verbatim}

\begin{verbatim}
sage: K.<a> = QuadraticField(2)
sage: A.<x,y,z> = AffineSpace(K, 3)
sage: C = Curve([y^2 - a*x^5, x - z], A)
sage: B = C.blowup()
\end{verbatim}

(continues on next page)
scheme_morphism(sage: B[2]) #

(continues on next page)
From: Affine Curve over Number Field in a
with defining polynomial \( x^2 - 2 \) with \( a = 1.414213562373095? \)
defined by \( s0 - 1, 2*z^3 + (-a)*s1^2 \)

To: Affine Curve over Number Field in a
with defining polynomial \( x^2 - 2 \) with \( a = 1.414213562373095? \)
defined by \(-a)*x^5 + y^2, x - z\)

Defn: Defined on coordinates by sending \((z, s0, s1)\) to
\((z*s0, z*s1, z)\)

\[\text{sage: } A.<x,y> = AffineSpace(QQ, 2)\]
\[\text{sage: } C = A.curve((y - 3/2)^3 - (x + 2)^5 - (x + 2)^6)\]
\[\text{sage: } Q = A([-2,3/2])\]
\[\text{sage: } C.blowup(Q)\]

((Affine Plane Curve over Rational Field
defined by \( x^3 - s1^3 + 7*x^2 + 16*x + 12 \),
Affine Plane Curve over Rational Field
defined by \( 8*y^3*s0^6 - 36*y^2*s0^6 + 8*y^2*s0^5 + 54*y*s0^6 - 24*y*s0^5 - 27*s0^6 + 18*s0^5 - 8 \),

([Scheme endomorphism of Affine Plane Curve over Rational Field
defined by \( x^3 - s1^3 + 7*x^2 + 16*x + 12 \)
Defn: Defined on coordinates by sending \((x, s1)\) to \((x, s1),\)
Scheme morphism:
From: Affine Plane Curve over Rational Field
defined by \( x^3 - s1^3 + 7*x^2 + 16*x + 12 \)
To: Affine Plane Curve over Rational Field
defined by \( 8*y^3*s0^6 - 36*y^2*s0^6 + 8*y^2*s0^5 + 54*y*s0^6 - 24*y*s0^5 - 27*s0^6 + 18*s0^5 - 8 \)
Defn: Defined on coordinates by sending \((x, s1)\) to
\((x*s1 + 2*s1 + 3/2, 1/s1),\)

[Scheme morphism:
From: Affine Plane Curve over Rational Field
defined by \( 8*y^3*s0^6 - 36*y^2*s0^6 + 8*y^2*s0^5 + 54*y*s0^6 - 24*y*s0^5 - 27*s0^6 + 18*s0^5 - 8 \)
To: Affine Plane Curve over Rational Field
defined by \( x^3 - s1^3 + 7*x^2 + 16*x + 12 \)
Defn: Defined on coordinates by sending \((y, s0)\) to
\((y*s0 - 3/2*s0 - 2, 1/s0),\)
Scheme endomorphism of Affine Plane Curve over Rational Field
defined by \( 8*y^3*s0^6 - 36*y^2*s0^6 + 8*y^2*s0^5 + 54*y*s0^6 - 24*y*s0^5 - 27*s0^6 + 18*s0^5 - 8 \)
Defn: Defined on coordinates by sending \((y, s0)\) to \((y, s0)\)),

(Scheme morphism:
From: Affine Plane Curve over Rational Field
defined by \( x^3 - s1^3 + 7*x^2 + 16*x + 12 \)
To: Affine Plane Curve over Rational Field
defined by \(-x^6 - 13*x^5 - 70*x^4 - 200*x^3 + y^3 - 320*x^2 - 92*y^2 - 272*x + 274*y - 795/8\)
Defn: Defined on coordinates by sending \((x, s1)\) to
\((x, x*s1 + 2*s1 + 3/2),\)

Scheme morphism:
From: Affine Plane Curve over Rational Field
defined by \( 8*y^3*s0^6 - 36*y^2*s0^6 + 8*y^2*s0^5 \)

(continues on next page)
To: Affine Plane Curve over Rational Field
defined by 
- x^6 - 13*x^5 - 70*x^4 - 200*x^3 + y^3
- 320*x^2 - 92*y^2 - 272*x + 274*y - 795/8
Defn: Defined on coordinates by sending (y, s0) to
(y*s0 - 3/2*s0 - 2, y))

plane_projection(AP=None)

Return a projection of this curve into an affine plane so that the image of the projection is a plane curve.

INPUT:

• AP – (default: None) the affine plane to project this curve into. This space must be defined over the
same base field as this curve, and must have dimension two. This space will be constructed if not
specified.

OUTPUT: A tuple of

• a scheme morphism from this curve into an affine plane
• the plane curve that defines the image of that morphism

EXAMPLES:
sage: A.<x,y,z,w> = AffineSpace(QQ, 4)
sage: C = Curve([x^2 - y*z*w, z^3 - w, w + x*y - 3*z^3], A)
sage: C.plane_projection()

(Scheme morphism:
  From: Affine Curve over Rational Field defined by
        -y*z*w + x^2, z^3 - w, -3*z^3 + x*y + w
  To:  Affine Space of dimension 2 over Rational Field
  Defn: Defined on coordinates by sending (x, y, z, w) to (x, y),
        Affine Plane Curve over Rational Field defined by
        x0^2*x1^7 - 16*x0^4)

sage: R.<a> = QQ[]
sage: K.<b> = NumberField(a^2 + 2)  #

sage: A.<x,y,z> = AffineSpace(K, 3)  #

sage: C = A.curve([x - b, y - 2])  #

sage: B.<a,b> = AffineSpace(K, 2)  #

sage: proj1 = C.plane_projection(AP=B)  #

sage: proj1  #

(Scheme morphism:
  From: Affine Curve over Number Field in b
        with defining polynomial a^2 + 2 defined by x + (-b), y - 2
  To:  Affine Space of dimension 2 over Number Field in b
        with defining polynomial a^2 + 2
  Defn: Defined on coordinates by sending (x, y, z) to
        (x, z),
        Affine Plane Curve over Number Field in b
        with defining polynomial a^2 + 2 defined by a + (-b))

sage: proj1[1].ambient_space() is B  #

  optional - sage.rings.number_field

True

sage: proj2 = C.plane_projection()  #

  optional - sage.rings.number_field

sage: proj2[1].ambient_space() is B  #

  optional - sage.rings.number_field

False

projection(indices, AS=None)

Return the projection of this curve onto the coordinates specified by indices.

INPUT:

- indices – a list or tuple of distinct integers specifying the indices of the coordinates to use in the projection. Can also be a list or tuple consisting of variables of the coordinate ring of the ambient space of this curve. If integers are used to specify the coordinates, 0 denotes the first coordinate. The length of indices must be between two and one less than the dimension of the ambient space of this curve, inclusive.
- AS – (default: None) the affine space the projected curve will be defined in. This space must be defined...
over the same base field as this curve, and must have dimension equal to the length of indices. This space is constructed if not specified.

**OUTPUT:** A tuple of

- a scheme morphism from this curve to affine space of dimension equal to the number of coordinates specified in indices
- the affine subscheme that is the image of that morphism. If the image is a curve, the second element of the tuple will be a curve.

**EXAMPLES:**

```python
sage: A.<x,y,z> = AffineSpace(QQ, 3)
sage: C = Curve([y^7 - x^2 + x^3 - 2*z, z^2 - x^7 - y^2], A)
sage: C.projection([0,1])
(Scheme morphism:
  From: Affine Curve over Rational Field
to: Affine Space of dimension 2 over Rational Field
  Defined on coordinates by sending (x, y, z) to
  (x, y),
Affine Plane Curve over Rational Field defined by x1^14 + 2*x0^3*x1^7 - 2*x0^2*x1^7 - 4*x0^7 + x0^6 - 2*x0^5 + x0^4 - 4*x1^2)
sage: C.projection([0,1,3,4])
Traceback (most recent call last):
...
ValueError: (=\[0, 1, 3, 4\]) must be a list or tuple of length between 2 and (=2), inclusive
```

```python
sage: A.<x,y,z,w> = AffineSpace(QQ, 4)
sage: C = Curve([x - 2, y - 3, z - 1], A)
sage: B.<a,b,c> = AffineSpace(QQ, 3)
sage: C.projection([0,1,2], AS=B)
(Scheme morphism:
  From: Affine Curve over Rational Field defined by x - 2, y - 3, z - 1
to: Affine Space of dimension 3 over Rational Field
  Defined on coordinates by sending (x, y, z, w) to
  (x, y, z),
Affine Curve over Rational Field defined by c - 1, b - 3, a - 2)
```

```python
sage: A.<x,y,z,w,u> = AffineSpace(GF(11), 5)
sage: C = Curve([x^3 - 5*y*z + u^2, x - y^2 + 3*z^2, w^2 + 2*u^3*y, y - u^2 + z*x], A)
sage: B.<a,b,c> = AffineSpace(GF(11), 3)
sage: proj1 = C.projection([1,2,4], AS=B)
sage: proj1
(Scheme morphism:
  From: Affine Curve over Finite Field of size 11 defined by x^3 - 5*y*z + u^2, -y^2 + 3*z^2 + x, 2*y^3 + w^2, x*z - u^2 + y
  to: Affine Space of dimension 3 over Finite Field of size 11
  Defined on coordinates by sending (x, y, z, w, u) to
  (a, b, c),
Affine Curve over Finite Field of size 11 defined by a - 1, b - 3, c - 2)
```

(continues on next page)
To: Affine Space of dimension 3 over Finite Field of size 11
Defn: Defined on coordinates by sending (x, y, z, w, u) to
(y, z, u),
Affine Curve over Finite Field of size 11 defined by a^2*b - 3*b^3 -
c^2 + a, c^6 + 5*a*b^4 + b^3*c^2 - 3*a*c^4 + 3*a^2*c^2 - a^3, a^2*c^4 -
3*b^2*c^4 - 2*a^3*c^2 - 5*a*b^2*c^2 + a^4 + 5*a*b^3 + 2*b^4 + b^2*c^2 -
3*b^2*c^2 + 3*a*b, a^4*c^2 + 2*b^4*c^2 - a^5 - 2*a*b^4 + 5*b*c^4 + a*b*c^2
- 5*a*b^2 + 4*b^3 + b*c^2 + 5*c^2 - 5*a, a^6 - 5*b^6 - 5*b^3*c^2 +
5*a*b^3 + 2*c^4 - 4*a*c^2 + 2*a^2 - 5*a^2 + b + c^2)
sage: proj1[1].ambient_space() is B
optional - sage.rings.finite_rings
True
sage: proj2 = C.projection([1,2,4])
optional - sage.rings.finite_rings
sage: proj2[1].ambient_space() is B
optional - sage.rings.finite_rings
False
sage: C.projection([1,2,3,5], AS=B)
Traceback (most recent call last):
...
TypeError: (=Affine Space of dimension 3 over Finite Field of size 11)
must have dimension (=4)
sage: A.<x,y,z,w> = AffineSpace(QQ, 4)
sage: C = A.curve([x*y - z^3, x*z - w^3, w^2 - x^3])
sage: C.projection([y,z])
(Scheme morphism:
From: Affine Curve over Rational Field defined by
-z^3 + x*y, -w^3 + x*z, -x^3 + w^2
To: Affine Space of dimension 2 over Rational Field
Defn: Defined on coordinates by sending (x, y, z, w) to (y, z),
Affine Plane Curve over Rational Field defined by x1^23 - x0^7*x1^4)
sage: B.<x,y,z> = AffineSpace(QQ, 3)
sage: C.projection([x,y,z], AS=B)
(Scheme morphism:
From: Affine Curve over Rational Field defined by
-z^3 + x*y, -w^3 + x*z, -x^3 + w^2
To: Affine Space of dimension 3 over Rational Field
Defn: Defined on coordinates by sending (x, y, z, w) to
(x, y, z),
Affine Curve over Rational Field defined by
z^3 - x^y, x^8 - x^z^2, x^7*z^2 - x^y^z)
sage: C.projection([y,z])
Traceback (most recent call last):
...
ValueError: (=y, z, z) must be a list or tuple of distinct indices or
variables

resolution_of_singularities(extend=False)
Return a nonsingular model for this affine curve created by blowing up its singular points.
The nonsingular model is given as a collection of affine patches that cover it. If extend is False and if the
base field is a number field, or if the base field is a finite field, the model returned may have singularities
with coordinates not contained in the base field. An error is returned if this curve is already nonsingular,
or if it has no singular points over its base field. This curve must be irreducible, and must be defined over
a number field or finite field.

INPUT:

• extend – (default: False) specifies whether to extend the base field when necessary to find all singular
points when this curve is defined over a number field. If extend is False, then only singularities with
coordinates in the base field of this curve will be resolved. However, setting extend to True will slow
down computations.

OUTPUT: A tuple of

• a tuple of curves in affine space of the same dimension as the ambient space of this curve, which
represent affine patches of the resolution of singularities.

• a tuple of tuples such that the jth element of the ith tuple is the transition map from the ith patch to the
jth patch.

• a tuple consisting of birational maps from the patches back to the original curve that were created by
composing the projection maps generated from the blow up computations. There the ith element will
be a map from the ith patch.

EXAMPLES:

\begin{verbatim}
sage: A.<x,y> = AffineSpace(QQ, 2)
sage: C = Curve([y^2 - x^3], A)
sage: C.resolution_of_singularities()
(((Affine Plane Curve over Rational Field defined by s1^2 - x,
  Affine Plane Curve over Rational Field defined by y*s0^3 - 1),
((Scheme endomorphism of Affine Plane Curve over Rational Field
defined by s1^2 - x
  Defn: Defined on coordinates by sending (x, s1) to (x, s1),
  Scheme morphism:
  From: Affine Plane Curve over Rational Field defined by s1^2 - x
  To:   Affine Plane Curve over Rational Field defined by y*s0^3 - 1
  Defn: Defined on coordinates by sending (x, s1) to (x*s1, 1/s1)),
(Scheme morphism:
  From: Affine Plane Curve over Rational Field defined by y*s0^3 - 1
  To:   Affine Plane Curve over Rational Field defined by s1^2 - x
  Defn: Defined on coordinates by sending (y, s0) to (y*s0, 1/s0),
  Scheme endomorphism of Affine Plane Curve over Rational Field
defined by y*s0^3 - 1
  Defn: Defined on coordinates by sending (y, s0) to (y, s0)),
(Scheme morphism:
  From: Affine Plane Curve over Rational Field defined by s1^2 - x
  To:   Affine Plane Curve over Rational Field defined by -x^3 + y^2
  Defn: Defined on coordinates by sending (x, s1) to (x, x*s1),
  Scheme morphism:
  From: Affine Plane Curve over Rational Field defined by y*s0^3 - 1
  To:   Affine Plane Curve over Rational Field defined by -x^3 + y^2
  Defn: Defined on coordinates by sending (y, s0) to (y*s0, y)))
sage: set_verbose(-1)
sage: K.<a> = QuadraticField(3)
#...
\end{verbatim}

(continues on next page)
sage: A.<x,y> = AffineSpace(K, 2)
    # Optional - sage.rings.number_field
sage: C = A.curve(x^4 + 2*x^2 + a*y^3 + 1)
    # Optional - sage.rings.number_field
sage: C.resolution_of_singularities(extend=True)[0]
    # long time (2 s) #
    # Optional - sage.rings.number_field
(Affine Plane Curve over Number Field in a0
with defining polynomial y^4 - 4*y^2 + 16
defined by 24*x^2*ss1^3 + 24*ss1^3 + (a0^3 - 8*a0),
Affine Plane Curve over Number Field in a0
with defining polynomial y^4 - 4*y^2 + 16
defined by 24*s1^2*ss0 + (a0^3 - 8*a0)*ss0^2 + (-6*a0^3)*s1,
Affine Plane Curve over Number Field in a0
with defining polynomial y^4 - 4*y^2 + 16
defined by 8*y^2*s0^4 + (4*a0^3)*y*s0^3 - 32*s0^2 + (a0^3 - 8*a0)*y)

sage: A.<x,y,z> = AffineSpace(GF(5), 3)
    # Optional - sage.rings.finite_rings
sage: C = Curve([y - x^3, (z - 2)^2 - y^3 - x^3], A)
    # Optional - sage.rings.finite_rings
sage: R = C.resolution_of_singularities()
    # Optional - sage.rings.finite_rings
sage: R[0]
    # Optional - sage.rings.finite_rings
(Affine Curve over Finite Field of size 5
defined by x^2 - s1, s1^4 - x*s2^2 + s1, x^3*s1 - s2^2 + x,
Affine Curve over Finite Field of size 5
defined by y*s2^2 - y^2 - 1, s2^4 - s0^3 - y^2 - 2, y^3*s0^3 - s2^2 + y,
Affine Curve over Finite Field of size 5
defined by s0^3*s1 + z*s1^3 + s1^4 - 2*s1^3 - 1,
z^3*s0^3 + z^3*s1^3 - 2*s0^3 - 2*s1^3 - 1,
z^2*s1^3 + z*s1^3 - s1^3 - z + s1 + 2)

sage: A.<x,y,z,w> = AffineSpace(QQ, 4)
sage: C = A.curve([(x - 2)^2 + y^2)^2 - (x - 2)^2 + y^2 + (x - 2)^3,
....: z - y - 7, w - 4])
sage: B = C.resolution_of_singularities()
sage: B[0]
(Affine Curve over Rational Field defined by s3, s1 - s2,
x^2*s2^4 - 4*x*s2^4 + 2*x^2*s2^2 + 4*s2^4 - 8*x*s2^2
+ x^2 + 7*s2^2 - 3*x + 1,
Affine Curve over Rational Field defined by s3, s2 - 1,
y^2*s0^4 + 2*y^2*s0^2 + y^3*s0^3 + y^2 - s0^2 - 1,
Affine Curve over Rational Field defined by s3, s1 - 1,
z^2*s0^4 - 14*z*s0^4 + 2*z^2*s0^2 + z^3*s0^3 + 49*s0^4
- 28*z^4 - 7*s0^3 + z^2 + 97*s0^2 - 14*z + 48,
Closed subscheme of Affine Space of dimension 4 over Rational Field
defined by: 1)

sage: A.<x,y> = AffineSpace(QQ, 2)
sage: C = Curve([y - x^2 + 1], A)

3.2. Integral affine curves over \( \mathbb{Q} \)
Curves, Release 10.1

sage: C.resolution_of_singularities()
Traceback (most recent call last):
...
TypeError: this curve is already nonsingular

sage: A.<x,y> = AffineSpace(QQ, 2)
sage: C = A.curve([(x^2 + y^2 - y - 2)*(y - x^2 + 2) + y^3])
sage: C.resolution_of_singularities()
Traceback (most recent call last):
...
TypeError: this curve has no singular points over its base field. If
working over a number field use extend=True

tangent_line(p)

Return the tangent line at the point p.

INPUT:

• p – a rational point of the curve

EXAMPLES:

sage: A3.<x,y,z> = AffineSpace(3, QQ)
sage: C = Curve([x + y + z, x^2 - y^2*z^2 + z^3])
sage: p = C(0,0,0)
sage: C.tangent_line(p)
Traceback (most recent call last):
...
ValueError: the curve is not smooth at (0, 0, 0)
sage: p = C(1,0,-1)
sage: C.tangent_line(p)
Affine Curve over Rational Field defined by x + y + z, 2*x + 3*z + 1

We check that the tangent line at p is the tangent space at p, translated to p.

sage: Tp = C.tangent_space(p)
sage: Tp
Closed subscheme of Affine Space of dimension 3 over Rational Field
defined by: x + y + z, 2*x + 3*z
sage: phi = A3.translation(A3.origin(), p)
sage: T = phi * Tp.embedding_morphism()
sage: T.image()
Closed subscheme of Affine Space of dimension 3 over Rational Field
defined by: -2*y + z + 1, x + y + z
sage: _ == C.tangent_line(p)
True

class sage.schemes.curves.affine_curve.AffinePlaneCurve(A,f)

Bases: AffineCurve

Affine plane curves.

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:

```python
sage: F = GF(5)
```

```python
sage: P2 = AffineSpace(2, F, names='xy')
```

```python
sage: R = P2.coordinate_ring()
```

```python
sage: x, y = R.gens()
```

```python
sage: f = y^2 - x^9 - x
```

```python
sage: C = Curve(f)
```

```python
sage: K = FractionField(R)
```

```python
sage: r = 1/x
```

```python
sage: C.divisor_of_function(r)  # todo: not implemented (broken)
```

```python
[-1, (0, 0, 1)]
```

```python
sage: r = 1/x^3
```

```python
sage: C.divisor_of_function(r)  # todo: not implemented (broken)
```

```python
[-3, (0, 0, 1)]
```

**is_ordinary_singularity(P)**

Return whether the singular point P of this affine plane curve is an ordinary singularity.

The point P is an ordinary singularity of this curve if it is a singular point, and if the tangents of this curve at P are distinct.

INPUT:

- P – a point on this curve

OUTPUT:

True or False depending on whether P is or is not an ordinary singularity of this curve, respectively. An error is raised if P is not a singular point of this curve.

EXAMPLES:

```python
sage: A.<x,y> = AffineSpace(QQ, 2)
```

```python
c = Curve([y^2 - x^3], A)
```

```python
Q = A([0, 0])
```

```python
c.is_ordinary_singularity(Q)
```

False
Curves, Release 10.1

sage: R.<a> = QQ[]
sage: K.<b> = NumberField(a^2 - 3)  #
  → optional - sage.rings.number_field
sage: A.<x,y> = AffineSpace(K, 2)  #
  → optional - sage.rings.number_field
sage: C = Curve([(x^2 + y^2 - 2*x)^2 - x^2 - y^2], A)  #
  → optional - sage.rings.number_field
sage: Q = A([0,0])  #
  → optional - sage.rings.number_field
sage: C.is_ordinary_singularity(Q)  #
  → optional - sage.rings.number_field
True

sage: A.<x,y> = AffineSpace(QQ, 2)
sage: C = A.curve([x^2*y - y^2*x + y^2 + x^3])
sage: Q = A([-1,-1])
sage: C.is_ordinary_singularity(Q)
Traceback (most recent call last):
...
TypeError: (=(-1, -1)) is not a singular point of (=Affine Plane Curve over Rational Field defined by x^3 + x^2*y - x*y^2 + y^2)

is_transverse(C, P)

Return whether the intersection of this curve with the curve C at the point P is transverse.

The intersection at P is transverse if P is a nonsingular point of both curves, and if the tangents of the curves at P are distinct.

INPUT:

• C – a curve in the ambient space of this curve.

• P – a point in the intersection of both curves.

OUTPUT: A boolean.

EXAMPLES:

sage: A.<x,y> = AffineSpace(QQ, 2)
sage: C = Curve([x^2 + y^2 - 1], A)
sage: D = Curve([x - 1], A)
sage: Q = A([1,0])
sage: C.is_transverse(D, Q)
False

sage: R.<a> = QQ[]
sage: K.<b> = NumberField(a^3 + 2)  #
  → optional - sage.rings.number_field
sage: A.<x,y> = AffineSpace(K, 2)  #
  → optional - sage.rings.number_field
sage: C = A.curve([x*y])  #
  → optional - sage.rings.number_field
sage: D = A.curve([y - b*x])  #
  → optional - sage.rings.number_field
sage: Q = A([0,0])  #
(continues on next page)
optional - sage.rings.number_field
sage: C.is_transverse(D, Q)  
#~
False

sage: A.<x,y> = AffineSpace(QQ, 2)
sage: C = Curve([y - x^3], A)
sage: D = Curve([y + x], A)
sage: Q = A([0,0])
sage: C.is_transverse(D, Q)  
True

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: \(x = x_0 + t, y = y_0 + \text{power series in } t\)

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]

multiplicity\((P)\)

Return the multiplicity of this affine plane curve at the point \(P\).

In the special case of affine plane curves, the multiplicity of an affine plane curve at the point \((0,0)\) can be computed as the minimum of the degrees of the homogeneous components of its defining polynomial. To compute the multiplicity of a different point, a linear change of coordinates is used.

This curve must be defined over a field. An error if raised if \(P\) is not a point on this curve.

INPUT:

• \(P\) – a point in the ambient space of this curve.
OUTPUT: An integer.

EXAMPLES:

```python
sage: A.<x,y> = AffineSpace(QQ, 2)
sage: C = Curve([y^2 - x^3], A)
sage: Q1 = A([1,1])
sage: C.multiplicity(Q1)
1
sage: Q2 = A([0,0])
sage: C.multiplicity(Q2)
2
sage: A.<x,y> = AffineSpace(QQbar,2)
    # optional - sage.rings.number_field
sage: C = Curve([-x^7 + (-7)*x^6 + y^6 + (-21)*x^5 + 12*y^5
    # optional - sage.rings.number_field
    ....: + (-35)*x^4 + 60*y^4 + (-35)*x^3 + 160*y^3
    ....: + (-21)*x^2 + 240*y^2 + (-7)*x + 192*y + 63], A)
sage: Q = A([-1,-2])
    # optional - sage.rings.number_field
sage: C.multiplicity(Q)
    # optional - sage.rings.number_field
6
```

```python
sage: A.<x,y> = AffineSpace(QQ, 2)
sage: C = A.curve([y^3 - x^3 + x^6])
sage: Q = A([1,1])
sage: C.multiplicity(Q)
Traceback (most recent call last):
  ...TypeError: (=1, 1) is not a point on (=Affine Plane Curve over Rational Field defined by x^6 - x^3 + y^3)
```

```python
plot(*args, **kwds)
```
Plot the real points on this affine plane curve.

INPUT:

- *args - optional tuples (variable, minimum, maximum) for plotting dimensions
- **kwds - optional keyword arguments passed on to implicit_plot

EXAMPLES:

A cuspidal curve:

```python
sage: R.<x, y> = QQ[]
sage: C = Curve(x^3 - y^2)
sage: C.plot()
Graphics object consisting of 1 graphics primitive
```

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

```python
sage: R.<x, y> = ZZ[]
sage: C = Curve(32*x^2 - 2097152*y^11 + 1441792*y^9 - 360448*y^7 + 39424*y^5 - ...
```
(continues on next page)
A line over \( \mathbb{R} \):

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

\texttt{rational\_parameterization()} 

Return a rational parameterization of this curve.

This curve must have rational coefficients and be absolutely irreducible (i.e. irreducible over the algebraic closure of the rational field). The curve must also be rational (have geometric genus zero).

The rational parameterization may have coefficients in a quadratic extension of the rational field.

\textbf{OUTPUT:}

- a birational map between \( \mathbb{A}^1 \) and this curve, given as a scheme morphism.

\textbf{EXAMPLES:}

\[
sage: A.<x,y> = AffineSpace(QQ, 2)
\]
\[
sage: C = Curve([y^2 - x], A)
\]
\[
sage: C.rational\_parameterization()
\]

Scheme morphism:
From: Affine Space of dimension 1 over Rational Field
To: Affine Plane Curve over Rational Field defined by y^2 - x
Defn: Defined on coordinates by sending \( (t) \) to
\((t^2, t)\)

\[
sage: A.<x,y> = AffineSpace(QQ, 2)
\]
\[
sage: C = Curve([(x^2 + y^2 - 2*x)^2 - x^2 - y^2], A)
\]
\[
sage: C.rational\_parameterization()
\]

Scheme morphism:
From: Affine Space of dimension 1 over Rational Field
To: Affine Plane Curve over Rational Field defined by x^4 + 2*x^2*y^2 + y^4 - 4*x^3 + 4*y^3 - 4*x*y^2 - 3*x^2 - y^2
Defn: Defined on coordinates by sending \( (t) \) to
\((-12*t^4 + 6*t^3 + 4*t^2 - 2*t)/(-25*t^4 + 40*t^3 - 26*t^2 + 8*t - 1), (-9*t^4 + 12*t^3 - 4*t + 1)/(-25*t^4 + 40*t^3 - 26*t^2 + 8*t - 1))\)

\[
sage: A.<x,y> = AffineSpace(QQ, 2)
\]
\[
sage: C = Curve([x^2 + y^2 + 7], A)
\]
\[
sage: C.rational\_parameterization()
\]

Scheme morphism:
From: Affine Space of dimension 1 over Number Field in \( a \) with defining polynomial \( a^2 + 7 \)
To: Affine Plane Curve over Number Field in \( a \) with defining polynomial \( a^2 + 7 \) defined by x^2 + y^2 + 7

3.2. Integral affine curves over \( \mathbb{Q} \)
Defn: Defined on coordinates by sending \((t)\) to 
\[((-7*t^2 + 7)/((-a)*t^2 + (-a)), 14*t/((-a)*t^2 + (-a)))\]

**tangents(P, factor=True)**

Return the tangents of this affine plane curve at the point \(P\).

The point \(P\) must be a point on this curve.

**INPUT:**

- \(P\) – a point on this curve
- \(\text{factor}=(\text{default: True})\) whether to attempt computing the polynomials of the individual tangent lines over the base field of this curve, or to just return the polynomial corresponding to the union of the tangent lines (which requires fewer computations)

**OUTPUT:** A list of polynomials in the coordinate ring of the ambient space.

**EXAMPLES:**

```sage
sage: set_verbose(-1)

sage: A.<x,y> = AffineSpace(QQbar, 2)  # optional - sage.rings.number_field

sage: C = Curve([x^5*y^3 + 2*x^4*y^4 + x^3*y^5 + 3*x^4*y^3 + 6*x^3*y^4 + 3*x^2*y^5 + 3*x^3*y^3 + 6*x^2*y^4 + 3*x*y^5 + x^5 + 10*x^4*y + 40*x^3*y^2 + 81*x^2*y^3 + 82*x*y^4 + 33*y^5], A)  # optional - sage.rings.number_field

sage: Q = A([0,0])  # optional - sage.rings.number_field

sage: C.tangents(Q)  # optional - sage.rings.number_field
[x + 3.425299577684708*y,
x + (1.949159013086856 + 1.179307909383728*I)*y,
x + (1.949159013086856 - 1.179307909383728*I)*y,
x + (1.338191198070795 + 0.2560234251008043*I)*y,
x + (1.338191198070795 - 0.2560234251008043*I)*y]

sage: C.tangents(Q, factor=False)  # optional - sage.rings.number_field
[120*x^5 + 1200*x^4*y + 4800*x^3*y^2 + 9720*x^2*y^3 + 9840*x*y^4 + 3960*y^5]
```

```sage
sage: R.<a> = QQ[]

sage: K.<b> = NumberField(a^2 - 3)  # optional - sage.rings.number_field

sage: A.<x,y> = AffineSpace(K, 2)  # optional - sage.rings.number_field

sage: C = Curve([(x^2 + y^2 - 2*x)^2 - x^2 - y^2], A)  # optional - sage.rings.number_field

sage: Q = A([0,0])  # optional - sage.rings.number_field

sage: C.tangents(Q)  # optional - sage.rings.number_field
[x + (-1/3*b)*y, x + (1/3*b)*y]
```

**Chapter 3. Affine curves**

---

32
```python
sage: A.<x,y> = AffineSpace(QQ, 2)
sage: C = A.curve([y^2 - x^3 - x^2])
sage: Q = A([0,0])
sage: C.tangents(Q)
[x - y, x + y]
```

```python
sage: A.<x,y> = AffineSpace(QQ, 2)
sage: C = A.curve([y*x - x^4 + 2*x^2])
sage: Q = A([1,1])
sage: C.tangents(Q)
```

```
Traceback (most recent call last):
  ...TypeError: (=1, 1) is not a point on (=Affine Plane Curve over Rational Field defined by -x^4 + 2*x^2 + x*y)
```

class `sage.schemes.curves.affine_curve.AffinePlaneCurve_field`(A,f)

Bases: `AffinePlaneCurve`, `AffineCurve_field`

Affine plane curves over fields.

`braid_monodromy()`

Compute the braid monodromy of a projection of the curve.

OUTPUT:

A list of braids. The braids correspond to paths based in the same point; each of this paths is the conjugated of a loop around one of the points in the discriminant of the projection of `self`.

NOTE:

The projection over the x axis is used if there are no vertical asymptotes. Otherwise, a linear change of variables is done to fall into the previous case.

EXAMPLES:

```python
sage: A.<x,y> = AffineSpace(QQ, 2)
sage: C = A.curve((x^2-y^3)*(x+3*y-5))
sage: C.braid_monodromy()  # optional - sirocco
[s1*s0*(s1*s2)^2*s0^2*s2^2*s0^-1*(s2^-1*s1^-1)^2*s0^-1*s1^-1,
s1*s0*(s1*s2)^2*(s0*s2^-1*s1*s2*s1*s2^-1)^2*(s2^-1*s1^-1)^2*s0^-1*s1^-1,
s1*s0*(s1*s2)^2*s2^2*s1^-1*s2^-1*s1^-1*s0^-1*s1^-1,
s1*s0*s2*s0^-1*s2*s1^-1]
```

`fundamental_group()`

Return a presentation of the fundamental group of the complement of `self`.

**Note:** The curve must be defined over the rationals or a number field with an embedding over \( \mathbb{Q} \).

EXAMPLES:

```python
sage: A.<x,y> = AffineSpace(QQ, 2)
sage: C = A.curve(y^2 - x^3 - x^2)
sage: C.fundamental_group()  # optional - sirocco
```

Finitely presented group < x0 | >
In the case of number fields, they need to have an embedding to the algebraic field:

```sage
sage: a = QQ[x](x^2 + 5).roots(QQbar)[0][0]
    # optional - sage.rings.number_field
sage: F = NumberField(a.minpoly(), 'a', embedding=a)
    # optional - sage.rings.number_field
sage: F.inject_variables()
    # optional - sage.rings.number_field
```

Defining a

```sage
sage: A.<x,y> = AffineSpace(F, 2)
    # optional - sage.rings.number_field
sage: C = A.curve(y^2 - a*x^3 - x^2)
    # optional - sage.rings.number_field
sage: C.fundamental_group()  # optional - sirocco
```

Finitely presented group < x0 | >

**Warning:** This functionality requires the sirocco package to be installed.

```sage
riemann_surface(**kwargs)
```

Return the complex Riemann surface determined by this curve

**OUTPUT:** A *RiemannSurface* object.

**EXAMPLES:**

```sage
sage: R.<x,y> = QQ[]
sage: C = Curve(x^3 + 3*y^3 + 5)
sage: C.riemann_surface()
Riemann surface defined by polynomial f = x^3 + 3*y^3 + 5 = 0, with 53 bits of precision
```

---

**class** `sage.schemes.curves.affine_curve.AffinePlaneCurve_finite_field(A, f)`

**Bases:** `AffinePlaneCurve_field`

Affine plane curves over finite fields.

**method** `rational_points(algorithm='enum')`

Return sorted list of all rational points on this curve.

**INPUT:**

- `algorithm` – possible choices:
  - `'enum'` – use *very* naive point enumeration to find all rational points on this curve over a finite field.
  - `'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.

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

We compare Brill-Noether and enumeration:

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

riemann_roch_basis(D)

Return a basis of the Riemann-Roch space of the divisor D.

This interfaces with Singular's Brill-Noether command.

This curve is assumed to be a plane curve defined by a polynomial equation \( f(x, y) = 0 \) 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)
INPUT:

- $D$ – an $n$-tuple of integers $(d_1, ..., d_n)$ representing the divisor $Div = d_1 P_1 + \cdots + d_n P_n$, where $X(F) = \{P_1, \ldots, P_n\}$. The ordering is that dictated by $\text{places\_on\_curve}$.

OUTPUT: A basis of $L(Div)$.

EXAMPLES:

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

class `sage.schemes.curves.affine_curve.IntegralAffineCurve(A, X)`

Bases: `AffineCurve_field`

Base class for integral affine curves.

**coordinate_functions()**

Return the coordinate functions.

EXAMPLES:

```sage
A.<x,y> = AffineSpace(GF(8), 2)
sage: C = Curve(x^5 + y^5 + x^y + 1)
sage: x, y = C.coordinate_functions()
sage: x^5 + y^5 + x^y + 1
0
```

**function(f)**

Return the function field element coerced from $f$.

INPUT:

- $f$ – an element of the coordinate ring of either the curve or its ambient space.

EXAMPLES:

```sage
A.<x,y> = AffineSpace(GF(8), 2)
sage: C = Curve(x^5 + y^5 + x^y + 1)
sage: f = C.function(x/y)
```

(continues on next page)
Curves, Release 10.1

function_field()

Return the function field of the curve.

EXAMPLES:

```
sage: A.<x,y> = AffineSpace(QQ, 2)
sage: C = Curve(x^3 - y^2 - x^4 - y^4)
sage: C.function_field()
Function field in y defined by y^4 + y^2 + x^4 - x^3
```

parametric_representation(place, name=None)

Return a power series representation of the branch of the curve given by place.

INPUT:

- place – a place on the curve

EXAMPLES:

```
sage: A.<x,y> = AffineSpace(QQ, 2)
sage: C = Curve(x^2 + y^2 -1)
sage: p = C(0,1)
sage: p.closed_point()
Point (x, y - 1)
sage: pl = _.place()
sage: C.parametric_representation(pl)
(s + ..., 1 - 1/2*s^2 - 1/8*s^4 - 1/16*s^6 + ...)
```

(continues on next page)
sage: p, = C.singular_closed_points()  # optional - sage.rings.finite_rings
sage: b1, b2 = p.places()  # optional - sage.rings.finite_rings
sage: xs, ys = C.parametric_representation(b1)  # optional - sage.rings.finite_rings
sage: f = xs^2 - xs^4 - ys^4  # optional - sage.rings.finite_rings
sage: [f.coefficient(i) for i in range(5)]  # optional - sage.rings.finite_rings
[0, 0, 0, 0, 0]
sage: xs, ys = C.parametric_representation(b2)  # optional - sage.rings.finite_rings
sage: f = xs^2 - xs^4 - ys^4  # optional - sage.rings.finite_rings
sage: [f.coefficient(i) for i in range(5)]  # optional - sage.rings.finite_rings
[0, 0, 0, 0, 0]

place_to_closed_point(place)

Return the closed point on the place.

INPUT:

- place – a place of the function field of the curve

EXAMPLES:

sage: A.<x,y> = AffineSpace(GF(4), 2)  # optional - sage.rings.finite_rings
sage: C = Curve(x^5 + y^5 + x*y + 1)  # optional - sage.rings.finite_rings
sage: F = C.function_field()  # optional - sage.rings.finite_rings
sage: pls = F.places(1)  # optional - sage.rings.finite_rings
sage: C.place_to_closed_point(pls[-1])  # optional - sage.rings.finite_rings
Point (x + 1, y + 1)
sage: C.place_to_closed_point(pls[-2])  # optional - sage.rings.finite_rings
Point (x + 1, y + 1)

places_at_infinity()

Return the places of the curve at infinity.

EXAMPLES:

sage: A.<x,y> = AffineSpace(QQ, 2)
sage: C = Curve(x^3 - y^2 - x^4 - y^4)
sage: C.places_at_infinity()  # optional - sage.rings.finite_rings
[Place (1/x, 1/x^2*y, 1/x^3*y^2, 1/x^4*y^3)]
sage: F = GF(9)  # optional - sage.rings.finite_rings
sage: A2.<x,y> = AffineSpace(F, 2)  # optional - sage.rings.finite_rings
sage: C = A2.curve(y^3 + y - x^4)  # optional - sage.rings.finite_rings
sage: C.places_at_infinity()  # optional - sage.rings.finite_rings
[Place (1/x, 1/x^3*y^2)]

sage: A.<x,y,z> = AffineSpace(GF(11), 3)  # optional - sage.rings.finite_rings
sage: C = Curve([x*z - y^2, y - z^2, x - y*z], A)  # optional - sage.rings.finite_rings
sage: C.places_at_infinity()  # optional - sage.rings.finite_rings
[Place (1/x, 1/x*z^2)]

places_on(point)
Return the places on the closed point.

INPUT:

* point – a closed point of the curve

OUTPUT: A list of the places of the function field of the curve.

EXAMPLES:

sage: A.<x,y> = AffineSpace(QQ, 2)
sage: C = Curve(x^3 - y^2 - x^4 - y^4)
sage: C.singular_closed_points()
[Point (x, y)]
sage: p, =_
sage: C.places_on(p)
[Place (x, y^2, 1/x*y^3 + 1/x*y)]

sage: k.<a> = GF(9)  # optional - sage.rings.finite_rings
sage: A.<x,y> = AffineSpace(k, 2)  # optional - sage.rings.finite_rings
sage: C = Curve(y^2 - x^5 - x^4 - 2*x^3 - 2*x - 2)  # optional - sage.rings.finite_rings
sage: pts = C.closed_points()  # optional - sage.rings.finite_rings
sage: pts  # optional - sage.rings.finite_rings
[Point (x, y + (a + 1)),
 Point (x, y + (-a - 1)),
 Point (x + (a + 1), y + (a - 1)),
 Point (x + (a + 1), y + (-a + 1)),
 Point (x - 1, y + (a + 1)),
 Point (x - 1, y + (-a - 1)),
 Point (x + (-a - 1), y + a),
 Point (x + (-a - 1), y + a),
]
Point \((x + (-a - 1), y + (-a)),\)
Point \((x + 1, y + 1),\)
Point \((x + 1, y - 1)\)

```
sage: p1, p2, p3 = pts[:3]  # optional - sage.rings.finite_rings
sage: C.places_on(p1)  # optional - sage.rings.finite_rings
[Place \((x, y + a + 1)\)]
```

```
sage: C.places_on(p2)  # optional - sage.rings.finite_rings
[Place \((x, y + 2a + 2)\)]
```

```
sage: C.places_on(p3)  # optional - sage.rings.finite_rings
[Place \((x + a + 1, y + a + 2)\)]
```

```
sage: F.<a> = GF(8)  # optional - sage.rings.finite_rings
sage: P.<x,y,z> = ProjectiveSpace(F, 2)  # optional - sage.rings.finite_rings
sage: Cp = Curve(x^3*y + y^3*z + x*z^3)  # optional - sage.rings.finite_rings
sage: C = Cp.affine_patch(0)  # optional - sage.rings.finite_rings

sage: C.singular_closed_points()  # optional - sage.rings.finite_rings
[]
```

```
sage: A.<x,y> = AffineSpace(GF(7^2), 2)  # optional - sage.rings.finite_rings
sage: C = Curve(x^2 - x^4 - y^4)  # optional - sage.rings.finite_rings
sage: C.singular_closed_points()  # optional - sage.rings.finite_rings
[Point \((x, y)\)]
```

```
sage: A.<x,y,z> = AffineSpace(GF(11), 3)  # optional - sage.rings.finite_rings
sage: C = Curve([x^2*z - y^2, y - z^2, x - y*z], A)  # optional - sage.rings.finite_rings
sage: C.singular_closed_points()  # optional - sage.rings.finite_rings
[]
```

```python
class sage.schemes.curves.affine_curve.IntegralAffineCurve_finite_field(A, X)
Bases: IntegralAffineCurve

Integral affine curves.

INPUT:

- \(A\) – an ambient space in which the curve lives
- \(X\) – list of polynomials that define the curve
```
EXAMPLES:

```sage
A.<x,y,z> = AffineSpace(GF(11), 3)  #
Optional - sage.rings.finite_rings
sage: C = Curve([x*z - y^2, y - z^2, x - y*z], A); C  #
Optional - sage.rings.finite_rings
Affine Curve over Finite Field of size 11
defined by -y^2 + x*z, -z^2 + y, -y*z + x
sage: C.function_field()  #
Optional - sage.rings.finite_rings
Function field in z defined by z^3 + 10*x
```

`closed_points(degree=1)`

Return a list of the closed points of `degree` of the curve.

**INPUT:**

- `degree` – a positive integer

**EXAMPLES:**

```sage
A.<x,y> = AffineSpace(GF(7), 2)  #
Optional - sage.rings.finite_rings
sage: C = Curve(x^2 - x^4 - y^4)  #
Optional - sage.rings.finite_rings
sage: C.closed_points()  #
Optional - sage.rings.finite_rings
[Point (x, y),
 Point (x + 1, y),
 Point (x + 2, y + 2),
 Point (x + 2, y - 2),
 Point (x - 2, y + 2),
 Point (x - 2, y - 2),
 Point (x - 1, y)]
```

`places(degree=1)`

Return all places on the curve of the degree.

**INPUT:**

- `degree` – positive integer

**EXAMPLES:**

```sage
F = GF(9)  #
Optional - sage.rings.finite_rings
sage: A2.<x,y> = AffineSpace(F, 2)  #
Optional - sage.rings.finite_rings
sage: C = A2.curve(y^3 + y - x^4)  #
Optional - sage.rings.finite_rings
sage: C.places()  #
Optional - sage.rings.finite_rings
[Place (1/x, 1/x^3*y^2),
 Place (x, y),
 Place (x, y + z2 + 1),
 Place (x, y + 2*z2 + 2),
(continues on next page)
```

3.2. Integral affine curves over $\mathbb{Q}$
class sage.schemes.curves.affine_curve.IntegralAffinePlaneCurve(A, f)
Bases: IntegralAffineCurve, AffinePlaneCurve_field

class sage.schemes.curves.affine_curve.IntegralAffinePlaneCurve_finite_field(A, f)
Bases: AffinePlaneCurve_finite_field, IntegralAffineCurve_finite_field

Integral affine plane curve over a finite field.

EXAMPLES:

```python
class sage.schemes.curves.affine_curve.IntegralAffinePlaneCurve(A, f)
Bases: IntegralAffineCurve, AffinePlaneCurve_field

class sage.schemes.curves.affine_curve.IntegralAffinePlaneCurve_finite_field(A, f)
Bases: AffinePlaneCurve_finite_field, IntegralAffineCurve_finite_field

Integral affine plane curve over a finite field.

EXAMPLES:

```
CHAPTER FOUR

PROJECTIVE CURVES

Projective curves in Sage are curves in a projective space or a projective plane.

EXAMPLES:

We can construct curves in either a projective plane:

```sage
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: C = Curve([y*z^2 - x^3], P); C
Projective Plane Curve over Rational Field defined by -x^3 + y*z^2
```

or in higher dimensional projective spaces:

```sage
sage: P.<x,y,z,w> = ProjectiveSpace(QQ, 3)
sage: C = Curve([y*w^3 - x^4, z*w^3 - x^4], P); C
Projective Curve over Rational Field defined by -x^4 + y*w^3, -x^4 + z*w^3
```

4.1 Integral projective curves over finite fields

If the curve is defined over a finite field and integral, that is reduced and irreducible, its function field is tightly coupled with the curve so that advanced computations based on Sage’s global function field machinery are available.

EXAMPLES:

```sage
sage: k = GF(2)  # optional - sage.rings.finite_rings
sage: P.<x,y,z> = ProjectiveSpace(k, 2)  # optional - sage.rings.finite_rings
sage: C = Curve(x^2*z - y^3, P)  # optional - sage.rings.finite_rings
sage: C.genus()  # optional - sage.rings.finite_rings
0
sage: C.function_field()  # optional - sage.rings.finite_rings
Function field in z defined by z + y^3
```

Closed points of arbitrary degree can be computed:

```sage
sage: C.closed_points()  # optional - sage.rings.finite_rings
(continues on next page)
Curves, Release 10.1

(continued from previous page)

\[[\text{Point} (x, y), \text{Point} (y, z), \text{Point} (x + z, y + z)]\]
\[
sage: \text{C}.\text{closed_points}(2) \#_
\]
\[
\text{optional} - \text{sage.rings.finite_rings}
\]
\[[\text{Point} (y^*2 + y^*2 + z^*2, x + z)]\]
\[
sage: \text{C}.\text{closed_points}(3) \#_
\]
\[
\text{optional} - \text{sage.rings.finite_rings}
\]
\[[\text{Point} (y^*3 + y^*2 + z^*3, x + y + z),
\text{Point} (x^*2 + y^*2 + z^*2, x^*y + x^*z + y^*z, y^*2 + x^*z + y^*z + z^*2)]\]

All singular closed points can be found:

\[
sage: \text{C}.\text{singular_closed_points}() \#_
\]
\[
\text{optional} - \text{sage.rings.finite_rings}
\]
\[[\text{Point} (x, y)]\]
\[
sage: p = \_\[0\] \#_
\]
\[
\text{optional} - \text{sage.rings.finite_rings}
\]
\[
sage: p.\text{places}() \# \text{ a unibranch singularity, that is, a cusp}
\]
\[
\text{optional} - \text{sage.rings.finite_rings}
\]
\[[\text{Place} (1/y)]\]
\[
sage: pls = \_\[0\] \#_
\]
\[
\text{optional} - \text{sage.rings.finite_rings}
\]
\[
sage: \text{C}.\text{place_to_closed_point}(\text{pls}) \#_
\]
\[
\text{optional} - \text{sage.rings.finite_rings}
\]
\text{Point} (x, y)

It is easy to transit to and from the function field of the curve:

\[
sage: \text{fx} = \text{C}(x/z) \#_
\]
\[
\text{optional} - \text{sage.rings.finite_rings}
\]
\[
sage: \text{fy} = \text{C}(y/z) \#_
\]
\[
\text{optional} - \text{sage.rings.finite_rings}
\]
\[
sage: \text{fx}^*2 - \text{fy}^*3 \#_
\]
\[
\text{optional} - \text{sage.rings.finite_rings}
\]
\[
0
\]
\[
sage: \text{fx}.\text{divisor}() \#_
\]
\[
\text{optional} - \text{sage.rings.finite_rings}
\]
\[
3*\text{Place} (1/y)
\]
\[
- 3*\text{Place} (y)
\]
\[
sage: p, = \text{fx}.\text{poles}() \#_
\]
\[
\text{optional} - \text{sage.rings.finite_rings}
\]
\[
sage: p \#_
\]
\[
\text{optional} - \text{sage.rings.finite_rings}
\]
\text{Place} (y)
\[
sage: \text{C}.\text{place_to_closed_point}(\text{p}) \#_
\]
\[
\text{optional} - \text{sage.rings.finite_rings}
\]
\text{Point} (y, z)
\[
sage: \_.\text{rational_point}() \#_
\]
\[
\text{optional} - \text{sage.rings.finite_rings}
\]
\[
(1 : 0 : 0)
\]
\[
sage: \_.\text{closed_point}() \#_
\]
\[
\text{optional} - \text{sage.rings.finite_rings}
\]
\text{Point} (y, z)

(continues on next page)
4.2 Integral projective curves over $\mathbb{Q}$

An integral curve over $\mathbb{Q}$ is also equipped with the function field. Unlike over finite fields, it is not possible to enumerate closed points.

EXAMPLES:

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: C = Curve(x^2*z^2 - x^4 - y^4, P)
sage: C.singular_closed_points()
[Point (x, y)]
sage: p, = _
sage: p.places()
[Place (1/y, 1/y^2*z - 1), Place (1/y, 1/y^2*z + 1)]
sage: fy = C.function(y/z)
sage: fy.divisor()
Place (1/y, 1/y^2*z - 1)
  + Place (1/y, 1/y^2*z + 1)
  + Place (y, z - 1)
  + Place (y, z + 1)
  - Place (y^4 + 1, z)
sage: supp = _.support()
sage: pl = supp[0]
sage: C.place_to_closed_point(pl)
Point (x, y)
sage: pl = supp[1]
sage: C.place_to_closed_point(pl)
Point (x, y)
sage: _.rational_point()
(0 : 0 : 1)
sage: _ in C
True
```

AUTHORS:

- William Stein (2005-11-13)
- David Joyner (2005-11-13)
- David Kohel (2006-01)
- Moritz Minzlaff (2010-11)
- Grayson Jorgenson (2016-08)
- Kwankyu Lee (2019-05): added integral projective curves

`sage.schemes.curves.projective_curve.Hasse_bounds(q, genus=1)`

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

INPUT:
• \( q \) (int) – a prime power
• \( \text{genus} \) (int, default 1) – a non-negative integer,

OUTPUT: A tuple. The Hasse bounds (lb,ub) for the cardinality of a curve of genus \( \text{genus} \) defined over \( \mathbb{F}_q \).

EXAMPLES:

```python
sage: Hasse_bounds(2)
(1, 5)
sage: Hasse_bounds(next_prime(10^30))
(9999999999999980000000000000058, 10000000000000020000000000000058)
```

class sage.schemes.curves.projective_curve.IntegralProjectiveCurve(A,f)

Bases: ProjectiveCurve_field

Integral projective curve.

coordinate_functions\( (i=None) \)

Return the coordinate functions for the \( i \)-th affine patch.

If \( i \) is None, return the homogeneous coordinate functions.

EXAMPLES:

```python
sage: P.<x,y,z> = ProjectiveSpace(GF(4), 2)  # optional - sage.rings.finite_rings
sage: C = Curve(x^5 + y^5 + x*y*z^3 + z^5)  # optional - sage.rings.finite_rings
sage: C.coordinate_functions(0)  # optional - sage.rings.finite_rings
(y, z)
sage: C.coordinate_functions(1)  # optional - sage.rings.finite_rings
(1/y, 1/y*z)
```

function\( (f) \)

Return the function field element coerced from \( x \).

EXAMPLES:

```python
sage: P.<x,y,z> = ProjectiveSpace(GF(4), 2)  # optional - sage.rings.finite_rings
sage: C = Curve(x^5 + y^5 + x*y*z^3 + z^5)  # optional - sage.rings.finite_rings
sage: f = C.function(x/y); f  # optional - sage.rings.finite_rings
1/y
sage: f.divisor()  # optional - sage.rings.finite_rings
Place (1/y, 1/y*z + 1)
+ Place (1/y, 1/y^2*z^2 + z2/y*z + 1)
+ Place (1/y, 1/y^2*z^2 + (z2 + 1)/y*z + 1)
+ Place (1/y, 1/y^2*z + 1)
- Place (y, z^2 + z2*z + 1)
- Place (y, z^2 + (z2 + 1)*z + 1)
- Place (y, z + 1)
```
function_field()

Return the function field of this curve.

EXAMPLES:

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: C = Curve(x^2 + y^2 + z^2, P)
sage: C.function_field()
Function field in z defined by z^2 + y^2 + 1
```

```
sage: P.<x,y,z> = ProjectiveSpace(GF(4), 2)
# optional - sage.rings.finite_rings
sage: C = Curve(x^5 + y^5 + x*y*z^3 + z^5)
# optional - sage.rings.finite_rings
sage: C.function_field()
# optional - sage.rings.finite_rings
Function field in z defined by z^5 + y*z^3 + y^5 + 1
```

place_to_closed_point(place)

Return the closed point at the place.

INPUT:

- place – a place of the function field of the curve

EXAMPLES:

```
sage: P.<x,y,z> = ProjectiveSpace(GF(5), 2)
# optional - sage.rings.finite_rings
sage: C = Curve(y^2*z^7 - x^9 - x*z^8)
# optional - sage.rings.finite_rings
sage: pls = C.places()
# optional - sage.rings.finite_rings
sage: C.place_to_closed_point(pls[-1])
Point (x - 2*z, y - 2*z)
sage: pls2 = C.places(2)
# optional - sage.rings.finite_rings
sage: C.place_to_closed_point(pls2[0])
Point (y^2 + y*z + z^2, x + y)
```

places_on(point)

Return the places on the closed point.

INPUT:

- point – a closed point of the curve

EXAMPLES:

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: C = Curve(x*y*z^4 - x^6 - y^6)
sage: C.singular_closed_points()
[Point (x, y)]
sage: p, = _
```

(continues on next page)
sage: C.places_on(p)
[Place (1/y, 1/y^2*z, 1/y^3*z^2, 1/y^4*z^3),
 Place (y, y^2*z, y^3*z^2, y^4*z^3)]
sage: pl1, pl2 = ...
sage: C.place_to_closed_point(pl1)
Point (x, y)
sage: C.place_to_closed_point(pl2)
Point (x, y)

sage: P.<x,y,z> = ProjectiveSpace(GF(5), 2)
---------------------------------------------------------------
optional - sage.rings.finite_rings
sage: C = Curve(x^2*z - y^3)
optional - sage.rings.finite_rings

sage: [C.places_on(p) for p in C.closed_points()]  #
optional - sage.rings.finite_rings
[[Place (1/y)],
 [Place (y)],
 [Place (y + 1)],
 [Place (y + 2)],
 [Place (y + 3)],
 [Place (y + 4)]]

sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)  
 optional - sage.rings.finite_rings
sage: C = Curve(y^2*z - x^3, P)  
 optional - sage.rings.finite_rings
sage: C.singular_closed_points()  
 optional - sage.rings.finite_rings
[Point (x, z)]

sage: P.<x,y,z> = ProjectiveSpace(GF(5), 2)  
 optional - sage.rings.finite_rings
sage: C = Curve(y^2*z^7 - x^9 - x*z^8)  
 optional - sage.rings.finite_rings
sage: C.singular_closed_points()  
 optional - sage.rings.finite_rings
[Point (x, z)]

class sage.schemes.curves.projective_curve.IntegralProjectiveCurve_finite_field(A, f) 

Bases: IntegralProjectiveCurve

Integral projective curve over a finite field.

INPUT:

• A – an ambient projective space
• f – homogeneous polynomials defining the curve

EXAMPLES:
Curves, Release 10.1

sage: P.<x,y,z> = ProjectiveSpace(GF(5), 2)  # optional - sage.rings.finite_rings
sage: C = Curve(y^2*z^7 - x^9 - x*z^8)  # optional - sage.rings.finite_rings
sage: C.function_field()  # optional - sage.rings.finite_rings
Function field in z defined by z^8 + 4*y^2*z^7 + 1
sage: C.closed_points()  # optional - sage.rings.finite_rings
[Point (x, z),
Point (x, y),
Point (x - 2*z, y + 2*z),
Point (x + 2*z, y + z),
Point (x + 2*z, y - z),
Point (x - 2*z, y - 2*z)]

**L_polynomial** (**name**='t')

Return the L-polynomial of this possibly singular curve.

**INPUT:**

- **name** – (default: t) name of the variable of the polynomial

**EXAMPLES:**

```
sage: A.<x,y> = AffineSpace(GF(3),2)  # optional - sage.rings.finite_rings
sage: C = Curve(y^2 - x^5 - x^4 - 2*x^3 - 2*x - 2)  # optional - sage.rings.finite_rings
sage: Cbar = C.projective_closure()  # optional - sage.rings.finite_rings
sage: Cbar.L_polynomial()  # optional - sage.rings.finite_rings
9*t^4 - 3*t^3 + t^2 - t + 1
```

**closed_points** (**degree**=1)

Return a list of closed points of **degree** of the curve.

**INPUT:**

- **degree** – a positive integer

**EXAMPLES:**

```
sage: A.<x,y> = AffineSpace(GF(9),2)  # optional - sage.rings.finite_rings
sage: C = Curve(y^2 - x^5 - x^4 - 2*x^3 - 2*x - 2)  # optional - sage.rings.finite_rings
sage: Cp = C.projective_closure()  # optional - sage.rings.finite_rings
sage: Cp.closed_points()  # optional - sage.rings.finite_rings
[Point (x0, x1),
Point (x0 + (-z2 - 1)*x2, x1),
Point (x0 + (z2 + 1)*x2, x1),
Point (x0 + z2*x2, x1 + (z2 - 1)*x2),
(Point (x0 + z2*x2, x1 + (z2 - 1)*x2),
(continues on next page)
```
number_of_rational_points$(r=1)$

Return the number of rational points of the curve with constant field extended by degree $r$.

INPUT:

- $r$ – positive integer (default: 1)

EXAMPLES:

```sage
sage: A.<x,y> = AffineSpace(GF(3), 2)
  # optional - sage.rings.finite_rings
sage: C = Curve(y^2 - x^5 - x^4 - 2*x^3 - 2*x - 2)
  # optional - sage.rings.finite_rings
sage: Cbar = C.projective_closure()
  # optional - sage.rings.finite_rings
sage: Cbar.number_of_rational_points(3)
  # optional - sage.rings.finite_rings
21
```

places$(degree=1)$

Return all places on the curve of the degree.

INPUT:

- degree – positive integer

EXAMPLES:

```sage
sage: P.<x,y,z> = ProjectiveSpace(GF(5), 2)
  # optional - sage.rings.finite_rings
sage: C = Curve(x^2*z - y^3)
  # optional - sage.rings.finite_rings
sage: C.places()
  # optional - sage.rings.finite_rings
[Place (1/y),
 Place (y),
 Place (y + 1),
 Place (y + 2),
 Place (y + 3),
 Place (y + 2*
     z - y^3),
 Place (y + 16),
 Place (y + 17),
 Place (y + 18),
 Place (y + 19),
 Place (y + 20),
 Place (y + 21),
 Place (y + 22),
 Place (y + 23),
 Place (y + 24),
 Place (y + 25),
 Place (y + 26),
 Place (y + 27),
 Place (y + 28),
 Place (y + 29),
 Place (y + 30),
 Place (y + 31)]
```
class sage.schemes.curves.projective_curve.IntegralProjectivePlaneCurve(A, f)
Bases: IntegralProjectiveCurve, ProjectivePlaneCurve_field

Integral projective plane curve over a finite field.

INPUT:

• A – ambient projective plane

• f – a homogeneous equation that defines the curve

EXAMPLES:

sage: A.<x,y> = AffineSpace(GF(9), 2) #
˓→optional - sage.rings.finite_rings
sage: C = Curve(y^2 - x^5 - x^4 - 2*x^3 - 2*x - 2) #
˓→optional - sage.rings.finite_rings
sage: Cb = C.projective_closure() #
˓→optional - sage.rings.finite_rings
sage: Cb.singular_closed_points() #
˓→optional - sage.rings.finite_rings
[Point (x0, x1)]

sage: Cb.function_field() #
˓→optional - sage.rings.finite_rings
Function field in y defined by y^2 + 2*x^5 + 2*x^4 + x^3 + x + 1

class sage.schemes.curves.projective_curve.ProjectiveCurve(A, X)
Bases: Curve_generic, AlgebraicScheme_subscheme_projective

Curves in projective spaces.

INPUT:

• A – ambient projective space

• X – list of multivariate polynomials; defining equations of the curve

EXAMPLES:
sage: P.<x,y,z,w,u> = ProjectiveSpace(GF(7), 4)
#\rightarrow optional - sage.rings.finite_rings
sage: C = Curve([y*u^2 - x^3, z*u^2 - x^3, w*u^2 - x^3, y^3 - x^3], P); C
#\rightarrow optional - sage.rings.finite_rings
Projective Curve over Finite Field of size 7 defined
by -x^3 + y*u^2, -x^3 + z*u^2, -x^3 + w*u^2, -x^3 + y^3

sage: K.<u> = CyclotomicField(11)
#\rightarrow optional - sage.rings.number_field
sage: P.<x,y,z,w> = ProjectiveSpace(K, 3)
#\rightarrow optional - sage.rings.number_field
sage: C = Curve([y*w - u*z^2 - x^2, x*w - 3*u^2*z*w], P); C
#\rightarrow optional - sage.rings.number_field
Projective Curve over Cyclotomic Field of order 11 and degree 10 defined
by -x^2 + (-u)*z^2 + y*w, x*w + (-3*u^2)*z*w

affine_patch(i, AA=None)

Return the i-th affine patch of this projective curve.

INPUT:

- i – affine coordinate chart of the projective ambient space of this curve to compute affine patch with respect to
- AA – (default: None) ambient affine space, this is constructed if it is not given

OUTPUT: A curve in affine space.

EXAMPLES:

sage: P.<x,y,z,w> = ProjectiveSpace(CC, 3)
sage: C = Curve([y*z - x^2, w^2 - x*y], P)
sage: C.affine_patch(0)
Affine Curve over Complex Field with 53 bits of precision defined
by y*z - 1.00000000000000, w^2 - y

sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: C = Curve(x^3 - x^2*y + y^3 - x^2*z, P)
sage: C.affine_patch(1)
Affine Plane Curve over Rational Field defined by x^3 - x^2*z - x^2 + 1

sage: A.<x,y> = AffineSpace(QQ, 2)
sage: P.<u,v,w> = ProjectiveSpace(QQ, 2)
sage: C = Curve([u^2 - v^2], P)
sage: C.affine_patch(1, A).ambient_space() == A
True

plane_projection(PP=None)

Return a projection of this curve into a projective plane.

INPUT:

- PP – (default: None) the projective plane the projected curve will be defined in. This space must be defined over the same base field as this curve, and must have dimension two. This space is constructed if not specified.
OUTPUT: A tuple of

• a scheme morphism from this curve into a projective plane
• the projective curve that is the image of that morphism

EXAMPLES:

```python
sage: P.<x,y,z,w,u,v> = ProjectiveSpace(QQ, 5)
sage: C = P.curve([x^4 - z^4, w - y, w^2 - x^2, y^3*u^2*z - w^4*w])
sage: L.<a,b,c> = ProjectiveSpace(QQ, 2)
sage: proj1 = C.plane_projection(PP=L)
sage: proj1
(Scheme morphism:
  From: Projective Curve over Rational Field
defined by x^4 - z^4, -y + w, -x^2 + y^2, -w^4 + 2*y^3*z*u
  To:  Projective Space of dimension 2 over Rational Field
  Defn: Defined on coordinates by sending (x : y : z : w : u : v) to
        (x : -z + u : -z + v),
Projective Plane Curve over Rational Field defined by a^8 + 6*a^7*b +
4*a^5*b^3 + 2*a^6*b^2*c - 4*a^5*b^2*c + 2*a^6*c^2)
sage: proj1[1].ambient_space() is L
True
sage: proj2 = C.projection()
sage: proj2[1].ambient_space() is L
False
```

```python
sage: P.<x,y,z,w,u> = ProjectiveSpace(GF(7), 4)

(Scheme morphism:
  From: Projective Curve over Finite Field of size 7
defined by x^2 + y^2, -y^3 - 3*y^2*z + z*w*u, -x^2 + u^2
  To:  Projective Space of dimension 2 over Finite Field of size 7
  Defn: Defined on coordinates by sending (x : y : z : w : u) to
        (x : z : -y + w),
Projective Plane Curve over Finite Field of size 7
defined by x0^10 + 2*x0^8*x1^2 + 2*x0^6*x1^4 - 3*x0^6*x1^3*x2
      + 2*x0^4*x1^2*x2^2 - 2*x0^4*x1^4*x2^2 + x0^2*x1^4*x2^4)
sage: P.<x,y,z> = ProjectiveSpace(GF(17), 2)

(Scheme morphism:
  From: Projective Curve over Finite Field of size 17
defined by x^2 + y^2, -y^3 - 3*y^2*z + z*w*u, -x^2 + u^2
  To:  Projective Space of dimension 2 over Finite Field of size 7
  Defn: Defined on coordinates by sending (x : y : z : w : u) to
        (x : z : -y + w),
Projective Plane Curve over Finite Field of size 7
defined by x0^10 + 2*x0^8*x1^2 + 2*x0^6*x1^4 - 3*x0^6*x1^3*x2
      + 2*x0^4*x1^2*x2^2 - 2*x0^4*x1^4*x2^2 + x0^2*x1^4*x2^4)
sage: C = P.curve(x^2 - y*z - z^2)
sage: C.plane_projection()
Traceback (most recent call last):
  ...TypeError: this curve is already a plane curve
```

```
project(P=None, PS=None)
```

Return a projection of this curve into projective space of dimension one less than the dimension of the
ambient space of this curve.

This curve must not already be a plane curve. Over finite fields, if this curve contains all points in its ambient space, then an error will be returned.

INPUT:

- \( P \) – (default: None) a point not on this curve that will be used to define the projection map; this is constructed if not specified.

- \( P_S \) – (default: None) the projective space the projected curve will be defined in. This space must be defined over the same base ring as this curve, and must have dimension one less than that of the ambient space of this curve. This space will be constructed if not specified.

OUTPUT: A tuple of

- a scheme morphism from this curve into a projective space of dimension one less than that of the ambient space of this curve

- the projective curve that is the image of that morphism

EXAMPLES:

```
sage: K.<a> = CyclotomicField(3)  # optional - sage.rings.number_field
sage: P.<x,y,z,w> = ProjectiveSpace(K, 3)  # optional - sage.rings.number_field
sage: C = Curve([y*z - x^2, z*w^2 - a*x^3], P)  # optional - sage.rings.number_field
sage: L.<a,b,c> = ProjectiveSpace(K, 2)  # optional - sage.rings.number_field
sage: proj1 = C.projection(PS=L)  # optional - sage.rings.number_field
sage: proj1[1].ambient_space() is L
True
sage: proj2 = C.projection()  # optional - sage.rings.number_field
sage: proj2[1].ambient_space() is L
False
```

```
sage: P.<x,y,z,w,a,b,c> = ProjectiveSpace(QQ, 6)
sage: C = Curve([y - x, z - a - b, w^2 - c^2, z - x - a, x^2 - w*z], P)
sage: C.projection()  # optional - sage.rings.number_field
(Scheme morphism:
  From: Projective Curve over Rational Field
  defined by -x + y, z - a - b, w^2 - c^2, -x + z - a, x^2 - z*w
  To: Projective Space of dimension 5 over Rational Field
  (continues on next page)
```
Defn: Defined on coordinates by sending \((x : y : z : w : a : b : c)\)
to \((x : y : -z + w : a : b : c)\),
Projective Curve over Rational Field defined by 
\(x_1 - x_4, x_0 - x_4, x_2^2 x_3 + x_3^2 + x_2^2 x_4 + 2 x_3 x_4^3 x_4, x_2 x_4^2 - x_3^2 - 2 x_3^2 x_4 + x_4^2 - x_5^2, x_2^2 x_4^2 + x_3^4 x_4^2 + x_4^3 x_3^2 - x_3^2 x_4^3 - x_4^2 x_5^2, x_4^4 - x_3^2 x_5^2 - 2 x_3^2 x_4^3 x_5^2 - x_4^2 x_5^2)\)

sage: P.<x,y,z,w> = ProjectiveSpace(GF(2), 3)  
    #_optional - sage.rings.finite_rings
sage: C = P.curve([[x - y] * (x - z) * (x - w) * (y - z) * (y - w)],  
    #_optional - sage.rings.finite_rings
    ....: x * y * z * w * (x + y + z + w))  
    #_optional - sage.rings.finite_rings
sage: C.projection()  
    #_optional - sage.rings.finite_rings
Traceback (most recent call last):
NotImplementedError: this curve contains all points of its ambient space

sage: P.<x,y,z,w,u> = ProjectiveSpace(GF(7), 4)  
    #_optional - sage.rings.finite_rings
sage: C = P.curve([x^3 - y^2 * z * u, w^2 - u^2 + 2^4 * z, 3^4 * w - y^2])  
    #_optional - sage.rings.finite_rings
sage: L.<a,b,c,d> = ProjectiveSpace(GF(7), 3)  
    #_optional - sage.rings.finite_rings
sage: C.projection(PS=L)  
    #_optional - sage.rings.finite_rings
(Scheme morphism:
From: Projective Curve over Finite Field of size 7
defined by x^3 - y^2 * z * u, 2^4 * z + w^2 - u^2, -y^2 + 3^4 * w
To: Projective Space of dimension 3 over Finite Field of size 7
Defn: Defined on coordinates by sending \((x : y : z : w : u)\) to
\((x : y : z : w)\),
Projective Curve over Finite Field of size 7 defined by b^2 - 3*a*d, 
a^5*b + a*b*c^3*d - 3*b*c^2*d^3, a^6 + a^2*c^3*d - 3*a*c^2*d^3)  

sage: Q.<a,b,c> = ProjectiveSpace(GF(7), 2)  
    #_optional - sage.rings.finite_rings
sage: C.projection(PS=Q)  
    #_optional - sage.rings.finite_rings
Traceback (most recent call last):
TypeError: (=Projective Space of dimension 2 over Finite Field of size 7) must have dimension (=3)

sage: PP.<x,y,z,w> = ProjectiveSpace(QQ, 3)
sage: C = PP.curve([x^3 - z^2 * y, w^2 - z^2 * x])
sage: Q = PP([1,0,1,1])
sage: C.projection(P=Q)
(Scheme morphism:
From: Projective Curve over Rational Field
defined by x^3 - y*z^2, -x*z + w^2
To: Projective Space of dimension 2 over Rational Field
(continues on next page)
Defn: Defined on coordinates by sending \((x : y : z : w)\) to 
\((y : -x + z : -x + w)\),
Projective Plane Curve over Rational Field defined by 
\[x0*x1^5 - 6*x0*x1^4*x2 + 14*x0*x1^3*x2^2 - 16*x0*x1^2*x2^3 + 9*x0*x1*x2^4 - 2*x0*x2^5 - x2^6\]
sage: LL.<a,b,c> = ProjectiveSpace(QQ, 2)
sage: Q = PP([0,0,0,1])
sage: C.projection(PS=LL, P=Q)
(Scheme morphism:
    From: Projective Curve over Rational Field
defined by \(x^3 - y*z^2, -x*z + w^2\)
    To: Projective Space of dimension 2 over Rational Field
    Defn: Defined on coordinates by sending \((x : y : z : w)\) to 
    \((x : y : z)\),
Projective Plane Curve over Rational Field defined by \(a^3 - b*c^2\)
sage: Q = PP([0,0,1,0])
sage: C.projection(P=Q)
Traceback (most recent call last):
... TypeError: \((0 : 0 : 1 : 0)\) must be a point not on this curve

sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: C = P.curve(y^2 - x^2 + z^2)
sage: C.projection()
Traceback (most recent call last):
... TypeError: this curve is already a plane curve

class sage.schemes.curves.projective_curve.ProjectiveCurve_field(A, X)
Bases: ProjectiveCurve, AlgebraicScheme_subscheme_projective_field

Projective curves over fields.

arithmetic_genus()

Return the arithmetic genus of this projective curve.

This is the arithmetic genus \(g_a(C)\) as defined in [Har1977]. If \(P\) is the Hilbert polynomial of the defining ideal of this curve, then the arithmetic genus of this curve is \(1 - P(0)\). This curve must be irreducible.

EXAMPLES:

sage: P.<x,y,z,w> = ProjectiveSpace(QQ, 3)
sage: C = P.curve([w*z - x^2, w^2 + y^2 + z^2])
sage: C.arithmetic_genus()
1

sage: P.<x,y,z,w,t> = ProjectiveSpace(GF(7), 4)
    # optional - sage.rings.finite_rings
sage: C = P.curve([t^3 - x^2*w, x^3 + y^3 + z^3, z - w])
    # optional - sage.rings.finite_rings
sage: C.arithmetic_genus()
# 10

Chapter 4. Projective curves
**is_complete_intersection()**

Return whether this projective curve is a complete intersection.

**EXAMPLES:**

```python
sage: P.<x,y,z,w> = ProjectiveSpace(QQ, 3)
sage: C = Curve([x*y - z*w, x^2 - y*w, y^2*w - x*z*w], P)
sage: C.is_complete_intersection()
False
```

```python
sage: P.<x,y,z,w> = ProjectiveSpace(QQ, 3)
sage: C = Curve([y^2*w - x^2, z^2*w - x^2], P)
sage: C.is_complete_intersection()
True
```

```python
sage: P.<x,y,z,w> = ProjectiveSpace(QQ, 3)
sage: C = Curve([z^2 - y*w, y*z - x*w, y^2 - x*z], P)
sage: C.is_complete_intersection()
False
```

**tangent_line(p)**

Return the tangent line at the point p.

**INPUT:**

- p – a rational point of the curve

**EXAMPLES:**

```python
sage: P.<x,y,z,w> = ProjectiveSpace(QQ, 3)
sage: C = Curve([x*y - z*w, x^2 - y*w, y^2*w - x*z*w], P)
sage: p = C(1,1,1,1)
sage: C.tangent_line(p)
Projective Curve over Rational Field
defined by -2*x + y + w, -3*x + z + 2*w
```

**class** `sage.schemes.curves.projective_curve.ProjectivePlaneCurve(A,f)`

Bases: `ProjectiveCurve`

Curves in projective planes.

**INPUT:**

- A – projective plane
- f – homogeneous polynomial in the homogeneous coordinate ring of the plane

**EXAMPLES:**

A projective plane curve defined over an algebraic closure of Q:

```python
sage: P.<x,y,z> = ProjectiveSpace(QQbar, 2)  #←
←optional – sage.rings.number_field
sage: set_verbose(-1)  #← optional – sage.rings.number_field
←supress warnings for slow computation
←optional – sage.rings.number_field
sage: C = Curve([y*z - x^2 - QQbar.gen()*z^2], P); C  #←
←optional – sage.rings.number_field
```

(continues on next page)
A projective plane curve defined over a finite field:

```
sage: P.<x,y,z> = ProjectiveSpace(GF(5^2, 'v'), 2)  # P
  → optional - sage.rings.finite_rings
sage: C = Curve([y^2*z - x*z^2 - z^3], P); C  # C
  → optional - sage.rings.finite_rings
```

Projective Plane Curve over Finite Field in v of size 5^2 defined by y^2*z - x*z^2 - z^3

**degree()**

Return the degree of this projective curve.

For a plane curve, this is just the degree of its defining polynomial.

OUTPUT: An integer.

EXAMPLES:

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: C = P.curve([y^7 - x^2*z^5 + 7*z^7])
sage: C.degree()
7
```

**divisor_of_function(r)**

Return the divisor of a function on a curve.

INPUT: r is a rational function on X

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

(continues on next page)
sage: r = 1/x^3
˓→optional - sage.rings.finite_rings
sage: C.divisor_of_function(r)  # todo: not implemented
˓→optional - sage.rings.finite_rings
[[[-3, (0, 0, 1)]

excellent_position(Q)

Return a transformation of this curve into one in excellent position with respect to the point Q.

Here excellent position is defined as in [Ful1989]. A curve $C$ of degree $d$ containing the point $(0 : 0 : 1)$ with multiplicity $r$ is said to be in excellent position if none of the coordinate lines are tangent to $C$ at any of the fundamental points $(1 : 0 : 0)$, $(0 : 1 : 0)$, and $(0 : 0 : 1)$, and if the two coordinate lines containing $(0 : 0 : 1)$ intersect $C$ transversally in $d - r$ distinct non-fundamental points, and if the other coordinate line intersects $C$ transversally at $d$ distinct, non-fundamental points.

INPUT:

• $Q$ – a point on this curve.

OUTPUT:

• a scheme morphism from this curve to a curve in excellent position that is a restriction of a change of coordinates map of the projective plane.

EXAMPLES:

sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: C = Curve([x*y - z^2], P)
sage: Q = P([1,1,1])
sage: C.excellent_position(Q)
Scheme morphism:
From: Projective Plane Curve over Rational Field defined by x*y - z^2
To: Projective Plane Curve over Rational Field defined by -x^2 - 3*x*y - 4*y^2 - x*z - 3*y*z
Defn: Defined on coordinates by sending (x : y : z) to (-x + 1/2*y + 1/2*z : -1/2*y + 1/2*z : x + 1/2*y - 1/2*z)

sage: R.<a> = QQ[]
sage: K.<b> = NumberField(a^2 - 3)
˓→optional - sage.rings.number_field
sage: P.<x,y,z> = ProjectiveSpace(K, 2)
˓→optional - sage.rings.number_field
sage: C = P.curve([z^2*y^3*x^4 - y^6*x^3 - 4*z^2*y^4*x^3 - 4*z^4*y^2*x^3
˓→...:+ 3*y^7*x^2 + 10*z^2*y^5*x^2 + 9*z^4*y^3*x^2
˓→...:+ 5*z^6*y*x^2 - 3*y^8*x - 9*z^2*y^6*x - 11*z^4*y^4*x
˓→...:- 7*z^6*y^2*x - 2*z^8*x + y^9 + 2*z^2*y^7 + 3*z^4*y^5
˓→...:+ 4*z^6*y^3 + 2*z^8*y])
sage: Q = P([1,0,0])
˓→optional - sage.rings.number_field
sage: C.excellent_position(Q)
˓→optional - sage.rings.number_field
Scheme morphism:
From: Projective Plane Curve over Number Field in b
with defining polynomial a^2 - 3

(continues on next page)
Curves, Release 10.1

(continued from previous page)

defined by \(-x^3*y^6 + 3*x^2*y^7 - 3*x*y^8 + y^9 + x^4*y^3*z^2\)
- \(4*x^3*y^4*z^2 + 10*x^2*y^5*z^2 - 9*x*y^6*z^2\)
+ \(2*y^7*z^2 - 4*x^3*y^2*z^4 + 9*x^2*y^3*z^4\)
- \(11*x*y^4*z^4 + 3*y^5*z^4 + 5*x^2*y^5*z^6\)
- \(7*x*y^2*z^6 + 4*y^3*z^6 - 2*x*z^8 + 2*y*z^8\)

To: Projective Plane Curve over Number Field in \(b\)
with defining polynomial \(a^2 - 3\)
defined by \(900*x^9 - 7410*x^8*y + 29282*x^7*y^2 - 69710*x^6*y^3\)
+ \(110818*x^5*y^4 - 30756*x^6*y^5 + 58692*x^5*y^6\)
- \(75600*x^4*y^7 - 67916*x^3*y^8 + 42364*x^2*y^9\)
+ \(16844*x*y^10 - 3586*y^11 + 786*x^7*z\)
- \(3958*x^6*y*z^2 + 9746*x^5*y^2*z^2 - 14694*x^4*y^3*z^2\)
+ \(15174*x^3*y^4*z^2 - 10802*x^2*y^5*z^2\)
+ \(5014*x*y^6*z^2 - 1266*y^7*z^2 - 144*x^6*z^3\)
+ \(512*x^5*y*z^3 - 16*x^4*y^2*z^3 + 2*y^5*z^3\)
+ \(8*x^5*y^2*z^3 - 8*x^4*y*z^3 - 16*x^3*y^2*z^4\)
+ \(16*x^2*y^3*z^4 + 8*y^4*z^4 - 8*y^5*z^4\)

Defn: Defined on coordinates by sending \((x : y : z)\) to
\(1/4*y + 1/2*z : -1/4*y + 1/2*z : x + 1/4*y - 1/2*z\)

\[\text{sage: set_verbose(-1)}\]
\[\text{sage: a = QQbar(sqrt(2))}\]
[...]
\[\text{sage: P.<x,y,z> = ProjectiveSpace(QQbar, 2)}\]
[...]
\[\text{sage: C = Curve([(-1/4*a)*x^3 + (-3/4*a)*x^2*y + (-3/4*a)*x*y^2 + (-1/4*a)*y^3 - 2*x*y*z], P)}\]
\[\text{sage: Q = P([0,0,1])}\]
[...]
\[\text{sage: C.excellent_position(Q)}\]

Scheme morphism:
From: Projective Plane Curve over Algebraic Field defined by
\((-0.35355339059327378)*x^3 + (-1.606601717798227)\times^2*y\)
+ \((-1.0606601717798227)*x*y^2 + (-0.35355339059327378)*y^3\)
+ \((-2)*x*y^z\)

To: Projective Plane Curve over Algebraic Field defined by
\((-2.8284271247461907)*x^3 + (-2)*x^2*y + 2*y^3\)
+ \((-2)*x^2*y + 2*y^2*z\)

Defn: Defined on coordinates by sending \((x : y : z)\) to
\((1/2*x + 1/2*y : (-1/2)*x + 1/2*y : 1/2*x + (-1/2)*y + z)\)

\[\text{is_ordinary_singularity}(P)\]
Return whether the singular point \(P\) of this projective plane curve is an ordinary singularity.

The point \(P\) is an ordinary singularity of this curve if it is a singular point, and if the tangents of this curve at \(P\) are distinct.

INPUT:
• \( P \) – a point on this curve.

OUTPUT:

• Boolean. True or False depending on whether \( P \) is or is not an ordinary singularity of this curve, respectively. An error is raised if \( P \) is not a singular point of this curve.

EXAMPLES:

```python
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: C = Curve([y^2*z^3 - x^5], P)
sage: Q = P([0,0,1])
sage: C.is_ordinary_singularity(Q)
False
```

```python
sage: R.<a> = QQ[]
sage: K.<b> = NumberField(a^2 - 3)  # optional - sage.rings.number_field
sage: P.<x,y,z> = ProjectiveSpace(K, 2)  # optional - sage.rings.number_field
sage: C = P.curve([x^2*y^3*z^4 - y^6*z^3 - 4*x^2*y^4*z^3 - 4*x^4*y^2*z^3])  # optional - sage.rings.number_field
sage: Q = P([0,1,1])  # optional - sage.rings.number_field
sage: C.is_ordinary_singularity(Q)  # optional - sage.rings.number_field
True
```

```python
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: C = P.curve([z^5 - y^5 + x^5 + x*y^2*z^2])
sage: Q = P([0,1,1])
sage: C.is_ordinary_singularity(Q)
Traceback (most recent call last):
  ... TypeError: (=Projective Plane Curve over Rational Field defined by x^5 - y^5 + x*y^2*z^2 + z^5)
```

`is_singular(P=None)`

Return whether this curve is singular or not, or if a point \( P \) is provided, whether \( P \) is a singular point of this curve.

INPUT:

• \( P \) – (default: None) a point on this curve

OUTPUT:

If no point \( P \) is provided, return True or False depending on whether this curve is singular or not. If a point \( P \) is provided, return True or False depending on whether \( P \) is or is not a singular point of this curve.

EXAMPLES:

Over \( Q \):

4.2. Integral projective curves over \( Q \)
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 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 github issue #12187 is fixed:

sage: F.<X,Y,Z> = GF(2)[]

sage: G = Curve(X^2 + Y*Z)

(continues on next page)
sage: G.is_singular()  # optional - sage.rings.finite_rings
False

sage: P.<x,y,z> = ProjectiveSpace(CC, 2)
sage: C = Curve([y^4 - x^3*z], P)
sage: Q = P([0,0,1])
sage: C.is_singular()
True

is_transverse(C, P)

Return whether the intersection of this curve with the curve C at the point P is transverse.

The intersection at P is transverse if P is a nonsingular point of both curves, and if the tangents of the curves at P are distinct.

INPUT:

* C – a curve in the ambient space of this curve.
* P – a point in the intersection of both curves.

OUTPUT: A boolean.

EXAMPLES:

sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: C = Curve([x^2 - y^2], P)
sage: D = Curve([x - y], P)
sage: Q = P([1,1,0])
sage: C.is_transverse(D, Q)
False

sage: K = QuadraticField(-1)
# optional - sage.rings.number_field
sage: P.<x,y,z> = ProjectiveSpace(K, 2)
# optional - sage.rings.number_field
sage: C = Curve([y^2*z - K.0*x^3], P)
# optional - sage.rings.number_field
sage: D = Curve([z*x + y^2], P)
# optional - sage.rings.number_field
sage: Q = P([0,0,1])
# optional - sage.rings.number_field
sage: C.is_transverse(D, Q)
False

sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: C = Curve([x^2 - 2*y^2 - 2*z^2], P)
sage: D = Curve([y - z], P)
sage: Q = P([2,1,1])
sage: C.is_transverse(D, Q)
True

4.2. Integral projective curves over \( \mathbb{Q} \)
local_coordinates($pt, n$)

Return local coordinates to precision $n$ at the given point.

Behaviour is flaky - some choices of $n$ are worse than others.

**INPUT:**

- **$pt$** – a rational point on $X$ which is not a point of ramification for the projection $(x, y) \rightarrow x$.
- **$n$** – the number of terms desired

**OUTPUT:** $x = x_0 + t$, $y = y_0 + \text{power series in } t$

**EXAMPLES:**

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

ordinary_model()

Return a birational map from this curve to a plane curve with only ordinary singularities.

Currently only implemented over number fields. If not all of the coordinates of the non-ordinary singularities of this curve are contained in its base field, then the domain and codomain of the map returned will be defined over an extension. This curve must be irreducible.

**OUTPUT:**

- a scheme morphism from this curve to a curve with only ordinary singularities that defines a birational map between the two curves.

**EXAMPLES:**

```sage
sage: setVerbose(-1)
sage: K = QuadraticField(3)  # optional - sage.rings.number_field
sage: P.<x,y,z> = ProjectiveSpace(K, 2)  # optional - sage.rings.number_field
sage: C = Curve([x^5 - K.0*y*z^4], P)  # optional - sage.rings.number_field
sage: C.ordinary_model()  # optional - sage.rings.number_field
Scheme morphism:
  From: Projective Plane Curve over Number Field in a
  with defining polynomial x^2 - 3 with a = 1.732050807568878?
  defined by x^5 + (-a)*y*z^4
(continues on next page)```
To: Projective Plane Curve over Number Field in $a$ with defining polynomial $x^2 - 3$ with $a = 1.732050807568878$?
defined by $-a)x^5y + (-4a)x^4y^2 + (-6a)x^3y^3 + (-4a)x^2y^4 + (-a)x*y^5 + (-a - 1)x^5z + (-a + 5)x^4y*z + (-6a - 10)x^3y^2z + (-4a - 10)x^2y^3z + (-a - 5)x*y^4z + y^5z$
Defn: Defined on coordinates by sending $(x : y : z)$ to
$(-1/4)x^2 + 1/2x^2y + 1/2x^2z - 1/4x^2z^2 : 1/4x^2 + 1/2x^2y + 1/2x^2z - 1/4x^2z^2 : -1/4x^2 + 1/4x^2z^2)$

sage: set_verbose(-1)
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: C = Curve([y^2*z^2 - x^4 - x^3*z], P)
sage: D = C.ordinary_model(); D
# long time (2 seconds)
Scheme morphism:
From: Projective Plane Curve over Rational Field defined by $-x^4 - x^3*z + y^2*z^2$
To: Projective Plane Curve over Rational Field defined by $4*x^6*y^3 - 24*x^5*y^4 + 36*x^4*y^5 + 8*x^6*y^2*z - 40*x^5*y^3*z + 24*x^4*y^4*z + 72*x^3*y^5*z - 4*x^6*y*z^2 + 8*x^5*y^2*z^2 - 56*x^4*y^3*z^2 + 104*x^3*y^4*z^2 + 44*x^2*y^5*z^2 + 8*x^6*z^3 - 16*x^5*y*z^3 + 40*x^4*y^2*z^3 + 48*x^3*y^3*z^3 + 44*x^2*y^4*z^3 + 8*x*y^5*z^3 - 8*x^6*z^4 + 36*x^4*y^4*z^4 - 56*x^3*y^3*z^4 - 20*x^2*y^2*z^4 + 40*x*y^3*z^4 - 16*y^5*z^4$
Defn: Defined on coordinates by sending $(x : y : z)$ to
$(-3/64*x^4 + 9/64*x^2*y^2 - 1/32*x*y^3 - 1/16x^3*z - 1/8*x^2*y^2*z - 1/16*y^3*z - 1/8*x^2*y^2*z^2 + 1/16*y^3*z^2 + 1/16*y^2*z^2 : -1/64*x^4 + 3/64*x^2*y^2*z - 1/32*x*y^3 + 1/16*x^2*y^2*z + 1/16*y^3*z + 1/16*y^2*z^2 : 3/64*x^4 - 3/32*x^3*y + 3/32*x^2*y^2 + 1/16*x^3*z - 3/16*x^2*y^2*z + 1/8*x^2*y^2*z - 1/8*x^2*y^2*z^2 + 1/16*y^2*z^2)$

sage: all(D.codomain().is_ordinary_singularity(Q) for Q in D.codomain().singular_points())
True

sage: set_verbose(-1)
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: C = Curve([(x^2 + y^2 - y*z - 2*z^2)*z + y^5], P)
sage: C.ordinary_model()
# long time (5 seconds)
Scheme morphism:
From: Projective Plane Curve over Number Field in $a$ with defining polynomial $y^2 - 2$ defined by $y^5 - x^4*z - x^2*y^2*z + 2*x^2*y*z^2 + y^3*z^2 + 4*x^2*z^3 + y^2*z^3 - 4*y*z^4 - 4*z^5$
To: Projective Plane Curve over Number Field in $a$ with defining polynomial $y^2 - 2$ defined by $(-29*a + 1)*x^8*y^6 + (10*a + 158)*x^7*y^7 + (-109*a - 31)*x^6*y^8 + (-80*a - 198)*x^5*y^5*z$

(continues on next page)
+ (531*a + 272)*x^7*y^6*z + (170*a - 718)*x^6*y^8*z^2
+ (1557*a - 809)*x^8*y^3*z^2 + (343*a - 1329)*x^4*y^8*z^2
+ (19*a - 636)*x^5*y^8*z + (-200*a - 628)*x^8*y^4*z^2
+ (1557*a - 114)*x^7*y^5*z^2 + (330*a - 1083)*x^3*y^8*z^3
+ (-259*a - 524)*x^8*y*z^4 + (720*a - 605)*x^7*y^3*z^4
+ (308*a - 211)*x^6*y^4*z^4 + (4548*a - 5462)*x^5*y^5*z^4
+ (2958*a - 6611)*x^4*y^6*z^4 + (3904*a - 7110)*x^5*y^6*z^4
+ (1789*a - 5161)*x^4*y^7*z^4 + (330*a - 1083)*x^3*y^8*z^4
+ (-259*a - 524)*x^8*y*z^5 + (720*a - 605)*x^7*y^2*z^5
+ (308*a - 211)*x^6*y^3*z^5 + (4548*a - 5462)*x^5*y^4*z^5
+ (2958*a - 6611)*x^4*y^5*z^5 + (3904*a - 7111)*x^3*y^6*z^5
+ (1789*a - 5161)*x^2*y^7*z^5

Defn: Defined on coordinates by sending (x : y : z) to

((-5/128*a - 5/128)*x^4 + (-5/32*a + 5/32)*x^3*y
+ (1/16*a - 1/16)*x^2*y^2 + (1/16*a - 1/16)*x*y^3
+ (1/32*a - 1/32)*y^4) : (-5/128*a - 5/128)*x^4 + (5/32*a)*x^3*y
+ (1/32*a - 1/32)*x^2*y^2 + (1/32*a - 1/32)*x*y^3
+ (1/32*a - 1/32)*y^4)

EXAMPLES:

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()  # optional - sage.plot
Graphics object consisting of 1 graphics primitive
```

The other affine patches of the same curve:

```
sage: C.plot(patch=0)  # optional - sage.plot
Graphics object consisting of 1 graphics primitive
sage: C.plot(patch=1)  # optional - sage.plot
Graphics object consisting of 1 graphics primitive
```

An elliptic curve:

```
sage: E = EllipticCurve('101a')
sage: C = Curve(E)
sage: C.plot()  # optional - sage.plot
Graphics object consisting of 1 graphics primitive
sage: C.plot(patch=0)  # optional - sage.plot
Graphics object consisting of 1 graphics primitive
sage: C.plot(patch=1)  # optional - sage.plot
Graphics object consisting of 1 graphics primitive
```

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()  # optional - sage.plot
Graphics object consisting of 1 graphics primitive
sage: C.plot(patch=0)  # optional - sage.plot
Graphics object consisting of 1 graphics primitive
sage: C.plot(patch=1)  # optional - sage.plot
Graphics object consisting of 1 graphics primitive
```

`quadratic_transform()`

Return a birational map from this curve to the proper transform of this curve with respect to the standard Cremona transformation.

The standard Cremona transformation is the birational automorphism of \( P^2 \) defined \( (x : y : z) \mapsto (yz : xz : xy) \).

OUTPUT:
• a scheme morphism representing the restriction of the standard Cremona transformation from this
curve to the proper transform.

EXAMPLES:

```python
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: C = Curve(x^3*y - z^4 - z^2*x^2, P)
sage: C.quadratic_transform()
Scheme morphism:
  From: Projective Plane Curve over Rational Field
defined by x^3*y - x^2*z^2 - z^4
  To:  Projective Plane Curve over Rational Field
defined by -x^3*y - x*y*z^2 + z^4
  Defn: Defined on coordinates by sending (x : y : z) to
           (y*z : x*z : x*y)
```

```python
sage: P.<x,y,z> = ProjectiveSpace(GF(17), 2)
sage: C = P.curve([y^7*z^2 - 16*x^9 + x*y*z^7 + 2*z^9])
sage: C.quadratic_transform()
Scheme morphism:
  From: Projective Plane Curve over Finite Field of size 17
defined by x^9 + y^7*z^2 + x*y*z^7 + 2*z^9
  To:  Projective Plane Curve over Finite Field of size 17
defined by 2*x^9*y^7 + x^8*y^6*z^2 + x^9*z^7 + y^7*z^9
  Defn: Defined on coordinates by sending (x : y : z) to
           (y*z : x*z : x*y)
```

tangents(P, factor=True)

Return the tangents of this projective plane curve at the point P.

These are found by homogenizing the tangents of an affine patch of this curve containing P. The point P
must be a point on this curve.

INPUT:

• P – a point on this curve.

• factor – (default: True) whether to attempt computing the polynomials of the individual tangent
  lines over the base field of this curve, or to just return the polynomial corresponding to the union
  of the tangent lines (which requires fewer computations).

OUTPUT:

A list of polynomials in the coordinate ring of the ambient space of this curve.

EXAMPLES:

```python
sage: set_verbose(-1)
sage: P.<x,y,z> = ProjectiveSpace(QQbar, 2)

sage: C = Curve([x^3*y + 2*x^2*y^2 + x*y^3 + x^3*z
            + 7*x^2*y*z + 14*x*y^2*z + 9*y^3*z], P)
sage: Q = P([0,0,1])
```
optional - sage.rings.number_field

sage: C.tangents(Q)
\[
x + 4.147899035704788?*y,
\]
\[
x + (1.426050482147607? + 0.3689894074818041?*I)*y,
\]
\[
x + (1.426050482147607? - 0.3689894074818041?*I)*y
\]

optional - sage.rings.number_field

sage: C.tangents(Q, factor=False)
\[
[6*x^3 + 42*x^2*y + 84*x*y^2 + 54*y^3]
\]

optional - sage.rings.number_field

sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: C = P.curve([x^2*y^3*z^4 - y^6*z^3 - 4*x^2*y^4*z^3 - 4*x^4*y^2*z^3 + 3*y^7*z^2 + 10*x^2*y^5*z^2 + 9*x^4*y^3*z^2 + 5*x^6*y*z^2 - 3*y^8*z - 9*x^2*y^6*z - 11*x^4*y^4*z - 7*x^6*y^2*z - 2*x^8*y])
sage: Q = P([0,1,1])
sage: C.tangents(Q)
\[-y + z, 3*x^2 - y^2 + 2*y*z - z^2\]

optional - sage.rings.number_field

sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: C = P.curve([z^3*x + y^4 - x^2*z^2])
sage: Q = P([1,1,1])
sage: C.tangents(Q)

Traceback (most recent call last):
...

TypeError: (=((1 : 1 : 1)) is not a point on (=Projective Plane Curve over Rational Field defined by y^4 - x^2*z^2)

class sage.schemes.curves.projective_curve.ProjectivePlaneCurve_field(A,f)

Bases: ProjectivePlaneCurve, ProjectiveCurve_field

Projective plane curves over fields.

arithmetic_genus()

Return the arithmetic genus of this projective curve.

This is the arithmetic genus \( g_a(C) \) as defined in [Har1977]. For a projective plane curve of degree \( d \), this is simply \((d - 1)(d-2)/2\). It need not equal the geometric genus (the genus of the normalization of the curve). This curve must be irreducible.

EXAMPLES:

sage: x,y,z = PolynomialRing(GF(5), 3, 'xyz').gens()  #...
sage: C = Curve(y^2*z^7 - x^9 - x*z^8); C  #...

Projective Plane 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()  #...

(continues on next page)

4.2. Integral projective curves over \( \mathbb{Q} \) 69
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: C = Curve([y^3*x - x^2*y*z - 7*z^4])
sage: C.arithmetic_genus()
3

fundamental_group()

Return a presentation of the fundamental group of the complement of self.

Note: The curve must be defined over the rationals or a number field with an embedding over \( \mathbb{Q} \).

EXAMPLES:

sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: C = P.curve(x^2*z - y^3)
sage: C.fundamental_group()             # optional - sirocco
Finitely presented group < x0 | x0^3 >

In the case of number fields, they need to have an embedding into the algebraic field:

sage: a = QQ[x](x^2 + 5).roots(QQbar)[0][0] # optional - sage.rings.number_field
sage: a # optional - sage.rings.number_field
-2.236067977499790?*I
sage: F = NumberField(a.minpoly(), 'a', embedding=a) # optional - sage.rings.number_field
sage: P.<x,y,z> = ProjectiveSpace(F, 2) # optional - sage.rings.number_field
sage: F.inject_variables() # optional - sage.rings.number_field
Defining a
sage: C = P.curve(x^2 + a * y^2) # optional - sage.rings.number_field
sage: C.fundamental_group()       # optional - sirocco
# optional - sirocco

Warning: This functionality requires the sirocco package to be installed.

rational_parameterization()

Return a rational parameterization of this curve.

This curve must have rational coefficients and be absolutely irreducible (i.e. irreducible over the algebraic closure of the rational field). The curve must also be rational (have geometric genus zero).

The rational parameterization may have coefficients in a quadratic extension of the rational field.

OUTPUT:
• a birational map between $\mathbb{P}^1$ and this curve, given as a scheme morphism.

EXAMPLES:

```sage
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: C = Curve([y^2*z - x^3], P)
sage: C.rational_parameterization()
Scheme morphism:
  From: Projective Space of dimension 1 over Rational Field
  To:  Projective Plane Curve over Rational Field
defn: Defined on coordinates by sending (s : t) to
        (s^2*t : s^3 : t^3)
```

```sage
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: C = Curve([x^3 - 4*y*z^2 + x*z^2 - x*y*z], P)
sage: C.rational_parameterization()
Scheme morphism:
  From: Projective Space of dimension 1 over Rational Field
  To:  Projective Plane Curve over Rational Field
defn: Defined on coordinates by sending (s : t) to
        (4*s^2*t + s*t^2 : s^2*t + t^3 : 4*s^3 + s^2*t)
```

```sage
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: C = Curve([x^2 + y^2 + z^2], P)
sage: C.rational_parameterization()  # optional - sage.rings.number_field
Scheme morphism:
  From: Projective Space of dimension 1 over Number Field in a
  with defining polynomial a^2 + 1
  To:  Projective Plane Curve over Number Field in a
  with defining polynomial a^2 + 1 defined by x^2 + y^2 + z^2
defn: Defined on coordinates by sending (s : t) to
        ((-a)*s^2 + (-a)*t^2 : s^2*t : t^3 : 2*s^3)
```

`riemann_surface(**kwargs)`  
Return the complex Riemann surface determined by this curve  

**OUTPUT**: A `RiemannSurface` object.

**EXAMPLES**:

```sage
sage: R.<x,y,z> = QQ[]
sage: C = Curve(x^3 + 3*y^3 + 5*z^3)
sage: C.riemann_surface()
Riemann surface defined by polynomial f = x^3 + 3*y^3 + 5 = 0, 
with 53 bits of precision
```

**class** `sage.schemes.curves.projective_curve.ProjectivePlaneCurve_finite_field(A,f)`  
**Bases**: `ProjectivePlaneCurve_field`  
Projective plane curves over finite fields  
**method** `rational_points(algorithm='enum', sort=True)`  
Return the rational points on this curve.
INPUT:

• algorithm – one of
  – 'enum' – straightforward enumeration

• 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, possibly sorted.

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

EXAMPLES:

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

sage: C = Curve(x - y + z)  # optional - sage.rings.finite_rings
sage: C.rational_points()  # optional - sage.rings.finite_rings
[(0 : 1 : 1), (1 : 1 : 0), (1 : 2 : 1), (2 : 3 : 1), (3 : 4 : 1), (4 : 0 : 1), (4 : 1 : 0), (4 : 1 : 1), (4 : 2 : 1), (4 : 3 : 1), (4 : 4 : 1)]

sage: C = Curve(x*z + z^2)  # optional - sage.rings.finite_rings
sage: C.rational_points('all')  # optional - sage.rings.finite_rings
[(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)]

sage: F = GF(7)  # optional - sage.rings.finite_rings
sage: P2.<X,Y,Z> = ProjectiveSpace(F, 2)  # optional - sage.rings.finite_rings
sage: C = Curve(X^3 + Y^3 - Z^3)  # optional - sage.rings.finite_rings
sage: C.rational_points()  # optional - sage.rings.finite_rings
[(0 : 1 : 0), (0 : 2 : 1), (0 : 4 : 1), (1 : 0 : 1), (2 : 0 : 1), (3 : 1 : 0), (4 : 0 : 1), (5 : 1 : 0), (6 : 1 : 0)]
```
sage: F = GF(1237)
#optional - sage.rings.finite_rings
sage: P2.<X,Y,Z> = ProjectiveSpace(F, 2)
#optional - sage.rings.finite_rings
sage: C = Curve(X^7 + 7*Y^6*Z + Z^4*X^2*Y*89)
#optional - sage.rings.finite_rings
sage: len(C.rational_points())
#optional - sage.rings.finite_rings
1237

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

sage: R.<x,y,z> = GF(2)[]
#optional - sage.rings.finite_rings
sage: f = x^3*y + y^3*z + x*z^3
#optional - sage.rings.finite_rings
sage: C = Curve(f); pts = C.rational_points()
#optional - sage.rings.finite_rings
sage: pts
#optional - sage.rings.finite_rings
[(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.

EXAMPLES:
sage: F = GF(2)  #
→optional - sage.rings.finite_rings
sage: P2.<X,Y,Z> = ProjectiveSpace(F, 2)  #
→optional - sage.rings.finite_rings
sage: C = Curve(X*Y*Z)  #
→optional - sage.rings.finite_rings
sage: a = C.rational_points_iterator()  #
→optional - sage.rings.finite_rings
sage: next(a)  #
→optional - sage.rings.finite_rings
(1 : 0 : 0)
sage: next(a)  #
→optional - sage.rings.finite_rings
(0 : 1 : 0)
sage: next(a)  #
→optional - sage.rings.finite_rings
(1 : 1 : 0)
sage: next(a)  #
→optional - sage.rings.finite_rings
(0 : 0 : 1)
sage: next(a)  #
→optional - sage.rings.finite_rings
(1 : 0 : 1)
sage: next(a)  #
→optional - sage.rings.finite_rings
(0 : 1 : 1)
sage: next(a)  #
→optional - sage.rings.finite_rings
Traceback (most recent call last):
...
StopIteration

sage: F = GF(3^2,'a')  #
→optional - sage.rings.finite_rings
sage: P2.<X,Y,Z> = ProjectiveSpace(F, 2)  #
→optional - sage.rings.finite_rings
sage: C = Curve(X^3 + 5*Y^2*Z - 33*X*Y*X)  #
→optional - sage.rings.finite_rings
sage: b = C.rational_points_iterator()  #
→optional - sage.rings.finite_rings
sage: next(b)  #
→optional - sage.rings.finite_rings
(0 : 1 : 0)
sage: next(b)  #
→optional - sage.rings.finite_rings
(0 : 0 : 1)
sage: next(b)  #
→optional - sage.rings.finite_rings
(2*a + 2 : a : 1)
sage: next(b)  #
→optional - sage.rings.finite_rings
(2 : a + 1 : 1)
(continues on next page)
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.

EXAMPLES:

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

```python
sage: R.<x,y,z> = GF(5)[]  #
    optional - sage.rings.finite_rings
sage: f = x^7 + y^7 + z^7  #
    optional - sage.rings.finite_rings
sage: C = Curve(f); pts = C.rational_points()  #
    optional - sage.rings.finite_rings
```

(continues on next page)
Note: Currently this only works over prime field and divisors supported on rational points.
CHAPTER
FIVE

RATIONAL POINTS OF CURVES

We can create points on projective curves:

```
sage: P.<x,y,z,w> = ProjectiveSpace(QQ, 3)
sage: C = Curve([x^3 - 2*x*z^2 - y^3, z^3 - w^3 - x*y*z], P)
sage: Q = C([1,1,0,0])
sage: Q.parent()
Set of rational points of Projective Curve over Rational Field
defined by x^3 - y^3 - 2*x*z^2, -x*y*z + z^3 - w^3
```
or on affine curves:

```
sage: A.<x,y> = AffineSpace(GF(23), 2)
sage: C = Curve([y - y^4 + 17*x^2 - 2*x + 22], A)
sage: Q = C([22,21])
sage: Q.parent()
Set of rational points of Affine Plane Curve over Finite Field of size 23
defined by -y^4 - 6*x^2 - 2*x + y - 1
```

AUTHORS:

• Grayson Jorgenson (2016-6): initial version

class sage.schemes.curves.point.AffineCurvePoint_field(X, v, check=True)
Bases: SchemeMorphism_point_affine_field

```
is_singular()
Return whether this point is a singular point of the affine curve it is on.
EXAMPLES:
```
```
sage: K = QuadraticField(-1)
  #
  ←optional - sage.rings.number_field
sage: A.<x,y,z> = AffineSpace(K, 3)
  #
  ←optional - sage.rings.number_field
sage: C = Curve([(x^4 + 2*z + 2)y, z - y + 1])
  #
  ←optional - sage.rings.number_field
sage: Q1 = C([0,0,-1])
  #
  ←optional - sage.rings.number_field
```
```
sage: Q1.is_singular()  # optional - sage.rings.number_field
True

sage: Q2 = C([-K.gen(),0,-1])  # optional - sage.rings.number_field
sage: Q2.is_singular()  # optional - sage.rings.number_field
False

class sage.schemes.curves.point.AffinePlaneCurvePoint_field(X, v, check=True)
Bases: AffineCurvePoint_field

Point of an affine plane curve over a field.

is_ordinary_singularity()
Return whether this point is an ordinary singularity of the affine plane curve it is on.

EXAMPLES:

sage: A.<x,y> = AffineSpace(QQ, 2)
sage: C = A.curve([x^5 - x^3*y^2 + 5*x^4 - x^3*y - 3*x^2*y^2 +
......: x*y^3 + 10*x^3 - 3*x^2*y - 3*x*y^2 + y^3 + 10*x^2 - 3*x*y - y^2 +
......: 5*x - y + 1])
sage: Q = C([-1,0])
sage: Q.is_ordinary_singularity()
True

sage: A.<x,y> = AffineSpace(GF(7), 2)
# optional - sage.rings.finite_rings
sage: C = A.curve([y^2 - x^7 - 6*x^3])
# optional - sage.rings.finite_rings
sage: Q = C([0,0])
# optional - sage.rings.finite_rings
sage: Q.is_ordinary_singularity()
False

is_transverse(D)
Return whether the intersection of the curve D at this point with the curve this point is on is transverse or not.

INPUT:

• D – a curve in the same ambient space as the curve this point is on.

EXAMPLES:

sage: A.<x,y> = AffineSpace(QQ, 2)
# optional - sage.rings.finite_rings
sage: C = A.curve([y^2 - x^7 - 6*x^3])
# optional - sage.rings.finite_rings
sage: Q = C([0,0])
# optional - sage.rings.finite_rings
sage: Q.is_transverse(D)
False
```python
sage: R.<a> = QQ[]
sage: K.<b> = NumberField(a^2 - 2)
# optional - sage.rings.number_field
sage: A.<x,y> = AffineSpace(K, 2)
# optional - sage.rings.number_field
sage: C = Curve([y^2 + x^2 - 1], A)
# optional - sage.rings.number_field
sage: D = Curve([y - x], A)
# optional - sage.rings.number_field
sage: Q = C([-1/2*b, -1/2*b])
# optional - sage.rings.number_field
sage: Q.is_transverse(D)
# optional - sage.rings.number_field
True
```

**Multiplicity**

Return the multiplicity of this point with respect to the affine curve it is on.

**Examples:**

```python
sage: A.<x,y> = AffineSpace(QQ, 2)
sage: C = A.curve([2*x^7 - 3*x^6*y + x^5*y^2 + 31*x^6 - 40*x^5*y +
....: 13*x^4*y^2 - x^3*y^3 + 207*x^5 - 228*x^4*y + 70*x^3*y^2 - 7*x^2*y^3
....: + 775*x^4 - 713*x^3*y + 193*x^2*y^2 + y^3 + 1764*x^3 -
....: 1293*x^2*y + 277*x*y^2 - 22*y^3 + 2451*x^2 - 1297*x*y + 172*y^2 +
....: 1935*x - 570*y + 675])
sage: Q = C([-2,1])
sage: Q.multiplicity()
4
```

**Tangents**

Return the tangents at this point of the affine plane curve this point is on.

**Output:** a list of polynomials in the coordinate ring of the ambient space of the curve this point is on.

**Examples:**

```python
sage: A.<x,y> = AffineSpace(QQ, 2)
sage: C = A.curve([x^5 - x^3*y^2 + 5*x^4 - x^3*y - 3*x^2*y^2 +
....: x*y^3 + 10*x^3 - 3*x^2*y - 3*x*y^2 + y^3 + 10*x^2 - 3*x*y - y^2 +
....: 5*x - y + 1])
sage: Q = C([-1,0])
sage: Q.tangents()
[y, x + 1, x - y + 1, x + y + 1]
```

**Class** `sage.schemes.curves.point.AffinePlaneCurvePoint_finite_field(X, v, check=True)`

Base classes: `AffinePlaneCurvePoint_field`, `SchemeMorphism_point_affine_finite_field`

Point of an affine plane curve over a finite field.

**Class** `sage.schemes.curves.point.IntegralAffineCurvePoint(X, v, check=True)`

Base classes: `AffineCurvePoint_field`

Point of an integral affine curve.
closed_point()

Return the closed point that corresponds to this rational point.

EXAMPLES:

```
sage: A.<x,y> = AffineSpace(GF(8), 2)
    #_optional - sage.rings.finite_rings
sage: C = Curve(x^5 + y^5 + x*y + 1)
    #_optional - sage.rings.finite_rings
sage: p = C([1,1])
    #_optional - sage.rings.finite_rings
sage: p.closed_point()
    #_optional - sage.rings.finite_rings
Point (x + 1, y + 1)
```

place()

Return a place on this point.

EXAMPLES:

```
sage: A.<x,y> = AffineSpace(GF(2), 2)
    #_optional - sage.rings.finite_rings
sage: C = Curve(x^5 + y^5 + x*y + 1)
    #_optional - sage.rings.finite_rings
sage: p = C(-1,-1)
    #_optional - sage.rings.finite_rings
sage: p
    #_optional - sage.rings.finite_rings
(1, 1)
sage: p.closed_point()
    #_optional - sage.rings.finite_rings
Point (x + 1, y + 1)
sage: _.place()
    #_optional - sage.rings.finite_rings
Place (x + 1, (1/(x^5 + 1))*y^4 + ((x^5 + x^4 + 1)/(x^5 + 1))*y^3
     + ((x^5 + x^3 + 1)/(x^5 + 1))*y^2 + (x^2/(x^5 + 1))*y)
```

places()

Return all places on this point.

EXAMPLES:

```
sage: A.<x,y> = AffineSpace(GF(2), 2)
    #_optional - sage.rings.finite_rings
sage: C = Curve(x^5 + y^5 + x*y + 1)
    #_optional - sage.rings.finite_rings
sage: p = C(-1,-1)
    #_optional - sage.rings.finite_rings
sage: p
    #_optional - sage.rings.finite_rings
(1, 1)
sage: p.closed_point()
    #_optional - sage.rings.finite_rings
Point (x + 1, y + 1)
sage: _.places()
    #_optional - sage.rings.finite_rings
(continues on next page)```
class sage.schemes.curves.point.IntegralAffineCurvePoint_finite_field(X, v, check=True)

Bases: IntegralAffineCurvePoint

Point of an integral affine curve over a finite field.

class sage.schemes.curves.point.IntegralAffinePlaneCurvePoint_finite_field(X, v, check=True)

Bases: IntegralAffineCurvePoint, AffinePlaneCurvePoint_field

Point of an integral affine plane curve over a finite field.

class sage.schemes.curves.point.IntegralProjectiveCurvePoint_field(X, v, check=True)

Bases: ProjectiveCurvePoint_field

closed_point()

Return the closed point corresponding to this rational point.

EXAMPLES:

sage: P.<x,y,z> = ProjectiveSpace(GF(17), 2)  #
            # optional - sage.rings.finite_rings
sage: C = Curve([x^4 - 16*y^3*z], P)  #
            # optional - sage.rings.finite_rings
sage: C.singular_points()  #
            # optional - sage.rings.finite_rings
[[0 : 0 : 1]]
sage: p = _[0]  #
            # optional - sage.rings.finite_rings
sage: p.closed_point()  #
            # optional - sage.rings.finite_rings
Point (x, y)

place()

Return a place on this point.

EXAMPLES:

sage: P.<x,y,z> = ProjectiveSpace(GF(17), 2)  #
            # optional - sage.rings.finite_rings
sage: C = Curve([x^4 - 16*y^3*z], P)  #
            # optional - sage.rings.finite_rings
sage: C.singular_points()  #
            # optional - sage.rings.finite_rings
[[0 : 0 : 1]]
sage: p = _[0]  #
            # optional - sage.rings.finite_rings
sage: p.place()  #
optional - sage.rings.finite_rings
sage: p.place()
optional - sage.rings.finite_rings
Place (y)

places()
Return all places on this point.

EXAMPLES:

```
sage: P.<x,y,z> = ProjectiveSpace(GF(17), 2)
optional - sage.rings.finite_rings
sage: C = Curve([x^4 - 16*y^3*z], P)
optional - sage.rings.finite_rings
sage: C.singular_points()
 optional - sage.rings.finite_rings
[(0 : 0 : 1)]
sage: p = _[0]
optional - sage.rings.finite_rings
sage: p.places()
optional - sage.rings.finite_rings

```

```
class sage.schemes.curves.point.IntegralProjectiveCurvePoint_finite_field(X, v, check=True)
Bases: IntegralProjectiveCurvePoint
Point of an integral projective curve over a finite field.

class sage.schemes.curves.point.IntegralProjectivePlaneCurvePoint_finite_field(X, v, check=True)
Bases: IntegralProjectiveCurvePoint, ProjectivePlaneCurvePoint_field
Point of an integral projective plane curve over a finite field.

class sage.schemes.curves.point.ProjectiveCurvePoint_field(X, v, check=True)
Bases: SchemeMorphism_point_projective_field
Point of a projective curve over a field.

is_singular()
Return whether this point is a singular point of the projective curve it is on.

EXAMPLES:

```
sage: P.<x,y,z,w> = ProjectiveSpace(QQ, 3)
sage: C = Curve([x^2 - y^2, z - w], P)
sage: Q1 = C([0,0,1,1])
sage: Q1.is_singular()
True
sage: Q2 = C([1,1,1,1])
```

sage: Q2.is_singular()
False

class sage.schemes.curves.point.ProjectivePlaneCurvePoint_field(X, v, check=True)

Bases: ProjectiveCurvePoint_field

Point of a projective plane curve over a field.

**is_ordinary_singularity()**

Return whether this point is an ordinary singularity of the projective plane curve it is on.

**EXAMPLES:**

```python
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: C = Curve([z^6 - x^6 - x^3*z^3 - x^3*y^3])
sage: Q = C([0,1,0])
sage: Q.is_ordinary_singularity()
False
```

```python
sage: R.<a> = QQ[]
sage: K.<b> = NumberField(a^2 - 3)
# optional - sage.rings.number_field
sage: P.<x,y,z> = ProjectiveSpace(K, 2)
# optional - sage.rings.number_field
sage: C = P.curve([x^2*y^3*z^4 - y^6*z^3 - 4*x^2*y^4*z^3 - 4*x^4*y^2*z^3 + 3*y^7*z^2 + 10*x^2*y^5*z^2 + 9*x^4*y^3*z^2 + 7*x^6*y^2*z - 2*x^8*y])
# optional - sage.rings.number_field
sage: Q = C([-1/2, 1/2, 1])
# optional - sage.rings.number_field
sage: Q.is_ordinary_singularity()  # optional - sage.rings.number_field
True
```

**is_transverse(D)**

Return whether the intersection of the curve D at this point with the curve this point is on is transverse or not.

**INPUT:**

- D – a curve in the same ambient space as the curve this point is on

**EXAMPLES:**

```python
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: C = Curve([x^2 - 2*y^2 - 2*z^2], P)
sage: D = Curve([y - z], P)
sage: Q = C([-1/2, 1/2, 1])
sage: Q.is_transverse(D)
True
```

```python
sage: P.<x,y,z> = ProjectiveSpace(GF(17), 2)
# optional - sage.rings.finite_rings
sage: Q.is_transverse(D)  # optional - sage.rings.finite_rings
True
```

(continues on next page)
Curves, Release 10.1

sage: C = Curve([x^4 - 16*y^3*z], P)  # optional - sage.rings.finite_rings
sage: D = Curve([y^2 - z*x], P)  # optional - sage.rings.finite_rings
sage: Q = C([0,0,1])  # optional - sage.rings.finite_rings
sage: Q.is_transverse(D)  # optional - sage.rings.finite_rings
False

multiplicity()

Return the multiplicity of this point with respect to the projective curve it is on.

EXAMPLES:

sage: P.<x,y,z> = ProjectiveSpace(GF(17), 2)  # optional - sage.rings.finite_rings
sage: C = Curve([y^3*z - 16*x^4], P)  # optional - sage.rings.finite_rings
sage: Q = C([0,0,1])  # optional - sage.rings.finite_rings
sage: Q.multiplicity()  # optional - sage.rings.finite_rings
3

tangents()

Return the tangents at this point of the projective plane curve this point is on.

OUTPUT:

A list of polynomials in the coordinate ring of the ambient space of the curve this point is on.

EXAMPLES:

sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: C = Curve([y^2*z^3 - x^5 + 18*y*x*z^3])
sage: Q = C([0,0,1])
sage: Q.tangents()
[y, 18*x + y]

class sage.schemes.curves.point.ProjectivePlaneCurvePoint_finite_field(X, v, check=True)

Bases: ProjectivePlaneCurvePoint_field, SchemeMorphism_point_projective_finite_field

Point of a projective plane curve over a finite field.
CLOSED POINTS OF INTEGRAL CURVES

A rational point of a curve in Sage is represented by its coordinates. If the curve is defined over finite field and integral, that is reduced and irreducible, then it is empowered by the global function field machinery of Sage. Thus closed points of the curve are computable, as represented by maximal ideals of the coordinate ring of the ambient space.

EXAMPLES:

```sage
sage: F.<a> = GF(2)  # optional - sage.rings.finite_rings
sage: P.<x,y> = AffineSpace(F, 2)  # optional - sage.rings.finite_rings
sage: C = Curve(y^2 + y - x^3)  # optional - sage.rings.finite_rings
sage: C.closed_points()  # optional - sage.rings.finite_rings
[Point (x, y), Point (x, y + 1)]

sage: C.closed_points(2)  # optional - sage.rings.finite_rings
[Point (y^2 + y + 1, x + 1),
 Point (y^2 + y + 1, x + y),
 Point (y^2 + y + 1, x + y + 1)]

sage: C.closed_points(3)  # optional - sage.rings.finite_rings
[Point (x^2 + x + y, x*y + 1, y^2 + x + 1),
 Point (x^2 + x + y + 1, x*y + x + 1, y^2 + x)]
```

Closed points of projective curves are represented by homogeneous maximal ideals:

```sage
sage: F.<a> = GF(2)  # optional - sage.rings.finite_rings
sage: P.<x,y,z> = ProjectiveSpace(F, 2)  # optional - sage.rings.finite_rings
sage: C = Curve(x^3*y + y^3*z + x*z^3)  # optional - sage.rings.finite_rings
sage: C.closed_points()  # optional - sage.rings.finite_rings
[Point (x, z), Point (x, y), Point (y, z)]

sage: C.closed_points(2)  # optional - sage.rings.finite_rings
[Point (y^2 + y*z + z^2, x + y + z)]

sage: C.closed_points(3)  # optional - sage.rings.finite_rings
```

(continues on next page)
Curves, Release 10.1

[Point (y^3 + y^2z + z^3, x + y),
  Point (y^3 + y^2z + y^2z^2, x^2y + x^2z + z^2, y^2 + x^2z),
  Point (x^2 + y^2z, x^2y + x^2z + z^2, y^2 + x^2z + y^2z),
  Point (x^3 + x^2z^2 + y^2z, y + z),
  Point (x^2 + y^2z + z^2, x^2y + x^2z + y^2z, y^2 + x^2z + y^2z + z^2),
  Point (x^2 + y^2z + z^2, x^2y + z^2, y^2 + x^2z + y^2z)]

Rational points are easily converted to closed points and vice versa if the closed point is of degree one:

```
sage: F.<a> = GF(2)  # optional - sage.rings.finite_rings
sage: P.<x,y,z> = ProjectiveSpace(F, 2)  # optional - sage.rings.finite_rings
sage: C = Curve(x^3*y + y^3*z + x*z^3)  # optional - sage.rings.finite_rings
sage: pl, p2, p3 = C.closed_points()  # optional - sage.rings.finite_rings
sage: pl.rational_point()  # optional - sage.rings.finite_rings
(0 : 1 : 0)
sage: p2.rational_point()  # optional - sage.rings.finite_rings
(0 : 0 : 1)
sage: p3.rational_point()  # optional - sage.rings.finite_rings
(1 : 0 : 0)
sage: _.closed_point()  # optional - sage.rings.finite_rings
Point (y, z)
sage: _ == p3  # optional - sage.rings.finite_rings
True
```

AUTHORS:

- Kwankyu Lee (2019-03): initial version

```python
class sage.schemes.curves.closed_point.CurveClosedPoint(S, P, check=False)
    Bases: SchemeTopologicalPoint_prime_ideal
    Base class of closed points of curves.

class sage.schemes.curves.closed_point.IntegralAffineCurveClosedPoint(curve, prime_ideal, degree)
    Bases: IntegralCurveClosedPoint
    Closed points of affine curves.

    projective(i=0)
        Return the point in the projective closure of the curve, of which this curve is the i-th affine patch.

        INPUT:

        - i – an integer

        EXAMPLES:
```
sage: F.<a> = GF(2)  # optional - sage.rings.finite_rings
sage: A.<x,y> = AffineSpace(F, 2)  # optional - sage.rings.finite_rings
sage: C = Curve(y^2 + y - x^3, A)  # optional - sage.rings.finite_rings
sage: p1, p2 = C.closed_points()  # optional - sage.rings.finite_rings
sage: p1  # optional - sage.rings.finite_rings
Point (x, y)
sage: p2  # optional - sage.rings.finite_rings
Point (x, y + 1)
sage: p1.projective()  # optional - sage.rings.finite_rings
Point (x1, x2)
sage: p2.projective(0)  # optional - sage.rings.finite_rings
Point (x1, x0 + x2)
sage: p2.projective(1)  # optional - sage.rings.finite_rings
Point (x0, x1 + x2)
sage: p2.projective(2)  # optional - sage.rings.finite_rings
Point (x0, x1 + x2)

rational_point()

Return the rational point if this closed point is of degree 1.

EXAMPLES:

sage: A.<x,y> = AffineSpace(GF(3^2), 2)  # optional - sage.rings.finite_rings
sage: C = Curve(y^2 - x^5 - x^4 - 2*x^3 - 2*x - 2)  # optional - sage.rings.finite_rings
sage: C.closed_points()  # optional - sage.rings.finite_rings
[Point (x, y + (z2 + 1)), Point (x, y + (-z2 - 1)), Point (x + (z2 + 1), y + (z2 - 1)), Point (x + (z2 + 1), y + (-z2 + 1)), Point (x - 1, y + (z2 + 1)), Point (x - 1, y + (-z2 - 1)), Point (x + (-z2 - 1), y + z2), Point (x + (-z2 - 1), y + (-z2)), Point (x + 1, y + 1), Point (x + 1, y - 1)]
sage: [p.rational_point() for p in _]  # optional - sage.rings.finite_rings
[(0, 2*z2 + 2), (0, z2 + 1), (2*z2 + 2, 2*z2 + 1), (continues on next page)
\[(2z^2 + 2, z^2 + 2),
(1, 2z^2 + 2),
(1, z^2 + 1),
(z^2 + 1, 2z^2),
(z^2 + 1, z^2),
(2, 2),
(2, 1)]\]

```
sage: set(_) == set(C.rational_points())  # optional - sage.rings.finite_rings
True
```
Curves, Release 10.1

optional - sage.rings.finite_rings

sage: p = pts[0]  #

optional - sage.rings.finite_rings

sage: p.curve()  #

optional - sage.rings.finite_rings

Affine Plane Curve over Finite Field in a of size 2^2 defined by x^3 + y^2 + y

degree()

Return the degree of the point.

EXAMPLES:

sage: F.<a> = GF(4)  #
optional - sage.rings.finite_rings

sage: P.<x,y> = AffineSpace(F, 2)  #
optional - sage.rings.finite_rings

sage: C = Curve(y^2 + y - x^3)  #
optional - sage.rings.finite_rings

sage: pts = C.closed_points()  #
optional - sage.rings.finite_rings

sage: p = pts[0]  #
optional - sage.rings.finite_rings

sage: p.degree()  #
optional - sage.rings.finite_rings
1

place()

Return a place on this closed point.

If there are more than one, arbitrary one is chosen.

EXAMPLES:

sage: F.<a> = GF(4)  #
optional - sage.rings.finite_rings

sage: P.<x,y> = AffineSpace(F, 2)  #
optional - sage.rings.finite_rings

sage: C = Curve(y^2 + y - x^3)  #
optional - sage.rings.finite_rings

sage: pts = C.closed_points()  #
optional - sage.rings.finite_rings

sage: p = pts[0]  #
optional - sage.rings.finite_rings

sage: p.place()  #
optional - sage.rings.finite_rings
Place (x, y)

places()

Return all places on this closed point.

EXAMPLES:

sage: F.<a> = GF(4)  #
optional - sage.rings.finite_rings

(continues on next page)
sage: P.<x,y> = AffineSpace(F, 2)
#optional - sage.rings.finite_rings
sage: C = Curve(y^2 + y - x^3)
#optional - sage.rings.finite_rings
sage: pts = C.closed_points()
#optional - sage.rings.finite_rings
sage: p = pts[0]
#optional - sage.rings.finite_rings
sage: p.places()
#optional - sage.rings.finite_rings
[Place (x, y)]

class sage.schemes.curves.closed_point.IntegralProjectiveCurveClosedPoint(curve, prime_ideal, degree)

Bases: IntegralCurveClosedPoint

Closed points of projective plane curves.

affine(i=None)

Return the point in the i-th affine patch of the curve.

INPUT:

• i – an integer; if not specified, it is chosen automatically.

EXAMPLES:

sage: F.<a> = GF(2)
#optional - sage.rings.finite_rings
sage: P.<x,y,z> = ProjectiveSpace(F, 2)
#optional - sage.rings.finite_rings
sage: C = Curve(x^3*y + y^3*z + x*z^3)
#optional - sage.rings.finite_rings
sage: p1, p2, p3 = C.closed_points()
#optional - sage.rings.finite_rings
sage: p1.affine()
#optional - sage.rings.finite_rings
Point (x, z)

sage: p2.affine()
#optional - sage.rings.finite_rings
Point (x, y)

sage: p3.affine()
#optional - sage.rings.finite_rings
Point (y, z)

sage: p3.affine(0)
#optional - sage.rings.finite_rings
Traceback (most recent call last):
...
ValueError: not in the affine patch

rational_point()

Return the rational point if this closed point is of degree 1.
EXAMPLES:

```python
sage: F.<a> = GF(4)  #
   -> optional - sage.rings.finite_rings
sage: P.<x,y,z> = ProjectiveSpace(F, 2)  #
   -> optional - sage.rings.finite_rings
sage: C = Curve(x^3*y + y^3*z + x*z^3)  #
   -> optional - sage.rings.finite_rings
sage: C.closed_points()  #
   -> optional - sage.rings.finite_rings
[Point (x, z),
  Point (x, y),
  Point (y, z),
  Point (x + a*z, y + (a + 1)*z),
  Point (x + (a + 1)*z, y + a*z)]
sage: [p.rational_point() for p in _]  #
   -> optional - sage.rings.finite_rings
[(0 : 1 : 0), (0 : 0 : 1), (1 : 0 : 0), (a : a + 1 : 1), (a + 1 : a : 1)]
sage: set(_) == set(C.rational_points())  #
   -> optional - sage.rings.finite_rings
True
```
CHAPTER
SEVEN

JACOBIANS OF CURVES

This module defines the base class of Jacobians as an abstract scheme.

AUTHORS:
• William Stein (2005)

sage.schemes.jacobians.abstract_jacobian.Jacobian(C)

EXAMPLES:

```python
sage: from sage.schemes.jacobians.abstract_jacobian import Jacobian
sage: P2.<x, y, z> = ProjectiveSpace(QQ, 2)
sage: C = Curve(x^3 + y^3 + z^3)
sage: Jacobian(C)
Jacobian of Projective Plane Curve over Rational Field defined by x^3 + y^3 + z^3
```

class sage.schemes.jacobians.abstract_jacobian.Jacobian_generic(C)

Bases: Scheme

Base class for Jacobians of projective curves.

The input must be a projective curve over a field.

EXAMPLES:

```python
sage: from sage.schemes.jacobians.abstract_jacobian import Jacobian
sage: P2.<x, y, z> = ProjectiveSpace(QQ, 2)
sage: C = Curve(x^3 + y^3 + z^3)
```

base_extend(R)

Return the natural extension of self over R.

INPUT:
• R – a field. The new base field.

OUTPUT: The Jacobian over the ring R.

EXAMPLES:

```python
sage: R.<x> = QQ['x']
```

(continues on next page)
**Curves, Release 10.1**

(continued from previous page)

```
defined by y^2 = x^3 - 10*x + 9
sage: F.<a> = QQ.extension(x^2 + 1)  # optional - sage.rings.number_field
sage: Jac.base_extend(F)  # optional - sage.rings.number_field
Jacobian of Hyperelliptic Curve over Number Field in a with defining
polynomial x^2 + 1 defined by y^2 = x^3 - 10*x + 9
```

\section{change\_ring}(R)

Return the Jacobian over the ring $R$.

**INPUT:**

- $R$ – a field. The new base ring.

**OUTPUT:** The Jacobian over the ring $R$.

**EXAMPLES:**

```
sage: R.<x> = QQ['x']
sage: H = HyperellipticCurve(x^3 - 10*x + 9)
sage: Jac = H.jacobian(); Jac
Jacobian of Hyperelliptic Curve over Rational Field
defined by y^2 = x^3 - 10*x + 9
sage: Jac.change_ring(RDF)
Jacobian of Hyperelliptic Curve over Real Double Field
defined by y^2 = x^3 - 10.0*x + 9.0
```

\section{curve()}

Return the curve of which \emph{self} is the Jacobian.

**EXAMPLES:**

```
sage: from sage.schemes.jacobians.abstract_jacobian import Jacobian
sage: P2.<x, y, z> = ProjectiveSpace(QQ, 2)
sage: J = Jacobian(Curve(x^3 + y^3 + z^3))
sage: J.curve()
Projective Plane Curve over Rational Field defined by x^3 + y^3 + z^3
```

\section{sage.schemes.jacobians.abstract_jacobian.is\_jacobian}(J)

Return True if $J$ is of type Jacobian\_generic.

**EXAMPLES:**

```
sage: from sage.schemes.jacobians.abstract_jacobian import Jacobian, is_Jacobian
sage: P2.<x, y, z> = ProjectiveSpace(QQ, 2)
sage: C = Curve(x^3 + y^3 + z^3)
sage: J = Jacobian(C)
sage: is_Jacobian(J)
True
sage: E = EllipticCurve('37a1')
sage: is_Jacobian(E)
False
```
8.1 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 $\mathbb{P}^3$. 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
Conics given by matrices

```sage
sage: C = 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: C = 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
```

Conics given by coefficients

```sage
sage: C = Conic(QQ, [1,2,3])
Projective Conic Curve over Rational Field defined by x^2 + 2*x*y + 3*z^2
sage: C = Conic(QQ, [1,2,3])
Projective Conic Curve over Rational Field defined by x^2 + 2*x*y + 3*z^2
```

The conic through a set of points

```sage
sage: C = Conic(QQ, [[10,2],[3,4],[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 + 91/4*y*z - 37/2*z^2
sage: C.rational_point()
(10 : 2 : 1)
sage: C.point([3,4])
(3 : 4 : 1)
sage: a = AffineSpace(GF(13), 2)
```

Chapter 8. Plane conics
8.2 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: ProjectivePlaneCurve_field

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

EXAMPLES:

```python
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 Field defined by X^2 + Y^2 - Z^2
```

```python
base_extend(S)

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

EXAMPLES:

```python
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
# optional - sage.rings.number_field
Projective Conic Curve over Number Field in i with defining polynomial x^2 + 1 with i = 1*I defined by x^2 + y^2 + z^2
sage: d.rational_point(algorithm='rnfisnorm')
# optional - sage.rings.number_field
(i : 1 : 0)
```

```python
cache_point(p)

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

EXAMPLES:

```python
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)
```

```python
coefficients()

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

EXAMPLES:

```python
```
sage: Conic(QQ, [1,2,3,4,5,6]).coefficients()
[1, 2, 3, 4, 5, 6]
sage: P.<x,y,z> = GF(13)[]
# optional - sage.rings.finite_rings
sage: a = Conic(x^2 + 5*x*y + y^2 + z^2).coefficients(); a
[1, 5, 0, 1, 0, 1]
sage: Conic(a)
# optional - sage.rings.finite_rings
Projective Conic Curve over Finite Field of size 13 defined by x^2 + 5*x*y + y^2 + z^2

derivative_matrix()

Give the derivative of the defining polynomial of the conic self, which is a linear map, as a $3 \times 3$ 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)[]
# optional - sage.rings.finite_rings
sage: c = Conic([t, 1, t^2, 1, 1, 0]); c
# optional - sage.rings.finite_rings
Projective Conic Curve over Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 2 (using GF2X) defined by t*x^2 + x*y + y^2 + (t^2)*x*z + y*z
sage: c.is_smooth()
# optional - sage.rings.finite_rings
True
sage: c.derivative_matrix()
# optional - sage.rings.finite_rings
[ 0 1 t^2]
[ 1 0 1]
[t^2 1 0]

determinant()

Return 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])
# → optional - sage.rings.finite_rings
sage: C.is_smooth()
# → optional - sage.rings.finite_rings
True
sage: C.determinant()
# → optional - sage.rings.finite_rings
Traceback (most recent call last):
...
ValueError: The conic self (= Projective Conic Curve over Finite Field of size 2 defined by x*y + y^2 + x*z + y*z) has no symmetric matrix because the base field has characteristic 2

\textbf{diagonal\_matrix()}

Return 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 \texttt{self}.

\textbf{EXAMPLES:}

sage: c = Conic(QQ, [1,2,3,4,5,6])
sage: d, t = c.diagonal_matrix(); d, t
d = 
\begin{bmatrix}
1 & 0 & 0 \\
0 & 3 & 0 \\
0 & 0 & 41/12
\end{bmatrix}
t = 
\begin{bmatrix}
1 & -1 & -7/6 \\
0 & 1 & -1/3 \\
0 & 0 & 1
\end{bmatrix}
sage: t.transpose()*c.symmetric_matrix()*t
\begin{bmatrix}
1 & 0 & 0 \\
0 & 3 & 0 \\
0 & 0 & 41/12
\end{bmatrix}

Diagonal matrices are only defined in characteristic different from 2:

sage: c = Conic(GF(4, 'a'), [0, 1, 1, 1, 1, 1])
# → optional - sage.rings.finite_rings
sage: c.is_smooth()
# → optional - sage.rings.finite_rings
True
sage: c.diagonal_matrix()
# → optional - sage.rings.finite_rings
Traceback (most recent call last):
...
ValueError: The conic self (= Projective Conic Curve over Finite Field in a of size 2^2 defined by x*y + y^2 + x*z + y*z + z^2) has no symmetric matrix because the base field has characteristic 2
diagonalization (names=None)

Return a diagonal conic \( C \), an isomorphism of schemes \( M : C \to \text{self} \) and the inverse \( N \) of \( M \).

EXAMPLES:

```python
sage: Conic(GF(5), [1,0,1,1,0,1]).diagonalization()
# optional - sage.rings.finite_rings
(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:

```python
sage: Conic(GF(2), [1,1,1,1,1,0]).diagonalization()
# optional - sage.rings.finite_rings
Traceback (most recent call last):
  ...
ValueError: The conic self (= Projective Conic Curve over Finite Field of size 2 defined by x^2 + x*y + y^2 + x*z + y*z) has no symmetric matrix because the base field has characteristic 2
```

An example over a global function field:

```python
sage: K = FractionField(PolynomialRing(GF(7), 't'))
# optional - sage.rings.finite_rings
sage: (t,) = K.gens()
# optional - sage.rings.finite_rings
sage: C = Conic(K, [t/2,0, 1, 2, 0, 3])
# optional - sage.rings.finite_rings
sage: C.diagonalization()
# optional - sage.rings.finite_rings
(Projective Conic Curve over Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 7 defined by (-3*t)*x^2 + 2*y^2 + (3*t + 3)/t*z^2,
 Scheme morphism:
  From: Projective Conic Curve over Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 7 defined by (-3*t)*x^2 + 2*y^2 + (3*t + 3)/t*z^2
to:  Projective Conic Curve over Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 7 defined by (-3*t)*x^2 + 2*y^2 + x*z + 3*z^2
Defn: Defined on coordinates by sending (x : y : z) to (x - 1/t*z : y : z),
Scheme morphism:
  From: Projective Conic Curve over Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 7 defined by (-3*t)*x^2 + 2*y^2 + x*z + 3*z^2
to:  Projective Conic Curve over Fraction Field of Univariate Polynomial Ring in t over Finite Field of size 7 defined by (-3*t)*x^2 + 2*y^2 + x*z + 3*z^2
Defn: Defined on coordinates by sending (x : y : z) to (x - 2*z : y : z))
```

(continues on next page)
From: Projective Conic Curve over Fraction Field of Univariate Polynomial Ring in \( t \) over Finite Field of size 7 defined by \((-3*t)*x^2 + 2*y^2 + x*z + 3*z^2\)

To: Projective Conic Curve over Fraction Field of Univariate Polynomial Ring in \( t \) over Finite Field of size 7 defined by \((-3*t)*x^2 + 2*y^2 + (3*t + 3)/t*z^2\)

Defn: Defined on coordinates by sending \((x : y : z)\) to \((x + 1/t*z : y : z)\)

gens()

Return the generators of the coordinate ring of \( \text{self} \).

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 \( \text{gens}() \) is required for the following construction:

```
sage: C.<a,b,c> = Conic(GF(3), [1, 1, 1])

# optional - sage.rings.finite_rings

sage: C
# optional - sage.rings.finite_rings

Projective Conic Curve over Finite Field of size 3 defined by a^2 + b^2 + c^2
```

has_rational_point(points=False, algorithm='default', read_cache=True)

Return True if and only if the conic \( \text{self} \) 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 be solved internally:

8.2. Projective plane conics over a field

101
sage: R.<t> = QQ[]
sage: C = Conic([-2,t^2+1,t^2-1])
sage: C.has_rational_point()
True

And they can also 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

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)[] # optional - sage.rings.finite_rings
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^2
sage: e.has_singular_point(point = True) # optional - sage.rings.finite_rings
(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) # optional - sage.rings.finite_rings
sage: Conic([a, a + 1, 1]).has_singular_point(point=True) # optional - sage.rings.finite_rings
(True, (a + 1 : 0 : 1))

sage: P.<t> = GF(2)[] # optional - sage.rings.finite_rings
(continues on next page)
\texttt{sage}: \texttt{C} = \texttt{Conic(P, [t,t,1]); C}  
\hspace{1cm} \# \texttt{optional - sage.rings.finite_rings}

Projective Conic Curve over Fraction Field of Univariate Polynomial Ring
\hspace{1cm} in t over Finite Field of size 2 \hspace{1cm} (using GF2X) defined by \(t^*x^2 + t^*y^2 + z^2\)

\texttt{sage}: \texttt{C.has_singular_point(point = False)}  
\hspace{1cm} \# \texttt{optional - sage.rings.finite_rings}

Traceback (most recent call last):
...  
NotImplementedError: Sorry, find singular point on conics not implemented
\hspace{1cm} over all fields of characteristic 2.

\textbf{hom}(x, Y=None)

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

\textbf{EXAMPLES}:

Here are a few morphisms given by matrices. In the first example, \texttt{Y} is omitted, in the second example, \texttt{Y} is specified.

\texttt{sage}: \texttt{c = Conic([-1, 1, 1])}
\texttt{sage}: \texttt{h = c.hom(Matrix([[1,1,0],[0,1,0],[0,0,1]])); h}

Scheme morphism:
\hspace{1cm} From: Projective Conic Curve over Rational Field defined by \(-x^2 + y^2 + z^2\)
\hspace{1cm} To: Projective Conic Curve over Rational Field defined by \(-x^2 + 2*x*y + z^2\)
\hspace{1cm} Defn: Defined on coordinates by sending (x : y : z) to (x + y : y : z)

\texttt{sage}: \texttt{h([-1, 1, 0])}
\hspace{1cm} (0 : 1 : 0)

\texttt{sage}: \texttt{c = Conic([-1, 1, 1])}
\texttt{sage}: \texttt{d = Conic([4, 1, -1])}
\texttt{sage}: \texttt{c.hom(Matrix([[0, 0, 1/2], [0, 1, 0], [1, 0, 0]]), d)}

Scheme morphism:
\hspace{1cm} From: Projective Conic Curve over Rational Field defined by \(-x^2 + y^2 + z^2\)
\hspace{1cm} To: Projective Conic Curve over Rational Field defined by \(4*x^2 + y^2 - z^2\)
\hspace{1cm} Defn: Defined on coordinates by sending (x : y : z) to (1/2*z : y : x)

\textbf{ValueError} is raised if the wrong codomain \texttt{Y} is specified:

\texttt{sage}: \texttt{c = Conic([-1, 1, 1])}
\texttt{sage}: \texttt{c.hom(Matrix([[0, 0, 1/2], [0, 1, 0], [1, 0, 0]]), c)}

Traceback (most recent call last):
...  
\textbf{ValueError}: The matrix \texttt{x} (= \begin{bmatrix}
0 & 0 & 1/2 \\
0 & 1 & 0 \\
1 & 0 & 0
\end{bmatrix}) does not define a map
\hspace{1cm} from \texttt{self} (= Projective Conic Curve over Rational Field defined by \(-x^2 + y^2 + \_ \_ \_ \_ _ \_ _ \_ \_ z^2\))
\hspace{1cm} to \texttt{Y} (= Projective Conic Curve over Rational Field defined by \(-x^2 + y^2 + z^2\))

The identity map between two representations of the same conic:

8.2. Projective plane conics over a field  \hspace{1cm} 103
Curves, Release 10.1

```python
sage: C = Conic([1,2,3,4,5,6])
sage: D = Conic([2,4,6,10,12])
sage: C.hom(identity_matrix(3), D)
Scheme morphism:
  From: Projective Conic Curve over Rational Field
defined by x^2 + 2*x*y + 4*y^2 + 3*x*z + 5*y*z + 6*z^2
  To: Projective Conic Curve over Rational Field
defined by 2*x^2 + 4*x*y + 8*y^2 + 6*x*z + 10*y*z + 12*z^2
  Defn: Defined on coordinates by sending (x : y : z) to (x : y : z)
```

An example not over the rational numbers:

```python
sage: P.<t> = QQ[]
sage: C = Conic([1,0,0,t,0,1/t])
sage: D = Conic([1/t^2, 0, -2/t^2, t, 0, (t + 1)/t^2])
sage: T = Matrix([[t,0,1], [0,1,0], [0,0,1]])
sage: C.hom(T, D)
Scheme morphism:
  From: Projective Conic Curve over Fraction Field of Univariate
  Polynomial Ring in t over Rational Field defined by x^2 + t*y^2 + 1/t*z^2
  To: Projective Conic Curve over Fraction Field of Univariate
  Polynomial Ring in t over Rational Field defined by 1/(t^2)*x^2 + t*y^2 - 2/(t^2)*x*z + (t + 1)/(t^2)*z^2
  Defn: Defined on coordinates by sending (x : y : z) to (t*x + z : y : z)
```

**is_diagonal()**

Return True if and only if the conic has the form \(ax^2 + by^2 + cz^2\).

**EXAMPLES:**

```python
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()**

Return True if and only if \(self\) is smooth.

**EXAMPLES:**

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

**matrix()**

Return 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)[]
    # optional - sage.rings.finite_rings
sage: C = Conic(x^2 + x*y + y^2 + x*z + z^2)
    # optional - sage.rings.finite_rings
sage: C.matrix()
    # optional - sage.rings.finite_rings
[1 1 1]
[0 1 0]
[0 0 1]
```

**parametrization(\texttt{point}=\texttt{None}, \texttt{morphism}=\texttt{True})**

Return a parametrization \(f\) of \(\text{self}\) together with the inverse of \(f\).

If \texttt{point} is specified, then that point is used for the parametrization. Otherwise, use \texttt{rational_point()} to find a point.

If \texttt{morphism} is \texttt{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])
    # optional - sage.rings.finite_rings
sage: f, g = c.parametrization(); f, g
    # optional - sage.rings.finite_rings
(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 ...,
(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 ...)
sage: set(f(p) for p in f.domain())
    # optional - sage.rings.finite_rings
\{(0 : 0 : 1), (0 : 1 : 1), (1 : 0 : 1)\}
```

Verification of the example

```
sage: h = g*f; h
    # optional - sage.rings.finite_rings
Scheme endomorphism of Projective Space of dimension 1
over Finite Field of size 2
  Defn: Defined on coordinates by sending (x : y) to ...
```
Curves, Release 10.1

(continued from previous page)

```
sage: h[0]/h[1]  # optional - sage.rings.finite_rings
x/y
sage: h.is_one()  # known bug (see :trac:`31892`)  # optional - sage.rings.finite_rings
True
sage: (x,y,z) = c.gens()  # optional - sage.rings.finite_rings
sage: x.parent()  # optional - sage.rings.finite_rings
Quotient of Multivariate Polynomial Ring in x, y, z
   over Finite Field of size 2 by the ideal (x^2 + x*y + y^2 + x*z + y*z)
sage: k = f*g  # optional - sage.rings.finite_rings
sage: k[0]*z-k[2]*x  # optional - sage.rings.finite_rings
0
sage: k[1]*z-k[2]*y  # optional - sage.rings.finite_rings
0
```

The morphisms are mathematically defined in all points, but don’t work completely in SageMath (see github issue #31892)

```
sage: f, g = c.parametrization([0,0,1])  # optional - sage.rings.finite_rings
sage: g([0,1,1])  # optional - sage.rings.finite_rings
(1 : 0)
sage: f([1,0])  # optional - sage.rings.finite_rings
(0 : 1 : 1)
sage: f([1,1])  # optional - sage.rings.finite_rings
(0 : 0 : 1)
sage: g([0,0,1])  # optional - sage.rings.finite_rings
(1 : 1)
```

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, x^2 + 7*y^2 + 2*z^2], [-1/2*x, 1/7*y + 1/14*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 has no rational points over Rational Field!

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) is not smooth, hence does not have a parametrization.

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 rational_point() and 
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 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])
# optional - sage.rings.finite_rings
sage: [c.random_rational_point() for i in range(10)]
[(1 : 0 : 1), (1 : 0 : 1), (1 : 0 : 1), (0 : 1 : 1), (1 : 0 : 1),
 (0 : 1 : 1), (1 : 0 : 1), (0 : 1 : 1), (1 : 0 : 1), (0 : 1 : 1),
 (continues on next page)
Curves, Release 10.1

(continued from previous page)

\[(0 : 0 : 1), (1 : 0 : 1), (1 : 0 : 1), (0 : 0 : 1), (1 : 0 : 1)\]

\[
sage: \ d = \text{Conic}(\mathbb{Q}, [1, 1, -1])
\]
\[
sage: \ d.\text{random\_rational\_point}(\text{den\_bound}=1, \text{num\_bound}=5) \quad \# \text{ random}
(-24/25 : 7/25 : 1)
\]
\[
sage: \text{Conic}(\mathbb{Q}, [1, 1, 1]).\text{random\_rational\_point()}
\]
Traceback (most recent call last):
...
ValueError: Conic Projective Conic Curve over Rational Field defined by
\[x^2 + y^2 + z^2\] has no rational points over Rational Field!

\textbf{rational\_point(algorithm='default', read\_cache=True)}

Return a point on \textit{self} defined over the base field.

Raises \texttt{ValueError} if no rational point exists.

See \textit{self}.\texttt{has\_rational\_point} for the algorithm used and for the use of the parameters \texttt{algorithm} and \texttt{read\_cache}.

\textbf{EXAMPLES:}

Examples over \(\mathbb{Q}\)

\[
sage: \ R.\langle x, y, z \rangle = \mathbb{Q}[]
\]
\[
sage: \ C = \text{Conic}(7*x^2 + 2*y*z + z^2)
\]
\[
sage: \ C.\text{rational\_point()}
(0 : 1 : 0)
\]
\[
sage: \ C = \text{Conic}(x^2 + 2*y^2 + z^2)
\]
\[
sage: \ C.\text{rational\_point()}
\]
Traceback (most recent call last):
...
ValueError: Conic Projective Conic Curve over Rational Field defined by
\[x^2 + 2*y^2 + z^2\] has no rational points over Rational Field!

\[
sage: \ C = \text{Conic}(x^2 + y^2 + 7*z^2)
\]
\[
sage: \ C.\text{rational\_point(algorithm = 'rnfisnorm')}
\]
Traceback (most recent call last):
...
ValueError: Conic Projective Conic Curve over Rational Field defined by
\[x^2 + y^2 + 7*z^2\] has no rational points over Rational Field!

Examples over number fields

\[
sage: \ P.\langle x \rangle = \mathbb{Q}[]
\]
\[
sage: \ L.\langle b \rangle = \text{NumberField}(x^3 - 5)
\]

\[
sage: \ C = \text{Conic}(L, [3, 2, -b])
\]

\[
sage: \ C.\text{rational\_point(algorithm = 'rnfisnorm')}
\]

\[
sage: \ p = C.\text{rational\_point(algorithm = 'rnfisnorm')}
\]

\[
sage: \ p \quad \# \text{ output is random}
\]

(continues on next page)
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'), # optional - magma #
       ...: read_cache=False)
sage: q  # output is random, # optional - magma #
       ...:  # output is random, # optional - magma #
(1/5*b^2 : 1/5*b^2 : 1)
sage: C.defining_polynomial()(list(q))  # optional - magma #
0
sage: len(str(p)) > 1.5*len(str(q))  # optional - magma #
True
sage: D.rational_point(algorithm='magma'), # random, optional - magma #
       ...: read_cache=False)
sage: E.rational_point(algorithm='magma'), # random, optional - magma #
       ...: read_cache=False)
sage: F = Conic([L.gen(), 30, -20])  # optional - magma #
       ...:  # optional - magma #
(continues on next page)
```
optional - sage.rings.number_field

sage: q
# random, optional - magma

→

optional - sage.rings.number_field

\((-10/7*s + 40/7 : 5/7*s - 6/7 : 1)\)

sage: p = F.rational_point(read_cache=False)
# random

→

optional - sage.rings.number_field

\((788210*s - 1114700 : -171135*s + 242022 : 1)\)

sage: len(str(p)) > len(str(q))
# optional - magma

→

True

sage: G = Conic([L.gen(), 30, -21])
#

→

optional - sage.rings.number_field

sage: G.has_rational_point(algorithm='magma')
# optional - magma

→

False

sage: G.has_rational_point(read_cache=False)
# optional - magma

→

False

sage: G.has_rational_point(algorithm='local', read_cache=False)
# optional - magma

→

False

sage: G.rational_point(algorithm='magma', read_cache=False)
# optional - magma

→

Traceback (most recent call last):
...

ValueError: Conic Projective Conic Curve over Number Field in s with defining polynomial \(x^2 - 2\) with \(s = 1.414213562373095\)
defined by \(s*x^2 + 30*y^2 - 21*z^2\) has no rational points over Number Field in s with defining polynomial \(x^2 - 2\) with \(s = 1.414213562373095\)

sage: G.rational_point(algorithm='magma', read_cache=False)

→

Traceback (most recent call last):
...

ValueError: Conic Projective Conic Curve over Number Field in s with defining polynomial \(x^2 - 2\) with \(s = 1.414213562373095\)
defined by \(s*x^2 + 30*y^2 - 21*z^2\) has no rational points over Number Field in s with defining polynomial \(x^2 - 2\) with \(s = 1.414213562373095\)

Examples over finite fields

sage: F.<a> = FiniteField(7^20)
#

→

optional - sage.rings.finite_rings

sage: C = Conic([1, a, -5]); C
#

→

optional - sage.rings.finite_rings

Projective Conic Curve over Finite Field in a of size 7^20
defined by \(x^2 + a*y^2 - 5*z^2\)

sage: C.rational_point() # output is random
#

→

optional - sage.rings.finite_rings

(continues on next page)
(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 + 3*a^5 + 3*a^4 + a^3 + a + 6
 : 5*a^18 + a^17 + a^16 + 6*a^15 + 4*a^14 + a^13 + 5*a^12 + 5*a^10
 + 2*a^9 + 6*a^8 + 6*a^7 + 6*a^6 + 2*a^4 + 3
 : 1)

Examples over \( \mathbb{R} \) and \( \mathbb{C} \)

```python
sage: Conic(CC, [1, 2, 3]).rational_point()
#optional - sage.rings.finite_rings
(0 : 1.22474487139159*I : 1)
```

```python
sage: Conic(RR, [1, 1, 1]).rational_point()
#optional - sage.rings.finite_rings
Traceback (most recent call last):
... ValueError: Conic Projective Conic Curve over Real Field with 53 bits of precision defined by x^2 + y^2 + z^2 has no rational points over Real Field with 53 bits of precision!
```

**singular_point()**

Return a singular rational point of self.

**EXAMPLES:**

```python
sage: Conic(GF(2), [1,1,1,1,1,1]).singular_point()          #optional - sage.rings.finite_rings
(1 : 1 : 1)
```

ValueError is raised if the conic has no rational singular point

```python
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*y + y^2 + x*z + y*z + z^2) has no rational singular point
```

**symmetric_matrix()**

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

**EXAMPLES:**

```python
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*y^2 - 3*z^2
```
upper_trivialangular_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_trivialangular_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_trivialangular_matrix() * v
x^2 + 2*x*y + y^2 + 3*x*z + z^2
```

variable_names()

Return 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
```

8.3 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: 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 + Y^2 + (-a)*Z^2
```
**has_rational_point** *(point=False, obstruction=False, algorithm='default', read_cache=True)*

Return 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 $\text{(out, S)}$, 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 $\mathbb{Q}$

```python
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:

```python
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_point
```

```
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))
```

(continues on next page)
### is_locally_solvable(p)

Return `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:**

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

```python
sage: [p1, p2] = K.places()
```

```python
sage: C.is_locally_solvable(p1)
```

```python
False
```

```python
sage: C.is_locally_solvable(p2)
```

```python
True
```

```python
sage: O = K.maximal_order()
```

```python
sage: f = (2*O).factor()
```

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

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

Return 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 \( \mathbb{Q} \), 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:**

```python
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
    with a = 2.236067977499790?
  To:   Algebraic Real Field
    Defn: a |--> -2.236067977499790?,
Ring morphism:
  From: Number Field in a with defining polynomial x^2 - 5
    with a = 2.236067977499790?
  To:   Algebraic Real Field
    Defn: a |--> 2.236067977499790?]
```

## 8.4 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: `ProjectiveConic_number_field`

Create a projective plane conic curve over \( \mathbb{Q} \).

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
```

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

Return True if and only if self has a point defined over 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 \(R\).

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 PARI/GP function pari:qfsolve
- 'rnfisnorm' – Use PARI's function pari:rnfisnorm (cannot be combined with obstruction = True)
- 'local' – Check if a local solution exists for all primes and infinite places of 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', # optional - w)
(continues on next page)```
is_locally_solvable($p$)

Return 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 $R$ to denote the infinite place.

EXAMPLES:

```python
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($finite=True$, $infinite=True$, $read_cache=True$)

Return 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:

```python
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 the PARI/GP function pari: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
  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
  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, x^2 + 7*y^2, -2*x^2], [-1/2*x, 1/7*y + 1/14*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 has no rational points over Rational Field!
```

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) is not smooth, hence does not have a parametrization.
```
8.5 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: ProjectiveConic_field, ProjectivePlaneCurve_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

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
```

count_points(n)

If the base field $B$ of $self$ is finite of order $q$, then returns the number of points over $F_q, ..., F_{q^n}$.

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()    # output is random
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 + y^2 + x^3 + y^3
sage: C.has_rational_point(point = True)
(Tru, (0 : 0 : 1))
```

(continues on next page)
8.6 Projective plane conics over a rational function field

The class `ProjectiveConic_rational_function_field` represents a projective plane conic over a rational function field $F(t)$, where $F$ is any field. Instances can be created using `Conic()`.

AUTHORS:

- Lennart Ackermans (2016-02-07): initial version

EXAMPLES:

Create a conic:

```python
sage: K = FractionField(PolynomialRing(QQ, 't'))
sage: P. = K[]
sage: C = Conic(X^2 + Y^2 - Z^2)
```

Points can be found using `has_rational_point()`:

```python
sage: K.<t> = FractionField(QQ['t'])
sage: C = Conic([1, -t, t])
sage: C.has_rational_point(point=True)
```

```python
t = 1
```

```python
sage: C.has_rational_point(point=True)
```

```python
(True, (0 : 1 : 1))
```
Create a projective plane conic curve over a rational function field $F(t)$, where $F$ is any field.

The algorithms used in this class come mostly from [HC2006].

**EXAMPLES:**

```python
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 Field defined by $X^2 + Y^2 - Z^2$

**REFERENCES:**

- [HC2006]
- [Ack2016]

**find_point**(supports, roots, case, solution=0)

Given a solubility certificate like in [HC2006], find a point on self. Assumes self is in reduced form (see [HC2006] for a definition).

If you don’t have a solubility certificate and just want to find a point, use the function has_rational_point() instead.

**INPUT:**

- self – conic in reduced form.
- supports – 3-tuple where supports[i] is a list of all monic irreducible $p \in F[t]$ that divide the i’th of the 3 coefficients.
- roots – 3-tuple containing lists of roots of all elements of supports[i], in the same order.
- case – 1 or 0, as in [HC2006].
- solution – (default: 0) a solution of (5) in [HC2006], if case = 0, 0 otherwise.

**OUTPUT:**

A point $(x, y, z) \in F(t)$ of self. Output is undefined when the input solubility certificate is incorrect.

**ALGORITHM:**

The algorithm used is the algorithm FindPoint in [HC2006], with a simplification from [Ack2016].

**EXAMPLES:**

```python
sage: K.<t> = FractionField(QQ['t'])
sage: C = Conic(K, [t^2 - 2, 2*t^3, -2*t^3 - 13*t^2 - 2*t + 18])
sage: C.has_rational_point(point=True) # indirect test
(True, (-3 : (t + 1)/t : 1))
```

Different solubility certificates give different points:

```python
sage: K.<t> = PolynomialRing(QQ, 't')
sage: C = Conic(K, [t^2 - 2, 2*t, -2*t^3 - 13*t^2 - 2*t + 18])
sage: supp = [[t^2 - 2], [t], [t^3 + 13/2*t^2 + t - 9]]
sage: tbar1 = QQ.extension(supp[0][0], 'tbar').gens()[0]
sage: tbar2 = QQ.extension(supp[1][0], 'tbar').gens()[0]
sage: tbar3 = QQ.extension(supp[2][0], 'tbar').gens()[0]
```

(continues on next page)
sage: roots = [tbar1 + 1, 1/3*tbar2^0, 2/3*tbar3^2 + 11/3*tbar3 - 3]
sage: C.find_point(supp, roots, 1)
(3 : t + 1 : 1)
sage: roots = [-tbar1 - 1, -1/3*tbar2^0, -2/3*tbar3^2 - 11/3*tbar3 + 3]
sage: C.find_point(supp, roots, 1)
(3 : -t - 1 : 1)

\texttt{has\_rational\_point(point=False, algorithm='default', read_cache=True)}

Returns True if and only if the conic \texttt{self} has a point over its base field $F(t)$, which is a field of rational functions.

If \texttt{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 \texttt{read\_cache} is True.

The default algorithm does not (yet) work for all base fields $F$. In particular, \texttt{sage} is required to have:

- an algorithm for finding the square root of elements in finite extensions of $F$;
- a factorization and gcd algorithm for $F[t]$;
- an algorithm for solving conics over $F$.

\textbf{ALGORITHM:}

The parameter \texttt{algorithm} specifies the algorithm to be used:

- 'default' – use a native Sage implementation, based on the algorithm Conic in [HC2006].
- 'magma' (requires Magma to be installed) – delegates the task to the Magma computer algebra system.

\textbf{EXAMPLES:}

We can find points for function fields over (extensions of) $\mathbb{Q}$ and finite fields:

\begin{verbatim}
sage: K.<t> = FractionField(PolynomialRing(QQ, 't'))
sage: C = Conic(K, [t^2 - 2, 2*t^3, -2*t^3 - 13*t^2 - 2*t + 18])
sage: C.has_rational_point(point=True)
(True, (-3 : (t + 1)/t : 1))
sage: R.<t> = FiniteField(23)
# optional - sage.rings.finite_rings
sage: C = Conic([2, t^2 + 1, t^2 + 5])
# optional - sage.rings.finite_rings
sage: C.has_rational_point()
# optional - sage.rings.finite_rings
sage: C.has_rational_point(point=True)
(True, (5*t : 8 : 1))
sage: F.<i> = QuadraticField(-1)
# optional - sage.rings.number_field
sage: R.<t> = F[]
# optional - sage.rings.number_field
sage: C = Conic([1, i*t, -t^2 + 4])
# optional - sage.rings.number_field
sage: C.has_rational_point(point=True)
# optional - sage.rings.number_field
sage: C.has_rational_point(point=True)
(True, (-t - 2*i : -2*i : 1))
\end{verbatim}
It works on non-diagonal conics as well:

```python
sage: K.<t> = QQ[]
sage: C = Conic([4, -4, 8, 1, -4, t + 4])
sage: C.has_rational_point(point=True)
(True, (1/2 : 1 : 0))
```

If no point exists output still depends on the argument `point`:

```python
sage: K.<t> = QQ[]
sage: C = Conic(K, [t^2, (t-1), -2*(t-1)])
sage: C.has_rational_point()
False
sage: C.has_rational_point(point=True)
(False, None)
```

Due to limitations in Sage of algorithms we depend on, it is not yet possible to find points on conics over multivariate function fields (see the requirements above):

```python
sage: F.<t1> = FractionField(QQ['t1'])
sage: K.<t2> = FractionField(F['t2'])
sage: a = K(1)
sage: b = 2*t2^2 + 2*t1*t2 - t1^2
sage: c = -3*t2^4 - 4*t1*t2^3 + 8*t1^2*t2^2 + 16*t1^3 - t2 - 48*t1^4
sage: C = Conic([a,b,c])
sage: C.has_rational_point()  
Traceback (most recent call last):
...  
NotImplementedError: is_square() not implemented for elements of Univariate Quotient Polynomial Ring in tbar over Fraction Field of Univariate Polynomial Ring in t1 over Rational Field with modulus tbar^2 + t1*tbar - 1/2*t1^2
```

In some cases, the algorithm requires us to be able to solve conics over $F$. In particular, the following does not work:

```python
sage: P.<u> = QQ[]
sage: E = P.fraction_field()
sage: Q.<Y> = E[]
sage: F.<v> = E.extension(Y^2 - u^3 - 1)
#optional - sage.rings.function_field
sage: R.<t> = F[]
#optional - sage.rings.function_field
sage: K = R.fraction_field()
#optional - sage.rings.function_field
sage: C = Conic(K, [u, v, 1])
#optional - sage.rings.function_field
sage: C.has_rational_point()  
Traceback (most recent call last):
...  
NotImplementedError: has_rational_point not implemented for conics over base field Univariate Quotient Polynomial Ring in v over Fraction Field of Univariate Polynomial Ring in u over Rational
```

(continues on next page)
Field with modulus $v^2 - u^3 - 1$

has_rational_point fails for some conics over function fields over finite fields, due to github issue #20003:

```
sage: K.<t> = PolynomialRing(GF(7))
"optional - sage.rings.finite_rings"
sage: C = Conic([5*t^2 + 4, t^2 + 3*t + 3, 6*t^2 + 3*t + 2],
"optional - sage.rings.finite_rings"
....:
      5*t^2 + 5, 4*t + 3, 4*t^2 + t + 5]
sage: C.has_rational_point()
"optional - sage.rings.finite_rings"
Traceback (most recent call last):
...
TypeError: self (=Scheme morphism:
  From: Projective Conic Curve over Fraction Field of Univariate
  Polynomial Ring in t over Finite Field of size 7 defined by
  (-2*t^2 - 3)*x^2 + (-t^3 + 3*t^2 - 2*t - 2)/(t + 3)*y^2 + (-t^6 + 3*t^5
  \mapsto t^3 - t^2 - t + 2)/(t^4 + t^3 - 3*t^2 + 3*t + 1)*z^2
To:  Projective Conic Curve over Fraction Field of Univariate
  Polynomial Ring in t over Finite Field of size 7 defined by
  (-2*t^2 - 3)*x^2 + (t^2 + 3*t + 3)*x*y + (-2*t^2 - 2)*y^2 + (-t^2 + 3*t
  \mapsto 2)*x^2 + (-3*t + 3)*y^2 + (-3*t^2 + t - 2)*z^2
Defn: Defined on coordinates by sending (x : y : z) to
x + (2*t - 2)/(t + 3)*y + (3*t^4 + 2*t^3 - 2*t^2 - 2*t + 3)/(t^4 + t^3
\mapsto - 3*t^2 + 3*t + 1)*z
  : y + (-t^3 - t^2 + 3*t - 1)/(t^3 - 3*t^2 + 2*t + 2)*z : z))
domain must equal right (=Scheme morphism:
  From: Projective Conic Curve over Fraction Field of Univariate
  Polynomial Ring in t over Finite Field of size 7 defined by
  (-2*t^3 - t^2 + 3*t + 3)*x^2 + (t - 3)*y^2 + (-t^7 + 2*t^5 + t^4 + 2*t^3
  \mapsto t^3 - 3*t^2 + 2*t + 2)*z^2
To:  Projective Conic Curve over Fraction Field of Univariate
  Polynomial Ring in t over Finite Field of size 7 defined by
  -2/(t^3 - 3*t^2 + 2*t + 2)*x^2 + 1/(t^9 - 2*t^8 + t^7 - t^6 + 3*t^5 - 3*t^3 + t^2
\mapsto 2*t + 3)*z^2
Defn: Defined on coordinates by sending (x : y : z) to
((t^3 - 3*t^2 + 2*t + 2)*x : (t^2 - 2)*y : (t^5 - 3*t^4 + t^2 + 3*t +
\mapsto 3)*z))
codomain
```
CHAPTER
NINE

PLANE QUARTICS

9.1 Quartic curve constructor

sage.schemes.plane_quartics.quartic_constructor.QuarticCurve(F, PP=None, check=False)

Return the quartic curve defined by the polynomial F.

INPUT:
• F – a polynomial in three variables, homogeneous of degree 4
• PP – a projective plane (default: None)
• check – whether to check for smoothness or not (default: False)

EXAMPLES:

```
sage: x,y,z = PolynomialRing(QQ, ['x', 'y', 'z']).gens()
sage: QuarticCurve(x**4 + y**4 + z**4)
Quartic Curve over Rational Field defined by x^4 + y^4 + z^4
```

9.2 Plane quartic curves over a general ring

These are generic genus 3 curves, as distinct from hyperelliptic curves of genus 3.

EXAMPLES:

```
sage: PP.<X,Y,Z> = ProjectiveSpace(2, QQ)
sage: f = X^4 + Y^4 + Z^4 - 3*X*Y*Z*(X+Y+Z)
sage: C = QuarticCurve(f); C
Quartic Curve over Rational Field defined by X^4 + Y^4 - 3*X*Y*Z*(X+Y+Z)
```

class sage.schemes.plane_quartics.quartic_generic.QuarticCurve_generic(A,f)

Bases: ProjectivePlaneCurve

genus()

    Return the genus of self.

EXAMPLES:
sage: x,y,z = PolynomialRing(QQ, ['x','y','z']).gens()
sage: Q = QuarticCurve(x**4 + y**4 + z**4)
sage: Q.genus()
3

sage.schemes.plane_quartics.quartic_generic.is_QuarticCurve(C)
Check whether C is a Quartic Curve.

EXAMPLES:

sage: from sage.schemes.plane_quartics.quartic_generic import is_QuarticCurve
sage: x,y,z = PolynomialRing(QQ, ['x','y','z']).gens()
sage: Q = QuarticCurve(x**4 + y**4 + z**4)
sage: is_QuarticCurve(Q)
True
10.1 Riemann matrices and endomorphism rings of algebraic Riemann surfaces

This module provides a class, `RiemannSurface`, to model the Riemann surface determined by a plane algebraic curve over a subfield of the complex numbers.

A homology basis is derived from the edges of a Voronoi cell decomposition based on the branch locus. The pull-back of these edges to the Riemann surface provides a graph on it that contains a homology basis.

The class provides methods for computing the Riemann period matrix of the surface numerically, using a certified homotopy continuation method due to [Kr2016].

The class also provides facilities for computing the endomorphism ring of the period lattice numerically, by determining integer (near) solutions to the relevant approximate linear equations.

One can also calculate the Abel-Jacobi map on the Riemann surface, and there is basic functionality to interface with divisors of curves to facilitate this.

AUTHORS:

- Alexandre Zotine, Nils Bruin (2017-06-10): initial version
- Nils Bruin, Jeroen Sijsling (2018-01-05): algebraization, isomorphisms
- Linden Disney-Hogg, Nils Bruin (2021-06-23): efficient integration
- Linden Disney-Hogg, Nils Bruin (2022-09-07): Abel-Jacobi map

EXAMPLES:

We compute the Riemann matrix of a genus 3 curve:

```sage
def from_sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
def R.<x,y> = QQ[]
def f = x^4-x^3*y+2*x^3+2*x^2*y+2*x^2-2*x*y^2+4*x*y-y^3+3*y^2+2*y+1
def S = RiemannSurface(f, prec=100)
def M = S.riemann_matrix()
```

We test the usual properties, i.e., that the period matrix is symmetric and that the imaginary part is positive definite:

```sage
def all(abs(a) < 1e-20 for a in (M.M.T).list())
def iM = Matrix(RDF,3,3,[a.imag_part() for a in M.list()])
def iM.is_positive_definite()
```

True

True
We compute the endomorphism ring and check it has \( \mathbb{Z} \)-rank 6:

```
sage: A = S.endomorphism_basis(80,8)
sage: len(A) == 6
True
```

In fact it is an order in a number field:

```
sage: T.<t> = QQ[]
sage: K.<a> = NumberField(t^6 - t^5 + 2*t^4 + 8*t^3 - t^2 - 5*t + 7)
sage: all(len(a.minpoly().roots(K)) == a.minpoly().degree() for a in A)
True
```

We can look at an extended example of the Abel-Jacobi functionality. We will demonstrate a particular half-canonical divisor on Klein’s Curve, known in the literature:

```
sage: f = x^3*y + y^3 + x
sage: S = RiemannSurface(f, integration_method='rigorous')
sage: BL = S.places_at_branch_locus(); BL
[Place (x, y, y^2),
 Place (x^7 + 27/4, y + 4/9*x^5, y^2 + 4/3*x^3),
 Place (x^7 + 27/4, y - 2/9*x^5, y^2 + 1/3*x^3)]
```

We can read off the output of places_at_branch_locus to choose our divisor, and we can calculate the canonical divisor using curve functionality:

```
sage: P0 = 1*BL[0]
sage: from sage.schemes.curves.constructor import Curve
sage: C = Curve(f)
sage: F = C.function_field()
sage: K = (F(x).differential()).divisor() - F(f.derivative(y)).divisor()
sage: Pinf, Pinf_prime = C.places_at_infinity()
sage: if K-3*Pinf-1*Pinf_prime: Pinf, Pinf_prime = (Pinf_prime, Pinf);
sage: D = P0 + 2*Pinf - Pinf_prime
```

Note we could check using exact techniques that \( 2D = K \):

```
sage: Z = K - 2*D
sage: (Z.degree() == 0, len(Z.basis_differential_space()) == S.genus, len(Z.basis_function_space()) == 1)
(True, True, True)
```

We can also check this using our Abel-Jacobi functions:

```
sage: avoid = C.places_at_infinity()
sage: Zeq, _ = S.strong_approximation(Z, avoid)
sage: Zlist = S.divisor_to_divisor_list(Zeq)
sage: AJ = S.abel_jacobi(Zlist) # long time (1 second)
sage: S.reduce_over_period_lattice(AJ).norm() < 1e-10 # long time
True
```

REFERENCES:

The initial version of this code was developed alongside [BSZ2019].
exception sage.schemes.riemann_surfaces.riemann_surface.ConvergenceError

Bases: ValueError

Error object suitable for raising and catching when Newton iteration fails.

EXAMPLES:

```
sage: from sage.schemes.riemann_surfaces.riemann_surface import ConvergenceError
sage: raise ConvergenceError("test")
Traceback (most recent call last):
...  ConvergenceError: test
sage: isinstance(ConvergenceError(), ValueError)
True
```

class sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface(f, prec=53, certification=True, differentials=None, integration_method='rigorous')

Bases: object

Construct a Riemann Surface. This is specified by the zeroes of a bivariate polynomial with rational coefficients \( f(z, w) = 0 \).

INPUT:

- \( f \) – a bivariate polynomial with rational coefficients. The surface is interpreted as the covering space of the coordinate plane in the first variable.

- \( \text{prec} \) – the desired precision of computations on the surface in bits (default: 53)

- \( \text{certification} \) – a boolean (default: True) value indicating whether homotopy continuation is certified or not. Uncertified homotopy continuation can be faster.

- \( \text{differentials} \) – (default: None). If specified, provides a list of polynomials \( h \) such that \( h / (df/dw)dz \) is a regular differential on the Riemann surface. This is taken as a basis of the regular differentials, so the genus is assumed to be equal to the length of this list. The results from the homology basis computation are checked against this value. Providing this parameter makes the computation independent from Singular. For a nonsingular plane curve of degree \( d \), an appropriate set is given by the monomials of degree up to \( d - 3 \).

- \( \text{integration_method} \) – (default: 'rigorous'). String specifying the integration method to use when calculating the integrals of differentials. The options are 'heuristic' and 'rigorous', the latter of which is often the most efficient.

EXAMPLES:

```
sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
sage: R.<z,w> = QQ[]
sage: f = w^2 - z^3 + 1
sage: RiemannSurface(f)
Riemann surface defined by polynomial f = -z^3 + w^2 + 1 = 0, with 53 bits of precision
```

Another Riemann surface with 100 bits of precision:

```
sage: S = RiemannSurface(f, prec=100); S
Riemann surface defined by polynomial f = -z^3 + w^2 + 1 = 0, with 100 bits of precision
```

(continues on next page)
We can also work with Riemann surfaces that are defined over fields with a complex embedding, but since the current interface for computing genus and regular differentials in Singular presently does not support extensions of $Q$, we need to specify a description of the differentials ourselves. We give an example of a CM elliptic curve:

```
sage: Qt.<t> = QQ[]
sage: K.<a> = NumberField(t^2-t+3,embedding=CC(0.5+1.6*I))
sage: R.<x,y> = K[]
sage: f = y^2 + y - (x^3 + (1-a)*x^2 - (2+a)*x - 2)
sage: S = RiemannSurface(f, prec=100, differentials=[1])
sage: A = S.endomorphism_basis()
sage: len(A)
2
sage: all(len(T.minpoly().roots(K)) > 0 for T in A)
True
```

The 'heuristic' integration method uses the method `integrate_vector` defined in `sage.numerical.gauss_legendre` to compute integrals of differentials. As mentioned there, this works by iteratively doubling the number of nodes used in the quadrature, and uses a heuristic based on the rate at which the result is seemingly converging to estimate the error. The 'rigorous' method uses results from [Neu2018], and bounds the algebraic integrands on circular domains using Cauchy's form of the remainder in Taylor approximation coupled to Fujiwara's bound on polynomial roots (see Bruin-DisneyHogg-Gao, in preparation). Note this method of bounding on circular domains is also implemented in `_compute_delta()`. The net result of this bounding is that one can know (an upper bound on) the number of nodes required to achieve a certain error. This means that for any given integral, assuming that the same number of nodes is required by both methods in order to achieve the desired error (not necessarily true in practice), approximately half the number of integrand evaluations are required. When the required number of nodes is high, e.g. when the precision required is high, this can make the 'rigorous' method much faster. However, the 'rigorous' method does not benefit as much from the caching of the nodes method over multiple integrals. The result of this is that, for calls of `matrix_of_integral_values()` if the computation is 'fast', the heuristic method may outperform the rigorous method, but for slower computations the rigorous method can be much faster:

```
sage: f = z*w^3 + z^3 + w
sage: p = 53
sage: Sh = RiemannSurface(f, prec=p, integration_method='heuristic')
sage: Sr = RiemannSurface(f, prec=p, integration_method='rigorous')
sage: from sage.numerical.gauss_legendre import nodes
sage: nodes.cache.clear()
sage: ct = time.time()
sage: Rh = Sh.riemann_matrix()
sage: ct1 = time.time()-ct
sage: nodes.cache.clear()
sage: ct = time.time()
sage: Rr = Sr.riemann_matrix()
sage: ct2 = time.time()-ct

```

```
sage: ct2/ct1  # random
1.2429363969691192
```

Note that for the above curve, the branch points are evenly distributed, and hence the implicit assumptions in the
heuristic method are more sensible, meaning that a higher precision is required to see the heuristic method being significantly slower than the rigorous method. For a worse conditioned curve, this effect is more pronounced:

```
sage: q = 1 / 10
sage: f = y^2 - (x^2 - 2*x + 1 + q^2) * (x^2 + 2*x + 1 + q^2)
sage: p = 500
sage: Sh = RiemannSurface(f, prec=p, integration_method='heuristic')
sage: Sr = RiemannSurface(f, prec=p, integration_method='rigorous')
sage: nodes.cache.clear()
sage: Rh = Sh.riemann_matrix()  # long time (8 seconds)
sage: nodes.cache.clear()
sage: Rr = Sr.riemann_matrix()  # long time (1 seconds)
```

This disparity in timings can get increasingly worse, and testing has shown that even for random quadrics the heuristic method can be as bad as 30 times slower.

```
abel_jacobi(divisor, verbose=False)
```

Return the Abel-Jacobi map of divisor.

Return a representative of the Abel-Jacobi map of a divisor with basepoint self._basepoint.

INPUT:

• divisor – list. A list with each entry a tuple of the form (v, P), where v is the valuation of the
divisor at point P. P as per the input to _aj_based().

• verbose – logical (default: False). Whether to report the progress of the computation, in terms of
how many elements of the list divisor have been completed.

OUTPUT: A vector of length self.genus.

EXAMPLES:

We can test that the Abel-Jacobi map between two branchpoints of a superelliptic curve of degree p is a
p-torsion point in the Jacobian:

```
sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
sage: R.<x,y> = QQ[]
sage: p = 4
sage: S = RiemannSurface(y^p-x^4+1, prec=100)
sage: divisor = [(-1, (-1, 0)), (1, (1, 0))]
```

```
sage: AJ = S.abel_jacobi(divisor)  # long time (15 seconds)
sage: AJxp = [p*z for z in AJ]  # long time
```

```
sage: bool(S.reduce_over_period_lattice(AJxp).norm()<1e-7)  # long time
True
```

```
cohomology_basis(option=1)
```

Compute the cohomology basis of this surface.

INPUT:

• option – Presently, this routine uses Singular’s adjointIdeal and passes the option parameter on.
   Legal values are 1, 2, 3, 4, where 1 is the default. See the Singular documentation for the meaning.
   The backend for this function may change, and support for this parameter may disappear.

OUTPUT:

This returns a list of polynomials g representing the holomorphic differentials \( g/(df/dw)dz \), where
\( f(z,w) = 0 \) is the equation specifying the Riemann surface.

EXAMPLES:
```python
sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
sage: R.<z,w> = QQ[]
sage: f = z^3*w + w^3 + z
sage: S = RiemannSurface(f)
sage: S.cohomology_basis()
[1, w, z]
```

curve()

Return the curve from which this Riemann surface is obtained.

Riemann surfaces explicitly obtained from a curve return that same object. For others, the curve is constructed and cached, so that an identical curve is returned upon subsequent calls.

OUTPUT: Curve from which Riemann surface is obtained.

EXAMPLES:

```python
sage: R.<x,y> = QQ[]
sage: C = Curve( y^3+x^3-1)
sage: S = C.riemann_surface()
sage: S.curve() is C
True
```

divisor_to_divisor_list(divisor, eps=None)

Turn a divisor into a list for abel_jacobi().

Given divisor in Curve(self.f).function_field().divisor_group(), consisting of places above finite points in the base, return an equivalent divisor list suitable for input into abel_jacobi().

INPUT:

- divisor – an element of Curve(self.f).function_field().divisor_group()
- eps – real number (optional); tolerance used to determine whether a complex number is close enough to a root of a polynomial

OUTPUT:

A list with elements of the form (v, (z, w)) representing the finite places.

EXAMPLES:

```python
sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
sage: R.<x,y> = QQ[]
sage: S = RiemannSurface(y^2-x^3+1)
sage: D = sum(S.places_at_branch_locus())
sage: S.divisor_to_divisor_list(D)
[(1, (1.00000000000000, 0.000000000000000)),
 (1, (-0.500000000000000 - 0.866025403784439*I, 0.000000000000000)),
 (1, (-0.500000000000000 + 0.866025403784439*I, 0.000000000000000))]
```

Todo: Currently this method can only handle places above finite points in the base. It would be useful to extend this to allow for places at infinity.

downstairs_edges()

Compute the edgeset of the Voronoi diagram.
OUTPUT:
A list of integer tuples corresponding to edges between vertices in the Voronoi diagram.

EXAMPLES:
Form a Riemann surface, one with a particularly simple branch locus:

```
sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
sage: R.<z,w> = QQ[]
sage: f = w^2 + z^3 - z^2
sage: S = RiemannSurface(f)
```

Compute the edges:

```
sage: S.downstairs_edges()
[(0, 1), (0, 5), (1, 4), (2, 3), (2, 4), (3, 5), (4, 5)]
```

This now gives an edgset which one could use to form a graph.

Note: The numbering of the vertices is given by the Voronoi package.

downstairs_graph()  
Return the Voronoi decomposition as a planar graph.

The result of this routine can be useful to interpret the labelling of the vertices. See also upstairs_graph().

OUTPUT:
The Voronoi decomposition as a graph, with appropriate planar embedding.

EXAMPLES:

```
sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
sage: R.<z,w> = QQ[]
sage: f = w^2 - z^4 + 1
sage: S = RiemannSurface(f)
sage: S.downstairs_graph()
Graph on 11 vertices
```

downstairs_graph()  
Compute the permutations of branches associated to each edge.

Over the vertices of the Voronoi decomposition around the branch locus, we label the fibres. By following along an edge, the lifts of the edge induce a permutation of that labelling.

OUTPUT:
A dictionary with as keys the edges of the Voronoi decomposition and as values the corresponding permutations.

EXAMPLES:

```
sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
sage: R.<z,w> = QQ[]
sage: f = w^2 + z^2+1
sage: S = RiemannSurface(f)
sage: S.edge_permutations()
```

(continues on next page)
sage: S.edge_permutations()
{(0, 2): (),
(0, 4): (),
(1, 2): (),
(1, 3): (0, 1),
(1, 6): (),
(2, 0): (),
(2, 1): (),
(2, 5): (0, 1),
(3, 1): (0, 1),
(3, 4): (),
(4, 0): (),
(4, 3): (),
(5, 2): (0, 1),
(5, 7): (),
(6, 1): (),
(6, 7): (),
(7, 5): (),
(7, 6): ()
}

endomorphism_basis(b=None, r=None)

Numerically compute a \( \mathbb{Z} \)-basis for the endomorphism ring.

Let \((I|M)\) be the normalized period matrix (\(M\) is the \(g \times g\) \texttt{riemann_matrix}()). We consider the system of matrix equations \(MA + C = (MB + D)M\) where \(A, B, C, D\) are \(g \times g\) integer matrices. We determine small integer (near) solutions using LLL reductions. These solutions are returned as \(2g \times 2g\) integer matrices obtained by stacking \((D|B)\) on top of \((C|A)\).

INPUT:

- \(b\) – integer (default provided). The equation coefficients are scaled by \(2^b\) before rounding to integers.
- \(r\) – integer (default: \(b/4\)). Solutions that have all coefficients smaller than \(2^r\) in absolute value are reported as actual solutions.

OUTPUT:

A list of \(2g \times 2g\) integer matrices that, for large enough \(r\) and \(b-r\), generate the endomorphism ring.

EXAMPLES:

sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
sage: R.<x,y> = QQ[]

sage: S = RiemannSurface(x^3 + y^3 + 1)

sage: B = S.endomorphism_basis(); B
#random

[1 0]
[0 -1]

[0 1]
[1 1]

sage: sorted([b.minpoly().disc() for b in B])
[-3, 1]

homology_basis()

Compute the homology basis of the Riemann surface.

OUTPUT:
A list of paths \( L = [P_1, \ldots, P_n] \). Each path \( P_i \) is of the form \((k, [p_1\ldots p_m, p_1])\), where \( k \) is the number of times to traverse the path (if negative, to traverse it backwards), and the \( p_i \) are vertices of the upstairs graph.

**EXAMPLES:**

In this example, there are two paths that form the homology basis:

```python
sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
sage: R.<z,w> = QQ[]
sage: g = w^2 - z^4 + 1
sage: S = RiemannSurface(g)
sage: S.homology_basis()  # random
[[[(1, [3, 1], [5, 0], [9, 0], [10, 0], [2, 0], [4, 0],
   [7, 1], [10, 1], [3, 1])]],
 [[[(1, [8, 0], [6, 0], [7, 0], [10, 0], [2, 0], [4, 0],
   [7, 1], [10, 1], [9, 1], [8, 0])]]]
```

In order to check that the answer returned above is reasonable, we test some basic properties. We express the faces of the downstairs graph as \( \mathbb{Z} \)-linear combinations of the edges and check that the projection of the homology basis upstairs projects down to independent linear combinations of an even number of faces:

```python
sage: dg = S.downstairs_graph()
sage: edges = dg.edges(sort=True)
sage: E = ZZ^len(edges)
sage: edge_to_E = { e[:2]: E.gen(i) for i,e in enumerate(edges) }
sage: edge_to_E.update({ (e[1],e[0]): -E.gen(i) for i,e in enumerate(edges) })
sage: face_span = E.submodule([sum(edge_to_E[e] for e in f) for f in dg.faces()])
sage: def path_to_E(path):
    ....:     k,P = path
    ....:     return k*sum(edge_to_E[(P[i][0],P[i+1][0])] for i in range(len(P)-1))
sage: hom_basis = [sum(path_to_E(p) for p in loop) for loop in S.homology_.basis()]
sage: face_span.submodule(hom_basis).rank()
2
sage: [sum(face_span.coordinate_vector(b))%2 for b in hom_basis]
[0, 0]
```

**homomorphism_basis**(other, \( b=None, r=None \))

Numerically compute a \( \mathbb{Z} \)-basis for module of homomorphisms to a given complex torus.

Given another complex torus (given as the analytic Jacobian of a Riemann surface), numerically compute a basis for the homomorphism module. The answer is returned as a list of \( 2g \times 2g \) integer matrices \( T = (D, B; C, A) \) such that if the columns of \((I|M_1)\) generate the lattice defining the Jacobian of the Riemann surface and the columns of \((I|M_2)\) do this for the codomain, then approximately we have \((I|M_2)T = (D + M_2C)(I|M_1)\), i.e., up to a choice of basis for \( \mathbb{C}^g \) as a complex vector space, we we realize \((I|M_1)\) as a sublattice of \((I|M_2)\).

**INPUT:**

- \( b \) – integer (default provided). The equation coefficients are scaled by \( 2^b \) before rounding to integers.
- \( r \) – integer (default: \( b/4 \)). Solutions that have all coefficients smaller than \( 2^r \) in absolute value are reported as actual solutions.

**OUTPUT:**

A list of \( 2g \times 2g \) integer matrices that, for large enough \( r \) and \( b-r \), generate the homomorphism module.
EXAMPLES:

```python
sage: S1 = EllipticCurve("11a1").riemann_surface()
sage: S2 = EllipticCurve("11a3").riemann_surface()
sage: [m.det() for m in S1.homomorphism_basis(S2)]

[[5]]
```

**homotopy_continuation**(edge)

Perform homotopy continuation along an edge of the Voronoi diagram using Newton iteration.

**INPUT:**

- edge – a tuple (z_start, z_end) indicating the straight line over which to perform the homotopy continuation

**OUTPUT:**

A list containing the initialised continuation data. Each entry in the list contains: the $t$ values that entry corresponds to, a list of complex numbers corresponding to the points which are reached when continued along the edge when traversing along the direction of the edge, and a value epsilon giving the minimum distance between the fibre values divided by 3. The ordering of these points indicates how they have been permuted due to the weaving of the curve.

**EXAMPLES:**

We check that continued values along an edge correspond (up to the appropriate permutation) to what is stored. Note that the permutation was originally computed from this data:

```python
sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
sage: R.<z,w> = QQ[]
sage: f = z^3*w + w^3 + z
sage: S = RiemannSurface(f)
sage: edge1 = sorted(S.edge_permutations())[0]
sage: sigma = S.edge_permutations()[edge1]
sage: edge = [S._vertices[i] for i in edge1]
sage: continued_values = S.homotopy_continuation(edge)[-1][1]
sage: stored_values = S.w_values(S._vertices[edge1[1]])
```

```python
sage: all(abs(continued_values[i]-stored_values[sigma(i)]) < 1e-8 for i in range(3))
```

True

**make_zw_interpolator**(upstairs_edge, initial_continuation=None)

Given a downstairs edge for which continuation data has been initialised, return a function that computes $z(t)$, $w(t)$, where $t$ in $[0, 1]$ is a parametrization of the edge.

**INPUT:**

- upstairs_edge – tuple ((z_start, sb), (z_end,)) giving the start and end values of the base coordinate along the straight-line path and the starting branch

- initial_continuation – list (optional); output of homotopy_continuation initialising the continuation data

**OUTPUT:**

A tuple ($g$, $d$), where $g$ is the function that computes the interpolation along the edge and $d$ is the difference of the $z$-values of the end and start point.

**EXAMPLES:**
```python
sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
sage: R.<z,w> = QQ[]
sage: f = w^2 - z^4 + 1
sage: S = RiemannSurface(f)
```

```python
sage: _ = S.homology_basis()
```

```python
sage: u_edge = [(0, 0), (1, 0)]
```

```python
sage: d_edge = tuple(u[0] for u in u_edge)
```

```python
sage: u_edge = [(S._vertices[i], j) for i, j in u_edge]
```

```python
sage: initial_continuation = S._L[d_edge]
```

```python
sage: g, d = S.make_zw_interpolator(u_edge, initial_continuation)
```

```python
sage: all(f(*g(i*0.1)).abs() < 1e-13 for i in range(10))
```

```
True
```

```
```

```
```python
sage: abs((g(1)[0] - g(0)[0]) - d) < 1e-13
```

```python
True
```

**Note:** The interpolator returned by this method can effectively hang if either $z_{\text{start}}$ or $z_{\text{end}}$ are branchpoints. In these situations it is better to take a different approach rather than continue to use the interpolator.

`matrix_of_integral_values(differentials, integration_method='heuristic')`

Compute the path integrals of the given differentials along the homology basis.

The returned answer has a row for each differential. If the Riemann surface is given by the equation $f(z, w) = 0$, then the differentials are encoded by polynomials $g$, signifying the differential $g(z, w)/(df/dw)dz$.

**INPUT:**

- `differentials` – a list of polynomials.
- `integration_method` – (default: 'heuristic'). String specifying the integration method to use. The options are 'heuristic' and 'rigorous'.

**OUTPUT:**

A matrix, one row per differential, containing the values of the path integrals along the homology basis of the Riemann surface.

**EXAMPLES:**

```python
sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
sage: R.<x,y> = QQ[]
```

```python
sage: S = RiemannSurface(x^3 + y^3 + 1)
```

```python
sage: B = S.cohomology_basis()
```

```python
sage: m = S.matrix_of_integral_values(B)
```

```python
sage: parent(m)
```

```
Full MatrixSpace of 1 by 2 dense matrices over Complex Field with 53 bits of →precision
```

```python
sage: (m[0, 0]/m[0, 1]).algdep(3).degree() # curve is CM, so the period is →quadratic
```

```
2
```

**Note:** If differentials is self.cohomology_basis(), the calculations of the integrals along the edges are written to self._integral_dict. This is as this data will be required when computing the
Abel-Jacobi map, and so it is helpful to have is stored rather than recomputing.

**monodromy_group()**

Compute local monodromy generators of the Riemann surface.

For each branch point, the local monodromy is encoded by a permutation. The permutations returned correspond to positively oriented loops around each branch point, with a fixed base point. This means the generators are properly conjugated to ensure that together they generate the global monodromy. The list has an entry for every finite point stored in `self.branch_locus`, plus an entry for the ramification above infinity.

**OUTPUT:**

A list of permutations, encoding the local monodromy at each branch point.

**EXAMPLES:**

```sage
def from_sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
R.<z, w> = QQ[]
f = z^3*w + w^3 + z
S = RiemannSurface(f)
G = S.monodromy_group(); G
[(0,1,2), (0,1), (0,2), (1,2), (1,2), (1,2), (0,1), (0,2), (0,2)]
```

The permutations give the local monodromy generators for the branch points:

```sage
def list(zip(S.branch_locus + [unsigned_infinity], G)) #abs tol 0.0000001
[(0.000000000000000, (0,1,2)),
 (-1.31362670141929, (0,1)),
 (-0.819032851784253 - 1.02703471138023*I, (0,2)),
 (-0.819032851784253 + 1.02703471138023*I, (1,2)),
 (0.292309440469772 - 1.28069133740100*I, (1,2)),
 (0.292309440469772 + 1.28069133740100*I, (1,2)),
 (1.18353676202412 - 0.569961265016465*I, (0,1)),
 (1.18353676202412 + 0.569961265016465*I, (0,2)),
 (Infinity, (0,2))]
```

We can check the ramification by looking at the cycle lengths and verify it agrees with the Riemann-Hurwitz formula:

```sage
def 2*S.genus-2 == -2*S.degree + sum(e-1 for g in G for e in g.cycle_type())
True
```

**period_matrix()**

Compute the period matrix of the surface.

**OUTPUT:**

A matrix of complex values.

**EXAMPLES:**

```sage
def from_sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
R.<z,w> = QQ[]
f = z^3*w + w^3 + z
S = RiemannSurface(f, prec=30)
M = S.period_matrix()
```
The results are highly arbitrary, so it is hard to check if the result produced is correct. The closely related `riemann_matrix` is somewhat easier to test:

```
sage: parent(M)
Full MatrixSpace of 3 by 6 dense matrices over Complex Field with 30 bits of precision
sage: M.rank()
3
```

One can check that the two methods give similar answers:

```
sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
sage: R.<x,y> = QQ[]

sage: f = y^2 - x^3 + 1
sage: S = RiemannSurface(f, integration_method="rigorous")

sage: T = RiemannSurface(f, integration_method="heuristic")

sage: RM_S = S.riemann_matrix()

sage: RM_T = T.riemann_matrix()

sage: (RM_S-RM_T).norm() < 1e-10
True
```

`places_at_branch_locus()`

Return the places above the branch locus.

Return a list of the places above the branch locus. This must be done over the base ring, and so the places are given in terms of the factors of the discriminant. Currently, this method only works when `self._R.base_ring() == QQ` as for other rings, the function field for `Curve(self.f)` is not implemented. To go from these divisors to a divisor list, see `divisor_to_divisor_list()`.

**OUTPUT:**

List of places of the functions field `Curve(self.f).function_field()`.

**EXAMPLES:**

```
sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
sage: R.<x,y> = QQ[]

sage: S = RiemannSurface(25*(x^4+y^4+1) - 34*(x^2*y^2+x^2+y^2))

sage: S.places_at_branch_locus()
[Place (x - 2, (x - 2)*y, y^2 - 17/5, y^3 - 17/5*y),
 Place (x + 2, (x + 2)*y, y^2 - 17/5, y^3 - 17/5*y),
 Place (x - 1/2, (x - 1/2)*y, y^2 - 17/20, y^3 - 17/20*y),
 Place (x + 1/2, (x + 1/2)*y, y^2 - 17/20, y^3 - 17/20*y),
 Place (x^4 - 34/25*x^2 + 1, y, y^2, y^3),
 Place (x^4 - 34/25*x^2 + 1, (x^4 - 34/25*x^2 + 1)*y, y^2 - 34/25*x^2 - 34/25,␣+(-34/25*x^2 - 34/25)*y)
```

`plot_paths()`

Make a graphical representation of the integration paths.

This returns a two dimensional plot containing the branch points (in red) and the integration paths (obtained from the Voronoi cells of the branch points). The integration paths are plotted by plotting the points that have been computed for homotopy continuation, so the density gives an indication of where numerically sensitive features occur.

**EXAMPLES:**
sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
sage: R.<x,y> = QQ[]
sage: S = RiemannSurface(y^2 - x^3 - x)
sage: S.plot_paths()
Graphics object consisting of 2 graphics primitives

plot_paths3d(thickness=0.01)

Return the homology basis as a graph in 3-space.

The homology basis of the surface is constructed by taking the Voronoi cells around the branch points and taking the inverse image of the edges on the Riemann surface. If the surface is given by the equation \( f(z, w) \), the returned object gives the image of this graph in 3-space with coordinates \((\text{Re}(z), \text{Im}(z), \text{Im}(w))\).

EXAMPLES:

sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
sage: R.<x,y> = QQ[]
S = RiemannSurface(y^2-x^3-x)
S.plot_paths3d()
Graphics3d Object

reduce_over_period_lattice(vector, method='ip', b=None, r=None, normalised=False)

Reduce a vector over the period lattice.

Given a vector of length \text{self.genus}, this method returns a vector in the same orbit of the period lattice that is short. There are two possible methods, 'svp' which returns a certified shortest vector, but can be much slower for higher genus curves, and 'ip', which is faster but not guaranteed to return the shortest vector. In general the latter will perform well when the lattice basis vectors are of similar size.

INPUT:

• vector – vector. A vector of length \text{self.genus} to reduce over the lattice.

• method – string (default: 'ip'). String specifying the method to use to reduce the vector. The options are 'ip' and 'svp'.

• b – integer (default provided): as for \text{homomorphism_basis()}, and used in its invocation if (re)calculating said basis.

• r – integer (default: b/4), as for \text{homomorphism_basis()}, and used in its invocation if (re)calculating said basis.

• normalised – logical (default: False). Whether to use the period matrix with the differentials normalised s.t. the \( A \)-matrix is the identity.

OUTPUT:

Complex vector of length \text{self.genus} in the same orbit as \text{vector} in the lattice.

EXAMPLES:

We can check that the lattice basis vectors themselves are reduced to zero:

sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
sage: R.<x,y> = QQ[]
sage: S = RiemannSurface(y^2-x^3-x^5+1)
sage: epsilon = S._RR(2)^(-S._prec+1)
sage: for vector in S.period_matrix().columns():
....:     print(bool(S.reduce_over_period_lattice(vector).norm()<epsilon))
We can also check that the method 'svp' always gives a smaller norm than 'ip':

```python
sage: for vector in S.period_matrix().columns():
    n1 = S.reduce_over_period_lattice(vector).norm()
    n2 = S.reduce_over_period_lattice(vector, method="svp").norm()
    print(bool(n2<=n1))
```

```plaintext
True
True
True
True
```

**riemann_matrix()**

Compute the Riemann matrix.

**OUTPUT:**

A matrix of complex values.

**EXAMPLES:**

```python
sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
sage: R.<z,w> = QQ[]
sage: f = z^3*w + w^3 + z
sage: S = RiemannSurface(f, prec=60)
```

```python
sage: M = S.riemann_matrix()
```

The Klein quartic has a Riemann matrix with values in a quadratic field:

```python
sage: x = polygen(QQ)
```

```python
sage: K.<a> = NumberField(x^2-x+2)
```

```python
sage: all(len(m.algdep(6).roots(K)) > 0 for m in M.list())
```

```
True
```

**rigorous_line_integral(upstairs_edge, differentials, bounding_data)**

Perform vectorized integration along a straight path.

Using the error bounds for Gauss-Legendre integration found in [Neu2018] and a method for bounding an algebraic integrand on a circular domains using Cauchy’s form of the remainder in Taylor approximation coupled to Fujiwara’s bound on polynomial roots (see Bruin-DisneyHogg-Gao, in preparation), this method calculates (semi-)rigorously the integral of a list of differentials along an edge of the upstairs graph.

**INPUT:**

- **upstairs_edge** – tuple. Either a pair of integer tuples corresponding to an edge of the upstairs graph, or a tuple ((z_start, sb), (z_end, )) as in the input of make_zw_interpolator.
- **differentials** – a list of polynomials; a polynomial \( g \) represents the differential \( g(z, w)/(df/dw)dz \) where \( f(z, w) = 0 \) is the equation defining the Riemann surface.
- **bounding_data** – tuple containing the data required for bounding the integrands. This should be in the form of the output from _bounding_data().
OUTPUT:

A complex number, the value of the line integral.

EXAMPLES:

```python
sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
sage: R.<z,w> = QQ[]
sage: f = w^2 - z^4 + 1
sage: S = RiemannSurface(f); S
Riemann surface defined by polynomial f = -z^4 + w^2 + 1 = 0, with 53 bits of precision
```

Since we make use of data from homotopy continuation, we need to compute the necessary data:

```python
sage: _ = S.homology_basis()
sage: differentials = S.cohomology_basis()
```

Todo: Note that bounding_data contains the information of the integrands, so one may want to check for consistency between bounding_data and differentials. If so one would not want to do so at the expense of speed.

Moreover, the current implementation bounds along a line by splitting it up into segments, each of which can be covered entirely by a single circle, and then placing inside that the ellipse required to bound as per [Neu2018]. This is reliably more efficient than the heuristic method, especially in poorly-conditioned cases where discriminant points are close together around the edges, but in the case where the branch locus is well separated, it can require slightly more nodes than necessary. One may want to include a method here to transition in this regime to an algorithm that covers the entire line with one ellipse, then bounds along that ellipse with multiple circles.

**rosati_involution**(*R*)

Compute the Rosati involution of an endomorphism.

The endomorphism in question should be given by its homology representation with respect to the symplectic basis of the Jacobian.

**INPUT:**


**OUTPUT:**

The result of applying the Rosati involution to *R*. 
EXAMPLES:

```python
sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
sage: A.<x,y> = QQ[]
sage: S = RiemannSurface(y^2 - (x^6 + 2*x^4 + 4*x^2 + 8), prec = 100)
sage: Rs = S.endomorphism_basis()
sage: S.rosati_involution(S.rosati_involution(Rs[1])) == Rs[1]
True
```

**simple_vector_line_integral(upstairs_edge, differentials)**

Perform vectorized integration along a straight path.

**INPUT:**

- upstairs_edge – tuple. Either a pair of integer tuples corresponding to an edge of the upstairs graph, or a tuple (((z_start, sb), (z_end, )) as in the input of make_zw_interpolator.
- differentials – a list of polynomials; a polynomial $g$ represents the differential $g(z, w)/(df/dw)dz$

where $f(z, w) = 0$ is the equation defining the Riemann surface.

**OUTPUT:**

A complex number, the value of the line integral.

**EXAMPLES:**

```python
sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
sage: R.<z,w> = QQ[]
sage: f = w^2 - z^4 + 1
sage: S = RiemannSurface(f); S
Riemann surface defined by polynomial f = -z^4 + w^2 + 1 = 0, with 53 bits of →precision
```

Since we make use of data from homotopy continuation, we need to compute the necessary data:

```python
sage: M = S.riemann_matrix()
sage: differentials = S.cohomology_basis()
sage: S.simple_vector_line_integral([(0, 0), (1, 0)], differentials) #abs tol 0.
(1.14590610929717e-16 - 0.352971844594760*I)
```

**Note:** Uses data that homology_basis() initializes, and may give incorrect values if homology_basis() has not initialized them. In practice it is more efficient to set differentials to a fast-callable version of differentials to speed up execution.

**strong_approximation(divisor, S)**

Apply the method of strong approximation to a divisor.

As described in [Neu2018], apply the method of strong approximation to divisor with list of places to avoid $S$. Currently, this method only works when self._R.base_ring() == QQ as for other rings, the function field for Curve(self.f) is not implemented.

**INPUT:**

- divisor – an element of Curve(self.f).function_field().divisor_group()
- $S$ – list of places to avoid
OUTPUT:

A tuple \((D, B)\), where \(D\) is a new divisor, linearly equivalent to \(\text{divisor}\), but not intersecting \(S\), and \(B\) is a list of tuples \((v, b)\) where \(b\) are the functions giving the linear equivalence, added with multiplicity \(v\).

EXAMPLES:

```
sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurfacesage: R.<x,y> = QQ[]
sage: S = RiemannSurface(y^2-x^3+1)
sage: avoid = Curve(S.f).places_at_infinity()
sage: D = 1*avoid[0]
sage: S.strong_approximation(D, avoid)
(- Place (x - 2, (x - 2)*y)
 + Place (x - 1, y)
 + Place (x^2 + x + 1, y),
 [(1, (1/(x - 2))*y))]
```

\textbf{symplectic_automorphism_group}(\text{endo\_basis=None, b=None, r=None})

Numerically compute the symplectic automorphism group as a permutation group.

INPUT:

- \text{endo\_basis} (default: None) – a \(\mathbb{Z}\)-basis of the endomorphisms of \(\text{self}\), as obtained from \text{endomorphism\_basis()}. If you have already calculated this basis, it saves time to pass it via this keyword argument. Otherwise the method will calculate it.

- \text{b} – integer (default provided): as for \text{homomorphism\_basis()}, and used in its invocation if (re)calculating said basis.

- \text{r} – integer (default: \(b/4\)). as for \text{homomorphism\_basis()}, and used in its invocation if (re)calculating said basis.

OUTPUT:

The symplectic automorphism group of the Jacobian of the Riemann surface. The automorphism group of the Riemann surface itself can be recovered from this; if the curve is hyperelliptic, then it is identical, and if not, then one divides out by the central element corresponding to multiplication by -1.

EXAMPLES:

```
sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
sage: A.<x,y> = QQ[]
sage: S = RiemannSurface(y^2 - (x^6 + 2*x^4 + 4*x^2 + 8), prec = 100)
sage: G = S.symplectic_automorphism_group()
sage: G.as_permutation_group().is_isomorphic(DihedralGroup(4))
True
```

\textbf{symplectic_isomorphisms}(\text{other=None, hom\_basis=None, b=None, r=None})

Numerically compute symplectic isomorphisms.

INPUT:

- \text{other} (default: self) – the codomain, another Riemann surface.

- \text{hom\_basis} (default: None) – a \(\mathbb{Z}\)-basis of the homomorphisms from \(\text{self}\) to \(\text{other}\), as obtained from \text{homomorphism\_basis()}. If you have already calculated this basis, it saves time to pass it via this keyword argument. Otherwise the method will calculate it.

- \text{b} – integer (default provided): as for \text{homomorphism\_basis()}, and used in its invocation if (re)calculating said basis.
• \( r \) – integer (default: \( b/4 \)). as for \texttt{homomorphism}\_\texttt{basis()}, and used in its invocation if (re)calculating said basis.

OUTPUT:
This returns the combinations of the elements of \texttt{homomorphism}\_\texttt{basis()} that correspond to symplectic isomorphisms between the Jacobians of \texttt{self} and \texttt{other}.

EXAMPLES:

```python
sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
sage: A.<x,y> = QQ[]
sage: f = y^2 - (x^6 + 2*x^4 + 4*x^2 + 8)
sage: X = RiemannSurface(f, prec=100)
sage: P = X.period_matrix()
sage: g = y^2 - (x^6 + x^4 + x^2 + 1)
sage: Y = RiemannSurface(g, prec=100)
sage: Q = Y.period_matrix()
sage: Rs = X.symplectic_isomorphisms(Y)
sage: Ts = X.tangent_representation_numerical(Rs)
sage: test1 = all(((T*P - Q*R).norm() < 2^(-80)) for [T, R] in zip(Ts, Rs))
sage: test2 = all(det(R) == 1 for R in Rs)
sage: test1 and test2
True
```

\texttt{tangent}\_\texttt{representation}\_\texttt{algebraic}(\texttt{Rs}, \texttt{other=None, epscomp=None})

Compute the algebraic tangent representations corresponding to the homology representations in \texttt{Rs}.

The representations on homology \texttt{Rs} have to be given with respect to the symplectic homology basis of the Jacobian of \texttt{self} and \texttt{other}. Such matrices can for example be obtained via \texttt{endomorphism}\_\texttt{basis()}.

Let \( P \) and \( Q \) be the period matrices of \texttt{self} and \texttt{other}. Then for a homology representation \( R \), the corresponding tangential representation \( T \) satisfies \( TP = QR \).

INPUT:
• \texttt{Rs} – a set of matrices on homology to be converted to their tangent representations.
• \texttt{other} (default: \texttt{self}) – the codomain, another Riemann surface.
• \texttt{epscomp} – real number (default: \( 2^{\texttt{prec} + 30} \)). Used to determine whether a complex number is close enough to a root of a polynomial.

OUTPUT:
The algebraic tangent representations of the matrices in \texttt{Rs}.

EXAMPLES:

```python
sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
sage: A.<x,y> = QQ[]
sage: S = RiemannSurface(y^2 - (x^6 + 2*x^4 + 4*x^2 + 8), prec = 100)
sage: Rs = S.endomorphism_basis()
sage: Ts = S.tangent_representation_algebraic(Rs)
sage: Ts[0].base_ring().maximal_order().discriminant() == 8
True
```

\texttt{tangent}\_\texttt{representation}\_\texttt{numerical}(\texttt{Rs}, \texttt{other=None})

Compute the numerical tangent representations corresponding to the homology representations in \texttt{Rs}.
The representations on homology $R_s$ have to be given with respect to the symplectic homology basis of the Jacobian of $\text{self}$ and $\text{other}$. Such matrices can for example be obtained via `endomorphism_basis()`.

Let $P$ and $Q$ be the period matrices of $\text{self}$ and $\text{other}$. Then for a homology representation $R$, the corresponding tangential representation $T$ satisfies $TP = QR$.

**INPUT:**

- $Rs$ – a set of matrices on homology to be converted to their tangent representations.
- $\text{other}$ (default: $\text{self}$) – the codomain, another Riemann surface.

**OUTPUT:**

The numerical tangent representations of the matrices in $Rs$.

**EXAMPLES:**

```python
sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
sage: A.<x,y> = QQ[]
sage: S = RiemannSurface(y^2 - (x^6 + 2*x^4 + 4*x^2 + 8), prec = 100)
sage: P = S.period_matrix()
sage: Rs = S.endomorphism_basis()
sage: Ts = S.tangent_representation_numerical(Rs)
sage: all(((T*P - P*R).norm() < 2^(-80)) for [T, R] in zip(Ts, Rs))
True
```

### upstairs_edges()

Compute the edgeset of the lift of the downstairs graph onto the Riemann surface.

**OUTPUT:**

An edgeset between vertices $(i, j)$, where $i$ corresponds to the $i$-th point in the Voronoi diagram vertices, and $j$ is the $j$-th $w$-value associated with that point.

**EXAMPLES:**

```python
sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
sage: R.<z,w> = QQ[]
sage: f = w^2 + z^3 - z^2
sage: S = RiemannSurface(f)
sage: edgeset = S.upstairs_edges()
sage: len(edgeset) == S.degree*len(S.downstairs_edges())
True
sage: {(v[0],w[0]) for v,w in edgeset} == set(S.downstairs_edges())
True
```

### upstairs_graph()

Return the graph of the upstairs edges.

This method can be useful for generating paths in the surface between points labelled by upstairs vertices, and verifying that a homology basis is likely computed correctly. See also `downstairs_graph()`.

**OUTPUT:**

The homotopy-continued Voronoi decomposition as a graph, with appropriate 3D embedding.

**EXAMPLES:**
sage: R.<z,w> = QQ[]
sage: S = Curve(w^2-z^4+1).riemann_surface()
sage: G = S.upstairs_graph(); G
Graph on 22 vertices
sage: G.genus()
1
sage: G.is_connected()
True

w_values(z0)

Return the points lying on the surface above z0.

INPUT:

• z0 – (complex) a point in the complex z-plane.

OUTPUT:

A set of complex numbers corresponding to solutions of f(z0, w) = 0.

EXAMPLES:

sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface
sage: R.<z,w> = QQ[]
sage: f = w^2 - z^4 + 1
sage: S = RiemannSurface(f)
Find the w-values above the origin, i.e. the solutions of w^2 + 1 = 0:

sage: S.w_values(0)  # abs tol 1e-14
[-1.00000000000000*I, 1.00000000000000*I]
Note that typically the method returns a list of length self.degree, but that at ramification points, this may no longer be true:

sage: S.w_values(1)  # abs tol 1e-14
[0.000000000000000]

class sage.schemes.riemann_surfaces.riemann_surface.RiemannSurfaceSum(L)

Bases: RiemannSurface

Represent the disjoint union of finitely many Riemann surfaces.

Rudimentary class to represent disjoint unions of Riemann surfaces. Exists mainly (and this is the only functionality actually implemented) to represents direct products of the complex tori that arise as analytic Jacobians of Riemann surfaces.

INPUT:

• L – list of RiemannSurface objects

EXAMPLES:

sage: _.<x> = QQ[]
sage: SC = HyperellipticCurve(x^6-2*x^4+3*x^2-7).riemann_surface(prec=60)
sage: S1 = HyperellipticCurve(x^3-2*x^2+3*x-7).riemann_surface(prec=60)
sage: S2 = HyperellipticCurve(1-2*x^3+3*x^2-7*x^3).riemann_surface(prec=60)
(continues on next page)
sage: len(SC.homomorphism_basis(S1+S2))
2

period_matrix()

Return the period matrix of the surface.

This is just the diagonal block matrix constructed from the period matrices of the constituents.

EXAMPLES:

sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface,
˓→RiemannSurfaceSum
sage: R.<x,y> = QQ[]
sage: S1 = RiemannSurface(y^2-x^3-x-1)
sage: S2 = RiemannSurface(y^2-x^3-x-5)
sage: S = RiemannSurfaceSum([S1,S2])
sage: S1S2 = S1.period_matrix().block_sum(S2.period_matrix())
sage: S.period_matrix() == S1S2[[0,1],[0,2,1,3]]
True

riemann_matrix()

Return the normalized period matrix of the surface.

This is just the diagonal block matrix constructed from the Riemann matrices of the constituents.

EXAMPLES:

sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface,
˓→RiemannSurfaceSum
sage: R.<x,y> = QQ[]
sage: S1 = RiemannSurface(y^2-x^3-x-1)
sage: S2 = RiemannSurface(y^2-x^3-x-5)
sage: S = RiemannSurfaceSum([S1,S2])
sage: S.riemann_matrix() == S1.riemann_matrix().block_sum(S2.riemann_matrix())
True

sage.schemes.riemann_surfaces.riemann_surface.bisect(L, t)

Find position in a sorted list using bisection.

Given a list \( L = [(t_0, \ldots), (t_1, \ldots), \ldots(t_n, \ldots)] \) with increasing \( t_i \), find the index \( i \) such that \( t_i \leq t < t_{i+1} \) using bisection. The rest of the tuple is available for whatever use required.

INPUT:

- \( L \) – A list of tuples such that the first term of each tuple is a real number between 0 and 1. These real numbers must be increasing.

- \( t \) – A real number between \( t_0 \) and \( t_n \).

OUTPUT:

An integer \( i \), giving the position in \( L \) where \( t \) would be in

EXAMPLES:

Form a list of the desired form, and pick a real number between 0 and 1:
Another example which demonstrates that if \( t \) is equal to one of the \( t_i \), it returns that index:

```
sage: L = [(0.0, 'a'), (0.1, 'b'), (0.45, 'c'), (0.5, 'd'), (0.65, 'e'), (1.0, 'f')]
sage: t = 0.5
sage: bisect(L,t)
3
```

```
sage.schemes.riemann_surfaces.riemann_surface.differential_basis_baker(f)

Compute a differential basis for a curve that is nonsingular outside \((1:0:0),(0:1:0),(0:0:1)\).

Baker’s theorem tells us that if a curve has its singularities at the coordinate vertices and meets some further easily tested genericity criteria, then we can read off a basis for the regular differentials from the interior of the Newton polygon spanned by the monomials. While this theorem only applies to special plane curves it is worth implementing because the analysis is relatively cheap and it applies to a lot of commonly encountered curves (e.g., curves given by a hyperelliptic model). Other advantages include that we can do the computation over any exact base ring (the alternative Singular based method for computing the adjoint ideal requires the rationals), and that we can avoid being affected by subtle bugs in the Singular code.

None is returned when \( f \) does not describe a curve of the relevant type. If \( f \) is of the relevant type, but is of genus 0 then [] is returned (which are both False values, but they are not equal).

INPUT:

• \( f \) – a bivariate polynomial

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: f = x^3 + y^3 + x^5*y^5
sage: differential_basis_baker(f)
[y^2, x*y, x*y^2, x^2, x^2*y, x^2*y^2, x^3*y^2, x^3*y^3]
sage: f = y^2 - (x-3)^2*x
sage: differential_basis_baker(f)
is None
True
sage: differential_basis_baker(x^2+y^2-1)
[]
```

```
sage.schemes.riemann_surfaces.riemann_surface.find_closest_element(item, lst)

Return the index of the closest element of a list.

Given List and item, return the index of the element \( l \) of List which minimises \((item-l).abs()\). If there are multiple such elements, the first is returned.

INPUT:

• item – value to minimize the distance to over the list

• lst – list to look for closest element in

EXAMPLES:

```
Sage: from sage.schemes.riemann_surfaces.riemann_surface import find_closest_element
Sage: i = 5
Sage: l = list(range(10))
Sage: i == find_closest_element(i, l)
True

Note that this method does no checks on the input, but will fail for inputs where the absolute value or subtraction do not make sense.

sage.schemes.riemann_surfaces.riemann_surface.integer_matrix_relations(M1, M2, b=None, r=None)

Determine integer relations between complex matrices.

Given two square matrices with complex entries of size \( g, h \) respectively, numerically determine an (approximate) \( \mathbb{Z} \)-basis for the \( 2g \times 2h \) matrices with integer entries of the shape \( (D, B; C, A) \) such that \( B+M1*A=(D+M1*C)*M2 \). By considering real and imaginary parts separately we obtain \( 2gh \) equations with real coefficients in \( 4gh \) variables. We scale the coefficients by a constant \( 2^b \) and round them to integers, in order to obtain an integer system of equations. Standard application of LLL allows us to determine near solutions.

The user can specify the parameter \( b \), but by default the system will choose a \( b \) based on the size of the coefficients and the precision with which they are given.

**INPUT:**

- \( M1 \) – square complex valued matrix
- \( M2 \) – square complex valued matrix of same size as \( M1 \)
- \( b \) – integer (default provided). The equation coefficients are scaled by \( 2^b \) before rounding to integers.
- \( r \) – integer (default: \( b/4 \)). The vectors found by LLL that satisfy the scaled equations to within \( 2^r \) are reported as solutions.

**OUTPUT:**

A list of \( 2g \times 2h \) integer matrices that, for large enough \( r, b−r \), generate the \( \mathbb{Z} \)-module of relevant transformations.

**EXAMPLES:**

```python
sage: from sage.schemes.riemann_surfaces.riemann_surface import integer_matrix_relations
sage: M1=M2=matrix(CC,2,2,[sqrt(d) for d in [2,-3,-3,-6]])
sage: T=integer_matrix_relations(M1,M2)
sage: id=parent(M1)(1)
sage: M1t=[id.augment(M1) * t for t in T]
sage: [((m[:,:2]^(-1)*m)[:,2:] - M2).norm() < 1e-13 for m in M1t]
[True, True]
```

**sage.schemes.riemann_surfaces.riemann_surface.numerical_inverse(C)**

Compute numerical inverse of a matrix via LU decomposition

**INPUT:**

- \( C \) – A real or complex invertible square matrix

**EXAMPLES:**
```python
sage: C = matrix(CC, 3, 3, [-4.5606e-31 + 1.2326e-31*I, ...: -0.21313 + 0.24166*I, ...: -3.4513e-31 + 0.16111*I, ...: -1.0175 + 9.8608e-32*I, ...: 0.30912 + 0.19962*I, ...: -4.9304e-32 + 0.39923*I, ...: 0.96793 - 3.4513e-31*I, ...: -0.091587 + 0.19276*I, ...: 3.9443e-31 + 0.38552*I])
```

```python
sage: from sage.schemes.riemann_surfaces.riemann_surface import numerical_inverse
```

```python
sage: 3e-16 < (C^-1*C-C^0).norm() < 1e-15
True
sage: (numerical_inverse(C)*C-C^0).norm() < 3e-16
True
```

```python
sage.schemes.riemann_surfaces.riemann_surface.reparameterize_differential_minpoly(minpoly, z0)
```

Rewrites a minimal polynomial to write is around $z_0$.

Given a minimal polynomial $m(z, g)$, where $g$ corresponds to a differential on the surface (that is, it is represented as a rational function, and implicitly carries a factor $dz$), we rewrite the minpoly in terms of variables $\bar{z}, \bar{g}$ s.t now $\bar{z} = 0 \iff z = z_0$.

**INPUT:**
- `minpoly` – a polynomial in two variables, where the first variable corresponds to the base coordinate on the Riemann surface
- `z0` – complex number or infinity; the point about which to reparameterize

**OUTPUT:**
A polynomial in two variables giving the reparameterize minimal polynomial.

**EXAMPLES:**
On the curve given by $w^2 - z^3 + 1 = 0$, we have differential $\frac{dz}{2w} = \frac{dz}{2\sqrt{z^3-1}}$ with minimal polynomial $g^2(z^3 - 1) - 1/4 = 0$. We can make the substitution $\bar{z} = z^{-1}$ to parameterise the differential about $z = \infty$ as

$$\frac{-\bar{z}^{-2}d\bar{z}}{2\sqrt{\bar{z}^{-3}-1}} = \frac{-d\bar{z}}{2\sqrt{\bar{z}(1-\bar{z}^3)}}.$$

Hence the transformed differential should have minimal polynomial $\bar{g}^2(1-\bar{z}^3) - 1/4 = 0$, and we can check this:

```python
sage: from sage.schemes.riemann_surfaces.riemann_surface import RiemannSurface,
˓→reparameterize_differential_minpoly
sage: R.<z,w> = QQ[]
sage: S = RiemannSurface(w^2-z^3+1)
sage: minpoly = S._cohomology_basis_bounding_data[1][0][2]
sage: z0 = Infinity
sage: reparameterize_differential_minpoly(minpoly, z0)
-zbar^4*gbar^2 + zbar*gbar^2 - 1/4
```

We can further check that reparameterising about 0 is the identity operation:
Note: As part of the routine, when reparameterising about infinity, a rational function is reduced and then the numerator is taken. Over an inexact ring this is numerically unstable, and so it is advisable to only reparameterize about infinity over an exact ring.

```
sage: reparameterize_differential_minpoly(minpoly, 0)(*minpoly.parent().gens()) == →minpoly
True
```

sage.schemes.riemann_surfaces.riemann_surface.voronoi_ghost(cpoints, n=6, CC=Complex Double Field)

Convert a set of complex points to a list of real tuples \((x, y)\), and appends \(n\) points in a big circle around them.

The effect is that, with \(n \geq 3\), a Voronoi decomposition will have only finite cells around the original points. Furthermore, because the extra points are placed on a circle centered on the average of the given points, with a radius \(3/2\) times the largest distance between the center and the given points, these finite cells form a simply connected region.

INPUT:

- `cpoints` – a list of complex numbers

OUTPUT:

A list of real tuples \((x, y)\) consisting of the original points and a set of points which surround them.

EXAMPLES:

```
sage: from sage.schemes.riemann_surfaces.riemann_surface import voronoi_ghost
sage: L = [1 + 1*I, 1 - 1*I, -1 + 1*I, -1 - 1*I]
sage: voronoi_ghost(L)  # abs tol 1e-6
```

```
[(1.0, 1.0),
 (1.0, -1.0),
 (-1.0, 1.0),
 (-1.0, -1.0),
 (2.121320343559643, 0.0),
 (1.0606601717798216, 1.8371173070873836),
 (-1.060660171779821, -1.8371173070873845),
 (-2.121320343559643, -1.8371173070873832),
 (1.0606601717798223, 1.8371173070873845)]
```
CHAPTER

ELEVEN

INDICES AND TABLES

• Index
• Module Index
• Search Page
S

sage.schemes.curves.affine_curve, 13
sage.schemes.curves.closed_point, 85
sage.schemes.curves.constructor,
1
sage.schemes.curves.curve, 5
sage.schemes.curves.point, 77
sage.schemes.curves.projective_curve, 43
sage.schemes.jacobians.abstract_jacobian, 93
sage.schemes.plane_conics.con_field, 97
sage.schemes.plane_conics.con_finite_field,
119
sage.schemes.plane_conics.con_number_field,
122
sage.schemes.plane_conics.con_rational_field,
115
sage.schemes.plane_conics.con_rational_function_field,
120
sage.schemes.plane_conics.constructor, 95
sage.schemes.plane_quartics.quartic_constructor,
125
sage.schemes.plane_quartics.quartic_generic,
125
sage.schemes.riemann_surfaces.riemann_surface,
127
<table>
<thead>
<tr>
<th>A</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>abel_jacobi()</td>
<td>(sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface.method), 131</td>
</tr>
<tr>
<td>affine()</td>
<td>(sage.schemes.curves.closed_point.IntegralProjectiveCurveClosedPoint method), 90</td>
</tr>
<tr>
<td>affine_patch()</td>
<td>(sage.schemes.curves.projective_curve.ProjectiveCurvePoint method), 79</td>
</tr>
<tr>
<td>AffineCurve</td>
<td>(class in sage.schemes.curves.affine_curve), 15</td>
</tr>
<tr>
<td>AffineCurve_field</td>
<td>(class in sage.schemes.curves.affine_curve), 16</td>
</tr>
<tr>
<td>AffineCurvePoint_field</td>
<td>(class in sage.schemes.curves.point), 77</td>
</tr>
<tr>
<td>AffinePlaneCurve</td>
<td>(class in sage.schemes.curves.affine_curve), 26</td>
</tr>
<tr>
<td>AffinePlaneCurve_field</td>
<td>(class in sage.schemes.curves.affine_curve), 33</td>
</tr>
<tr>
<td>AffinePlaneCurve_point_field</td>
<td>(class in sage.schemes.curves.point), 78</td>
</tr>
<tr>
<td>AffinePlaneCurve_point_field_finite_field</td>
<td>(class in sage.schemes.curves.point), 79</td>
</tr>
<tr>
<td>arithmetic_genus()</td>
<td>(sage.schemes.curves.projective_curve.ProjectiveCurve_field method), 56</td>
</tr>
<tr>
<td>arithmetic_genus()</td>
<td>(sage.schemes.curves.projective_curve.ProjectiveCurve_field method), 69</td>
</tr>
<tr>
<td>B</td>
<td></td>
</tr>
<tr>
<td>base_extend()</td>
<td>(sage.schemes.jacobians.abstract_jacobian.Jacobian_generic method), 93</td>
</tr>
<tr>
<td>base_extend()</td>
<td>(sage.schemes.plane_conics.con_field.ProjectiveConic_field method), 97</td>
</tr>
<tr>
<td>bisect()</td>
<td>(module sage.schemes.riemann_surfaces.riemann_surface), 148</td>
</tr>
<tr>
<td>blowup()</td>
<td>(sage.schemes.curves.affine_curve.AffineCurve_field method), 16</td>
</tr>
<tr>
<td>braid_monodromy()</td>
<td>(sage.schemes.curves.affine_curve.AffinePlaneCurve_field method), 33</td>
</tr>
<tr>
<td>C</td>
<td></td>
</tr>
<tr>
<td>cache_point()</td>
<td>(sage.schemes.plane_conics.con_field.ProjectiveConic_field method), 128</td>
</tr>
<tr>
<td>change_ring()</td>
<td>(sage.schemes.curves.curve.Curve_generic method), 97</td>
</tr>
<tr>
<td>change_ring()</td>
<td>(sage.schemes.jacobians.abstract_jacobian.Jacobian_generic method), 94</td>
</tr>
<tr>
<td>closed_point()</td>
<td>(sage.schemes.curves.point.IntegralAffineCurvePoint method), 79</td>
</tr>
<tr>
<td>closed_points()</td>
<td>(sage.schemes.curves.affine_curve.IntegralAffineCurvePoint method), 81</td>
</tr>
<tr>
<td>closed_points()</td>
<td>(sage.schemes.curves.projective_curve.IntegralProjectiveCurvePoint method), 41</td>
</tr>
<tr>
<td>closed_points()</td>
<td>(sage.schemes.curves.projective_curve.IntegralProjectiveCurvePoint method), 49</td>
</tr>
<tr>
<td>coefficients()</td>
<td>(sage.schemes.plane_conics.con_field.ProjectiveConic_field method), 97</td>
</tr>
<tr>
<td>cohomology_basis()</td>
<td>(sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface method), 131</td>
</tr>
<tr>
<td>Conic()</td>
<td>(module sage.schemes.plane_conics.constructor), 95</td>
</tr>
<tr>
<td>ConvergenceError</td>
<td></td>
</tr>
<tr>
<td>curve()</td>
<td>(module sage.schemes.curves.constructor), 1</td>
</tr>
<tr>
<td>curve()</td>
<td>(sage.schemes.curves.closed_point.IntegralCurveClosedPoint method), 88</td>
</tr>
<tr>
<td>curve()</td>
<td>(sage.schemes.jacobians.abstract_jacobian.Jacobian_generic method), 94</td>
</tr>
<tr>
<td>curve()</td>
<td>(sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface method), 132</td>
</tr>
<tr>
<td>Curve()</td>
<td>(module sage.schemes.curves.constructor), 119</td>
</tr>
<tr>
<td>Curve()</td>
<td>(sage.schemes.curves.closed_point.IntegralCurveClosedPoint method), 86</td>
</tr>
<tr>
<td>Curve()</td>
<td>(sage.schemes.curves.closed_point.IntegralProjectiveCurveClosedPoint method), 86</td>
</tr>
<tr>
<td>Curve()</td>
<td>(sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface method), 132</td>
</tr>
<tr>
<td>Curve()</td>
<td>(sage.schemes.curves.closed_point.IntegralProjectiveCurveClosedPoint method), 86</td>
</tr>
<tr>
<td>D</td>
<td></td>
</tr>
<tr>
<td>defining_polynomial()</td>
<td>(sage.schemes.curves.curve.Curve_generic method), 56</td>
</tr>
</tbody>
</table>

157
Curves, Release 10.1

- **degree()** (sage.schemes.curves.projective_curve.ProjectivePlaneCurve method), 58
- **derivative_matrix()** (sage.schemes.plane_conics_field.ProjectiveConic_field method), 96
- **determinant()** (sage.schemes.plane_conics_field.ProjectiveConic_field method), 98
- **diagonal_matrix()** (sage.schemes.plane_conics_field.ProjectiveConic_field method), 99
- **diagonalization()** (sage.schemes.plane_conics_field.ProjectiveConic_field method), 99
- **differential_basis_baker()** (in module sage.schemes.riemann_surfaces.riemann_surface), 149
- **dimension()** (sage.schemes.curves.curve.Curve_generic method), 6
- **divisor()** (sage.schemes.curves.curve.Curve_generic method), 6
- **divisor_group()** (sage.schemes.curves.curve.Curve_generic method), 7
- **divisor_of_function()** (sage.schemes.curves.affine_curve.IntegralAffineCurve method), 36
- **divisor_of_function()** (sage.schemes.curves.projective_curve.IntegralProjectiveCurve method), 46
- **divisor_to_divisor_list()** (sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface method), 132
- **downstairs_edges()** (sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface method), 132
- **downstairs_graph()** (sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface method), 133
- **edge_permutations()** (sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface method), 133
- **endomorphism_basis()** (sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface method), 134
- **excellent_position()** (sage.schemes.curves.projective_curve.ProjectivePlaneCurve method), 59
- **find_closest_element()** (in module sage.schemes.riemann_surfaces.riemann_surface), 149
- **find_point()** (sage.schemes.plane_conics_field.ProjectiveConic_field method), 121
- **function()** (sage.schemes.curves.affine_curve.IntegralAffineCurve method), 36
- **function()** (sage.schemes.curves.projective_curve.IntegralProjectiveCurve method), 46
- **function_field()** (sage.schemes.curves.affine_curve.IntegralAffineCurve method), 37
- **fundamental_group()** (sage.schemes.curves.projective_curve.IntegralProjectiveCurvet field method), 59
- **fundamental_group()** (sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface method), 70
- **has_rational_point()** (sage.schemes.plane_conics_field.ProjectiveConic_field method), 101
- **has_rational_point()** (sage.schemes.plane_conics_field.ProjectiveConic_field method), 101
- **has_rational_point()** (sage.schemes.plane_conics_finite_field.ProjectiveConic_finite_field method), 119
- **has_rational_point()** (sage.schemes.plane_conics_number_field.ProjectiveConic_number_field method), 112
- **has_rational_point()** (sage.schemes.plane_conics_rational_field.ProjectiveConic_rational_field method), 116
- **has_rational_point()** (sage.schemes.plane_conics_rational_function_field.ProjectiveConic_rational_function_field method), 122
- **Hasse_bounds()** (in module sage.schemes.curves.projective_curve), 102
- **hom()** (sage.schemes.curves.projective_curve.IntegralProjectiveCurve method), 45
- **homology_basis()** (sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface method), 134
- **homomorphism_basis()** (sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface method), 135
- **edge_permutations()**
- **divisor()**
- **divisor_group()**
- **divisor_of_function()**
- **divisor_to_divisor_list()**
- **downstairs_edges()**
- **downstairs_graph()**
- **edge_permutations()**
- **divisor()**
- **divisor_group()**
- **divisor_of_function()**
- **divisor_to_divisor_list()**
- **downstairs_edges()**
- **downstairs_graph()**
- **edge_permutations()**
- **divisor()**
- **divisor_group()**
- **divisor_of_function()**
- **divisor_to_divisor_list()**
- **downstairs_edges()**
- **downstairs_graph()**
- **edge_permutations()**
- **divisor()**
- **divisor_group()**
- **divisor_of_function()**
- **divisor_to_divisor_list()**
- **downstairs_edges()**
- **downstairs_graph()**
- **edge_permutations()**
- **divisor()**
- **divisor_group()**
- **divisor_of_function()**
- **divisor_to_divisor_list()**
- **downstairs_edges()**
- **downstairs_graph()**
homotopy_continuation()  (sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface method), 136

integer_matrix_relations()  (in module sage.schemes.riemann_surfaces.riemann_surface), 150

IntegralAffineCurve  (class in sage.schemes.curves.affine_curve), 36

IntegralAffineCurve_finite_field  (class in sage.schemes.curves.affine_curve), 40

IntegralAffineCurveClosedPoint  (class in sage.schemes.curves.closed_point), 86

IntegralAffineCurvePoint  (class in sage.schemes.curves.point), 79

IntegralAffinePlaneCurve  (class in sage.schemes.curves.affine_curve), 42

IntegralAffinePlaneCurve_finite_field  (class in sage.schemes.curves.affine_curve), 42

IntegralAffinePlaneCurvePoint  (class in sage.schemes.curves.point), 81

IntegralAffinePlaneCurvePoint_finite_field  (class in sage.schemes.curves.point), 81

IntegralCurveClosedPoint  (class in sage.schemes.curves.closed_point), 88

IntegralProjectiveCurve  (class in sage.schemes.curves.projective_curve), 46

IntegralProjectiveCurve_finite_field  (class in sage.schemes.curves.projective_curve), 48

IntegralProjectiveCurveClosedPoint  (class in sage.schemes.curves.closed_point), 90

IntegralProjectiveCurvePoint  (class in sage.schemes.curves.point), 81

IntegralProjectiveCurvePoint_finite_field  (class in sage.schemes.curves.point), 82

IntegralProjectivePlaneCurve  (class in sage.schemes.curves.projective_curve), 51

IntegralProjectivePlaneCurve_finite_field  (class in sage.schemes.curves.projective_curve), 51

IntegralProjectivePlaneCurvePoint  (class in sage.schemes.curves.projective_curve), 82

IntegralProjectivePlaneCurvePoint_finite_field  (class in sage.schemes.curves.projective_curve), 82

Jacobian()  (in module sage.schemes.jacobians.abstract_jacobian), 93

Jacobian_generic  (class in sage.schemes.jacobians.abstract_jacobian), 93

intersection_points()  (sage.schemes.curves.curve.Curve_generic method), 8

intersects_at()  (sage.schemes.curves.curve.Curve_generic method), 9

is_complete_intersection()  (sage.schemes.curves.projective_curve.IntegralProjectiveCurve_finite_field method), 49

L_polynomial()  (sage.schemes.curves.projective_curve.IntegralProjectiveCurve_finite_field method), 49

index 159
Curves, Release 10.1

local_coordinates() (sage.schemes.curves.projective_curve.ProjectivePlaneCurve method), 63

local_obstructions() (sage.schemes.curves.projective_curve.ProjectivePlaneCurve method), 115

local_obstructions() (sage.schemes.curves.affine_curve.AffinePlaneCurve method), 29

local_obstructions() (sage.schemes.plane_conics.con_field.ProjectiveConic_field method), 104

local_obstructions() (sage.schemes.plane_conics.con_number_field.ProjectiveConic_number_field method), 115

local_obstructions() (sage.schemes.plane_conics.con_rational_field.ProjectiveConic_rational_field method), 117

make_zw_interpolator() (sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface method), 136

matrix() (sage.schemes.plane_conics.con_field.ProjectiveConic_field method), 104

matrix() (sage.schemes.curves.affine_curve.AffinePlaneCurve method), 29

matrix() (sage.schemes.curves.affine_curve.IntegralAffineCurve method), 37

matrix() (sage.schemes.curves.riemann_surfaces.riemann_surface.RiemannSurface method), 138

period_matrix() (sage.schemes.curves.riemann_surfaces.riemann_surface.RiemannSurface method), 138

period_matrix() (sage.schemes.curves.riemann_surfaces.riemann_surface.RiemannSurfaceSum method), 148

period_matrix() (sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface method), 138

period_matrix() (sage.schemes.plane_conics.con_field.ProjectiveConic_field method), 117

period_matrix() (sage.schemes.plane_conics.con_rational_field.ProjectiveConic_rational_field method), 117

module
sage.schemes.curves.affine_curve, 13
sage.schemes.curves.closed_point, 85
sage.schemes.curves.constructor, 1
sage.schemes.curves.curve, 5
sage.schemes.curves.point, 77
sage.schemes.curves.projective_curve, 43
sage.schemes.jacobians.abstract_jacobian, 93
sage.schemes.plane_conics.con_field, 97
sage.schemes.plane_conics.con_finite_field, 119
sage.schemes.plane_conics.con_number_field, 115
sage.schemes.plane_conics.con_rational_field, 120
sage.schemes.plane_conics.con_rational_function_field, 125
sage.schemes.plane_conics.constructor, 95
sage.schemes.plane_quartics.quartic_constructor, 125
sage.schemes.plane_quartics.quartic_generic, 125
sage.schemes.riemann_surfaces.riemann_surface, 127

monodromy_group() (sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface method), 138

multiplicity() (sage.schemes.curves.affine_curve.AffinePlaneCurve method), 29

multiplicity() (sage.schemes.curves.affine_curve.IntegralAffineCurve method), 37

multiplicity() (sage.schemes.curves.projective_curve.IntegralProjectiveCurvePoint field method), 50

multiplicity() (sage.schemes.curves.point.ProjectivePlaneCurvePoint field method), 84

multiplicity() (sage.schemes.curves.point.IntegralAffineCurvePoint method), 80

multiplicity() (sage.schemes.curves.point.IntegralProjectiveCurvePoint method), 82

numerator_converse() (in module sage.schemes.riemann_surfaces.riemann_surface), 150

ordinary_model() (sage.schemes.curves.projective_curve.ProjectivePlaneCurve method), 64

parametric_representation() (sage.schemes.curves.affine_curve.IntegralAffineCurve method), 37

parametric_representation() (sage.schemes.plane_conics.con_field.ProjectiveConic_field method), 105

parametric_representation() (sage.schemes.plane_conics.con_rational_field.ProjectiveConic_rational_field method), 117

period_matrix() (sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface method), 138

period_matrix() (sage.schemes.riemann_surfaces.riemann_surface.RiemannSurfaceSum method), 148

place() (sage.schemes.curves.closed_point.IntegralCurveClosedPoint method), 89

place() (sage.schemes.curves.affine_curve.IntegralAffineCurve method), 38

place() (sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface method), 89

place() (sage.schemes.curves.projective_curve.IntegralProjectivePoint method), 81

place() (sage.schemes.curves.affine_curve.IntegralAffineCurve method), 38

place() (sage.schemes.curves.affine_curve.IntegralAffineCurve method), 47

place() (sage.schemes.curves.projective_curve.IntegralProjectiveCurve method), 47

place() (sage.schemes.curves.projective_curve.IntegralProjectiveCurve method), 50

place_to_closed_point() (sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface method), 89

place_to_closed_point() (sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface method), 89

place_to_closed_point() (sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface method), 89

places() (sage.schemes.curves.affine_curve.IntegralAffineCurve method), 41

places() (sage.schemes.curves.affine_curve.IntegralAffineCurve method), 41

places() (sage.schemes.curves.projective_curve.IntegralProjectiveCurve method), 47

places() (sage.schemes.curves.projective_curve.IntegralProjectiveCurve method), 50

places() (sage.schemes.curves.projective_curve.IntegralProjectiveCurve method), 50

places() (sage.schemes.curves.affine_curve.AffinePlaneCurve method), 29

places() (sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface method), 89

places() (sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface method), 82

places() (sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface method), 89

places() (sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface method), 139

Index
Curves, Release 10.1

riemann_roch_basis() (sage.schemes.curves.affine_curve.AffinePlaneCurve_finite_field method), 35
riemann_roch_basis() (sage.schemes.curves.projective_curve.ProjectivePlaneCurve_finite_field method), 75
riemann_surface() (sage.schemes.curves.affine_curve.AffinePlaneCurve_field method), 34
riemann_surface() (sage.schemes.curves.projective_curve.ProjectivePlaneCurve_field method), 71
RiemannSurface (class in sage.schemes.riemann_surfaces.riemann_surface), 129
RiemannSurfaceSum (class in sage.schemes.riemann_surfaces.riemann_surface), 147
rigorous_line_integral() (sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface method), 141
rosati_involution() (sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface method), 142

S
sage.schemes.curves.affine_curve module, 13
sage.schemes.curves.closed_point module, 85
sage.schemes.curves.constructor module, 1
sage.schemes.curves.curve module, 5
sage.schemes.curves.point module, 77
sage.schemes.curves.projective_curve module, 43
sage.schemes.jacobians.abstract_jacobian module, 93
sage.schemes.plane_conics.con_field module, 97
sage.schemes.plane_conics.con_finite_field module, 119
sage.schemes.plane_conics.con_number_field module, 112
sage.schemes.plane_conics.con_rational_field module, 115
sage.schemes.plane_conics.con_rational_function_field module, 120
sage.schemes.plane_conics.constructor module, 95
sage.schemes.plane_quartics.quartic_constructor module, 125
sage.schemes.plane_quartics.quartic_generic module, 125

U
union() (sage.schemes.curves.curve.Curve_generic method), 12

162 Index
upper_triangular_matrix()  
(sage.schemes.plane_conics.con_field.ProjectiveConic_field  
method), 111

upstairs_edges() (sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface  
method), 146

upstairs_graph() (sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface  
method), 146

V

variable_names() (sage.schemes.plane_conics.con_field.ProjectiveConic_field  
method), 112

voronoi_ghost() (in module  
sage.schemes.riemann_surfaces.riemann_surface),  
152

W

w_values() (sage.schemes.riemann_surfaces.riemann_surface.RiemannSurface  
method), 147