Schemes

Release 10.2

The Sage Development Team

Dec 06, 2023

CONTENTS

CHAPTER

ONE

SCHEME IMPLEMENTATION OVERVIEW

Various parts of schemes were implemented by different authors. This document aims to give an overview of the different classes of schemes working together coherently.

1.1 Generic

- **Scheme:** A scheme whose datatype might not be defined in terms of algebraic equations: e.g. the Jacobian of a curve may be represented by means of a Scheme.
- **AlgebraicScheme:** A scheme defined by means of polynomial equations, which may be reducible or defined over a ring other than a field. In particular, the defining ideal need not be a radical ideal, and an algebraic scheme may be defined over $\mathrm{Spec}(R)$.
- **AmbientSpaces:** Most effective models of algebraic scheme will be defined not by generic gluings, but by embeddings in some fixed ambient space.

1.2 Ambients

- **AffineSpace:** Affine spaces and their affine subschemes form the most important universal objects from which algebraic schemes are built. The affine spaces form universal objects in the sense that a morphism is uniquely determined by the images of its coordinate functions and any such images determine a well-defined morphism.

  By default affine spaces will embed in some ordinary projective space, unless it is created as an affine patch of another object.
- **ProjectiveSpace:** Projective spaces are the most natural ambient spaces for most projective objects. They are locally universal objects.
- **ProjectiveSpace_ordinary (not implemented):** The ordinary projective spaces have the standard weights $[1, .., 1]$ on their coefficients.
- **ProjectiveSpace_weighted (not implemented):** A special subtype for non-standard weights.
- **ToricVariety:** Toric varieties are (partial) compactifications of algebraic tori $(\mathbf{C}^*)^n$ compatible with torus action. Affine and projective spaces are examples of toric varieties, but it is not envisioned that these special cases should inherit from *ToricVariety*.

1.3 Subschemes

- **AlgebraicScheme_subscheme_affine:** An algebraic scheme defined by means of an embedding in a fixed ambient affine space.
- **AlgebraicScheme_subscheme_projective:** An algebraic scheme defined by means of an embedding in a fixed ambient projective space.
- **QuasiAffineScheme (not yet implemented):** An open subset $U = X \setminus Z$ of a closed subset X of affine space; note that this is mathematically a quasi-projective scheme, but its ambient space is an affine space and its points are represented by affine rather than projective points.

Note: AlgebraicScheme_quasi is implemented, as a base class for this.

- **QuasiProjectiveScheme (not yet implemented):** An open subset of a closed subset of projective space; this datatype stores the defining polynomial, polynomials, or ideal defining the projective closure X plus the closed subscheme Z of X whose complement $U = X \setminus Z$ is the quasi-projective scheme.

Note: The quasi-affine and quasi-projective datatype lets one create schemes like the multiplicative group scheme $\mathbb{G}_m = \mathbb{A}^1 \setminus \{(0)\}$ and the non-affine scheme $\mathbb{A}^2 \setminus \{(0,0)\}$. The latter is not affine and is not of the form $\mathrm{Spec}(R)$.

1.4 Point sets

- **PointSets and points over a ring (to do):** For algebraic schemes X/S and T/S over S, one can form the point set $X(T)$ of morphisms from $T \to X$ over S.

  A projective space object in the category of schemes is a locally free object – the images of the generator functions *locally* determine a point. Over a field, one can choose one of the standard affine patches by the condition that a coordinate function $X_i \neq 0$.

```
sage: PP.<X,Y,Z> = ProjectiveSpace(2, QQ)
sage: PP
Projective Space of dimension 2 over Rational Field
sage: PP(QQ)
Set of rational points of Projective Space
of dimension 2 over Rational Field
sage: PP(QQ)([-2, 3, 5])
(-2/5 : 3/5 : 1)
```

  Over a ring, this is not true anymore. For example, even over an integral domain which is not a PID, there may be no *single* affine patch which covers a point.

```
sage: R.<x> = ZZ[]
sage: S.<t> = R.quo(x^2 + 5)
sage: P.<X,Y,Z> = ProjectiveSpace(2, S)
sage: P(S)
Set of rational points of Projective Space of dimension 2 over
Univariate Quotient Polynomial Ring in t over Integer Ring with
modulus x^2 + 5
```

In order to represent the projective point $(2 : 1 + t) = (1 - t : 3)$ we note that the first representative is not well-defined at the prime $p = (2, 1 + t)$ and the second element is not well-defined at the prime $q = (1 - t, 3)$, but that $p + q = (1)$, so globally the pair of coordinate representatives is well-defined.

```
sage: P([2, 1 + t])
(2 : t + 1 : 1)
```

In fact, we need a test `R.ideal([2, 1 + t]) == R.ideal([1])` in order to make this meaningful.

1.5 Berkovich Analytic Spaces

- **Berkovich Analytic Space (not yet implemented)** The construction of analytic spaces from schemes due to Berkovich. Any Berkovich space should inherit from `Berkovich`
- **Berkovich Analytic Space over Cp** A special case of the general Berkovich analytic space construction. Affine Berkovich space over $\mathbf{C}_p$ is the set of seminorms on the polynomial ring $\mathbf{C}_p[x]$, while projective Berkovich space over $\mathbf{C}_p$ is the one-point compactification of affine Berkovich space $\mathbf{C}_p$. Points are represented using the classification (due to Berkovich) of a corresponding decreasing sequence of disks in $\mathbf{C}_p$.

AUTHORS:

- David Kohel, William Stein (2006-01-03): initial version
- Andrey Novoseltsev (2010-09-24): updated due to addition of toric varieties

CHAPTER

TWO

SCHEMES

AUTHORS:

- William Stein, David Kohel, Kiran Kedlaya (2008): added zeta_series
- Volker Braun (2011-08-11): documenting, improving, refactoring.

class sage.schemes.generic.scheme.**AffineScheme**(*R*, *S=None*, *category=None*)

Bases: UniqueRepresentation, *Scheme*

Class for general affine schemes.

See also:

For affine spaces over a base ring and subschemes thereof, see `sage.schemes.generic.algebraic_scheme.AffineSpace`.

Element

alias of *SchemeTopologicalPoint_prime_ideal*

base_extend(*R*)

Extend the base ring/scheme.

INPUT:

- R – an affine scheme or a commutative ring

EXAMPLES:

```
sage: Spec_ZZ = Spec(ZZ);  Spec_ZZ
Spectrum of Integer Ring
sage: Spec_ZZ.base_extend(QQ)
Spectrum of Rational Field

sage: Spec(ZZ['x']).base_extend(Spec(QQ))
Spectrum of Univariate Polynomial Ring in x over Rational Field
```

coordinate_ring()

Return the underlying ring of this scheme.

OUTPUT:

A commutative ring.

EXAMPLES:

```
sage: Spec(QQ).coordinate_ring()
Rational Field
sage: Spec(PolynomialRing(QQ, 3, 'x')).coordinate_ring()
Multivariate Polynomial Ring in x0, x1, x2 over Rational Field
```

dimension()

Return the absolute dimension of this scheme.

OUTPUT:

Integer.

EXAMPLES:

```
sage: S = Spec(ZZ)
sage: S.dimension_absolute()
1
sage: S.dimension()
1
```

dimension_absolute()

Return the absolute dimension of this scheme.

OUTPUT:

Integer.

EXAMPLES:

```
sage: S = Spec(ZZ)
sage: S.dimension_absolute()
1
sage: S.dimension()
1
```

dimension_relative()

Return the relative dimension of this scheme over its base.

OUTPUT:

Integer.

EXAMPLES:

```
sage: S = Spec(ZZ)
sage: S.dimension_relative()
0
```

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

Return the scheme morphism from `self` to `Y` defined by `x`.

INPUT:

- `x` – anything that determines a scheme morphism; if `x` is a scheme, try to determine a natural map to `x`
- `Y` – the codomain scheme (optional); if `Y` is not given, try to determine `Y` from context
- `check` – boolean (optional, default: `True`); whether to check the defining data for consistency

OUTPUT:

The scheme morphism from `self` to `Y` defined by `x`.

EXAMPLES:

We construct the inclusion from $\mathrm{Spec}(\mathbf{Q})$ into $\mathrm{Spec}(\mathbf{Z})$ induced by the inclusion from $\mathbf{Z}$ into $\mathbf{Q}$:

```
sage: X = Spec(QQ)
sage: X.hom(ZZ.hom(QQ))
Affine Scheme morphism:
  From: Spectrum of Rational Field
  To:   Spectrum of Integer Ring
  Defn: Natural morphism:
          From: Integer Ring
          To:   Rational Field
```

is_noetherian()

Return `True` if `self` is Noetherian, `False` otherwise.

EXAMPLES:

```
sage: Spec(ZZ).is_noetherian()
True
```

class sage.schemes.generic.scheme.**Scheme**(*X=None*, *category=None*)

Bases: `Parent`

The base class for all schemes.

INPUT:

- `X` – a scheme, scheme morphism, commutative ring, commutative ring morphism, or `None` (optional). Determines the base scheme. If a commutative ring is passed, the spectrum of the ring will be used as base.
- `category` – the category (optional). Will be automatically constructed by default.

EXAMPLES:

```
sage: from sage.schemes.generic.scheme import Scheme
sage: Scheme(ZZ)
<sage.schemes.generic.scheme.Scheme_with_category object at ...>
```

A scheme is in the category of all schemes over its base:

```
sage: ProjectiveSpace(4, QQ).category()
Category of schemes over Rational Field
```

There is a special and unique $Spec(\mathbf{Z})$ that is the default base scheme:

```
sage: Spec(ZZ).base_scheme() is Spec(QQ).base_scheme()
True
```

base_extend(*Y*)

Extend the base of the scheme.

Derived classes must override this method.

EXAMPLES:

```
sage: from sage.schemes.generic.scheme import Scheme
sage: X = Scheme(ZZ)
sage: X.base_scheme()
Spectrum of Integer Ring
sage: X.base_extend(QQ)
Traceback (most recent call last):
...
NotImplementedError
```

base_morphism()

Return the structure morphism from `self` to its base scheme.

OUTPUT:

A scheme morphism.

EXAMPLES:

```
sage: A = AffineSpace(4, QQ)
sage: A.base_morphism()
Scheme morphism:
  From: Affine Space of dimension 4 over Rational Field
  To:   Spectrum of Rational Field
  Defn: Structure map

sage: X = Spec(QQ)
sage: X.base_morphism()
Scheme morphism:
  From: Spectrum of Rational Field
  To:   Spectrum of Integer Ring
  Defn: Structure map
```

base_ring()

Return the base ring of the scheme self.

OUTPUT:

A commutative ring.

EXAMPLES:

```
sage: A = AffineSpace(4, QQ)
sage: A.base_ring()
Rational Field

sage: X = Spec(QQ)
sage: X.base_ring()
Integer Ring
```

base_scheme()

Return the base scheme.

OUTPUT:

A scheme.

EXAMPLES:

```
sage: A = AffineSpace(4, QQ)
sage: A.base_scheme()
Spectrum of Rational Field

sage: X = Spec(QQ)
sage: X.base_scheme()
Spectrum of Integer Ring
```

coordinate_ring()

Return the coordinate ring.

OUTPUT:

The global coordinate ring of this scheme, if defined. Otherwise raise a `ValueError`.

EXAMPLES:

```
sage: R.<x, y> = QQ[]
sage: I = (x^2 - y^2)*R
sage: X = Spec(R.quotient(I))
sage: X.coordinate_ring()
Quotient of Multivariate Polynomial Ring in x, y over Rational Field
 by the ideal (x^2 - y^2)
```

count_points(n)

Count points over finite fields.

INPUT:

- `n` – integer.

OUTPUT:

An integer. The number of points over $\mathbf{F}_q, \ldots, \mathbf{F}_{q^n}$ on a scheme over a finite field $\mathbf{F}_q$.

EXAMPLES:

```
sage: # needs sage.schemes
sage: P.<x> = PolynomialRing(GF(3))
sage: C = HyperellipticCurve(x^3 + x^2 + 1)
sage: C.count_points(4)
[6, 12, 18, 96]
sage: C.base_extend(GF(9,'a')).count_points(2)                          #␣
↪needs sage.rings.finite_rings
[12, 96]
```

```
sage: P.<x,y,z> = ProjectiveSpace(GF(4, 't'), 2)                        #␣
↪needs sage.rings.finite_rings
sage: X = P.subscheme([y^2*z - x^3 - z^3])                              #␣
↪needs sage.rings.finite_rings
sage: X.count_points(2)                                                 #␣
↪needs sage.libs.singular sage.rings.finite_rings
[5, 17]
```

dimension()

Return the absolute dimension of this scheme.

OUTPUT:

Integer.

EXAMPLES:

```
sage: R.<x, y> = QQ[]
sage: I = (x^2 - y^2)*R
sage: X = Spec(R.quotient(I))
sage: X.dimension_absolute()
Traceback (most recent call last):
...
NotImplementedError
sage: X.dimension()
Traceback (most recent call last):
...
NotImplementedError
```

dimension_absolute()

Return the absolute dimension of this scheme.

OUTPUT:

Integer.

EXAMPLES:

```
sage: R.<x, y> = QQ[]
sage: I = (x^2 - y^2)*R
sage: X = Spec(R.quotient(I))
sage: X.dimension_absolute()
Traceback (most recent call last):
...
NotImplementedError
sage: X.dimension()
Traceback (most recent call last):
...
NotImplementedError
```

dimension_relative()

Return the relative dimension of this scheme over its base.

OUTPUT:

Integer.

EXAMPLES:

```
sage: R.<x, y> = QQ[]
sage: I = (x^2 - y^2)*R
sage: X = Spec(R.quotient(I))
sage: X.dimension_relative()
Traceback (most recent call last):
...
NotImplementedError
```

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

Return the scheme morphism from `self` to `Y` defined by `x`.

INPUT:

- x – anything that determines a scheme morphism; if x is a scheme, try to determine a natural map to x
- Y – the codomain scheme (optional); if Y is not given, try to determine Y from context
- check – boolean (optional, default: True); whether to check the defining data for consistency

OUTPUT:

The scheme morphism from self to Y defined by x.

EXAMPLES:

```
sage: P = ProjectiveSpace(ZZ, 3)
sage: P.hom(Spec(ZZ))
Scheme morphism:
  From: Projective Space of dimension 3 over Integer Ring
  To:   Spectrum of Integer Ring
  Defn: Structure map
```

identity_morphism()

Return the identity morphism.

OUTPUT:

The identity morphism of the scheme self.

EXAMPLES:

```
sage: X = Spec(QQ)
sage: X.identity_morphism()
Scheme endomorphism of Spectrum of Rational Field
  Defn: Identity map
```

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

Create a point.

INPUT:

- v – anything that defines a point
- check – boolean (optional, default: True); whether to check the defining data for consistency

OUTPUT:

A point of the scheme.

EXAMPLES:

```
sage: A2 = AffineSpace(QQ, 2)
sage: A2.point([4, 5])
(4, 5)

sage: R.<t> = PolynomialRing(QQ)
sage: E = EllipticCurve([t + 1, t, t, 0, 0])                                #␣
↪needs sage.schemes
sage: E.point([0, 0])                                                       #␣
↪needs sage.schemes
(0 : 0 : 1)
```

point_homset(*S=None*)

Return the set of S-valued points of this scheme.

INPUT:

- S – a commutative ring.

OUTPUT:

The set of morphisms $Spec(S)oX$.

EXAMPLES:

```
sage: P = ProjectiveSpace(ZZ, 3)
sage: P.point_homset(ZZ)
Set of rational points of Projective Space of dimension 3 over Integer Ring
sage: P.point_homset(QQ)
Set of rational points of Projective Space of dimension 3 over Rational Field
sage: P.point_homset(GF(11))
Set of rational points of Projective Space of dimension 3 over
 Finite Field of size 11
```

point_set(*S=None*)

Return the set of S-valued points of this scheme.

INPUT:

- S – a commutative ring.

OUTPUT:

The set of morphisms $Spec(S)oX$.

EXAMPLES:

```
sage: P = ProjectiveSpace(ZZ, 3)
sage: P.point_homset(ZZ)
Set of rational points of Projective Space of dimension 3 over Integer Ring
sage: P.point_homset(QQ)
Set of rational points of Projective Space of dimension 3 over Rational Field
sage: P.point_homset(GF(11))
Set of rational points of Projective Space of dimension 3 over
 Finite Field of size 11
```

structure_morphism()

Return the structure morphism from `self` to its base scheme.

OUTPUT:

A scheme morphism.

EXAMPLES:

```
sage: A = AffineSpace(4, QQ)
sage: A.base_morphism()
Scheme morphism:
  From: Affine Space of dimension 4 over Rational Field
  To:   Spectrum of Rational Field
  Defn: Structure map
```

(continues on next page)

(continued from previous page)

```
sage: X = Spec(QQ)
sage: X.base_morphism()
Scheme morphism:
  From: Spectrum of Rational Field
  To:   Spectrum of Integer Ring
  Defn: Structure map
```

union(*X*)

Return the disjoint union of the schemes `self` and `X`.

EXAMPLES:

```
sage: S = Spec(QQ)
sage: X = AffineSpace(1, QQ)
sage: S.union(X)
Traceback (most recent call last):
...
NotImplementedError
```

zeta_function()

Compute the zeta function of a generic scheme.

Derived classes should override this method.

OUTPUT: rational function in one variable.

EXAMPLES:

```
sage: P.<x,y,z> = ProjectiveSpace(GF(4, 't'), 2)                    #␣
↪needs sage.rings.finite_rings
sage: X = P.subscheme([y^2*z - x^3 - z^3])                          #␣
↪needs sage.rings.finite_rings
sage: X.zeta_function()                                             #␣
↪needs sage.rings.finite_rings
Traceback (most recent call last):
...
NotImplementedError
```

zeta_series(*n*, *t*)

Return the zeta series.

Compute a power series approximation to the zeta function of a scheme over a finite field.

INPUT:

- `n` – the number of terms of the power series to compute
- `t` – the variable which the series should be returned

OUTPUT:

A power series approximating the zeta function of `self`

EXAMPLES:

```
sage: P.<x> = PolynomialRing(GF(3))
sage: C = HyperellipticCurve(x^3 + x^2 + 1)                                    #␣
↪needs sage.schemes
sage: R.<t> = PowerSeriesRing(Integers())
sage: C.zeta_series(4, t)                                                      #␣
↪needs sage.schemes
1 + 6*t + 24*t^2 + 78*t^3 + 240*t^4 + O(t^5)
sage: (1+2*t+3*t^2)/(1-t)/(1-3*t) + O(t^5)
1 + 6*t + 24*t^2 + 78*t^3 + 240*t^4 + O(t^5)
```

If the scheme has a method `zeta_function`, this is used to provide the required approximation. Otherwise this function depends on `count_points`, which is only defined for prime order fields for general schemes. Nonetheless, since github issue #15108 and github issue #15148, it supports hyperelliptic curves over non-prime fields:

```
sage: C.base_extend(GF(9, 'a')).zeta_series(4, t)                              #␣
↪needs sage.rings.finite_rings sage.schemes
1 + 12*t + 120*t^2 + 1092*t^3 + 9840*t^4 + O(t^5)
```

```
sage: P.<x,y,z> = ProjectiveSpace(GF(4, 't'), 2)                               #␣
↪needs sage.rings.finite_rings
sage: X = P.subscheme([y^2*z - x^3 - z^3])                                     #␣
↪needs sage.rings.finite_rings

sage: R.<t> = PowerSeriesRing(Integers())
sage: X.zeta_series(2, t)                                                      #␣
↪needs sage.libs.singular sage.rings.finite_rings
1 + 5*t + 21*t^2 + O(t^3)
```

sage.schemes.generic.scheme.**is_AffineScheme**(x)

Return True if x is an affine scheme.

EXAMPLES:

```
sage: from sage.schemes.generic.scheme import is_AffineScheme
sage: is_AffineScheme(5)
False
sage: E = Spec(QQ)
sage: is_AffineScheme(E)
True
```

sage.schemes.generic.scheme.**is_Scheme**(x)

Test whether `x` is a scheme.

INPUT:

- `x` – anything.

OUTPUT:

Boolean. Whether `x` derives from *Scheme*.

EXAMPLES:

```
sage: from sage.schemes.generic.scheme import is_Scheme
sage: is_Scheme(5)
False
sage: X = Spec(QQ)
sage: is_Scheme(X)
True
```

CHAPTER

THREE

THE SPEC FUNCTOR

AUTHORS:

- William Stein (2006): initial implementation
- Peter Bruin (2014): rewrite Spec as a functor

`sage.schemes.generic.spec.`**`Spec`**(*R*, *S=None*)

Apply the Spec functor to R.

INPUT:

- `R` – either a commutative ring or a ring homomorphism
- `S` – a commutative ring (optional), the base ring

OUTPUT:

- `AffineScheme` – the affine scheme $\mathrm{Spec}(R)$

EXAMPLES:

```
sage: Spec(QQ)
Spectrum of Rational Field
sage: Spec(PolynomialRing(QQ, 'x'))
Spectrum of Univariate Polynomial Ring in x over Rational Field
sage: Spec(PolynomialRing(QQ, 'x', 3))
Spectrum of Multivariate Polynomial Ring in x0, x1, x2 over Rational Field
sage: X = Spec(PolynomialRing(GF(49,'a'), 3, 'x')); X                          #␣
↪needs sage.rings.finite_rings
Spectrum of Multivariate Polynomial Ring in x0, x1, x2
 over Finite Field in a of size 7^2
sage: TestSuite(X).run()                                                       #␣
↪needs sage.rings.finite_rings
```

Applying `Spec` twice to the same ring gives identical output (see github issue #17008):

```
sage: A = Spec(ZZ); B = Spec(ZZ)
sage: A is B
True
```

A `TypeError` is raised if the input is not a commutative ring:

```
sage: Spec(5)
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
TypeError: x (=5) is not in Category of commutative rings
sage: Spec(FreeAlgebra(QQ, 2, 'x'))                                    #␣
↪needs sage.combinat sage.modules
Traceback (most recent call last):
...
TypeError: x (=Free Algebra on 2 generators (x0, x1) over Rational Field)
is not in Category of commutative rings
```

class sage.schemes.generic.spec.**SpecFunctor**(*base_ring=None*)

Bases: Functor, UniqueRepresentation

The Spec functor.

CHAPTER

FOUR

SCHEME OBTAINED BY GLUING TWO OTHER SCHEMES

class sage.schemes.generic.glue.**GluedScheme**(*f*, *g*, *check=True*)

Bases: *Scheme*

INPUT:

- `f` - open immersion from a scheme U to a scheme X
- `g` - open immersion from U to a scheme Y

OUTPUT: The scheme obtained by gluing X and Y along the open set U.

Note: Checking that f and g are open immersions is not implemented.

---

gluing_maps()

CHAPTER

FIVE

POINTS ON SCHEMES

class sage.schemes.generic.point.**SchemePoint**(*S*, *parent=None*)

Bases: Element

Base class for points on a scheme, either topological or defined by a morphism.

scheme()

Return the scheme on which self is a point.

EXAMPLES:

```
sage: from sage.schemes.generic.point import SchemePoint
sage: S = Spec(ZZ)
sage: P = SchemePoint(S)
sage: P.scheme()
Spectrum of Integer Ring
```

class sage.schemes.generic.point.**SchemeRationalPoint**(*f*)

Bases: *SchemePoint*

INPUT:

- f - a morphism of schemes

morphism()

class sage.schemes.generic.point.**SchemeTopologicalPoint**(*S*)

Bases: *SchemePoint*

Base class for topological points on schemes.

class sage.schemes.generic.point.**SchemeTopologicalPoint_affine_open**(*u*, *x*)

Bases: *SchemeTopologicalPoint*

INPUT:

- u – morphism with domain an affine scheme U
- x – topological point on U

affine_open()

Return the affine open subset U.

embedding_of_affine_open()

Return the embedding from the affine open subset U into this scheme.

point_on_affine()

Return the scheme point on the affine open U.

class sage.schemes.generic.point.**SchemeTopologicalPoint_prime_ideal**(*S*, *P*, *check=False*)

Bases: *SchemeTopologicalPoint*

INPUT:

- S – an affine scheme
- P – a prime ideal of the coordinate ring of S, or anything that can be converted into such an ideal

prime_ideal()

Return the prime ideal that defines this scheme point.

EXAMPLES:

```
sage: from sage.schemes.generic.point import SchemeTopologicalPoint_prime_ideal
sage: P2.<x, y, z> = ProjectiveSpace(2, QQ)
sage: pt = SchemeTopologicalPoint_prime_ideal(P2, y*z - x^2)
sage: pt.prime_ideal()
Ideal (-x^2 + y*z) of Multivariate Polynomial Ring in x, y, z over Rational␣
↪Field
```

sage.schemes.generic.point.**is_SchemeRationalPoint**(*x*)

sage.schemes.generic.point.**is_SchemeTopologicalPoint**(*x*)

CHAPTER

SIX

AMBIENT SPACES

class sage.schemes.generic.ambient_space.**AmbientSpace**(*n*, *R=Integer Ring*)

Bases: *Scheme*

Base class for ambient spaces over a ring.

INPUT:

- n - dimension
- R - ring

ambient_space()

Return the ambient space of the scheme self, in this case self itself.

EXAMPLES:

```
sage: P = ProjectiveSpace(4, ZZ)
sage: P.ambient_space() is P
True

sage: A = AffineSpace(2, GF(3))
sage: A.ambient_space()
Affine Space of dimension 2 over Finite Field of size 3
```

base_extend(*R*)

Return the natural extension of self over R.

INPUT:

- R – a commutative ring, such that there is a natural map from the base ring of self to R.

OUTPUT:

- an ambient space over R of the same structure as self.

Note: A ValueError is raised if there is no such natural map. If you need to drop this condition, use self.change_ring(R).

EXAMPLES:

```
sage: P.<x, y, z> = ProjectiveSpace(2, ZZ)
sage: PQ = P.base_extend(QQ); PQ
Projective Space of dimension 2 over Rational Field
sage: PQ.base_extend(GF(5))
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: no natural map from the base ring (=Rational Field)
to R (=Finite Field of size 5)!
```

change_ring(*R*)

Return an ambient space over ring R and otherwise the same as self.

INPUT:

- R – commutative ring

OUTPUT:

- ambient space over R

Note: There is no need to have any relation between R and the base ring of self, if you want to have such a relation, use `self.base_extend(R)` instead.

---

defining_polynomials()

Return the defining polynomials of the scheme self. Since self is an ambient space, this is an empty list.

EXAMPLES:

```
sage: ProjectiveSpace(2, QQ).defining_polynomials()
()
sage: AffineSpace(0, ZZ).defining_polynomials()
()
```

dimension()

Return the absolute dimension of this scheme.

EXAMPLES:

```
sage: A2Q = AffineSpace(2, QQ)
sage: A2Q.dimension_absolute()
2
sage: A2Q.dimension()
2
sage: A2Z = AffineSpace(2, ZZ)
sage: A2Z.dimension_absolute()
3
sage: A2Z.dimension()
3
```

dimension_absolute()

Return the absolute dimension of this scheme.

EXAMPLES:

```
sage: A2Q = AffineSpace(2, QQ)
sage: A2Q.dimension_absolute()
2
sage: A2Q.dimension()
```

(continues on next page)

(continued from previous page)

```
2
sage: A2Z = AffineSpace(2, ZZ)
sage: A2Z.dimension_absolute()
3
sage: A2Z.dimension()
3
```

dimension_relative()

Return the relative dimension of this scheme over its base.

EXAMPLES:

```
sage: A2Q = AffineSpace(2, QQ)
sage: A2Q.dimension_relative()
2
sage: A2Z = AffineSpace(2, ZZ)
sage: A2Z.dimension_relative()
2
```

gen(*n=0*)

Return the n-th generator of the coordinate ring of the scheme self.

EXAMPLES:

```
sage: P.<x, y, z> = ProjectiveSpace(2, ZZ)
sage: P.gen(1)
y
```

gens()

Return the generators of the coordinate ring of the scheme self.

EXAMPLES:

```
sage: AffineSpace(0, QQ).gens()
()

sage: P.<x, y, z> = ProjectiveSpace(2, GF(5))
sage: P.gens()
(x, y, z)
```

identity_morphism()

Return the identity morphism.

OUTPUT: the identity morphism of the scheme `self`

EXAMPLES:

```
sage: A = AffineSpace(2, GF(3))
sage: A.identity_morphism()
Scheme endomorphism of Affine Space of dimension 2 over Finite Field of size 3
  Defn: Identity map

sage: P = ProjectiveSpace(3, ZZ)
sage: P.identity_morphism()
```

(continues on next page)

(continued from previous page)

```
Scheme endomorphism of Projective Space of dimension 3 over Integer Ring
  Defn: Identity map
```

is_projective()

Return whether this ambient space is projective n-space.

EXAMPLES:

```
sage: AffineSpace(3, QQ).is_projective()
False
sage: ProjectiveSpace(3, QQ).is_projective()
True
```

ngens()

Return the number of generators of the coordinate ring of the scheme self.

EXAMPLES:

```
sage: AffineSpace(0, QQ).ngens()
0

sage: ProjectiveSpace(50, ZZ).ngens()
51
```

sage.schemes.generic.ambient_space.**is_AmbientSpace**(x)

Return True if x is an ambient space.

EXAMPLES:

```
sage: from sage.schemes.generic.ambient_space import is_AmbientSpace
sage: is_AmbientSpace(ProjectiveSpace(3, ZZ))
True
sage: is_AmbientSpace(AffineSpace(2, QQ))
True
sage: P.<x, y, z> = ProjectiveSpace(2, ZZ)
sage: is_AmbientSpace(P.subscheme([x + y + z]))
False
```

CHAPTER

SEVEN

ALGEBRAIC SCHEMES

An algebraic scheme is defined by a set of polynomials in some suitable affine or projective coordinates. Possible ambient spaces are

- Affine spaces (*AffineSpace*),
- Projective spaces (*ProjectiveSpace*), or
- Toric varieties (*ToricVariety*).

Note that while projective spaces are of course toric varieties themselves, they are implemented differently in Sage due to efficiency considerations. You still can create a projective space as a toric variety if you wish.

In the following, we call the corresponding subschemes affine algebraic schemes, projective algebraic schemes, or toric algebraic schemes. In the future other ambient spaces, perhaps by means of gluing relations, may be introduced.

Generally, polynomials $p_0, p_1, \ldots, p_n$ define an ideal $I = \langle p_0, p_1, \ldots, p_n \rangle$. In the projective and toric case, the polynomials (and, therefore, the ideal) must be homogeneous. The associated subscheme $V(I)$ of the ambient space is, roughly speaking, the subset of the ambient space on which all polynomials vanish simultaneously.

Warning: You should not construct algebraic scheme objects directly. Instead, use `.subscheme()` methods of ambient spaces. See below for examples.

EXAMPLES:

We first construct the ambient space, here the affine space $\mathbf{Q}^2$:

```
sage: A2 = AffineSpace(2, QQ, 'x, y')
sage: A2.coordinate_ring().inject_variables()
Defining x, y
```

Now we can write polynomial equations in the variables x and y. For example, one equation cuts out a curve (a one-dimensional subscheme):

```
sage: V = A2.subscheme([x^2 + y^2 - 1]); V
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x^2 + y^2 - 1
sage: V.dimension()                                                          #␣
↪needs sage.libs.singular
1
```

Here is a more complicated example in a projective space:

```
sage: P3 = ProjectiveSpace(3, QQ, 'x')
sage: P3.inject_variables()
Defining x0, x1, x2, x3
sage: Q = matrix([[x0, x1, x2], [x1, x2, x3]]).minors(2); Q                    #␣
↪needs sage.modules
[-x1^2 + x0*x2, -x1*x2 + x0*x3, -x2^2 + x1*x3]
sage: twisted_cubic = P3.subscheme(Q); twisted_cubic                           #␣
↪needs sage.libs.singular sage.modules
Closed subscheme of Projective Space of dimension 3 over Rational Field defined by:
  -x1^2 + x0*x2,
  -x1*x2 + x0*x3,
  -x2^2 + x1*x3
sage: twisted_cubic.dimension()                                                #␣
↪needs sage.libs.singular sage.modules
1
```

Note that there are 3 equations in the 3-dimensional ambient space, yet the subscheme is 1-dimensional. One can show that it is not possible to eliminate any of the equations, that is, the twisted cubic is **not** a complete intersection of two polynomial equations.

Let us look at one affine patch, for example the one where $x_0 = 1$

```
sage: patch = twisted_cubic.affine_patch(0); patch                             #␣
↪needs sage.libs.singular sage.modules
Closed subscheme of Affine Space of dimension 3 over Rational Field defined by:
  -x1^2 + x2,
  -x1*x2 + x3,
  -x2^2 + x1*x3
sage: patch.embedding_morphism()                                               #␣
↪needs sage.libs.singular sage.modules
Scheme morphism:
  From: Closed subscheme of Affine Space of dimension 3 over Rational Field
        defined by: -x1^2 + x2, -x1*x2 + x3, -x2^2 + x1*x3
  To:   Closed subscheme of Projective Space of dimension 3 over Rational Field
        defined by: x1^2 - x0*x2, x1*x2 - x0*x3, x2^2 - x1*x3
  Defn: Defined on coordinates by sending (x1, x2, x3) to (1 : x1 : x2 : x3)
```

AUTHORS:

- David Kohel, William Stein (2005): initial version
- Andrey Novoseltsev (2010-05-17): subschemes of toric varieties
- Volker Braun (2010-12-24): documentation of schemes and refactoring; added coordinate neighborhoods and is_smooth()
- Ben Hutz (2014): subschemes of Cartesian products of projective space
- Ben Hutz (2017): split subschemes types into respective folders

class sage.schemes.generic.algebraic_scheme.**AlgebraicScheme**(*A*)

Bases: *Scheme*

An algebraic scheme presented as a subscheme in an ambient space.

This is the base class for all algebraic schemes, that is, schemes defined by equations in affine, projective, or toric ambient spaces.

ambient_space()

Return the ambient space of this algebraic scheme.

EXAMPLES:

```
sage: A.<x, y> = AffineSpace(2, GF(5))
sage: S = A.subscheme([])
sage: S.ambient_space()
Affine Space of dimension 2 over Finite Field of size 5

sage: P.<x, y, z> = ProjectiveSpace(2, ZZ)
sage: S = P.subscheme([x - y, x - z])
sage: S.ambient_space() is P
True
```

coordinate_ring()

Return the coordinate ring of this algebraic scheme. The result is cached.

OUTPUT:

The coordinate ring. Usually a polynomial ring, or a quotient thereof.

EXAMPLES:

```
sage: P.<x, y, z> = ProjectiveSpace(2, ZZ)
sage: S = P.subscheme([x - y, x - z])
sage: S.coordinate_ring()
Quotient of Multivariate Polynomial Ring in x, y, z over Integer Ring
 by the ideal (x - y, x - z)
```

embedding_center()

Return the distinguished point, if there is any.

If the scheme Y was constructed as a neighbourhood of a point $p \in X$, then *embedding_morphism()* returns a local isomorphism $f : Y \to X$ around the preimage point $f^{-1}(p)$. The latter is returned by *embedding_center()*.

OUTPUT:

A point of `self`. Raises `AttributeError` if there is no distinguished point, depending on how `self` was constructed.

EXAMPLES:

```
sage: P3.<w,x,y,z> = ProjectiveSpace(QQ, 3)
sage: X = P3.subscheme( (w^2-x^2)*(y^2-z^2) )
sage: p = [1,-1,3,4]
sage: nbhd = X.neighborhood(p); nbhd
Closed subscheme of Affine Space of dimension 3 over Rational Field defined by:
  w^2*y^2 - x^2*y^2 + 6*w^2*y - 6*x^2*y + 2*w*y^2 +
  2*x*y^2 - 7*w^2 + 7*x^2 + 12*w*y + 12*x*y - 14*w - 14*x
sage: nbhd.embedding_center()
(0, 0, 0)
sage: nbhd.embedding_morphism()(nbhd.embedding_center())
(1/4 : -1/4 : 3/4 : 1)
sage: nbhd.embedding_morphism()
Scheme morphism:
```

(continues on next page)

(continued from previous page)

```
  From: Closed subscheme of Affine Space of dimension 3 over Rational Field
        defined by: w^2*y^2 - x^2*y^2 + 6*w^2*y - 6*x^2*y + 2*w*y^2 + 2*x*y^2
                    - 7*w^2 + 7*x^2 + 12*w*y + 12*x*y - 14*w - 14*x
  To:   Closed subscheme of Projective Space of dimension 3 over Rational Field
        defined by: w^2*y^2 - x^2*y^2 - w^2*z^2 + x^2*z^2
  Defn: Defined on coordinates by sending (w, x, y) to
        (w + 1 : x - 1 : y + 3 : 4)
```

embedding_morphism()

Return the default embedding morphism of `self`.

If the scheme Y was constructed as a neighbourhood of a point $p \in X$, then *embedding_morphism()* returns a local isomorphism $f : Y \to X$ around the preimage point $f^{-1}(p)$. The latter is returned by *embedding_center()*.

If the algebraic scheme Y was not constructed as a neighbourhood of a point, then the embedding in its *ambient_space()* is returned.

OUTPUT:

A scheme morphism whose `domain()` is `self`.

- By default, it is the tautological embedding into its own ambient space *ambient_space()*.
- If the algebraic scheme (which itself is a subscheme of an auxiliary *ambient_space()*) was constructed as a patch or neighborhood of a point then the embedding is the embedding into the original scheme.
- A `NotImplementedError` is raised if the construction of the embedding morphism is not implemented yet.

EXAMPLES:

```
sage: A2.<x,y> = AffineSpace(QQ, 2)
sage: C = A2.subscheme(x^2 + y^2 - 1)
sage: C.embedding_morphism()                                                #␣
↪needs sage.libs.singular
  Scheme morphism:
  From: Closed subscheme of Affine Space of dimension 2 over Rational Field
        defined by: x^2 + y^2 - 1
  To:   Affine Space of dimension 2 over Rational Field
  Defn: Defined on coordinates by sending (x, y) to (x, y)

sage: # needs sage.graphs sage.geometry.polyhedron sage.libs.singular
sage: P1xP1.<x,y,u,v> = toric_varieties.P1xP1()
sage: P1 = P1xP1.subscheme(x - y)
sage: P1.embedding_morphism()
Scheme morphism:
  From: Closed subscheme of 2-d CPR-Fano toric variety covered
        by 4 affine patches defined by: x - y
  To:   2-d CPR-Fano toric variety covered by 4 affine patches
  Defn: Defined on coordinates by sending [x : y : u : v] to [y : y : u : v]
```

So far, the embedding was just in the own ambient space. Now a bit more interesting examples:

```
sage: P2.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: X = P2.subscheme((x^2-y^2)*z)
sage: p = (1,1,0)
sage: nbhd = X.neighborhood(p); nbhd
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  -y^2*z - 2*y*z
```

Note that $p = (1, 1, 0)$ is a singular point of X. So the neighborhood of p is not just affine space. The `neighborhood()` method returns a presentation of the neighborhood as a subscheme of an auxiliary 2-dimensional affine space:

```
sage: nbhd.ambient_space()
Affine Space of dimension 2 over Rational Field
```

But its *embedding_morphism()* is not into this auxiliary affine space, but the original subscheme X:

```
sage: nbhd.embedding_morphism()
Scheme morphism:
  From: Closed subscheme of Affine Space of dimension 2 over Rational Field
        defined by: -y^2*z - 2*y*z
  To:   Closed subscheme of Projective Space of dimension 2 over Rational Field
        defined by: x^2*z - y^2*z
  Defn: Defined on coordinates by sending (y, z) to (1 : y + 1 : z)
```

A couple more examples:

```
sage: # needs sage.geometry.polyhedron sage.graphs sage.libs.singular
sage: patch1 = P1xP1.affine_patch(1); patch1
2-d affine toric variety
sage: patch1.embedding_morphism()
Scheme morphism:
  From: 2-d affine toric variety
  To:   2-d CPR-Fano toric variety covered by 4 affine patches
  Defn: Defined on coordinates by sending [y : u] to [1 : y : u : 1]
sage: subpatch = P1.affine_patch(1); subpatch
Closed subscheme of 2-d affine toric variety defined by:
  -y + 1
sage: subpatch.embedding_morphism()
Scheme morphism:
  From: Closed subscheme of 2-d affine toric variety defined by: -y + 1
  To:   Closed subscheme of 2-d CPR-Fano toric variety covered
        by 4 affine patches defined by: x - y
  Defn: Defined on coordinates by sending [y : u] to [1 : y : u : 1]
```

identity_morphism()

Return the identity morphism.

OUTPUT: the identity morphism of the scheme `self`

EXAMPLES:

```
sage: X = Spec(QQ)
sage: X.identity_morphism()
Scheme endomorphism of Spectrum of Rational Field
  Defn: Identity map
```

is_projective()

Return True if self is presented as a subscheme of an ambient projective space.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: PP.<x,y,z,w> = ProjectiveSpace(3, QQ)
sage: f = x^3 + y^3 + z^3 + w^3
sage: R = f.parent()
sage: I = [f] + [f.derivative(zz) for zz in PP.gens()]
sage: V = PP.subscheme(I)
sage: V.is_projective()
True
sage: AA.<x,y,z,w> = AffineSpace(4, QQ)
sage: V = AA.subscheme(I)
sage: V.is_projective()
False
```

Note that toric varieties are implemented differently than projective spaces. This is why this method returns `False` for toric varieties:

```
sage: # needs sage.geometry.polyhedron sage.graphs
sage: PP.<x,y,z,w> = toric_varieties.P(3)
sage: V = PP.subscheme(x^3 + y^3 + z^3 + w^3)
sage: V.is_projective()
False
```

ngens()

Return the number of generators of the ambient space of this algebraic scheme.

EXAMPLES:

```
sage: A.<x, y> = AffineSpace(2, GF(5))
sage: S = A.subscheme([])
sage: S.ngens()
2
sage: P.<x, y, z> = ProjectiveSpace(2, ZZ)
sage: S = P.subscheme([x - y, x - z])
sage: P.ngens()
3
```

class sage.schemes.generic.algebraic_scheme.**AlgebraicScheme_quasi**(X, Y)

Bases: *AlgebraicScheme*

The quasi-affine or quasi-projective scheme $X - Y$, where X and Y are both closed subschemes of a common ambient affine or projective space.

Warning: You should not create objects of this class directly. The preferred method to construct such subschemes is to use `complement()` method of algebraic schemes.

OUTPUT:

An instance of *AlgebraicScheme_quasi*.

EXAMPLES:

```
sage: P.<x, y, z> = ProjectiveSpace(2, ZZ)
sage: S = P.subscheme([])
sage: T = P.subscheme([x - y])
sage: T.complement(S)
Quasi-projective subscheme X - Y
 of Projective Space of dimension 2 over Integer Ring,
 where X is defined by:
   (no polynomials)
 and Y is defined by:
   x - y
```

X()

Return the scheme X such that self is represented as $X - Y$.

EXAMPLES:

```
sage: P.<x, y, z> = ProjectiveSpace(2, ZZ)
sage: S = P.subscheme([])
sage: T = P.subscheme([x - y])
sage: U = T.complement(S)
sage: U.X() is S
True
```

Y()

Return the scheme Y such that self is represented as $X - Y$.

EXAMPLES:

```
sage: P.<x, y, z> = ProjectiveSpace(2, ZZ)
sage: S = P.subscheme([])
sage: T = P.subscheme([x - y])
sage: U = T.complement(S)
sage: U.Y() is T
True
```

rational_points(***kwds*)

Return the set of rational points on this algebraic scheme over the field F.

INPUT:

kwds:

- bound - integer (optional, default=0). The bound for the coordinates for subschemes with dimension at least 1.
- F - field (optional, default=base ring). The field to compute the rational points over.

EXAMPLES:

```
sage: A.<x, y> = AffineSpace(2, GF(7))
sage: S = A.subscheme([x^2 - y])
sage: T = A.subscheme([x - y])
sage: U = T.complement(S)
sage: U.rational_points()
[(2, 4), (3, 2), (4, 2), (5, 4), (6, 1)]
```

(continues on next page)

(continued from previous page)

```
sage: U.rational_points(F=GF(7^2, 'b'))                                    #␣
↪needs sage.rings.finite_rings
[(2, 4), (3, 2), (4, 2), (5, 4), (6, 1), (b, b + 4), (b + 1, 3*b + 5),
 (b + 2, 5*b + 1), (b + 3, 6), (b + 4, 2*b + 6), (b + 5, 4*b + 1),
 (b + 6, 6*b + 5), (2*b, 4*b + 2), (2*b + 1, b + 3), (2*b + 2, 5*b + 6),
 (2*b + 3, 2*b + 4), (2*b + 4, 6*b + 4), (2*b + 5, 3*b + 6), (2*b + 6, 3),
 (3*b, 2*b + 1), (3*b + 1, b + 2), (3*b + 2, 5), (3*b + 3, 6*b + 3),
 (3*b + 4, 5*b + 3), (3*b + 5, 4*b + 5), (3*b + 6, 3*b + 2),
 (4*b, 2*b + 1), (4*b + 1, 3*b + 2), (4*b + 2, 4*b + 5),
 (4*b + 3, 5*b + 3), (4*b + 4, 6*b + 3), (4*b + 5, 5), (4*b + 6, b + 2),
 (5*b, 4*b + 2), (5*b + 1, 3), (5*b + 2, 3*b + 6), (5*b + 3, 6*b + 4),
 (5*b + 4, 2*b + 4), (5*b + 5, 5*b + 6), (5*b + 6, b + 3), (6*b, b + 4),
 (6*b + 1, 6*b + 5), (6*b + 2, 4*b + 1), (6*b + 3, 2*b + 6), (6*b + 4, 6),
 (6*b + 5, 5*b + 1), (6*b + 6, 3*b + 5)]
```

class sage.schemes.generic.algebraic_scheme.**AlgebraicScheme_subscheme**(*A*, *polynomials*)

Bases: *AlgebraicScheme*

An algebraic scheme presented as a closed subscheme is defined by explicit polynomial equations. This is as opposed to a general scheme, which could, e.g., be the Neron model of some object, and for which we do not want to give explicit equations.

INPUT:

- `A` - ambient space (e.g. affine or projective n-space)
- `polynomials` - single polynomial, ideal or iterable of defining polynomials; in any case polynomials must belong to the coordinate ring of the ambient space and define valid polynomial functions (e.g. they should be homogeneous in the case of a projective space)

OUTPUT:

- algebraic scheme

EXAMPLES:

```
sage: from sage.schemes.generic.algebraic_scheme import AlgebraicScheme_subscheme
sage: P.<x, y, z> = ProjectiveSpace(2, QQ)
sage: P.subscheme([x^2 - y*z])
Closed subscheme of Projective Space of dimension 2 over Rational Field defined by:
  x^2 - y*z
sage: AlgebraicScheme_subscheme(P, [x^2 - y*z])
Closed subscheme of Projective Space of dimension 2 over Rational Field defined by:
  x^2 - y*z
```

Jacobian()

Return the Jacobian ideal.

This is the ideal generated by

- the $d \times d$ minors of the Jacobian matrix, where d is the *codimension()* of the algebraic scheme, and
- the defining polynomials of the algebraic scheme. Note that some authors do not include these in the definition of the Jacobian ideal. An example of a reference that does include the defining equations is [Laz2004], p. 181.

OUTPUT:

An ideal in the coordinate ring of the ambient space.

EXAMPLES:

```
sage: P3.<w,x,y,z> = ProjectiveSpace(3, QQ)
sage: twisted_cubic = P3.subscheme(matrix([[w, x, y],                    #␣
↪needs sage.libs.singular
....:                                     [x, y, z]]).minors(2))
sage: twisted_cubic.Jacobian()                                           #␣
↪needs sage.libs.singular
Ideal (-x^2 + w*y, -x*y + w*z, -y^2 + x*z, x*z, -2*w*z, w*y, 3*w*y,
       -2*w*x, w^2, y*z, -2*x*z, w*z, 3*w*z, -2*w*y, w*x, z^2, -2*y*z,
       x*z, 3*x*z, -2*w*z, w*y)
 of Multivariate Polynomial Ring in w, x, y, z over Rational Field
sage: twisted_cubic.defining_ideal()                                     #␣
↪needs sage.libs.singular
Ideal (-x^2 + w*y, -x*y + w*z, -y^2 + x*z)
 of Multivariate Polynomial Ring in w, x, y, z over Rational Field
```

This example addresses issue github issue #20512:

```
sage: X = P3.subscheme([])
sage: X.Jacobian() == P3.coordinate_ring().unit_ideal()                  #␣
↪needs sage.libs.singular
True
```

Jacobian_matrix()

Return the matrix $\frac{\partial f_i}{\partial x_j}$ of (formal) partial derivatives.

OUTPUT:

A matrix of polynomials.

EXAMPLES:

```
sage: P3.<w,x,y,z> = ProjectiveSpace(3, QQ)
sage: twisted_cubic = P3.subscheme(matrix([[w, x, y],                    #␣
↪needs sage.libs.singular
....:                                     [x, y, z]]).minors(2))
sage: twisted_cubic.Jacobian_matrix()                                    #␣
↪needs sage.libs.singular
[   y -2*x    w    0]
[   z   -y   -x    w]
[   0    z -2*y    x]
```

This example addresses issue github issue #20512:

```
sage: X = P3.subscheme([])
sage: X.Jacobian_matrix().base_ring() == P3.coordinate_ring()            #␣
↪needs sage.libs.singular
True
```

base_extend(R)

Return the base change to the ring R of this scheme.

EXAMPLES:

```
sage: P.<x, y, z> = ProjectiveSpace(2, GF(11))
sage: S = P.subscheme([x^2 - y*z])
sage: S.base_extend(GF(11^2, 'b'))                                              #␣
↪needs sage.rings.finite_rings
Closed subscheme of Projective Space of dimension 2
 over Finite Field in b of size 11^2
 defined by: x^2 - y*z
sage: S.base_extend(ZZ)
Traceback (most recent call last):
...
ValueError: no natural map from the base ring (=Finite Field of size 11)
to R (=Integer Ring)!
```

change_ring(*R*)

Returns a new algebraic subscheme which is this subscheme coerced to R.

INPUT:

- R – ring or morphism.

OUTPUT:

- A new algebraic subscheme which is this subscheme coerced to R.

EXAMPLES:

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: X = P.subscheme([3*x^2 - y^2])
sage: H = Hom(X, X)
sage: X.change_ring(GF(3))
Closed subscheme of Projective Space of dimension 1
 over Finite Field of size 3 defined by: -y^2
```

```
sage: # needs sage.rings.number_field
sage: K.<w> = QuadraticField(2)
sage: R.<z> = K[]
sage: L.<v> = K.extension(z^3 - 5)
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: X = P.subscheme(x - w*y)                                                  #␣
↪needs sage.libs.singular
sage: X.change_ring(L)                                                          #␣
↪needs sage.libs.singular
Closed subscheme of Projective Space of dimension 1 over
 Number Field in v with defining polynomial z^3 - 5 over its base field
 defined by: x + (-w)*y
```

```
sage: # needs sage.rings.number_field
sage: K.<w> = QuadraticField(2)
sage: R.<z> = K[]
sage: L.<v> = K.extension(z^3 - 5)
sage: P.<x,y,z> = AffineSpace(L, 3)
sage: X = P.subscheme([x - w*y, z^2 - v*x])                                     #␣
↪needs sage.libs.singular
sage: emb = L.embeddings(QQbar)                                                 #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.libs.singular
sage: X.change_ring(emb[0])                                                 #␣
↪needs sage.libs.singular
Closed subscheme of Affine Space of dimension 3 over Algebraic Field
 defined by:
  x + (-1.414213562373095? + 0.?e-16*I)*y,
  z^2 + (0.8549879733383485? + 1.480882609682365?*I)*x
```

```
sage: # needs sage.rings.number_field
sage: K.<w> = QuadraticField(2)
sage: R.<z> = K[]
sage: L.<v> = K.extension(z^3 - 5)
sage: P.<x,y,z> = AffineSpace(L, 3)
sage: X = P.subscheme([x - w*y, z^2 - v*x])                                 #␣
↪needs sage.libs.singular
sage: emb = L.embeddings(QQbar)                                             #␣
↪needs sage.libs.singular
sage: X.change_ring(emb[1])                                                 #␣
↪needs sage.libs.singular
Closed subscheme of Affine Space of dimension 3 over Algebraic Field
 defined by:
  x + (-1.414213562373095? + 0.?e-16*I)*y,
  z^2 + (0.8549879733383485? - 1.480882609682365?*I)*x
```

```
sage: # needs sage.rings.number_field
sage: K.<w> = QuadraticField(-3)
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: X = P.subscheme(x - w*y)                                              #␣
↪needs sage.libs.singular
sage: X.change_ring(CC)                                                     #␣
↪needs sage.libs.singular
Closed subscheme of Projective Space of dimension 1
 over Complex Field with 53 bits of precision defined by:
  x + (-1.73205080756888*I)*y
```

```
sage: # needs sage.rings.number_field
sage: K.<w> = QuadraticField(3)
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: X = P.subscheme(x - w*y)                                              #␣
↪needs sage.libs.singular
sage: X.change_ring(RR)                                                     #␣
↪needs sage.libs.singular
Closed subscheme of Projective Space of dimension 1
 over Real Field with 53 bits of precision defined by:
  x - 1.73205080756888*y
```

```
sage: # needs sage.rings.number_field
sage: K.<v> = CyclotomicField(7)
sage: O = K.maximal_order()
sage: P.<x,y> = ProjectiveSpace(O, 1)
sage: X = P.subscheme([x^2 + O(v)*y^2])                                     #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.libs.singular
sage: X.change_ring(CC)                                                    #␣
↪needs sage.libs.singular
Closed subscheme of Projective Space of dimension 1
 over Complex Field with 53 bits of precision defined by:
  x^2 + (0.623489801858734 + 0.781831482468030*I)*y^2
sage: X.change_ring(K).change_ring(K.embeddings(QQbar)[3])                 #␣
↪needs sage.libs.singular
Closed subscheme of Projective Space of dimension 1
 over Algebraic Field defined by:
  x^2 + (-0.90096886790241?? - 0.4338837391175581?*I)*y^2
```

```
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: f = x^6 - 2
sage: L.<b> = NumberField(f, embedding=f.roots(CC)[2][0])
sage: A.<x,y> = AffineSpace(L, 2)
sage: H = Hom(A, A)
sage: X = A.subscheme([b*x^2, y^2])                                        #␣
↪needs sage.libs.singular
sage: X.change_ring(CC)                                                    #␣
↪needs sage.libs.singular
Closed subscheme of Affine Space of dimension 2
 over Complex Field with 53 bits of precision defined by:
  (-0.561231024154687 - 0.972080648619833*I)*x^2,
  y^2
```

codimension()

Return the codimension of the algebraic subscheme.

OUTPUT:

Integer.

EXAMPLES:

```
sage: PP.<x,y,z,w,v> = ProjectiveSpace(4, QQ)
sage: V = PP.subscheme(x*y)
sage: V.codimension()                                                      #␣
↪needs sage.libs.singular
1
sage: V.dimension()                                                        #␣
↪needs sage.libs.singular
3
```

complement(*other=None*)

Return the scheme-theoretic complement other - self, where self and other are both closed algebraic subschemes of the same ambient space.

If other is unspecified, it is taken to be the ambient space of self.

EXAMPLES:

```
sage: A.<x, y, z> = AffineSpace(3, ZZ)
sage: X = A.subscheme([x + y - z])
sage: Y = A.subscheme([x - y + z])
sage: Y.complement(X)
Quasi-affine subscheme X - Y of
 Affine Space of dimension 3 over Integer Ring,
 where X is defined by:
   x + y - z
 and Y is defined by:
   x - y + z
sage: Y.complement()
Quasi-affine subscheme X - Y of
 Affine Space of dimension 3 over Integer Ring,
 where X is defined by:
   (no polynomials)
 and Y is defined by:
   x - y + z
sage: P.<x, y, z> = ProjectiveSpace(2, QQ)
sage: X = P.subscheme([x^2 + y^2 + z^2])
sage: Y = P.subscheme([x*y + y*z + z*x])
sage: Y.complement(X)
Quasi-projective subscheme X - Y of
 Projective Space of dimension 2 over Rational Field,
 where X is defined by:
   x^2 + y^2 + z^2
 and Y is defined by:
   x*y + x*z + y*z
sage: Y.complement(P)
Quasi-projective subscheme X - Y of
 Projective Space of dimension 2 over Rational Field,
 where X is defined by:
   (no polynomials)
 and Y is defined by:
   x*y + x*z + y*z
```

defining_ideal()

Return the ideal that defines this scheme as a subscheme of its ambient space.

OUTPUT:

An ideal in the coordinate ring of the ambient space.

EXAMPLES:

```
sage: P.<x, y, z> = ProjectiveSpace(2, ZZ)
sage: S = P.subscheme([x^2 - y*z, x^3 + z^3])
sage: S.defining_ideal()
Ideal (x^2 - y*z, x^3 + z^3) of Multivariate Polynomial Ring in x, y, z
 over Integer Ring
```

defining_polynomials()

Return the polynomials that define this scheme as a subscheme of its ambient space.

OUTPUT:

A tuple of polynomials in the coordinate ring of the ambient space.

EXAMPLES:

```
sage: P.<x, y, z> = ProjectiveSpace(2, ZZ)
sage: S = P.subscheme([x^2 - y*z, x^3 + z^3])
sage: S.defining_polynomials()
(x^2 - y*z, x^3 + z^3)
```

intersection(*other*)

Return the scheme-theoretic intersection of self and other in their common ambient space.

EXAMPLES:

```
sage: A.<x, y> = AffineSpace(2, ZZ)
sage: X = A.subscheme([x^2 - y])
sage: Y = A.subscheme([y])
sage: X.intersection(Y)
Closed subscheme of Affine Space of dimension 2 over Integer Ring defined by:
  x^2 - y,
  y
```

irreducible_components()

Return the irreducible components of this algebraic scheme, as subschemes of the same ambient space.

OUTPUT:

an immutable sequence of irreducible subschemes of the ambient space of this scheme

The components are cached.

EXAMPLES:

We define what is clearly a union of four hypersurfaces in $\P^4_{\mathbf{Q}}$ then find the irreducible components:

```
sage: PP.<x,y,z,w,v> = ProjectiveSpace(4, QQ)
sage: V = PP.subscheme((x^2 - y^2 - z^2) * (w^5 - 2*v^2*z^3) * w * (v^3 - x^
↪2*z))
sage: V.irreducible_components()                                    #␣
↪needs sage.libs.singular
[
Closed subscheme of Projective Space of dimension 4 over Rational Field defined␣
↪by:
  w,
Closed subscheme of Projective Space of dimension 4 over Rational Field defined␣
↪by:
  x^2 - y^2 - z^2,
Closed subscheme of Projective Space of dimension 4 over Rational Field defined␣
↪by:
  x^2*z - v^3,
Closed subscheme of Projective Space of dimension 4 over Rational Field defined␣
↪by:
  w^5 - 2*z^3*v^2
]
```

We verify that the irrelevant ideal is not accidentally returned (see github issue #6920):

```
sage: PP.<x,y,z,w> = ProjectiveSpace(3, QQ)
sage: f = x^3 + y^3 + z^3 + w^3
```

(continues on next page)

(continued from previous page)

```
sage: R = f.parent()
sage: I = [f] + [f.derivative(zz) for zz in PP.gens()]
sage: V = PP.subscheme(I)
sage: V.irreducible_components()                                              #␣
↪needs sage.libs.singular
[

]
```

The same polynomial as above defines a scheme with a nontrivial irreducible component in affine space (instead of the empty scheme as above):

```
sage: AA.<x,y,z,w> = AffineSpace(4, QQ)
sage: V = AA.subscheme(I)
sage: V.irreducible_components()                                              #␣
↪needs sage.libs.singular
[
Closed subscheme of Affine Space of dimension 4 over Rational Field defined by:
  w,
  z,
  y,
  x
]
```

is_irreducible()

Return whether this subscheme is or is not irreducible.

OUTPUT: Boolean.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: K = QuadraticField(-3)
sage: P.<x,y,z,w,t,u> = ProjectiveSpace(K, 5)
sage: X = P.subscheme([x*y - z^2 - K.0*t^2, t*w*x + y*z^2 - u^3])
sage: X.is_irreducible()                                                      #␣
↪needs sage.libs.singular
True
```

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: X = P.subscheme([(y + x - z)^2])
sage: X.is_irreducible()                                                      #␣
↪needs sage.libs.singular
False
```

```
sage: A.<x,y,z,w> = AffineSpace(GF(17), 4)
sage: X = A.subscheme([
....:         x*y*z^2 - x*y*z*w - z*w^2 + w^3,
....:         x^3*y*z*w - x*y^3*z - x^2*y*z*w - x^2*w^3 + y^2*w^2 + x*w^3
....:     ])
sage: X.is_irreducible()                                                      #␣
↪needs sage.libs.singular
False
```

normalize_defining_polynomials()

Function to normalize the coefficients of defining polynomials of given subscheme.

Normalization as in removing denominator from all the coefficients, and then removing any common factor between the coefficients. It takes LCM of denominators and then removes common factor among coefficients, if any.

EXAMPLES:

```
sage: A.<x,y> = AffineSpace(2, QQ)
sage: S = A.subscheme([2*x^2 + 4*x*y, 1/8*x + 1/3*y])
sage: S.normalize_defining_polynomials()
sage: S.defining_polynomials()
(x^2 + 2*x*y, 3*x + 8*y)
```

rational_points(***kwds*)

Return the rational points on the algebraic subscheme.

For a dimension 0 subscheme, if the base ring is a numerical field such as the ComplexField the results returned could be very far from correct. If the polynomials defining the subscheme are defined over a number field, you will get better results calling rational points with F defined as the number field and the base ring as the field of definition. If the base ring is a number field, the embedding into `F` must be known.

In the case of numerically approximated points, the points are returned over as points of the ambient space.

For a dimension greater than 0 scheme, depending on bound size, either the points in the ambient space are enumerated or a sieving algorithm lifting points modulo primes is used. See the documentation in homset for the details of the sieving algorithm.

INPUT:

kwds:

- `bound` - integer (optional, default=0). The bound for the coordinates for subschemes with dimension at least 1.
- `prec` - integer (optional, default=53). The precision to use to compute the elements of bounded height for number fields.
- `F` - field (optional, default=base ring). The field to compute the rational points over.
- `point_tolerance` - positive real number (optional, default=10^(-10)). For numerically inexact fields, two points are considered the same if their coordinates are within tolerance.
- `zero_tolerance` - positive real number (optional, default=10^(-10)). For numerically inexact fields, points are on the subscheme if they satisfy the equations to within tolerance.
- `tolerance` - a rational number in (0,1] used in doyle-krumm algorithm-4

OUTPUT: list of points in subscheme or ambient space

Warning: For numerically inexact fields such as ComplexField or RealField the list of points returned is very likely to be incomplete at best.

EXAMPLES:

Enumerate over a projective scheme over a number field:

```
sage: # needs sage.rings.number_field
sage: u = QQ['u'].0
sage: K.<v> = NumberField(u^2 + 3)
sage: A.<x,y> = ProjectiveSpace(K, 1)
sage: X = A.subscheme(x^2 - y^2)
sage: X.rational_points(bound=3)                                            #␣
↪needs sage.libs.singular
[(-1 : 1), (1 : 1)]
```

One can enumerate points up to a given bound on a projective scheme over the rationals:

```
sage: E = EllipticCurve('37a')                                              #␣
↪needs sage.schemes
sage: E.rational_points(bound=8)                                            #␣
↪needs sage.schemes
[(-1 : -1 : 1), (-1 : 0 : 1), (0 : -1 : 1), (0 : 0 : 1), (0 : 1 : 0),
 (1/4 : -5/8 : 1), (1/4 : -3/8 : 1), (1 : -1 : 1), (1 : 0 : 1),
 (2 : -3 : 1), (2 : 2 : 1)]
```

For a small finite field, the complete set of points can be enumerated.

```
sage: Etilde = E.base_extend(GF(3))                                         #␣
↪needs sage.schemes
sage: Etilde.rational_points()                                              #␣
↪needs sage.schemes
[(0 : 0 : 1), (0 : 1 : 0), (0 : 2 : 1), (1 : 0 : 1),
 (1 : 2 : 1), (2 : 0 : 1), (2 : 2 : 1)]
```

The class of hyperelliptic curves does not (yet) support desingularization of the places at infinity into two points:

```
sage: FF = FiniteField(7)
sage: P.<x> = PolynomialRing(FiniteField(7))
sage: C = HyperellipticCurve(x^8 + x + 1)                                   #␣
↪needs sage.schemes
sage: C.rational_points()                                                   #␣
↪needs sage.schemes
[(0 : 1 : 0), (0 : 1 : 1), (0 : 6 : 1), (2 : 0 : 1),
 (4 : 0 : 1), (6 : 1 : 1), (6 : 6 : 1)]
```

```
sage: # needs sage.rings.number_field
sage: K.<v> = QuadraticField(-3)
sage: P.<x,y,z> = ProjectiveSpace(K, 2)
sage: X = P.subscheme([x^2 - v^2*x*z, y*x - v*z^2])
sage: X.rational_points(F=CC)
[(-3.00000000000000 : -0.577350269189626*I : 1.00000000000000),
 (0.000000000000000 : 1.00000000000000 : 0.000000000000000)]
```

```
sage: # needs sage.rings.number_field
sage: K.<v> = QuadraticField(3)
sage: A.<x,y> = AffineSpace(K, 2)
sage: X = A.subscheme([x^2 - v^2*y, y*x - v])
```

(continues on next page)

(continued from previous page)

```
sage: X.rational_points(F=RR)
[(1.73205080756888, 1.00000000000000)]
```

Todo: Implement Stoll's model in weighted projective space to resolve singularities and find two points (1 : 1 : 0) and (-1 : 1 : 0) at infinity.

reduce()

Return the corresponding reduced algebraic space associated to this scheme.

EXAMPLES: First we construct the union of a doubled and tripled line in the affine plane over $\mathbf{Q}$

```
sage: A.<x,y> = AffineSpace(2, QQ)
sage: X = A.subscheme([(x-1)^2*(x-y)^3]); X
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x^5 - 3*x^4*y + 3*x^3*y^2 - x^2*y^3 - 2*x^4 + 6*x^3*y
  - 6*x^2*y^2 + 2*x*y^3 + x^3 - 3*x^2*y + 3*x*y^2 - y^3
sage: X.dimension()                                                    #␣
↪needs sage.libs.singular
1
```

Then we compute the corresponding reduced scheme:

```
sage: Y = X.reduce(); Y                                                #␣
↪needs sage.libs.singular
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x^2 - x*y - x + y
```

Finally, we verify that the reduced scheme Y is the union of those two lines:

```
sage: # needs sage.libs.singular
sage: L1 = A.subscheme([x - 1]); L1
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x - 1
sage: L2 = A.subscheme([x - y]); L2
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x - y
sage: W = L1.union(L2); W              # taken in ambient space
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x^2 - x*y - x + y
sage: Y == W
True
```

specialization(*D=None*, *phi=None*)

Specialization of this subscheme.

Given a family of maps defined over a polynomial ring. A specialization is a particular member of that family. The specialization can be specified either by a dictionary or a `SpecializationMorphism`.

INPUT:

- `D` – dictionary (optional)
- `phi` – `SpecializationMorphism` (optional)

OUTPUT: SchemeMorphism_polynomial

EXAMPLES:

```
sage: R.<c> = PolynomialRing(QQ)
sage: P.<x,y> = ProjectiveSpace(R, 1)
sage: X = P.subscheme([x^2 + c*y^2])
sage: X.specialization(dict({c:2}))
Closed subscheme of Projective Space of dimension 1 over Rational Field defined␣
↪by:
  x^2 + 2*y^2
```

```
sage: R.<c> = PolynomialRing(QQ)
sage: S.<a,b> = R[]
sage: P.<x,y,z> = AffineSpace(S, 3)
sage: X = P.subscheme([x^2 + a*c*y^2 - b*z^2])
sage: from sage.rings.polynomial.flatten import SpecializationMorphism
sage: phi = SpecializationMorphism(P.coordinate_ring(),
....:                              dict({c: 2, a: 1}))
sage: X.specialization(phi=phi)                                                 #␣
↪needs sage.libs.singular
Closed subscheme of Affine Space of dimension 3
 over Univariate Polynomial Ring in b over Rational Field defined by:
  x^2 + 2*y^2 + (-b)*z^2
```

union(*other*)

Return the scheme-theoretic union of self and other in their common ambient space.

EXAMPLES: We construct the union of a line and a tripled-point on the line.

```
sage: A.<x,y> = AffineSpace(2, QQ)
sage: I = ideal([x, y])^3
sage: P = A.subscheme(I)
sage: L = A.subscheme([y - 1])
sage: S = L.union(P); S                                                         #␣
↪needs sage.libs.singular
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  y^4 - y^3,
  x*y^3 - x*y^2,
  x^2*y^2 - x^2*y,
  x^3*y - x^3
sage: S.dimension()                                                             #␣
↪needs sage.libs.singular
1
sage: S.reduce()                                                                #␣
↪needs sage.libs.singular
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  y^2 - y,
  x*y - x
```

We can also use the notation "+" for the union:

```
sage: A.subscheme([x]) + A.subscheme([y^2 - (x^3+1)])                           #␣
↪needs sage.libs.singular
```

(continues on next page)

(continued from previous page)

```
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x^4 - x*y^2 + x
```

Saving and loading:

```
sage: loads(S.dumps()) == S                                                 #␣
↪needs sage.libs.singular
True
```

weil_restriction()

Compute the Weil restriction of this variety over some extension field. If the field is a finite field, then this computes the Weil restriction to the prime subfield.

A Weil restriction of scalars - denoted $Res_{L/k}$ - is a functor which, for any finite extension of fields L/k and any algebraic variety X over L, produces another corresponding variety $Res_{L/k}(X)$, defined over k. It is useful for reducing questions about varieties over large fields to questions about more complicated varieties over smaller fields.

This function does not compute this Weil restriction directly but computes on generating sets of polynomial ideals:

Let d be the degree of the field extension L/k, let a a generator of L/k and p the minimal polynomial of L/k. Denote this ideal by I.

Specifically, this function first maps each variable x to its representation over k: $\sum_{i=0}^{d-1} a^i x_i$. Then each generator of I is evaluated over these representations and reduced modulo the minimal polynomial p. The result is interpreted as a univariate polynomial in a and its coefficients are the new generators of the returned ideal.

If the input and the output ideals are radical, this is equivalent to the statement about algebraic varieties above.

OUTPUT: Affine subscheme - the Weil restriction of `self`.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: K.<w> = NumberField(x^5 - 2)
sage: R.<x> = K[]
sage: L.<v> = K.extension(x^2 + 1)
sage: A.<x,y> = AffineSpace(L, 2)
sage: X = A.subscheme([y^2 - L(w)*x^3 - v])                                 #␣
↪needs sage.libs.singular
sage: X.weil_restriction()                                                  #␣
↪needs sage.libs.singular
Closed subscheme of Affine Space of dimension 4
 over Number Field in w with defining polynomial x^5 - 2 defined by:
  (-w)*z0^3 + (3*w)*z0*z1^2 + z2^2 - z3^2,
  (-3*w)*z0^2*z1 + w*z1^3 + 2*z2*z3 - 1
sage: X.weil_restriction().ambient_space() is A.weil_restriction()          #␣
↪needs sage.libs.singular
True
```

```
sage: A.<x,y,z> = AffineSpace(GF(5^2, 't'), 3)                                #␣
↪needs sage.rings.finite_rings
sage: X = A.subscheme([y^2 - x*z, z^2 + 2*y])                                 #␣
↪needs sage.libs.singular sage.rings.finite_rings
sage: X.weil_restriction()                                                    #␣
↪needs sage.libs.singular sage.rings.finite_rings
Closed subscheme of Affine Space of dimension 6
 over Finite Field of size 5 defined by:
  z2^2 - 2*z3^2 - z0*z4 + 2*z1*z5,
  2*z2*z3 + z3^2 - z1*z4 - z0*z5 - z1*z5,
  z4^2 - 2*z5^2 + 2*z2,
  2*z4*z5 + z5^2 + 2*z3
```

sage.schemes.generic.algebraic_scheme.**is_AlgebraicScheme**(*x*)

Test whether `x` is an algebraic scheme.

INPUT:

- `x` – anything.

OUTPUT:

Boolean. Whether `x` is an algebraic scheme, that is, a subscheme of an ambient space over a ring defined by polynomial equations.

EXAMPLES:

```
sage: A2 = AffineSpace(2, QQ, 'x, y')
sage: A2.coordinate_ring().inject_variables()
Defining x, y
sage: V = A2.subscheme([x^2 + y^2]); V
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x^2 + y^2
sage: from sage.schemes.generic.algebraic_scheme import is_AlgebraicScheme
sage: is_AlgebraicScheme(V)
True
```

Affine space is itself not an algebraic scheme, though the closed subscheme defined by no equations is:

```
sage: from sage.schemes.generic.algebraic_scheme import is_AlgebraicScheme
sage: is_AlgebraicScheme(AffineSpace(10, QQ))
False
sage: V = AffineSpace(10, QQ).subscheme([]); V
Closed subscheme of Affine Space of dimension 10 over Rational Field defined by:
  (no polynomials)
sage: is_AlgebraicScheme(V)
True
```

We create a more complicated closed subscheme:

```
sage: A,x = AffineSpace(10, QQ).objgens()
sage: X = A.subscheme([sum(x)]); X
Closed subscheme of Affine Space of dimension 10 over Rational Field defined by:
  x0 + x1 + x2 + x3 + x4 + x5 + x6 + x7 + x8 + x9
sage: is_AlgebraicScheme(X)
True
```

```
sage: is_AlgebraicScheme(QQ)
False
sage: S = Spec(QQ)
sage: is_AlgebraicScheme(S)
False
```

CHAPTER

EIGHT

HYPERSURFACES IN AFFINE AND PROJECTIVE SPACE

AUTHORS:

- William Stein <wstein@gmail.com> (2005-12-08)
- David Kohel <kohel@maths.usyd.edu.au> (2005-12-08)
- Alex Ghitza <aghitza@alum.mit.edu> (2009-04-17)

class sage.schemes.generic.hypersurface.**AffineHypersurface**(*poly*, *ambient=None*)

Bases: *AlgebraicScheme_subscheme_affine*

The affine hypersurface defined by the given polynomial.

EXAMPLES:

```
sage: A.<x, y, z> = AffineSpace(ZZ, 3)
sage: AffineHypersurface(x*y - z^3, A)
Affine hypersurface defined by -z^3 + x*y
 in Affine Space of dimension 3 over Integer Ring
```

```
sage: A.<x, y, z> = QQ[]
sage: AffineHypersurface(x*y - z^3)
Affine hypersurface defined by -z^3 + x*y
 in Affine Space of dimension 3 over Rational Field
```

defining_polynomial()

Return the polynomial equation that cuts out this affine hypersurface.

EXAMPLES:

```
sage: R.<x, y, z> = ZZ[]
sage: H = AffineHypersurface(x*z + y^2)
sage: H.defining_polynomial()
y^2 + x*z
```

class sage.schemes.generic.hypersurface.**ProjectiveHypersurface**(*poly*, *ambient=None*)

Bases: *AlgebraicScheme_subscheme_projective*

The projective hypersurface defined by the given polynomial.

EXAMPLES:

```
sage: P.<x, y, z> = ProjectiveSpace(ZZ, 2)
sage: ProjectiveHypersurface(x - y, P)
```

(continues on next page)

(continued from previous page)

```
Projective hypersurface defined by x - y
 in Projective Space of dimension 2 over Integer Ring
```

```
sage: R.<x, y, z> = QQ[]
sage: ProjectiveHypersurface(x - y)
Projective hypersurface defined by x - y
 in Projective Space of dimension 2 over Rational Field
```

defining_polynomial()

Return the polynomial equation that cuts out this projective hypersurface.

EXAMPLES:

```
sage: R.<x, y, z> = ZZ[]
sage: H = ProjectiveHypersurface(x*z + y^2)
sage: H.defining_polynomial()
y^2 + x*z
```

sage.schemes.generic.hypersurface.**is_Hypersurface**(*self*)

Return True if `self` is a hypersurface, i.e. an object of the type *ProjectiveHypersurface* or *AffineHypersurface*.

EXAMPLES:

```
sage: from sage.schemes.generic.hypersurface import is_Hypersurface
sage: R.<x, y, z> = ZZ[]
sage: H = ProjectiveHypersurface(x*z + y^2)
sage: is_Hypersurface(H)
True
```

```
sage: H = AffineHypersurface(x*z + y^2)
sage: is_Hypersurface(H)
True
```

```
sage: H = ProjectiveSpace(QQ, 5)
sage: is_Hypersurface(H)
False
```

CHAPTER

NINE

SET OF HOMOMORPHISMS BETWEEN TWO SCHEMES

For schemes X and Y, this module implements the set of morphisms $Hom(X, Y)$. This is done by *SchemeHomset_generic*.

As a special case, the Hom-sets can also represent the points of a scheme. Recall that the K-rational points of a scheme X over k can be identified with the set of morphisms $Spec(K) \to X$. In Sage the rational points are implemented by such scheme morphisms. This is done by *SchemeHomset_points* and its subclasses.

Note: You should not create the Hom-sets manually. Instead, use the `Hom()` method that is inherited by all schemes.

AUTHORS:

- William Stein (2006): initial version.
- Volker Braun (2011-08-11): significant improvement and refactoring.
- Ben Hutz (June 2012): added support for projective ring

class sage.schemes.generic.homset.**SchemeHomsetFactory**

Bases: UniqueFactory

Factory for Hom-sets of schemes.

EXAMPLES:

```
sage: A2 = AffineSpace(QQ, 2)
sage: A3 = AffineSpace(QQ, 3)
sage: Hom = A3.Hom(A2)
```

The Hom-sets are uniquely determined by domain and codomain:

```
sage: Hom is copy(Hom)
True
sage: Hom is A3.Hom(A2)
True
```

The Hom-sets are identical if the domains and codomains are identical:

```
sage: loads(Hom.dumps()) is Hom
True
sage: A3_iso = AffineSpace(QQ, 3)
sage: A3_iso is A3
True
sage: Hom_iso = A3_iso.Hom(A2)
```

(continues on next page)

(continued from previous page)

```
sage: Hom_iso is Hom
True
```

create_key_and_extra_args(*X*, *Y*, *category=None*, *base=None*, *check=True*, *as_point_homset=False*)

Create a key that uniquely determines the Hom-set.

INPUT:

- `X` – a scheme. The domain of the morphisms.
- `Y` – a scheme. The codomain of the morphisms.
- `category` – a category for the Hom-sets (default: schemes over given base).
- `base` – a scheme or a ring. The base scheme of domain and codomain schemes. If a ring is specified, the spectrum of that ring will be used as base scheme.
- `check` – boolean (default: `True`).

EXAMPLES:

```
sage: A2 = AffineSpace(QQ, 2)
sage: A3 = AffineSpace(QQ, 3)
sage: A3.Hom(A2)    # indirect doctest
Set of morphisms
  From: Affine Space of dimension 3 over Rational Field
  To:   Affine Space of dimension 2 over Rational Field
sage: from sage.schemes.generic.homset import SchemeHomsetFactory
sage: SHOMfactory = SchemeHomsetFactory('test')
sage: key, extra = SHOMfactory.create_key_and_extra_args(A3, A2, check=False)
sage: key
(..., ..., Category of schemes over Rational Field, False)
sage: extra
{'X': Affine Space of dimension 3 over Rational Field,
 'Y': Affine Space of dimension 2 over Rational Field,
 'base_ring': Rational Field,
 'check': False}
```

create_object(*version*, *key*, ***extra_args*)

Create a *SchemeHomset_generic*.

INPUT:

- `version` – object version. Currently not used.
- `key` – a key created by *create_key_and_extra_args()*.
- `extra_args` – a dictionary of extra keyword arguments.

EXAMPLES:

```
sage: A2 = AffineSpace(QQ, 2)
sage: A3 = AffineSpace(QQ, 3)
sage: A3.Hom(A2) is A3.Hom(A2)   # indirect doctest
True
sage: from sage.schemes.generic.homset import SchemeHomsetFactory
sage: SHOMfactory = SchemeHomsetFactory('test')
sage: SHOMfactory.create_object(0, [id(A3), id(A2), A3.category(), False],
```

(continues on next page)

(continued from previous page)

```
....:                                check=True, X=A3, Y=A2, base_ring=QQ)
Set of morphisms
  From: Affine Space of dimension 3 over Rational Field
  To:   Affine Space of dimension 2 over Rational Field
```

class sage.schemes.generic.homset.**SchemeHomset_generic**(*X*, *Y*, *category=None*, *check=True*, *base=None*)

Bases: HomsetWithBase

The base class for Hom-sets of schemes.

INPUT:

- `X` – a scheme. The domain of the Hom-set.
- `Y` – a scheme. The codomain of the Hom-set.
- `category` – a category (optional). The category of the Hom-set.
- `check` – boolean (optional, default: `True`). Whether to check the defining data for consistency.

EXAMPLES:

```
sage: from sage.schemes.generic.homset import SchemeHomset_generic
sage: A2 = AffineSpace(QQ,2)
sage: Hom = SchemeHomset_generic(A2, A2); Hom
Set of morphisms
  From: Affine Space of dimension 2 over Rational Field
  To:   Affine Space of dimension 2 over Rational Field
sage: Hom.category()
Category of endsets of schemes over Rational Field
```

Element

alias of *SchemeMorphism*

natural_map()

Return a natural map in the Hom space.

OUTPUT:

A *SchemeMorphism* if there is a natural map from domain to codomain. Otherwise, a `NotImplementedError` is raised.

EXAMPLES:

```
sage: A = AffineSpace(4, QQ)
sage: A.structure_morphism()   # indirect doctest
Scheme morphism:
  From: Affine Space of dimension 4 over Rational Field
  To:   Spectrum of Rational Field
  Defn: Structure map
```

class sage.schemes.generic.homset.**SchemeHomset_points**(*X*, *Y*, *category=None*, *check=True*, *base=Integer Ring*)

Bases: *SchemeHomset_generic*

Set of rational points of the scheme.

Recall that the K-rational points of a scheme X over k can be identified with the set of morphisms $Spec(K)oX$. In Sage, the rational points are implemented by such scheme morphisms.

If a scheme has a finite number of points, then the homset is supposed to implement the Python iterator interface. See *SchemeHomset_points_toric_field* for example.

INPUT:

See *SchemeHomset_generic*.

EXAMPLES:

```
sage: from sage.schemes.generic.homset import SchemeHomset_points
sage: SchemeHomset_points(Spec(QQ), AffineSpace(ZZ,2))
Set of rational points of Affine Space of dimension 2 over Rational Field
```

cardinality()

Return the number of points.

OUTPUT:

An integer or infinity.

EXAMPLES:

```
sage: toric_varieties.P2().point_set().cardinality()                    #␣
↪needs sage.geometry.polyhedron sage.graphs
+Infinity

sage: P2 = toric_varieties.P2(base_ring=GF(3))                          #␣
↪needs sage.geometry.polyhedron sage.graphs
sage: P2.point_set().cardinality()                                      #␣
↪needs sage.geometry.polyhedron sage.graphs
13
```

extended_codomain()

Return the codomain with extended base, if necessary.

OUTPUT:

The codomain scheme, with its base ring extended to the codomain. That is, the codomain is of the form $Spec(R)$ and the base ring of the domain is extended to R.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: P2 = ProjectiveSpace(QQ, 2)
sage: x = polygen(ZZ, 'x')
sage: K.<a> = NumberField(x^2 + x - (3^3-3))
sage: K_points = P2(K);  K_points
Set of rational points of Projective Space of dimension 2
 over Number Field in a with defining polynomial x^2 + x - 24
sage: K_points.codomain()
Projective Space of dimension 2 over Rational Field
sage: K_points.extended_codomain()
Projective Space of dimension 2
 over Number Field in a with defining polynomial x^2 + x - 24
```

list()

Return a tuple containing all points.

OUTPUT:

A tuple containing all points of the toric variety.

EXAMPLES:

```
sage: P1 = toric_varieties.P1(base_ring=GF(3))                                #␣
↪needs sage.geometry.polyhedron sage.graphs
sage: P1.point_set().list()                                                   #␣
↪needs sage.geometry.polyhedron sage.graphs
([0 : 1], [1 : 0], [1 : 1], [1 : 2])
```

value_ring()

Return R for a point Hom-set $X(Spec(R))$.

OUTPUT:

A commutative ring.

EXAMPLES:

```
sage: P2 = ProjectiveSpace(ZZ, 2)
sage: P2(QQ).value_ring()
Rational Field
```

sage.schemes.generic.homset.**is_SchemeHomset**(*H*)

Test whether H is a scheme Hom-set.

EXAMPLES:

```
sage: f = Spec(QQ).identity_morphism();  f
Scheme endomorphism of Spectrum of Rational Field
  Defn: Identity map
sage: from sage.schemes.generic.homset import is_SchemeHomset
sage: is_SchemeHomset(f)
False
sage: is_SchemeHomset(f.parent())
True
sage: is_SchemeHomset('a string')
False
```

CHAPTER

TEN

SCHEME MORPHISM

Note: You should never create the morphisms directly. Instead, use the `hom()` and `Hom()` methods that are inherited by all schemes.

If you want to extend the Sage library with some new kind of scheme, your new class (say, `MyScheme`) should provide a method

- `MyScheme._morphism(*args, **kwds)` returning a morphism between two schemes in your category, usually defined via polynomials. Your morphism class should derive from *SchemeMorphism_polynomial*. These morphisms will usually be elements of the Hom-set *SchemeHomset_generic*.

Optionally, you can also provide a special Hom-set class for your subcategory of schemes. If you want to do this, you should also provide a method

- `MyScheme._homset(*args, **kwds)` returning a Hom-set, which must be an element of a derived class of *SchemeHomset_generic*. If your new Hom-set class does not use `MyScheme._morphism` then you do not have to provide it.

Note that points on schemes are morphisms $Spec(K) \to X$, too. But we typically use a different notation, so they are implemented in a different derived class. For this, you should implement a method

- `MyScheme._point(*args, **kwds)` returning a point, that is, a morphism $Spec(K) \to X$. Your point class should derive from *SchemeMorphism_point*.

Optionally, you can also provide a special Hom-set for the points, for example the point Hom-set can provide a method to enumerate all points. If you want to do this, you should also provide a method

- `MyScheme._point_homset(*args, **kwds)` returning the *homset* of points. The Hom-sets of points are implemented in classes named `SchemeHomset_points_...`. If your new Hom-set class does not use `MyScheme._point` then you do not have to provide it.

AUTHORS:

- David Kohel, William Stein
- William Stein (2006-02-11): fixed bug where P(0,0,0) was allowed as a projective point.
- Volker Braun (2011-08-08): Renamed classes, more documentation, misc cleanups.
- Ben Hutz (June 2012): added support for projective ring
- Simon King (2013-10): copy the changes of `Morphism` that have been introduced in github issue #14711.

class sage.schemes.generic.morphism.**SchemeMorphism**(*parent*, *codomain=None*)

Bases: `Element`

Base class for scheme morphisms

INPUT:

- parent – the parent of the morphism.

Todo: For historical reasons, *SchemeMorphism* copies code from Map rather than inheriting from it. Proper inheritance should be used instead. See github issue #14711.

EXAMPLES:

```
sage: X = Spec(ZZ)
sage: Hom = X.Hom(X)
sage: from sage.schemes.generic.morphism import SchemeMorphism
sage: f = SchemeMorphism(Hom)
sage: type(f)
<class 'sage.schemes.generic.morphism.SchemeMorphism'>
```

base_ring()

Return the base ring of self, that is, the ring over which the defining polynomials of self are defined.

OUTPUT:

- ring

EXAMPLES:

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: H = Hom(P, P)
sage: f = H([3/5*x^2, 6*y^2])
sage: f.base_ring()
Rational Field
```

```
sage: R.<t> = PolynomialRing(ZZ, 1)
sage: P.<x,y> = ProjectiveSpace(R, 1)
sage: H = Hom(P, P)
sage: f = H([3*x^2, y^2])
sage: f.base_ring()
Multivariate Polynomial Ring in t over Integer Ring
```

Points have correct base rings too (github issue #34336):

```
sage: x = P(t, 5); x
(t : 5)
sage: x.base_ring()
Multivariate Polynomial Ring in t over Integer Ring
```

```
sage: # needs sage.rings.finite_rings sage.schemes
sage: E = EllipticCurve(GF((17,2)), [1,2,3,4,5])
sage: P = E.random_point()
sage: P.base_ring()
Finite Field in z2 of size 17^2
```

category()

Return the category of the Hom-set.

OUTPUT:

A category.

EXAMPLES:

```
sage: A2 = AffineSpace(QQ, 2)
sage: A2.structure_morphism().category()
Category of homsets of schemes
```

category_for()

Return the category which this morphism belongs to.

EXAMPLES:

```
sage: A2 = AffineSpace(QQ, 2)
sage: A2.structure_morphism().category_for()
Category of schemes
```

codomain()

The constant function from the codomain.

EXAMPLES:

```
sage: A.<x,y> = AffineSpace(QQ['x,y'])
sage: H = A.Hom(A)
sage: f = H([y, x^2 + y])
sage: f.codomain() is A
True
```

domain()

The constant function from the domain.

EXAMPLES:

```
sage: A.<x,y> = AffineSpace(QQ['x,y'])
sage: H = A.Hom(A)
sage: f = H([y, x^2 + y])
sage: f.domain() is A
True
```

glue_along_domains(*other*)

Glue two morphism

INPUT:

- `other` – a scheme morphism with the same domain.

OUTPUT:

Assuming that `self` and `other` are open immersions with the same domain, return scheme obtained by gluing along the images.

EXAMPLES:

We construct a scheme isomorphic to the projective line over $\mathrm{Spec}(\mathbf{Q})$ by gluing two copies of $\mathbb{A}^1$ minus a point:

```
sage: # needs sage.libs.singular
sage: R.<x,y> = PolynomialRing(QQ, 2)
```

(continues on next page)

(continued from previous page)

```
sage: S.<xbar, ybar> = R.quotient(x*y - 1)
sage: Rx = PolynomialRing(QQ, 'x')
sage: i1 = Rx.hom([xbar])
sage: Ry = PolynomialRing(QQ, 'y')
sage: i2 = Ry.hom([ybar])
sage: Sch = Schemes()
sage: f1 = Sch(i1)
sage: f2 = Sch(i2)
```

Now f1 and f2 have the same domain, which is a $\mathbb{A}^1$ minus a point. We glue along the domain:

```
sage: # needs sage.libs.singular
sage: P1 = f1.glue_along_domains(f2); P1
Scheme obtained by gluing X and Y along U, where
  X: Spectrum of Univariate Polynomial Ring in x over Rational Field
  Y: Spectrum of Univariate Polynomial Ring in y over Rational Field
  U: Spectrum of Quotient of Multivariate Polynomial Ring in x, y
     over Rational Field by the ideal (x*y - 1)
sage: a, b = P1.gluing_maps()
sage: a
Affine Scheme morphism:
  From: Spectrum of Quotient of Multivariate Polynomial Ring in x, y
        over Rational Field by the ideal (x*y - 1)
  To:   Spectrum of Univariate Polynomial Ring in x over Rational Field
  Defn: Ring morphism:
          From: Univariate Polynomial Ring in x over Rational Field
          To:   Quotient of Multivariate Polynomial Ring in x, y over
                Rational Field by the ideal (x*y - 1)
          Defn: x |--> xbar
sage: b
Affine Scheme morphism:
  From: Spectrum of Quotient of Multivariate Polynomial Ring in x, y
        over Rational Field by the ideal (x*y - 1)
  To:   Spectrum of Univariate Polynomial Ring in y over Rational Field
  Defn: Ring morphism:
          From: Univariate Polynomial Ring in y over Rational Field
          To:   Quotient of Multivariate Polynomial Ring in x, y over
                Rational Field by the ideal (x*y - 1)
          Defn: y |--> ybar
```

is_endomorphism()

Return whether the morphism is an endomorphism.

OUTPUT:

Boolean. Whether the domain and codomain are identical.

EXAMPLES:

```
sage: X = AffineSpace(QQ, 2)
sage: X.structure_morphism().is_endomorphism()
False
sage: X.identity_morphism().is_endomorphism()
True
```

class sage.schemes.generic.morphism.**SchemeMorphism_id**(X)

Bases: *SchemeMorphism*

Return the identity morphism from X to itself.

INPUT:

- `X` – the scheme.

EXAMPLES:

```
sage: X = Spec(ZZ)
sage: X.identity_morphism()  # indirect doctest
Scheme endomorphism of Spectrum of Integer Ring
  Defn: Identity map
```

class sage.schemes.generic.morphism.**SchemeMorphism_point**(*parent*, *codomain=None*)

Bases: *SchemeMorphism*

Base class for rational points on schemes.

Recall that the K-rational points of a scheme X over k can be identified with the set of morphisms $Spec(K)oX$. In Sage, the rational points are implemented by such scheme morphisms.

EXAMPLES:

```
sage: from sage.schemes.generic.morphism import SchemeMorphism
sage: f = SchemeMorphism(Spec(ZZ).Hom(Spec(ZZ)))
sage: type(f)
<class 'sage.schemes.generic.morphism.SchemeMorphism'>
```

change_ring(*R*, *check=True*)

Returns a new *SchemeMorphism_point* which is this point coerced to `R`.

If `check` is true, then the initialization checks are performed.

INPUT:

- `R` – ring or morphism.
- `check` – Boolean

OUTPUT: *SchemeMorphism_point*

EXAMPLES:

```
sage: P.<x,y,z> = ProjectiveSpace(ZZ, 2)
sage: X = P.subscheme(x^2 - y^2)
sage: X(23,23,1).change_ring(GF(13))
(10 : 10 : 1)
```

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: P(-2/3,1).change_ring(CC)                                    #␣
↪needs sage.rings.real_mpfr
(-0.666666666666667 : 1.00000000000000)
```

```
sage: P.<x,y> = ProjectiveSpace(ZZ, 1)
sage: P(152,113).change_ring(Zp(5))                                #␣
↪needs sage.rings.padics
(2 + 5^2 + 5^3 + O(5^20) : 3 + 2*5 + 4*5^2 + O(5^20))
```

```
sage: # needs sage.rings.number_field
sage: K.<v> = QuadraticField(-7)
sage: O = K.maximal_order()
sage: P.<x,y> = ProjectiveSpace(O, 1)
sage: H = End(P)
sage: F = H([x^2 + O(v)*y^2, y^2])
sage: F.change_ring(K).change_ring(K.embeddings(QQbar)[0])
Scheme endomorphism of Projective Space of dimension 1
 over Algebraic Field
  Defn: Defined on coordinates by sending (x : y) to
        (x^2 + (-2.645751311064591?*I)*y^2 : y^2)
```

```
sage: # needs sage.rings.number_field
sage: R.<x> = PolynomialRing(QQ)
sage: K.<a> = NumberField(x^2 - x + 1)
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: Q = P([a + 1, 1])
sage: emb = K.embeddings(QQbar)
sage: Q.change_ring(emb[0])
(1.500000000000000? - 0.866025403784439?*I : 1)
sage: Q.change_ring(emb[1])
(1.500000000000000? + 0.866025403784439?*I : 1)
```

```
sage: # needs sage.rings.number_field
sage: K.<v> = QuadraticField(2)
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: Q = P([v,1])
sage: Q.change_ring(K.embeddings(QQbar)[0])
(-1.414213562373095? : 1)
```

```
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: f = x^6 - 2
sage: L.<b> = NumberField(f, embedding=f.roots(QQbar)[1][0])
sage: A.<x,y> = AffineSpace(L, 2)
sage: P = A([b,1])
sage: P.change_ring(QQbar)
(1.122462048309373?, 1)
```

scheme()

Return the scheme whose point is represented.

OUTPUT:

A scheme.

EXAMPLES:

```
sage: A = AffineSpace(2, QQ)
sage: a = A(1,2)
sage: a.scheme()
Affine Space of dimension 2 over Rational Field
```

specialization(*D=None*, *phi=None*, *ambient=None*)

Specialization of this point.

Given a family of points defined over a polynomial ring. A specialization is a particular member of that family. The specialization can be specified either by a dictionary or a `SpecializationMorphism`.

INPUT:

- `D` – dictionary (optional)
- `phi` – SpecializationMorphism (optional)
- `ambient` – ambient space of specialized point (optional)

OUTPUT: *SchemeMorphism_polynomial*

EXAMPLES:

```
sage: R.<c> = PolynomialRing(QQ)
sage: P.<x,y> = ProjectiveSpace(R, 1)
sage: Q = P([c,1])
sage: Q.specialization({c: 1})
(1 : 1)
```

```
sage: R.<a,b> = PolynomialRing(QQ)
sage: P.<x,y> = ProjectiveSpace(R, 1)
sage: Q = P([a^2 + 2*a*b + 34, 1])
sage: from sage.rings.polynomial.flatten import SpecializationMorphism
sage: phi = SpecializationMorphism(P.coordinate_ring(), {a: 2, b: -1})
sage: T = Q.specialization(phi=phi); T
(34 : 1)
sage: Q2 = P([a,1])
sage: T2 = Q2.specialization(phi=phi)
sage: T2.codomain() is T.codomain()
True
sage: T3 = Q2.specialization(phi=phi, ambient=T.codomain())
sage: T3.codomain() is T.codomain()
True
```

```
sage: R.<c> = PolynomialRing(QQ)
sage: P.<x,y> = ProjectiveSpace(R, 1)
sage: X = P.subscheme([x - c*y])
sage: Q = X([c, 1])
sage: Q2 = Q.specialization({c:2}); Q2
(2 : 1)
sage: Q2.codomain()
Closed subscheme of Projective Space of dimension 1 over Rational Field
 defined by: x - 2*y
```

```
sage: R.<l> = PolynomialRing(QQ)
sage: S.<k,j> = PolynomialRing(R)
sage: K.<a,b,c,d> = S[]
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: H = End(P)
sage: Q = P([a^2, b^2])
```

(continues on next page)

(continued from previous page)

```
sage: Q.specialization({a: 2})
(4 : b^2)
```

class sage.schemes.generic.morphism.**SchemeMorphism_polynomial**(*parent*, *polys*, *check=True*)

Bases: *SchemeMorphism*

A morphism of schemes determined by polynomials that define what the morphism does on points in the ambient space.

INPUT:

- `parent` – Hom-set whose domain and codomain are affine or projective schemes.
- `polys` – a list/tuple/iterable of polynomials defining the scheme morphism.
- `check` – boolean (optional, default:`True`). Whether to check the input for consistency.

EXAMPLES:

An example involving the affine plane:

```
sage: R.<x,y> = QQ[]
sage: A2 = AffineSpace(R)
sage: H = A2.Hom(A2)
sage: f = H([x - y, x*y])
sage: f([0, 1])
(-1, 0)
```

An example involving the projective line:

```
sage: R.<x,y> = QQ[]
sage: P1 = ProjectiveSpace(R)
sage: H = P1.Hom(P1)
sage: f = H([x^2 + y^2, x*y])
sage: f([0, 1])
(1 : 0)
```

Some checks are performed to make sure the given polynomials define a morphism:

```
sage: f = H([exp(x),exp(y)])                                                #␣
↪needs sage.symbolic
Traceback (most recent call last):
...
TypeError: polys (=[e^x, e^y]) must be elements of Multivariate
Polynomial Ring in x, y over Rational Field
```

change_ring(*R*, *check=True*)

Returns a new *SchemeMorphism_polynomial* which is this map coerced to `R`.

If `check` is `True`, then the initialization checks are performed.

INPUT:

- `R` – ring or morphism.
- `check` – Boolean

OUTPUT:

- A new *SchemeMorphism_polynomial* which is this map coerced to R.

EXAMPLES:

```
sage: P.<x,y> = ProjectiveSpace(ZZ, 1)
sage: H = Hom(P, P)
sage: f = H([3*x^2, y^2])
sage: f.change_ring(GF(3))
Scheme endomorphism of Projective Space of dimension 1 over Finite Field of␣
↪size 3
  Defn: Defined on coordinates by sending (x : y) to (0 : y^2)
```

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: H = Hom(P, P)
sage: f = H([5/2*x^3 + 3*x*y^2 - y^3, 3*z^3 + y*x^2, x^3 - z^3])
sage: f.change_ring(GF(3))
Scheme endomorphism of Projective Space of dimension 2 over Finite Field of␣
↪size 3
  Defn: Defined on coordinates by sending (x : y : z) to
        (x^3 - y^3 : x^2*y : x^3 - z^3)
```

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: X = P.subscheme([5*x^2 - y^2])
sage: H = Hom(X, X)
sage: f = H([x, y])
sage: f.change_ring(GF(3))
Scheme endomorphism of Closed subscheme of Projective Space of dimension 1
over Finite Field of size 3 defined by: -x^2 - y^2
  Defn: Defined on coordinates by sending (x : y) to (x : y)
```

Check that github issue #16834 is fixed:

```
sage: # needs sage.rings.real_mpfr
sage: A.<x,y,z> = AffineSpace(RR, 3)
sage: h = Hom(A, A)
sage: f = h([x^2 + 1.5, y^3, z^5 - 2.0])
sage: f.change_ring(CC)
Scheme endomorphism of Affine Space of dimension 3 over
 Complex Field with 53 bits of precision
  Defn: Defined on coordinates by sending (x, y, z) to
        (x^2 + 1.50000000000000, y^3, z^5 - 2.00000000000000)
```

```
sage: A.<x,y> = AffineSpace(ZZ, 2)
sage: B.<u,v> = ProjectiveSpace(QQ, 1)
sage: h = Hom(A,B)
sage: f = h([x^2, y^2])
sage: f.change_ring(QQ)
Scheme morphism:
  From: Affine Space of dimension 2 over Rational Field
  To:   Projective Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (x, y) to (x^2 : y^2)
```

```
sage: A.<x,y> = AffineSpace(QQ, 2)
sage: H = Hom(A, A)
sage: f = H([3*x^2/y, y^2/x])
sage: f.change_ring(RR)                                                    #␣
↪needs sage.rings.real_mpfr
Scheme endomorphism of Affine Space of dimension 2 over Real Field with
53 bits of precision
  Defn: Defined on coordinates by sending (x, y) to
        (3.00000000000000*x^2/y, y^2/x)
```

```
sage: # needs sage.rings.number_field
sage: R.<x> = PolynomialRing(QQ)
sage: K.<a> = NumberField(x^3 - x + 1)
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: H = End(P)
sage: f = H([x^2 + a*x*y + a^2*y^2, y^2])
sage: emb = K.embeddings(QQbar)
sage: f.change_ring(emb[0])
Scheme endomorphism of Projective Space of dimension 1
 over Algebraic Field
   Defn: Defined on coordinates by sending (x : y) to
         (x^2 + (-1.324717957244746?)*x*y + 1.754877666246693?*y^2 : y^2)
sage: f.change_ring(emb[1])
Scheme endomorphism of Projective Space of dimension 1
 over Algebraic Field
   Defn: Defined on coordinates by sending (x : y) to
         (x^2 + (0.662358978622373?0? - 0.562279512062301?3?*I)*x*y
          + (0.1225611668766537? - 0.744861766619745?*I)*y^2 : y^2)
```

```
sage: # needs sage.rings.number_field sage.symbolic
sage: K.<v> = QuadraticField(2, embedding=QQbar(sqrt(2)))
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: H = End(P)
sage: f = H([x^2 + v*y^2, y^2])
sage: f.change_ring(QQbar)
Scheme endomorphism of Projective Space of dimension 1
 over Algebraic Field
  Defn: Defined on coordinates by sending (x : y) to
        (x^2 + 1.414213562373095?*y^2 : y^2)
```

```
sage: # needs sage.rings.number_field sage.symbolic
sage: from sage.misc.verbose import set_verbose
sage: set_verbose(-1)
sage: K.<w> = QuadraticField(2, embedding=QQbar(-sqrt(2)))
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: X = P.subscheme(x - y)
sage: H = End(X)
sage: f = H([6*x^2 + 2*x*y + 16*y^2, -w*x^2 - 4*x*y - 4*y^2])
sage: f.change_ring(QQbar)
Scheme endomorphism of Closed subscheme of Projective Space of dimension 1
 over Algebraic Field defined by: x - y
  Defn: Defined on coordinates by sending (x : y) to
```

(continues on next page)

(continued from previous page)

```
        (6*x^2 + 2*x*y + 16*y^2 : 1.4142135623730950?*x^2 + (-4)*x*y + (-4)*y^2)
```

```
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: f = x^6 - 2
sage: L.<b> = NumberField(f, embedding=f.roots(QQbar)[1][0])
sage: A.<x,y> = AffineSpace(L, 2)
sage: H = Hom(A, A)
sage: F = H([b*x/y, 1 + y])
sage: F.change_ring(QQbar)
Scheme endomorphism of Affine Space of dimension 2 over Algebraic Field
  Defn: Defined on coordinates by sending (x, y) to
        (1.122462048309373?*x/y, y + 1)
```

```
sage: # needs sage.rings.number_field
sage: K.<a> = QuadraticField(-1)
sage: A.<x,y> = AffineSpace(K, 2)
sage: H = End(A)
sage: phi = H([x/y, y])
sage: emb = K.embeddings(QQbar)[0]
sage: phi.change_ring(emb)
Scheme endomorphism of Affine Space of dimension 2 over Algebraic Field
  Defn: Defined on coordinates by sending (x, y) to (x/y, y)
```

coordinate_ring()

Return the coordinate ring of the ambient projective space.

OUTPUT: A multivariable polynomial ring over the base ring.

EXAMPLES:

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: H = Hom(P, P)
sage: f = H([3/5*x^2, 6*y^2])
sage: f.coordinate_ring()
Multivariate Polynomial Ring in x, y over Rational Field
```

```
sage: R.<t> = PolynomialRing(ZZ, 1)
sage: P.<x,y> = ProjectiveSpace(R, 1)
sage: H = Hom(P, P)
sage: f = H([3*x^2, y^2])
sage: f.coordinate_ring()
Multivariate Polynomial Ring in x, y over Multivariate Polynomial Ring
in t over Integer Ring
```

defining_polynomials()

Return the defining polynomials.

OUTPUT:

An immutable sequence of polynomials that defines this scheme morphism.

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: A.<x,y> = AffineSpace(R)
sage: H = A.Hom(A)
sage: H([x^3 + y, 1 - x - y]).defining_polynomials()
(x^3 + y, -x - y + 1)
```

specialization(*D=None*, *phi=None*, *homset=None*)

Specialization of this map.

Given a family of maps defined over a polynomial ring. A specialization is a particular member of that family. The specialization can be specified either by a dictionary or a `SpecializationMorphism`.

INPUT:

- `D` – dictionary (optional)
- `phi` – SpecializationMorphism (optional)
- `homset` – homset of specialized map (optional)

OUTPUT: *SchemeMorphism_polynomial*

EXAMPLES:

```
sage: R.<c> = PolynomialRing(QQ)
sage: P.<x,y> = ProjectiveSpace(R, 1)
sage: H = End(P)
sage: f = H([x^2 + c*y^2, y^2])
sage: f.specialization({c: 1})
Scheme endomorphism of Projective Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (x : y) to (x^2 + y^2 : y^2)
```

```
sage: R.<a,b> = PolynomialRing(QQ)
sage: P.<x,y> = ProjectiveSpace(R, 1)
sage: H = End(P)
sage: f = H([x^3 + a*x*y^2 + b*y^3, y^3])
sage: from sage.rings.polynomial.flatten import SpecializationMorphism
sage: phi = SpecializationMorphism(P.coordinate_ring(), {a: 2, b: -1})
sage: F = f.specialization(phi=phi); F
Scheme endomorphism of Projective Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (x : y) to
        (x^3 + 2*x*y^2 - y^3 : y^3)
sage: g = H([x^2 + a*y^2, y^2])
sage: G = g.specialization(phi=phi)
sage: G.parent() is F.parent()
True
sage: G = g.specialization(phi=phi, homset=F.parent())
sage: G.parent() is F.parent()
True
```

```
sage: R.<c> = PolynomialRing(QQ)
sage: P.<x,y> = ProjectiveSpace(R, 1)
sage: X = P.subscheme([x - c*y])
sage: H = End(X)
sage: f = H([x^2, c*y^2])
```

(continues on next page)

(continued from previous page)

```
sage: f.specialization({c: 2})
Scheme endomorphism of Closed subscheme of Projective Space of dimension 1
 over Rational Field defined by: x - 2*y
      Defn: Defined on coordinates by sending (x : y) to (x^2 : 2*y^2)
```

```
sage: R.<c> = QQ[]
sage: P.<x,y> = ProjectiveSpace(R, 1)
sage: f = DynamicalSystem_projective([x^2 + c*y^2, y^2], domain=P)
sage: F = f.dynatomic_polynomial(3)                                          #␣
↪needs sage.libs.pari
sage: g = F.specialization({c: 1}); g
x^6 + x^5*y + 4*x^4*y^2 + 3*x^3*y^3 + 7*x^2*y^4 + 4*x*y^5 + 5*y^6
sage: g == f.specialization({c:1}).dynatomic_polynomial(3)                   #␣
↪needs sage.libs.pari
True
```

```
sage: R1.<alpha, beta> = QQ[]
sage: A.<x> = AffineSpace(Frac(R1), 1)
sage: f = DynamicalSystem_affine([alpha/(x^2 + 1/alpha)/(x - 1/beta^2)])
sage: f.specialization({alpha: 5, beta: 10})
Dynamical System of Affine Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (x) to
          (5/(x^3 - 1/100*x^2 + 1/5*x - 1/500))
sage: f_5_10 = f.specialization({alpha: 5}).specialization({beta: 10})
sage: f_5_10 == f.specialization({alpha: 5, beta: 10})
True
```

class sage.schemes.generic.morphism.**SchemeMorphism_polynomial_id**(*X*)

Bases: *SchemeMorphism_id*, *SchemeMorphism_polynomial*

Return the identity morphism from X to itself.

INPUT:

- `X` – an affine or projective scheme

EXAMPLES:

```
sage: X = Spec(ZZ)
sage: X.identity_morphism()  # indirect doctest
Scheme endomorphism of Spectrum of Integer Ring
  Defn: Identity map
```

class sage.schemes.generic.morphism.**SchemeMorphism_spec**(*parent*, *phi*, *check=True*)

Bases: *SchemeMorphism*

Morphism of spectra of rings

INPUT:

- `parent` – Hom-set whose domain and codomain are affine schemes.
- `phi` – a ring morphism with matching domain and codomain.
- `check` – boolean (optional, default:`True`). Whether to check the input for consistency.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: phi = R.hom([QQ(7)]); phi
Ring morphism:
  From: Univariate Polynomial Ring in x over Rational Field
  To:   Rational Field
  Defn: x |--> 7

sage: X = Spec(QQ); Y = Spec(R)
sage: f = X.hom(phi); f
Affine Scheme morphism:
  From: Spectrum of Rational Field
  To:   Spectrum of Univariate Polynomial Ring in x over Rational Field
  Defn: Ring morphism:
          From: Univariate Polynomial Ring in x over Rational Field
          To:   Rational Field
          Defn: x |--> 7

sage: f.ring_homomorphism()
Ring morphism:
  From: Univariate Polynomial Ring in x over Rational Field
  To:   Rational Field
  Defn: x |--> 7
```

ring_homomorphism()

Return the underlying ring homomorphism.

OUTPUT:

A ring homomorphism.

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: phi = R.hom([QQ(7)])
sage: X = Spec(QQ); Y = Spec(R)
sage: f = X.hom(phi)
sage: f.ring_homomorphism()
Ring morphism:
  From: Univariate Polynomial Ring in x over Rational Field
  To:   Rational Field
  Defn: x |--> 7
```

class sage.schemes.generic.morphism.**SchemeMorphism_structure_map**(*parent*, *codomain=None*)

Bases: *SchemeMorphism*

The structure morphism

INPUT:

- `parent` – Hom-set with codomain equal to the base scheme of the domain.

EXAMPLES:

```
sage: Spec(ZZ).structure_morphism()    # indirect doctest
Scheme endomorphism of Spectrum of Integer Ring
  Defn: Structure map
```

sage.schemes.generic.morphism.**is_SchemeMorphism**(*f*)

Test whether `f` is a scheme morphism.

INPUT:

- `f` – anything.

OUTPUT:

Boolean. Return `True` if `f` is a scheme morphism or a point on an elliptic curve.

EXAMPLES:

```
sage: A.<x,y> = AffineSpace(QQ, 2); H = A.Hom(A)
sage: f = H([y, x^2 + y]); f
Scheme endomorphism of Affine Space of dimension 2 over Rational Field
  Defn: Defined on coordinates by sending (x, y) to (y, x^2 + y)
sage: from sage.schemes.generic.morphism import is_SchemeMorphism
sage: is_SchemeMorphism(f)
True
```

CHAPTER

ELEVEN

DIVISORS ON SCHEMES

AUTHORS:

- William Stein
- David Kohel
- David Joyner
- Volker Braun (2010-07-16): Documentation, doctests, coercion fixes, bugfixes.

EXAMPLES:

```
sage: x,y,z = ProjectiveSpace(2, GF(5), names='x,y,z').gens()
sage: C = Curve(y^2*z^7 - x^9 - x*z^8)
sage: pts = C.rational_points(); pts
[(0 : 0 : 1), (0 : 1 : 0), (2 : 2 : 1), (2 : 3 : 1), (3 : 1 : 1), (3 : 4 : 1)]
sage: D1 = C.divisor(pts[0])*3
sage: D2 = C.divisor(pts[1])
sage: D3 = 10*C.divisor(pts[5])
sage: D1.parent() is D2.parent()
True
sage: D = D1 - D2 + D3; D
3*(x, y) - (x, z) + 10*(x + 2*z, y + z)
sage: D[1][0]
-1
sage: D[1][1]
Ideal (x, z) of Multivariate Polynomial Ring in x, y, z over Finite Field of size 5
sage: C.divisor([(3, pts[0]), (-1, pts[1]), (10, pts[5])])
3*(x, y) - (x, z) + 10*(x + 2*z, y + z)
```

sage.schemes.generic.divisor.**CurvePointToIdeal**(*C*, *P*)

Return the vanishing ideal of a point on a curve.

EXAMPLES:

```
sage: x,y = AffineSpace(2, QQ, names='xy').gens()
sage: C = Curve(y^2 - x^9 - x)
sage: from sage.schemes.generic.divisor import CurvePointToIdeal
sage: CurvePointToIdeal(C, (0,0))
Ideal (x, y) of Multivariate Polynomial Ring in x, y over Rational Field
```

class sage.schemes.generic.divisor.**Divisor_curve**(*v*, *parent=None*, *check=True*, *reduce=True*)

Bases: *Divisor_generic*

For any curve C, use `C.divisor(v)` to construct a divisor on C. Here v can be either

- a rational point on C
- a list of rational points
- a list of 2-tuples (c, P), where c is an integer and P is a rational point.

TODO: Divisors shouldn't be restricted to rational points. The problem is that the divisor group is the formal sum of the group of points on the curve, and there's no implemented notion of point on E/K that has coordinates in L. This is what should be implemented, by adding an appropriate class to `schemes/generic/morphism.py`.

EXAMPLES:

```
sage: E = EllipticCurve([0, 0, 1, -1, 0])
sage: P = E(0,0)
sage: 10*P
(161/16 : -2065/64 : 1)
sage: D = E.divisor(P)
sage: D
(x, y)
sage: 10*D
10*(x, y)
sage: E.divisor([P, P])
2*(x, y)
sage: E.divisor([(3,P), (-4,5*P)])
3*(x, y) - 4*(x - 1/4*z, y + 5/8*z)
```

coefficient(*P*)

Return the coefficient of a given point P in this divisor.

EXAMPLES:

```
sage: x,y = AffineSpace(2, GF(5), names='xy').gens()
sage: C = Curve(y^2 - x^9 - x)
sage: pts = C.rational_points(); pts
[(0, 0), (2, 2), (2, 3), (3, 1), (3, 4)]
sage: D = C.divisor(pts[0])
sage: D.coefficient(pts[0])
1
sage: D = C.divisor([(3, pts[0]), (-1, pts[1])]); D
3*(x, y) - (x - 2, y - 2)
sage: D.coefficient(pts[0])
3
sage: D.coefficient(pts[1])
-1
```

support()

Return the support of this divisor, which is the set of points that occur in this divisor with nonzero coefficients.

EXAMPLES:

```
sage: x,y = AffineSpace(2, GF(5), names='xy').gens()
sage: C = Curve(y^2 - x^9 - x)
sage: pts = C.rational_points(); pts
[(0, 0), (2, 2), (2, 3), (3, 1), (3, 4)]
sage: D = C.divisor_group()([(3, pts[0]), (-1, pts[1])]); D
```

(continues on next page)

(continued from previous page)

```
3*(x, y) - (x - 2, y - 2)
sage: D.support()
[(0, 0), (2, 2)]
```

class sage.schemes.generic.divisor.**Divisor_generic**(*v*, *parent*, *check=True*, *reduce=True*)

Bases: FormalSum

A Divisor.

scheme()

Return the scheme that this divisor is on.

EXAMPLES:

```
sage: A.<x, y> = AffineSpace(2, GF(5))
sage: C = Curve(y^2 - x^9 - x)
sage: pts = C.rational_points(); pts
[(0, 0), (2, 2), (2, 3), (3, 1), (3, 4)]
sage: D = C.divisor(pts[0])*3 - C.divisor(pts[1]); D
3*(x, y) - (x - 2, y - 2)
sage: D.scheme()
Affine Plane Curve over Finite Field of size 5 defined by -x^9 + y^2 - x
```

sage.schemes.generic.divisor.**is_Divisor**(*x*)

Test whether `x` is an instance of *Divisor_generic*

INPUT:

- `x` – anything.

OUTPUT:

`True` or `False`.

EXAMPLES:

```
sage: from sage.schemes.generic.divisor import is_Divisor
sage: x,y = AffineSpace(2, GF(5), names='xy').gens()
sage: C = Curve(y^2 - x^9 - x)
sage: is_Divisor(C.divisor([]))
True
sage: is_Divisor("Ceci n'est pas un diviseur")
False
```

CHAPTER

TWELVE

DIVISOR GROUPS

AUTHORS:

- David Kohel (2006): Initial version
- Volker Braun (2010-07-16): Documentation, doctests, coercion fixes, bugfixes.

sage.schemes.generic.divisor_group.**DivisorGroup**(*scheme*, *base_ring=None*)

Return the group of divisors on the scheme.

INPUT:

- `scheme` – a scheme.
- `base_ring` – usually either $\mathbf{Z}$ (default) or $\mathbf{Q}$. The coefficient ring of the divisors. Not to be confused with the base ring of the scheme!

OUTPUT:

An instance of `DivisorGroup_generic`.

EXAMPLES:

```
sage: from sage.schemes.generic.divisor_group import DivisorGroup
sage: DivisorGroup(Spec(ZZ))
Group of ZZ-Divisors on Spectrum of Integer Ring
sage: DivisorGroup(Spec(ZZ), base_ring=QQ)
Group of QQ-Divisors on Spectrum of Integer Ring
```

class sage.schemes.generic.divisor_group.**DivisorGroup_curve**(*scheme*, *base_ring*)

Bases: *DivisorGroup_generic*

Special case of the group of divisors on a curve.

class sage.schemes.generic.divisor_group.**DivisorGroup_generic**(*scheme*, *base_ring*)

Bases: `FormalSums`

The divisor group on a variety.

base_extend(*R*)

EXAMPLES:

```
sage: from sage.schemes.generic.divisor_group import DivisorGroup
sage: DivisorGroup(Spec(ZZ), ZZ).base_extend(QQ)
Group of QQ-Divisors on Spectrum of Integer Ring
sage: DivisorGroup(Spec(ZZ), ZZ).base_extend(GF(7))
Group of (Finite Field of size 7)-Divisors on Spectrum of Integer Ring
```

Divisor groups are unique:

```
sage: A.<x, y> = AffineSpace(2, CC)                                          #␣
↪needs sage.rings.real_mpfr
sage: C = Curve(y^2 - x^9 - x)                                               #␣
↪needs sage.rings.real_mpfr sage.schemes
sage: DivisorGroup(C, ZZ).base_extend(QQ) is DivisorGroup(C, QQ)             #␣
↪needs sage.rings.real_mpfr sage.schemes
True
```

scheme()

Return the scheme supporting the divisors.

EXAMPLES:

```
sage: from sage.schemes.generic.divisor_group import DivisorGroup
sage: Div = DivisorGroup(Spec(ZZ))   # indirect test
sage: Div.scheme()
Spectrum of Integer Ring
```

sage.schemes.generic.divisor_group.**is_DivisorGroup**(*x*)

Return whether `x` is a *DivisorGroup_generic*.

INPUT:

- `x` – anything.

OUTPUT:

`True` or `False`.

EXAMPLES:

```
sage: from sage.schemes.generic.divisor_group import is_DivisorGroup, DivisorGroup
sage: Div = DivisorGroup(Spec(ZZ), base_ring=QQ)
sage: is_DivisorGroup(Div)
True
sage: is_DivisorGroup('not a divisor')
False
```

CHAPTER

THIRTEEN

AFFINE SCHEMES

13.1 Affine n space over a ring

sage.schemes.affine.affine_space.**AffineSpace**(*n*, *R=None*, *names=None*, *ambient_projective_space=None*, *default_embedding_index=None*)

Return affine space of dimension `n` over the ring `R`.

EXAMPLES:

The dimension and ring can be given in either order:

```
sage: AffineSpace(3, QQ, 'x')
Affine Space of dimension 3 over Rational Field
sage: AffineSpace(5, QQ, 'x')
Affine Space of dimension 5 over Rational Field
sage: A = AffineSpace(2, QQ, names='XY'); A
Affine Space of dimension 2 over Rational Field
sage: A.coordinate_ring()
Multivariate Polynomial Ring in X, Y over Rational Field
```

Use the divide operator for base extension:

```
sage: AffineSpace(5, names='x')/GF(17)
Affine Space of dimension 5 over Finite Field of size 17
```

The default base ring is $\mathbf{Z}$:

```
sage: AffineSpace(5, names='x')
Affine Space of dimension 5 over Integer Ring
```

There is also an affine space associated to each polynomial ring:

```
sage: R = GF(7)['x, y, z']
sage: A = AffineSpace(R); A
Affine Space of dimension 3 over Finite Field of size 7
sage: A.coordinate_ring() is R
True
```

class sage.schemes.affine.affine_space.**AffineSpace_field**(*n*, *R*, *names*, *ambient_projective_space*, *default_embedding_index*)

Bases: *AffineSpace_generic*

curve(*F*)

Return a curve defined by F in this affine space.

INPUT:

- F – a polynomial, or a list or tuple of polynomials in the coordinate ring of this affine space

EXAMPLES:

```
sage: A.<x,y,z> = AffineSpace(QQ, 3)
sage: A.curve([y - x^4, z - y^5])                                           #␣
↪needs sage.schemes
Affine Curve over Rational Field defined by -x^4 + y, -y^5 + z
```

line_through(*p*, *q*)

Return the line through p and q.

INPUT:

- p, q – distinct rational points of the affine space

EXAMPLES:

```
sage: # needs sage.libs.singular sage.schemes
sage: A3.<x,y,z> = AffineSpace(3, QQ)
sage: p1 = A3(1, 2, 3)
sage: p2 = A3(4, 5, 6)
sage: L = A3.line_through(p1, p2); L
Affine Curve over Rational Field defined by -1/6*x + 1/6*y - 1/6,
  -1/6*x + 1/6*z - 1/3, -1/6*y + 1/6*z - 1/6, -1/6*x + 1/3*y - 1/6*z
sage: L(p1)
(1, 2, 3)
sage: L(p2)
(4, 5, 6)
sage: A3.line_through(p1, p1)
Traceback (most recent call last):
...
ValueError: not distinct points
```

points_of_bounded_height(***kwds*)

Return an iterator of the points in this affine space of absolute height of at most the given bound.

Bound check is strict for the rational field. Requires this space to be affine space over a number field. Uses the Doyle-Krumm algorithm 4 (algorithm 5 for imaginary quadratic) for computing algebraic numbers up to a given height [DK2013].

The algorithm requires floating point arithmetic, so the user is allowed to specify the precision for such calculations. Additionally, due to floating point issues, points slightly larger than the bound may be returned. This can be controlled by lowering the tolerance.

INPUT:

kwds:

- bound - a real number
- tolerance - a rational number in (0,1] used in doyle-krumm algorithm-4
- precision - the precision to use for computing the elements of bounded height of number fields

OUTPUT:

- an iterator of points in self

EXAMPLES:

```
sage: A.<x,y> = AffineSpace(QQ, 2)
sage: list(A.points_of_bounded_height(bound=3))
[(0, 0), (1, 0), (-1, 0), (1/2, 0), (-1/2, 0), (2, 0), (-2, 0), (0, 1),
 (1, 1), (-1, 1), (1/2, 1), (-1/2, 1), (2, 1), (-2, 1), (0, -1), (1, -1),
 (-1, -1), (1/2, -1), (-1/2, -1), (2, -1), (-2, -1), (0, 1/2), (1, 1/2),
 (-1, 1/2), (1/2, 1/2), (-1/2, 1/2), (2, 1/2), (-2, 1/2), (0, -1/2), (1, -1/2),
 (-1, -1/2), (1/2, -1/2), (-1/2, -1/2), (2, -1/2), (-2, -1/2), (0, 2), (1, 2),
 (-1, 2), (1/2, 2), (-1/2, 2), (2, 2), (-2, 2), (0, -2), (1, -2), (-1, -2),
 (1/2, -2), (-1/2, -2), (2, -2), (-2, -2)]
```

```
sage: u = QQ['u'].0
sage: A.<x,y> = AffineSpace(NumberField(u^2 - 2, 'v'), 2)                    #␣
↪needs sage.rings.number_field
sage: len(list(A.points_of_bounded_height(bound=2, tolerance=0.1)))          #␣
↪needs sage.rings.number_field
529
```

translation(*p*, *q=None*)

Return the automorphism of the affine space translating `p` to the origin.

If `q` is given, the automorphism translates `p` to `q`.

INPUT:

- `p` – a rational point
- `q` – (default: `None`) a rational point

EXAMPLES:

```
sage: A.<x,y,z> = AffineSpace(QQ, 3)
sage: p = A(1,2,3)
sage: q = A(4,5,6)
sage: A.translation(p, q)
Scheme endomorphism of Affine Space of dimension 3 over Rational Field
  Defn: Defined on coordinates by sending (x, y, z) to
        (x + 3, y + 3, z + 3)
sage: phi = A.translation(p)
sage: psi = A.translation(A.origin(), q)
sage: psi * phi
Scheme endomorphism of Affine Space of dimension 3 over Rational Field
  Defn: Defined on coordinates by sending (x, y, z) to
        (x + 3, y + 3, z + 3)
sage: psi * phi == A.translation(p, q)
True
```

weil_restriction()

Compute the Weil restriction of this affine space over some extension field.

If the field is a finite field, then this computes the Weil restriction to the prime subfield.

OUTPUT: Affine space of dimension `d * self.dimension_relative()` over the base field of `self.base_ring()`.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: K.<w> = NumberField(x^5 - 2)
sage: AK.<x,y> = AffineSpace(K, 2)
sage: AK.weil_restriction()
Affine Space of dimension 10 over Rational Field
sage: R.<x> = K[]
sage: L.<v> = K.extension(x^2 + 1)
sage: AL.<x,y> = AffineSpace(L, 2)
sage: AL.weil_restriction()
Affine Space of dimension 4 over Number Field in w
 with defining polynomial x^5 - 2
```

class sage.schemes.affine.affine_space.**AffineSpace_finite_field**(*n*, *R*, *names*, *ambient_projective_space*, *default_embedding_index*)

Bases: *AffineSpace_field*

class sage.schemes.affine.affine_space.**AffineSpace_generic**(*n*, *R*, *names*, *ambient_projective_space*, *default_embedding_index*)

Bases: *AmbientSpace*, *AffineScheme*

Affine space of dimension n over the ring R.

EXAMPLES:

```
sage: X.<x,y,z> = AffineSpace(3, QQ)
sage: X.base_scheme()
Spectrum of Rational Field
sage: X.base_ring()
Rational Field
sage: X.category()
Category of schemes over Rational Field
sage: X.structure_morphism()
Scheme morphism:
  From: Affine Space of dimension 3 over Rational Field
  To:   Spectrum of Rational Field
  Defn: Structure map
```

Loading and saving:

```
sage: loads(X.dumps()) == X
True
```

We create several other examples of affine spaces:

```
sage: AffineSpace(5, PolynomialRing(QQ, 'z'), 'Z')
Affine Space of dimension 5 over Univariate Polynomial Ring in z over Rational Field

sage: AffineSpace(RealField(), 3, 'Z')                                          #␣
↪needs sage.rings.real_mpfr
Affine Space of dimension 3 over Real Field with 53 bits of precision
```

(continues on next page)

(continued from previous page)

```
sage: AffineSpace(Qp(7), 2, 'x')                                                #␣
↪needs sage.rings.padics
Affine Space of dimension 2 over 7-adic Field with capped relative precision 20
```

Even 0-dimensional affine spaces are supported:

```
sage: AffineSpace(0)
Affine Space of dimension 0 over Integer Ring
```

change_ring(*R*)

Return an affine space over ring R and otherwise the same as this space.

INPUT:

- R – commutative ring or morphism.

OUTPUT: An affine space over R.

Note: There is no need to have any relation between R and the base ring of this space, if you want to have such a relation, use `self.base_extend(R)` instead.

EXAMPLES:

```
sage: A.<x,y,z> = AffineSpace(3, ZZ)
sage: AQ = A.change_ring(QQ); AQ
Affine Space of dimension 3 over Rational Field
sage: AQ.change_ring(GF(5))
Affine Space of dimension 3 over Finite Field of size 5
```

```
sage: K.<w> = QuadraticField(5)                                                 #␣
↪needs sage.rings.number_field
sage: A = AffineSpace(K, 2, 't')                                                #␣
↪needs sage.rings.number_field
sage: A.change_ring(K.embeddings(CC)[1])                                        #␣
↪needs sage.rings.number_field
Affine Space of dimension 2 over Complex Field with 53 bits of precision
```

chebyshev_polynomial(*n*, *kind='first'*, *monic=False*)

Generate an endomorphism of this affine line by a Chebyshev polynomial.

Chebyshev polynomials are a sequence of recursively defined orthogonal polynomials. Chebyshev of the first kind are defined as $T_0(x) = 1$, $T_1(x) = x$, and $T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$. Chebyshev of the second kind are defined as $U_0(x) = 1$, $U_1(x) = 2x$, and $U_{n+1}(x) = 2xU_n(x) - U_{n-1}(x)$.

INPUT:

- n – a non-negative integer.
- kind – first or second specifying which kind of chebyshev the user would like to generate. Defaults to first.
- monic – True or False specifying if the polynomial defining the system should be monic or not. Defaults to False.

OUTPUT: DynamicalSystem_affine

EXAMPLES:

```
sage: A.<x> = AffineSpace(QQ, 1)
sage: A.chebyshev_polynomial(5, 'first')                                    #␣
↪needs sage.schemes
Dynamical System of Affine Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (x) to (16*x^5 - 20*x^3 + 5*x)
```

```
sage: A.<x> = AffineSpace(QQ, 1)
sage: A.chebyshev_polynomial(3, 'second')                                   #␣
↪needs sage.schemes
Dynamical System of Affine Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (x) to (8*x^3 - 4*x)
```

```
sage: A.<x> = AffineSpace(QQ, 1)
sage: A.chebyshev_polynomial(3, 2)                                          #␣
↪needs sage.schemes
Traceback (most recent call last):
...
ValueError: keyword 'kind' must have a value of either 'first' or 'second'
```

```
sage: A.<x> = AffineSpace(QQ, 1)
sage: A.chebyshev_polynomial(-4, 'second')
Traceback (most recent call last):
...
ValueError: first parameter 'n' must be a non-negative integer
```

```
sage: A = AffineSpace(QQ, 2, 'x')
sage: A.chebyshev_polynomial(2)
Traceback (most recent call last):
...
TypeError: affine space must be of dimension 1
```

```
sage: A.<x> = AffineSpace(QQ, 1)
sage: A.chebyshev_polynomial(7, monic=True)                                 #␣
↪needs sage.schemes
Dynamical System of Affine Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (x) to (x^7 - 7*x^5 + 14*x^3 - 7*x)
```

```
sage: F.<t> = FunctionField(QQ)
sage: A.<x> = AffineSpace(F, 1)
sage: A.chebyshev_polynomial(4, monic=True)                                 #␣
↪needs sage.schemes
Dynamical System of Affine Space of dimension 1
over Rational function field in t over Rational Field
  Defn: Defined on coordinates by sending (x) to (x^4 + (-4)*x^2 + 2)
```

coordinate_ring()

Return the coordinate ring of this scheme, if defined.

EXAMPLES:

```
sage: R = AffineSpace(2, GF(9,'alpha'), 'z').coordinate_ring(); R           #␣
↪needs sage.rings.finite_rings
```

(continues on next page)

(continued from previous page)

```
Multivariate Polynomial Ring in z0, z1 over Finite Field in alpha of size 3^2
sage: AffineSpace(3, R, 'x').coordinate_ring()                              #␣
↪needs sage.rings.finite_rings
Multivariate Polynomial Ring in x0, x1, x2 over Multivariate Polynomial Ring
in z0, z1 over Finite Field in alpha of size 3^2
```

ngens()

Return the number of generators of self, i.e. the number of variables in the coordinate ring of self.

EXAMPLES:

```
sage: AffineSpace(3, QQ).ngens()
3
sage: AffineSpace(7, ZZ).ngens()
7
```

origin()

Return the rational point at the origin of this affine space.

EXAMPLES:

```
sage: A.<x,y,z> = AffineSpace(QQ, 3)
sage: A.origin()
(0, 0, 0)
sage: _ == A(0,0,0)
True
```

projective_embedding(*i=None*, *PP=None*)

Return a morphism from this space into an ambient projective space of the same dimension.

INPUT:

- `i` – integer (default: dimension of self = last coordinate) determines which projective embedding to compute. The embedding is that which has a 1 in the i-th coordinate, numbered from 0.
- `PP` – (default: None) ambient projective space, i.e., codomain of morphism; this is constructed if it is not given.

EXAMPLES:

```
sage: AA = AffineSpace(2, QQ, 'x')
sage: pi = AA.projective_embedding(0); pi
Scheme morphism:
  From: Affine Space of dimension 2 over Rational Field
  To:   Projective Space of dimension 2 over Rational Field
  Defn: Defined on coordinates by sending (x0, x1) to (1 : x0 : x1)
sage: z = AA(3, 4)
sage: pi(z)
(1/4 : 3/4 : 1)
sage: pi(AA(0,2))
(1/2 : 0 : 1)
sage: pi = AA.projective_embedding(1); pi
Scheme morphism:
  From: Affine Space of dimension 2 over Rational Field
  To:   Projective Space of dimension 2 over Rational Field
```

(continues on next page)

(continued from previous page)

```
  Defn: Defined on coordinates by sending (x0, x1) to (x0 : 1 : x1)
sage: pi(z)
(3/4 : 1/4 : 1)
sage: pi = AA.projective_embedding(2)
sage: pi(z)
(3 : 4 : 1)
```

```
sage: A.<x,y> = AffineSpace(ZZ, 2)
sage: A.projective_embedding(2).codomain().affine_patch(2) == A
True
```

rational_points(*F=None*)

Return the list of F-rational points on the affine space self, where F is a given finite field, or the base ring of self.

EXAMPLES:

```
sage: A = AffineSpace(1, GF(3))
sage: A.rational_points()
[(0), (1), (2)]
sage: A.rational_points(GF(3^2, 'b'))                                    #␣
↪needs sage.rings.finite_rings
[(0), (b), (b + 1), (2*b + 1), (2), (2*b), (2*b + 2), (b + 2), (1)]

sage: AffineSpace(2, ZZ).rational_points(GF(2))
[(0, 0), (0, 1), (1, 0), (1, 1)]
```

subscheme(*X*, ***kwds*)

Return the closed subscheme defined by X.

INPUT:

- X - a list or tuple of equations.

EXAMPLES:

```
sage: A.<x,y> = AffineSpace(QQ, 2)
sage: X = A.subscheme([x, y^2, x*y^2]); X
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x,
  y^2,
  x*y^2
```

```
sage: # needs sage.libs.singular
sage: X.defining_polynomials ()
(x, y^2, x*y^2)
sage: I = X.defining_ideal(); I
Ideal (x, y^2, x*y^2) of Multivariate Polynomial Ring in x, y over Rational␣
↪Field
sage: I.groebner_basis()
[y^2, x]
sage: X.dimension()
0
```

(continues on next page)

(continued from previous page)

```
sage: X.base_ring()
Rational Field
sage: X.base_scheme()
Spectrum of Rational Field
sage: X.structure_morphism()
Scheme morphism:
  From: Closed subscheme of Affine Space of dimension 2 over Rational Field
        defined by: x, y^2, x*y^2
  To:   Spectrum of Rational Field
  Defn: Structure map
sage: X.dimension()
0
```

sage.schemes.affine.affine_space.**is_AffineSpace**(*x*)

Return `True` if `x` is an affine space.

EXAMPLES:

```
sage: from sage.schemes.affine.affine_space import is_AffineSpace
sage: is_AffineSpace(AffineSpace(5, names='x'))
True
sage: is_AffineSpace(AffineSpace(5, GF(9, 'alpha'), names='x'))                #␣
↪needs sage.rings.finite_rings
True
sage: is_AffineSpace(Spec(ZZ))
False
```

13.2 Morphisms on affine schemes

This module implements morphisms from affine schemes. A morphism from an affine scheme to an affine scheme is determined by rational functions that define what the morphism does on points in the ambient affine space. A morphism from an affine scheme to a projective scheme is determined by homogeneous polynomials.

EXAMPLES:

```
sage: A2.<x,y> = AffineSpace(QQ, 2)
sage: P2.<x0,x1,x2> = ProjectiveSpace(QQ, 2)
sage: A2.hom([x, x + y], A2)
Scheme endomorphism of Affine Space of dimension 2 over Rational Field
  Defn: Defined on coordinates by sending (x, y) to (x, x + y)
sage: A2.hom([1, x, x + y], P2)
Scheme morphism:
  From: Affine Space of dimension 2 over Rational Field
  To:   Projective Space of dimension 2 over Rational Field
  Defn: Defined on coordinates by sending (x, y) to (1 : x : x + y)
```

AUTHORS:

- David Kohel, William Stein: initial version
- Volker Braun (2011-08-08): renamed classes, more documentation, misc cleanups
- Ben Hutz (2013-03): iteration functionality and new directory structure for affine/projective

- Kwankyu Lee (2020-02): added indeterminacy_locus() and image()

class sage.schemes.affine.affine_morphism.**SchemeMorphism_polynomial_affine_space**(*parent*, *polys*, *check=True*)

Bases: *SchemeMorphism_polynomial*

A morphism of schemes determined by rational functions.

EXAMPLES:

```
sage: RA.<x,y> = QQ[]
sage: A2 = AffineSpace(RA)
sage: RP.<u,v,w> = QQ[]
sage: P2 = ProjectiveSpace(RP)
sage: H = A2.Hom(P2)
sage: f = H([x, y, 1])
sage: f
Scheme morphism:
  From: Affine Space of dimension 2 over Rational Field
  To:   Projective Space of dimension 2 over Rational Field
  Defn: Defined on coordinates by sending (x, y) to (x : y : 1)
```

as_dynamical_system()

Return this endomorphism as a DynamicalSystem_affine.

OUTPUT:

- DynamicalSystem_affine

EXAMPLES:

```
sage: A.<x,y,z> = AffineSpace(ZZ, 3)
sage: H = End(A)
sage: f = H([x^2, y^2, z^2])
sage: type(f.as_dynamical_system())                                    #␣
↪needs sage.schemes
<class 'sage.dynamics.arithmetic_dynamics.affine_ds.DynamicalSystem_affine'>
```

```
sage: A.<x,y> = AffineSpace(ZZ, 2)
sage: H = End(A)
sage: f = H([x^2 - y^2, y^2])
sage: type(f.as_dynamical_system())                                    #␣
↪needs sage.schemes
<class 'sage.dynamics.arithmetic_dynamics.affine_ds.DynamicalSystem_affine'>
```

```
sage: A.<x> = AffineSpace(GF(5), 1)
sage: H = End(A)
sage: f = H([x^2])
sage: type(f.as_dynamical_system())                                    #␣
↪needs sage.schemes
<class 'sage.dynamics.arithmetic_dynamics.affine_ds.DynamicalSystem_affine_
↪finite_field'>
```

```
sage: P.<x,y> = AffineSpace(RR, 2)
sage: f = DynamicalSystem([x^2 + y^2, y^2], P)                          #␣
↪needs sage.schemes
sage: g = f.as_dynamical_system()                                       #␣
↪needs sage.schemes
sage: g is f                                                            #␣
↪needs sage.schemes
True
```

degree()

Return the degree of the affine morphism.

EXAMPLES:

```
sage: R.<x> = AffineSpace(QQ, 1)
sage: H = Hom(R, R)
sage: f = H([x^7])
sage: f.degree()
7
```

```
sage: R.<x,y,z> = AffineSpace(QQ, 3)
sage: H = Hom(R, R)
sage: f = H([x^3, y^2 + 5, z^4 + y])
sage: f.degree()
4
```

global_height(*prec=None*)

Take the height of the homogenization, and return the global height of the coefficients as a projective point.

INPUT:

- `prec` – desired floating point precision (default: default RealField precision).

OUTPUT: A real number.

EXAMPLES:

```
sage: A.<x> = AffineSpace(QQ, 1)
sage: H = Hom(A, A)
sage: f = H([1/1331*x^2 + 4000])
sage: f.global_height()                                                 #␣
↪needs sage.symbolic
15.4877354584971
```

```
sage: # needs sage.rings.number_field
sage: R.<x> = PolynomialRing(QQ)
sage: k.<w> = NumberField(x^2 + 5)
sage: A.<x,y> = AffineSpace(k, 2)
sage: H = Hom(A, A)
sage: f = H([13*w*x^2 + 4*y, 1/w*y^2])
sage: f.global_height(prec=2)
4.0
```

```
sage: A.<x> = AffineSpace(ZZ, 1)
sage: H = Hom(A, A)
sage: f = H([7*x^2 + 1513])
sage: f.global_height()                                                    #␣
↪needs sage.symbolic
7.32184971378836
```

```
sage: A.<x> = AffineSpace(QQ, 1)
sage: B.<y,z> = AffineSpace(QQ, 2)
sage: H = Hom(A, B)
sage: f = H([1/3*x^2 + 10, 7*x^3])
sage: f.global_height()                                                    #␣
↪needs sage.symbolic
3.40119738166216
```

```
sage: P.<x,y> = AffineSpace(QQ, 2)
sage: A.<z> = AffineSpace(QQ, 1)
sage: H = Hom(P, A)
sage: f = H([1/1331*x^2 + 4000*y])
sage: f.global_height()                                                    #␣
↪needs sage.symbolic
15.4877354584971
```

homogenize(*n*)

Return the homogenization of this map.

If it's domain is a subscheme, the domain of the homogenized map is the projective embedding of the domain. The domain and codomain can be homogenized at different coordinates: `n[0]` for the domain and `n[1]` for the codomain.

INPUT:

- `n` – a tuple of nonnegative integers. If `n` is an integer, then the two values of the tuple are assumed to be the same

OUTPUT: a morphism from the projective embedding of the domain of this map

EXAMPLES:

```
sage: A.<x,y> = AffineSpace(ZZ, 2)
sage: H = Hom(A, A)
sage: f = H([(x^2-2)/x^5, y^2])
sage: f.homogenize(2)
Scheme endomorphism of Projective Space of dimension 2 over Integer Ring
  Defn: Defined on coordinates by sending (x0 : x1 : x2) to
        (x0^2*x2^5 - 2*x2^7 : x0^5*x1^2 : x0^5*x2^2)
```

```
sage: # needs sage.rings.real_mpfr
sage: A.<x,y> = AffineSpace(CC, 2)
sage: H = Hom(A, A)
sage: f = H([(x^2-2)/(x*y), y^2 - x])
sage: f.homogenize((2, 0))
Scheme endomorphism of Projective Space of dimension 2
 over Complex Field with 53 bits of precision
```

(continues on next page)

(continued from previous page)

```
  Defn: Defined on coordinates by sending (x0 : x1 : x2) to
        (x0*x1*x2^2 : x0^2*x2^2 + (-2.000000000000000)*x2^4 : x0*x1^3 - x0^
↪2*x1*x2)
```

```
sage: A.<x,y> = AffineSpace(ZZ, 2)
sage: X = A.subscheme([x - y^2])
sage: H = Hom(X, X)
sage: f = H([9*y^2, 3*y])
sage: f.homogenize(2)                                                    #␣
↪needs sage.libs.singular
Scheme endomorphism of Closed subscheme of Projective Space
 of dimension 2 over Integer Ring defined by: x1^2 - x0*x2
  Defn: Defined on coordinates by sending (x0 : x1 : x2) to
        (9*x1^2 : 3*x1*x2 : x2^2)
```

```
sage: R.<t> = PolynomialRing(ZZ)
sage: A.<x,y> = AffineSpace(R, 2)
sage: H = Hom(A, A)
sage: f = H([(x^2-2)/y, y^2 - x])
sage: f.homogenize((2, 0))
Scheme endomorphism of Projective Space of dimension 2
 over Univariate Polynomial Ring in t over Integer Ring
  Defn: Defined on coordinates by sending (x0 : x1 : x2) to
        (x1*x2^2 : x0^2*x2 + (-2)*x2^3 : x1^3 - x0*x1*x2)
```

```
sage: A.<x> = AffineSpace(QQ, 1)
sage: H = End(A)
sage: f = H([x^2 - 1])
sage: f.homogenize((1, 0))
Scheme endomorphism of Projective Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (x0 : x1) to
        (x1^2 : x0^2 - x1^2)
```

```
sage: # needs sage.rings.number_field
sage: R.<a> = PolynomialRing(QQbar)
sage: A.<x,y> = AffineSpace(R, 2)
sage: H = End(A)
sage: f = H([QQbar(sqrt(2))*x*y, a*x^2])                                 #␣
↪needs sage.symbolic
sage: f.homogenize(2)                                                    #␣
↪needs sage.libs.singular sage.symbolic
Scheme endomorphism of Projective Space of dimension 2
 over Univariate Polynomial Ring in a over Algebraic Field
  Defn: Defined on coordinates by sending (x0 : x1 : x2) to
        (1.414213562373095?*x0*x1 : a*x0^2 : x2^2)
```

```
sage: P.<x,y,z> = AffineSpace(QQ, 3)
sage: H = End(P)
sage: f = H([x^2 - 2*x*y + z*x, z^2 -y^2 , 5*z*y])
sage: f.homogenize(2).dehomogenize(2) == f
True
```

```
sage: K.<c> = FunctionField(QQ)
sage: A.<x> = AffineSpace(K, 1)
sage: f = Hom(A, A)([x^2 + c])
sage: f.homogenize(1)
Scheme endomorphism of Projective Space of dimension 1
 over Rational function field in c over Rational Field
  Defn: Defined on coordinates by sending (x0 : x1) to
        (x0^2 + c*x1^2 : x1^2)
```

```
sage: # needs sage.rings.number_field
sage: A.<z> = AffineSpace(QQbar, 1)
sage: H = End(A)
sage: f = H([2*z / (z^2 + 2*z + 3)])
sage: f.homogenize(1)
Scheme endomorphism of Projective Space of dimension 1
 over Algebraic Field
  Defn: Defined on coordinates by sending (x0 : x1) to
        (x0*x1 : 1/2*x0^2 + x0*x1 + 3/2*x1^2)
```

```
sage: # needs sage.rings.number_field
sage: R.<c,d> = QQbar[]
sage: A.<x> = AffineSpace(R, 1)
sage: H = Hom(A, A)
sage: F = H([d*x^2 + c])
sage: F.homogenize(1)
Scheme endomorphism of Projective Space of dimension 1
 over Multivariate Polynomial Ring in c, d over Algebraic Field
  Defn: Defined on coordinates by sending (x0 : x1) to
        (d*x0^2 + c*x1^2 : x1^2)
```

jacobian()

Return the Jacobian matrix of partial derivative of this map.

The (i, j) entry of the Jacobian matrix is the partial derivative `diff(functions[i], variables[j])`.

OUTPUT:

- matrix with coordinates in the coordinate ring of the map.

EXAMPLES:

```
sage: A.<z> = AffineSpace(QQ, 1)
sage: H = End(A)
sage: f = H([z^2 - 3/4])
sage: f.jacobian()                                                          #␣
↪needs sage.modules
[2*z]
```

```
sage: A.<x,y> = AffineSpace(QQ, 2)
sage: H = End(A)
sage: f = H([x^3 - 25*x + 12*y, 5*y^2*x - 53*y + 24])
sage: f.jacobian()                                                          #␣
↪needs sage.modules
```

(continues on next page)

(continued from previous page)

```
[ 3*x^2 - 25          12]
[      5*y^2 10*x*y - 53]
```

```
sage: A.<x,y> = AffineSpace(ZZ, 2)
sage: H = End(A)
sage: f = H([(x^2 - x*y)/(1+y), (5+y)/(2+x)])
sage: f.jacobian()                                                          #␣
↪needs sage.modules
[         (2*x - y)/(y + 1) (-x^2 - x)/(y^2 + 2*y + 1)]
[  (-y - 5)/(x^2 + 4*x + 4)                   1/(x + 2)]
```

local_height(*v*, *prec=None*)

Return the maximum of the local heights of the coefficients in any of the coordinate functions of this map.

INPUT:

- `v` – a prime or prime ideal of the base ring.
- `prec` – desired floating point precision (default: default RealField precision).

OUTPUT:

- a real number.

EXAMPLES:

```
sage: P.<x,y> = AffineSpace(QQ, 2)
sage: H = Hom(P, P)
sage: f = H([1/1331*x^2 + 1/4000*y^2, 210*x*y])
sage: f.local_height(1331)                                                  #␣
↪needs sage.rings.real_mpfr
7.19368581839511
```

```
sage: P.<x,y,z> = AffineSpace(QQ, 3)
sage: H = Hom(P, P)
sage: f = H([4*x^2 + 3/100*y^2, 8/210*x*y, 1/10000*z^2])
sage: f.local_height(2)                                                     #␣
↪needs sage.rings.real_mpfr
2.77258872223978
```

```
sage: P.<x,y,z> = AffineSpace(QQ, 3)
sage: H = Hom(P, P)
sage: f = H([4*x^2 + 3/100*y^2, 8/210*x*y, 1/10000*z^2])
sage: f.local_height(2, prec=2)                                             #␣
↪needs sage.rings.real_mpfr
3.0
```

```
sage: # needs sage.rings.number_field
sage: R.<z> = PolynomialRing(QQ)
sage: K.<w> = NumberField(z^2 - 2)
sage: P.<x,y> = AffineSpace(K, 2)
sage: H = Hom(P, P)
sage: f = H([2*x^2 + w/3*y^2, 1/w*y^2])
```

(continues on next page)

(continued from previous page)

```
sage: f.local_height(K.ideal(3))
1.09861228866811
```

local_height_arch(*i*, *prec=None*)

Return the maximum of the local height at the `i`-th infinite place of the coefficients in any of the coordinate functions of this map.

INPUT:

- `i` – an integer.
- `prec` – desired floating point precision (default: default RealField precision).

OUTPUT:

- a real number.

EXAMPLES:

```
sage: P.<x,y> = AffineSpace(QQ, 2)
sage: H = Hom(P, P)
sage: f = H([1/1331*x^2 + 1/4000*y^2, 210*x*y]);
sage: f.local_height_arch(0)                                                #␣
↪needs sage.rings.real_mpfr
5.34710753071747
```

```
sage: P.<x,y> = AffineSpace(QQ, 2)
sage: H = Hom(P, P)
sage: f = H([1/1331*x^2 + 1/4000*y^2, 210*x*y]);
sage: f.local_height_arch(0, prec=5)                                        #␣
↪needs sage.rings.real_mpfr
5.2
```

```
sage: # needs sage.rings.number_field
sage: R.<z> = PolynomialRing(QQ)
sage: K.<w> = NumberField(z^2 - 2)
sage: P.<x,y> = AffineSpace(K, 2)
sage: H = Hom(P, P)
sage: f = H([2*x^2 + w/3*y^2, 1/w*y^2])
sage: f.local_height_arch(1)
0.69314718055994530941723212145820
```

class sage.schemes.affine.affine_morphism.**SchemeMorphism_polynomial_affine_space_field**(*parent*, *polys*, *check=True*)

Bases: *SchemeMorphism_polynomial_affine_space*

image()

Return the scheme-theoretic image of the morphism.

OUTPUT: a subscheme of the ambient space of the codomain

EXAMPLES:

```
sage: # needs sage.libs.singular
sage: A1.<w> = AffineSpace(QQ, 1)
sage: A2.<x,y> = AffineSpace(QQ, 2)
sage: f = A2.hom([x + y], A1)
sage: f.image()
Closed subscheme of Affine Space of dimension 1 over Rational Field defined by:
  (no polynomials)
sage: f = A2.hom([x, x], A2)
sage: f.image()
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x - y
sage: f = A2.hom([x^2, x^3], A2)
sage: f.image()
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x^3 - y^2
sage: P2.<x0,x1,x2> = ProjectiveSpace(QQ, 2)
sage: f = A2.hom([x, x^2, x^3], P2)
sage: f.image()
Closed subscheme of Projective Space of dimension 2 over Rational Field defined␣
↪by:
  x1^2 - x0*x2
```

indeterminacy_locus()

Return the indeterminacy locus of this map as a rational map on the domain.

The indeterminacy locus is the intersection of all the base indeterminacy locuses of maps that define the same rational map as by this map.

OUTPUT: a subscheme of the domain of the map

EXAMPLES:

```
sage: A.<x,y> = AffineSpace(QQ, 2)
sage: H = End(A)
sage: f = H([x - y, x^2 - y^2])
sage: f.indeterminacy_locus()                                                  #␣
↪needs sage.libs.singular
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  1
```

```
sage: A.<x,y> = AffineSpace(QQ, 2)
sage: f = A.hom([x, x/y], A)
sage: f.indeterminacy_locus()                                                  #␣
↪needs sage.libs.singular
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  y
```

indeterminacy_points(*F=None*)

Return the points in the indeterminacy locus of this map.

If the dimension of the indeterminacy locus is not zero, an error is raised.

INPUT:

- `F` – a field; if not given, the base ring of the domain is assumed

OUTPUT: indeterminacy points of the map defined over F

EXAMPLES:

```
sage: A.<x,y> = AffineSpace(QQ, 2)
sage: H = End(A)
sage: f = H([x - y, x^2 - y^2])
sage: f.indeterminacy_points()                                                  #␣
↪needs sage.libs.singular
[]
```

```
sage: A2.<x,y> = AffineSpace(QQ, 2)
sage: P2.<x0,x1,x2> = ProjectiveSpace(QQ, 2)
sage: f = A2.hom([x*y, y, x], P2)
sage: f.indeterminacy_points()                                                  #␣
↪needs sage.libs.singular
[(0, 0)]
```

reduce_base_field()

Return this map defined over the field of definition of the coefficients.

The base field of the map could be strictly larger than the field where all of the coefficients are defined. This function reduces the base field to the minimal possible. This can be done when the base ring is a number field, QQbar, a finite field, or algebraic closure of a finite field.

OUTPUT: a scheme morphism

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: K.<t> = GF(5^4)
sage: A.<x> = AffineSpace(K, 1)
sage: A2.<a,b> = AffineSpace(K, 2)
sage: H = End(A)
sage: H2 = Hom(A, A2)
sage: H3 = Hom(A2, A)
sage: f = H([x^2 + 2*(t^3 + t^2 + t + 3)])
sage: f.reduce_base_field()
Scheme endomorphism of Affine Space of dimension 1
 over Finite Field in t2 of size 5^2
  Defn: Defined on coordinates by sending (x) to (x^2 + (2*t2))
sage: f2 = H2([x^2 + 4, 2*x])
sage: f2.reduce_base_field()
Scheme morphism:
  From: Affine Space of dimension 1 over Finite Field of size 5
  To:   Affine Space of dimension 2 over Finite Field of size 5
  Defn: Defined on coordinates by sending (x) to (x^2 - 1, 2*x)
sage: f3 = H3([a^2 + t*b])
sage: f3.reduce_base_field()
Scheme morphism:
  From: Affine Space of dimension 2 over Finite Field in t of size 5^4
  To:   Affine Space of dimension 1 over Finite Field in t of size 5^4
  Defn: Defined on coordinates by sending (a, b) to (a^2 + t*b)
```

```
sage: # needs sage.rings.number_field
sage: K.<v> = CyclotomicField(4)
sage: A.<x> = AffineSpace(K, 1)
sage: H = End(A)
sage: f = H([x^2 + v])
sage: g = f.reduce_base_field(); g
Scheme endomorphism of Affine Space of dimension 1
 over Cyclotomic Field of order 4 and degree 2
  Defn: Defined on coordinates by sending (x) to (x^2 + v)
sage: g.base_ring() is K
True
```

```
sage: # needs sage.rings.number_field
sage: A.<x> = AffineSpace(QQbar, 1)
sage: H = End(A)
sage: f = H([(QQbar(sqrt(2))*x^2 + 1/QQbar(sqrt(3))) / (5*x)])              #␣
↪needs sage.symbolic
sage: f.reduce_base_field()                                                 #␣
↪needs sage.symbolic
Scheme endomorphism of Affine Space of dimension 1 over Number Field in a
 with defining polynomial y^4 - 4*y^2 + 1 with a = ...?
  Defn: Defined on coordinates by sending (x) to
        (((a^3 - 3*a)*x^2 + (-1/3*a^2 + 2/3))/(5*x))
```

```
sage: # needs sage.rings.number_field
sage: R.<x> = PolynomialRing(QQ)
sage: A.<x> = AffineSpace(QQbar, 1)
sage: H = End(A)
sage: f = H([QQbar(3^(1/3))*x^2 + QQbar(sqrt(-2))])                         #␣
↪needs sage.symbolic
sage: f.reduce_base_field()                                                 #␣
↪needs sage.symbolic
Scheme endomorphism of Affine Space of dimension 1 over Number
Field in a with defining polynomial y^6 + 6*y^4 - 6*y^3 + 12*y^2 + 36*y + 17
 with a = 1.442249570307409? + 1.414213562373095?*I
  Defn: Defined on coordinates by sending (x) to
        ((-48/269*a^5 + 27/269*a^4 - 320/269*a^3 + 468/269*a^2 - 772/269*a
        - 1092/269)*x^2 + (48/269*a^5 - 27/269*a^4 + 320/269*a^3 - 468/269*a^2
        + 1041/269*a + 1092/269))
```

```
sage: # needs sage.rings.number_field
sage: R.<x> = PolynomialRing(QQ)
sage: K.<a> = NumberField(x^3 - x + 1,
....:                     embedding=(x^3 + x + 1).roots(ring=CC)[0][0])
sage: A.<x> = AffineSpace(K, 1)
sage: A2.<u,v> = AffineSpace(K, 2)
sage: H = Hom(A, A2)
sage: f = H([x^2 + a*x + 3, 5*x])
sage: f.reduce_base_field()
Scheme morphism:
  From: Affine Space of dimension 1 over Number Field in a with
        defining polynomial x^3 - x + 1 with a = -1.324717957244746?
```

(continues on next page)

(continued from previous page)

```
  To:   Affine Space of dimension 2 over Number Field in a with
        defining polynomial x^3 - x + 1 with a = -1.324717957244746?
  Defn: Defined on coordinates by sending (x) to (x^2 + a*x + 3, 5*x)
```

```
sage: # needs sage.rings.number_field
sage: K.<v> = QuadraticField(2)
sage: A.<x> = AffineSpace(K, 1)
sage: H = End(A)
sage: f = H([3*x^2 + x + 1])
sage: f.reduce_base_field()
Scheme endomorphism of Affine Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (x) to (3*x^2 + x + 1)
```

```
sage: # needs sage.rings.finite_rings
sage: K.<t> = GF(5^6)
sage: A.<x> = AffineSpace(K, 1)
sage: H = End(A)
sage: f = H([x^2 + x*(t^3 + 2*t^2 + 4*t) + (t^5 + 3*t^4 + t^2 + 4*t)])
sage: f.reduce_base_field()
Scheme endomorphism of Affine Space of dimension 1
 over Finite Field in t of size 5^6
  Defn: Defined on coordinates by sending (x) to
        (x^2 + (t^3 + 2*t^2 - t)*x + (t^5 - 2*t^4 + t^2 - t))
```

weil_restriction()

Compute the Weil restriction of this morphism over some extension field.

If the field is a finite field, then this computes the Weil restriction to the prime subfield.

A Weil restriction of scalars - denoted $Res_{L/k}$ - is a functor which, for any finite extension of fields L/k and any algebraic variety X over L, produces another corresponding variety $Res_{L/k}(X)$, defined over k. It is useful for reducing questions about varieties over large fields to questions about more complicated varieties over smaller fields. Since it is a functor it also applied to morphisms. In particular, the functor applied to a morphism gives the equivalent morphism from the Weil restriction of the domain to the Weil restriction of the codomain.

OUTPUT: Scheme morphism on the Weil restrictions of the domain
and codomain of the map.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: K.<v> = QuadraticField(5)
sage: A.<x,y> = AffineSpace(K, 2)
sage: H = End(A)
sage: f = H([x^2 - y^2, y^2])
sage: f.weil_restriction()                                                  #␣
↪needs sage.libs.singular
Scheme endomorphism of Affine Space of dimension 4 over Rational Field
  Defn: Defined on coordinates by sending (z0, z1, z2, z3) to
        (z0^2 + 5*z1^2 - z2^2 - 5*z3^2, 2*z0*z1 - 2*z2*z3, z2^2 + 5*z3^2,␣
↪2*z2*z3)
```

```
sage: # needs sage.rings.number_field
sage: K.<v> = QuadraticField(5)
sage: PS.<x,y> = AffineSpace(K, 2)
sage: H = Hom(PS, PS)
sage: f = H([x, y])
sage: F = f.weil_restriction()
sage: P = PS(2, 1)
sage: Q = P.weil_restriction()
sage: f(P).weil_restriction() == F(Q)                                       #␣
↪needs sage.libs.singular
True
```

class sage.schemes.affine.affine_morphism.**SchemeMorphism_polynomial_affine_space_finite_field**(*parent*, *polys*, *check=True*)

Bases: *SchemeMorphism_polynomial_affine_space_field*

class sage.schemes.affine.affine_morphism.**SchemeMorphism_polynomial_affine_subscheme_field**(*parent*, *polys*, *check=True*)

Bases: *SchemeMorphism_polynomial_affine_space_field*

Morphisms from subschemes of affine spaces defined over fields.

image()

Return the scheme-theoretic image of the morphism.

OUTPUT: a subscheme of the ambient space of the codomain

EXAMPLES:

```
sage: A1.<w> = AffineSpace(QQ, 1)
sage: A2.<x,y> = AffineSpace(QQ, 2)
sage: X = A2.subscheme(0)
sage: f = X.hom([x + y], A1)
sage: f.image()                                                             #␣
↪needs sage.libs.singular
Closed subscheme of Affine Space of dimension 1 over Rational Field defined by:
  (no polynomials)
```

```
sage: A2.<x,y> = AffineSpace(QQ, 2)
sage: X = A2.subscheme([x*y^2 - y^3 - 1])
sage: f = X.hom([y, y/x], A2)
sage: f.image()                                                             #␣
↪needs sage.libs.singular
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  -x^3*y + x^3 - y
```

indeterminacy_locus()

Return the indeterminacy locus of this map.

The map defines a rational map on the domain. The output is the subscheme of the domain on which the rational map is not defined by any representative of the rational map. See *representatives()*.

EXAMPLES:

```
sage: A2.<x1,x2> = AffineSpace(QQ, 2)
sage: X = A2.subscheme(0)
sage: A1.<x> = AffineSpace(QQ, 1)
sage: f = X.hom([x1/x2], A1)
sage: f.indeterminacy_locus()                                                #␣
↪needs sage.libs.singular
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x2
```

```
sage: A2.<x1,x2> = AffineSpace(QQ, 2)
sage: X = A2.subscheme(0)
sage: P1.<a,b> = ProjectiveSpace(QQ, 1)
sage: f = X.hom([x1,x2], P1)
sage: L = f.indeterminacy_locus()                                            #␣
↪needs sage.libs.singular
sage: L.rational_points()                                                    #␣
↪needs sage.libs.singular
[(0, 0)]
```

```
sage: A2.<x,y> = AffineSpace(QQ, 2)
sage: X = A2.subscheme([x^2 - y^2 - y])
sage: A1.<a> = AffineSpace(QQ, 1)
sage: f = X.hom([x/y], A1)
sage: f.indeterminacy_locus()                                                #␣
↪needs sage.libs.singular
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  y,
  x
```

```
sage: A3.<x,y,z> = AffineSpace(QQ, 3)
sage: X = A3.subscheme(x^2 - y*z - x)
sage: A2.<a,b> = AffineSpace(QQ, 2)
sage: f = X.hom([y, y/x], A2)
sage: L = f.indeterminacy_locus(); L                                         #␣
↪needs sage.libs.singular
Closed subscheme of Affine Space of dimension 3 over Rational Field defined by:
  x,
  y*z
sage: L.dimension()                                                          #␣
↪needs sage.libs.singular
1
```

is_morphism()

Return True if the map is defined everywhere on the domain.

EXAMPLES:

```
sage: P2.<x,y,z> = ProjectiveSpace(QQ,2)
sage: P1.<a,b> = ProjectiveSpace(QQ,1)
sage: X = P2.subscheme([x^2 - y^2 - y*z])
sage: f = X.hom([x,y], P1)
sage: f.is_morphism()                                                        #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.libs.singular
True
```

representatives()

Return all maps representing the same rational map as by this map.

EXAMPLES:

```
sage: A2.<x,y> = AffineSpace(QQ, 2)
sage: X = A2.subscheme(0)
sage: f = X.hom([x, x/y], A2)
sage: f.representatives()                                                   #␣
↪needs sage.libs.singular
[Scheme morphism:
   From: Closed subscheme of Affine Space of dimension 2 over Rational Field
         defined by: 0
   To:   Affine Space of dimension 2 over Rational Field
   Defn: Defined on coordinates by sending (x, y) to (x, x/y)]
```

```
sage: # needs sage.libs.singular
sage: A2.<x,y> = AffineSpace(QQ, 2)
sage: A1.<a> = AffineSpace(QQ, 1)
sage: X = A2.subscheme([x^2 - y^2 - y])
sage: f = X.hom([x/y], A1)
sage: f.representatives()
[Scheme morphism:
   From: Closed subscheme of Affine Space of dimension 2 over Rational Field
         defined by: x^2 - y^2 - y
   To:   Affine Space of dimension 1 over Rational Field
   Defn: Defined on coordinates by sending (x, y) to (x/y),
 Scheme morphism:
   From: Closed subscheme of Affine Space of dimension 2 over Rational Field
         defined by: x^2 - y^2 - y
   To:   Affine Space of dimension 1 over Rational Field
   Defn: Defined on coordinates by sending (x, y) to ((y + 1)/x)]
sage: g = _[1]
sage: g.representatives()
[Scheme morphism:
   From: Closed subscheme of Affine Space of dimension 2 over Rational Field
         defined by: x^2 - y^2 - y
   To:   Affine Space of dimension 1 over Rational Field
   Defn: Defined on coordinates by sending (x, y) to (x/y),
 Scheme morphism:
   From: Closed subscheme of Affine Space of dimension 2 over Rational Field
         defined by: x^2 - y^2 - y
   To:   Affine Space of dimension 1 over Rational Field
   Defn: Defined on coordinates by sending (x, y) to ((y + 1)/x)]
```

```
sage: A2.<x,y> = AffineSpace(QQ, 2)
sage: P1.<a,b> = ProjectiveSpace(QQ, 1)
sage: X = A2.subscheme([x^2 - y^2 - y])
sage: f = X.hom([x, y], P1)
```

(continues on next page)

(continued from previous page)

```
sage: f.representatives()                                                   #␣
↪needs sage.libs.singular
[Scheme morphism:
   From: Closed subscheme of Affine Space of dimension 2 over Rational Field
         defined by: x^2 - y^2 - y
   To:   Projective Space of dimension 1 over Rational Field
   Defn: Defined on coordinates by sending (x, y) to
         (x : y),
 Scheme morphism:
   From: Closed subscheme of Affine Space of dimension 2 over Rational Field
         defined by: x^2 - y^2 - y
   To:   Projective Space of dimension 1 over Rational Field
   Defn: Defined on coordinates by sending (x, y) to (y + 1 : x)]
```

13.3 Points on affine varieties

Scheme morphism for points on affine varieties.

AUTHORS:

- David Kohel, William Stein
- Volker Braun (2011-08-08): Renamed classes, more documentation, misc cleanups.
- Ben Hutz (2013)

class sage.schemes.affine.affine_point.**SchemeMorphism_point_affine**(*X*, *v*, *check=True*)

Bases: *SchemeMorphism_point*

A rational point on an affine scheme.

INPUT:

- `X` – a subscheme of an ambient affine space over a ring R
- `v` – a list/tuple/iterable of coordinates in R
- `check` – boolean (optional, default:`True`); whether to check the input for consistency

EXAMPLES:

```
sage: A = AffineSpace(2, QQ)
sage: A(1, 2)
(1, 2)
```

global_height(*prec=None*)

Returns the logarithmic height of the point.

INPUT:

- `prec` – desired floating point precision (default: default RealField precision).

OUTPUT:

- a real number.

EXAMPLES:

```
sage: P.<x,y> = AffineSpace(QQ, 2)
sage: Q = P(41, 1/12)
sage: Q.global_height()                                                    #␣
↪needs sage.rings.real_mpfr
3.71357206670431
```

```
sage: P = AffineSpace(ZZ, 4, 'x')
sage: Q = P(3, 17, -51, 5)
sage: Q.global_height()                                                    #␣
↪needs sage.rings.real_mpfr
3.93182563272433
```

```
sage: R.<x> = PolynomialRing(QQ)
sage: k.<w> = NumberField(x^2 + 5)                                         #␣
↪needs sage.rings.number_field
sage: A = AffineSpace(k, 2, 'z')                                           #␣
↪needs sage.rings.number_field
sage: A([3, 5*w + 1]).global_height(prec=100)                              #␣
↪needs sage.rings.number_field sage.rings.real_mpfr
2.4181409534757389986565376694
```

Todo: P-adic heights.

homogenize(*n*)

Return the homogenization of the point at the `nth` coordinate.

INPUT:

- `n` – integer between 0 and dimension of the map, inclusive.

OUTPUT:

- A point in the projectivization of the codomain of the map .

EXAMPLES:

```
sage: A.<x,y> = AffineSpace(ZZ, 2)
sage: Q = A(2, 3)
sage: Q.homogenize(2).dehomogenize(2) == Q
True

::

sage: A.<x,y> = AffineSpace(QQ, 2)
sage: Q = A(2, 3)
sage: P = A(0, 1)
sage: Q.homogenize(2).codomain() == P.homogenize(2).codomain()
True
```

class sage.schemes.affine.affine_point.**SchemeMorphism_point_affine_field**(*X*, *v*, *check=True*)

Bases: *SchemeMorphism_point_affine*

intersection_multiplicity(*X*)

Return the intersection multiplicity of the codomain of this point and `X` at this point.

This uses the intersection_multiplicity implementations for projective/affine subschemes. This point must be a point on an affine subscheme.

INPUT:

- `X` – a subscheme in the same ambient space as that of the codomain of this point.

OUTPUT: Integer.

EXAMPLES:

```
sage: # needs sage.libs.singular
sage: A.<x,y> = AffineSpace(GF(17), 2)
sage: X = A.subscheme([y^2 - x^3 + 2*x^2 - x])
sage: Y = A.subscheme([y - 2*x + 2])
sage: Q1 = Y([1,0])
sage: Q1.intersection_multiplicity(X)
2
sage: Q2 = X([4,6])
sage: Q2.intersection_multiplicity(Y)
1
```

```
sage: A.<x,y,z,w> = AffineSpace(QQ, 4)
sage: X = A.subscheme([x^2 - y*z^2, z - 2*w^2])
sage: Q = A([2,1,2,-1])
sage: Q.intersection_multiplicity(X)
Traceback (most recent call last):
...
TypeError: this point must be a point on an affine subscheme
```

multiplicity()

Return the multiplicity of this point on its codomain.

Uses the subscheme multiplicity implementation. This point must be a point on an affine subscheme.

OUTPUT: an integer.

EXAMPLES:

```
sage: A.<x,y,z> = AffineSpace(QQ, 3)
sage: X = A.subscheme([y^2 - x^7*z])
sage: Q1 = X([1,1,1])
sage: Q1.multiplicity()                                                    #␣
↪needs sage.libs.singular
1
sage: Q2 = X([0,0,2])
sage: Q2.multiplicity()                                                    #␣
↪needs sage.libs.singular
2
```

weil_restriction()

Compute the Weil restriction of this point over some extension field.

If the field is a finite field, then this computes the Weil restriction to the prime subfield.

A Weil restriction of scalars - denoted $Res_{L/k}$ - is a functor which, for any finite extension of fields L/k and any algebraic variety X over L, produces another corresponding variety $Res_{L/k}(X)$, defined over k. It is useful for reducing questions about varieties over large fields to questions about more complicated varieties

over smaller fields. This functor applied to a point gives the equivalent point on the Weil restriction of its codomain.

OUTPUT: Scheme point on the Weil restriction of the codomain of this point.

EXAMPLES:

```
sage: # needs sage.libs.singular sage.rings.finite_rings
sage: A.<x,y,z> = AffineSpace(GF(5^3, 't'), 3)
sage: X = A.subscheme([y^2 - x*z, z^2 + y])
sage: Y = X.weil_restriction()
sage: P = X([1, -1, 1])
sage: Q = P.weil_restriction();Q
(1, 0, 0, 4, 0, 0, 1, 0, 0)
sage: Q.codomain() == Y
True
```

```
sage: # needs sage.libs.singular sage.rings.number_field
sage: R.<x> = QQ[]
sage: K.<w> = NumberField(x^5 - 2)
sage: R.<x> = K[]
sage: L.<v> = K.extension(x^2 + w)
sage: A.<x,y> = AffineSpace(L, 2)
sage: P = A([w^3 - v, 1 + w + w*v])
sage: P.weil_restriction()
(w^3, -1, w + 1, w)
```

class sage.schemes.affine.affine_point.**SchemeMorphism_point_affine_finite_field**(*X*, *v*, *check=True*)

Bases: *SchemeMorphism_point_affine_field*

13.4 Subschemes of affine space

AUTHORS:

- David Kohel, William Stein (2005): initial version
- Ben Hutz (2013): affine subschemes

class sage.schemes.affine.affine_subscheme.**AlgebraicScheme_subscheme_affine**(*A*, *polynomials*, *embedding_center=None*, *embedding_codomain=None*, *embedding_images=None*)

Bases: *AlgebraicScheme_subscheme*

An algebraic subscheme of affine space.

INPUT:

- `A` – ambient affine space
- `polynomials` – single polynomial, ideal or iterable of defining polynomials

EXAMPLES:

```
sage: A3.<x, y, z> = AffineSpace(QQ, 3)
sage: A3.subscheme([x^2 - y*z])
Closed subscheme of Affine Space of dimension 3 over Rational Field defined by:
  x^2 - y*z
```

dimension()

Return the dimension of the affine algebraic subscheme.

EXAMPLES:

```
sage: # needs sage.libs.singular
sage: A.<x,y> = AffineSpace(2, QQ)
sage: A.subscheme([]).dimension()
2
sage: A.subscheme([x]).dimension()
1
sage: A.subscheme([x^5]).dimension()
1
sage: A.subscheme([x^2 + y^2 - 1]).dimension()
1
sage: A.subscheme([x*(x-1), y*(y-1)]).dimension()
0
```

Something less obvious:

```
sage: A.<x,y,z,w> = AffineSpace(4, QQ)
sage: X = A.subscheme([x^2, x^2*y^2 + z^2, z^2 - w^2, 10*x^2 + w^2 - z^2])
sage: X
Closed subscheme of Affine Space of dimension 4 over Rational Field defined by:
  x^2,
  x^2*y^2 + z^2,
  z^2 - w^2,
  10*x^2 - z^2 + w^2
sage: X.dimension()                                                          #␣
↪needs sage.libs.singular
1
```

intersection_multiplicity(*X*, *P*)

Return the intersection multiplicity of this subscheme and the subscheme `X` at the point `P`.

The intersection of this subscheme with `X` must be proper, that is $\mathrm{codim}(self \cap X) = \mathrm{codim}(self) + \mathrm{codim}(X)$, and must also be finite. We use Serre's Tor formula to compute the intersection multiplicity. If I, J are the defining ideals of `self`, `X`, respectively, then this is $\sum_{i=0}^{\infty}(-1)^i \mathrm{length}(\mathrm{Tor}^i_{\mathcal{O}_{A,p}}(\mathcal{O}_{A,p}/I, \mathcal{O}_{A,p}/J))$ where A is the affine ambient space of these subschemes.

INPUT:

- `X` – subscheme in the same ambient space as this subscheme.
- `P` – a point in the intersection of this subscheme with `X`.

OUTPUT: An integer.

EXAMPLES:

```
sage: A.<x,y> = AffineSpace(QQ, 2)
sage: C = Curve([y^2 - x^3 - x^2], A)                                    #␣
↪needs sage.libs.singular
sage: D = Curve([y^2 + x^3], A)                                          #␣
↪needs sage.libs.singular
sage: Q = A([0,0])
sage: C.intersection_multiplicity(D, Q)                                  #␣
↪needs sage.libs.singular
4
```

```
sage: # needs sage.rings.number_field
sage: R.<a> = QQ[]
sage: K.<b> = NumberField(a^6 - 3*a^5 + 5*a^4 - 5*a^3 + 5*a^2 - 3*a + 1)
sage: A.<x,y,z,w> = AffineSpace(K, 4)
sage: X = A.subscheme([x*y, y*z + 7, w^3 - x^3])
sage: Y = A.subscheme([x - z^3 + z + 1])
sage: Q = A([0,
....:        -7*b^5 + 21*b^4 - 28*b^3 + 21*b^2 - 21*b + 14,
....:        -b^5 + 2*b^4 - 3*b^3 + 2*b^2 - 2*b,
....:        0])
sage: X.intersection_multiplicity(Y, Q)                                  #␣
↪needs sage.libs.singular
3
```

```
sage: A.<x,y,z> = AffineSpace(QQ, 3)
sage: X = A.subscheme([z^2 - 1])
sage: Y = A.subscheme([z - 1, y - x^2])
sage: Q = A([1,1,1])
sage: X.intersection_multiplicity(Y, Q)                                  #␣
↪needs sage.libs.singular
Traceback (most recent call last):
...
TypeError: the intersection of this subscheme and (=Closed subscheme of Affine␣
↪Space of dimension 3
over Rational Field defined by: z - 1, -x^2 + y) must be proper and finite
```

```
sage: A.<x,y,z,w,t> = AffineSpace(QQ, 5)
sage: X = A.subscheme([x*y, t^2*w, w^3*z])
sage: Y = A.subscheme([y*w + z])
sage: Q = A([0,0,0,0,0])
sage: X.intersection_multiplicity(Y, Q)                                  #␣
↪needs sage.libs.singular
Traceback (most recent call last):
...
TypeError: the intersection of this subscheme and (=Closed subscheme of Affine␣
↪Space of dimension 5
over Rational Field defined by: y*w + z) must be proper and finite
```

is_smooth(*point=None*)

Test whether the algebraic subscheme is smooth.

INPUT:

- point – A point or None (default). The point to test smoothness at.

OUTPUT:

Boolean. If no point was specified, returns whether the algebraic subscheme is smooth everywhere. Otherwise, smoothness at the specified point is tested.

EXAMPLES:

```
sage: A2.<x,y> = AffineSpace(2, QQ)
sage: cuspidal_curve = A2.subscheme([y^2 - x^3])
sage: cuspidal_curve
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  -x^3 + y^2
sage: smooth_point = cuspidal_curve.point([1,1])
sage: smooth_point in cuspidal_curve
True
sage: singular_point = cuspidal_curve.point([0,0])
sage: singular_point in cuspidal_curve
True
sage: cuspidal_curve.is_smooth(smooth_point)                                #␣
↪needs sage.libs.singular
True
sage: cuspidal_curve.is_smooth(singular_point)                              #␣
↪needs sage.libs.singular
False
sage: cuspidal_curve.is_smooth()                                            #␣
↪needs sage.libs.singular
False
```

multiplicity(*P*)

Return the multiplicity of P on this subscheme.

This is computed as the multiplicity of the local ring of this subscheme corresponding to P. This subscheme must be defined over a field. An error is raised if P is not a point on this subscheme.

INPUT:

- P – a point on this subscheme.

OUTPUT:

An integer.

EXAMPLES:

```
sage: A.<x,y,z,w> = AffineSpace(QQ, 4)
sage: X = A.subscheme([z*y - x^7, w - 2*z])
sage: Q1 = A([1,1/3,3,6])
sage: X.multiplicity(Q1)                                                    #␣
↪needs sage.libs.singular
1
sage: Q2 = A([0,0,0,0])
sage: X.multiplicity(Q2)                                                    #␣
↪needs sage.libs.singular
2
```

```
sage: A.<x,y,z,w,v> = AffineSpace(GF(23), 5)
sage: C = A.curve([x^8 - y, y^7 - z, z^3 - 1, w^5 - v^3])                    #␣
↪needs sage.libs.singular sage.schemes
sage: Q = A([22,1,1,0,0])
sage: C.multiplicity(Q)                                                      #␣
↪needs sage.libs.singular sage.schemes
3
```

```
sage: # needs sage.rings.number_field
sage: K.<a> = QuadraticField(-1)
sage: A.<x,y,z,w,t> = AffineSpace(K, 5)
sage: X = A.subscheme([y^7 - x^2*z^5 + z^3*t^8 - x^2*y^4*z - t^8])
sage: Q1 = A([1,1,0,1,-1])
sage: X.multiplicity(Q1)                                                     #␣
↪needs sage.libs.singular
1
sage: Q2 = A([0,0,0,-a,0])
sage: X.multiplicity(Q2)                                                     #␣
↪needs sage.libs.singular
7
```

Check that github issue #27479 is fixed:

```
sage: A1.<x> = AffineSpace(QQ, 1)
sage: X = A1.subscheme([x^1789 + x])
sage: Q = X([0])
sage: X.multiplicity(Q)                                                      #␣
↪needs sage.libs.singular
1
```

projective_closure(*i=None*, *PP=None*)

Return the projective closure of this affine subscheme.

INPUT:

- `i` – (default: None) determines the embedding to use to compute the projective closure of this affine subscheme. The embedding used is the one which has a 1 in the i-th coordinate, numbered from 0.
- `PP` – (default: None) ambient projective space, i.e., ambient space of codomain of morphism; this is constructed if it is not given

OUTPUT: a projective subscheme

EXAMPLES:

```
sage: A.<x,y,z,w> = AffineSpace(QQ, 4)
sage: X = A.subscheme([x^2 - y, x*y - z, y^2 - w,
....:                  x*z - w, y*z - x*w, z^2 - y*w])
sage: X.projective_closure()                                                 #␣
↪needs sage.libs.singular
Closed subscheme of Projective Space of dimension 4 over Rational Field
 defined by:
  x0^2 - x1*x4,
  x0*x1 - x2*x4,
  x1^2 - x3*x4,
```

(continues on next page)

(continued from previous page)

```
  x0*x2 - x3*x4,
  x1*x2 - x0*x3,
  x2^2 - x1*x3
```

```
sage: A.<x,y,z> = AffineSpace(QQ, 3)
sage: P.<a,b,c,d> = ProjectiveSpace(QQ, 3)
sage: X = A.subscheme([z - x^2 - y^2])
sage: X.projective_closure(1, P).ambient_space() == P                          #␣
↪needs sage.libs.singular
True
```

projective_embedding(*i=None*, *PP=None*)

Return a morphism from this affine scheme into an ambient projective space of the same dimension.

The codomain of this morphism is the projective closure of this affine scheme in PP, if given, or otherwise in a new projective space that is constructed.

INPUT:

- i – integer (default: dimension of self = last coordinate) determines which projective embedding to compute. The embedding is that which has a 1 in the i-th coordinate, numbered from 0.
- **PP – (default: None) ambient projective space, i.e., ambient space** of codomain of morphism; this is constructed if it is not given.

EXAMPLES:

```
sage: A.<x, y, z> = AffineSpace(3, ZZ)
sage: S = A.subscheme([x*y - z])
sage: S.projective_embedding()                                                 #␣
↪needs sage.libs.singular
Scheme morphism:
  From: Closed subscheme of Affine Space of dimension 3 over Integer Ring
        defined by: x*y - z
  To:   Closed subscheme of Projective Space of dimension 3 over Integer Ring
        defined by: x0*x1 - x2*x3
  Defn: Defined on coordinates by sending (x, y, z) to (x : y : z : 1)
```

```
sage: A.<x, y, z> = AffineSpace(3, ZZ)
sage: P = ProjectiveSpace(3, ZZ, 'u')
sage: S = A.subscheme([x^2 - y*z])
sage: S.projective_embedding(1, P)                                             #␣
↪needs sage.libs.singular
Scheme morphism:
  From: Closed subscheme of Affine Space of dimension 3 over Integer Ring
        defined by: x^2 - y*z
  To:   Closed subscheme of Projective Space of dimension 3 over Integer Ring
        defined by: u0^2 - u2*u3
  Defn: Defined on coordinates by sending (x, y, z) to (x : 1 : y : z)
```

```
sage: A.<x,y,z> = AffineSpace(QQ, 3)
sage: X = A.subscheme([y - x^2, z - x^3])
sage: X.projective_embedding()                                                 #␣
↪needs sage.libs.singular
```

(continues on next page)

(continued from previous page)

```
Scheme morphism:
  From: Closed subscheme of Affine Space of dimension 3 over Rational Field
        defined by: -x^2 + y, -x^3 + z
  To:   Closed subscheme of Projective Space of dimension 3 over Rational Field
        defined by: x0^2 - x1*x3, x0*x1 - x2*x3, x1^2 - x0*x2
  Defn: Defined on coordinates by sending (x, y, z) to (x : y : z : 1)
```

When taking a closed subscheme of an affine space with a projective embedding, the subscheme inherits the embedding:

```
sage: A.<u,v> = AffineSpace(2, QQ, default_embedding_index=1)
sage: X = A.subscheme(u - v)                                                #␣
↪needs sage.libs.singular
sage: X.projective_embedding()                                              #␣
↪needs sage.libs.singular
Scheme morphism:
  From: Closed subscheme of Affine Space of dimension 2 over Rational Field
        defined by: u - v
  To:   Closed subscheme of Projective Space of dimension 2 over Rational Field
        defined by: x0 - x2
  Defn: Defined on coordinates by sending (u, v) to (u : 1 : v)
sage: phi = X.projective_embedding()                                        #␣
↪needs sage.libs.singular
sage: psi = A.projective_embedding()
sage: phi(X(2, 2)) == psi(A(X(2, 2)))                                       #␣
↪needs sage.libs.singular
True
```

class sage.schemes.affine.affine_subscheme.**AlgebraicScheme_subscheme_affine_field**(*A*, *polynomials*, *embedding_center=None*, *embedding_codomain=None*, *embedding_images=None*)

Bases: *AlgebraicScheme_subscheme_affine*

Algebraic subschemes of projective spaces defined over fields.

tangent_space(*p*)

Return the tangent space at the point `p`.

The points of the tangent space are the tangent vectors at `p`.

INPUT:

- `p` – a rational point

EXAMPLES:

```
sage: A3.<x,y,z> = AffineSpace(3, QQ)
sage: X = A3.subscheme(z - x*y)
sage: X.tangent_space(A3.origin())                                          #␣
↪needs sage.libs.singular
```

(continues on next page)

(continued from previous page)

```
Closed subscheme of Affine Space of dimension 3 over Rational Field
 defined by:
  z
sage: X.tangent_space(X(1,1,1))                                                #␣
↪needs sage.libs.singular
Closed subscheme of Affine Space of dimension 3 over Rational Field
 defined by:
  -x - y + z
```

Tangent space at a point may have higher dimension than the dimension of the point.

```
sage: # needs sage.libs.singular
sage: C = Curve([x + y + z, x^2 - y^2*z^2 + z^3])
sage: C.singular_points()
[(0, 0, 0)]
sage: p = C(0,0,0)
sage: C.tangent_space(p)
Closed subscheme of Affine Space of dimension 3 over Rational Field
 defined by:
  x + y + z
sage: _.dimension()
2
sage: q = C(1,0,-1)
sage: C.tangent_space(q)
Closed subscheme of Affine Space of dimension 3 over Rational Field
 defined by:
  x + y + z,
  2*x + 3*z
sage: _.dimension()
1
```

13.5 Enumeration of rational points on affine schemes

Naive algorithms for enumerating rational points over $\mathbf{Q}$ or finite fields over for general schemes.

Warning: Incorrect results and infinite loops may occur if using a wrong function.

(For instance using an affine function for a projective scheme or a finite field function for a scheme defined over an infinite field.)

EXAMPLES:

Affine, over $\mathbf{Q}$:

```
sage: from sage.schemes.affine.affine_rational_point import enum_affine_rational_field
sage: A.<x,y,z> = AffineSpace(3, QQ)
sage: S = A.subscheme([2*x - 3*y])
sage: enum_affine_rational_field(S, 2)
[(0, 0, -2), (0, 0, -1), (0, 0, -1/2), (0, 0, 0),
 (0, 0, 1/2), (0, 0, 1), (0, 0, 2)]
```

Affine over a finite field:

```
sage: from sage.schemes.affine.affine_rational_point import enum_affine_finite_field
sage: A.<w,x,y,z> = AffineSpace(4, GF(2))
sage: enum_affine_finite_field(A(GF(2)))
[(0, 0, 0, 0), (0, 0, 0, 1), (0, 0, 1, 0), (0, 0, 1, 1), (0, 1, 0, 0),
 (0, 1, 0, 1), (0, 1, 1, 0), (0, 1, 1, 1), (1, 0, 0, 0), (1, 0, 0, 1),
 (1, 0, 1, 0), (1, 0, 1, 1), (1, 1, 0, 0), (1, 1, 0, 1), (1, 1, 1, 0),
 (1, 1, 1, 1)]
```

AUTHORS:

- David R. Kohel <kohel@maths.usyd.edu.au>: original version.
- John Cremona and Charlie Turner <charlotteturner@gmail.com> (06-2010): improvements to clarity and documentation.

sage.schemes.affine.affine_rational_point.**enum_affine_finite_field**(*X*)

Enumerates affine points on scheme `X` defined over a finite field.

INPUT:

- `X` - a scheme defined over a finite field or a set of abstract rational points of such a scheme.

OUTPUT:

- a list containing the affine points of `X` over the finite field, sorted.

EXAMPLES:

```
sage: from sage.schemes.affine.affine_rational_point import enum_affine_finite_field
sage: F = GF(7)
sage: A.<w,x,y,z> = AffineSpace(4, F)
sage: C = A.subscheme([w^2 + x + 4, y*z*x - 6, z*y + w*x])
sage: enum_affine_finite_field(C(F))
[]
sage: C = A.subscheme([w^2 + x + 4, y*z*x - 6])
sage: enum_affine_finite_field(C(F))
[(0, 3, 1, 2), (0, 3, 2, 1), (0, 3, 3, 3), (0, 3, 4, 4), (0, 3, 5, 6),
 (0, 3, 6, 5), (1, 2, 1, 3), (1, 2, 2, 5), (1, 2, 3, 1), (1, 2, 4, 6),
 (1, 2, 5, 2), (1, 2, 6, 4), (2, 6, 1, 1), (2, 6, 2, 4), (2, 6, 3, 5),
 (2, 6, 4, 2), (2, 6, 5, 3), (2, 6, 6, 6), (3, 1, 1, 6), (3, 1, 2, 3),
 (3, 1, 3, 2), (3, 1, 4, 5), (3, 1, 5, 4), (3, 1, 6, 1), (4, 1, 1, 6),
 (4, 1, 2, 3), (4, 1, 3, 2), (4, 1, 4, 5), (4, 1, 5, 4), (4, 1, 6, 1),
 (5, 6, 1, 1), (5, 6, 2, 4), (5, 6, 3, 5), (5, 6, 4, 2), (5, 6, 5, 3),
 (5, 6, 6, 6), (6, 2, 1, 3), (6, 2, 2, 5), (6, 2, 3, 1), (6, 2, 4, 6),
 (6, 2, 5, 2), (6, 2, 6, 4)]
```

```
sage: A.<x,y,z> = AffineSpace(3, GF(3))
sage: S = A.subscheme(x + y)
sage: enum_affine_finite_field(S)
[(0, 0, 0), (0, 0, 1), (0, 0, 2), (1, 2, 0), (1, 2, 1), (1, 2, 2),
 (2, 1, 0), (2, 1, 1), (2, 1, 2)]
```

ALGORITHM:

Checks all points in affine space to see if they lie on X.

Warning: If `X` is defined over an infinite field, this code will not finish!

AUTHORS:

- John Cremona and Charlie Turner (06-2010)

sage.schemes.affine.affine_rational_point.**enum_affine_number_field**(*X*, ***kwds*)

Enumerates affine points on scheme `X` defined over a number field. Simply checks all of the points of absolute height up to `B` and adds those that are on the scheme to the list.

This algorithm computes 2 lists: L containing elements x in K such that H_k(x) <= B, and a list L' containing elements x in K that, due to floating point issues, may be slightly larger then the bound. This can be controlled by lowering the tolerance.

ALGORITHM:

This is an implementation of the revised algorithm (Algorithm 4) in [DK2013]. Algorithm 5 is used for imaginary quadratic fields.

INPUT:

kwds:

- `bound` - a real number
- `tolerance` - a rational number in (0,1] used in doyle-krumm algorithm-4
- `precision` - the precision to use for computing the elements of bounded height of number fields.

OUTPUT:

- a list containing the affine points of `X` of absolute height up to `B`, sorted.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: from sage.schemes.affine.affine_rational_point import enum_affine_number_field
sage: u = QQ['u'].0
sage: K = NumberField(u^2 + 2, 'v')
sage: A.<x,y,z> = AffineSpace(K, 3)
sage: X = A.subscheme([y^2 - x])
sage: enum_affine_number_field(X(K), bound=2**0.5)
[(0, 0, -1), (0, 0, -v), (0, 0, -1/2*v), (0, 0, 0), (0, 0, 1/2*v),
 (0, 0, v), (0, 0, 1), (1, -1, -1), (1, -1, -v), (1, -1, -1/2*v),
 (1, -1, 0), (1, -1, 1/2*v), (1, -1, v), (1, -1, 1), (1, 1, -1),
 (1, 1, -v), (1, 1, -1/2*v), (1, 1, 0), (1, 1, 1/2*v), (1, 1, v), (1, 1, 1)]
```

```
sage: # needs sage.rings.number_field
sage: from sage.schemes.affine.affine_rational_point import enum_affine_number_field
sage: u = QQ['u'].0
sage: K = NumberField(u^2 + 3, 'v')
sage: A.<x,y> = AffineSpace(K, 2)
sage: X = A.subscheme(x - y)
sage: enum_affine_number_field(X, bound=3**0.25)
[(-1, -1), (-1/2*v - 1/2, -1/2*v - 1/2), (1/2*v - 1/2, 1/2*v - 1/2),
 (0, 0), (-1/2*v + 1/2, -1/2*v + 1/2), (1/2*v + 1/2, 1/2*v + 1/2), (1, 1)]
```

sage.schemes.affine.affine_rational_point.**enum_affine_rational_field**(*X*, *B*)

Enumerates affine rational points on scheme X up to bound B.

INPUT:

- X - a scheme or set of abstract rational points of a scheme.
- B - a positive integer bound.

OUTPUT:

- a list containing the affine points of X of height up to B, sorted.

EXAMPLES:

```
sage: A.<x,y,z> = AffineSpace(3, QQ)
sage: from sage.schemes.affine.affine_rational_point import enum_affine_rational_
↪field
sage: enum_affine_rational_field(A(QQ), 1)
[(-1, -1, -1), (-1, -1, 0), (-1, -1, 1), (-1, 0, -1), (-1, 0, 0), (-1, 0, 1),
 (-1, 1, -1), (-1, 1, 0), (-1, 1, 1), (0, -1, -1), (0, -1, 0), (0, -1, 1),
 (0, 0, -1), (0, 0, 0), (0, 0, 1), (0, 1, -1), (0, 1, 0), (0, 1, 1), (1, -1, -1),
 (1, -1, 0), (1, -1, 1), (1, 0, -1), (1, 0, 0), (1, 0, 1), (1, 1, -1), (1, 1, 0),
 (1, 1, 1)]
```

```
sage: A.<w,x,y,z> = AffineSpace(4, QQ)
sage: S = A.subscheme([x^2 - y*z + 1, w^3 + z + y^2])
sage: enum_affine_rational_field(S(QQ), 1)
[(0, 0, -1, -1)]
sage: enum_affine_rational_field(S(QQ), 2)
[(0, 0, -1, -1), (1, -1, -1, -2), (1, 1, -1, -2)]
```

```
sage: A.<x,y> = AffineSpace(2, QQ)
sage: C = Curve(x^2 + y - x)                                                    #␣
↪needs sage.libs.singular
sage: enum_affine_rational_field(C, 10)         # long time (3 s)               #␣
↪needs sage.libs.singular
[(-2, -6), (-1, -2), (-2/3, -10/9), (-1/2, -3/4), (-1/3, -4/9),
 (0, 0), (1/3, 2/9), (1/2, 1/4), (2/3, 2/9), (1, 0),
 (4/3, -4/9), (3/2, -3/4), (5/3, -10/9), (2, -2), (3, -6)]
```

AUTHORS:

- David R. Kohel <kohel@maths.usyd.edu.au>: original version.
- Charlie Turner (06-2010): small adjustments.
- Raman Raghukul 2018: updated.

13.6 Set of homomorphisms between two affine schemes

For schemes X and Y, this module implements the set of morphisms $Hom(X, Y)$. This is done by *SchemeHomset_generic*.

As a special case, the Hom-sets can also represent the points of a scheme. Recall that the K-rational points of a scheme X over k can be identified with the set of morphisms $Spec(K) \to X$. In Sage the rational points are implemented by such scheme morphisms. This is done by *SchemeHomset_points* and its subclasses.

Note: You should not create the Hom-sets manually. Instead, use the `Hom()` method that is inherited by all schemes.

AUTHORS:

- William Stein (2006): initial version.
- Ben Hutz (2018): add numerical point support

class sage.schemes.affine.affine_homset.**SchemeHomset_points_affine**(*X*, *Y*, *category=None*, *check=True*, *base=Integer Ring*)

Bases: *SchemeHomset_points*

Set of rational points of an affine variety.

INPUT:

See *SchemeHomset_generic*.

EXAMPLES:

```
sage: from sage.schemes.affine.affine_homset import SchemeHomset_points_affine
sage: SchemeHomset_points_affine(Spec(QQ), AffineSpace(ZZ,2))
Set of rational points of Affine Space of dimension 2 over Rational Field
```

numerical_points(*F=None*, ***kwds*)

Return some or all numerical approximations of rational points of an affine scheme.

This is for dimension 0 subschemes only and the points are determined through a groebner calculation over the base ring and then numerically approximating the roots of the resulting polynomials. If the base ring is a number field, the embedding into `F` must be known.

INPUT:

`F` - numerical ring

kwds:

- `zero_tolerance` - positive real number (optional, default=10^(-10)). For numerically inexact fields, points are on the subscheme if they satisfy the equations to within tolerance.

OUTPUT: A list of points in the ambient space.

Warning: For numerically inexact fields the list of points returned may contain repeated or be missing points due to tolerance.

EXAMPLES:

```
sage: # needs sage.libs.singular sage.rings.number_field
sage: K.<v> = QuadraticField(3)
sage: A.<x,y> = AffineSpace(K, 2)
sage: X = A.subscheme([x^3 - v^2*y, y - v*x^2 + 3])
sage: L = X(K).numerical_points(F=RR); L  # abs tol 1e-14
[(-1.18738247880014, -0.558021142104134),
 (1.57693558184861, 1.30713548084184),
 (4.80659931965815, 37.0162574656220)]
sage: L[0].codomain()
Affine Space of dimension 2 over Real Field with 53 bits of precision
```

```
sage: A.<x,y> = AffineSpace(QQ, 2)
sage: X = A.subscheme([y^2 - x^2 - 3*x, x^2 - 10*y])
sage: len(X(QQ).numerical_points(F=ComplexField(100)))                    #␣
↪needs sage.libs.singular
4
```

```
sage: A.<x1, x2> = AffineSpace(QQ, 2)
sage: E = A.subscheme([30*x1^100 + 1000*x2^2 + 2000*x1*x2 + 1, x1 + x2])
sage: len(E(A.base_ring()).numerical_points(F=CDF, zero_tolerance=1e-9))    #␣
↪needs sage.libs.singular
100
```

points(***kwds*)

Return some or all rational points of an affine scheme.

For dimension 0 subschemes points are determined through a groebner basis calculation. For schemes or subschemes with dimension greater than 1 points are determined through enumeration up to the specified bound.

Over a finite field, all points are returned. Over an infinite field, all points satisfying the bound are returned. For a zero-dimensional subscheme, all points are returned regardless of whether the field is infinite or not.

For number fields, this uses the Doyle-Krumm algorithm 4 (algorithm 5 for imaginary quadratic) for computing algebraic numbers up to a given height [DK2013].

The algorithm requires floating point arithmetic, so the user is allowed to specify the precision for such calculations. Additionally, due to floating point issues, points slightly larger than the bound may be returned. This can be controlled by lowering the tolerance.

INPUT:

kwds:

- `bound` - real number (optional, default: 0). The bound for the height of the coordinates. Only used for subschemes with dimension at least 1.
- `zero_tolerance` - positive real number (optional, default=10^(-10)). For numerically inexact fields, points are on the subscheme if they satisfy the equations to within tolerance.
- `tolerance` - a rational number in (0,1] used in doyle-krumm algorithm-4 for enumeration over number fields.
- `precision` - the precision to use for computing the elements of bounded height of number fields.

OUTPUT:

- a list of rational points of a affine scheme

Warning: For numerically inexact fields such as ComplexField or RealField the list of points returned is very likely to be incomplete. It may also contain repeated points due to tolerance.

EXAMPLES: The bug reported at #11526 is fixed:

```
sage: A2 = AffineSpace(ZZ, 2)
sage: F = GF(3)
sage: A2(F).points()
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
```

```
sage: A.<x,y> = ZZ[]
sage: I = A.ideal(x^2 - y^2 - 1)
sage: V = AffineSpace(ZZ, 2)
sage: X = V.subscheme(I)
sage: M = X(ZZ)
sage: M.points(bound=1)
[(-1, 0), (1, 0)]
```

```
sage: u = QQ['u'].0
sage: K.<v> = NumberField(u^2 + 3)                                    #␣
↪needs sage.rings.number_field
sage: A.<x,y> = AffineSpace(K, 2)                                     #␣
↪needs sage.rings.number_field
sage: len(A(K).points(bound=2))                                       #␣
↪needs sage.rings.number_field
1849
```

```
sage: A.<x,y> = AffineSpace(QQ, 2)
sage: E = A.subscheme([x^2 + y^2 - 1, y^2 - x^3 + x^2 + x - 1])
sage: E(A.base_ring()).points()                                       #␣
↪needs sage.libs.singular
[(-1, 0), (0, -1), (0, 1), (1, 0)]
```

```
sage: A.<x,y> = AffineSpace(CC, 2)                                    #␣
↪needs sage.rings.real_mpfr
sage: E = A.subscheme([y^3 - x^3 - x^2, x*y])
sage: E(A.base_ring()).points()                                       #␣
↪needs sage.libs.singular sage.rings.real_mpfr
verbose 0 (...: affine_homset.py, points)
Warning: computations in the numerical fields are inexact;points
may be computed partially or incorrectly.
[(-1.00000000000000, 0.000000000000000),
 (0.000000000000000, 0.000000000000000)]
```

```
sage: A.<x1,x2> = AffineSpace(CDF, 2)                                 #␣
↪needs sage.rings.complex_double
sage: E = A.subscheme([x1^2 + x2^2 + x1*x2, x1 + x2])                 #␣
↪needs sage.libs.singular sage.rings.complex_double
sage: E(A.base_ring()).points()                                       #␣
↪needs sage.libs.singular sage.rings.complex_double
verbose 0 (...: affine_homset.py, points)
```

(continues on next page)

(continued from previous page)

```
Warning: computations in the numerical fields are inexact;points
may be computed partially or incorrectly.
[(0.0, 0.0)]
```

class sage.schemes.affine.affine_homset.**SchemeHomset_points_spec**(*X*, *Y*, *category=None*, *check=True*, *base=None*)

Bases: *SchemeHomset_generic*

Set of rational points of an affine variety.

INPUT:

See *SchemeHomset_generic*.

EXAMPLES:

```
sage: from sage.schemes.affine.affine_homset import SchemeHomset_points_spec
sage: SchemeHomset_points_spec(Spec(QQ), Spec(QQ))
Set of rational points of Spectrum of Rational Field
```

class sage.schemes.affine.affine_homset.**SchemeHomset_polynomial_affine_space**(*X*, *Y*, *category=None*, *check=True*, *base=None*)

Bases: *SchemeHomset_generic*

Set of morphisms between affine spaces defined by polynomials.

EXAMPLES:

```
sage: A.<x,y> = AffineSpace(2, QQ)
sage: Hom(A, A)
Set of morphisms
  From: Affine Space of dimension 2 over Rational Field
  To:   Affine Space of dimension 2 over Rational Field
```

identity()

The identity morphism of this homset.

EXAMPLES:

```
sage: A.<x,y> = AffineSpace(2, QQ)
sage: I = A.identity_morphism()
sage: I.parent()
Set of morphisms
  From: Affine Space of dimension 2 over Rational Field
  To:   Affine Space of dimension 2 over Rational Field
sage: _.identity() == I
True
```

CHAPTER

FOURTEEN

PROJECTIVE SCHEMES

14.1 Projective n space over a ring

EXAMPLES:

We construct projective space over various rings of various dimensions.

The simplest projective space:

```
sage: ProjectiveSpace(0)
Projective Space of dimension 0 over Integer Ring
```

A slightly bigger projective space over $\mathbf{Q}$:

```
sage: X = ProjectiveSpace(1000, QQ); X
Projective Space of dimension 1000 over Rational Field
sage: X.dimension()
1000
```

We can use "over" notation to create projective spaces over various base rings.

```
sage: X = ProjectiveSpace(5)/QQ; X
Projective Space of dimension 5 over Rational Field
sage: X/CC                                                                    #␣
↪needs sage.rings.real_mpfr
Projective Space of dimension 5 over Complex Field with 53 bits of precision
```

The third argument specifies the printing names of the generators of the homogeneous coordinate ring. Using the method `objgens()` you can obtain both the space and the generators as ready to use variables.

```
sage: P2, vars = ProjectiveSpace(10, QQ, 't').objgens()
sage: vars
(t0, t1, t2, t3, t4, t5, t6, t7, t8, t9, t10)
```

You can alternatively use the special syntax with < and >.

```
sage: P2.<x,y,z> = ProjectiveSpace(2, QQ)
sage: P2
Projective Space of dimension 2 over Rational Field
sage: P2.coordinate_ring()
Multivariate Polynomial Ring in x, y, z over Rational Field
```

The first of the three lines above is just equivalent to the two lines:

```
sage: P2 = ProjectiveSpace(2, QQ, 'xyz')
sage: x,y,z = P2.gens()
```

For example, we use x, y, z to define the intersection of two lines.

```
sage: V = P2.subscheme([x + y + z, x + y - z]); V
Closed subscheme of Projective Space of dimension 2 over Rational Field defined by:
  x + y + z,
  x + y - z
sage: V.dimension()                                                          #␣
↪needs sage.libs.singular
0
```

AUTHORS:

- Ben Hutz: (June 2012): support for rings
- Ben Hutz (9/2014): added support for Cartesian products
- Rebecca Lauren Miller (March 2016) : added point_transformation_matrix

sage.schemes.projective.projective_space.**ProjectiveSpace**(*n*, *R=None*, *names=None*)

Return projective space of dimension `n` over the ring `R`.

EXAMPLES: The dimension and ring can be given in either order.

```
sage: ProjectiveSpace(3, QQ)
Projective Space of dimension 3 over Rational Field
sage: ProjectiveSpace(5, QQ)
Projective Space of dimension 5 over Rational Field
sage: P = ProjectiveSpace(2, QQ, names='XYZ'); P
Projective Space of dimension 2 over Rational Field
sage: P.coordinate_ring()
Multivariate Polynomial Ring in X, Y, Z over Rational Field
```

The divide operator does base extension.

```
sage: ProjectiveSpace(5)/GF(17)
Projective Space of dimension 5 over Finite Field of size 17
```

The default base ring is $\mathbf{Z}$.

```
sage: ProjectiveSpace(5)
Projective Space of dimension 5 over Integer Ring
```

There is also an projective space associated each polynomial ring.

```
sage: R = GF(7)['x,y,z']
sage: P = ProjectiveSpace(R); P
Projective Space of dimension 2 over Finite Field of size 7
sage: P.coordinate_ring()
Multivariate Polynomial Ring in x, y, z over Finite Field of size 7
sage: P.coordinate_ring() is R
True
```

```
sage: ProjectiveSpace(3, Zp(5), 'y')                                    #␣
↪needs sage.rings.padics
Projective Space of dimension 3 over 5-adic Ring with capped relative precision 20
```

```
sage: ProjectiveSpace(2, QQ, 'x,y,z')
Projective Space of dimension 2 over Rational Field
```

```
sage: PS.<x,y> = ProjectiveSpace(1, CC); PS                             #␣
↪needs sage.rings.real_mpfr
Projective Space of dimension 1 over Complex Field with 53 bits of precision
```

```
sage: R.<x,y,z> = QQ[]
sage: ProjectiveSpace(R).variable_names()
('x', 'y', 'z')
```

Projective spaces are not cached, i.e., there can be several with the same base ring and dimension (to facilitate gluing constructions).

```
sage: R.<x> = QQ[]
sage: ProjectiveSpace(R)
Projective Space of dimension 0 over Rational Field
```

class sage.schemes.projective.projective_space.**ProjectiveSpace_field**(*n*, *R=Integer Ring*, *names=None*)

Bases: *ProjectiveSpace_ring*

curve(*F*)

Return a curve defined by `F` in this projective space.

INPUT:

- `F` – a polynomial, or a list or tuple of polynomials in the coordinate ring of this projective space

EXAMPLES:

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: P.curve([y^2 - x*z])                                              #␣
↪needs sage.schemes
Projective Plane Curve over Rational Field defined by y^2 - x*z
```

line_through(*p*, *q*)

Return the line through `p` and `q`.

INPUT:

- `p`, `q` – distinct rational points of the projective space

EXAMPLES:

```
sage: P3.<x0,x1,x2,x3> = ProjectiveSpace(3, QQ)
sage: p1 = P3(1, 2, 3, 4)
sage: p2 = P3(4, 3, 2, 1)
sage: P3.line_through(p1, p2)                                           #␣
↪needs sage.libs.singular sage.schemes
Projective Curve over Rational Field defined by
```

(continues on next page)

(continued from previous page)

```
  -5/4*x0 + 5/2*x1 - 5/4*x2,         -5/2*x0 + 15/4*x1 - 5/4*x3,
  -5/4*x0 + 15/4*x2 - 5/2*x3,        -5/4*x1 + 5/2*x2 - 5/4*x3
sage: p3 = P3(2,4,6,8)
sage: P3.line_through(p1, p3)
Traceback (most recent call last):
...
ValueError: not distinct points
```

subscheme_from_Chow_form(*Ch*, *dim*)

Returns the subscheme defined by the Chow equations associated to the Chow form Ch.

These equations define the subscheme set-theoretically, but only for smooth subschemes and hypersurfaces do they define the subscheme as a scheme.

ALGORITHM:

The Chow form is a polynomial in the Plucker coordinates. The Plucker coordinates are the bracket polynomials. We first re-write the Chow form in terms of the dual Plucker coordinates. Then we expand $Ch(span(p, L)$ for a generic point p and a generic linear subspace L. The coefficients as polynomials in the coordinates of p are the equations defining the subscheme. [DalbecSturmfels].

INPUT:

- Ch - a homogeneous polynomial.
- dim - the dimension of the associated scheme.

OUTPUT: a projective subscheme.

EXAMPLES:

```
sage: P = ProjectiveSpace(QQ, 4, 'z')
sage: R.<x0,x1,x2,x3,x4> = PolynomialRing(QQ)
sage: H = x1^2 + x2^2 + 5*x3*x4
sage: P.subscheme_from_Chow_form(H, 3)                                  #␣
↪needs sage.modules
Closed subscheme of Projective Space of dimension 4 over Rational Field defined␣
↪by:
  -5*z0*z1 + z2^2 + z3^2
```

```
sage: P = ProjectiveSpace(QQ, 3, 'z')
sage: R.<x0,x1,x2,x3,x4,x5> = PolynomialRing(QQ)
sage: H = x1 - x2 - x3 + x5 + 2*x0
sage: P.subscheme_from_Chow_form(H, 1)                                  #␣
↪needs sage.modules
Closed subscheme of Projective Space of dimension 3 over Rational Field
defined by:
  -z1 + z3,
  z0 + z2 + z3,
  -z1 - 2*z3,
  -z0 - z1 + 2*z2
```

```
sage: # needs sage.libs.singular
sage: P.<x0,x1,x2,x3> = ProjectiveSpace(GF(7), 3)
sage: X = P.subscheme([x3^2 + x1*x2, x2 - x0])
```

(continues on next page)

(continued from previous page)

```
sage: Ch = X.Chow_form(); Ch
t0^2 - 2*t0*t3 + t3^2 - t2*t4 - t4*t5
sage: Y = P.subscheme_from_Chow_form(Ch, 1); Y
Closed subscheme of Projective Space of dimension 3
 over Finite Field of size 7 defined by:
  x1*x2 + x3^2,
  -x0*x2 + x2^2,
  -x0*x1 - x1*x2 - 2*x3^2,
  x0^2 - x0*x2,
  x0*x1 + x3^2,
  -2*x0*x3 + 2*x2*x3,
  2*x0*x3 - 2*x2*x3,
  x0^2 - 2*x0*x2 + x2^2
sage: I = Y.defining_ideal()
sage: I.saturation(I.ring().ideal(list(I.ring().gens())))[0]
Ideal (x0 - x2, x1*x2 + x3^2) of Multivariate Polynomial Ring
 in x0, x1, x2, x3 over Finite Field of size 7
```

class sage.schemes.projective.projective_space.**ProjectiveSpace_finite_field**(*n*, *R=Integer Ring*, *names=None*)

Bases: *ProjectiveSpace_field*

rational_points(*F=None*)

Return the list of F-rational points on this projective space, where F is a given finite field, or the base ring of this space.

EXAMPLES:

```
sage: P = ProjectiveSpace(1, GF(3))
sage: P.rational_points()
[(0 : 1), (1 : 1), (2 : 1), (1 : 0)]
sage: sorted(P.rational_points(GF(3^2, 'b')), key=str)                    #␣
↪needs sage.rings.finite_rings
[(0 : 1), (1 : 0), (1 : 1), (2 : 1),
 (2*b + 1 : 1), (2*b + 2 : 1), (2*b : 1),
 (b + 1 : 1), (b + 2 : 1), (b : 1)]
```

rational_points_dictionary()

Return dictionary of points.

OUTPUT:

- dictionary

EXAMPLES:

```
sage: P1 = ProjectiveSpace(GF(7), 1, 'x')
sage: P1.rational_points_dictionary()
{(0 : 1): 0,
 (1 : 0): 7,
 (1 : 1): 1,
 (2 : 1): 2,
 (3 : 1): 3,
```

(continues on next page)

(continued from previous page)

```
 (4 : 1): 4,
 (5 : 1): 5,
 (6 : 1): 6}
```

class sage.schemes.projective.projective_space.**ProjectiveSpace_rational_field**(*n*, *R=Integer Ring*, *names=None*)

Bases: *ProjectiveSpace_field*

rational_points(*bound=0*)

Returns the projective points $(x_0 : \cdots : x_n)$ over $\mathbf{Q}$ with $|x_i| \leq$ bound.

ALGORITHM:

The very simple algorithm works as follows: every point $(x_0 : \cdots : x_n)$ in projective space has a unique largest index i for which x_i is not zero. The algorithm then iterates downward on this index. We normalize by choosing x_i positive. Then, the points $x_0, \ldots, x_{i-1}$ are the points of affine i-space that are relatively prime to x_i. We access these by using the Tuples method.

INPUT:

- bound - integer.

EXAMPLES:

```
sage: PP = ProjectiveSpace(0, QQ)
sage: PP.rational_points(1)
[(1)]
sage: PP = ProjectiveSpace(1, QQ)
sage: PP.rational_points(2)
[(-2 : 1), (-1 : 1), (0 : 1), (1 : 1), (2 : 1), (-1/2 : 1), (1/2 : 1), (1 : 0)]
sage: PP = ProjectiveSpace(2, QQ)
sage: PP.rational_points(2)
[(-2 : -2 : 1), (-1 : -2 : 1), (0 : -2 : 1), (1 : -2 : 1), (2 : -2 : 1),
 (-2 : -1 : 1), (-1 : -1 : 1), (0 : -1 : 1), (1 : -1 : 1), (2 : -1 : 1),
 (-2 : 0 : 1), (-1 : 0 : 1), (0 : 0 : 1), (1 : 0 : 1), (2 : 0 : 1), (-2 : 1 :␣
↪1),
 (-1 : 1 : 1), (0 : 1 : 1), (1 : 1 : 1), (2 : 1 : 1), (-2 : 2 : 1),
 (-1 : 2 : 1), (0 : 2 : 1), (1 : 2 : 1), (2 : 2 : 1), (-1/2 : -1 : 1),
 (1/2 : -1 : 1), (-1 : -1/2 : 1), (-1/2 : -1/2 : 1), (0 : -1/2 : 1),
 (1/2 : -1/2 : 1), (1 : -1/2 : 1), (-1/2 : 0 : 1), (1/2 : 0 : 1), (-1 : 1/2 :␣
↪1),
 (-1/2 : 1/2 : 1), (0 : 1/2 : 1), (1/2 : 1/2 : 1), (1 : 1/2 : 1), (-1/2 : 1 :␣
↪1),
 (1/2 : 1 : 1), (-2 : 1 : 0), (-1 : 1 : 0), (0 : 1 : 0), (1 : 1 : 0),
 (2 : 1 : 0), (-1/2 : 1 : 0), (1/2 : 1 : 0), (1 : 0 : 0)]
```

AUTHORS:

- Benjamin Antieau (2008-01-12)

class sage.schemes.projective.projective_space.**ProjectiveSpace_ring**(*n*, *R=Integer Ring*, *names=None*)

Bases: UniqueRepresentation, *AmbientSpace*

Projective space of dimension n over the ring R.

EXAMPLES:

```
sage: X.<x,y,z,w> = ProjectiveSpace(3, QQ)
sage: X.base_scheme()
Spectrum of Rational Field
sage: X.base_ring()
Rational Field
sage: X.structure_morphism()
Scheme morphism:
  From: Projective Space of dimension 3 over Rational Field
  To:   Spectrum of Rational Field
  Defn: Structure map
sage: X.coordinate_ring()
Multivariate Polynomial Ring in x, y, z, w over Rational Field
```

Loading and saving:

```
sage: loads(X.dumps()) == X
True
sage: P = ProjectiveSpace(ZZ, 1, 'x')
sage: loads(P.dumps()) is P
True
```

Equality and hashing:

```
sage: ProjectiveSpace(QQ, 3, 'a') == ProjectiveSpace(ZZ, 3, 'a')
False
sage: ProjectiveSpace(ZZ, 1, 'a') == ProjectiveSpace(ZZ, 0, 'a')
False
sage: ProjectiveSpace(ZZ, 2, 'a') == AffineSpace(ZZ, 2, 'a')
False

sage: ProjectiveSpace(QQ, 3, 'a') != ProjectiveSpace(ZZ, 3, 'a')
True
sage: ProjectiveSpace(ZZ, 1, 'a') != ProjectiveSpace(ZZ, 0, 'a')
True
sage: ProjectiveSpace(ZZ, 2, 'a') != AffineSpace(ZZ, 2, 'a')
True

sage: hash(ProjectiveSpace(QQ, 3, 'a')) == hash(ProjectiveSpace(ZZ, 3, 'a'))
False
sage: hash(ProjectiveSpace(ZZ, 1, 'a')) == hash(ProjectiveSpace(ZZ, 0, 'a'))
False
sage: hash(ProjectiveSpace(ZZ, 2, 'a')) == hash(AffineSpace(ZZ, 2, 'a'))
False
```

Lattes_map(E, m)

Given an elliptic curve `E` and an integer `m` return the Lattes map associated to multiplication by m.

In other words, the rational map on the quotient $E/\{\pm 1\} \cong \mathbb{P}^1$ associated to $[m] : E \to E$.

INPUT:

- `E` – an elliptic curve.
- `m` – an integer.

OUTPUT: a dynamical system on this projective space.

EXAMPLES:

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: E = EllipticCurve(QQ,[-1, 0])                                    #␣
↪needs sage.schemes
sage: P.Lattes_map(E, 2)                                               #␣
↪needs sage.schemes
Dynamical System of Projective Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (x : y) to
        (1/4*x^4 + 1/2*x^2*y^2 + 1/4*y^4 : x^3*y - x*y^3)
```

affine_patch(*i*, *AA=None*)

Return the i^{th} affine patch of this projective space.

This is an ambient affine space $\mathbb{A}^n_R$, where R is the base ring of `self`, whose "projective embedding" map is 1 in the i^{th} factor.

INPUT:

- `i` – integer between 0 and dimension of `self`, inclusive.
- `AA` – (default: None) ambient affine space, this is constructed if it is not given.

OUTPUT:

- An ambient affine space with fixed projective_embedding map.

EXAMPLES:

```
sage: PP = ProjectiveSpace(5) / QQ
sage: AA = PP.affine_patch(2)
sage: AA
Affine Space of dimension 5 over Rational Field
sage: AA.projective_embedding()
Scheme morphism:
  From: Affine Space of dimension 5 over Rational Field
  To:   Projective Space of dimension 5 over Rational Field
  Defn: Defined on coordinates by sending (x0, x1, x3, x4, x5) to
        (x0 : x1 : 1 : x3 : x4 : x5)
sage: AA.projective_embedding(0)
Scheme morphism:
  From: Affine Space of dimension 5 over Rational Field
  To:   Projective Space of dimension 5 over Rational Field
  Defn: Defined on coordinates by sending (x0, x1, x3, x4, x5) to
        (1 : x0 : x1 : x3 : x4 : x5)
```

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: P.affine_patch(0).projective_embedding(0).codomain() == P
True
```

cartesian_product(*other*)

Return the Cartesian product of this projective space and `other`.

INPUT:

- `other` - A projective space with the same base ring as this space.

OUTPUT:

- A Cartesian product of projective spaces.

EXAMPLES:

```
sage: P1 = ProjectiveSpace(QQ, 1, 'x')
sage: P2 = ProjectiveSpace(QQ, 2, 'y')
sage: PP = P1.cartesian_product(P2); PP
Product of projective spaces P^1 x P^2 over Rational Field
sage: PP.gens()
(x0, x1, y0, y1, y2)
```

change_ring(*R*)

Return a projective space over ring R.

INPUT:

- R – commutative ring or morphism.

OUTPUT:

- projective space over R.

Note: There is no need to have any relation between R and the base ring of this space, if you want to have such a relation, use `self.base_extend(R)` instead.

EXAMPLES:

```
sage: P.<x, y, z> = ProjectiveSpace(2, ZZ)
sage: PQ = P.change_ring(QQ); PQ
Projective Space of dimension 2 over Rational Field
sage: PQ.change_ring(GF(5))
Projective Space of dimension 2 over Finite Field of size 5
```

```
sage: K.<w> = QuadraticField(2)                                    #␣
↪needs sage.rings.number_field
sage: P = ProjectiveSpace(K, 2, 't')                               #␣
↪needs sage.rings.number_field
sage: P.change_ring(K.embeddings(QQbar)[0])                        #␣
↪needs sage.rings.number_field
Projective Space of dimension 2 over Algebraic Field
```

chebyshev_polynomial(*n*, *kind='first'*, *monic=False*)

Generates an endomorphism of this projective line by a Chebyshev polynomial.

Chebyshev polynomials are a sequence of recursively defined orthogonal polynomials. Chebyshev of the first kind are defined as $T_0(x) = 1$, $T_1(x) = x$, and $T_{n+1}(x) = 2xT_n(x) - T_{n-1}(x)$. Chebyshev of the second kind are defined as $U_0(x) = 1$, $U_1(x) = 2x$, and $U_{n+1}(x) = 2xU_n(x) - U_{n-1}(x)$.

INPUT:

- n – a non-negative integer.
- kind – `first` or `second` specifying which kind of chebyshev the user would like to generate. Defaults to `first`.

- monic – True or False specifying if the polynomial defining the system should be monic or not. Defaults to False.

OUTPUT: DynamicalSystem_projective

EXAMPLES:

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: P.chebyshev_polynomial(5, 'first')                                    #␣
↪needs sage.symbolic
Dynamical System of Projective Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (x : y) to
        (16*x^5 - 20*x^3*y^2 + 5*x*y^4 : y^5)
```

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: P.chebyshev_polynomial(3, 'second')                                   #␣
↪needs sage.symbolic
Dynamical System of Projective Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (x : y) to
        (8*x^3 - 4*x*y^2 : y^3)
```

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: P.chebyshev_polynomial(3, 2)                                          #␣
↪needs sage.symbolic
Traceback (most recent call last):
...
ValueError: keyword 'kind' must have a value of either 'first' or 'second'
```

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: P.chebyshev_polynomial(-4, 'second')
Traceback (most recent call last):
...
ValueError: first parameter 'n' must be a non-negative integer
```

```
sage: P = ProjectiveSpace(QQ, 2, 'x')
sage: P.chebyshev_polynomial(2)
Traceback (most recent call last):
...
TypeError: projective space must be of dimension 1
```

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: P.chebyshev_polynomial(3, monic=True)                                 #␣
↪needs sage.symbolic
Dynamical System of Projective Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (x : y) to
        (x^3 - 3*x*y^2 : y^3)
```

```
sage: F.<t> = FunctionField(QQ)
sage: P.<y,z> = ProjectiveSpace(F, 1)
sage: P.chebyshev_polynomial(4, monic=True)                                 #␣
↪needs sage.symbolic
Dynamical System of Projective Space of dimension 1
 over Rational function field in t over Rational Field
```

(continues on next page)

(continued from previous page)

```
  Defn: Defined on coordinates by sending (y : z) to
        (y^4 + (-4)*y^2*z^2 + 2*z^4 : z^4)
```

coordinate_ring()

Return the coordinate ring of this scheme.

EXAMPLES:

```
sage: ProjectiveSpace(3, GF(19^2,'alpha'), 'abcd').coordinate_ring()        #␣
↪needs sage.rings.finite_rings
Multivariate Polynomial Ring in a, b, c, d over Finite Field in alpha of size␣
↪19^2
```

```
sage: ProjectiveSpace(3).coordinate_ring()
Multivariate Polynomial Ring in x0, x1, x2, x3 over Integer Ring
```

```
sage: ProjectiveSpace(2, QQ, ['alpha', 'beta', 'gamma']).coordinate_ring()
Multivariate Polynomial Ring in alpha, beta, gamma over Rational Field
```

hyperplane_transformation_matrix(*plane_1*, *plane_2*)

Return a PGL element sending `plane_1` to `plane_2`.

`plane_1` and `plane_2` must be hyperplanes (subschemes of codimension 1, each defined by a single linear homogeneous equation).

INPUT:

- `plane_1`, `plane_2` – hyperplanes of this projective space

OUTPUT: An element of PGL

EXAMPLES:

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: plane1 = P.subscheme(x)
sage: plane2 = P.subscheme(y)
sage: m = P.hyperplane_transformation_matrix(plane1, plane2); m             #␣
↪needs sage.modules
[0 1]
[1 0]
sage: plane2(m*P((0,1)))                                                    #␣
↪needs sage.modules
(1 : 0)
```

```
sage: P.<x,y,z,w> = ProjectiveSpace(QQ, 3)
sage: plane1 = P.subscheme(x + 2*y + z)
sage: plane2 = P.subscheme(2*x + y + z)
sage: P.hyperplane_transformation_matrix(plane1, plane2)                    #␣
↪needs sage.modules
[1 0 0 0]
[0 4 0 0]
[0 0 2 0]
[0 0 0 1]
```

```
sage: P.<x,y> = ProjectiveSpace(ZZ, 1)
sage: plane1 = P.subscheme(x + y)
sage: plane2 = P.subscheme(y)
sage: P.hyperplane_transformation_matrix(plane1, plane2)                    #␣
↪needs sage.modules
[-1  0]
[ 1  1]
```

```
sage: # needs sage.rings.number_field
sage: K.<v> = CyclotomicField(3)
sage: P.<x,y,z> = ProjectiveSpace(K, 2)
sage: plane1 = P.subscheme(x - 2*v*y + z)
sage: plane2 = P.subscheme(x + v*y + v*z)
sage: m = P.hyperplane_transformation_matrix(plane1, plane2); m             #␣
↪needs sage.modules
[   v    0    0]
[   0 -2*v    0]
[   0    0    1]
```

```
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: K.<k> = NumberField(x^2 + 1)
sage: P.<x,y,z,w> = ProjectiveSpace(K, 3)
sage: plane1 = P.subscheme(k*x + 2*k*y + z)
sage: plane2 = P.subscheme(7*k*x + y + 9*z)
sage: m = P.hyperplane_transformation_matrix(plane1, plane2); m             #␣
↪needs sage.modules
[   1    0    0    0]
[   0 14*k    0    0]
[   0    0  7/9    0]
[   0    0    0    1]
```

```
sage: # needs sage.rings.number_field
sage: K.<v> = CyclotomicField(3)
sage: R.<t> = K[]
sage: F.<w> = K.extension(t^5 + 2)
sage: G.<u> = F.absolute_field()
sage: P.<x,y,z> = ProjectiveSpace(G, 2)
sage: plane1 = P.subscheme(x - 2*u*y + z)
sage: plane2 = P.subscheme(x + u*y + z)
sage: m = P.hyperplane_transformation_matrix(plane1, plane2)                #␣
↪needs sage.modules
sage: plane2(m*P((2*u, 1, 0)))                                              #␣
↪needs sage.modules
(-u : 1 : 0)
```

```
sage: P.<x,y,z> = ProjectiveSpace(FiniteField(2), 2)
sage: plane1 = P.subscheme(x + y + z)
sage: plane2 = P.subscheme(z)
sage: P.hyperplane_transformation_matrix(plane1, plane2)                    #␣
↪needs sage.modules
[1 0 0]
```

(continues on next page)

(continued from previous page)

```
[1 1 0]
[1 1 1]
```

```
sage: R.<t> = QQ[]
sage: P.<x,y,z> = ProjectiveSpace(R, 2)
sage: plane1 = P.subscheme(x + 9*t*y + z)
sage: plane2 = P.subscheme(x + z)
sage: P.hyperplane_transformation_matrix(plane1, plane2)                    #␣
↪needs sage.modules
[  1 9*t   0]
[  1   0   0]
[  0   0   1]
```

is_linearly_independent(*points*, *n=None*)

Return whether the set of points is linearly independent.

Alternatively, specify `n` to check if every subset of size `n` is linearly independent.

INPUT:

- `points` – a list of points in this projective space.
- `n` – (Optional) A positive integer less than or equal to the length of `points`. Specifies the size of the subsets to check for linear independence.

OUTPUT:

- `True` if `points` is linearly independent, `False` otherwise.

EXAMPLES:

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: points = [P((1, 0, 1)), P((1, 2, 1)), P((1, 3, 4))]
sage: P.is_linearly_independent(points)                                     #␣
↪needs sage.modules
True
```

```
sage: P.<x,y,z> = ProjectiveSpace(GF(5), 2)
sage: points = [P((1, 0, 1)), P((1, 2, 1)), P((1, 3, 4)), P((0, 0, 1))]
sage: P.is_linearly_independent(points, 2)                                  #␣
↪needs sage.modules
True
```

```
sage: R.<c> = QQ[]
sage: P.<x,y,z> = ProjectiveSpace(R, 2)
sage: points = [P((c, 0, 1)), P((0, c, 1)), P((1, 0, 4)), P((0, 0, 1))]
sage: P.is_linearly_independent(points, 3)                                  #␣
↪needs sage.modules
False
```

```
sage: R.<c> = QQ[]
sage: P.<x,y,z> = ProjectiveSpace(FractionField(R), 2)
sage: points = [P((c, 0, 1)), P((0, c, 1)), P((1, 3, 4)), P((0, 0, 1))]
sage: P.is_linearly_independent(points, 3)                                  #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.modules
True
```

```
sage: # needs sage.rings.number_field
sage: K.<k> = CyclotomicField(3)
sage: P.<x,y,z> = ProjectiveSpace(K, 2)
sage: points = [P((k, k^2, 1)), P((0, k, 1)), P((1, 0, 4)), P((0, 0, 1))]
sage: P.is_linearly_independent(points, 3)                                  #␣
↪needs sage.modules
True
```

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: points = [P((1, 0)), P((1, 1))]
sage: P.is_linearly_independent(points)                                     #␣
↪needs sage.modules
True
```

is_projective()

Return that this ambient space is projective n-space.

EXAMPLES:

```
sage: ProjectiveSpace(3,QQ).is_projective()
True
```

ngens()

Return the number of generators of this projective space.

This is the number of variables in the coordinate ring of `self`.

EXAMPLES:

```
sage: ProjectiveSpace(3, QQ).ngens()
4
sage: ProjectiveSpace(7, ZZ).ngens()
8
```

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

Create a point on this projective space.

INPUT:

- `v` – anything that defines a point
- `check` – boolean (optional, default: `True`); whether to check the defining data for consistency

OUTPUT: A point of this projective space.

EXAMPLES:

```
sage: P2 = ProjectiveSpace(QQ, 2)
sage: P2.point([4,5])
(4 : 5 : 1)
```

```
sage: P = ProjectiveSpace(QQ, 1)
sage: P.point(infinity)
(1 : 0)
```

```
sage: P = ProjectiveSpace(QQ, 2)
sage: P.point(infinity)
Traceback (most recent call last):
...
ValueError: +Infinity not well defined in dimension > 1
```

```
sage: P = ProjectiveSpace(ZZ, 2)
sage: P.point([infinity])
Traceback (most recent call last):
 ...
ValueError: [+Infinity] not well defined in dimension > 1
```

point_transformation_matrix(*points_source*, *points_target*, *normalize=True*)

Returns a unique element of PGL that transforms one set of points to another.

Given a projective space of dimension n and a set of n+2 source points and a set of n+2 target points in the same projective space, such that no n+1 points of each set are linearly dependent find the unique element of PGL that translates the source points to the target points.

Warning: over non-exact rings such as the ComplexField, the returned matrix could be very far from correct.

INPUT:

- `points_source` – points in source projective space.
- `points_target` – points in target projective space.
- `normalize` – (default: `True`) If the returned matrix should be normalized. Only works over exact rings. If the base ring is a field, the matrix is normalized so that the last nonzero entry in the last row is 1. If the base ring is a ring, then the matrix is normalized so that the entries are elements of the base ring.

OUTPUT: Transformation matrix - element of PGL.

ALGORITHM:

See [Hutz2007], Proposition 2.16 for details.

EXAMPLES:

```
sage: P1.<a,b,c> = ProjectiveSpace(QQ, 2)
sage: points_source = [P1([1, 4, 1]), P1([1, 2, 2]), P1([3, 5, 1]), P1([1, -1,␣
↪1])]
sage: points_target = [P1([5, -2, 7]), P1([3, -2, 3]), P1([6, -5, 9]), P1([3, 6,
↪ 7])]
sage: m = P1.point_transformation_matrix(points_source, points_target); m   #␣
↪needs sage.modules
[ -13/59 -128/59  -25/59]
[538/177    8/59  26/177]
[ -45/59 -196/59       1]
```

(continues on next page)

(continued from previous page)

```
sage: [m*points_source[i] == points_target[i] for i in range(4)]              #␣
↪needs sage.modules
[True, True, True, True]
```

```
sage: P.<a,b> = ProjectiveSpace(GF(13),  1)
sage: points_source = [P([-6, 7]), P([1, 4]), P([3, 2])]
sage: points_target = [P([-1, 2]), P([0, 2]), P([-1, 6])]
sage: P.point_transformation_matrix(points_source, points_target)             #␣
↪needs sage.modules
[10  4]
[10  1]
```

```
sage: P.<a,b> = ProjectiveSpace(QQ, 1)
sage: points_source = [P([-6, -4]), P([1, 4]), P([3, 2])]
sage: points_target = [P([-1, 2]), P([0, 2]), P([-7, -3])]
sage: P.point_transformation_matrix(points_source, points_target)             #␣
↪needs sage.modules
Traceback (most recent call last):
...
ValueError: source points not independent
```

```
sage: R.<t> = FunctionField(QQ)
sage: P.<a,b> = ProjectiveSpace(R, 1)
sage: points_source = [P([-6*t, 7]), P([1, 4]), P([3, 2])]
sage: points_target = [P([-1, 2*t]), P([0, 2]), P([-1, 6])]
sage: P.point_transformation_matrix(points_source, points_target)             #␣
↪needs sage.modules
[             (1/3*t + 7/12)/(t^2 - 53/24*t)       (-1/12*t - 7/48)/(t^2 - 53/
↪24*t)]
[(-2/3*t^2 - 7/36*t - 35/12)/(t^2 - 53/24*t)                                  ␣
↪  1]
```

```
sage: P1.<a,b,c> = ProjectiveSpace(RR, 2)
sage: points_source = [P1([1, 4, 1]), P1([1, 2, 2]), P1([3, 5, 1]), P1([1, -1,␣
↪1])]
sage: points_target = [P1([5, -2, 7]), P1([3, -2, 3]), P1([6, -5, 9]), P1([3, 6,
↪ 7])]
sage: P1.point_transformation_matrix(points_source,        # abs tol 1e-13  #␣
↪needs sage.modules
....:                                 points_target)
[-0.0619047619047597  -0.609523809523810  -0.119047619047621]
[  0.853968253968253  0.0380952380952380  0.0412698412698421]
[ -0.214285714285712  -0.933333333333333   0.280952380952379]
```

```
sage: P1.<a,b,c> = ProjectiveSpace(ZZ, 2)
sage: points_source = [P1([1, 4, 1]), P1([1, 2, 2]), P1([3, 5, 1]), P1([1, -1,␣
↪1])]
sage: points_target = [P1([5, -2, 7]), P1([3, -2, 3]), P1([6, -5, 9]), P1([3, 6,
↪ 7])]
sage: P1.point_transformation_matrix(points_source, points_target)            #␣
↪needs sage.modules
```

(continues on next page)

(continued from previous page)

```
[ -39 -384  -75]
[ 538   24   26]
[-135 -588  177]
```

```
sage: P1.<a,b,c> = ProjectiveSpace(ZZ, 2)
sage: points_source = [P1([1, 4, 1]), P1([1, 2, 2]), P1([3, 5, 1]), P1([1, -1,␣
↪1])]
sage: points_target = [P1([5, -2, 7]), P1([3, -2, 3]), P1([6, -5, 9]), P1([3, 6,
↪ 7])]
sage: P1.point_transformation_matrix(points_source, points_target,          #␣
↪needs sage.modules
....:                                 normalize=False)
[-13/30 -64/15   -5/6]
[269/45   4/15  13/45]
[  -3/2 -98/15  59/30]
```

```
sage: R.<t> = ZZ[]
sage: P.<a,b> = ProjectiveSpace(R, 1)
sage: points_source = [P([-6*t, 7]), P([1, 4]), P([3, 2])]
sage: points_target = [P([-1, 2*t]), P([0, 2]), P([-1, 6])]
sage: P.point_transformation_matrix(points_source, points_target)           #␣
↪needs sage.modules
[         -48*t - 84           12*t + 21]
[96*t^2 + 28*t + 420    -144*t^2 + 318*t]
```

points_of_bounded_height(***kwds*)

Return an iterator of the points in `self` of absolute multiplicative height of at most the given bound.

ALGORITHM:

This is an implementation of Algorithm 6 in [Krumm2016].

INPUT:

kwds:

- bound - a real number
- `precision` - (default: 53) a positive integer

OUTPUT:

- an iterator of points of bounded height

EXAMPLES:

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: sorted(list(P.points_of_bounded_height(bound=2)))
[(-2 : 1), (-1 : 1), (-1/2 : 1), (0 : 1),
 (1/2 : 1), (1 : 0), (1 : 1), (2 : 1)]
```

```
sage: u = QQ['u'].0
sage: P.<x,y,z> = ProjectiveSpace(NumberField(u^2 - 2, 'v'), 2)             #␣
↪needs sage.rings.number_field
sage: len(list(P.points_of_bounded_height(bound=2)))                        #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.rings.number_field
265
```

```
sage: # needs sage.rings.number_field
sage: CF.<a> = CyclotomicField(3)
sage: R.<x> = CF[]
sage: L.<l> = CF.extension(x^3 + 2)
sage: Q.<x,y> = ProjectiveSpace(L, 1)
sage: sorted(list(Q.points_of_bounded_height(bound=1)))
[(0 : 1), (1 : 0), (a + 1 : 1), (a : 1),
 (-1 : 1), (-a - 1 : 1), (-a : 1), (1 : 1)]
```

```
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: F.<a> = NumberField(x^4 - 8*x^2 + 3)
sage: P.<x,y,z> = ProjectiveSpace(F, 2)
sage: all(exp(p.global_height()) <= 1                                    #␣
↪needs sage.symbolic
....:     for p in P.points_of_bounded_height(bound=1))
True
```

```
sage: K.<a> = CyclotomicField(3)                                         #␣
↪needs sage.rings.number_field
sage: P.<x,y,z> = ProjectiveSpace(K, 2)                                  #␣
↪needs sage.rings.number_field
sage: len(list(P.points_of_bounded_height(bound=1)))                     #␣
↪needs sage.rings.number_field
57
```

```
sage: u = QQ['u'].0
sage: K.<k> = NumberField(u^2 - 2)                                       #␣
↪needs sage.rings.number_field
sage: P.<x,y> = ProjectiveSpace(K, 1)                                    #␣
↪needs sage.rings.number_field
sage: len(list(P.points_of_bounded_height(bound=2)))                     #␣
↪needs sage.rings.number_field
24
```

```
sage: R.<x> = QQ[]
sage: K.<k> = NumberField(x^4 - 8*x^2 + 3)                               #␣
↪needs sage.rings.number_field
sage: P.<x,y> = ProjectiveSpace(K, 1)                                    #␣
↪needs sage.rings.number_field
sage: len(list(P.points_of_bounded_height(bound=2)))                     #␣
↪needs sage.rings.number_field
108
```

```
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: K.<v> = NumberField(x^5 + x^3 + 1)
```

(continues on next page)

(continued from previous page)

```
sage: P.<x,y,z> = ProjectiveSpace(K, 2)
sage: L = P.points_of_bounded_height(bound=1.2)
sage: len(list(L))
109
```

```
sage: # needs sage.rings.number_field
sage: K.<v> = QuadraticField(2)
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: sorted(list(P.points_of_bounded_height(bound=2)))
[(-v - 2 : 1), (-v - 1 : 1), (-2 : 1), (-1/2*v - 1 : 1), (-v : 1), (-1 : 1),
 (-1/2*v : 1), (v - 2 : 1), (-1/2 : 1), (-v + 1 : 1), (1/2*v - 1 : 1), (0 : 1),
 (-1/2*v + 1 : 1), (v - 1 : 1), (1/2 : 1), (-v + 2 : 1), (1/2*v : 1), (1 : 0),
 (1 : 1), (v : 1), (1/2*v + 1 : 1), (2 : 1), (v + 1 : 1), (v + 2 : 1)]
```

```
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: K.<a> = NumberField(3*x^2 + 1)
sage: P.<z,w> = ProjectiveSpace(K, 1)
sage: sorted(list(P.points_of_bounded_height(bound=1)))
[(-1 : 1), (-3/2*a - 1/2 : 1), (3/2*a - 1/2 : 1), (0 : 1),
 (-3/2*a + 1/2 : 1), (3/2*a + 1/2 : 1), (1 : 0), (1 : 1)]
```

```
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: K.<a> = NumberField(3*x^2 + 1)
sage: O = K.maximal_order()
sage: P.<z,w> = ProjectiveSpace(O, 1)
sage: len(sorted(list(P.points_of_bounded_height(bound=2))))
44
```

```
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: K.<a> = NumberField(x^3 - 7)
sage: O = K.maximal_order()
sage: P.<z,w> = ProjectiveSpace(O, 1)
sage: len(sorted(list(P.points_of_bounded_height(bound=2))))
28
```

```
sage: P.<w,z> = ProjectiveSpace(ZZ, 1)
sage: sorted(list(P.points_of_bounded_height(bound=2)))
[(-2 : -1), (-2 : 1), (-1 : -2), (-1 : -1),
 (-1 : 0), (-1 : 1), (-1 : 2), (0 : -1)]
```

```
sage: R.<x> = QQ[]
sage: P.<z,w> = ProjectiveSpace(R, 1)
sage: P.points_of_bounded_height(bound=2)
Traceback (most recent call last):
...
NotImplementedError: self must be a projective space over
a number field or a ring of integers
```

```
sage: # needs sage.rings.number_field
sage: K.<i> = NumberField(x^2 + 1)
sage: PK.<t> = K[]
sage: L.<a> = K.extension(t^4  - i)
sage: P.<z,w> = ProjectiveSpace(L, 1)
sage: sorted(list(P.points_of_bounded_height(bound=1)))
[(0 : 1), (1 : 0), (a : 1), (a^2 : 1), (a^3 : 1), (i : 1),
 (i*a : 1), (i*a^2 : 1), (i*a^3 : 1), (-1 : 1), (-a : 1), (-a^2 : 1),
 (-a^3 : 1), (-i : 1), (-i*a : 1), (-i*a^2 : 1), (-i*a^3 : 1), (1 : 1)]
```

subscheme(*X*)

Return the closed subscheme defined by `X`.

INPUT:

- `X` - a list or tuple of equations.

EXAMPLES:

```
sage: A.<x,y,z> = ProjectiveSpace(2, QQ)
sage: X = A.subscheme([x*z^2, y^2*z, x*y^2]); X
Closed subscheme of Projective Space of dimension 2 over Rational Field defined␣
↪by:
  x*z^2,
  y^2*z,
  x*y^2
sage: X.defining_polynomials ()
(x*z^2, y^2*z, x*y^2)
sage: I = X.defining_ideal(); I
Ideal (x*z^2, y^2*z, x*y^2) of Multivariate Polynomial Ring in x, y, z
 over Rational Field
sage: I.groebner_basis()                                                    #␣
↪needs sage.libs.singular
[x*y^2, y^2*z,  x*z^2]
sage: X.dimension()                                                         #␣
↪needs sage.libs.singular
0
sage: X.base_ring()
Rational Field
sage: X.base_scheme()
Spectrum of Rational Field
sage: X.structure_morphism()
Scheme morphism:
  From: Closed subscheme of Projective Space of dimension 2
        over Rational Field defined by: x*z^2, y^2*z, x*y^2
  To:   Spectrum of Rational Field
  Defn: Structure map
```

veronese_embedding(*d*, *CS=None*, *order='lex'*)

Return the degree `d` Veronese embedding from this projective space.

INPUT:

- `d` – a positive integer.

- CS – a projective ambient space to embed into. If this projective space has dimension N, the dimension of CS must be $\binom{N+d}{d} - 1$. This is constructed if not specified. Default: None.
- order – a monomial order to use to arrange the monomials defining the embedding. The monomials will be arranged from greatest to least with respect to this order. Default: 'lex'.

OUTPUT:

- a scheme morphism from this projective space to CS.

EXAMPLES:

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: vd = P.veronese_embedding(4, order='invlex')                    #␣
↪needs sage.combinat
sage: vd                                                              #␣
↪needs sage.combinat
Scheme morphism:
  From: Projective Space of dimension 1 over Rational Field
  To:   Projective Space of dimension 4 over Rational Field
  Defn: Defined on coordinates by sending (x : y) to
        (y^4 : x*y^3 : x^2*y^2 : x^3*y : x^4)
```

Veronese surface:

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: Q.<q,r,s,t,u,v> = ProjectiveSpace(QQ, 5)
sage: vd = P.veronese_embedding(2, Q)                                 #␣
↪needs sage.combinat
sage: vd                                                              #␣
↪needs sage.combinat
Scheme morphism:
  From: Projective Space of dimension 2 over Rational Field
  To:   Projective Space of dimension 5 over Rational Field
  Defn: Defined on coordinates by sending (x : y : z) to
        (x^2 : x*y : x*z : y^2 : y*z : z^2)
sage: vd(P.subscheme([]))                                             #␣
↪needs sage.combinat sage.libs.singular
Closed subscheme of Projective Space of dimension 5 over Rational Field
 defined by:
  -u^2 + t*v,
  -s*u + r*v,
  -s*t + r*u,
  -s^2 + q*v,
  -r*s + q*u,
  -r^2 + q*t
```

sage.schemes.projective.projective_space.**is_ProjectiveSpace**(x)

Return True if x is a projective space.

In other words, if x is an ambient space $\mathbb{P}^n_R$, where R is a ring and $n \geq 0$ is an integer.

EXAMPLES:

```
sage: from sage.schemes.projective.projective_space import is_ProjectiveSpace
sage: is_ProjectiveSpace(ProjectiveSpace(5, names='x'))
True
```

(continues on next page)

(continued from previous page)

```
sage: is_ProjectiveSpace(ProjectiveSpace(5, GF(9, 'alpha'), names='x'))          #␣
↪needs sage.rings.finite_rings
True
sage: is_ProjectiveSpace(Spec(ZZ))
False
```

14.2 Morphisms on projective schemes

This module defines morphisms from projective schemes. A morphism from a projective scheme to a projective scheme is defined by homogeneous polynomials of the same degree that define what the morphism does on points in the ambient projective space. A morphism from a projective scheme to an affine scheme is determined by rational function, that is, quotients of homogeneous polynomials of the same degree.

EXAMPLES:

```
sage: P2.<x0,x1,x2> = ProjectiveSpace(QQ, 2)
sage: A2.<x,y> = AffineSpace(QQ, 2)
sage: P2.hom([x0, x1, x1 + x2], P2)
Scheme endomorphism of Projective Space of dimension 2 over Rational Field
  Defn: Defined on coordinates by sending (x0 : x1 : x2) to (x0 : x1 : x1 + x2)
sage: P2.hom([x1/x0, (x1 + x2)/x0], A2)
Scheme morphism:
  From: Projective Space of dimension 2 over Rational Field
  To:   Affine Space of dimension 2 over Rational Field
  Defn: Defined on coordinates by sending (x0 : x1 : x2) to (x1/x0, (x1 + x2)/x0)
```

AUTHORS:

- David Kohel, William Stein: initial version
- William Stein (2006-02-11): fixed bug where P(0,0,0) was allowed as a projective point
- Volker Braun (2011-08-08): renamed classes, more documentation, misc cleanups
- Ben Hutz (2013-03): iteration functionality and new directory structure for affine/projective, height functionality
- Brian Stout, Ben Hutz (2013-11): added minimal model functionality
- Dillon Rose (2014-01): speed enhancements
- Ben Hutz (2015-11): iteration of subschemes
- Kwankyu Lee (2020-02): added indeterminacy_locus() and image()
- Kwankyu Lee (2022-05): added graph(), projective_degrees(), and degree()

class sage.schemes.projective.projective_morphism.**SchemeMorphism_polynomial_projective_space**(*parent*, *polys*, *check=True*)

Bases: *SchemeMorphism_polynomial*

A morphism of schemes determined by rational functions that define what the morphism does on points in the ambient projective space.

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: P1 = ProjectiveSpace(R)
sage: H = P1.Hom(P1)
sage: H([y,2*x])
Scheme endomorphism of Projective Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (x : y) to
        (y : 2*x)
```

An example of a morphism between projective plane curves (see github issue #10297):

```
sage: # needs sage.schemes
sage: P2.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: f = x^3 + y^3 + 60*z^3
sage: g = y^2*z - (x^3 - 6400*z^3/3)
sage: C = Curve(f)
sage: E = Curve(g)
sage: xbar,ybar,zbar = C.coordinate_ring().gens()
sage: H = C.Hom(E)
sage: H([zbar, xbar - ybar, -(xbar+ybar)/80])
Scheme morphism:
  From: Projective Plane Curve over Rational Field defined by x^3 + y^3 + 60*z^3
  To:   Projective Plane Curve over Rational Field defined by -x^3 + y^2*z + 6400/
↪3*z^3
  Defn: Defined on coordinates by sending (x : y : z) to
        (z : x - y : -1/80*x - 1/80*y)
```

A more complicated example:

```
sage: P2.<x,y,z> = ProjectiveSpace(2, QQ)
sage: P1 = P2.subscheme(x - y)
sage: H12 = P1.Hom(P2)
sage: H12([x^2, x*z, z^2])
Scheme morphism:
  From: Closed subscheme of Projective Space of dimension 2 over Rational Field
        defined by: x - y
  To:   Projective Space of dimension 2 over Rational Field
  Defn: Defined on coordinates by sending (x : y : z) to (x^2 : x*z : z^2)
```

We illustrate some error checking:

```
sage: R.<x,y> = QQ[]
sage: P1 = ProjectiveSpace(R)
sage: H = P1.Hom(P1)
sage: f = H([x - y, x*y])
Traceback (most recent call last):
...
ValueError: polys (=[x - y, x*y]) must be of the same degree

sage: H([x - 1, x*y + x])
Traceback (most recent call last):
...
ValueError: polys (=[x - 1, x*y + x]) must be homogeneous
```

(continues on next page)

(continued from previous page)

```
sage: H([exp(x), exp(y)])                                                   #␣
↪needs sage.symbolic
Traceback (most recent call last):
...
TypeError: polys (=[e^x, e^y]) must be elements of
Multivariate Polynomial Ring in x, y over Rational Field
```

We can also compute the forward image of subschemes through elimination. In particular, let $X = V(h_1, \ldots, h_t)$ and define the ideal $I = (h_1, \ldots, h_t, y_0 - f_0(\bar{x}), \ldots, y_n - f_n(\bar{x}))$. Then the elimination ideal $I_{n+1} = I \cap K[y_0, \ldots, y_n]$ is a homogeneous ideal and $f(X) = V(I_{n+1})$:

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: H = End(P)
sage: f = H([(x-2*y)^2, (x-2*z)^2, x^2])
sage: X = P.subscheme(y-z)
sage: f(f(f(X)))                                                            #␣
↪needs sage.libs.singular
Closed subscheme of Projective Space of dimension 2 over Rational Field
 defined by:
  y - z
```

```
sage: P.<x,y,z,w> = ProjectiveSpace(QQ, 3)
sage: H = End(P)
sage: f = H([(x-2*y)^2, (x-2*z)^2, (x-2*w)^2, x^2])
sage: f(P.subscheme([x,y,z]))                                               #␣
↪needs sage.libs.singular
Closed subscheme of Projective Space of dimension 3 over Rational Field
 defined by:
  w,
  y,
  x
```

as_dynamical_system()

Return this endomorphism as a DynamicalSystem_projective.

OUTPUT:

- DynamicalSystem_projective

EXAMPLES:

```
sage: P.<x,y,z> = ProjectiveSpace(ZZ, 2)
sage: H = End(P)
sage: f = H([x^2, y^2, z^2])
sage: type(f.as_dynamical_system())                                         #␣
↪needs sage.schemes
<class 'sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_
↪projective'>
```

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: H = End(P)
sage: f = H([x^2 - y^2, y^2])
sage: type(f.as_dynamical_system())                                         #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.schemes
<class 'sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_
↪projective_field'>
```

```
sage: P.<x,y> = ProjectiveSpace(GF(5), 1)
sage: H = End(P)
sage: f = H([x^2, y^2])
sage: type(f.as_dynamical_system())                                    #␣
↪needs sage.schemes
<class 'sage.dynamics.arithmetic_dynamics.projective_ds.DynamicalSystem_
↪projective_finite_field'>
```

```
sage: P.<x,y> = ProjectiveSpace(RR, 1)
sage: f = DynamicalSystem([x^2 + y^2, y^2], P)                         #␣
↪needs sage.schemes
sage: g = f.as_dynamical_system()                                      #␣
↪needs sage.schemes
sage: g is f                                                           #␣
↪needs sage.schemes
True
```

degree()

Return the degree of this map.

The degree is defined as the degree of the homogeneous polynomials that are the coordinates of this map.

OUTPUT:

- A positive integer

EXAMPLES:

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: H = Hom(P, P)
sage: f = H([x^2 + y^2, y^2])
sage: f.degree()
2
```

```
sage: # needs sage.rings.real_mpfr
sage: P.<x,y,z> = ProjectiveSpace(CC, 2)
sage: H = Hom(P, P)
sage: f = H([x^3 + y^3, y^2*z, z*x*y])
sage: f.degree()
3
```

```
sage: R.<t> = PolynomialRing(QQ)
sage: P.<x,y,z> = ProjectiveSpace(R, 2)
sage: H = Hom(P, P)
sage: f = H([x^2 + t*y^2, (2-t)*y^2, z^2])
sage: f.degree()
2
```

```
sage: P.<x,y,z> = ProjectiveSpace(ZZ, 2)
sage: X = P.subscheme(x^2 - y^2)
sage: H = Hom(X, X)
sage: f = H([x^2, y^2, z^2])
sage: f.degree()
2
```

dehomogenize(*n*)

Return the standard dehomogenization at the `n[0]` coordinate for the domain and the `n[1]` coordinate for the codomain.

Note that the new function is defined over the fraction field of the base ring of this map.

INPUT:

- **n – a tuple of nonnegative integers. If n is an integer, then the two values of** the tuple are assumed to be the same.

OUTPUT:

- `SchemeMorphism_polynomial_affine_space`.

EXAMPLES:

```
sage: P.<x,y> = ProjectiveSpace(ZZ, 1)
sage: H = Hom(P, P)
sage: f = H([x^2 + y^2, y^2])
sage: f.dehomogenize(0)
Scheme endomorphism of Affine Space of dimension 1 over Integer Ring
  Defn: Defined on coordinates by sending (y) to (y^2/(y^2 + 1))
```

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: H = Hom(P, P)
sage: f = H([x^2 - y^2, y^2])
sage: f.dehomogenize((0,1))
Scheme morphism:
  From: Affine Space of dimension 1 over Rational Field
  To:   Affine Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (y) to ((-y^2 + 1)/y^2)
```

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: H = Hom(P, P)
sage: f = H([x^2 + y^2, y^2 - z^2, 2*z^2])
sage: f.dehomogenize(2)
Scheme endomorphism of Affine Space of dimension 2 over Rational Field
  Defn: Defined on coordinates by sending (x, y) to
        (1/2*x^2 + 1/2*y^2, 1/2*y^2 - 1/2)
```

```
sage: R.<t> = PolynomialRing(QQ)
sage: P.<x,y,z> = ProjectiveSpace(FractionField(R),2)
sage: H = Hom(P,P)
sage: f = H([x^2 + t*y^2, t*y^2 - z^2, t*z^2])
sage: f.dehomogenize(2)
Scheme endomorphism of Affine Space of dimension 2 over Fraction Field
 of Univariate Polynomial Ring in t over Rational Field
```

(continues on next page)

(continued from previous page)

```
  Defn: Defined on coordinates by sending (x, y) to
        (1/t*x^2 + y^2, y^2 - 1/t)
```

```
sage: P.<x,y,z> = ProjectiveSpace(ZZ, 2)
sage: X = P.subscheme(x^2 - y^2)
sage: H = Hom(X, X)
sage: f = H([x^2, y^2, x*z])
sage: f.dehomogenize(2)                                                    #␣
↪needs sage.libs.singular
Scheme endomorphism of Closed subscheme of Affine Space of dimension 2
 over Integer Ring defined by: x^2 - y^2
  Defn: Defined on coordinates by sending (x, y) to (x, y^2/x)
```

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: H = End(P)
sage: f = H([x^2 - 2*x*y, y^2])
sage: f.dehomogenize(0).homogenize(0) == f
True
```

```
sage: # needs sage.rings.number_field
sage: K.<w> = QuadraticField(3)
sage: O = K.ring_of_integers()
sage: P.<x,y> = ProjectiveSpace(O, 1)
sage: H = End(P)
sage: f = H([x^2 - O(w)*y^2, y^2])
sage: f.dehomogenize(1)
Scheme endomorphism of Affine Space of dimension 1 over
 Maximal Order in Number Field in w with defining polynomial x^2 - 3
  with w = 1.732050807568878?
  Defn: Defined on coordinates by sending (x) to (x^2 - w)
```

```
sage: P1.<x,y> = ProjectiveSpace(QQ, 1)
sage: P2.<u,v,w> = ProjectiveSpace(QQ, 2)
sage: H = Hom(P2, P1)
sage: f = H([u*w, v^2 + w^2])
sage: f.dehomogenize((2,1))
Scheme morphism:
  From: Affine Space of dimension 2 over Rational Field
  To:   Affine Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (u, v) to (u/(v^2 + 1))
```

global_height(*prec=None*)

Return the global height of the coefficients as a projective point.

INPUT:

- `prec` – desired floating point precision (default: default RealField precision).

OUTPUT:

- a real number.

EXAMPLES:

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: H = Hom(P, P)
sage: f = H([1/1331*x^2 + 1/4000*y^2, 210*x*y]);
sage: f.global_height()                                                    #␣
↪needs sage.symbolic
20.8348429892146
```

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: H = Hom(P, P)
sage: f = H([1/1331*x^2 + 1/4000*y^2, 210*x*y]);
sage: f.global_height(prec=11)                                             #␣
↪needs sage.symbolic
20.8
```

```
sage: P.<x,y,z> = ProjectiveSpace(ZZ, 2)
sage: H = Hom(P, P)
sage: f = H([4*x^2 + 100*y^2, 210*x*y, 10000*z^2]);
sage: f.global_height()                                                    #␣
↪needs sage.symbolic
8.51719319141624
```

```
sage: # needs sage.rings.number_field
sage: R.<z> = PolynomialRing(QQ)
sage: K.<w> = NumberField(z^2 - 2)
sage: O = K.maximal_order()
sage: P.<x,y> = ProjectiveSpace(O, 1)
sage: H = Hom(P, P)
sage: f = H([2*x^2 + 3*O(w)*y^2, O(w)*y^2])
sage: f.global_height()
1.09861228866811
```

```
sage: # needs sage.rings.number_field sage.symbolic
sage: P.<x,y> = ProjectiveSpace(QQbar, 1)
sage: P2.<u,v,w> = ProjectiveSpace(QQbar, 2)
sage: H = Hom(P, P2)
sage: f = H([x^2 + QQbar(I)*x*y + 3*y^2, y^2, QQbar(sqrt(5))*x*y])
sage: f.global_height()
1.09861228866811
```

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: A.<z,w> = ProjectiveSpace(QQ, 1)
sage: H = Hom(P, A)
sage: f = H([1/1331*x^2 + 4000*y*z, y^2])
sage: f.global_height()                                                    #␣
↪needs sage.symbolic
15.4877354584971
```

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem([1/25*x^2 + 25/3*x*y + y^2, 1*y^2])
sage: exp(f.global_height())                                               #␣
↪needs sage.symbolic
```

(continues on next page)

(continued from previous page)

```
625.000000000000
```

Scaling should not change the result:

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem([1/25*x^2 + 25/3*x*y + y^2, 1*y^2])
sage: f.global_height()                                                    #␣
↪needs sage.symbolic
6.43775164973640
sage: c = 10000
sage: f.scale_by(c)
sage: f.global_height()                                                    #␣
↪needs sage.symbolic
6.43775164973640
```

is_morphism()

Return `True` if this map is a morphism.

The map is a morphism if and only if the ideal generated by the defining polynomials is the unit ideal (no common zeros of the defining polynomials).

OUTPUT:

- Boolean

EXAMPLES:

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: H = Hom(P, P)
sage: f = H([x^2 + y^2, y^2])
sage: f.is_morphism()                                                      #␣
↪needs sage.libs.singular
True
```

```
sage: P.<x,y,z> = ProjectiveSpace(RR, 2)
sage: H = Hom(P, P)
sage: f = H([x*z - y*z, x^2 - y^2, z^2])
sage: f.is_morphism()                                                      #␣
↪needs sage.libs.singular
False
```

```
sage: R.<t> = PolynomialRing(GF(5))
sage: P.<x,y,z> = ProjectiveSpace(R, 2)
sage: H = Hom(P, P)
sage: f = H([x*z - t*y^2, x^2 - y^2, t*z^2])
sage: f.is_morphism()                                                      #␣
↪needs sage.libs.singular
True
```

Map that is not morphism on projective space, but is over a subscheme:

```
sage: P.<x,y,z> = ProjectiveSpace(RR, 2)
sage: X = P.subscheme([x*y + y*z])
sage: H = Hom(X, X)
```

(continues on next page)

(continued from previous page)

```
sage: f = H([x*z - y*z, x^2 - y^2, z^2])
sage: f.is_morphism()                                                       #␣
↪needs sage.libs.singular
True
```

local_height(*v*, *prec=None*)

Return the maximum of the local height of the coefficients in any of the coordinate functions of this map.

INPUT:

- `v` – a prime or prime ideal of the base ring.
- `prec` – desired floating point precision (default: default RealField precision).

OUTPUT:

- a real number.

EXAMPLES:

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: H = Hom(P, P)
sage: f = H([1/1331*x^2 + 1/4000*y^2, 210*x*y])
sage: f.local_height(1331)                                                  #␣
↪needs sage.rings.real_mpfr
7.19368581839511
```

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: H = Hom(P, P)
sage: f = H([1/1331*x^2 + 1/4000*y^2, 210*x*y])
sage: f.local_height(1331, prec=2)                                          #␣
↪needs sage.rings.real_mpfr
8.0
```

This function does not automatically normalize:

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: H = Hom(P, P)
sage: f = H([4*x^2 + 3/100*y^2, 8/210*x*y, 1/10000*z^2])
sage: f.local_height(2)                                                     #␣
↪needs sage.rings.real_mpfr
2.77258872223978
sage: f.normalize_coordinates()                                             #␣
↪needs sage.libs.singular
sage: f.local_height(2)                                                     #␣
↪needs sage.libs.singular
0.000000000000000
```

```
sage: # needs sage.rings.number_field
sage: R.<z> = PolynomialRing(QQ)
sage: K.<w> = NumberField(z^2 - 2)
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: H = Hom(P, P)
sage: f = H([2*x^2 + w/3*y^2, 1/w*y^2])
```

(continues on next page)

(continued from previous page)

```
sage: f.local_height(K.ideal(3))
1.09861228866811
```

local_height_arch(*i*, *prec=None*)

Return the maximum of the local height at the `i`-th infinite place of the coefficients in any of the coordinate functions of this map.

INPUT:

- `i` – an integer.
- `prec` – desired floating point precision (default: default RealField precision).

OUTPUT:

- a real number.

EXAMPLES:

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: H = Hom(P, P)
sage: f = H([1/1331*x^2 + 1/4000*y^2, 210*x*y])
sage: f.local_height_arch(0)                                                #␣
↪needs sage.rings.real_mpfr
5.34710753071747
```

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: H = Hom(P, P)
sage: f = H([1/1331*x^2 + 1/4000*y^2, 210*x*y])
sage: f.local_height_arch(0, prec=5)                                        #␣
↪needs sage.rings.real_mpfr
5.2
```

```
sage: # needs sage.rings.number_field
sage: R.<z> = PolynomialRing(QQ)
sage: K.<w> = NumberField(z^2 - 2)
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: H = Hom(P, P)
sage: f = H([2*x^2 + w/3*y^2, 1/w*y^2])
sage: f.local_height_arch(1)
0.69314718055994530941723212145820
```

normalize_coordinates(***kwds*)

Ensures that this morphism has integral coefficients. If the coordinate ring has a GCD, then it ensures that the coefficients have no common factor.

It also makes the leading coefficients of the first polynomial positive (if positive has meaning in the coordinate ring). This is done in place.

When `ideal` or `valuation` is specified, normalization occurs with respect to the absolute value defined by the `ideal` or `valuation`. That is, the coefficients are scaled such that one coefficient has absolute value 1 while the others have absolute value less than or equal to 1. Only supported when the base ring is a number field.

INPUT:

kwds:

- `ideal` – (optional) a prime ideal of the base ring of this morphism.
- `valuation` – (optional) a valuation of the base ring of this morphism.

OUTPUT:

- None.

EXAMPLES:

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: H = Hom(P, P)
sage: f = H([5/4*x^3, 5*x*y^2])
sage: f.normalize_coordinates(); f
Scheme endomorphism of Projective Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (x : y) to (x^2 : 4*y^2)
```

```
sage: P.<x,y,z> = ProjectiveSpace(GF(7), 2)
sage: X = P.subscheme(x^2 - y^2)
sage: H = Hom(X, X)
sage: f = H([x^3 + x*y^2, x*y^2, x*z^2])
sage: f.normalize_coordinates(); f                                          #␣
↪needs sage.libs.singular
Scheme endomorphism of Closed subscheme of Projective Space of dimension 2
 over Finite Field of size 7 defined by: x^2 - y^2
  Defn: Defined on coordinates by sending (x : y : z) to (2*y^2 : y^2 : z^2)
```

```
sage: R.<a,b> = QQ[]
sage: P.<x,y,z> = ProjectiveSpace(R, 2)
sage: H = End(P)
sage: f = H([a*(x*z + y^2)*x^2, a*b*(x*z + y^2)*y^2, a*(x*z + y^2)*z^2])
sage: f.normalize_coordinates(); f
Scheme endomorphism of Projective Space of dimension 2 over
 Multivariate Polynomial Ring in a, b over Rational Field
  Defn: Defined on coordinates by sending (x : y : z) to (x^2 : b*y^2 : z^2)
```

```
sage: # needs sage.rings.number_field
sage: K.<w> = QuadraticField(5)
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: f = DynamicalSystem([w*x^2 + (1/5*w)*y^2, w*y^2])
sage: f.normalize_coordinates(); f
Dynamical System of Projective Space of dimension 1 over Number Field in w
 with defining polynomial x^2 - 5 with w = 2.236067977499790?
  Defn: Defined on coordinates by sending (x : y) to (5*x^2 + y^2 : 5*y^2)
```

```
sage: # needs sage.rings.number_field
sage: R.<t> = PolynomialRing(ZZ)
sage: K.<b> = NumberField(t^3 - 11)
sage: a = 7/(b - 1)
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: f = DynamicalSystem_projective([a*y^2 - (a*y - x)^2, y^2])
sage: f.normalize_coordinates(); f
Dynamical System of Projective Space of dimension 1 over
 Number Field in b with defining polynomial t^3 - 11
```

(continues on next page)

(continued from previous page)

```
  Defn: Defined on coordinates by sending (x : y) to
        (-100*x^2 + (140*b^2 + 140*b + 140)*x*y + (-77*b^2 - 567*b - 1057)*y^2
         : 100*y^2)
```

We can used `ideal` to scale with respect to a norm defined by an ideal:

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem_projective([2*x^3, 2*x^2*y + 4*x*y^2])
sage: f.normalize_coordinates(ideal=2); f
Dynamical System of Projective Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (x : y) to (x^3 : x^2*y + 2*x*y^2)
```

```
sage: # needs sage.rings.number_field
sage: R.<w> = QQ[]
sage: A.<a> = NumberField(w^2 + 1)
sage: P.<x,y,z> = ProjectiveSpace(A, 2)
sage: X = P.subscheme(x^2 - y^2)
sage: H = Hom(X, X)
sage: f = H([(a+1)*x^3 + 2*x*y^2, 4*x*y^2, 8*x*z^2])
sage: f.normalize_coordinates(ideal=A.prime_above(2)); f
Scheme endomorphism of Closed subscheme of Projective Space of dimension 2 over
 Number Field in a with defining polynomial w^2 + 1 defined by: x^2 - y^2
  Defn: Defined on coordinates by sending (x : y : z) to
        ((-a + 2)*x*y^2 : (-2*a + 2)*x*y^2 : (-4*a + 4)*x*z^2)
```

We can pass in a valuation to `valuation`:

```
sage: g = H([(a+1)*x^3 + 2*x*y^2, 4*x*y^2, 8*x*z^2])                    #␣
↪needs sage.rings.number_field
sage: g.normalize_coordinates(valuation=A.valuation(A.prime_above(2)))  #␣
↪needs sage.rings.number_field
sage: g == f                                                            #␣
↪needs sage.rings.number_field
True
```

```
sage: P.<x,y> = ProjectiveSpace(Qp(3), 1)                               #␣
↪needs sage.rings.padics
sage: f = DynamicalSystem_projective([3*x^2 + 6*y^2, 9*x*y])            #␣
↪needs sage.rings.padics
sage: f.normalize_coordinates(); f                                      #␣
↪needs sage.rings.padics
Dynamical System of Projective Space of dimension 1 over
 3-adic Field with capped relative precision 20
  Defn: Defined on coordinates by sending (x : y) to
        (x^2 + (2 + O(3^20))*y^2 : (3 + O(3^21))*x*y)
```

Check that #35797 is fixed:

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

(continues on next page)

(continued from previous page)

```
sage: f = DynamicalSystem_projective([a*(z^2 + w^2), z*w])
sage: f.normalize_coordinates(); f
Dynamical System of Projective Space of dimension 1 over
Number Field in a with defining polynomial 3*x^2 + 1
Defn: Defined on coordinates by sending (z : w) to
    ((-3/2*a + 1/2)*z^2 + (-3/2*a + 1/2)*w^2 : (-3/2*a - 3/2)*z*w)
```

```
sage: R.<a,b> = QQ[]
sage: P.<x,y,z> = ProjectiveSpace(FractionField(R), 2)
sage: H = End(P)
sage: f = H([a/b*(x*z + y^2)*x^2, a*b*(x*z + y^2)*y^2, a*(x*z + y^2)*z^2])
sage: f.normalize_coordinates(); f
Scheme endomorphism of Projective Space of dimension 2 over Fraction
Field of Multivariate Polynomial Ring in a, b over Rational Field
Defn: Defined on coordinates by sending (x : y : z) to
    (x^2 : (b^2)*y^2 : b*z^2)
```

scale_by(*t*)

Scales each coordinate by a factor of `t`.

A `TypeError` occurs if the point is not in the coordinate ring of the parent after scaling.

INPUT:

- `t` – a ring element.

OUTPUT:

- None.

EXAMPLES:

```
sage: A.<x,y> = ProjectiveSpace(QQ, 1)
sage: H = Hom(A, A)
sage: f = H([x^3 - 2*x*y^2, x^2*y])
sage: f.scale_by(1/x)
sage: f
Scheme endomorphism of Projective Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (x : y) to (x^2 - 2*y^2 : x*y)
```

```
sage: R.<t> = PolynomialRing(QQ)
sage: P.<x,y> = ProjectiveSpace(R, 1)
sage: H = Hom(P,P)
sage: f = H([3/5*x^2, 6*y^2])
sage: f.scale_by(5/3*t); f
Scheme endomorphism of Projective Space of dimension 1 over
 Univariate Polynomial Ring in t over Rational Field
  Defn: Defined on coordinates by sending (x : y) to (t*x^2 : 10*t*y^2)
```

```
sage: P.<x,y,z> = ProjectiveSpace(GF(7), 2)
sage: X = P.subscheme(x^2 - y^2)
sage: H = Hom(X, X)
sage: f = H([x^2, y^2, z^2])
sage: f.scale_by(x - y); f                                                    #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.libs.singular
Scheme endomorphism of Closed subscheme of Projective Space of dimension 2
 over Finite Field of size 7 defined by: x^2 - y^2
  Defn: Defined on coordinates by sending (x : y : z) to
        (x*y^2 - y^3 : x*y^2 - y^3 : x*z^2 - y*z^2)
```

wronskian_ideal()

Return the ideal generated by the critical point locus.

This is the vanishing of the maximal minors of the Jacobian matrix. Not implemented for subvarieties.

OUTPUT: an ideal in the coordinate ring of the domain of this map.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: R.<x> = PolynomialRing(QQ)
sage: K.<w> = NumberField(x^2 + 11)
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: H = End(P)
sage: f = H([x^2 - w*y^2, w*y^2])
sage: f.wronskian_ideal()
Ideal ((4*w)*x*y) of Multivariate Polynomial Ring in x, y
 over Number Field in w with defining polynomial x^2 + 11
```

```
sage: # needs sage.rings.number_field
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: P2.<u,v,t> = ProjectiveSpace(K, 2)
sage: H = Hom(P, P2)
sage: f = H([x^2 - 2*y^2, y^2, x*y])
sage: f.wronskian_ideal()
Ideal (4*x*y, 2*x^2 + 4*y^2, -2*y^2) of
 Multivariate Polynomial Ring in x, y over Rational Field
```

class sage.schemes.projective.projective_morphism.**SchemeMorphism_polynomial_projective_space_field**(*paren*
polys,
check

Bases: *SchemeMorphism_polynomial_projective_space*

base_indeterminacy_locus()

Return the base indeterminacy locus of this map.

The base indeterminacy locus is the set of points in projective space at which all of the defining polynomials of the rational map simultaneously vanish.

OUTPUT: a subscheme of the domain of the map

EXAMPLES:

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: H = End(P)
sage: f = H([x*z - y*z, x^2 - y^2, z^2])
sage: f.base_indeterminacy_locus()
Closed subscheme of Projective Space of dimension 2 over Rational Field defined␣
↪by:
```

(continues on next page)

(continued from previous page)

```
  x*z - y*z,
  x^2 - y^2,
  z^2
```

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: H = End(P)
sage: f = H([x^2, y^2, z^2])
sage: f.base_indeterminacy_locus()
Closed subscheme of Projective Space of dimension 2 over Rational Field
 defined by:
  x^2,
  y^2,
  z^2
```

```
sage: P1.<x,y,z> = ProjectiveSpace(RR, 2)
sage: P2.<t,u,v,w> = ProjectiveSpace(RR, 3)
sage: H = Hom(P1, P2)
sage: h = H([y^3*z^3, x^3*z^3, y^3*z^3, x^2*y^2*z^2])
sage: h.base_indeterminacy_locus()                                          #␣
↪needs sage.rings.real_mpfr
Closed subscheme of Projective Space of dimension 2 over
 Real Field with 53 bits of precision defined by:
  y^3*z^3,
  x^3*z^3,
  y^3*z^3,
  x^2*y^2*z^2
```

If defining polynomials are not normalized, output scheme will not be normalized:

```
sage: P.<x,y,z> = ProjectiveSpace(QQ,2)
sage: H = End(P)
sage: f = H([x*x^2,x*y^2,x*z^2])
sage: f.base_indeterminacy_locus()
Closed subscheme of Projective Space of dimension 2 over Rational Field
 defined by:
  x^3,
  x*y^2,
  x*z^2
```

image()

Return the scheme-theoretic image of the morphism.

OUTPUT: a subscheme of the ambient space of the codomain

EXAMPLES:

```
sage: P2.<x0,x1,x2> = ProjectiveSpace(QQ, 2)
sage: f = P2.hom([x0^3, x0^2*x1, x0*x1^2], P2)
sage: f.image()                                                             #␣
↪needs sage.libs.singular
Closed subscheme of Projective Space of dimension 2 over Rational Field defined␣
↪by:
  x1^2 - x0*x2
```

(continues on next page)

(continued from previous page)

```
sage: f = P2.hom([x0 - x1, x0 - x2, x1 - x2], P2)
sage: f.image()                                                             #␣
↪needs sage.libs.singular
Closed subscheme of Projective Space of dimension 2 over Rational Field defined␣
↪by:
  x0 - x1 + x2
```

```
sage: P2.<x0,x1,x2> = ProjectiveSpace(QQ, 2)
sage: A2.<x,y> = AffineSpace(QQ, 2)
sage: f = P2.hom([1, x0/x1], A2)
sage: f.image()                                                             #␣
↪needs sage.libs.singular
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  -x + 1
```

indeterminacy_locus()

Return the indeterminacy locus of this map as a rational map on the domain.

The indeterminacy locus is the intersection of all the base indeterminacy locuses of maps that define the same rational map as by this map.

OUTPUT: a subscheme of the domain of the map

EXAMPLES:

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: H = End(P)
sage: f = H([x^2, y^2, z^2])
sage: f.indeterminacy_locus()                                               #␣
↪needs sage.libs.singular
... DeprecationWarning: The meaning of indeterminacy_locus() has changed.
Read the docstring. See https://github.com/sagemath/sage/issues/29145 for␣
↪details.
Closed subscheme of Projective Space of dimension 2 over Rational Field defined␣
↪by:
  z,
  y,
  x
```

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: H = End(P)
sage: f = H([x*z - y*z, x^2 - y^2, z^2])
sage: f.indeterminacy_locus()                                               #␣
↪needs sage.libs.singular
Closed subscheme of Projective Space of dimension 2 over Rational Field defined␣
↪by:
  z,
  x^2 - y^2
```

There is related *base_indeterminacy_locus()* method. This computes the indeterminacy locus only from the defining polynomials of the map:

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: H = End(P)
sage: f = H([x*z - y*z, x^2 - y^2, z^2])
sage: f.base_indeterminacy_locus()
Closed subscheme of Projective Space of dimension 2 over Rational Field defined␣
↪by:
  x*z - y*z,
  x^2 - y^2,
  z^2
```

indeterminacy_points(*F=None*, *base=False*)

Return the points in the indeterminacy locus of this map.

If the dimension of the indeterminacy locus is not zero, an error is raised.

INPUT:

- `F` – a field; if not given, the base ring of the domain is assumed
- `base` – if `True`, the base indeterminacy locus is used

OUTPUT: indeterminacy points of the map defined over `F`

EXAMPLES:

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: H = End(P)
sage: f = H([x*z - y*z, x^2 - y^2, z^2])
sage: f.indeterminacy_points()                                                  #␣
↪needs sage.libs.singular
... DeprecationWarning: The meaning of indeterminacy_locus() has changed.
Read the docstring. See https://github.com/sagemath/sage/issues/29145 for␣
↪details.
[(-1 : 1 : 0), (1 : 1 : 0)]
```

```
sage: P1.<x,y,z> = ProjectiveSpace(RR, 2)
sage: P2.<t,u,v,w> = ProjectiveSpace(RR, 3)
sage: H = Hom(P1, P2)
sage: h = H([x + y, y, z + y, y])
sage: set_verbose(None)
sage: h.indeterminacy_points(base=True)                                         #␣
↪needs sage.libs.singular
[]
sage: g = H([y^3*z^3, x^3*z^3, y^3*z^3, x^2*y^2*z^2])
sage: g.indeterminacy_points(base=True)                                         #␣
↪needs sage.libs.singular
Traceback (most recent call last):
...
ValueError: indeterminacy scheme is not dimension 0
```

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: H = End(P)
sage: f = H([x^2 + y^2, x*z, x^2 + y^2])
sage: f.indeterminacy_points()                                                  #␣
↪needs sage.libs.singular
```

(continues on next page)

(continued from previous page)

```
[(0 : 0 : 1)]

sage: R.<t> = QQ[]
sage: K.<a> = NumberField(t^2 + 1)                                          #␣
↪needs sage.rings.number_field
sage: f.indeterminacy_points(F=K)                                           #␣
↪needs sage.libs.singular sage.rings.number_field
[(-a : 1 : 0), (0 : 0 : 1), (a : 1 : 0)]
sage: set_verbose(None)
sage: f.indeterminacy_points(F=QQbar, base=True)                            #␣
↪needs sage.libs.singular sage.rings.number_field
[(-1*I : 1 : 0), (0 : 0 : 1), (1*I : 1 : 0)]
```

```
sage: set_verbose(None)
sage: K.<t> = FunctionField(QQ)
sage: P.<x,y,z> = ProjectiveSpace(K, 2)
sage: H = End(P)
sage: f = H([x^2 - t^2*y^2, y^2 - z^2, x^2 - t^2*z^2])
sage: f.indeterminacy_points(base=True)                                     #␣
↪needs sage.libs.singular
[(-t : -1 : 1), (-t : 1 : 1), (t : -1 : 1), (t : 1 : 1)]
```

```
sage: # needs sage.rings.padics
sage: set_verbose(None)
sage: P.<x,y,z> = ProjectiveSpace(Qp(3), 2)
sage: H = End(P)
sage: f = H([x^2 - 7*y^2, y^2 - z^2, x^2 - 7*z^2])
sage: f.indeterminacy_points(base=True)                                     #␣
↪needs sage.libs.singular
[(2 + 3 + 3^2 + 2*3^3 + 2*3^5 + 2*3^6 + 3^8
    + 3^9 + 2*3^11 + 3^15 + 2*3^16 + 3^18 + O(3^20) : 1 + O(3^20) : 1 + O(3^
↪20)),
 (2 + 3 + 3^2 + 2*3^3 + 2*3^5 + 2*3^6 + 3^8 + 3^9 + 2*3^11 + 3^15
    + 2*3^16 + 3^18 + O(3^20) : 2 + 2*3 + 2*3^2 + 2*3^3 + 2*3^4 + 2*3^5
    + 2*3^6 + 2*3^7 + 2*3^8 + 2*3^9 + 2*3^10 + 2*3^11 + 2*3^12 + 2*3^13
    + 2*3^14 + 2*3^15 + 2*3^16 + 2*3^17 + 2*3^18 + 2*3^19 + O(3^20) : 1 + O(3^
↪20)),
 (1 + 3 + 3^2 + 2*3^4 + 2*3^7 + 3^8 + 3^9 + 2*3^10 + 2*3^12 + 2*3^13
    + 2*3^14 + 3^15 + 2*3^17 + 3^18 + 2*3^19 + O(3^20) : 1 + O(3^20) : 1 + O(3^
↪20)),
 (1 + 3 + 3^2 + 2*3^4 + 2*3^7 + 3^8 + 3^9 + 2*3^10 + 2*3^12 + 2*3^13
    + 2*3^14 + 3^15 + 2*3^17 + 3^18 + 2*3^19 + O(3^20) : 2 + 2*3 + 2*3^2
    + 2*3^3 + 2*3^4 + 2*3^5 + 2*3^6 + 2*3^7 + 2*3^8 + 2*3^9 + 2*3^10 + 2*3^11
    + 2*3^12 + 2*3^13 + 2*3^14 + 2*3^15 + 2*3^16 + 2*3^17 + 2*3^18 + 2*3^19
    + O(3^20) : 1 + O(3^20))]
```

rational_preimages(*Q*, *k=1*)

Determine all of the rational k-th preimages of `Q` by this map.

Given a rational point `Q` in the domain of this map, return all the rational points `P` in the domain with $f^k(P) == Q$. In other words, the set of k-th preimages of `Q`. The map must be defined over a number field and be an endomorphism for $k > 1$.

If Q is a subscheme, then return the subscheme that maps to Q by this map. In particular, $f^{-k}(V(h_1,\dots,h_t)) = V(h_1 \circ f^k,\dots,h_t \circ f^k)$.

INPUT:

- Q - a rational point or subscheme in the domain of this map.
- k - positive integer.

OUTPUT:

- a list of rational points or a subscheme in the domain of this map.

EXAMPLES:

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: H = End(P)
sage: f = H([16*x^2 - 29*y^2, 16*y^2])
sage: f.rational_preimages(P(-1, 4))                                    #␣
↪needs sage.libs.singular
[(-5/4 : 1), (5/4 : 1)]
```

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: H = End(P)
sage: f = H([76*x^2 - 180*x*y + 45*y^2 + 14*x*z + 45*y*z - 90*z^2,
....:        67*x^2 - 180*x*y - 157*x*z + 90*y*z,
....:        -90*z^2])
sage: f.rational_preimages(P(-9, -4, 1))                                #␣
↪needs sage.libs.singular
[(0 : 4 : 1)]
```

A non-periodic example

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: H = End(P)
sage: f = H([x^2 + y^2, 2*x*y])
sage: f.rational_preimages(P(17, 15))                                   #␣
↪needs sage.libs.singular
[(3/5 : 1), (5/3 : 1)]
```

```
sage: P.<x,y,z,w> = ProjectiveSpace(QQ, 3)
sage: H = End(P)
sage: f = H([x^2 - 2*y*w - 3*w^2, -2*x^2 + y^2 - 2*x*z + 4*y*w + 3*w^2,
....:        x^2 - y^2 + 2*x*z + z^2 - 2*y*w - w^2,
....:        w^2])
sage: f.rational_preimages(P(0, -1, 0, 1))                              #␣
↪needs sage.libs.singular
[]
```

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: H = End(P)
sage: f = H([x^2 + y^2, 2*x*y])
sage: f.rational_preimages([CC.0, 1])                                   #␣
↪needs sage.libs.singular
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
TypeError: point must be in codomain of self
```

A number field example

```
sage: # needs sage.rings.number_field
sage: z = QQ['z'].0
sage: K.<a> = NumberField(z^2 - 2)
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: H = End(P)
sage: f = H([x^2 + y^2, y^2])
sage: f.rational_preimages(P(3, 1))                                             #␣
↪needs sage.libs.singular
[(-a : 1), (a : 1)]
```

```
sage: # needs sage.rings.number_field
sage: z = QQ['z'].0
sage: K.<a> = NumberField(z^2 - 2)
sage: P.<x,y,z> = ProjectiveSpace(K, 2)
sage: X = P.subscheme([x^2 - z^2])
sage: H = End(X)
sage: f= H([x^2 - z^2, a*y^2, z^2 - x^2])
sage: f.rational_preimages(X([1, 2, -1]))                                       #␣
↪needs sage.libs.singular
[]
```

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: X = P.subscheme([x^2 - z^2])
sage: H = End(X)
sage: f = H([x^2-z^2, y^2, z^2-x^2])
sage: f.rational_preimages(X([0, 1, 0]))                                        #␣
↪needs sage.libs.singular
Closed subscheme of Projective Space of dimension 2 over Rational Field defined␣
↪by:
  x^2 - z^2,
  -x^2 + z^2,
  0,
  -x^2 + z^2
```

```
sage: P.<x, y> = ProjectiveSpace(QQ, 1)
sage: H = End(P)
sage: f = H([x^2 - y^2, y^2])
sage: f.rational_preimages(P.subscheme([x]))                                    #␣
↪needs sage.libs.singular
Closed subscheme of Projective Space of dimension 1 over Rational Field
 defined by: x^2 - y^2
```

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: H = End(P)
sage: f = H([x^2 - 29/16*y^2, y^2])
sage: f.rational_preimages(P(5/4, 1), k=4)                                      #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.libs.singular
[(-3/4 : 1), (3/4 : 1), (-7/4 : 1), (7/4 : 1)]
```

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: P2.<u,v,w> = ProjectiveSpace(QQ, 2)
sage: H = Hom(P, P2)
sage: f = H([x^2, y^2, x^2-y^2])
sage: f.rational_preimages(P2(1, 1, 0))                                    #␣
↪needs sage.libs.singular
[(-1 : 1), (1 : 1)]
```

reduce_base_field()

Return this map defined over the field of definition of the coefficients.

The base field of the map could be strictly larger than the field where all of the coefficients are defined. This function reduces the base field to the minimal possible. This can be done when the base ring is a number field, QQbar, a finite field, or algebraic closure of a finite field.

OUTPUT: A scheme morphism.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: K.<t> = GF(3^4)
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: P2.<a,b,c> = ProjectiveSpace(K, 2)
sage: H = End(P)
sage: H2 = Hom(P, P2)
sage: H3 = Hom(P2, P)
sage: f = H([x^2 + (2*t^3 + 2*t^2 + 1)*y^2, y^2])
sage: f.reduce_base_field()                                                #␣
↪needs sage.libs.singular sage.modules
Scheme endomorphism of Projective Space of dimension 1
 over Finite Field in t2 of size 3^2
  Defn: Defined on coordinates by sending (x : y) to (x^2 + t2*y^2 : y^2)
sage: f2 = H2([x^2 + 5*y^2, y^2, 2*x*y])
sage: f2.reduce_base_field()                                               #␣
↪needs sage.libs.singular sage.modules
Scheme morphism:
  From: Projective Space of dimension 1 over Finite Field of size 3
  To:   Projective Space of dimension 2 over Finite Field of size 3
  Defn: Defined on coordinates by sending (x : y) to (x^2 - y^2 : y^2 : -x*y)
sage: f3 = H3([a^2 + t*b^2, c^2])
sage: f3.reduce_base_field()                                               #␣
↪needs sage.libs.singular sage.modules
Scheme morphism:
  From: Projective Space of dimension 2 over Finite Field in t of size 3^4
  To:   Projective Space of dimension 1 over Finite Field in t of size 3^4
  Defn: Defined on coordinates by sending (a : b : c) to (a^2 + t*b^2 : c^2)
```

```
sage: # needs sage.rings.number_field
sage: K.<v> = CyclotomicField(4)
sage: P.<x,y> = ProjectiveSpace(K, 1)
```

(continues on next page)

(continued from previous page)

```
sage: H = End(P)
sage: f = H([x^2 + 2*y^2, y^2])
sage: f.reduce_base_field()                                                 #␣
↪needs sage.libs.singular
Scheme endomorphism of Projective Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (x : y) to (x^2 + 2*y^2 : y^2)
```

```
sage: # needs sage.rings.finite_rings
sage: K.<v> = GF(5)
sage: L = K.algebraic_closure()
sage: P.<x,y> = ProjectiveSpace(L, 1)
sage: H = End(P)
sage: f = H([(L.gen(2))*x^2 + L.gen(4)*y^2, x*y])
sage: f.reduce_base_field()                                                 #␣
↪needs sage.libs.singular
Scheme endomorphism of Projective Space of dimension 1
 over Finite Field in z4 of size 5^4
  Defn: Defined on coordinates by sending (x : y) to
        ((z4^3 + z4^2 + z4 - 2)*x^2 + z4*y^2 : x*y)
sage: f = DynamicalSystem_projective([L.gen(3)*x^2 + L.gen(2)*y^2, x*y])    #␣
↪needs sage.schemes
sage: f.reduce_base_field()                                                 #␣
↪needs sage.libs.singular sage.schemes
Dynamical System of Projective Space of dimension 1
 over Finite Field in z6 of size 5^6
  Defn: Defined on coordinates by sending (x : y) to
        ((-z6^5 + z6^4 - z6^3 - z6^2 - 2*z6 - 2)*x^2
         + (z6^5 - 2*z6^4 + z6^2 - z6 + 1)*y^2 : x*y)
```

class sage.schemes.projective.projective_morphism.**SchemeMorphism_polynomial_projective_space_finite_fie**

Bases: *SchemeMorphism_polynomial_projective_space_field*

class sage.schemes.projective.projective_morphism.**SchemeMorphism_polynomial_projective_subscheme_field**(

Bases: *SchemeMorphism_polynomial_projective_space_field*

Morphisms from subschemes of projective spaces defined over fields.

degree()

Return the degree of this rational map.

EXAMPLES:

```
sage: # needs sage.schemes
sage: k = GF(11)
sage: E = EllipticCurve(k, [1,1])
sage: Q = E(6, 5)
sage: phi = E.scalar_multiplication(2)
sage: mor = phi.as_morphism()
```

(continues on next page)

(continued from previous page)

```
sage: mor.degree()
4
```

graph()

Return the graph of this morphism.

The graph is a subscheme of the product of the ambient spaces of the domain and the codomain. If the ambient space of the codomain is an affine space, it is first embedded into a projective space.

EXAMPLES:

We get the standard quadratic curve as the graph of a quadratic function of an affine line.

```
sage: A1.<x> = AffineSpace(1, QQ)
sage: X = A1.subscheme(0)  # affine line
sage: phi = X.hom([x^2], A1)
sage: mor = phi.homogenize(0)                                                  #␣
↪needs sage.libs.singular
sage: G = mor.graph(); G                                                       #␣
↪needs sage.libs.singular
Closed subscheme of Product of projective spaces P^1 x P^1
  over Rational Field defined by: x1^2*x2 - x0^2*x3
sage: G.affine_patch([0, 0])                                                   #␣
↪needs sage.libs.singular
Closed subscheme of Affine Space of dimension 2
  over Rational Field defined by: x0^2 - x1
```

image()

Return the scheme-theoretic image of the morphism.

EXAMPLES:

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: P2.<x0,x1,x2> = ProjectiveSpace(QQ, 2)
sage: X = P2.subscheme(0)
sage: f = X.hom([x1,x0], P)
sage: f.image()                                                                #␣
↪needs sage.libs.singular
Closed subscheme of Projective Space of dimension 1 over Rational Field defined␣
↪by:
  (no polynomials)
```

```
sage: P2.<x,y,z> = ProjectiveSpace(QQ,2)
sage: X = P2.subscheme([z^3 - x*y^2 + y^3])
sage: f = X.hom([x*z, x*y, x^2 + y*z], P2)
sage: f.image()                                                                #␣
↪needs sage.libs.singular
Closed subscheme of Projective Space of dimension 2 over Rational Field defined␣
↪by:
  x^6 + 2*x^3*y^3 + x*y^5 + y^6 - x^3*y^2*z - y^5*z
```

indeterminacy_locus()

Return the indeterminacy locus of this map.

The map defines a rational map on the domain. The output is the subscheme of the domain on which the rational map is not defined by any representative of the rational map. See *representatives()*.

EXAMPLES:

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: P2.<x0,x1,x2> = ProjectiveSpace(QQ, 2)
sage: X = P2.subscheme(0)
sage: f = X.hom([x1,x0], P)
sage: L = f.indeterminacy_locus()                                          #␣
↪needs sage.libs.singular
sage: L.rational_points()                                                  #␣
↪needs sage.libs.singular
[(0 : 0 : 1)]
```

```
sage: P2.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: P1.<a,b> = ProjectiveSpace(QQ, 1)
sage: X = P2.subscheme([x^2 - y^2 - y*z])
sage: f = X.hom([x,y], P1)
sage: f.indeterminacy_locus()                                              #␣
↪needs sage.libs.singular
Closed subscheme of Projective Space of dimension 2 over Rational Field defined␣
↪by:
  z,
  y,
  x
```

```
sage: P3.<x,y,z,w> = ProjectiveSpace(QQ, 3)
sage: P2.<a,b,c> = ProjectiveSpace(QQ, 2)
sage: X = P3.subscheme(x^2 - w*y - x*z)
sage: f = X.hom([x*y, y*z, z*x], P2)
sage: L = f.indeterminacy_locus()                                          #␣
↪needs sage.libs.singular
sage: L.dimension()                                                        #␣
↪needs sage.libs.singular
0
sage: L.degree()                                                           #␣
↪needs sage.libs.singular
2
sage: L.rational_points()                                                  #␣
↪needs sage.libs.singular
[(0 : 0 : 0 : 1), (0 : 1 : 0 : 0)]
```

```
sage: P3.<x,y,z,w> = ProjectiveSpace(QQ, 3)
sage: A2.<a,b> = AffineSpace(QQ, 2)
sage: X = P3.subscheme(x^2 - w*y - x*z)
sage: f = X.hom([x/z, y/x], A2)
sage: L = f.indeterminacy_locus()                                          #␣
↪needs sage.libs.singular
sage: L.rational_points()                                                  #␣
↪needs sage.libs.singular
[(0 : 0 : 0 : 1), (0 : 1 : 0 : 0)]
```

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: X = P.subscheme(x - y)
sage: H = End(X)
sage: f = H([x^2 - 4*y^2, y^2 - z^2, 4*z^2 - x^2])
sage: Z = f.indeterminacy_locus(); Z                                    #␣
↪needs sage.libs.singular
Closed subscheme of Projective Space of dimension 2 over Rational Field defined␣
↪by:
  z,
  y,
  x
```

is_morphism()

Return True if the map is defined everywhere on the domain.

EXAMPLES:

```
sage: P2.<x,y,z> = ProjectiveSpace(QQ,2)
sage: P1.<a,b> = ProjectiveSpace(QQ, 1)
sage: X = P2.subscheme([x^2 - y^2 - y*z])
sage: f = X.hom([x,y], P1)
sage: f.is_morphism()                                                   #␣
↪needs sage.libs.singular
True
```

projective_degrees()

Return the projective degrees of this rational map.

EXAMPLES:

```
sage: # needs sage.schemes
sage: k = GF(11)
sage: E = EllipticCurve(k, [1,1])
sage: Q = E(6, 5)
sage: phi = E.scalar_multiplication(2)
sage: mor = phi.as_morphism()
sage: mor.projective_degrees()
(12, 3)
```

representatives()

Return all maps representing the same rational map as by this map.

EXAMPLES:

```
sage: P2.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: X = P2.subscheme(0)
sage: f = X.hom([x^2*y, x^2*z, x*y*z], P2)
sage: f.representatives()                                               #␣
↪needs sage.libs.singular
[Scheme morphism:
   From: Closed subscheme of Projective Space of dimension 2
         over Rational Field defined by: 0
   To:   Projective Space of dimension 2 over Rational Field
   Defn: Defined on coordinates by sending (x : y : z) to (x*y : x*z : y*z)]
```

```
sage: P2.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: P1.<a,b> = ProjectiveSpace(QQ, 1)
sage: X = P2.subscheme([x^2 - y^2 - y*z])
sage: f = X.hom([x, y], P1)
sage: f.representatives()                                                   #␣
↪needs sage.libs.singular
[Scheme morphism:
   From: Closed subscheme of Projective Space of dimension 2
         over Rational Field defined by: x^2 - y^2 - y*z
   To:   Projective Space of dimension 1 over Rational Field
   Defn: Defined on coordinates by sending (x : y : z) to (y + z : x),
 Scheme morphism:
   From: Closed subscheme of Projective Space of dimension 2
         over Rational Field defined by: x^2 - y^2 - y*z
   To:   Projective Space of dimension 1 over Rational Field
   Defn: Defined on coordinates by sending (x : y : z) to (x : y)]
sage: g = _[0]                                                              #␣
↪needs sage.libs.singular
sage: g.representatives()                                                   #␣
↪needs sage.libs.singular
[Scheme morphism:
   From: Closed subscheme of Projective Space of dimension 2
         over Rational Field defined by: x^2 - y^2 - y*z
   To:   Projective Space of dimension 1 over Rational Field
   Defn: Defined on coordinates by sending (x : y : z) to (y + z : x),
 Scheme morphism:
   From: Closed subscheme of Projective Space of dimension 2
         over Rational Field defined by: x^2 - y^2 - y*z
   To:   Projective Space of dimension 1 over Rational Field
   Defn: Defined on coordinates by sending (x : y : z) to (x : y)]
```

```
sage: P2.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: X = P2.subscheme([x^2 - y^2 - y*z])
sage: A1.<a> = AffineSpace(QQ, 1)
sage: g = X.hom([y/x], A1)
sage: g.representatives()                                                   #␣
↪needs sage.libs.singular
[Scheme morphism:
   From: Closed subscheme of Projective Space of dimension 2
         over Rational Field defined by: x^2 - y^2 - y*z
   To:   Affine Space of dimension 1 over Rational Field
   Defn: Defined on coordinates by sending (x : y : z) to (x/(y + z)),
 Scheme morphism:
   From: Closed subscheme of Projective Space of dimension 2
         over Rational Field defined by: x^2 - y^2 - y*z
   To:   Affine Space of dimension 1 over Rational Field
   Defn: Defined on coordinates by sending (x : y : z) to (y/x)]
sage: g0, g1 = _                                                            #␣
↪needs sage.libs.singular
sage: emb = A1.projective_embedding(0)
sage: emb*g0                                                                #␣
↪needs sage.libs.singular
```

(continues on next page)

(continued from previous page)

```
Scheme morphism:
  From: Closed subscheme of Projective Space of dimension 2
        over Rational Field defined by: x^2 - y^2 - y*z
  To:   Projective Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (x : y : z) to (y + z : x)
sage: emb*g1                                                                #␣
↪needs sage.libs.singular
Scheme morphism:
  From: Closed subscheme of Projective Space of dimension 2
        over Rational Field defined by: x^2 - y^2 - y*z
  To:   Projective Space of dimension 1 over Rational Field
  Defn: Defined on coordinates by sending (x : y : z) to (x : y)
```

ALGORITHM:

The algorithm is from Proposition 1.1 in [Sim2004].

14.3 Points on projective varieties

Scheme morphism for points on projective varieties

AUTHORS:

- David Kohel, William Stein
- William Stein (2006-02-11): fixed bug where P(0,0,0) was allowed as a projective point.
- Volker Braun (2011-08-08): Renamed classes, more documentation, misc cleanups.
- Ben Hutz (June 2012) added support for projective ring; (March 2013) iteration functionality and new directory structure for affine/projective, height functionality

class sage.schemes.projective.projective_point.**SchemeMorphism_point_abelian_variety_field**(*X*, *v*, *check=True*)

Bases: AdditiveGroupElement, *SchemeMorphism_point_projective_field*

A rational point of an abelian variety over a field.

EXAMPLES:

```
sage: # needs sage.schemes
sage: E = EllipticCurve([0,0,1,-1,0])
sage: origin = E(0)
sage: origin.domain()
Spectrum of Rational Field
sage: origin.codomain()
Elliptic Curve defined by y^2 + y = x^3 - x over Rational Field
```

class sage.schemes.projective.projective_point.**SchemeMorphism_point_projective_field**(*X*, *v*, *check=True*)

Bases: *SchemeMorphism_point_projective_ring*

A rational point of projective space over a field.

INPUT:

- X – a homset of a subscheme of an ambient projective space over a field K.
- v – a list or tuple of coordinates in K.
- check – boolean (optional, default:True). Whether to check the input for consistency.

EXAMPLES:

```
sage: # needs sage.rings.real_mpfr
sage: P = ProjectiveSpace(3, RR)
sage: P(2, 3, 4, 5)
(0.400000000000000 : 0.600000000000000 : 0.800000000000000 : 1.00000000000000)
```

clear_denominators()

scales by the least common multiple of the denominators.

OUTPUT: None.

EXAMPLES:

```
sage: R.<t> = PolynomialRing(QQ)
sage: P.<x,y,z> = ProjectiveSpace(FractionField(R), 2)
sage: Q = P([t, 3/t^2, 1])
sage: Q.clear_denominators(); Q
(t^3 : 3 : t^2)
```

```
sage: # needs sage.rings.number_field
sage: R.<x> = PolynomialRing(QQ)
sage: K.<w> = NumberField(x^2 - 3)
sage: P.<x,y,z> = ProjectiveSpace(K, 2)
sage: Q = P([1/w, 3, 0])
sage: Q.clear_denominators(); Q
(w : 9 : 0)
```

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: X = P.subscheme(x^2 - y^2)
sage: Q = X([1/2, 1/2, 1])
sage: Q.clear_denominators(); Q
(1 : 1 : 2)
```

```
sage: PS.<x,y> = ProjectiveSpace(QQ, 1)
sage: Q = PS.point([1, 2/3], False); Q
(1 : 2/3)
sage: Q.clear_denominators(); Q
(3 : 2)
```

intersection_multiplicity(X)

Return the intersection multiplicity of the codomain of this point and X at this point.

This uses the intersection_multiplicity implementations for projective/affine subschemes. This point must be a point of a projective subscheme.

INPUT:

- X – a subscheme in the same ambient space as that of the codomain of this point.

OUTPUT: Integer.

EXAMPLES:

```
sage: P.<x,y,z,w> = ProjectiveSpace(QQ, 3)
sage: X = P.subscheme([x*z - y^2])
sage: Y = P.subscheme([x^3 - y*w^2 + z*w^2, x*y - z*w])
sage: Q1 = X([1/2, 1/4, 1/8, 1])
sage: Q1.intersection_multiplicity(Y)                                    #␣
↪needs sage.libs.singular
1
sage: Q2 = X([0,0,0,1])
sage: Q2.intersection_multiplicity(Y)                                    #␣
↪needs sage.libs.singular
5
sage: Q3 = X([0,0,1,0])
sage: Q3.intersection_multiplicity(Y)                                    #␣
↪needs sage.libs.singular
6
```

```
sage: P.<x,y,z,w> = ProjectiveSpace(QQ, 3)
sage: X = P.subscheme([x^2 - y^2])
sage: Q = P([1,1,1,0])
sage: Q.intersection_multiplicity(X)
Traceback (most recent call last):
...
TypeError: this point must be a point on a projective subscheme
```

multiplicity()

Return the multiplicity of this point on its codomain.

Uses the subscheme multiplicity implementation. This point must be a point on a projective subscheme.

OUTPUT: an integer.

EXAMPLES:

```
sage: P.<x,y,z,w,t> = ProjectiveSpace(QQ, 4)
sage: X = P.subscheme([y^6 - x^3*w^2*t + t^5*w, x^2 - t^2])
sage: Q1 = X([1,0,2,1,1])
sage: Q1.multiplicity()                                                  #␣
↪needs sage.libs.singular
1
sage: Q2 = X([0,0,-2,1,0])
sage: Q2.multiplicity()                                                  #␣
↪needs sage.libs.singular
8
```

normalize_coordinates()

Normalizes the point so that the last non-zero coordinate is 1.

OUTPUT: None.

EXAMPLES:

```
sage: P.<x,y,z> = ProjectiveSpace(GF(5), 2)
sage: Q = P.point([GF(5)(1), GF(5)(3), GF(5)(0)], False); Q
(1 : 3 : 0)
```

(continues on next page)

(continued from previous page)

```
sage: Q.normalize_coordinates(); Q
(2 : 1 : 0)
```

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: X = P.subscheme(x^2 - y^2);
sage: Q = X.point([23, 23, 46], False); Q
(23 : 23 : 46)
sage: Q.normalize_coordinates(); Q
(1/2 : 1/2 : 1)
```

class sage.schemes.projective.projective_point.**SchemeMorphism_point_projective_finite_field**(*X*, *v*, *check=True*)

Bases: *SchemeMorphism_point_projective_field*

class sage.schemes.projective.projective_point.**SchemeMorphism_point_projective_ring**(*X*, *v*, *check=True*)

Bases: *SchemeMorphism_point*

A rational point of projective space over a ring.

INPUT:

- `X` – a homset of a subscheme of an ambient projective space over a ring K.
- `v` – a list or tuple of coordinates in K.
- `check` – boolean (optional, default:`True`). Whether to check the input for consistency.

EXAMPLES:

```
sage: P = ProjectiveSpace(2, ZZ)
sage: P(2,3,4)
(2 : 3 : 4)
```

dehomogenize(*n*)

Dehomogenizes at the nth coordinate.

INPUT:

- `n` – non-negative integer.

OUTPUT:

- `SchemeMorphism_point_affine`.

EXAMPLES:

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: X = P.subscheme(x^2 - y^2)
sage: Q = X(23, 23, 46)
sage: Q.dehomogenize(2)                                                    #␣
↪needs sage.libs.singular
(1/2, 1/2)
```

```
sage: # needs sage.libs.pari
sage: R.<t> = PolynomialRing(QQ)
sage: S = R.quo(R.ideal(t^3))
sage: P.<x,y,z> = ProjectiveSpace(S, 2)
sage: Q = P(t, 1, 1)
sage: Q.dehomogenize(1)
(tbar, 1)
```

```
sage: P.<x,y,z> = ProjectiveSpace(GF(5), 2)
sage: Q = P(1, 3, 1)
sage: Q.dehomogenize(0)
(3, 1)
```

```
sage: P.<x,y,z> = ProjectiveSpace(GF(5), 2)
sage: Q = P(1, 3, 0)
sage: Q.dehomogenize(2)
Traceback (most recent call last):
...
ValueError: can...t dehomogenize at 0 coordinate
```

global_height(*prec=None*)

Return the absolute logarithmic height of the point.

INPUT:

- prec – desired floating point precision (default: default RealField precision).

OUTPUT:

- a real number.

EXAMPLES:

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: Q = P.point([4, 4, 1/30])
sage: Q.global_height()                                                     #␣
↪needs sage.symbolic
4.78749174278205
```

```
sage: P.<x,y,z> = ProjectiveSpace(ZZ, 2)
sage: Q = P([4, 1, 30])
sage: Q.global_height()                                                     #␣
↪needs sage.symbolic
3.40119738166216
```

```
sage: R.<x> = PolynomialRing(QQ)
sage: k.<w> = NumberField(x^2 + 5)                                          #␣
↪needs sage.rings.number_field
sage: A = ProjectiveSpace(k, 2, 'z')                                        #␣
↪needs sage.rings.number_field
sage: A([3, 5*w + 1, 1]).global_height(prec=100)                            #␣
↪needs sage.rings.number_field
2.4181409534757389986565376694
```

```
sage: P.<x,y,z> = ProjectiveSpace(QQbar, 2)                                    #␣
↪needs sage.rings.number_field
sage: Q = P([QQbar(sqrt(3)), QQbar(sqrt(-2)), 1])                              #␣
↪needs sage.rings.number_field sage.symbolic
sage: Q.global_height()                                                        #␣
↪needs sage.rings.number_field sage.symbolic
0.549306144334055
```

```
sage: # needs sage.rings.number_field
sage: K = UniversalCyclotomicField()
sage: P.<x,y,z> = ProjectiveSpace(K, 2)
sage: Q = P.point([K(4/3), K.gen(7), K.gen(5)])
sage: Q.global_height()
1.38629436111989
```

is_preperiodic(*f*, *err=0.1*, *return_period=False*)

Determine if the point is preperiodic with respect to the map `f`.

This is implemented for both projective space and subschemes. There are two optional keyword arguments: `error_bound` sets the error_bound used in the canonical height computation and `return_period` a boolean which controls if the period is returned if the point is preperiodic. If `return_period` is `True` and this point is not preperiodic, then $(0, 0)$ is returned for the period.

ALGORITHM:

We know that a point is preperiodic if and only if it has canonical height zero. However, we can only compute the canonical height up to numerical precision. This function first computes the canonical height of the point to the given error bound. If it is larger than that error bound, then it must not be preperiodic. If it is less than the error bound, then we expect preperiodic. In this case we begin computing the orbit stopping if either we determine the orbit is finite, or the height of the point is large enough that it must be wandering. We can determine the height cutoff by computing the height difference constant, i.e., the bound between the height and the canonical height of a point (which depends only on the map and not the point itself). If the height of the point is larger than the difference bound, then the canonical height cannot be zero so the point cannot be preperiodic.

INPUT:

- `f` – an endomorphism of this point's codomain.

kwds:

- `err` – a positive real number (optional - default: 0.1).
- `return_period` – boolean (optional - default: `False`).

OUTPUT:

- boolean – `True` if preperiodic.
- if `return_period` is `True`, then `(0,0)` if wandering, and `(m,n)` if preperiod `m` and period `n`.

EXAMPLES:

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem_projective([x^3 - 3*x*y^2, y^3], domain=P)           #␣
↪needs sage.schemes
sage: Q = P(-1, 1)
sage: Q.is_preperiodic(f)                                                      #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.libs.singular sage.schemes
True
```

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: X = P.subscheme(z)
sage: f = DynamicalSystem([x^2 - y^2, y^2, z^2], domain=X)                    #␣
↪needs sage.schemes
sage: p = X((-1, 1, 0))
sage: p.is_preperiodic(f, return_period=True)                                 #␣
↪needs sage.libs.singular sage.schemes
(0, 2)
```

```
sage: P.<x,y> = ProjectiveSpace(QQ,1)
sage: f = DynamicalSystem_projective([x^2 - 29/16*y^2, y^2], domain=P)        #␣
↪needs sage.schemes
sage: Q = P(1, 4)
sage: Q.is_preperiodic(f, return_period=True)                                 #␣
↪needs sage.libs.singular sage.schemes
(1, 3)
sage: Q = P(1, 1)
sage: Q.is_preperiodic(f, return_period=True)                                 #␣
↪needs sage.libs.singular sage.schemes
(0, 0)
```

```
sage: # needs sage.rings.number_field
sage: R.<x> = PolynomialRing(QQ)
sage: K.<a> = NumberField(x^2 + 1)
sage: P.<x,y> = ProjectiveSpace(K, 1)
sage: f = DynamicalSystem_projective([x^5 + 5/4*x*y^4, y^5], domain=P)        #␣
↪needs sage.schemes
sage: Q = P([-1/2*a + 1/2, 1])
sage: Q.is_preperiodic(f)                                                     #␣
↪needs sage.schemes
True
sage: Q = P([a, 1])
sage: Q.is_preperiodic(f)                                                     #␣
↪needs sage.schemes
False
```

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: f = DynamicalSystem_projective([                                        #␣
↪needs sage.schemes
....:         -38/45*x^2 + (2*y - 7/45*z)*x + (-1/2*y^2 - 1/2*y*z + z^2),
....:         -67/90*x^2 + (2*y + z*157/90)*x - y*z,
....:         z^2
....:     ], domain=P)
sage: Q = P([1, 3, 1])
sage: Q.is_preperiodic(f, return_period=True)                                 #␣
↪needs sage.libs.singular sage.schemes
(0, 9)
```

```
sage: P.<x,y,z,w> = ProjectiveSpace(QQ, 3)
sage: f = DynamicalSystem_projective([                                    #␣
↪needs sage.schemes
....:          (-y - w)*x + (-13/30*y^2 + 13/30*w*y + w^2),
....:          -1/2*x^2 + (-y + 3/2*w)*x + (-1/3*y^2 + 4/3*w*y),
....:          -3/2*z^2 + 5/2*z*w + w^2,
....:          w^2
....:      ], domain=P)
sage: Q = P([3,0,4/3,1])
sage: Q.is_preperiodic(f, return_period=True)                             #␣
↪needs sage.libs.singular sage.schemes
(2, 24)
```

```
sage: # needs sage.rings.number_field sage.schemes sage.symbolic
sage: from sage.misc.verbose import set_verbose
sage: set_verbose(-1)
sage: P.<x,y,z> = ProjectiveSpace(QQbar, 2)
sage: f = DynamicalSystem_projective([x^2, QQbar(sqrt(-1))*y^2, z^2],
....:                                domain=P)
sage: Q = P([1, 1, 1])
sage: Q.is_preperiodic(f)
True
```

```
sage: # needs sage.rings.number_field sage.schemes sage.symbolic
sage: set_verbose(-1)
sage: P.<x,y,z> = ProjectiveSpace(QQbar, 2)
sage: f = DynamicalSystem_projective([x^2, y^2, z^2], domain=P)
sage: Q = P([QQbar(sqrt(-1)), 1, 1])
sage: Q.is_preperiodic(f)
True
```

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: f = DynamicalSystem_projective([16*x^2 - 29*y^2, 16*y^2], domain=P)  #␣
↪needs sage.schemes
sage: Q = P(-1,4)
sage: Q.is_preperiodic(f)                                                 #␣
↪needs sage.libs.singular sage.schemes
True
```

```
sage: P.<x,y,z> = ProjectiveSpace(GF(3), 2)
sage: F = DynamicalSystem([x^2 - 2*y^2, y^2, z^2])                        #␣
↪needs sage.schemes
sage: Q = P(1, 1, 1)
sage: Q.is_preperiodic(F, return_period=True)                             #␣
↪needs sage.schemes
(1, 1)
```

local_height(*v*, *prec=None*)

Returns the maximum of the local height of the coordinates of this point.

INPUT:

- v – a prime or prime ideal of the base ring.

- prec – desired floating point precision (default: default RealField precision).

OUTPUT:

- a real number.

EXAMPLES:

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: Q = P.point([4, 4, 1/150], False)
sage: Q.local_height(5)                                                     #␣
↪needs sage.rings.real_mpfr
3.21887582486820
```

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: Q = P([4, 1, 30])
sage: Q.local_height(2)                                                     #␣
↪needs sage.rings.real_mpfr
0.693147180559945
```

local_height_arch(*i*, *prec=None*)

Returns the maximum of the local heights at the i-th infinite place of this point.

INPUT:

- i – an integer.
- prec – desired floating point precision (default: default RealField precision).

OUTPUT:

- a real number.

EXAMPLES:

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: Q = P.point([4, 4, 1/150], False)
sage: Q.local_height_arch(0)                                                #␣
↪needs sage.rings.real_mpfr
1.38629436111989
```

```
sage: # needs sage.rings.number_field
sage: P.<x,y,z> = ProjectiveSpace(QuadraticField(5, 'w'), 2)
sage: Q = P.point([4, 1, 30], False)
sage: Q.local_height_arch(1)
3.4011973816621553754132366916O7
```

multiplier(*f*, *n*, *check=True*)

Returns the multiplier of this point of period n by the function f.

f must be an endomorphism of projective space.

INPUT:

- f - a endomorphism of this point's codomain.
- n - a positive integer, the period of this point.
- check – check if P is periodic of period n, Default:True.

OUTPUT:

- a square matrix of size `self.codomain().dimension_relative()` in the `base_ring` of this point.

EXAMPLES:

```
sage: P.<x,y,z,w> = ProjectiveSpace(QQ, 3)
sage: f = DynamicalSystem_projective([x^2, y^2, 4*w^2, 4*z^2], domain=P)     #␣
↪needs sage.schemes
sage: Q = P.point([4, 4, 1, 1], False)
sage: Q.multiplier(f, 1)                                                     #␣
↪needs sage.schemes
[ 2  0 -8]
[ 0  2 -8]
[ 0  0 -2]
```

normalize_coordinates()

Removes the gcd from the coordinates of this point (including -1).

Warning: The gcd will depend on the base ring.

OUTPUT: None.

EXAMPLES:

```
sage: P = ProjectiveSpace(ZZ, 2, 'x')
sage: p = P([-5, -15, -20])
sage: p.normalize_coordinates(); p
(1 : 3 : 4)
```

```
sage: # needs sage.rings.padics
sage: P = ProjectiveSpace(Zp(7), 2, 'x')
sage: p = P([-5, -15, -2])
sage: p.normalize_coordinates(); p
(5 + O(7^20) : 1 + 2*7 + O(7^20) : 2 + O(7^20))
```

```
sage: R.<t> = PolynomialRing(QQ)
sage: P = ProjectiveSpace(R, 2, 'x')
sage: p = P([3/5*t^3, 6*t, t])
sage: p.normalize_coordinates(); p
(3/5*t^2 : 6 : 1)
```

```
sage: P.<x,y> = ProjectiveSpace(Zmod(20), 1)
sage: Q = P(3, 6)
sage: Q.normalize_coordinates()
sage: Q
(1 : 2)
```

Since the base ring is a polynomial ring over a field, only the gcd c is removed.

```
sage: R.<c> = PolynomialRing(QQ)
sage: P = ProjectiveSpace(R, 1)
sage: Q = P(2*c, 4*c)
sage: Q.normalize_coordinates();Q
(2 : 4)
```

A polynomial ring over a ring gives the more intuitive result.

```
sage: R.<c> = PolynomialRing(ZZ)
sage: P = ProjectiveSpace(R, 1)
sage: Q = P(2*c, 4*c)
sage: Q.normalize_coordinates();Q
(1 : 2)
```

```
sage: # needs sage.libs.singular
sage: R.<t> = PolynomialRing(QQ, 1)
sage: S = R.quotient_ring(R.ideal(t^3))
sage: P.<x,y> = ProjectiveSpace(S, 1)
sage: Q = P(t + 1, t^2 + t)
sage: Q.normalize_coordinates()
sage: Q
(1 : tbar)
```

scale_by(*t*)

Scale the coordinates of the point by `t`.

A `TypeError` occurs if the point is not in the base_ring of the codomain after scaling.

INPUT:

- `t` – a ring element.

OUTPUT: None.

EXAMPLES:

```
sage: R.<t> = PolynomialRing(QQ)
sage: P = ProjectiveSpace(R, 2, 'x')
sage: p = P([3/5*t^3, 6*t, t])
sage: p.scale_by(1/t); p
(3/5*t^2 : 6 : 1)
```

```
sage: # needs sage.libs.pari
sage: R.<t> = PolynomialRing(QQ)
sage: S = R.quo(R.ideal(t^3))
sage: P.<x,y,z> = ProjectiveSpace(S, 2)
sage: Q = P(t, 1, 1)
sage: Q.scale_by(t);Q
(tbar^2 : tbar : tbar)
```

```
sage: P.<x,y,z> = ProjectiveSpace(ZZ,2)
sage: Q = P(2, 2, 2)
sage: Q.scale_by(1/2);Q
(1 : 1 : 1)
```

14.4 Subschemes of projective space

AUTHORS:

- David Kohel (2005): initial version.
- William Stein (2005): initial version.
- Volker Braun (2010-12-24): documentation of schemes and refactoring. Added coordinate neighborhoods and is_smooth()
- Ben Hutz (2013) refactoring

class sage.schemes.projective.projective_subscheme.**AlgebraicScheme_subscheme_projective**(*A*, *polynomials*)

Bases: *AlgebraicScheme_subscheme*

Construct an algebraic subscheme of projective space.

Warning: You should not create objects of this class directly. The preferred method to construct such subschemes is to use `subscheme()` method of *projective space*.

INPUT:

- `A` – ambient *projective space*.
- `polynomials` – single polynomial, ideal or iterable of defining homogeneous polynomials.

EXAMPLES:

```
sage: P.<x, y, z> = ProjectiveSpace(2, QQ)
sage: P.subscheme([x^2 - y*z])
Closed subscheme of Projective Space of dimension 2 over Rational Field defined by:
  x^2 - y*z
```

affine_patch(*i*, *AA=None*)

Return the i^{th} affine patch of this projective scheme.

This is the intersection with this i^{th} affine patch of its ambient space.

INPUT:

- `i` – integer between 0 and dimension of self, inclusive.
- **`AA` – (default: None) ambient affine space, this is constructed** if it is not given.

OUTPUT:

An affine algebraic scheme with fixed `embedding_morphism()` equal to the default `projective_embedding()` map`.

EXAMPLES:

```
sage: # needs sage.libs.singular
sage: PP = ProjectiveSpace(2, QQ, names='X,Y,Z')
sage: X,Y,Z = PP.gens()
sage: C = PP.subscheme(X^3*Y + Y^3*Z + Z^3*X)
sage: U = C.affine_patch(0); U
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  Y^3*Z + Z^3 + Y
sage: U.embedding_morphism()
Scheme morphism:
  From: Closed subscheme of Affine Space of dimension 2 over Rational Field
        defined by: Y^3*Z + Z^3 + Y
  To:   Closed subscheme of Projective Space of dimension 2 over Rational Field
        defined by: X^3*Y + Y^3*Z + X*Z^3
  Defn: Defined on coordinates by sending (Y, Z) to (1 : Y : Z)
sage: U.projective_embedding() is U.embedding_morphism()
True
```

```
sage: A.<x,y,z> = AffineSpace(QQ, 3)
sage: X = A.subscheme([x - y*z])
sage: Y = X.projective_embedding(1).codomain()                                  #␣
↪needs sage.libs.singular
sage: Y.affine_patch(1, A).ambient_space() == A                                 #␣
↪needs sage.libs.singular
True
```

```
sage: P.<u,v,w> = ProjectiveSpace(2, ZZ)
sage: S = P.subscheme([u^2 - v*w])
sage: A.<x, y> = AffineSpace(2, ZZ)
sage: S.affine_patch(1, A)                                                      #␣
↪needs sage.libs.singular
Closed subscheme of Affine Space of dimension 2 over Integer Ring defined by:
  x^2 - y
```

degree()

Return the degree of this projective subscheme.

If $P(t) = a_m t^m + \ldots + a_0$ is the Hilbert polynomial of this subscheme, then the degree is $a_m m!$.

OUTPUT: Integer.

EXAMPLES:

```
sage: P.<x,y,z,w,t,u> = ProjectiveSpace(QQ, 5)
sage: X = P.subscheme([x^7 + x*y*z*t^4 - u^7])
sage: X.degree()                                                                #␣
↪needs sage.libs.singular
7

sage: P.<x,y,z,w> = ProjectiveSpace(GF(13), 3)
sage: X = P.subscheme([y^3 - w^3, x + 7*z])
sage: X.degree()                                                                #␣
↪needs sage.libs.singular
3
```

(continues on next page)

(continued from previous page)

```
sage: # needs sage.libs.singular sage.schemes
sage: P.<x,y,z,w,u> = ProjectiveSpace(QQ, 4)
sage: C = P.curve([x^7 - y*z^3*w^2*u, w*x^2 - y*u^2, z^3 + y^3])
sage: C.degree()
63
```

dimension()

Return the dimension of the projective algebraic subscheme.

OUTPUT:

Integer.

EXAMPLES:

```
sage: # needs sage.libs.singular
sage: P2.<x,y,z> = ProjectiveSpace(2, QQ)
sage: P2.subscheme([]).dimension()
2
sage: P2.subscheme([x]).dimension()
1
sage: P2.subscheme([x^5]).dimension()
1
sage: P2.subscheme([x^2 + y^2 - z^2]).dimension()
1
sage: P2.subscheme([x*(x-z), y*(y-z)]).dimension()
0
```

Something less obvious:

```
sage: P3.<x,y,z,w,t> = ProjectiveSpace(4, QQ)
sage: X = P3.subscheme([x^2, x^2*y^2 + z^2*t^2, z^2 - w^2, 10*x^2 + w^2 - z^2])
sage: X
Closed subscheme of Projective Space of dimension 4 over Rational Field defined␣
↪by:
  x^2,
  x^2*y^2 + z^2*t^2,
  z^2 - w^2,
  10*x^2 - z^2 + w^2
sage: X.dimension()                                                             #␣
↪needs sage.libs.singular
1
```

dual()

Return the projective dual of the given subscheme of projective space.

INPUT:

- `X` – A subscheme of projective space. At present, `X` is required to be an irreducible and reduced hypersurface defined over $\mathbf{Q}$ or a finite field.

OUTPUT:

- The dual of `X` as a subscheme of the dual projective space.

EXAMPLES:

The dual of a smooth conic in the plane is also a smooth conic:

```
sage: R.<x, y, z> = QQ[]
sage: P.<x, y, z> = ProjectiveSpace(2, QQ)
sage: I = R.ideal(x^2 + y^2 + z^2)
sage: X = P.subscheme(I)
sage: X.dual()                                                              #␣
↪needs sage.libs.singular
Closed subscheme of Projective Space of dimension 2 over Rational Field
 defined by:
  y0^2 + y1^2 + y2^2
```

The dual of the twisted cubic curve in projective 3-space is a singular quartic surface. In the following example, we compute the dual of this surface, which by double duality is equal to the twisted cubic itself. The output is the twisted cubic as an intersection of three quadrics:

```
sage: R.<x, y, z, w> = QQ[]
sage: P.<x, y, z, w> = ProjectiveSpace(3, QQ)
sage: I = R.ideal(y^2*z^2 - 4*x*z^3 - 4*y^3*w + 18*x*y*z*w - 27*x^2*w^2)
sage: X = P.subscheme(I)
sage: X.dual()                                                              #␣
↪needs sage.libs.singular
Closed subscheme of Projective Space of dimension 3 over
 Rational Field defined by:
  y2^2 - y1*y3,
  y1*y2 - y0*y3,
  y1^2 - y0*y2
```

The singular locus of the quartic surface in the last example is itself supported on a twisted cubic:

```
sage: X.Jacobian().radical()                                                #␣
↪needs sage.libs.singular
Ideal (z^2 - 3*y*w, y*z - 9*x*w, y^2 - 3*x*z) of Multivariate
Polynomial Ring in x, y, z, w over Rational Field
```

An example over a finite field:

```
sage: R = PolynomialRing(GF(61), 'a,b,c')
sage: P.<a, b, c> = ProjectiveSpace(2, R.base_ring())
sage: X = P.subscheme(R.ideal(a*a + 2*b*b + 3*c*c))
sage: X.dual()                                                              #␣
↪needs sage.libs.singular sage.rings.finite_rings
Closed subscheme of Projective Space of dimension 2 over
 Finite Field of size 61 defined by:
  y0^2 - 30*y1^2 - 20*y2^2
```

intersection_multiplicity(*X*, *P*)

Return the intersection multiplicity of this subscheme and the subscheme `X` at the point `P`.

This uses the intersection_multiplicity function for affine subschemes on affine patches of this subscheme and `X` that contain `P`.

INPUT:

- `X` – subscheme in the same ambient space as this subscheme.
- `P` – a point in the intersection of this subscheme with `X`.

OUTPUT: An integer.

EXAMPLES:

```
sage: # needs sage.schemes
sage: P.<x,y,z> = ProjectiveSpace(GF(5), 2)
sage: C = Curve([x^4 - z^2*y^2], P)
sage: D = Curve([y^4*z - x^5 - x^3*z^2], P)
sage: Q1 = P([0,1,0])
sage: C.intersection_multiplicity(D, Q1)                                    #␣
↪needs sage.libs.singular
4
sage: Q2 = P([0,0,1])
sage: C.intersection_multiplicity(D, Q2)                                    #␣
↪needs sage.libs.singular
6
```

```
sage: # needs sage.rings.number_field
sage: R.<a> = QQ[]
sage: K.<b> = NumberField(a^4 + 1)
sage: P.<x,y,z,w> = ProjectiveSpace(K, 3)
sage: X = P.subscheme([x^2 + y^2 - z*w])
sage: Y = P.subscheme([y*z - x*w, z - w])
sage: Q1 = P([b^2,1,0,0])
sage: X.intersection_multiplicity(Y, Q1)                                    #␣
↪needs sage.libs.singular
1
sage: Q2 = P([1/2*b^3 - 1/2*b, 1/2*b^3 - 1/2*b, 1, 1])
sage: X.intersection_multiplicity(Y, Q2)                                    #␣
↪needs sage.libs.singular
1
```

```
sage: P.<x,y,z,w> = ProjectiveSpace(QQ, 3)
sage: X = P.subscheme([x^2 - z^2, y^3 - w*x^2])
sage: Y = P.subscheme([w^2 - 2*x*y + z^2, y^2 - w^2])
sage: Q = P([1,1,-1,1])
sage: X.intersection_multiplicity(Y, Q)                                     #␣
↪needs sage.libs.singular
Traceback (most recent call last):
...
TypeError: the intersection of this subscheme and (=Closed subscheme of Affine␣
↪Space of dimension 3
over Rational Field defined by: z^2 + w^2 - 2*y, y^2 - w^2)
must be proper and finite
```

is_smooth(*point=None*)

Test whether the algebraic subscheme is smooth.

INPUT:

- `point` – A point or `None` (default). The point to test smoothness at.

OUTPUT:

Boolean. If no point was specified, returns whether the algebraic subscheme is smooth everywhere. Otherwise, smoothness at the specified point is tested.

EXAMPLES:

```
sage: # needs sage.libs.singular
sage: P2.<x,y,z> = ProjectiveSpace(2, QQ)
sage: cuspidal_curve = P2.subscheme([y^2*z - x^3])
sage: cuspidal_curve
Closed subscheme of Projective Space of dimension 2 over Rational Field defined␣
↪by:
  -x^3 + y^2*z
sage: cuspidal_curve.is_smooth([1,1,1])
True
sage: cuspidal_curve.is_smooth([0,0,1])
False
sage: cuspidal_curve.is_smooth()
False
sage: P2.subscheme([y^2*z - x^3 + z^3 + 1/10*x*y*z]).is_smooth()
True
```

multiplicity(*P*)

Return the multiplicity of P on this subscheme.

This is computed as the multiplicity of the corresponding point on an affine patch of this subscheme that contains P. This subscheme must be defined over a field. An error is returned if P not a point on this subscheme.

INPUT:

- P – a point on this subscheme.

OUTPUT:

An integer.

EXAMPLES:

```
sage: P.<x,y,z,w,t> = ProjectiveSpace(QQ, 4)
sage: X = P.subscheme([y^2 - x*t, w^7 - t*w*x^5 - z^7])
sage: Q1 = P([0,0,1,1,1])
sage: X.multiplicity(Q1)                                                    #␣
↪needs sage.libs.singular
1
sage: Q2 = P([1,0,0,0,0])
sage: X.multiplicity(Q2)                                                    #␣
↪needs sage.libs.singular
3
sage: Q3 = P([0,0,0,0,1])
sage: X.multiplicity(Q3)                                                    #␣
↪needs sage.libs.singular
7
```

```
sage: # needs sage.rings.real_mpfr
sage: P.<x,y,z,w> = ProjectiveSpace(CC, 3)
sage: X = P.subscheme([z^5*x^2*w - y^8])
sage: Q = P([2,0,0,1])
sage: X.multiplicity(Q)                                                     #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.libs.singular
5
```

```
sage: # needs sage.libs.singular sage.schemes
sage: P.<x,y,z,w> = ProjectiveSpace(GF(29), 3)
sage: C = Curve([y^17 - x^5*w^4*z^8, x*y - z^2], P)
sage: Q = P([3,0,0,1])
sage: C.multiplicity(Q)
8
```

neighborhood(*point*)

Return an affine algebraic subscheme isomorphic to a neighborhood of the `point`.

INPUT:

- `point` – a point of the projective subscheme.

OUTPUT:

An affine algebraic scheme (polynomial equations in affine space) `result` such that

- `embedding_morphism` is an isomorphism to a neighborhood of `point`
- `embedding_center` is mapped to `point`.

EXAMPLES:

```
sage: P.<x,y,z>= ProjectiveSpace(QQ, 2)
sage: S = P.subscheme(x + 2*y + 3*z)
sage: s = S.point([0,-3,2]); s
(0 : -3/2 : 1)
sage: patch = S.neighborhood(s); patch
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  x + 3*z
sage: patch.embedding_morphism()
Scheme morphism:
  From: Closed subscheme of Affine Space of dimension 2 over Rational Field
        defined by: x + 3*z
  To:   Closed subscheme of Projective Space of dimension 2 over Rational Field
        defined by: x + 2*y + 3*z
  Defn: Defined on coordinates by sending (x, z) to (x : -3/2 : z + 1)
sage: patch.embedding_center()
(0, 0)
sage: patch.embedding_morphism()([0,0])
(0 : -3/2 : 1)
sage: patch.embedding_morphism()(patch.embedding_center())
(0 : -3/2 : 1)
```

nth_iterate(*f*, *n*)

The nth forward image of this scheme by the map `f`.

INPUT:

- `f` – a `DynamicalSystem_projective` with `self` in `f.domain()`
- `n` – a positive integer.

OUTPUT:

- A subscheme in `f.codomain()`

EXAMPLES:

```
sage: P.<x,y,z,w> = ProjectiveSpace(QQ, 3)
sage: f = DynamicalSystem_projective([y^2, z^2, x^2, w^2])                    #␣
↪needs sage.schemes
sage: f.nth_iterate(P.subscheme([x - w, y - z]), 3)                           #␣
↪needs sage.libs.singular sage.schemes
Closed subscheme of Projective Space of dimension 3 over Rational Field
 defined by:
  y - z,
  x - w
```

```
sage: PS.<x,y,z> = ProjectiveSpace(ZZ, 2)
sage: f = DynamicalSystem_projective([x^2, y^2, z^2])                         #␣
↪needs sage.schemes
sage: X = PS.subscheme([x - y])
sage: X.nth_iterate(f, -2)                                                    #␣
↪needs sage.libs.singular sage.schemes
Traceback (most recent call last):
...
TypeError: must be a forward orbit
```

```
sage: PS.<x,y,z> = ProjectiveSpace(ZZ, 2)
sage: P2.<u,v,w> = ProjectiveSpace(QQ, 2)
sage: H = Hom(PS, P2)
sage: f = H([x^2, y^2, z^2])
sage: X = PS.subscheme([x - y])
sage: X.nth_iterate(f, 2)
Traceback (most recent call last):
...
TypeError: map must be a dynamical system for iteration
```

```
sage: PS.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: f = DynamicalSystem_projective([x^2, y^2, z^2])                         #␣
↪needs sage.schemes
sage: X = PS.subscheme([x - y])
sage: X.nth_iterate(f, 2.5)                                                   #␣
↪needs sage.schemes
Traceback (most recent call last):
...
TypeError: Attempt to coerce non-integral RealNumber to Integer
```

orbit(*f*, *N*)

Return the orbit of this scheme by `f`.

If N is an integer it returns $[self, f(self), \ldots, f^N(self)]$. If N is a list or tuple $N = [m, k]$ it returns $[f^m(self), \ldots, f^k(self)]$.

INPUT:

- `f` – a `DynamicalSystem_projective` with `self` in `f.domain()`
- `N` – a non-negative integer or list or tuple of two non-negative integers

OUTPUT:

- a list of projective subschemes

EXAMPLES:

```
sage: # needs sage.libs.singular sage.schemes
sage: P.<x,y,z,w> = ProjectiveSpace(QQ, 3)
sage: f = DynamicalSystem_projective([(x-2*y)^2, (x-2*z)^2,
....:                                 (x-2*w)^2, x^2])
sage: f.orbit(P.subscheme([x]), 5)
[Closed subscheme of Projective Space of dimension 3 over Rational Field
  defined by: x,
 Closed subscheme of Projective Space of dimension 3 over Rational Field
  defined by: w,
 Closed subscheme of Projective Space of dimension 3 over Rational Field
  defined by: z - w,
 Closed subscheme of Projective Space of dimension 3 over Rational Field
  defined by: y - z,
 Closed subscheme of Projective Space of dimension 3 over Rational Field
  defined by: x - y,
 Closed subscheme of Projective Space of dimension 3 over Rational Field
  defined by: x - w]
```

```
sage: PS.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: P1.<u,v> = ProjectiveSpace(QQ, 1)
sage: H = Hom(PS, P1)
sage: f = H([x^2, y^2])
sage: X = PS.subscheme([x - y])
sage: X.orbit(f, 2)
Traceback (most recent call last):
...
TypeError: map must be a dynamical system for iteration
```

```
sage: PS.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: f = DynamicalSystem_projective([x^2, y^2, z^2])                    #␣
↪needs sage.schemes
sage: X = PS.subscheme([x - y])
sage: X.orbit(f, [-1,2])                                                 #␣
↪needs sage.schemes
Traceback (most recent call last):
...
TypeError: orbit bounds must be non-negative
```

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

Create a point on this projective subscheme.

INPUT:

- `v` – anything that defines a point
- `check` – boolean (optional, default: `True`); whether to check the defining data for consistency

OUTPUT: A point of the subscheme.

EXAMPLES:

```
sage: P2.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: X = P2.subscheme([x - y, y - z])
sage: X.point([1,1,1])
(1 : 1 : 1)
```

```
sage: P2.<x,y> = ProjectiveSpace(QQ, 1)
sage: X = P2.subscheme([y])
sage: X.point(infinity)
(1 : 0)
```

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: X = P.subscheme(x^2 + 2*y^2)
sage: X.point(infinity)
Traceback (most recent call last):
...
TypeError: Coordinates [1, 0] do not define a point on Closed subscheme
of Projective Space of dimension 1 over Rational Field defined by:
  x^2 + 2*y^2
```

preimage(*f*, *k=1*, *check=True*)

The subscheme that maps to this scheme by the map f^k.

In particular, $f^{-k}(V(h_1, \ldots, h_t)) = V(h_1 \circ f^k, \ldots, h_t \circ f^k)$. Map must be a morphism and also must be an endomorphism for $k > 1$.

INPUT:

- `f` - a map whose codomain contains this scheme
- `k` - a positive integer
- `check` – Boolean, if `False` no input checking is done

OUTPUT:

a subscheme in the domain of `f`

EXAMPLES:

```
sage: PS.<x,y,z> = ProjectiveSpace(ZZ, 2)
sage: H = End(PS)
sage: f = H([y^2, x^2, z^2])
sage: X = PS.subscheme([x - y])
sage: X.preimage(f)                                                    #␣
↪needs sage.libs.singular
Closed subscheme of Projective Space of dimension 2 over Integer Ring
 defined by:
  -x^2 + y^2
```

```
sage: P.<x,y,z,w,t> = ProjectiveSpace(QQ, 4)
sage: H = End(P)
sage: f = H([x^2 - y^2, y^2, z^2, w^2, t^2 + w^2])
sage: f.rational_preimages(P.subscheme([x - z, t^2, w - t]))           #␣
↪needs sage.libs.singular
Closed subscheme of Projective Space of dimension 4 over Rational Field
 defined by:
```

(continues on next page)

(continued from previous page)

```
  x^2 - y^2 - z^2,
  w^4 + 2*w^2*t^2 + t^4,
  -t^2
```

```
sage: P1.<x,y> = ProjectiveSpace(QQ, 1)
sage: P3.<u,v,w,t> = ProjectiveSpace(QQ, 3)
sage: H = Hom(P1, P3)
sage: X = P3.subscheme([u - v, 2*u - w, u + t])
sage: f = H([x^2, y^2, x^2 + y^2, x*y])
sage: X.preimage(f)                                                    #␣
↪needs sage.libs.singular
Closed subscheme of Projective Space of dimension 1 over Rational Field
 defined by:
  x^2 - y^2,
  x^2 - y^2,
  x^2 + x*y
```

```
sage: P1.<x,y> = ProjectiveSpace(QQ, 1)
sage: P3.<u,v,w,t> = ProjectiveSpace(QQ, 3)
sage: H = Hom(P3, P1)
sage: X = P1.subscheme([x - y])
sage: f = H([u^2, v^2])
sage: X.preimage(f)                                                    #␣
↪needs sage.libs.singular
Traceback (most recent call last):
...
TypeError: map must be a morphism
```

```
sage: PS.<x,y,z> = ProjectiveSpace(ZZ, 2)
sage: H = End(PS)
sage: f = H([x^2, x^2, x^2])
sage: X = PS.subscheme([x - y])
sage: X.preimage(f)                                                    #␣
↪needs sage.libs.singular
Traceback (most recent call last):
...
TypeError: map must be a morphism
```

```
sage: PS.<x,y,z> = ProjectiveSpace(ZZ, 2)
sage: P1.<u,v> = ProjectiveSpace(ZZ, 1)
sage: Y = P1.subscheme([u^2 - v^2])
sage: H = End(PS)
sage: f = H([x^2, y^2, z^2])
sage: Y.preimage(f)                                                    #␣
↪needs sage.libs.singular
Traceback (most recent call last):
...
TypeError: subscheme must be in ambient space of codomain
```

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: Y = P.subscheme([x - y])
```

(continues on next page)

(continued from previous page)

```
sage: H = End(P)
sage: f = H([x^2, y^2, z^2])
sage: Y.preimage(f, k=2)                                                    #␣
↪needs sage.libs.singular
Closed subscheme of Projective Space of dimension 2 over Rational Field
 defined by:
  x^4 - y^4
```

veronese_embedding(*d*, *CS=None*, *order='lex'*)

Return the degree `d` Veronese embedding of this projective subscheme.

INPUT:

- `d` – a positive integer.
- `CS` – a projective ambient space to embed into. If the projective ambient space of this subscheme is of dimension N, the dimension of `CS` must be $\binom{N+d}{d} - 1$. This is constructed if not specified. Default: `None`.
- `order` – a monomial order to use to arrange the monomials defining the embedding. The monomials will be arranged from greatest to least with respect to this order. Default: `'lex'`.

OUTPUT:

- a scheme morphism from this subscheme to its image by the degree `d` Veronese embedding.

EXAMPLES:

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: L = P.subscheme([y - x])
sage: v = L.veronese_embedding(2); v                                        #␣
↪needs sage.libs.singular
Scheme morphism:
  From: Closed subscheme of Projective Space of dimension 2
        over Rational Field defined by: -x + y
  To:   Closed subscheme of Projective Space of dimension 5
        over Rational Field defined by:
          -x4^2 + x3*x5,
          x2 - x4,
          x1 - x3,
          x0 - x3
  Defn: Defined on coordinates by sending (x : y : z) to
        (x^2 : x*y : x*z : y^2 : y*z : z^2)
sage: v.codomain().degree()                                                 #␣
↪needs sage.libs.singular
2
sage: C = P.subscheme([y*z - x^2])
sage: C.veronese_embedding(2).codomain().degree()                           #␣
↪needs sage.libs.singular
4
```

twisted cubic:

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: Q.<u,v,s,t> = ProjectiveSpace(QQ, 3)
sage: P.subscheme([]).veronese_embedding(3, Q)                              #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.libs.singular
Scheme morphism:
  From: Closed subscheme of Projective Space of dimension 1
        over Rational Field defined by: (no polynomials)
  To:   Closed subscheme of Projective Space of dimension 3
        over Rational Field defined by:
          -s^2 + v*t,
          -v*s + u*t,
          -v^2 + u*s
  Defn: Defined on coordinates by sending (x : y) to
        (x^3 : x^2*y : x*y^2 : y^3)
```

class sage.schemes.projective.projective_subscheme.**AlgebraicScheme_subscheme_projective_field**(*A*, *polynomials*)

Bases: *AlgebraicScheme_subscheme_projective*

Algebraic subschemes of projective spaces defined over fields.

Chow_form()

Return the Chow form associated to this subscheme.

For a k-dimensional subvariety of $\mathbb{P}^N$ of degree D. The $(N-k-1)$-dimensional projective linear subspaces of $\mathbb{P}^N$ meeting X form a hypersurface in the Grassmannian $G(N-k-1,N)$. The homogeneous form of degree D defining this hypersurface in Plucker coordinates is called the Chow form of X.

The base ring needs to be a number field, finite field, or $\overline{\mathbf{Q}}$.

ALGORITHM:

For a k-dimension subscheme X consider the $k+1$ linear forms $l_i = u_{i0}x_0+\cdots+u_{in}x_n$. Let J be the ideal in the polynomial ring $K[x_i,u_{ij}]$ defined by the equations of X and the l_i. Let J' be the saturation of J with respect to the irrelevant ideal of the ambient projective space of X. The elimination ideal $I = J' \cap K[u_{ij}]$ is a principal ideal, let R be its generator. The Chow form is obtained by writing R as a polynomial in Plucker coordinates (i.e. bracket polynomials). [DS1994].

OUTPUT: a homogeneous polynomial.

EXAMPLES:

```
sage: P.<x0,x1,x2,x3> = ProjectiveSpace(GF(17), 3)
sage: X = P.subscheme([x3 + x1, x2 - x0, x2 - x3])
sage: X.Chow_form()                                                    #␣
↪needs sage.libs.singular
t0 - t1 + t2 + t3
```

```
sage: P.<x0,x1,x2,x3> = ProjectiveSpace(QQ, 3)
sage: X = P.subscheme([x3^2 - 101*x1^2 - 3*x2*x0])
sage: X.Chow_form()                                                    #␣
↪needs sage.libs.singular
t0^2 - 101*t2^2 - 3*t1*t3
```

```
sage: # needs sage.libs.singular
sage: P.<x0,x1,x2,x3> = ProjectiveSpace(QQ, 3)
sage: X = P.subscheme([x0*x2 - x1^2, x0*x3 - x1*x2, x1*x3 - x2^2])
sage: Ch = X.Chow_form(); Ch
t2^3 + 2*t2^2*t3 + t2*t3^2 - 3*t1*t2*t4 - t1*t3*t4 + t0*t4^2 + t1^2*t5
sage: Y = P.subscheme_from_Chow_form(Ch, 1); Y
Closed subscheme of Projective Space of dimension 3 over Rational Field
 defined by:
  x2^2*x3 - x1*x3^2,                    -x2^3 + x0*x3^2,
  -x2^2*x3 + x1*x3^2,                   x1*x2*x3 - x0*x3^2,
  3*x1*x2^2 - 3*x0*x2*x3,               -2*x1^2*x3 + 2*x0*x2*x3,
  -3*x1^2*x2 + 3*x0*x1*x3,              x1^3 - x0^2*x3,
  x2^3 - x1*x2*x3,                      -3*x1*x2^2 + 2*x1^2*x3 + x0*x2*x3,
  2*x0*x2^2 - 2*x0*x1*x3,               3*x1^2*x2 - 2*x0*x2^2 - x0*x1*x3,
  -x0*x1*x2 + x0^2*x3,                  -x0*x1^2 + x0^2*x2,
  -x1^3 + x0*x1*x2,                     x0*x1^2 - x0^2*x2
sage: I = Y.defining_ideal()
sage: I.saturation(I.ring().ideal(list(I.ring().gens())))[0]
Ideal (x2^2 - x1*x3, x1*x2 - x0*x3, x1^2 - x0*x2)
 of Multivariate Polynomial Ring in x0, x1, x2, x3 over Rational Field
```

global_height(*prec=None*)

Return the (projective) global height of the subscheme.

INPUT:

- `prec` – desired floating point precision (default: default `RealField` precision).

OUTPUT:

- a real number.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: NF.<a> = NumberField(x^2 - 5)
sage: P.<x,y,z> = ProjectiveSpace(NF, 2)
sage: X = P.subscheme([x^2 + y*z, 2*y*z, 3*x*y])
sage: X.global_height()                                                    #␣
↪needs sage.libs.singular
0.000000000000000
```

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: X = P.subscheme([z^2 - 101*y^2 - 3*x*z])
sage: X.global_height()  # long time                                       #␣
↪needs sage.libs.singular
4.61512051684126
```

local_height(*v*, *prec=None*)

Return the (projective) local height of the subscheme.

INPUT:

- `v` – a prime or prime ideal of the base ring.
- `prec` – desired floating point precision (default: default `RealField` precision).

OUTPUT:

- a real number.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: NF.<a> = NumberField(x^2 - 5)
sage: I = NF.ideal(3)
sage: P.<x,y,z> = ProjectiveSpace(NF, 2)
sage: X = P.subscheme([3*x*y - 5*x*z, y^2])
sage: X.local_height(I)                                                     #␣
↪needs sage.libs.singular
0.000000000000000
```

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: X = P.subscheme([z^2 - 101*y^2 - 3*x*z])
sage: X.local_height(2)                                                     #␣
↪needs sage.libs.singular
0.000000000000000
```

local_height_arch(*i*, *prec=None*)

Return the local height at the `i`-th infinite place of the subscheme.

INPUT:

- `i` – an integer.
- `prec` – desired floating point precision (default: default `RealField` precision).

OUTPUT:

- a real number.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: NF.<a> = NumberField(x^2 - 5)
sage: P.<x,y,z> = ProjectiveSpace(NF, 2)
sage: X = P.subscheme([x^2 + y*z, 3*x*y])
sage: X.local_height_arch(1)                                                #␣
↪needs sage.libs.singular
0.000000000000000000000000000000000
```

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: X = P.subscheme([z^2 - 101*y^2 - 3*x*z])
sage: X.local_height_arch(1)                                                #␣
↪needs sage.libs.singular
4.61512051684126
```

14.5 Enumeration of rational points on projective schemes

Naive algorithms for enumerating rational points over $\mathbf{Q}$ or finite fields over for general schemes.

Warning: Incorrect results and infinite loops may occur if using a wrong function. (For instance using an affine function for a projective scheme or a finite field function for a scheme defined over an infinite field.)

EXAMPLES:

Projective, over $\mathbf{Q}$:

```
sage: from sage.schemes.projective.projective_rational_point import enum_projective_
↪rational_field
sage: P.<X,Y,Z> = ProjectiveSpace(2, QQ)
sage: C = P.subscheme([X + Y - Z])
sage: enum_projective_rational_field(C, 3)
[(-2 : 3 : 1), (-1 : 1 : 0), (-1 : 2 : 1), (-1/2 : 3/2 : 1),
 (0 : 1 : 1), (1/3 : 2/3 : 1), (1/2 : 1/2 : 1), (2/3 : 1/3 : 1),
 (1 : 0 : 1), (3/2 : -1/2 : 1), (2 : -1 : 1), (3 : -2 : 1)]
```

Projective over a finite field:

```
sage: from sage.schemes.projective.projective_rational_point import enum_projective_
↪finite_field
sage: E = EllipticCurve('72').change_ring(GF(19))                                #␣
↪needs sage.schemes
sage: enum_projective_finite_field(E)                                            #␣
↪needs sage.schemes
[(0 : 1 : 0), (1 : 0 : 1), (3 : 0 : 1), (4 : 9 : 1), (4 : 10 : 1),
 (6 : 6 : 1), (6 : 13 : 1), (7 : 6 : 1), (7 : 13 : 1), (9 : 4 : 1),
 (9 : 15 : 1), (12 : 8 : 1), (12 : 11 : 1), (13 : 8 : 1), (13 : 11 : 1),
 (14 : 3 : 1), (14 : 16 : 1), (15 : 0 : 1), (16 : 9 : 1), (16 : 10 : 1),
 (17 : 7 : 1), (17 : 12 : 1), (18 : 9 : 1), (18 : 10 : 1)]
```

AUTHORS:

- David R. Kohel <kohel@maths.usyd.edu.au>: original version.
- John Cremona and Charlie Turner <charlotteturner@gmail.com> (06-2010): improvements to clarity and documentation.
- Raghukul Raman <raghukul.raman01@gmail.com> (2018): Added sieve algorithm

`sage.schemes.projective.projective_rational_point.`**`enum_projective_finite_field`**(*X*)

Enumerates projective points on scheme `X` defined over a finite field.

INPUT:

- `X` - a scheme defined over a finite field or a set of abstract rational points of such a scheme.

OUTPUT:

- a list containing the projective points of `X` over the finite field, sorted.

EXAMPLES:

```
sage: from sage.schemes.projective.projective_rational_point import enum_projective_
↪finite_field
sage: F = GF(53)
sage: P.<X,Y,Z> = ProjectiveSpace(2, F)
sage: len(enum_projective_finite_field(P(F)))
2863
sage: 53^2 + 53 + 1
2863
```

```
sage: # needs sage.rings.finite_rings
sage: F = GF(9, 'a')
sage: P.<X,Y,Z> = ProjectiveSpace(2,F)
sage: C = Curve(X^3 - Y^3 + Z^2*Y)                                              #␣
↪needs sage.schemes
sage: enum_projective_finite_field(C(F))                                        #␣
↪needs sage.schemes
[(0 : 0 : 1), (0 : 1 : 1), (0 : 2 : 1), (1 : 1 : 0), (a + 1 : 2*a : 1),
 (a + 1 : 2*a + 1 : 1), (a + 1 : 2*a + 2 : 1), (2*a + 2 : a : 1),
 (2*a + 2 : a + 1 : 1), (2*a + 2 : a + 2 : 1)]
```

```
sage: F = GF(5)
sage: P2F.<X,Y,Z> = ProjectiveSpace(2, F)
sage: enum_projective_finite_field(P2F)
[(0 : 0 : 1), (0 : 1 : 0), (0 : 1 : 1), (0 : 2 : 1), (0 : 3 : 1), (0 : 4 : 1),
 (1 : 0 : 0), (1 : 0 : 1), (1 : 1 : 0), (1 : 1 : 1), (1 : 2 : 1), (1 : 3 : 1),
 (1 : 4 : 1), (2 : 0 : 1), (2 : 1 : 0), (2 : 1 : 1), (2 : 2 : 1), (2 : 3 : 1),
 (2 : 4 : 1), (3 : 0 : 1), (3 : 1 : 0), (3 : 1 : 1), (3 : 2 : 1), (3 : 3 : 1),
 (3 : 4 : 1), (4 : 0 : 1), (4 : 1 : 0), (4 : 1 : 1), (4 : 2 : 1), (4 : 3 : 1),
 (4 : 4 : 1)]
```

ALGORITHM:

Checks all points in projective space to see if they lie on `X`.

Warning: If `X` is defined over an infinite field, this code will not finish!

AUTHORS:

- John Cremona and Charlie Turner (06-2010).

`sage.schemes.projective.projective_rational_point.`**`enum_projective_number_field`**(*X*, ***kwds*)

Enumerates projective points on scheme `X` defined over a number field.

Simply checks all of the points of absolute height of at most `B` and adds those that are on the scheme to the list.

This algorithm computes 2 lists: L containing elements x in K such that H_k(x) <= B, and a list L' containing elements x in K that, due to floating point issues, may be slightly larger then the bound. This can be controlled by lowering the tolerance.

ALGORITHM:

This is an implementation of the revised algorithm (Algorithm 4) in [DK2013]. Algorithm 5 is used for imaginary quadratic fields.

INPUT:

kwds:

- bound - a real number
- tolerance - a rational number in (0,1] used in doyle-krumm algorithm-4
- precision - the precision to use for computing the elements of bounded height of number fields.

OUTPUT:

- a list containing the projective points of X of absolute height up to B, sorted.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: from sage.schemes.projective.projective_rational_point import enum_projective_
↪number_field
sage: u = QQ['u'].0
sage: K = NumberField(u^3 - 5, 'v')
sage: P.<x,y,z> = ProjectiveSpace(K, 2)
sage: X = P.subscheme([x - y])
sage: enum_projective_number_field(X(K), bound=RR(5^(1/3)), prec=2^10)          #␣
↪needs sage.symbolic
[(0 : 0 : 1), (1 : 1 : 0), (-1 : -1 : 1), (1 : 1 : 1)]
```

```
sage: # needs sage.rings.number_field
sage: u = QQ['u'].0
sage: K = NumberField(u^2 + 3, 'v')
sage: A.<x,y> = ProjectiveSpace(K, 1)
sage: X = A.subscheme(x - y)
sage: enum_projective_number_field(X, bound=2)
[(1 : 1)]
```

sage.schemes.projective.projective_rational_point.**enum_projective_rational_field**(*X*, *B*)

Enumerates projective, rational points on scheme X of height up to bound B.

INPUT:

- X - a scheme or set of abstract rational points of a scheme.
- B - a positive integer bound.

OUTPUT:

- a list containing the projective points of X of height up to B, sorted.

EXAMPLES:

```
sage: P.<X,Y,Z> = ProjectiveSpace(2, QQ)
sage: C = P.subscheme([X + Y - Z])
sage: from sage.schemes.projective.projective_rational_point import enum_projective_
↪rational_field
sage: enum_projective_rational_field(C(QQ), 6)
[(-5 : 6 : 1), (-4 : 5 : 1), (-3 : 4 : 1), (-2 : 3 : 1),
 (-3/2 : 5/2 : 1), (-1 : 1 : 0), (-1 : 2 : 1), (-2/3 : 5/3 : 1),
 (-1/2 : 3/2 : 1), (-1/3 : 4/3 : 1), (-1/4 : 5/4 : 1),
 (-1/5 : 6/5 : 1), (0 : 1 : 1), (1/6 : 5/6 : 1), (1/5 : 4/5 : 1),
 (1/4 : 3/4 : 1), (1/3 : 2/3 : 1), (2/5 : 3/5 : 1), (1/2 : 1/2 : 1),
 (3/5 : 2/5 : 1), (2/3 : 1/3 : 1), (3/4 : 1/4 : 1), (4/5 : 1/5 : 1),
```

(continues on next page)

(continued from previous page)

```
 (5/6 : 1/6 : 1), (1 : 0 : 1), (6/5 : -1/5 : 1), (5/4 : -1/4 : 1),
 (4/3 : -1/3 : 1), (3/2 : -1/2 : 1), (5/3 : -2/3 : 1), (2 : -1 : 1),
 (5/2 : -3/2 : 1), (3 : -2 : 1), (4 : -3 : 1), (5 : -4 : 1),
 (6 : -5 : 1)]
sage: enum_projective_rational_field(C,6) == enum_projective_rational_field(C(QQ),6)
True
```

```
sage: P3.<W,X,Y,Z> = ProjectiveSpace(3, QQ)
sage: enum_projective_rational_field(P3, 1)
[(-1 : -1 : -1 : 1), (-1 : -1 : 0 : 1), (-1 : -1 : 1 : 0), (-1 : -1 : 1 : 1),
 (-1 : 0 : -1 : 1), (-1 : 0 : 0 : 1), (-1 : 0 : 1 : 0), (-1 : 0 : 1 : 1),
 (-1 : 1 : -1 : 1), (-1 : 1 : 0 : 0), (-1 : 1 : 0 : 1), (-1 : 1 : 1 : 0),
 (-1 : 1 : 1 : 1), (0 : -1 : -1 : 1), (0 : -1 : 0 : 1), (0 : -1 : 1 : 0),
 (0 : -1 : 1 : 1), (0 : 0 : -1 : 1), (0 : 0 : 0 : 1), (0 : 0 : 1 : 0),
 (0 : 0 : 1 : 1), (0 : 1 : -1 : 1), (0 : 1 : 0 : 0), (0 : 1 : 0 : 1),
 (0 : 1 : 1 : 0), (0 : 1 : 1 : 1), (1 : -1 : -1 : 1), (1 : -1 : 0 : 1),
 (1 : -1 : 1 : 0), (1 : -1 : 1 : 1), (1 : 0 : -1 : 1), (1 : 0 : 0 : 0),
 (1 : 0 : 0 : 1), (1 : 0 : 1 : 0), (1 : 0 : 1 : 1), (1 : 1 : -1 : 1),
 (1 : 1 : 0 : 0), (1 : 1 : 0 : 1), (1 : 1 : 1 : 0), (1 : 1 : 1 : 1)]
```

ALGORITHM:

We just check all possible projective points in correct dimension of projective space to see if they lie on `X`.

AUTHORS:

- John Cremona and Charlie Turner (06-2010)

`sage.schemes.projective.projective_rational_point.`**`sieve`**(*X*, *bound*)

Returns the list of all projective, rational points on scheme `X` of height up to bound.

Height of a projective point $X = (x_1, x_2, \ldots, x_n)$ is given by $H_X = \max(y_1, y_2, \ldots, y_n)$, where the values y_i are the normalized coordinates such that all y_i are integers and $\gcd(y_1, y_2, \ldots, y_n) = 1$.

ALGORITHM:

Main idea behind the algorithm is to find points modulo primes and then reconstruct them using chinese remainder theorem. We find modulo primes parallelly and then lift them and apply LLL in parallel.

For the algorithm to work correctly, sufficient primes need to be present, these are calculated using the bound given in this([Hutz2015]) paper.

INPUT:

- `X` - a scheme with ambient space defined over projective space
- bound - a positive integer bound

OUTPUT:

- a list containing the projective rational points of `X` of height up to bound, sorted

EXAMPLES:

```
sage: from sage.schemes.projective.projective_rational_point import sieve
sage: P.<x,y,z,q> = ProjectiveSpace(QQ, 3)
sage: Y = P.subscheme([x^2 - 3^2*y^2 + z*q, x + z + 4*q])
sage: sorted(sieve(Y, 12))              # long time                              #␣
↪needs sage.libs.singular
```

(continues on next page)

(continued from previous page)

```
[(-4 : -4/3 : 0 : 1), (-4 : 4/3 : 0 : 1),
 (-1 : -1/3 : 1 : 0), (-1 : 1/3 : 1 : 0)]
```

```
sage: from sage.schemes.projective.projective_rational_point import sieve
sage: E = EllipticCurve('37a')                                              #␣
↪needs sage.schemes
sage: sorted(sieve(E, 14))              # long time                         #␣
↪needs sage.libs.singular sage.schemes
[(-1 : -1 : 1), (-1 : 0 : 1), (0 : -1 : 1),
 (0 : 0 : 1), (0 : 1 : 0), (1/4 : -5/8 : 1),
 (1/4 : -3/8 : 1), (1 : -1 : 1), (1 : 0 : 1),
 (2 : -3 : 1), (2 : 2 : 1), (6 : 14 : 1)]
```

14.6 Set of homomorphisms between two projective schemes

For schemes X and Y, this module implements the set of morphisms $Hom(X,Y)$. This is done by *SchemeHomset_generic*.

As a special case, the Hom-sets can also represent the points of a scheme. Recall that the K-rational points of a scheme X over k can be identified with the set of morphisms $Spec(K) \to X$. In Sage the rational points are implemented by such scheme morphisms. This is done by *SchemeHomset_points* and its subclasses.

Note: You should not create the Hom-sets manually. Instead, use the `Hom()` method that is inherited by all schemes.

AUTHORS:

- William Stein (2006): initial version.
- Volker Braun (2011-08-11): significant improvement and refactoring.
- Ben Hutz (June 2012): added support for projective ring
- Ben Hutz (2018): add numerical point support

class sage.schemes.projective.projective_homset.**SchemeHomset_points_abelian_variety_field**(*X*, *Y*, *category=None*, *check=True*, *base=Integer Ring*)

Bases: *SchemeHomset_points_projective_field*

Set of rational points of an Abelian variety.

INPUT:

See *SchemeHomset_generic*.

base_extend(*R*)

Extend the base ring.

This is currently not implemented except for the trivial case R==ZZ.

INPUT:

- R – a ring.

EXAMPLES:

```
sage: # needs sage.schemes
sage: E = EllipticCurve('37a')
sage: Hom = E.point_homset();  Hom
Abelian group of points on Elliptic Curve defined
 by y^2 + y = x^3 - x over Rational Field
sage: Hom.base_ring()
Rational Field
sage: Hom.base_extend(QQ)
Traceback (most recent call last):
...
NotImplementedError: Abelian variety point sets are not
implemented as modules over rings other than ZZ
```

class sage.schemes.projective.projective_homset.**SchemeHomset_points_projective_field**(*X*, *Y*, *category=None*, *check=True*, *base=Integer Ring*)

Bases: *SchemeHomset_points*

Set of rational points of a projective variety over a field.

INPUT:

See *SchemeHomset_generic*.

EXAMPLES:

```
sage: from sage.schemes.projective.projective_homset import SchemeHomset_points_
↪projective_field
sage: SchemeHomset_points_projective_field(Spec(QQ), ProjectiveSpace(QQ,2))
Set of rational points of Projective Space of dimension 2 over Rational Field
```

numerical_points(*F=None*, ***kwds*)

Return some or all numerical approximations of rational points of a projective scheme.

This is for dimension 0 subschemes only and the points are determined through a groebner calculation over the base ring and then numerically approximating the roots of the resulting polynomials. If the base ring is a number field, the embedding into F must be known.

INPUT:

F - numerical ring

kwds:

- point_tolerance - positive real number (optional, default: 10^{-10}). For numerically inexact fields, two points are considered the same if their coordinates are within tolerance.
- zero_tolerance - positive real number (optional, default: 10^{-10}). For numerically inexact fields, points are on the subscheme if they satisfy the equations to within tolerance.

OUTPUT: A list of points in the ambient space.

Warning: For numerically inexact fields the list of points returned may contain repeated or be missing points due to tolerance.

EXAMPLES:

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: E = P.subscheme([y^3 - x^3 - x*z^2, x*y*z])
sage: L = E(QQ).numerical_points(F=RR); L                                    #␣
↪needs sage.libs.singular
[(0.000000000000000 : 0.000000000000000 : 1.00000000000000),
 (1.00000000000000 : 1.00000000000000 : 0.000000000000000)]
sage: L[0].codomain()                                                        #␣
↪needs sage.libs.singular
Projective Space of dimension 2 over Real Field with 53 bits of precision
```

```
sage: S.<a> = QQ[]
sage: K.<v> = NumberField(a^5 - 7, embedding=CC(7)**(1/5))                   #␣
↪needs sage.rings.number_field
sage: P.<x,y,z> = ProjectiveSpace(K, 2)                                      #␣
↪needs sage.rings.number_field
sage: X = P.subscheme([x^2 - v^2*z^2, y - v*z])                              #␣
↪needs sage.rings.number_field
sage: len(X(K).numerical_points(F=CDF))                                      #␣
↪needs sage.libs.singular sage.rings.number_field
2
```

```
sage: P.<x1, x2, x3> = ProjectiveSpace(QQ, 2)
sage: E = P.subscheme([3000*x1^50 + 9875643*x2^2*x3^48 + 12334545*x2^50, x1 +␣
↪x2])
sage: len(E(P.base_ring()).numerical_points(F=CDF, zero_tolerance=1e-6))     #␣
↪needs sage.libs.singular
49
```

points(***kwds*)

Return some or all rational points of a projective scheme.

For dimension 0 subschemes points are determined through a groebner basis calculation. For schemes or subschemes with dimension greater than 1 points are determined through enumeration up to the specified bound.

INPUT:

kwds:

- `bound` - real number (optional, default: 0). The bound for the coordinates for subschemes with dimension at least 1.
- `precision` - integer (optional, default: 53). The precision to use to compute the elements of bounded height for number fields.
- `point_tolerance` - positive real number (optional, default: 10^{-10}). For numerically inexact fields, two points are considered the same if their coordinates are within tolerance.
- `zero_tolerance` - positive real number (optional, default: 10^{-10}). For numerically inexact fields, points are on the subscheme if they satisfy the equations to within tolerance.

- `tolerance` - a rational number in (0,1] used in doyle-krumm algorithm-4 for enumeration over number fields.

OUTPUT:

- a list of rational points of a projective scheme

Warning: For numerically inexact fields such as ComplexField or RealField the list of points returned is very likely to be incomplete. It may also contain repeated points due to tolerances.

EXAMPLES:

```
sage: P.<x,y> = ProjectiveSpace(QQ, 1)
sage: P(QQ).points(bound=4)
[(-4 : 1), (-3 : 1), (-2 : 1), (-3/2 : 1), (-4/3 : 1), (-1 : 1),
 (-3/4 : 1), (-2/3 : 1), (-1/2 : 1), (-1/3 : 1), (-1/4 : 1), (0 : 1),
 (1/4 : 1), (1/3 : 1), (1/2 : 1), (2/3 : 1), (3/4 : 1), (1 : 0), (1 : 1),
 (4/3 : 1), (3/2 : 1), (2 : 1), (3 : 1), (4 : 1)]
```

```
sage: u = QQ['u'].0
sage: K.<v> = NumberField(u^2 + 3)                                    #␣
↪needs sage.rings.number_field
sage: P.<x,y,z> = ProjectiveSpace(K, 2)                               #␣
↪needs sage.rings.number_field
sage: len(P(K).points(bound=1.8))                                     #␣
↪needs sage.rings.number_field
309
```

```
sage: P1 = ProjectiveSpace(GF(2), 1)
sage: F.<a> = GF(4, 'a')                                              #␣
↪needs sage.rings.finite_rings
sage: P1(F).points()                                                  #␣
↪needs sage.libs.singular sage.rings.finite_rings
[(0 : 1), (1 : 0), (1 : 1), (a : 1), (a + 1 : 1)]
```

```
sage: P.<x,y,z> = ProjectiveSpace(QQ, 2)
sage: E = P.subscheme([(y^3-y*z^2) - (x^3-x*z^2), (y^3-y*z^2) + (x^3-x*z^2)])
sage: E(P.base_ring()).points()                                       #␣
↪needs sage.libs.singular
[(-1 : -1 : 1), (-1 : 0 : 1), (-1 : 1 : 1), (0 : -1 : 1), (0 : 0 : 1),
 (0 : 1 : 1), (1 : -1 : 1), (1 : 0 : 1), (1 : 1 : 1)]
```

```
sage: # needs sage.rings.real_mpfr
sage: P.<x,y,z> = ProjectiveSpace(CC, 2)
sage: E = P.subscheme([y^3 - x^3 - x*z^2, x*y*z])
sage: L = E(P.base_ring()).points(); sorted(L, key=str)               #␣
↪needs sage.libs.singular
verbose 0 (...: projective_homset.py, points) Warning: computations in
the numerical fields are inexact;points may be computed partially or␣
↪incorrectly.
[(-0.500000000000000 + 0.866025403784439*I : 1.00000000000000 : 0.
↪000000000000000),
```

(continues on next page)

(continued from previous page)

```
 (-0.500000000000000 - 0.866025403784439*I : 1.00000000000000 : 0.
↪000000000000000),
 (-1.00000000000000*I : 0.000000000000000 : 1.00000000000000),
 (0.000000000000000 : 0.000000000000000 : 1.00000000000000),
 (1.00000000000000 : 1.00000000000000 : 0.000000000000000),
 (1.00000000000000*I : 0.000000000000000 : 1.00000000000000)]
sage: L[0].codomain()                                                    #␣
↪needs sage.libs.singular
Projective Space of dimension 2 over Complex Field with 53 bits of precision
```

```
sage: # needs sage.rings.complex_double
sage: P.<x,y,z> = ProjectiveSpace(CDF, 2)
sage: E = P.subscheme([y^2 + x^2 + z^2, x*y*z])
sage: len(E(P.base_ring()).points())                                     #␣
↪needs sage.libs.singular
verbose 0 (...: projective_homset.py, points) Warning: computations in
the numerical fields are inexact;points may be computed partially or␣
↪incorrectly.
6
```

class sage.schemes.projective.projective_homset.**SchemeHomset_points_projective_ring**(*X*, *Y*, *category=None*, *check=True*, *base=Integer Ring*)

Bases: *SchemeHomset_points*

Set of rational points of a projective variety over a commutative ring.

INPUT:

See *SchemeHomset_generic*.

EXAMPLES:

```
sage: from sage.schemes.projective.projective_homset import SchemeHomset_points_
↪projective_ring
sage: SchemeHomset_points_projective_ring(Spec(ZZ), ProjectiveSpace(ZZ,2))
Set of rational points of Projective Space of dimension 2 over Integer Ring
```

points(*B=0*)

Return some or all rational points of a projective scheme.

INPUT:

- `B` – integer (optional, default=0). The bound for the coordinates.

EXAMPLES:

```
sage: from sage.schemes.projective.projective_homset import SchemeHomset_points_
↪projective_ring
sage: H = SchemeHomset_points_projective_ring(Spec(ZZ), ProjectiveSpace(ZZ, 2))
sage: H.points(3)
[(0 : 0 : 1), (0 : 1 : -3), (0 : 1 : -2), (0 : 1 : -1), (0 : 1 : 0), (0 : 1 :␣
```

(continues on next page)

(continued from previous page)

```
↪1),
 (0 : 1 : 2), (0 : 1 : 3), (0 : 2 : -3), (0 : 2 : -1), (0 : 2 : 1), (0 : 2 : 3),
 (0 : 3 : -2), (0 : 3 : -1), (0 : 3 : 1), (0 : 3 : 2), (1 : -3 : -3),
 (1 : -3 : -2), (1 : -3 : -1), (1 : -3 : 0), (1 : -3 : 1), (1 : -3 : 2),
 (1 : -3 : 3), (1 : -2 : -3), (1 : -2 : -2), (1 : -2 : -1), (1 : -2 : 0),
 (1 : -2 : 1), (1 : -2 : 2), (1 : -2 : 3), (1 : -1 : -3), (1 : -1 : -2),
 (1 : -1 : -1), (1 : -1 : 0), (1 : -1 : 1), (1 : -1 : 2), (1 : -1 : 3),
 (1 : 0 : -3), (1 : 0 : -2), (1 : 0 : -1), (1 : 0 : 0), (1 : 0 : 1), (1 : 0 :␣
↪2),
 (1 : 0 : 3), (1 : 1 : -3), (1 : 1 : -2), (1 : 1 : -1), (1 : 1 : 0), (1 : 1 :␣
↪1),
 (1 : 1 : 2), (1 : 1 : 3), (1 : 2 : -3), (1 : 2 : -2), (1 : 2 : -1), (1 : 2 :␣
↪0),
 (1 : 2 : 1), (1 : 2 : 2), (1 : 2 : 3), (1 : 3 : -3), (1 : 3 : -2), (1 : 3 : -
↪1),
 (1 : 3 : 0), (1 : 3 : 1), (1 : 3 : 2), (1 : 3 : 3), (2 : -3 : -3),
 (2 : -3 : -2), (2 : -3 : -1), (2 : -3 : 0), (2 : -3 : 1), (2 : -3 : 2),
 (2 : -3 : 3), (2 : -2 : -3), (2 : -2 : -1), (2 : -2 : 1), (2 : -2 : 3),
 (2 : -1 : -3), (2 : -1 : -2), (2 : -1 : -1), (2 : -1 : 0), (2 : -1 : 1),
 (2 : -1 : 2), (2 : -1 : 3), (2 : 0 : -3), (2 : 0 : -1), (2 : 0 : 1),
 (2 : 0 : 3), (2 : 1 : -3), (2 : 1 : -2), (2 : 1 : -1), (2 : 1 : 0), (2 : 1 :␣
↪1),
 (2 : 1 : 2), (2 : 1 : 3), (2 : 2 : -3), (2 : 2 : -1), (2 : 2 : 1), (2 : 2 : 3),
 (2 : 3 : -3), (2 : 3 : -2), (2 : 3 : -1), (2 : 3 : 0), (2 : 3 : 1), (2 : 3 :␣
↪2),
 (2 : 3 : 3), (3 : -3 : -2), (3 : -3 : -1), (3 : -3 : 1), (3 : -3 : 2),
 (3 : -2 : -3), (3 : -2 : -2), (3 : -2 : -1), (3 : -2 : 0), (3 : -2 : 1),
 (3 : -2 : 2), (3 : -2 : 3), (3 : -1 : -3), (3 : -1 : -2), (3 : -1 : -1),
 (3 : -1 : 0), (3 : -1 : 1), (3 : -1 : 2), (3 : -1 : 3), (3 : 0 : -2),
 (3 : 0 : -1), (3 : 0 : 1), (3 : 0 : 2), (3 : 1 : -3), (3 : 1 : -2),
 (3 : 1 : -1), (3 : 1 : 0), (3 : 1 : 1), (3 : 1 : 2), (3 : 1 : 3), (3 : 2 : -3),
 (3 : 2 : -2), (3 : 2 : -1), (3 : 2 : 0), (3 : 2 : 1), (3 : 2 : 2), (3 : 2 : 3),
 (3 : 3 : -2), (3 : 3 : -1), (3 : 3 : 1), (3 : 3 : 2)]
```

class sage.schemes.projective.projective_homset.**SchemeHomset_polynomial_projective_space**(*X*, *Y*, *category=None*, *check=True*, *base=None*)

Bases: *SchemeHomset_generic*

Set of morphisms of a projective space.

EXAMPLES:

```
sage: P.<x,y,z> = ProjectiveSpace(2, QQ)
sage: Hom(P, P)
Set of morphisms
  From: Projective Space of dimension 2 over Rational Field
  To:   Projective Space of dimension 2 over Rational Field
```

identity()

Return the identity morphism of this hom-set.

EXAMPLES:

```
sage: P.<x,y,z> = ProjectiveSpace(2, QQ)
sage: Hom(P, P)
Set of morphisms
  From: Projective Space of dimension 2 over Rational Field
  To:   Projective Space of dimension 2 over Rational Field
sage: _.identity()
Scheme endomorphism of Projective Space of dimension 2 over Rational Field
  Defn: Identity map
```

CHAPTER

FIFTEEN

PRODUCTS OF PROJECTIVE SPACES

15.1 Products of projective spaces

This class builds on the projective space class and its point and morphism classes.

Products of projective spaces of varying dimension are convenient ambient spaces for complete intersections.

Group actions on them, and the interplay with representation theory, provide many interesting examples of algebraic varieties.

EXAMPLES:

We construct products projective spaces of various dimensions over the same ring:

```
sage: P1 = ProjectiveSpace(ZZ, 1, 'x')
sage: P2 = ProjectiveSpace(ZZ, 2, 'y')
sage: ProductProjectiveSpaces([P1, P2])
Product of projective spaces P^1 x P^2 over Integer Ring
```

We can also construct the product by specifying the dimensions and the base ring:

```
sage: ProductProjectiveSpaces([1, 2, 3], QQ, 'z')
Product of projective spaces P^1 x P^2 x P^3 over Rational Field

sage: P2xP2 = ProductProjectiveSpaces([2, 2], QQ, names=['x', 'y'])
sage: P2xP2.coordinate_ring().inject_variables()
Defining x0, x1, x2, y0, y1, y2
```

sage.schemes.product_projective.space.**ProductProjectiveSpaces**(*n*, *R=None*, *names='x'*)

Return the Cartesian product of projective spaces.

The input `n` is either a list of projective space over the same base ring or the list of dimensions, `R` the base ring, and `names` the variable names.

INPUT:

- `n` – a list of integers or a list of projective spaces
- `R` – a ring
- `names` – a string or list of strings

EXAMPLES:

```
sage: P1 = ProjectiveSpace(QQ, 2, 'x')
sage: P2 = ProjectiveSpace(QQ, 3, 'y')
sage: ProductProjectiveSpaces([P1, P2])
Product of projective spaces P^2 x P^3 over Rational Field
```

```
sage: ProductProjectiveSpaces([2, 2], GF(7), 'y')
Product of projective spaces P^2 x P^2 over Finite Field of size 7
```

```
sage: P1 = ProjectiveSpace(ZZ, 2, 'x')
sage: P2 = ProjectiveSpace(QQ, 3, 'y')
sage: ProductProjectiveSpaces([P1, P2])
Traceback (most recent call last):
...
AttributeError: components must be over the same base ring
```

class sage.schemes.product_projective.space.**ProductProjectiveSpaces_field**(*N*, *R=Rational Field*, *names=None*)

Bases: *ProductProjectiveSpaces_ring*

points_of_bounded_height(***kwds*)

Returns an iterator of the points in this product of projective spaces with the absolute heights of the components of at most the given bound.

Bound check is strict for the rational field. Requires the base field of this space to be a number field. Uses the Doyle-Krumm algorithm 4 (algorithm 5 for imaginary quadratic) for computing algebraic numbers up to a given height [DK2013].

The algorithm requires floating point arithmetic, so the user is allowed to specify the precision for such calculations. Additionally, due to floating point issues, points slightly larger than the bound may be returned. This can be controlled by lowering the tolerance.

INPUT:

- `bound` – a real number
- `tolerance` – a rational number in (0,1] used in doyle-krumm algorithm-4
- `precision` – the precision to use for computing the elements of bounded height of number fields.

OUTPUT: an iterator of points in this space

EXAMPLES:

```
sage: PP = ProductProjectiveSpaces(QQ, [1, 2])
sage: sorted(list(PP.points_of_bounded_height(bound=1)))
[(-1 : 1 , -1 : -1 : 1), (-1 : 1 , -1 : 0 : 1), (-1 : 1 , -1 : 1 : 0),
 (-1 : 1 , -1 : 1 : 1), (-1 : 1 , 0 : -1 : 1), (-1 : 1 , 0 : 0 : 1),
 (-1 : 1 , 0 : 1 : 0), (-1 : 1 , 0 : 1 : 1), (-1 : 1 , 1 : -1 : 1),
 (-1 : 1 , 1 : 0 : 0), (-1 : 1 , 1 : 0 : 1), (-1 : 1 , 1 : 1 : 0),
 (-1 : 1 , 1 : 1 : 1), (0 : 1 , -1 : -1 : 1), (0 : 1 , -1 : 0 : 1),
 (0 : 1 , -1 : 1 : 0), (0 : 1 , -1 : 1 : 1), (0 : 1 , 0 : -1 : 1),
 (0 : 1 , 0 : 0 : 1), (0 : 1 , 0 : 1 : 0), (0 : 1 , 0 : 1 : 1),
 (0 : 1 , 1 : -1 : 1), (0 : 1 , 1 : 0 : 0), (0 : 1 , 1 : 0 : 1),
 (0 : 1 , 1 : 1 : 0), (0 : 1 , 1 : 1 : 1), (1 : 0 , -1 : -1 : 1),
 (1 : 0 , -1 : 0 : 1), (1 : 0 , -1 : 1 : 0), (1 : 0 , -1 : 1 : 1),
```

(continues on next page)

(continued from previous page)

```
 (1 : 0 , 0 : -1 : 1), (1 : 0 , 0 : 0 : 1), (1 : 0 , 0 : 1 : 0),
 (1 : 0 , 0 : 1 : 1), (1 : 0 , 1 : -1 : 1), (1 : 0 , 1 : 0 : 0),
 (1 : 0 , 1 : 0 : 1), (1 : 0 , 1 : 1 : 0), (1 : 0 , 1 : 1 : 1),
 (1 : 1 , -1 : -1 : 1), (1 : 1 , -1 : 0 : 1), (1 : 1 , -1 : 1 : 0),
 (1 : 1 , -1 : 1 : 1), (1 : 1 , 0 : -1 : 1), (1 : 1 , 0 : 0 : 1),
 (1 : 1 , 0 : 1 : 0), (1 : 1 , 0 : 1 : 1), (1 : 1 , 1 : -1 : 1),
 (1 : 1 , 1 : 0 : 0), (1 : 1 , 1 : 0 : 1), (1 : 1 , 1 : 1 : 0),
 (1 : 1 , 1 : 1 : 1)]
```

```
sage: u = QQ['u'].0
sage: P = ProductProjectiveSpaces([1, 1], NumberField(u^2 - 2, 'v'))        #␣
↪needs sage.rings.number_field
sage: sorted(list(P.points_of_bounded_height(bound=1.5)))                   #␣
↪needs sage.rings.number_field
[(-v : 1 , -v : 1), (-v : 1 , -1 : 1), (-v : 1 , -1/2*v : 1), (-v : 1 , 0 : 1),
 (-v : 1 , 1/2*v : 1), (-v : 1 , 1 : 0), (-v : 1 , 1 : 1), (-v : 1 , v : 1),
 (-1 : 1 , -v : 1), (-1 : 1 , -1 : 1), (-1 : 1 , -1/2*v : 1), (-1 : 1 , 0 : 1),
 (-1 : 1 , 1/2*v : 1), (-1 : 1 , 1 : 0), (-1 : 1 , 1 : 1), (-1 : 1 , v : 1),
 (-1/2*v : 1 , -v : 1), (-1/2*v : 1 , -1 : 1), (-1/2*v : 1 , -1/2*v : 1),
 (-1/2*v : 1 , 0 : 1), (-1/2*v : 1 , 1/2*v : 1), (-1/2*v : 1 , 1 : 0),
 (-1/2*v : 1 , 1 : 1), (-1/2*v : 1 , v : 1), (0 : 1 , -v : 1), (0 : 1 , -1 : 1),
 (0 : 1 , -1/2*v : 1), (0 : 1 , 0 : 1), (0 : 1 , 1/2*v : 1), (0 : 1 , 1 : 0),
 (0 : 1 , 1 : 1), (0 : 1 , v : 1), (1/2*v : 1 , -v : 1), (1/2*v : 1 , -1 : 1),
 (1/2*v : 1 , -1/2*v : 1), (1/2*v : 1 , 0 : 1), (1/2*v : 1 , 1/2*v : 1),
 (1/2*v : 1 , 1 : 0), (1/2*v : 1 , 1 : 1), (1/2*v : 1 , v : 1), (1 : 0 , -v :␣
↪1),
 (1 : 0 , -1 : 1), (1 : 0 , -1/2*v : 1), (1 : 0 , 0 : 1), (1 : 0 , 1/2*v : 1),
 (1 : 0 , 1 : 0), (1 : 0 , 1 : 1), (1 : 0 , v : 1), (1 : 1 , -v : 1),
 (1 : 1 , -1 : 1), (1 : 1 , -1/2*v : 1), (1 : 1 , 0 : 1), (1 : 1 , 1/2*v : 1),
 (1 : 1 , 1 : 0), (1 : 1 , 1 : 1), (1 : 1 , v : 1), (v : 1 , -v : 1),
 (v : 1 , -1 : 1), (v : 1 , -1/2*v : 1), (v : 1 , 0 : 1), (v : 1 , 1/2*v : 1),
 (v : 1 , 1 : 0), (v : 1 , 1 : 1), (v : 1 , v : 1)]
```

class sage.schemes.product_projective.space.**ProductProjectiveSpaces_finite_field**(*N*, *R=RationalField*, *names=None*)

Bases: *ProductProjectiveSpaces_field*

rational_points(*F=None*)

Return the list of F-rational points on this product of projective spaces, where F is a given finite field, or the base ring of this space.

EXAMPLES:

```
sage: P = ProductProjectiveSpaces([1, 1], GF(5))
sage: P.rational_points()
[(0 : 1 , 0 : 1), (1 : 1 , 0 : 1), (2 : 1 , 0 : 1), (3 : 1 , 0 : 1), (4 : 1 , 0␣
↪: 1), (1 : 0 , 0 : 1),
 (0 : 1 , 1 : 1), (1 : 1 , 1 : 1), (2 : 1 , 1 : 1), (3 : 1 , 1 : 1), (4 : 1 , 1␣
↪: 1), (1 : 0 , 1 : 1),
 (0 : 1 , 2 : 1), (1 : 1 , 2 : 1), (2 : 1 , 2 : 1), (3 : 1 , 2 : 1), (4 : 1 , 2␣
```

(continues on next page)

(continued from previous page)

```
↪: 1), (1 : 0 , 2 : 1),
 (0 : 1 , 3 : 1), (1 : 1 , 3 : 1), (2 : 1 , 3 : 1), (3 : 1 , 3 : 1), (4 : 1 , 3␣
↪: 1), (1 : 0 , 3 : 1),
 (0 : 1 , 4 : 1), (1 : 1 , 4 : 1), (2 : 1 , 4 : 1), (3 : 1 , 4 : 1), (4 : 1 , 4␣
↪: 1), (1 : 0 , 4 : 1),
 (0 : 1 , 1 : 0), (1 : 1 , 1 : 0), (2 : 1 , 1 : 0), (3 : 1 , 1 : 0), (4 : 1 , 1␣
↪: 0), (1 : 0 , 1 : 0)]
```

```
sage: P = ProductProjectiveSpaces([1, 1], GF(2))
sage: sorted(P.rational_points(GF(2^2, 'a')), key=str)                    #␣
↪needs sage.rings.finite_rings
[(0 : 1 , 0 : 1), (0 : 1 , 1 : 0), (0 : 1 , 1 : 1), (0 : 1 , a + 1 : 1), (0 : 1␣
↪, a : 1),
 (1 : 0 , 0 : 1), (1 : 0 , 1 : 0), (1 : 0 , 1 : 1), (1 : 0 , a + 1 : 1), (1 : 0␣
↪, a : 1),
 (1 : 1 , 0 : 1), (1 : 1 , 1 : 0), (1 : 1 , 1 : 1), (1 : 1 , a + 1 : 1), (1 : 1␣
↪, a : 1),
 (a + 1 : 1 , 0 : 1), (a + 1 : 1 , 1 : 0), (a + 1 : 1 , 1 : 1), (a + 1 : 1 , a␣
↪+ 1 : 1), (a + 1 : 1 , a : 1),
 (a : 1 , 0 : 1), (a : 1 , 1 : 0), (a : 1 , 1 : 1), (a : 1 , a + 1 : 1), (a : 1␣
↪, a : 1)]
```

class sage.schemes.product_projective.space.**ProductProjectiveSpaces_ring**(*N*, *R=Rational Field*, *names=None*)

Bases: *AmbientSpace*

Cartesian product of projective spaces $\mathbb{P}^{n_1} \times \cdots \times \mathbb{P}^{n_r}$.

EXAMPLES:

```
sage: P.<x0,x1,x2,x3,x4> = ProductProjectiveSpaces([1, 2], QQ); P
Product of projective spaces P^1 x P^2 over Rational Field
sage: P.coordinate_ring()
Multivariate Polynomial Ring in x0, x1, x2, x3, x4 over Rational Field
sage: P[0]
Projective Space of dimension 1 over Rational Field
sage: P[1]
Projective Space of dimension 2 over Rational Field
sage: Q = P(6, 3, 2, 2, 2); Q
(2 : 1 , 1 : 1 : 1)
sage: Q[0]
(2 : 1)
sage: H = Hom(P,P)
sage: f = H([x0^2*x3, x2*x1^2, x2^2, 2*x3^2, x4^2])
sage: f(Q)
(4 : 1 , 1 : 2 : 1)
```

affine_patch(*I*, *return_embedding=False*)

Return the I^{th} affine patch of this projective space product where `I` is a multi-index.

INPUT:

- `I` – a list or tuple of positive integers.
- `return_embedding` – Boolean, if true the projective embedding is also returned.

OUTPUT:

- An affine space.
- An embedding into a product of projective spaces (optional).

EXAMPLES:

```
sage: PP = ProductProjectiveSpaces([2, 2, 2], ZZ, 'x')
sage: phi = PP.affine_patch([0, 1, 2], True)
sage: phi.domain()
Affine Space of dimension 6 over Integer Ring
sage: phi
Scheme morphism:
  From: Affine Space of dimension 6 over Integer Ring
  To:   Product of projective spaces P^2 x P^2 x P^2 over Integer Ring
  Defn: Defined on coordinates by sending (x0, x1, x2, x3, x4, x5) to
        (1 : x0 : x1 , x2 : 1 : x3 , x4 : x5 : 1)
```

change_ring(*R*)

Return a product of projective spaces over a ring R and otherwise the same as this projective space.

INPUT:

- R – commutative ring or morphism

OUTPUT:

- Product of projective spaces over R.

Note: There is no need to have any relation between R and the base ring of this space, if you want to have such a relation, use `self.base_extend(R)` instead.

EXAMPLES:

```
sage: T.<x,y,z,u,v,w> = ProductProjectiveSpaces([2, 2], QQ)
sage: T.change_ring(GF(17))
Product of projective spaces P^2 x P^2 over Finite Field of size 17
```

components()

Return the components of this product of projective spaces.

OUTPUT: a list of projective spaces

EXAMPLES:

```
sage: P.<x,y,z,u,v> = ProductProjectiveSpaces(QQ, [2, 1])
sage: P.components()
[Projective Space of dimension 2 over Rational Field,
Projective Space of dimension 1 over Rational Field]
```

dimension()

Return the absolute dimension of the product of projective spaces.

OUTPUT: a positive integer

EXAMPLES:

```
sage: T.<x,y,z,u,v,w> = ProductProjectiveSpaces([2, 2], GF(17))
sage: T.dimension_absolute()
4
sage: T.dimension()
4
```

dimension_absolute()

Return the absolute dimension of the product of projective spaces.

OUTPUT: a positive integer

EXAMPLES:

```
sage: T.<x,y,z,u,v,w> = ProductProjectiveSpaces([2, 2], GF(17))
sage: T.dimension_absolute()
4
sage: T.dimension()
4
```

dimension_absolute_components()

Return the absolute dimension of the product of projective spaces.

OUTPUT: a list of positive integers

EXAMPLES:

```
sage: T.<x,y,z,u,v,w> = ProductProjectiveSpaces([2, 2], GF(17))
sage: T.dimension_absolute_components()
[2, 2]
sage: T.dimension_components()
[2, 2]
```

dimension_components()

Return the absolute dimension of the product of projective spaces.

OUTPUT: a list of positive integers

EXAMPLES:

```
sage: T.<x,y,z,u,v,w> = ProductProjectiveSpaces([2, 2], GF(17))
sage: T.dimension_absolute_components()
[2, 2]
sage: T.dimension_components()
[2, 2]
```

dimension_relative()

Return the relative dimension of the product of projective spaces.

OUTPUT: a positive integer

EXAMPLES:

```
sage: T.<a,x,y,z,u,v,w> = ProductProjectiveSpaces([3, 2], QQ)
sage: T.dimension_relative()
5
```

dimension_relative_components()

Return the relative dimension of the product of projective spaces.

OUTPUT: a list of positive integers

EXAMPLES:

```
sage: T.<a,x,y,z,u,v,w> = ProductProjectiveSpaces([3, 2], QQ)
sage: T.dimension_relative_components()
[3, 2]
```

ngens()

Return the number of generators of this space.

This is the number of variables in the coordinate ring of the projective space.

OUTPUT: an integer

EXAMPLES:

```
sage: T = ProductProjectiveSpaces([1, 1, 1], GF(5), 'x')
sage: T.ngens()
6
```

num_components()

Returns the number of components of this space.

OUTPUT: an integer

EXAMPLES:

```
sage: T = ProductProjectiveSpaces([1, 1, 1], GF(5), 'x')
sage: T.num_components()
3
```

segre_embedding(*PP=None*, *var='u'*)

Return the Segre embedding of this space into the appropriate projective space.

INPUT:

- **PP – (default: None) ambient image projective space;**
  this is constructed if it is not given.
- `var` – string, variable name of the image projective space, default u (optional).

OUTPUT:

Hom – from this space to the appropriate subscheme of projective space.

Todo: Cartesian products with more than two components.

EXAMPLES:

```
sage: X.<y0,y1,y2,y3,y4,y5> = ProductProjectiveSpaces(ZZ, [2, 2])
sage: phi = X.segre_embedding(); phi                                    #␣
↪needs sage.libs.singular
Scheme morphism:
  From: Product of projective spaces P^2 x P^2 over Integer Ring
```

(continues on next page)

(continued from previous page)

```
  To:   Closed subscheme of Projective Space of dimension 8 over Integer Ring
        defined by:
          -u5*u7 + u4*u8,       -u5*u6 + u3*u8,       -u4*u6 + u3*u7,
          -u2*u7 + u1*u8,       -u2*u4 + u1*u5,       -u2*u6 + u0*u8,
          -u1*u6 + u0*u7,       -u2*u3 + u0*u5,       -u1*u3 + u0*u4
  Defn: Defined by sending (y0 : y1 : y2 , y3 : y4 : y5) to
        (y0*y3 : y0*y4 : y0*y5 : y1*y3 : y1*y4 : y1*y5 : y2*y3 : y2*y4 : y2*y5).

::

sage: T = ProductProjectiveSpaces([1, 2], CC, 'z')                          #␣
↪needs sage.rings.real_mpfr
sage: T.segre_embedding()                                                   #␣
↪needs sage.libs.singular sage.rings.real_mpfr
Scheme morphism:
  From: Product of projective spaces P^1 x P^2
        over Complex Field with 53 bits of precision
  To:   Closed subscheme of Projective Space of dimension 5
        over Complex Field with 53 bits of precision defined by:
          -u2*u4 + u1*u5,       -u2*u3 + u0*u5,       -u1*u3 + u0*u4
  Defn: Defined by sending (z0 : z1 , z2 : z3 : z4) to
        (z0*z2 : z0*z3 : z0*z4 : z1*z2 : z1*z3 : z1*z4).

::

sage: T = ProductProjectiveSpaces([1, 2, 1], QQ, 'z')
sage: T.segre_embedding()                                                   #␣
↪needs sage.libs.singular
Scheme morphism:
  From: Product of projective spaces P^1 x P^2 x P^1 over Rational Field
  To:   Closed subscheme of Projective Space of dimension 11
        over Rational Field defined by:
          -u9*u10 + u8*u11,     -u7*u10 + u6*u11,     -u7*u8 + u6*u9,
          -u5*u10 + u4*u11,     -u5*u8 + u4*u9,       -u5*u6 + u4*u7,
          -u5*u9 + u3*u11,      -u5*u8 + u3*u10,      -u5*u8 + u2*u11,
          -u4*u8 + u2*u10,      -u3*u8 + u2*u9,       -u3*u6 + u2*u7,
          -u3*u4 + u2*u5,       -u5*u7 + u1*u11,      -u5*u6 + u1*u10,
          -u3*u7 + u1*u9,       -u3*u6 + u1*u8,       -u5*u6 + u0*u11,
          -u4*u6 + u0*u10,      -u3*u6 + u0*u9,       -u2*u6 + u0*u8,
          -u1*u6 + u0*u7,       -u1*u4 + u0*u5,       -u1*u2 + u0*u3
  Defn: Defined by sending (z0 : z1 , z2 : z3 : z4 , z5 : z6) to
        (z0*z2*z5 : z0*z2*z6 : z0*z3*z5 : z0*z3*z6 : z0*z4*z5 : z0*z4*z6
         : z1*z2*z5 : z1*z2*z6 : z1*z3*z5 : z1*z3*z6 : z1*z4*z5 : z1*z4*z6).
```

subscheme(*X*)

Return the closed subscheme defined by `X`.

INPUT:

- `X` – a list or tuple of equations

OUTPUT:

`AlgebraicScheme_subscheme_projective_cartesian_product`.

EXAMPLES:

```
sage: P.<x,y,z,w> = ProductProjectiveSpaces([1, 1], GF(5))
sage: X = P.subscheme([x - y, z - w]); X
Closed subscheme of Product of projective spaces P^1 x P^1
 over Finite Field of size 5 defined by:
  x - y,
  z - w
sage: X.defining_polynomials()
[x - y, z - w]
sage: I = X.defining_ideal(); I
Ideal (x - y, z - w) of Multivariate Polynomial Ring in x, y, z, w
 over Finite Field of size 5
sage: X.dimension()                                                    #␣
↪needs sage.libs.singular
0
sage: X.base_ring()
Finite Field of size 5
sage: X.base_scheme()
Spectrum of Finite Field of size 5
sage: X.structure_morphism()
Scheme morphism:
  From: Closed subscheme of Product of projective spaces P^1 x P^1
        over Finite Field of size 5 defined by: x - y, z - w
  To:   Spectrum of Finite Field of size 5
  Defn: Structure map
```

sage.schemes.product_projective.space.**is_ProductProjectiveSpaces**(x)

Return True if `x` is a product of projective spaces.

This is an ambient space defined by $\mathbb{P}^n_R \times \cdots \times \mathbb{P}^m_R$, where R is a ring and $n, \ldots, m \geq 0$ are integers.

OUTPUT: Boolean.

EXAMPLES:

```
sage: is_ProductProjectiveSpaces(ProjectiveSpace(5, names='x'))
False
sage: is_ProductProjectiveSpaces(ProductProjectiveSpaces([1, 2, 3], ZZ, 'x'))
True
```

15.2 Set of homomorphisms

AUTHORS:

- Volker Braun and Ben Hutz (2014): initial version
- Raghukul Raman (2018): code cleanup and added support for rational field

class sage.schemes.product_projective.homset.**SchemeHomset_points_product_projective_spaces_field**(*X*, *Y*, *category=N*, *check=T*, *base=In*, *Ring*)

Bases: *SchemeHomset_points_product_projective_spaces_ring*

points(***kwds*)

Return some or all rational points of a projective scheme.

Over a finite field, all points are returned. Over an infinite field, all points satisfying the bound are returned. For a zero-dimensional subscheme, all points are returned regardless of whether the base ring is a field or not.

For number fields, this uses the Doyle-Krumm algorithm 4 (algorithm 5 for imaginary quadratic) for computing algebraic numbers up to a given height [DK2013] or uses the chinese remainder theorem and points modulo primes for larger bounds.

The algorithm requires floating point arithmetic, so the user is allowed to specify the precision for such calculations. Additionally, due to floating point issues, points slightly larger than the bound may be returned. This can be controlled by lowering the tolerance.

INPUT:

- `bound` - a real number
- `tolerance` - a rational number in (0,1] used in Doyle-Krumm algorithm 4
- `precision` - the precision to use for computing the elements of bounded height of number fields.
- `algorithm` - either `'sieve'` or `'enumerate'` algorithms can be used over $\mathbf{Q}$. If not specified, `'enumerate'` is used only for small height bounds.

OUTPUT: A list of rational points of the projective scheme.

EXAMPLES:

```
sage: P.<x,y,z,w> = ProductProjectiveSpaces([1, 1], QQ)
sage: X = P.subscheme([x - y, z^2 - 2*w^2])
sage: X(P.base_ring()).points()                                             #␣
↪needs sage.libs.singular
[]
```

```
sage: u = QQ['u'].0
sage: K = NumberField(u^2 - 2, 'v')                                         #␣
↪needs sage.rings.number_field
sage: P.<x,y,z,w> = ProductProjectiveSpaces([1, 1], K)                      #␣
↪needs sage.rings.number_field
sage: X = P.subscheme([x^2 - y^2, z^2 - 2*w^2])
sage: sorted(X(P.base_ring()).points())                                     #␣
↪needs sage.libs.singular sage.rings.number_field
[(-1 : 1 , -v : 1), (-1 : 1 , v : 1), (1 : 1 , -v : 1), (1 : 1 , v : 1)]
```

```
sage: u = QQ['u'].0
sage: K = NumberField(u^2 + 1, 'v')                                         #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.rings.number_field
sage: P.<x,y,z,w> = ProductProjectiveSpaces([1, 1], K)                    #␣
↪needs sage.rings.number_field
sage: P(K).points(bound=1)                                                #␣
↪needs sage.libs.singular sage.rings.number_field
[(-1 : 1 , -1 : 1), (-1 : 1 , -v : 1), (-1 : 1 , 0 : 1), (-1 : 1 , v : 1),
 (-1 : 1 , 1 : 0), (-1 : 1 , 1 : 1), (-v : 1 , -1 : 1), (-v : 1 , -v : 1),
 (-v : 1 , 0 : 1), (-v : 1 , v : 1), (-v : 1 , 1 : 0), (-v : 1 , 1 : 1),
 (0 : 1 , -1 : 1), (0 : 1 , -v : 1), (0 : 1 , 0 : 1), (0 : 1 , v : 1),
 (0 : 1 , 1 : 0), (0 : 1 , 1 : 1), (v : 1 , -1 : 1), (v : 1 , -v : 1),
 (v : 1 , 0 : 1), (v : 1 , v : 1), (v : 1 , 1 : 0), (v : 1 , 1 : 1),
 (1 : 0 , -1 : 1), (1 : 0 , -v : 1), (1 : 0 , 0 : 1), (1 : 0 , v : 1),
 (1 : 0 , 1 : 0), (1 : 0 , 1 : 1), (1 : 1 , -1 : 1), (1 : 1 , -v : 1),
 (1 : 1 , 0 : 1), (1 : 1 , v : 1), (1 : 1 , 1 : 0), (1 : 1 , 1 : 1)]
```

```
sage: P.<x,y,z,u,v> = ProductProjectiveSpaces([2, 1], GF(3))
sage: P(P.base_ring()).points()
[(0 : 0 : 1 , 0 : 1), (0 : 0 : 1 , 1 : 0), (0 : 0 : 1 , 1 : 1), (0 : 0 : 1 , 2␣
↪: 1),
 (0 : 1 : 0 , 0 : 1), (0 : 1 : 0 , 1 : 0), (0 : 1 : 0 , 1 : 1), (0 : 1 : 0 , 2␣
↪: 1),
 (0 : 1 : 1 , 0 : 1), (0 : 1 : 1 , 1 : 0), (0 : 1 : 1 , 1 : 1), (0 : 1 : 1 , 2␣
↪: 1),
 (0 : 2 : 1 , 0 : 1), (0 : 2 : 1 , 1 : 0), (0 : 2 : 1 , 1 : 1), (0 : 2 : 1 , 2␣
↪: 1),
 (1 : 0 : 0 , 0 : 1), (1 : 0 : 0 , 1 : 0), (1 : 0 : 0 , 1 : 1), (1 : 0 : 0 , 2␣
↪: 1),
 (1 : 0 : 1 , 0 : 1), (1 : 0 : 1 , 1 : 0), (1 : 0 : 1 , 1 : 1), (1 : 0 : 1 , 2␣
↪: 1),
 (1 : 1 : 0 , 0 : 1), (1 : 1 : 0 , 1 : 0), (1 : 1 : 0 , 1 : 1), (1 : 1 : 0 , 2␣
↪: 1),
 (1 : 1 : 1 , 0 : 1), (1 : 1 : 1 , 1 : 0), (1 : 1 : 1 , 1 : 1), (1 : 1 : 1 , 2␣
↪: 1),
 (1 : 2 : 1 , 0 : 1), (1 : 2 : 1 , 1 : 0), (1 : 2 : 1 , 1 : 1), (1 : 2 : 1 , 2␣
↪: 1),
 (2 : 0 : 1 , 0 : 1), (2 : 0 : 1 , 1 : 0), (2 : 0 : 1 , 1 : 1), (2 : 0 : 1 , 2␣
↪: 1),
 (2 : 1 : 0 , 0 : 1), (2 : 1 : 0 , 1 : 0), (2 : 1 : 0 , 1 : 1), (2 : 1 : 0 , 2␣
↪: 1),
 (2 : 1 : 1 , 0 : 1), (2 : 1 : 1 , 1 : 0), (2 : 1 : 1 , 1 : 1), (2 : 1 : 1 , 2␣
↪: 1),
 (2 : 2 : 1 , 0 : 1), (2 : 2 : 1 , 1 : 0), (2 : 2 : 1 , 1 : 1), (2 : 2 : 1 , 2␣
↪: 1)]
```

```
sage: PP.<x,y,z,u,v> = ProductProjectiveSpaces([2, 1], QQ)
sage: X = PP.subscheme([x + y, u*u - v*u])
sage: X.rational_points(bound=2)                                          #␣
↪needs sage.libs.singular
[(-2 : 2 : 1 , 0 : 1),
 (-2 : 2 : 1 , 1 : 1),
 (-1 : 1 : 0 , 0 : 1),
```

(continues on next page)

(continued from previous page)

```
 (-1 : 1 : 0 , 1 : 1),
 (-1 : 1 : 1 , 0 : 1),
 (-1 : 1 : 1 , 1 : 1),
 (-1/2 : 1/2 : 1 , 0 : 1),
 (-1/2 : 1/2 : 1 , 1 : 1),
 (0 : 0 : 1 , 0 : 1),
 (0 : 0 : 1 , 1 : 1),
 (1/2 : -1/2 : 1 , 0 : 1),
 (1/2 : -1/2 : 1 , 1 : 1),
 (1 : -1 : 1 , 0 : 1),
 (1 : -1 : 1 , 1 : 1),
 (2 : -2 : 1 , 0 : 1),
 (2 : -2 : 1 , 1 : 1)]
```

better to enumerate with low codimension:

```
sage: PP.<x,y,z,u,v,a,b,c> = ProductProjectiveSpaces([2, 1, 2], QQ)
sage: X = PP.subscheme([x*u^2*a, b*z*u*v, z*v^2*c])
sage: len(X.rational_points(bound=1, algorithm='enumerate'))                #␣
↪needs sage.libs.singular
232
```

class sage.schemes.product_projective.homset.**SchemeHomset_points_product_projective_spaces_ring**(*X*, *Y*, *category=No*, *check=Tr*, *base=Inte*, *Ring*)

Bases: *SchemeHomset_points*

Set of rational points of a product of projective spaces.

INPUT: See *SchemeHomset_generic*.

EXAMPLES:

```
sage: from sage.schemes.product_projective.homset import SchemeHomset_points_
↪product_projective_spaces_ring
sage: SchemeHomset_points_product_projective_spaces_ring(
....:     Spec(QQ), ProductProjectiveSpaces([1, 1], QQ, 'z'))
Set of rational points of Product of projective spaces P^1 x P^1 over Rational Field
```

15.3 Polynomial morphisms for products of projective spaces

This class builds on the projective space class and its point and morphism classes.

EXAMPLES:

```
sage: P1xP1.<x,y,u,v> = ProductProjectiveSpaces(QQ, [1, 1])
sage: H = End(P1xP1)
sage: H([x^2*u, y^2*v, x*v^2, y*u^2])
Scheme endomorphism of Product of projective spaces P^1 x P^1 over Rational Field
  Defn: Defined by sending (x : y , u : v) to (x^2*u : y^2*v , x*v^2 : y*u^2).
```

class sage.schemes.product_projective.morphism.**ProductProjectiveSpaces_morphism_ring**(*parent*, *polys*, *check=True*)

Bases: *SchemeMorphism_polynomial*

The class of morphisms on products of projective spaces.

The components are projective space morphisms.

EXAMPLES:

```
sage: T.<x,y,z,w,u> = ProductProjectiveSpaces([2, 1], QQ)
sage: H = T.Hom(T)
sage: H([x^2, y^2, z^2, w^2, u^2])
Scheme endomorphism of Product of projective spaces P^2 x P^1 over Rational Field
  Defn: Defined by sending (x : y : z , w : u) to (x^2 : y^2 : z^2 , w^2 : u^2).
```

as_dynamical_system()

Return this endomorphism as a DynamicalSystem_product_projective.

OUTPUT:

- DynamicalSystem_product_projective

EXAMPLES:

```
sage: Z.<a,b,x,y,z> = ProductProjectiveSpaces([1, 2], ZZ)
sage: H = End(Z)
sage: f = H([a^3, b^3, x^2, y^2, z^2])
sage: type(f.as_dynamical_system())                                    #␣
↪needs sage.schemes
<class 'sage.dynamics.arithmetic_dynamics.product_projective_ds.DynamicalSystem_
↪product_projective'>
```

global_height(*prec=None*)

Return the maximum of the absolute logarithmic heights of the coefficients in any of the coordinate functions of this map.

INPUT:

- prec – desired floating point precision (default: default RealField precision).

OUTPUT:

- a real number.

Todo: Add functionality for $\overline{\mathbf{Q}}$, implement function to convert the map defined over $\overline{\mathbf{Q}}$ to map over a number field.

EXAMPLES:

```
sage: P1xP1.<x,y,u,v> = ProductProjectiveSpaces([1, 1], ZZ)
sage: H = End(P1xP1)
sage: f = H([x^2*u, 3*y^2*v, 5*x*v^2, y*u^2])
sage: f.global_height()                                                    #␣
↪needs sage.rings.real_mpfr
1.60943791243410
```

```
sage: # needs sage.rings.number_field
sage: u = QQ['u'].0
sage: R = NumberField(u^2 - 2, 'v')
sage: PP.<x,y,a,b> = ProductProjectiveSpaces([1, 1], R)
sage: H = End(PP)
sage: O = R.maximal_order()
sage: g = H([3*O(u)*x^2, 13*x*y, 7*a*y, 5*b*x + O(u)*a*y])
sage: g.global_height()                                                    #␣
↪needs sage.rings.real_mpfr
2.56494935746154
```

is_morphism()

Return `True` if this mapping is a morphism of products of projective spaces.

For each component space of the codomain of this mapping we consider the subscheme of the domain of this map generated by the corresponding coordinates of the map. This map is a morphism if and only if each of these subschemes has no points.

OUTPUT: Boolean.

EXAMPLES:

```
sage: Z.<a,b,x,y,z> = ProductProjectiveSpaces([1, 2], ZZ)
sage: H = End(Z)
sage: f = H([a^2, b^2, x*z - y*z, x^2 - y^2, z^2])
sage: f.is_morphism()                                                      #␣
↪needs sage.libs.singular
False
```

```
sage: P.<x,y,z,u,v,w> = ProductProjectiveSpaces([2, 2], QQ)
sage: H = End(P)
sage: f = H([u, v, w, u^2, v^2, w^2])
sage: f.is_morphism()                                                      #␣
↪needs sage.libs.singular
True
```

```
sage: P.<x,y,z,w,u> = ProductProjectiveSpaces([2, 1], QQ)
sage: Q.<a,b,c,d,e> = ProductProjectiveSpaces([1, 2], QQ)
sage: H = Hom(P, Q)
sage: f = H([x^2, y^2, u^3, w^3, u^3])
```

(continues on next page)

(continued from previous page)

```
sage: f.is_morphism()                                                       #␣
↪needs sage.libs.singular
False
```

local_height(*v*, *prec=None*)

Return the maximum of the local height of the coefficients in any of the coordinate functions of this map.

INPUT:

- `v` – a prime or prime ideal of the base ring.
- `prec` – desired floating point precision (default: default RealField precision).

OUTPUT: A real number.

EXAMPLES:

```
sage: T.<x,y,z,w,u> = ProductProjectiveSpaces([2, 1], QQ)
sage: H = T.Hom(T)
sage: f = H([4*x^2 + 3/100*y^2, 8/210*x*y, 1/10000*z^2, 20*w^2, 1/384*u*w])
sage: f.local_height(2)                                                     #␣
↪needs sage.rings.real_mpfr
4.85203026391962
```

```
sage: # needs sage.rings.number_field
sage: R.<z> = PolynomialRing(QQ)
sage: K.<w> = NumberField(z^2 - 5)
sage: P.<x,y,a,b> = ProductProjectiveSpaces([1, 1], K)
sage: H = Hom(P, P)
sage: f = H([2*x^2 + w/3*y^2, 1/w*y^2, a^2, 6*b^2 + 1/9*a*b])
sage: f.local_height(K.ideal(3))                                            #␣
↪needs sage.rings.real_mpfr
2.19722457733622
```

15.4 Points for products of projective spaces

This class builds on the projective space class and its point and morphism classes.

EXAMPLES:

We construct products projective spaces of various dimensions over the same ring.:

```
sage: P1xP1.<x,y, u,v> = ProductProjectiveSpaces(QQ, [1, 1])
sage: P1xP1([2, 1, 3, 1])
(2 : 1 , 3 : 1)
```

class sage.schemes.product_projective.point.**ProductProjectiveSpaces_point_field**(*parent*, *polys*, *check=True*)

Bases: *ProductProjectiveSpaces_point_ring*

intersection_multiplicity(*X*)

Return the intersection multiplicity of the codomain of this point and subscheme `X` at this point.

This uses the subscheme implementation of intersection_multiplicity. This point must be a point on a subscheme of a product of projective spaces.

INPUT:

- `X` – a subscheme in the same ambient space as the codomain of this point.

OUTPUT: An integer.

EXAMPLES:

```
sage: PP.<x,y,z,u,v> = ProductProjectiveSpaces(QQ, [2, 1])
sage: X = PP.subscheme([y^2*z^3*u - x^5*v])
sage: Y = PP.subscheme([u^3 - v^3, x - y])
sage: Q = X([0,0,1,1,1])
sage: Q.intersection_multiplicity(Y)                                    #␣
↪needs sage.libs.singular
2
```

multiplicity()

Return the multiplicity of this point on its codomain.

This uses the subscheme implementation of multiplicity. This point must be a point on a subscheme of a product of projective spaces.

OUTPUT: an integer.

EXAMPLES:

```
sage: PP.<x,y,z,w,u,v,t> = ProductProjectiveSpaces(QQ, [3, 2])
sage: X = PP.subscheme([x^8*t - y^8*t + z^5*w^3*v])
sage: Q1 = X([1,1,0,0,-1,-1,1])
sage: Q1.multiplicity()                                                 #␣
↪needs sage.libs.singular
1
sage: Q2 = X([0,0,0,1,0,1,1])
sage: Q2.multiplicity()                                                 #␣
↪needs sage.libs.singular
5
sage: Q3 = X([0,0,0,1,1,0,0])
sage: Q3.multiplicity()                                                 #␣
↪needs sage.libs.singular
6
```

class sage.schemes.product_projective.point.**ProductProjectiveSpaces_point_finite_field**(*parent*, *polys*, *check=True*)

Bases: *ProductProjectiveSpaces_point_field*

class sage.schemes.product_projective.point.**ProductProjectiveSpaces_point_ring**(*parent*, *polys*, *check=True*)

Bases: *SchemeMorphism_point*

The class of points on products of projective spaces.

The components are projective space points.

EXAMPLES:

```
sage: T.<x,y,z,w,u> = ProductProjectiveSpaces([2, 1], QQ)
sage: T.point([1, 2, 3, 4, 5])
(1/3 : 2/3 : 1 , 4/5 : 1)
```

change_ring(*R*, ***kwds*)

Return a new ProductProjectiveSpaces_point which is this point coerced to R.

If the keyword check is True, then the initialization checks are performed. The user may specify the embedding into R with a keyword.

INPUT:

- R – ring.

kwds:

- check – Boolean.
- embedding – field embedding from the base ring of this point to R.

OUTPUT: ProductProjectiveSpaces_point.

EXAMPLES:

```
sage: T.<x,y,z,u,v,w> = ProductProjectiveSpaces([1, 1, 1], ZZ)
sage: P = T.point([5, 3, 15, 4, 2, 6])
sage: P.change_ring(GF(3))
(1 : 0 , 0 : 1 , 1 : 0)
```

dehomogenize(*L*)

Dehomogenize k^{th} point at $L[k]^{th}$ coordinate.

This function computes the appropriate affine patch using L and then returns the dehomogenized point on of this affine space.

INPUT:

- L - a list of non-negative integers

OUTPUT:

- SchemeMorphism_point_affine.

EXAMPLES:

```
sage: PP = ProductProjectiveSpaces([2, 2, 2], QQ, 'x')
sage: A = PP([2, 4, 6, 23, 46, 23, 9, 3, 1])
sage: A.dehomogenize([0, 1, 2])
(2, 3, 1/2, 1/2, 9, 3)
```

```
sage: # needs sage.rings.real_mpfr sage.symbolic
sage: PP.<a,b,x,y,z> = ProductProjectiveSpaces([1, 2], CC)
sage: X = PP.subscheme([a^2 + b^2])
sage: P = X([2, 2*i, -3, 6*i, 3 - 6*i])
sage: P.dehomogenize([1,0])
(-1.00000000000000*I, -2.00000000000000*I, -1.00000000000000 + 2.
↪00000000000000*I)
```

```
sage: PP = ProductProjectiveSpaces([1, 1], ZZ)
sage: A = PP([0,1,2,4])
sage: A.dehomogenize([0,0])
Traceback (most recent call last):
...
ValueError: can...t dehomogenize at 0 coordinate
```

global_height(*prec=None*)

Return the absolute logarithmic height of the point.

This function computes the maximum of global height of each component point in the product. Global height of component point is computed using function for projective point.

INPUT:

- `prec` – desired floating point precision (default: default RealField precision).

OUTPUT: A real number.

EXAMPLES:

```
sage: PP = ProductProjectiveSpaces(QQ, [2, 2], 'x')
sage: Q = PP([1, 7, 5, 18, 2, 3])
sage: Q.global_height()                                                    #␣
↪needs sage.symbolic
2.89037175789616
```

```
sage: PP = ProductProjectiveSpaces(ZZ, [1, 1], 'x')
sage: A = PP([-30, 2, 1, 6])
sage: A.global_height()                                                    #␣
↪needs sage.symbolic
2.70805020110221
```

```
sage: # needs sage.rings.number_field
sage: R.<x> = PolynomialRing(QQ)
sage: k.<w> = NumberField(x^2 + 5)
sage: PP = ProductProjectiveSpaces(k, [1, 2], 'y')
sage: Q = PP([3, 5*w + 1, 1, 7*w, 10])
sage: Q.global_height()
2.75062910527236
```

```
sage: PP = ProductProjectiveSpaces(QQbar, [1, 1], 'x')                     #␣
↪needs sage.rings.number_field
sage: Q = PP([1, QQbar(sqrt(2)), QQbar(5^(1/3)), QQbar(3^(1/3))])         #␣
↪needs sage.rings.number_field sage.symbolic
sage: Q.global_height()                                                    #␣
↪needs sage.rings.number_field sage.symbolic
0.536479304144700
```

local_height(*v*, *prec=None*)

Return the maximum of the local height of the coordinates of this point.

This function computes the maximum of local height of each component point in the product. Local height of component point is computed using function for projective point.

INPUT:

- v – a prime or prime ideal of the base ring.
- prec – desired floating point precision (default: default RealField precision).

OUTPUT: A real number.

EXAMPLES:

```
sage: PP = ProductProjectiveSpaces(QQ, [1, 1], 'x')
sage: A = PP([11, 5, 10, 2])
sage: A.local_height(5)                                                    #␣
↪needs sage.rings.real_mpfr
1.60943791243410
```

```
sage: P = ProductProjectiveSpaces(QQ, [1, 2], 'x')
sage: Q = P([1, 4, 1/2, 2, 32])
sage: Q.local_height(2)                                                    #␣
↪needs sage.rings.real_mpfr
4.15888308335967
```

normalize_coordinates()

Remove common factors (componentwise) from the coordinates of this point (including -1).

OUTPUT: None.

EXAMPLES:

```
sage: T.<x,y,z,u,v,w> = ProductProjectiveSpaces([2, 2], ZZ)
sage: P = T.point([5, 10, 15, 4, 2, 6]);
sage: P.normalize_coordinates()
sage: P
(1 : 2 : 3 , 2 : 1 : 3)
```

scale_by(*t*)

Scale the coordinates of the point by t, done componentwise.

A TypeError occurs if the point is not in the base ring of the codomain after scaling.

INPUT:

- t – a ring element

EXAMPLES:

```
sage: T.<x, y, z, u, v, w> = ProductProjectiveSpaces([1, 1, 1], ZZ)
sage: P = T.point([5, 10, 15, 4, 2, 6]);
sage: P.scale_by([2, 1, 1])
sage: P
(10 : 20 , 15 : 4 , 2 : 6)
```

15.5 Subschemes of products of projective spaces

AUTHORS:

- Ben Hutz (2014): subschemes of Cartesian products of projective space

class sage.schemes.product_projective.subscheme.**AlgebraicScheme_subscheme_product_projective**(*A*, *polynomials*)

Bases: *AlgebraicScheme_subscheme_projective*

Construct an algebraic subscheme of a product of projective spaces.

Warning: You should not create objects of this class directly. The preferred method to construct such subschemes is to use *subscheme()* method of *Product of Projective Spaces*.

INPUT:

- `A` – ambient *Product of Projective Spaces*.
- `polynomials` – single polynomial, ideal or iterable of defining multi-homogeneous polynomials.

EXAMPLES:

```
sage: P.<x, y, u, v> = ProductProjectiveSpaces([1, 1], QQ)
sage: P.subscheme([u*x^2 - v*y*x])
Closed subscheme of Product of projective spaces P^1 x P^1 over Rational Field
 defined by:
  x^2*u - x*y*v
```

affine_patch(*I*, *return_embedding=False*)

Return the I^{th} affine patch of this projective scheme where I is a multi-index.

INPUT:

- `I` – a list or tuple of positive integers
- `return_embedding` – Boolean, if true the projective embedding is also returned

OUTPUT:

- An affine algebraic scheme
- An embedding into a product of projective space (optional)

EXAMPLES:

```
sage: PP.<x,y,z,w,u,v> = ProductProjectiveSpaces([3, 1],QQ)
sage: W = PP.subscheme([y^2*z - x^3, z^2 - w^2, u^3 - v^3])
sage: W.affine_patch([0, 1], True)
(Closed subscheme of Affine Space of dimension 4 over Rational Field defined by:
  x0^2*x1 - 1,
  x1^2 - x2^2,
  x3^3 - 1,
 Scheme morphism:
```

(continues on next page)

(continued from previous page)

```
  From: Closed subscheme of Affine Space of dimension 4
        over Rational Field defined by: x0^2*x1 - 1, x1^2 - x2^2, x3^3 - 1
  To:   Closed subscheme of Product of projective spaces P^3 x P^1
        over Rational Field defined by: -x^3 + y^2*z, z^2 - w^2, u^3 - v^3
  Defn: Defined on coordinates by sending (x0, x1, x2, x3) to
        (1 : x0 : x1 : x2 , x3 : 1))
```

dimension()

Return the dimension of the algebraic subscheme.

OUTPUT: An integer.

EXAMPLES:

```
sage: X.<x,y,z,w,u,v> = ProductProjectiveSpaces([2, 2], QQ)
sage: L = (-w - v)*x + (-w*y - u*z)
sage: Q = ((-u*w - v^2)*x^2 + ((-w^2 - u*w + (-u*v - u^2))*y + (-w^2 - u*v)*z)*x
....:      + ((-w^2 - u*w - u^2)*y^2 + (-u*w - v^2)*z*y + (-w^2 + (-v - u)*w)*z^
↪2))
sage: W = X.subscheme([L, Q])
sage: W.dimension()                                                          #␣
↪needs sage.libs.singular
2
```

```
sage: PP.<x,y,z,u,v,s,t> = ProductProjectiveSpaces([2, 1, 1], QQ)
sage: X = PP.subscheme([x^3, x^5 + y^5, z^6, x*u - v*y, s^2 - t^2])
sage: X.dimension()                                                          #␣
↪needs sage.libs.singular
-1
```

```
sage: PP = ProductProjectiveSpaces([2, 1, 3], CC, 't')                       #␣
↪needs sage.rings.real_mpfr
sage: PP.subscheme([]).dimension()                                           #␣
↪needs sage.libs.singular sage.rings.real_mpfr
6
```

```
sage: PP = ProductProjectiveSpaces([1, 3, 1], ZZ, 't')
sage: PP.subscheme([]).dimension()                                           #␣
↪needs sage.libs.singular
5
```

```
sage: PP.<x,y,u,v,s,t> = ProductProjectiveSpaces([1,1,1], CC)                #␣
↪needs sage.rings.real_mpfr
sage: X = PP.subscheme([x^2 - y^2, u - v, s^2 - t^2])                        #␣
↪needs sage.libs.singular sage.rings.real_mpfr
sage: X.dimension()                                                          #␣
↪needs sage.libs.singular sage.rings.real_mpfr
0
```

intersection_multiplicity(*X*, *P*)

Return the intersection multiplicity of this subscheme and the subscheme `X` at the point `P`.

This uses the intersection_multiplicity function for affine subschemes on affine patches of this subscheme and X that contain P.

INPUT:

- X – subscheme in the same ambient space as this subscheme.
- P – a point in the intersection of this subscheme with X.

OUTPUT: An integer.

EXAMPLES:

Multiplicity of a fixed point of the map $z^2 + \frac{1}{4}$:

```
sage: PP.<x,y,u,v> = ProductProjectiveSpaces(QQ, [1, 1])
sage: G = PP.subscheme([(x^2 + 1/4*y^2)*v - y^2*u])
sage: D = PP.subscheme([x*v - y*u])
sage: sorted(G.intersection(D).rational_points())                          #␣
↪needs sage.libs.singular
[(1/2 : 1 , 1/2 : 1), (1 : 0 , 1 : 0)]
sage: Q = PP([1/2,1,1/2,1])
sage: G.intersection_multiplicity(D, Q)                                    #␣
↪needs sage.libs.singular
2
```

```
sage: # needs sage.rings.finite_rings
sage: F.<a> = GF(4)
sage: PP.<x,y,z,u,v,w> = ProductProjectiveSpaces(F, [2, 2])
sage: X = PP.subscheme([z^5 + 3*x*y^4 + 8*y^5, u^2 - v^2])
sage: Y = PP.subscheme([x^6 + z^6, w*z - v*y])
sage: Q = PP([a,a+1,1,a,a,1])
sage: X.intersection_multiplicity(Y, Q)                                    #␣
↪needs sage.libs.singular
16
```

```
sage: PP.<x,y,z,u,v,w> = ProductProjectiveSpaces(QQ, [2, 2])
sage: X = PP.subscheme([x^2*u^3 + y*z*u*v^2, x - y])
sage: Y = PP.subscheme([u^3 - w^3, x*v - y*w, z^3*w^2 - y^3*u*v])
sage: Q = PP([0,0,1,0,1,0])
sage: X.intersection_multiplicity(Y, Q)                                    #␣
↪needs sage.libs.singular
Traceback (most recent call last):
...
TypeError: the intersection of this subscheme and (=Closed subscheme of Affine␣
↪Space of dimension 4
over Rational Field defined by: x2^3 - x3^3, -x1*x3 + x0, -x1^3*x2 + x3^2) must␣
↪be proper and finite
```

is_smooth(*point=None*)

Test whether the algebraic subscheme is smooth.

EXAMPLES:

```
sage: X.<x,y,z,w,u,v> = ProductProjectiveSpaces([2, 2],QQ)
sage: L = (-w - v)*x + (-w*y - u*z)
```

(continues on next page)

(continued from previous page)

```
sage: Q = ((-u*w - v^2)*x^2 + ((-w^2 - u*w + (-u*v - u^2))*y + (-w^2 - u*v)*z)*x
....:      + ((-w^2 - u*w - u^2)*y^2 + (-u*w - v^2)*z*y + (-w^2 + (-v - u)*w)*z^
↪2))
sage: W = X.subscheme([L, Q])
sage: W.is_smooth()
Traceback (most recent call last):
...
NotImplementedError: Not Implemented
```

multiplicity(*P*)

Return the multiplicity of P on this subscheme.

This is computed as the multiplicity of the corresponding point on an affine patch of this subscheme that contains P. This subscheme must be defined over a field. An error is returned if P not a point on this subscheme.

INPUT:

- P – a point on this subscheme.

OUTPUT: An integer.

EXAMPLES:

```
sage: PP.<x,y,z,w> = ProductProjectiveSpaces(QQ, [1, 1])
sage: X = PP.subscheme([x^4*z^3 - y^4*w^3])
sage: Q1 = PP([1,1,1,1])
sage: X.multiplicity(Q1)                                                    #␣
↪needs sage.libs.singular
1
sage: Q2 = PP([0,1,1,0])
sage: X.multiplicity(Q2)                                                    #␣
↪needs sage.libs.singular
3
```

```
sage: PP.<x,y,z,w,u> = ProductProjectiveSpaces(GF(11), [1,2])
sage: X = PP.subscheme([x^7*u - y^7*z, u^6*x^2 - w^3*z^3*x*y - w^6*y^2])
sage: Q1 = PP([1,0,10,1,0])
sage: X.multiplicity(Q1)                                                    #␣
↪needs sage.libs.singular sage.rings.finite_rings
1
sage: Q2 = PP([1,0,1,0,0])
sage: X.multiplicity(Q2)                                                    #␣
↪needs sage.libs.singular sage.rings.finite_rings
4
```

segre_embedding(*PP=None*)

Return the Segre embedding of this subscheme into the appropriate projective space.

INPUT:

- PP – (default: None) ambient image projective space; this is constructed if it is not given.

OUTPUT:

Hom from this subscheme to the appropriate subscheme of projective space

EXAMPLES:

```
sage: X.<x,y,z,w,u,v> = ProductProjectiveSpaces([2, 2], QQ)
sage: P = ProjectiveSpace(QQ, 8, 't')
sage: L = (-w - v)*x + (-w*y - u*z)
sage: Q = ((-u*w - v^2)*x^2 + ((-w^2 - u*w + (-u*v - u^2))*y + (-w^2 - u*v)*z)*x
....:      + ((-w^2 - u*w - u^2)*y^2 + (-u*w - v^2)*z*y + (-w^2 + (-v - u)*w)*z^
↪2))
sage: W = X.subscheme([L,Q])
sage: phi = W.segre_embedding(P)                                                #␣
↪needs sage.libs.singular
sage: phi.codomain().ambient_space() == P                                       #␣
↪needs sage.libs.singular
True
```

```
sage: PP.<x,y,u,v,s,t> = ProductProjectiveSpaces([1, 1, 1], CC)                 #␣
↪needs sage.rings.real_mpfr
sage: PP.subscheme([]).segre_embedding()                                        #␣
↪needs sage.libs.singular sage.rings.real_mpfr
Scheme morphism:
  From: Closed subscheme of Product of projective spaces P^1 x P^1 x P^1
        over Complex Field with 53 bits of precision defined by:
          (no polynomials)
  To:   Closed subscheme of Projective Space of dimension 7
        over Complex Field with 53 bits of precision defined by:
          -u5*u6 + u4*u7,       -u3*u6 + u2*u7,       -u3*u4 + u2*u5,
          -u3*u5 + u1*u7,       -u3*u4 + u1*u6,       -u3*u4 + u0*u7,
          -u2*u4 + u0*u6,       -u1*u4 + u0*u5,       -u1*u2 + u0*u3
  Defn: Defined by sending (x : y , u : v , s : t) to
        (x*u*s : x*u*t : x*v*s : x*v*t : y*u*s : y*u*t : y*v*s : y*v*t).
```

```
sage: PP.<x,y,z,u,v,s,t> = ProductProjectiveSpaces([2, 1, 1], ZZ)
sage: PP.subscheme([x^3, u - v, s^2 - t^2]).segre_embedding()                   #␣
↪needs sage.libs.singular
Scheme morphism:
  From: Closed subscheme of Product of projective spaces P^2 x P^1 x P^1
        over Integer Ring defined by:
          x^3,                  u - v,                s^2 - t^2
  To:   Closed subscheme of Projective Space of dimension 11
        over Integer Ring defined by:
          u10^2 - u11^2,        u9 - u11,             u8 - u10,
          -u7*u10 + u6*u11,     u6*u10 - u7*u11,      u6^2 - u7^2,
          u5 - u7,              u4 - u6,              u3^3,
          -u3*u10 + u2*u11,     u2*u10 - u3*u11,      -u3*u6 + u2*u7,
          u2*u6 - u3*u7,        u2*u3^2,              u2^2 - u3^2,
          u1 - u3,              u0 - u2
  Defn: Defined by sending (x : y : z , u : v , s : t) to
        (x*u*s : x*u*t : x*v*s : x*v*t : y*u*s : y*u*t : y*v*s : y*v*t
         : z*u*s : z*u*t : z*v*s : z*v*t).
```

15.6 Enumeration of rational points on product projective schemes

Naive algorithms for enumerating rational points over $\mathbf{Q}$, number fields or finite fields over general schemes.

Warning: Incorrect results and infinite loops may occur if using a wrong function. (For instance using an affine function for a product projective scheme or a finite field function for a scheme defined over an infinite field.)

EXAMPLES:

Product Projective, over $\mathbf{Q}$:

```
sage: PP.<x,y,z> = ProductProjectiveSpaces([1,0], QQ)
sage: from sage.schemes.product_projective.rational_point import \
        enum_product_projective_rational_field
sage: enum_product_projective_rational_field(PP, 3)
[(-3 : 1 , 1), (-2 : 1 , 1), (-3/2 : 1 , 1),
 (-1 : 1 , 1), (-2/3 : 1 , 1), (-1/2 : 1 , 1),
 (-1/3 : 1 , 1), (0 : 1 , 1), (1/3 : 1 , 1),
 (1/2 : 1 , 1), (2/3 : 1 , 1), (1 : 0 , 1),
 (1 : 1 , 1), (3/2 : 1 , 1), (2 : 1 , 1),
 (3 : 1 , 1)]
```

Product projective over finite field:

```
sage: P1.<x,y,a,b> = ProductProjectiveSpaces([1, 1], GF(7))
sage: X = P1.subscheme([2*x + 3*y])
sage: from sage.schemes.product_projective.rational_point import \
        enum_product_projective_finite_field
sage: enum_product_projective_finite_field(X)
[(2 : 1 , 0 : 1), (2 : 1 , 1 : 0), (2 : 1 , 1 : 1),
 (2 : 1 , 2 : 1), (2 : 1 , 3 : 1), (2 : 1 , 4 : 1),
 (2 : 1 , 5 : 1), (2 : 1 , 6 : 1)]
```

AUTHORS:

- Volker Braun and Ben Hutz (2014): initial version
- Raghukul Raman (2018): code cleanup and added support for rational fields

sage.schemes.product_projective.rational_point.**enum_product_projective_finite_field**(*X*)

Enumerates projective points on scheme `X` defined over a finite field.

INPUT:

- `X` - a scheme defined over a finite field or a set of abstract rational points of such a scheme.

OUTPUT:

- a list containing the projective points of `X` over the finite field, sorted.

EXAMPLES:

```
sage: PP.<x,y,z,w> = ProductProjectiveSpaces([1, 1], GF(3))
sage: from sage.schemes.product_projective.rational_point import \
        enum_product_projective_finite_field
sage: enum_product_projective_finite_field(PP)
```

(continues on next page)

(continued from previous page)

```
[(0 : 1 , 0 : 1), (0 : 1 , 1 : 0), (0 : 1 , 1 : 1),
 (0 : 1 , 2 : 1), (1 : 0 , 0 : 1), (1 : 0 , 1 : 0),
 (1 : 0 , 1 : 1), (1 : 0 , 2 : 1), (1 : 1 , 0 : 1),
 (1 : 1 , 1 : 0), (1 : 1 , 1 : 1), (1 : 1 , 2 : 1),
 (2 : 1 , 0 : 1), (2 : 1 , 1 : 0), (2 : 1 , 1 : 1),
 (2 : 1 , 2 : 1)]
```

```
sage: PP.<x0,x1,x2,x3> = ProductProjectiveSpaces([1, 1], GF(17))
sage: X = PP.subscheme([x0^2 + 2*x1^2])
sage: from sage.schemes.product_projective.rational_point import \
        enum_product_projective_finite_field
sage: len(enum_product_projective_finite_field(X))
36
```

sage.schemes.product_projective.rational_point.**enum_product_projective_number_field**(*X*, ***kwds*)

Enumerates product projective points on scheme X defined over a number field.

Simply checks all of the points of absolute height of at most B and adds those that are on the scheme to the list.

This algorithm computes 2 lists: L containing elements x in K such that H_k(x) <= B, and a list L' containing elements x in K that, due to floating point issues, may be slightly larger then the bound. This can be controlled by lowering the tolerance.

ALGORITHM:

This is an implementation of the revised algorithm (Algorithm 4) in [DK2013]. Algorithm 5 is used for imaginary quadratic fields.

INPUT:

kwds:

- bound - a real number
- tolerance - a rational number in (0,1] used in doyle-krumm algorithm-4
- precision - the precision to use for computing the elements of bounded height of number fields.

OUTPUT:

- a list containing the product projective points of X of absolute height up to B, sorted.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: u = QQ['u'].0
sage: K = NumberField(u^2 + 2, 'v')
sage: PP.<x,y,z,w> = ProductProjectiveSpaces([1, 1], K)
sage: X = PP.subscheme([x^2 + 2*y^2])
sage: from sage.schemes.product_projective.rational_point import \
        enum_product_projective_number_field
sage: enum_product_projective_number_field(X, bound=1.5)
[(-v : 1 , -1 : 1), (-v : 1 , -v : 1), (-v : 1 , -1/2*v : 1),
 (-v : 1 , 0 : 1), (-v : 1 , 1/2*v : 1), (-v : 1 , v : 1),
 (-v : 1 , 1 : 0), (-v : 1 , 1 : 1), (v : 1 , -1 : 1),
 (v : 1 , -v : 1), (v : 1 , -1/2*v : 1), (v : 1 , 0 : 1),
```

(continues on next page)

(continued from previous page)

```
 (v : 1 , 1/2*v : 1), (v : 1 , v : 1), (v : 1 , 1 : 0),
 (v : 1 , 1 : 1)]
```

sage.schemes.product_projective.rational_point.**enum_product_projective_rational_field**(*X*, *B*)

Enumerate projective, rational points on scheme X of height up to bound B.

INPUT:

- X – a scheme or set of abstract rational points of a scheme
- B – a positive integer bound

OUTPUT:

- a list containing the product projective points of X of height up to B, sorted.

EXAMPLES:

```
sage: PP.<x0,x1,x2,x3,x4> = ProductProjectiveSpaces([1, 2], QQ)
sage: from sage.schemes.product_projective.rational_point import \
        enum_product_projective_rational_field
sage: enum_product_projective_rational_field(PP, 1)
[(-1 : 1 , -1 : -1 : 1), (-1 : 1 , -1 : 0 : 1), (-1 : 1 , -1 : 1 : 0),
 (-1 : 1 , -1 : 1 : 1), (-1 : 1 , 0 : -1 : 1), (-1 : 1 , 0 : 0 : 1),
 (-1 : 1 , 0 : 1 : 0), (-1 : 1 , 0 : 1 : 1), (-1 : 1 , 1 : -1 : 1),
 (-1 : 1 , 1 : 0 : 0), (-1 : 1 , 1 : 0 : 1), (-1 : 1 , 1 : 1 : 0),
 (-1 : 1 , 1 : 1 : 1), (0 : 1 , -1 : -1 : 1), (0 : 1 , -1 : 0 : 1),
 (0 : 1 , -1 : 1 : 0), (0 : 1 , -1 : 1 : 1), (0 : 1 , 0 : -1 : 1),
 (0 : 1 , 0 : 0 : 1), (0 : 1 , 0 : 1 : 0), (0 : 1 , 0 : 1 : 1),
 (0 : 1 , 1 : -1 : 1), (0 : 1 , 1 : 0 : 0), (0 : 1 , 1 : 0 : 1),
 (0 : 1 , 1 : 1 : 0), (0 : 1 , 1 : 1 : 1), (1 : 0 , -1 : -1 : 1),
 (1 : 0 , -1 : 0 : 1), (1 : 0 , -1 : 1 : 0), (1 : 0 , -1 : 1 : 1),
 (1 : 0 , 0 : -1 : 1), (1 : 0 , 0 : 0 : 1), (1 : 0 , 0 : 1 : 0),
 (1 : 0 , 0 : 1 : 1), (1 : 0 , 1 : -1 : 1), (1 : 0 , 1 : 0 : 0),
 (1 : 0 , 1 : 0 : 1), (1 : 0 , 1 : 1 : 0), (1 : 0 , 1 : 1 : 1),
 (1 : 1 , -1 : -1 : 1), (1 : 1 , -1 : 0 : 1), (1 : 1 , -1 : 1 : 0),
 (1 : 1 , -1 : 1 : 1), (1 : 1 , 0 : -1 : 1), (1 : 1 , 0 : 0 : 1),
 (1 : 1 , 0 : 1 : 0), (1 : 1 , 0 : 1 : 1), (1 : 1 , 1 : -1 : 1),
 (1 : 1 , 1 : 0 : 0), (1 : 1 , 1 : 0 : 1), (1 : 1 , 1 : 1 : 0),
 (1 : 1 , 1 : 1 : 1)]
```

```
sage: PP.<x,y,z,u,v> = ProductProjectiveSpaces([2, 1], QQ)
sage: X = PP.subscheme([x^2 + x*y + y*z, u*u - v*u])
sage: from sage.schemes.product_projective.rational_point import \
        enum_product_projective_rational_field
sage: enum_product_projective_rational_field(X, 4)
[(-2 : 4 : 1 , 0 : 1), (-2 : 4 : 1 , 1 : 1), (-1 : 1 : 0 , 0 : 1),
 (-1 : 1 : 0 , 1 : 1), (-2/3 : -4/3 : 1 , 0 : 1), (-2/3 : -4/3 : 1 , 1 : 1),
 (-1/2 : -1/2 : 1 , 0 : 1), (-1/2 : -1/2 : 1 , 1 : 1),
 (0 : 0 : 1 , 0 : 1), (0 : 0 : 1 , 1 : 1), (0 : 1 : 0 , 0 : 1),
 (0 : 1 : 0 , 1 : 1), (1 : -1/2 : 1 , 0 : 1), (1 : -1/2 : 1 , 1 : 1)]
```

sage.schemes.product_projective.rational_point.**sieve**(*X*, *bound*)

Returns the list of all rational points on scheme X of height up to bound.

ALGORITHM:

Main idea behind the algorithm is to find points modulo primes and then reconstruct them using chinese remainder theorem. We compute the points modulo primes parallelly and then lift them via chinese remainder theorem in parallel. The LLL reduction algorithm is applied to each component of the points, and finally the result is merged and converted to a point on the subscheme.

For the algorithm to work correctly, sufficient primes need to be chosen, these are determined using the bounds dependent on the bound given in [Hutz2015].

INPUT:

- `X` - a scheme with ambient space defined over a product of projective spaces
- `bound` - a positive integer bound

OUTPUT:

- a list containing the rational points of `X` of height up to `bound`, sorted

EXAMPLES:

```
sage: from sage.schemes.product_projective.rational_point import sieve
sage: PP.<x,y,z,u,v> = ProductProjectiveSpaces([2, 1], QQ)
sage: X = PP.subscheme([x^2 + y^2 - x*z, u*u - v*u])
sage: sieve(X, 2)                                                           #␣
↪needs sage.libs.singular
[(0 : 0 : 1 , 0 : 1), (0 : 0 : 1 , 1 : 1), (1/2 : -1/2 : 1 , 0 : 1),
 (1/2 : -1/2 : 1 , 1 : 1), (1/2 : 1/2 : 1 , 0 : 1), (1/2 : 1/2 : 1 , 1 : 1),
 (1 : 0 : 1 , 0 : 1), (1 : 0 : 1 , 1 : 1)]
```

CHAPTER

SIXTEEN

TORIC VARIETIES

16.1 Toric varieties

This module provides support for (normal) toric varieties, corresponding to rational polyhedral fans. See also *fano_variety* for a more restrictive class of (weak) Fano toric varieties.

An **excellent reference on toric varieties** is the book "Toric Varieties" by David A. Cox, John B. Little, and Hal Schenck [CLS2011].

The interface to this module is provided through functions *AffineToricVariety()* and *ToricVariety()*, although you may also be interested in *normalize_names()*.

Note: We do NOT build "general toric varieties" from affine toric varieties. Instead, we are using the quotient representation of toric varieties with the homogeneous coordinate ring (a.k.a. Cox's ring or the total coordinate ring). This description works best for simplicial fans of the full dimension.

AUTHORS:

- Andrey Novoseltsev (2010-05-17): initial version.
- Volker Braun (2010-07-24): Cohomology and characteristic classes added.

EXAMPLES:

We start with constructing the affine plane as an affine toric variety. First, we need to have a corresponding cone:

```
sage: quadrant = Cone([(1,0), (0,1)])
```

If you don't care about variable names and the base field, that's all we need for now:

```
sage: A2 = AffineToricVariety(quadrant)
sage: A2
2-d affine toric variety
sage: origin = A2(0,0)
sage: origin
[0 : 0]
```

Only affine toric varieties have points whose (homogeneous) coordinates are all zero.

```
sage: parent(origin)
Set of rational points of 2-d affine toric variety
```

As you can see, by default toric varieties live over the field of rational numbers:

```
sage: A2.base_ring()
Rational Field
```

While usually toric varieties are considered over the field of complex numbers, for computational purposes it is more convenient to work with fields that have exact representation on computers. You can also always do

```
sage: C2 = AffineToricVariety(quadrant, base_field=CC)
sage: C2.base_ring()
Complex Field with 53 bits of precision
sage: C2(1, 2+i)                                                              #␣
↪needs sage.symbolic
[1.00000000000000 : 2.00000000000000 + 1.00000000000000*I]
```

or even

```
sage: F = CC["a, b"].fraction_field()
sage: F.inject_variables()
Defining a, b
sage: A2 = AffineToricVariety(quadrant, base_field=F)
sage: A2(a,b)
[a : b]
```

OK, if you need to work only with affine spaces, *AffineSpace()* may be a better way to construct them. Our next example is the product of two projective lines realized as the toric variety associated to the `face fan` of the "diamond":

```
sage: diamond = lattice_polytope.cross_polytope(2)
sage: diamond.vertices()
M( 1,  0),        M( 0,  1),
M(-1,  0),        M( 0, -1)
in 2-d lattice M
sage: fan = FaceFan(diamond)
sage: P1xP1 = ToricVariety(fan)
sage: P1xP1
2-d toric variety covered by 4 affine patches
sage: P1xP1.fan().rays()
M( 1,  0),        M( 0,  1),
M(-1,  0),        M( 0, -1)
in 2-d lattice M
sage: P1xP1.gens()
(z0, z1, z2, z3)
```

We got four coordinates - two for each of the projective lines, but their names are perhaps not very well chosen. Let's make (x, y) to be coordinates on the first line and (s, t) on the second one:

```
sage: P1xP1 = ToricVariety(fan, coordinate_names="x s y t")
sage: P1xP1.gens()
(x, s, y, t)
```

Now, if we want to define subschemes of this variety, the defining polynomials must be homogeneous in each of these pairs:

```
sage: P1xP1.inject_variables()
Defining x, s, y, t
sage: P1xP1.subscheme(x)
```

(continues on next page)

(continued from previous page)

```
Closed subscheme of 2-d toric variety covered by 4 affine patches defined by:
  x
sage: P1xP1.subscheme(x^2 + y^2)
Closed subscheme of 2-d toric variety covered by 4 affine patches defined by:
  x^2 + y^2
sage: P1xP1.subscheme(x^2 + s^2)
Traceback (most recent call last):
...
ValueError: x^2 + s^2 is not homogeneous
on 2-d toric variety covered by 4 affine patches
sage: P1xP1.subscheme([x^2*s^2 + x*y*t^2 + y^2*t^2, s^3 + t^3])
Closed subscheme of 2-d toric variety covered by 4 affine patches defined by:
  x^2*s^2 + x*y*t^2 + y^2*t^2,
  s^3 + t^3
```

While we don't build toric varieties from affine toric varieties, we still can access the "building pieces":

```
sage: patch = P1xP1.affine_patch(2)
sage: patch
2-d affine toric variety
sage: patch.fan().rays()
M(1, 0),
M(0, 1)
in 2-d lattice M
sage: patch.embedding_morphism()
Scheme morphism:
  From: 2-d affine toric variety
  To:   2-d toric variety covered by 4 affine patches
  Defn: Defined on coordinates by sending [x : s] to [x : s : 1 : 1]
```

The patch above was specifically chosen to coincide with our representation of the affine plane before, but you can get the other three patches as well. (While any cone of a fan will correspond to an affine toric variety, the main interest is usually in the generating fans as "the biggest" affine subvarieties, and these are precisely the patches that you can get from *affine_patch()*.)

All two-dimensional toric varieties are "quite nice" because any two-dimensional cone is generated by exactly two rays. From the point of view of the corresponding toric varieties, this means that they have at worst quotient singularities:

```
sage: P1xP1.is_orbifold()
True
sage: P1xP1.is_smooth()
True
sage: TV = ToricVariety(NormalFan(diamond))
sage: TV.fan().rays()
N( 1,  1),        N( 1, -1),
N(-1, -1),        N(-1,  1)
in 2-d lattice N
sage: TV.is_orbifold()
True
sage: TV.is_smooth()
False
```

In higher dimensions worse things can happen:

```
sage: TV3 = ToricVariety(NormalFan(lattice_polytope.cross_polytope(3)))
sage: TV3.fan().rays()
N( 1, -1, -1),    N( 1,  1, -1),    N( 1,  1,  1),    N( 1, -1,  1),
N(-1, -1,  1),    N(-1, -1, -1),    N(-1,  1, -1),    N(-1,  1,  1)
in 3-d lattice N
sage: TV3.is_orbifold()
False
```

Fortunately, we can perform a (partial) resolution:

```
sage: TV3_res = TV3.resolve_to_orbifold()
sage: TV3_res.is_orbifold()
True
sage: TV3_res.fan().ngenerating_cones()
12
sage: TV3.fan().ngenerating_cones()
6
```

In this example we had to double the number of affine patches. The result is still singular:

```
sage: TV3_res.is_smooth()
False
```

You can resolve it further using *resolve()* method, but (at least for now) you will have to specify which rays should be inserted into the fan. See also *CPRFanoToricVariety()*, which can construct some other "nice partial resolutions."

The intersection theory on toric varieties is very well understood, and there are explicit algorithms to compute many quantities of interest. The most important tools are the *cohomology ring* and the *Chow group*. For d-dimensional compact toric varieties with at most orbifold singularities, the rational cohomology ring $H^*(X, \mathbf{Q})$ and the rational Chow ring $A^*(X, \mathbf{Q}) = A_{d-*}(X) \otimes \mathbf{Q}$ are isomorphic except for a doubling in degree. More precisely, the Chow group has the same rank

$$A_{d-k}(X) \otimes \mathbf{Q} \simeq H^{2k}(X, \mathbf{Q})$$

and the intersection in of Chow cycles matches the cup product in cohomology.

In this case, you should work with the cohomology ring description because it is much faster. For example, here is a weighted projective space with a curve of $\mathbf{Z}_3$-orbifold singularities:

```
sage: P4_11133 = toric_varieties.P4_11133()
sage: P4_11133.is_smooth(), P4_11133.is_orbifold()
(False, True)
sage: cone = P4_11133.fan(3)[8]
sage: cone.is_smooth(), cone.is_simplicial()
(False, True)
sage: HH = P4_11133.cohomology_ring();  HH
Rational cohomology ring of a 4-d CPR-Fano toric variety covered by 5 affine patches
sage: P4_11133.cohomology_basis()                                              #␣
↪needs sage.libs.singular
(([1],), ([z4],), ([z4^2],), ([z4^3],), ([z4^4],))
```

Every cone defines a torus orbit closure, and hence a (co)homology class:

```
sage: HH.gens()
([3*z4], [3*z4], [z4], [z4], [z4])
sage: list(map(HH, P4_11133.fan(1)))
[[3*z4], [3*z4], [z4], [z4], [z4]]
sage: list(map(HH, P4_11133.fan(4)))
[[9*z4^4], [9*z4^4], [9*z4^4], [9*z4^4], [9*z4^4]]
sage: HH(cone)
[3*z4^3]
```

We can compute intersection numbers by integrating top-dimensional cohomology classes:

```
sage: D = P4_11133.divisor(0)
sage: HH(D)
[3*z4]
sage: P4_11133.integrate(HH(D)^4)
9
sage: P4_11133.integrate(HH(D) * HH(cone))
1
```

Although computationally less efficient, we can do the same computations with the rational Chow group:

```
sage: AA = P4_11133.Chow_group(QQ)
sage: list(map(AA, P4_11133.fan(1)))  # long time (5s on sage.math, 2012)
[( 0 | 0 | 0 | 3 | 0 ), ( 0 | 0 | 0 | 3 | 0 ), ( 0 | 0 | 0 | 1 | 0 ), ( 0 | 0 | 0 | 1 |␣
↪0 ), ( 0 | 0 | 0 | 1 | 0 )]
sage: list(map(AA, P4_11133.fan(4)))  # long time (5s on sage.math, 2012)
[( 1 | 0 | 0 | 0 | 0 ), ( 1 | 0 | 0 | 0 | 0 ), ( 1 | 0 | 0 | 0 | 0 ), ( 1 | 0 | 0 | 0 |␣
↪0 ), ( 1 | 0 | 0 | 0 | 0 )]
sage: AA(cone).intersection_with_divisor(D)  # long time (4s on sage.math, 2013)
( 1 | 0 | 0 | 0 | 0 )
sage: AA(cone).intersection_with_divisor(D).count_points()  # long time
1
```

The real advantage of the Chow group is that

- it works just as well over $\mathbf{Z}$, so torsion information is also easily available, and
- its combinatorial description also works over worse-than-orbifold singularities. By contrast, the cohomology groups can become very complicated to compute in this case, and one usually only has a spectral sequence but no toric algorithm.

Below you will find detailed descriptions of available functions. If you are familiar with toric geometry, you will likely see that many important objects and operations are unavailable. However, this module is under active development and hopefully will improve in future releases of Sage. If there are some particular features that you would like to see implemented ASAP, please consider reporting them to the Sage Development Team or even implementing them on your own as a patch for inclusion!

`sage.schemes.toric.variety.`**AffineToricVariety**(*cone*, **args*, ***kwds*)

Construct an affine toric variety.

INPUT:

- `cone` – `strictly convex rational polyhedral cone`.

This cone will be used to construct a `rational polyhedral fan`, which will be passed to *ToricVariety()* with the rest of positional and keyword arguments.

OUTPUT: A *toric variety*.

Note: The generating rays of the fan of this variety are guaranteed to be listed in the same order as the rays of the original cone.

EXAMPLES:

We will create the affine plane as an affine toric variety:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: A2 = AffineToricVariety(quadrant)
sage: origin = A2(0,0)
sage: origin
[0 : 0]
sage: parent(origin)
Set of rational points of 2-d affine toric variety
```

Only affine toric varieties have points whose (homogeneous) coordinates are all zero.

class sage.schemes.toric.variety.**CohomologyClass**(*cohomology_ring*, *representative*)

Bases: QuotientRingElement

An element of the *CohomologyRing*.

Warning: You should not create instances of this class manually. The generators of the cohomology ring as well as the cohomology classes associated to cones of the fan can be obtained from *ToricVariety_field.cohomology_ring()*.

EXAMPLES:

```
sage: # needs sage.libs.singular
sage: P2 = toric_varieties.P2()
sage: P2.cohomology_ring().gen(0)
[z]
sage: HH = P2.cohomology_ring()
sage: HH.gen(0)
[z]
sage: cone = P2.fan(1)[0];  HH(cone)
[z]
```

deg()

The degree of the cohomology class.

OUTPUT:

An integer d such that the cohomology class is in degree $2d$. If the cohomology class is of mixed degree, the highest degree is returned.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: P2.cohomology_ring().gen(0).deg()                                    #␣
↪needs sage.libs.singular
1
sage: P2.cohomology_ring().zero().deg()                                    #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.libs.singular
-1
```

exp()

Exponentiate `self`.

Note: The exponential $\exp(x)$ of a rational number x is usually not rational. Therefore, the cohomology class must not have a constant (degree zero) part. The coefficients in the Taylor series of $\exp$ are rational, so any cohomology class without constant term can be exponentiated.

OUTPUT:

The cohomology class $\exp($ `self` $)$ if the constant part vanishes, otherwise a `ValueError` is raised.

EXAMPLES:

```
sage: # needs sage.libs.singular
sage: P2 = toric_varieties.P2()
sage: H_class = P2.cohomology_ring().gen(0)
sage: H_class
[z]
sage: H_class.exp()
[1/2*z^2 + z + 1]
```

part_of_degree(*d*)

Project the (mixed-degree) cohomology class to the given degree.

$$pr_d : \ H^\bullet(X_\Delta, \mathbf{Q}) \to H^{2d}(X_\Delta, \mathbf{Q})$$

INPUT:

- An integer `d`

OUTPUT:

- The degree-`2d` part of the cohomology class.

EXAMPLES:

```
sage: # needs sage.libs.singular
sage: P1xP1 = toric_varieties.P1xP1()
sage: t = P1xP1.cohomology_ring().gen(0)
sage: y = P1xP1.cohomology_ring().gen(2)
sage: 3*t + 4*t^2*y + y + t*y + t + 1
[t*y + 4*t + y + 1]
sage: (3*t + 4*t^2*y + y + t*y + t + 1).part_of_degree(1)
[4*t + y]
```

class sage.schemes.toric.variety.**CohomologyRing**(*variety*)

Bases: `QuotientRing_generic`, `UniqueRepresentation`

The (even) cohomology ring of a toric variety.

Irregardles of the variety's base ring, we always work with the variety over $\mathbf{C}$ and its topology.

The cohomology is always the singular cohomology with $\mathbf{Q}$-coefficients. Note, however, that the cohomology of smooth toric varieties is torsion-free, so there is no loss of information in that case.

Currently, the toric variety must not be "too singular". See *ToricVariety_field.cohomology_ring()* for a detailed description of which toric varieties are admissible. For such varieties the odd-dimensional cohomology groups vanish.

Warning: You should not create instances of this class manually. Use *ToricVariety_field.cohomology_ring()* to generate the cohomology ring.

INPUT:

- `variety` – a toric variety. Currently, the toric variety must be at least an orbifold. See *ToricVariety_field.cohomology_ring()* for a detailed description of which toric varieties are admissible.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: P2.cohomology_ring()
Rational cohomology ring of a 2-d CPR-Fano toric variety covered by 3 affine patches
```

This is equivalent to:

```
sage: from sage.schemes.toric.variety import CohomologyRing
sage: CohomologyRing(P2)
Rational cohomology ring of a 2-d CPR-Fano toric variety covered by 3 affine patches
```

gen(*i*)

Return the generators of the cohomology ring.

INPUT:

- `i` – integer.

OUTPUT:

The `i`-th generator of the cohomology ring. If we denote the toric variety by `X`, then this generator is associated to the ray `X.fan().ray(i)`, which spans the one-cone `X.fan(1)[i]`

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: P2.cohomology_ring().gen(2)                                              #␣
↪needs sage.libs.singular
[z]
```

gens()

Return the generators of the cohomology ring.

OUTPUT:

A tuple of generators, one for each toric divisor of the toric variety `X`. The order is the same as the ordering of the rays of the fan `X.fan().rays()`, which is also the same as the ordering of the one-cones in `X.fan(1)`

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: P2.cohomology_ring().gens()                                              #␣
↪needs sage.libs.singular
([z], [z], [z])
```

sage.schemes.toric.variety.**ToricVariety**(*fan*, *coordinate_names=None*, *names=None*, *coordinate_indices=None*, *base_ring=Rational Field*, *base_field=None*)

Construct a toric variety.

INPUT:

- `fan` – `rational polyhedral fan`;
- `coordinate_names` – names of variables for the coordinate ring, see *normalize_names()* for acceptable formats. If not given, indexed variable names will be created automatically;
- `names` – an alias of `coordinate_names` for internal use. You may specify either `names` or `coordinate_names`, but not both;
- `coordinate_indices` – list of integers, indices for indexed variables. If not given, the index of each variable will coincide with the index of the corresponding ray of the fan;
- `base_ring` – base ring of the toric variety (default: $\mathbf{Q}$). Must be a field.
- `base_field` – alias for `base_ring`. Takes precedence if both are specified.

OUTPUT: A *toric variety*.

EXAMPLES:

We will create the product of two projective lines:

```
sage: fan = FaceFan(lattice_polytope.cross_polytope(2))
sage: fan.rays()
M( 1,  0),    M( 0,  1),
M(-1,  0),    M( 0, -1)
in 2-d lattice M
sage: P1xP1 = ToricVariety(fan)
sage: P1xP1.gens()
(z0, z1, z2, z3)
```

Let's create some points:

```
sage: P1xP1(1,1,1,1)
[1 : 1 : 1 : 1]
sage: P1xP1(0,1,1,1)
[0 : 1 : 1 : 1]
sage: P1xP1(0,1,0,1)
Traceback (most recent call last):
...
TypeError: coordinates (0, 1, 0, 1) are in the exceptional set
```

We cannot set to zero both coordinates of the same projective line!

Let's change the names of the variables. We have to re-create our toric variety:

```
sage: P1xP1 = ToricVariety(fan, "x s y t")
sage: P1xP1.gens()
(x, s, y, t)
```

Now (x, y) correspond to one line and (s, t) to the other one.

```
sage: P1xP1.inject_variables()
Defining x, s, y, t
sage: P1xP1.subscheme(x*s - y*t)
Closed subscheme of 2-d toric variety covered by 4 affine patches defined by:
  x*s - y*t
```

Here is a shorthand for defining the toric variety and homogeneous coordinates in one go:

```
sage: P1xP1.<a,b,c,d> = ToricVariety(fan)
sage: (a^2+b^2) * (c+d)
a^2*c + b^2*c + a^2*d + b^2*d
```

class sage.schemes.toric.variety.**ToricVariety_field**(*fan*, *coordinate_names*, *coordinate_indices*, *base_field*)

Bases: *AmbientSpace*

Construct a toric variety associated to a rational polyhedral fan.

Warning: This class does not perform any checks of correctness of input. Use *ToricVariety()* and *AffineToricVariety()* to construct toric varieties.

INPUT:

- `fan` – `rational polyhedral fan`;
- `coordinate_names` – names of variables, see *normalize_names()* for acceptable formats. If `None`, indexed variable names will be created automatically;
- `coordinate_indices` – list of integers, indices for indexed variables. If `None`, the index of each variable will coincide with the index of the corresponding ray of the fan;
- `base_field` – base field of the toric variety.

OUTPUT: A *toric variety*.

Aut_dimension()

Return the dimension of the automorphism group

There are three kinds of symmetries of toric varieties:

- Toric automorphisms (rescaling of homogeneous coordinates)
- Demazure roots. These are translations $x_i \to x_i + \epsilon x^m$ of a homogeneous coordinate x_i by a monomial x^m of the same homogeneous degree.
- Symmetries of the fan. These yield discrete subgroups.

OUTPUT:

An integer. The dimension of the automorphism group. Equals the dimension of the M-lattice plus the number of Demazure roots.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: P2.Aut_dimension()
8
```

Chern_character(*deg=None*)

Return the Chern character (of the tangent bundle) of the toric variety.

INPUT:

- deg – integer (optional). The degree of the Chern character.

OUTPUT:

- If the degree is specified, the degree-deg part of the Chern character.
- If no degree is specified, the total Chern character.

REFERENCES:

- Wikipedia article Chern_character#The_Chern_character

EXAMPLES:

```
sage: # needs sage.libs.singular
sage: dP6 = toric_varieties.dP6()
sage: dP6.Chern_character()
[3*w^2 + y + 2*v + 2*z + w + 2]
sage: dP6.ch()
[3*w^2 + y + 2*v + 2*z + w + 2]
sage: dP6.ch(1) == dP6.c(1)
True
```

Chern_class(*deg=None*)

Return Chern classes of the (tangent bundle of the) toric variety.

INPUT:

- deg – integer (optional). The degree of the Chern class.

OUTPUT:

- If the degree is specified, the deg-th Chern class.
- If no degree is specified, the total Chern class.

REFERENCES:

- Wikipedia article Chern_class

EXAMPLES:

```
sage: # needs sage.libs.singular
sage: X = toric_varieties.dP6()
sage: X.Chern_class()
[-6*w^2 + y + 2*v + 2*z + w + 1]
sage: X.c()
[-6*w^2 + y + 2*v + 2*z + w + 1]
sage: X.c(1)
[y + 2*v + 2*z + w]
sage: X.c(2)
[-6*w^2]
sage: X.integrate( X.c(2) )
6
sage: X.integrate( X.c(2) ) == X.Euler_number()
True
```

Chow_group(*base_ring=Integer Ring*)

Return the toric Chow group.

INPUT:

- `base_ring` – either `ZZ` (default) or `QQ`. The coefficient ring of the Chow group.

OUTPUT: A *sage.schemes.toric.chow_group.ChowGroup_class*.

EXAMPLES:

```
sage: A = toric_varieties.P2().Chow_group(); A
Chow group of 2-d CPR-Fano toric variety covered by 3 affine patches
sage: A.gens()
(( 0 | 0 | 1 ), ( 0 | 1 | 0 ), ( 1 | 0 | 0 ))
```

Demazure_roots()

Return the Demazure roots.

OUTPUT:

The roots as points of the M-lattice.

REFERENCES:

- [De1970]
- [Baz2011]

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: P2.Demazure_roots()
(M(-1, 0), M(-1, 1), M(0, -1), M(0, 1), M(1, -1), M(1, 0))
```

Here are the remaining three examples listed in [Baz2011], Example 2.1 and 2.3:

```
sage: s = 3
sage: cones = [(0,1), (1,2), (2,3), (3,0)]
sage: Hs = ToricVariety(Fan(rays=[(1,0), (0,-1), (-1,s), (0,1)], cones=cones))
sage: Hs.Demazure_roots()
(M(-1, 0), M(1, 0), M(0, 1), M(1, 1), M(2, 1), M(3, 1))

sage: P11s = ToricVariety(Fan(rays=[(1,0), (0,-1), (-1,s)],
....:                         cones=[(0,1), (1,2), (2,0)]))
sage: P11s.Demazure_roots()
(M(-1, 0), M(1, 0), M(0, 1), M(1, 1), M(2, 1), M(3, 1))
sage: P11s.Demazure_roots() == Hs.Demazure_roots()
True

sage: Bs = ToricVariety(Fan(rays=[(s,1), (s,-1), (-s,-1), (-s,1)], cones=cones))
sage: Bs.Demazure_roots()
()
```

Euler_number()

Return the topological Euler number of the toric variety.

Sometimes, this is also called the Euler characteristic. *chi()* is a synonym for *Euler_number()*.

REFERENCES:

- Wikipedia article Euler_characteristic

EXAMPLES:

```
sage: # needs sage.libs.singular
sage: P1xP1 = toric_varieties.P1xP1()
sage: P1xP1.Euler_number()
4
sage: P1xP1.chi()
4
```

K()

Returns the canonical divisor of the toric variety.

EXAMPLES:

Lets test that the del Pezzo surface dP_6 has degree 6, as its name implies:

```
sage: dP6 = toric_varieties.dP6()
sage: HH = dP6.cohomology_ring()
sage: dP6.K()
-V(x) - V(u) - V(y) - V(v) - V(z) - V(w)
sage: dP6.integrate( HH(dP6.K())^2 )                                    #␣
↪needs sage.libs.singular
6
```

Kaehler_cone()

Return the closure of the Kähler cone of `self`.

OUTPUT: cone.

Note: This cone sits in the rational divisor class group of `self` and the choice of coordinates agrees with *rational_class_group()*.

EXAMPLES:

```
sage: P1xP1 = toric_varieties.P1xP1()
sage: Kc = P1xP1.Kaehler_cone()
sage: Kc
2-d cone in 2-d lattice
sage: Kc.rays()
Divisor class [0, 1],
Divisor class [1, 0]
in Basis lattice of The toric rational divisor class group
of a 2-d CPR-Fano toric variety covered by 4 affine patches
sage: [ divisor_class.lift() for divisor_class in Kc.rays() ]
[V(y), V(t)]
sage: Kc.lattice()
Basis lattice of The toric rational divisor class group of a
2-d CPR-Fano toric variety covered by 4 affine patches
```

Mori_cone()

Returns the Mori cone of `self`.

OUTPUT: cone.

Note:

- The Mori cone is dual to the Kähler cone.
- We think of the Mori cone as living inside the row span of the Gale transform matrix (computed by `self.fan().Gale_transform()`).
- The points in the Mori cone are the effective curves in the variety.
- The `i`-th entry in each Mori vector is the intersection number of the curve corresponding to the generator of the `i`-th ray of the fan with the corresponding divisor class. The very last entry is associated to the origin of the fan lattice.
- The Mori vectors are also known as the gauged linear sigma model charge vectors.

EXAMPLES:

```
sage: P4_11169 = toric_varieties.P4_11169_resolved()
sage: P4_11169.Mori_cone()
2-d cone in 7-d lattice
sage: P4_11169.Mori_cone().rays()
(3, 2, 0, 0, 0,  1, -6),
(0, 0, 1, 1, 1, -3,  0)
in Ambient free module of rank 7 over the principal ideal domain Integer Ring
```

Spec(*cone=None*, *names=None*)

Return the spectrum associated to the dual cone.

Let $\sigma \in N_{\mathbf{R}}$ be a cone and $\sigma^\vee \cap M$ the associated semigroup of lattice points in the dual cone. Then

$$S = \mathbf{C}[\sigma^\vee \cap M]$$

is a $\mathbf{C}$-algebra. It is spanned over $\mathbf{C}$ by the points of $\sigma \cap N$, addition is formal linear combination of lattice points, and multiplication of lattice points is the semigroup law (that is, addition of lattice points). The $\mathbf{C}$-algebra S then defines a scheme $Spec(S)$.

For example, if $\sigma = \{(x, y) | x \geq 0, y \geq 0\}$ is the first quadrant then S is the polynomial ring in two variables. The associated scheme is $Spec(S) = \mathbf{C}^2$.

The same construction works over any base field, this introduction only used $\mathbf{C}$ for simplicity.

INPUT:

- `cone` – a `Cone`. Can be omitted for an affine toric variety, in which case the (unique) generating cone is used.
- `names` – (optional). Names of variables for the semigroup ring, see *normalize_names()* for acceptable formats. If not given, indexed variable names will be created automatically.

OUTPUT:

The spectrum of the semigroup ring $\mathbf{C}[\sigma^\vee \cap M]$.

EXAMPLES:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: AffineToricVariety(quadrant).Spec()
Spectrum of Multivariate Polynomial Ring in z0, z1 over Rational Field
```

A more interesting example:

```
sage: A2Z2 = Cone([(0,1), (2,1)])
sage: AffineToricVariety(A2Z2).Spec(names='u,v,t')                              #␣
↪needs fpylll sage.libs.singular
Spectrum of Quotient of Multivariate Polynomial Ring
in u, v, t over Rational Field by the ideal (-u*v + t^2)
```

Stanley_Reisner_ideal()

Return the Stanley-Reisner ideal.

OUTPUT:

- The Stanley-Reisner ideal in the polynomial ring over $\mathbf{Q}$ generated by the homogeneous coordinates.

EXAMPLES:

```
sage: fan = Fan([[0,1,3], [3,4], [2,0], [1,2,4]],
....:           [(-3, -2, 1), (0, 0, 1), (3, -2, 1), (-1, -1, 1), (1, -1, 1)])
sage: X = ToricVariety(fan, coordinate_names='A B C D E', base_field=GF(5))
sage: SR = X.Stanley_Reisner_ideal(); SR
Ideal (A*E, C*D, A*B*C, B*D*E) of
 Multivariate Polynomial Ring in A, B, C, D, E over Rational Field
```

Td(*deg=None*)

Return the Todd class (of the tangent bundle) of the toric variety.

INPUT:

- deg – integer (optional). The desired degree part.

OUTPUT:

- If the degree is specified, the degree-deg part of the Todd class.
- If no degree is specified, the total Todd class.

REFERENCES:

- Wikipedia article Todd_class

EXAMPLES:

```
sage: # needs sage.libs.singular
sage: dP6 = toric_varieties.dP6()
sage: dP6.Todd_class()
[-w^2 + 1/2*y + v + z + 1/2*w + 1]
sage: dP6.Td()
[-w^2 + 1/2*y + v + z + 1/2*w + 1]
sage: dP6.integrate( dP6.Td() )
1
```

Todd_class(*deg=None*)

Return the Todd class (of the tangent bundle) of the toric variety.

INPUT:

- deg – integer (optional). The desired degree part.

OUTPUT:

- If the degree is specified, the degree-deg part of the Todd class.

- If no degree is specified, the total Todd class.

REFERENCES:

- Wikipedia article Todd_class

EXAMPLES:

```
sage: # needs sage.libs.singular
sage: dP6 = toric_varieties.dP6()
sage: dP6.Todd_class()
[-w^2 + 1/2*y + v + z + 1/2*w + 1]
sage: dP6.Td()
[-w^2 + 1/2*y + v + z + 1/2*w + 1]
sage: dP6.integrate( dP6.Td() )
1
```

affine_algebraic_patch(*cone=None*, *names=None*)

Return the patch corresponding to `cone` as an affine algebraic subvariety.

INPUT:

- `cone` – a Cone σ of the fan. It can be omitted for an affine toric variety, in which case the single generating cone is used.

OUTPUT:

A *affine algebraic subscheme* corresponding to the patch $Spec(\sigma^\vee \cap M)$ associated to the cone σ.

See also *affine_patch()*, which expresses the patches as subvarieties of affine toric varieties instead.

EXAMPLES:

```
sage: cone = Cone([(0,1), (2,1)])
sage: A2Z2 = AffineToricVariety(cone)
sage: A2Z2.affine_algebraic_patch()                                    #␣
↪needs fpylll sage.libs.singular
Closed subscheme of Affine Space of dimension 3 over Rational Field defined by:
  -z0*z1 + z2^2
sage: A2Z2.affine_algebraic_patch(Cone([(0,1)]), names='x, y, t')      #␣
↪needs fpylll sage.libs.singular
Closed subscheme of Affine Space of dimension 3 over Rational Field defined by:
  1
```

affine_patch(*i*)

Return the `i`-th affine patch of `self`.

INPUT:

- `i` – integer, index of a generating cone of the fan of `self`.

OUTPUT:

- affine *toric variety* corresponding to the `i`-th generating cone of the fan of `self`.

The result is cached, so the `i`-th patch is always the same object in memory.

See also *affine_algebraic_patch()*, which expresses the patches as subvarieties of affine space instead.

EXAMPLES:

```
sage: fan = FaceFan(lattice_polytope.cross_polytope(2))
sage: P1xP1 = ToricVariety(fan, "x s y t")
sage: patch0 = P1xP1.affine_patch(0)
sage: patch0
2-d affine toric variety
sage: patch0.embedding_morphism()
Scheme morphism:
  From: 2-d affine toric variety
  To:   2-d toric variety covered by 4 affine patches
  Defn: Defined on coordinates by sending [y : t] to [1 : 1 : y : t]
sage: patch1 = P1xP1.affine_patch(1)
sage: patch1.embedding_morphism()
Scheme morphism:
  From: 2-d affine toric variety
  To:   2-d toric variety covered by 4 affine patches
  Defn: Defined on coordinates by sending [s : y] to [1 : s : y : 1]
sage: patch1 is P1xP1.affine_patch(1)
True
```

c(*deg=None*)

Return Chern classes of the (tangent bundle of the) toric variety.

INPUT:

- `deg` – integer (optional). The degree of the Chern class.

OUTPUT:

- If the degree is specified, the `deg`-th Chern class.
- If no degree is specified, the total Chern class.

REFERENCES:

- Wikipedia article Chern_class

EXAMPLES:

```
sage: # needs sage.libs.singular
sage: X = toric_varieties.dP6()
sage: X.Chern_class()
[-6*w^2 + y + 2*v + 2*z + w + 1]
sage: X.c()
[-6*w^2 + y + 2*v + 2*z + w + 1]
sage: X.c(1)
[y + 2*v + 2*z + w]
sage: X.c(2)
[-6*w^2]
sage: X.integrate( X.c(2) )
6
sage: X.integrate( X.c(2) ) == X.Euler_number()
True
```

cartesian_product(*other*, *coordinate_names=None*, *coordinate_indices=None*)

Return the Cartesian product of `self` with `other`.

INPUT:

- other – a *toric variety*;
- coordinate_names – names of variables for the coordinate ring, see *normalize_names()* for acceptable formats. If not given, indexed variable names will be created automatically;
- coordinate_indices – list of integers, indices for indexed variables. If not given, the index of each variable will coincide with the index of the corresponding ray of the fan.

OUTPUT: A *toric variety*.

EXAMPLES:

```
sage: P1 = ToricVariety(Fan([Cone([(1,)]), Cone([(-1,)])]))
sage: P1xP1 = P1.cartesian_product(P1); P1xP1
2-d toric variety covered by 4 affine patches
sage: P1xP1.fan().rays()
N+N(-1,  0),      N+N( 1,  0),
N+N( 0, -1),      N+N( 0,  1)
in 2-d lattice N+N
```

ch(*deg=None*)

Return the Chern character (of the tangent bundle) of the toric variety.

INPUT:

- deg – integer (optional). The degree of the Chern character.

OUTPUT:

- If the degree is specified, the degree-deg part of the Chern character.
- If no degree is specified, the total Chern character.

REFERENCES:

- Wikipedia article Chern_character#The_Chern_character

EXAMPLES:

```
sage: # needs sage.libs.singular
sage: dP6 = toric_varieties.dP6()
sage: dP6.Chern_character()
[3*w^2 + y + 2*v + 2*z + w + 2]
sage: dP6.ch()
[3*w^2 + y + 2*v + 2*z + w + 2]
sage: dP6.ch(1) == dP6.c(1)
True
```

change_ring(*F*)

Return a toric variety over F and otherwise the same as self.

INPUT:

- F – field.

OUTPUT: *toric variety* over F.

Note: There is no need to have any relation between F and the base field of self. If you do want to have such a relation, use base_extend() instead.

EXAMPLES:

```
sage: P1xA1 = toric_varieties.P1xA1()
sage: P1xA1.base_ring()
Rational Field
sage: P1xA1_RR = P1xA1.change_ring(RR)
sage: P1xA1_RR.base_ring()
Real Field with 53 bits of precision
sage: P1xA1_QQ = P1xA1_RR.change_ring(QQ)
sage: P1xA1_QQ.base_ring()
Rational Field
sage: P1xA1_RR.base_extend(QQ)
Traceback (most recent call last):
...
ValueError: no natural map from the base ring
(=Real Field with 53 bits of precision) to R (=Rational Field)!
sage: R = PolynomialRing(QQ, 2, 'a')
sage: P1xA1.change_ring(R)
Traceback (most recent call last):
...
TypeError: need a field to construct a toric variety;
got Multivariate Polynomial Ring in a0, a1 over Rational Field
```

chi()

Return the topological Euler number of the toric variety.

Sometimes, this is also called the Euler characteristic. *chi()* is a synonym for *Euler_number()*.

REFERENCES:

- Wikipedia article Euler_characteristic

EXAMPLES:

```
sage: # needs sage.libs.singular
sage: P1xP1 = toric_varieties.P1xP1()
sage: P1xP1.Euler_number()
4
sage: P1xP1.chi()
4
```

cohomology_basis(*d=None*)

Return a basis for the cohomology of the toric variety.

INPUT:

- d (optional) – integer.

OUTPUT:

- Without the optional argument, a list whose d-th entry is a basis for $H^{2d}(X, \mathbf{Q})$
- If the argument is an integer d, returns basis for $H^{2d}(X, \mathbf{Q})$

EXAMPLES:

```
sage: # needs sage.libs.singular
sage: X = toric_varieties.dP8()
sage: X.cohomology_basis()
```

(continues on next page)

(continued from previous page)

```
(([1],), ([z], [y]), ([y*z],))
sage: X.cohomology_basis(1)
([z], [y])
sage: X.cohomology_basis(dimension(X))[0] == X.volume_class()
True
```

cohomology_ring()

Return the cohomology ring of the toric variety.

OUTPUT:

- If the toric variety is over $\mathbf{C}$ and has at most finite orbifold singularities: $H^\bullet(X, \mathbf{Q})$ as a polynomial quotient ring.
- Other cases are not handled yet.

Note:

- Toric varieties over any field of characteristic 0 are treated as if they were varieties over $\mathbf{C}$.
- The integral cohomology of smooth toric varieties is torsion-free, so in this case there is no loss of information when going to rational coefficients.
- `self.cohomology_ring().gen(i)` is the divisor class corresponding to the `i`-th ray of the fan.

EXAMPLES:

```
sage: X = toric_varieties.dP6()
sage: X.cohomology_ring()
Rational cohomology ring of a 2-d CPR-Fano toric variety covered by 6 affine␣
↪patches
sage: X.cohomology_ring().defining_ideal()
Ideal (-u - y + z + w, x - y - v + w, x*y, x*v, x*z, u*v, u*z, u*w, y*z, y*w,␣
↪v*w)
 of Multivariate Polynomial Ring in x, u, y, v, z, w over Rational Field
sage: X.cohomology_ring().defining_ideal().ring()
Multivariate Polynomial Ring in x, u, y, v, z, w over Rational Field
sage: X.variable_names()
('x', 'u', 'y', 'v', 'z', 'w')
sage: X.cohomology_ring().gens()                                              #␣
↪needs sage.libs.singular
([y + v - w], [-y + z + w], [y], [v], [z], [w])
```

coordinate_ring()

Return the coordinate ring of `self`.

For toric varieties this is the homogeneous coordinate ring (a.k.a. Cox's ring and total ring).

OUTPUT: A polynomial ring.

EXAMPLES:

```
sage: P1xP1 = toric_varieties.P1xP1()
sage: P1xP1.coordinate_ring()
Multivariate Polynomial Ring in s, t, x, y over Rational Field
```

count_points()

Return the number of points of `self`.

This is an alias for `point_set().cardinality()`, see *cardinality()* for details.

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: V = ToricVariety(FaceFan(o))
sage: V2 = V.change_ring(GF(2))
sage: V2.point_set().cardinality()
27
sage: V2.count_points()
27
```

dimension_singularities()

Return the dimension of the singular set.

OUTPUT:

Integer. The dimension of the singular set of the toric variety. Often the singular set is a reducible subvariety, and this method will return the dimension of the largest-dimensional component.

Returns -1 if the toric variety is smooth.

EXAMPLES:

```
sage: toric_varieties.P4_11169().dimension_singularities()
1
sage: toric_varieties.Conifold().dimension_singularities()
0
sage: toric_varieties.P2().dimension_singularities()
-1
```

divisor(*arg*, *base_ring=None*, *check=True*, *reduce=True*)

Return a divisor.

INPUT:

The arguments are the same as in *sage.schemes.toric.divisor.ToricDivisor()*, with the exception of defining a divisor with a single integer: this method considers it to be the index of a ray of the *fan()* of `self`.

OUTPUT:

- A *sage.schemes.toric.divisor.ToricDivisor_generic*

EXAMPLES:

```
sage: dP6 = toric_varieties.dP6()
sage: dP6.coordinate_ring()
Multivariate Polynomial Ring in x, u, y, v, z, w over Rational Field
sage: dP6.divisor(list(range(6)))
V(u) + 2*V(y) + 3*V(v) + 4*V(z) + 5*V(w)
sage: dP6.inject_variables()
Defining x, u, y, v, z, w
sage: dP6.divisor(x*u^3)
V(x) + 3*V(u)
```

You can also construct divisors based on ray indices:

```
sage: dP6.divisor(0)
V(x)
sage: for i in range(dP6.fan().nrays()):
....:     print('{} : generated by ray {}'.format(dP6.divisor(i),
....:           dP6.fan().ray(i)))
V(x) : generated by ray N(0, 1)
V(u) : generated by ray N(-1, 0)
V(y) : generated by ray N(-1, -1)
V(v) : generated by ray N(0, -1)
V(z) : generated by ray N(1, 0)
V(w) : generated by ray N(1, 1)
```

divisor_group(*base_ring=Integer Ring*)

Return the group of Weil divisors.

INPUT:

- `base_ring` – the coefficient ring, usually `ZZ` (default) or `QQ`.

OUTPUT:

The (free abelian) group of Cartier divisors, that is, formal linear combinations of polynomial equations over the coefficient ring `base_ring`.

These need not be toric (=defined by monomials), but allow general polynomials. The output will be an instance of *sage.schemes.generic.divisor_group.DivisorGroup_generic*.

Warning: You almost certainly want the group of toric divisors, see *toric_divisor_group()*. The toric divisor group is generated by the rays of the fan. The general divisor group has no toric functionality implemented.

EXAMPLES:

```
sage: dP6 = toric_varieties.dP6()
sage: Div = dP6.divisor_group(); Div
Group of ZZ-Divisors on 2-d CPR-Fano toric variety covered by 6 affine patches
sage: Div(x)                                                                #␣
↪needs sage.symbolic
V(x)
```

embedding_morphism()

Return the default embedding morphism of `self`.

Such a morphism is always defined for an affine patch of a toric variety (which is also a toric varieties itself).

OUTPUT:

- `scheme morphism` if the default embedding morphism was defined for `self`, otherwise a `ValueError` exception is raised.

EXAMPLES:

```
sage: fan = FaceFan(lattice_polytope.cross_polytope(2))
sage: P1xP1 = ToricVariety(fan, "x s y t")
sage: P1xP1.embedding_morphism()
Traceback (most recent call last):
...
ValueError: no default embedding was defined for this toric variety
sage: patch = P1xP1.affine_patch(0)
sage: patch
2-d affine toric variety
sage: patch.embedding_morphism()
Scheme morphism:
  From: 2-d affine toric variety
  To:   2-d toric variety covered by 4 affine patches
  Defn: Defined on coordinates by sending [y : t] to [1 : 1 : y : t]
```

fan(*dim=None*, *codim=None*)

Return the underlying fan of `self` or its cones.

INPUT:

- `dim` – dimension of the requested cones;
- `codim` – codimension of the requested cones.

OUTPUT:

- `rational polyhedral fan` if no parameters were given, `tuple` of `cones` otherwise.

EXAMPLES:

```
sage: fan = FaceFan(lattice_polytope.cross_polytope(2))
sage: P1xP1 = ToricVariety(fan)
sage: P1xP1.fan()
Rational polyhedral fan in 2-d lattice M
sage: P1xP1.fan() is fan
True
sage: P1xP1.fan(1)[0]
1-d cone of Rational polyhedral fan in 2-d lattice M
```

inject_coefficients(*scope=None*, *verbose=True*)

Inject generators of the base field of `self` into `scope`.

This function is useful if the base field is the field of rational functions.

INPUT:

- `scope` – namespace (default: global, not just the scope from which this function was called);
- `verbose` – if `True` (default), names of injected generators will be printed.

OUTPUT: None.

EXAMPLES:

```
sage: fan = FaceFan(lattice_polytope.cross_polytope(2))
sage: F = QQ["a, b"].fraction_field()
sage: P1xP1 = ToricVariety(fan, base_field=F)
sage: P1xP1.inject_coefficients()
Defining a, b
```

We check that we can use names a and b, github issue #10498 is fixed:

```
sage: a + b
a + b
sage: a + b in P1xP1.coordinate_ring()
True
```

integrate(*cohomology_class*)

Integrate a cohomology class over the toric variety.

INPUT:

- `cohomology_class` – A cohomology class given as a polynomial in `self.cohomology_ring()`

OUTPUT:

The integral of the cohomology class over the variety. The volume normalization is given by *volume_class()*, that is, `self.integrate(self.volume_class())` is always one (if the volume class exists).

EXAMPLES:

```
sage: # needs sage.libs.singular
sage: dP6 = toric_varieties.dP6()
sage: HH = dP6.cohomology_ring()
sage: D = [ HH(c) for c in dP6.fan(dim=1) ]
sage: matrix([ [ D[i]*D[j] for i in range(0,6) ] for j in range(0,6) ])
[ [w^2] [-w^2]    [0]    [0]    [0] [-w^2]]
[[-w^2]  [w^2] [-w^2]    [0]    [0]    [0]]
[   [0] [-w^2]  [w^2] [-w^2]    [0]    [0]]
[   [0]    [0] [-w^2]  [w^2] [-w^2]    [0]]
[   [0]    [0]    [0] [-w^2]  [w^2] [-w^2]]
[[-w^2]    [0]    [0]    [0] [-w^2]  [w^2]]
sage: matrix([ [ dP6.integrate(D[i]*D[j]) for i in range(0,6) ] for j in␣
↪range(0,6) ])
[-1  1  0  0  0  1]
[ 1 -1  1  0  0  0]
[ 0  1 -1  1  0  0]
[ 0  0  1 -1  1  0]
[ 0  0  0  1 -1  1]
[ 1  0  0  0  1 -1]
```

If the toric variety is an orbifold, the intersection numbers are usually fractional:

```
sage: # needs sage.libs.singular
sage: P2_123 = toric_varieties.P2_123()
sage: HH = P2_123.cohomology_ring()
sage: D = [ HH(c) for c in P2_123.fan(dim=1) ]
sage: matrix([ [ P2_123.integrate(D[i]*D[j]) for i in range(0,3) ] for j in␣
↪range(0,3) ])
[2/3   1 1/3]
[  1 3/2 1/2]
[1/3 1/2 1/6]
sage: A = P2_123.Chow_group(QQ)
sage: matrix([ [ A(P2_123.divisor(i))
....:            .intersection_with_divisor(P2_123.divisor(j))
```

(continues on next page)

(continued from previous page)

```
....:             .count_points() for i in range(0,3) ] for j in range(0,3) ])
[2/3   1 1/3]
[  1 3/2 1/2]
[1/3 1/2 1/6]
```

is_affine()

Check if `self` is an affine toric variety.

An affine toric variety is a toric variety whose fan is the face lattice of a single cone. See also *AffineToricVariety()*.

OUTPUT: A boolean.

EXAMPLES:

```
sage: toric_varieties.A2().is_affine()
True
sage: toric_varieties.P1xA1().is_affine()
False
```

is_complete()

Check if `self` is complete.

OUTPUT:

- `True` if `self` is complete and `False` otherwise.

EXAMPLES:

```
sage: P1xP1 = toric_varieties.P1xP1()
sage: P1xP1.is_complete()
True
sage: P1xP1.affine_patch(0).is_complete()
False
```

is_homogeneous(*polynomial*)

Check if `polynomial` is homogeneous.

The coordinate ring of a toric variety is multigraded by relations between generating rays of the underlying fan.

INPUT:

- `polynomial` – polynomial in the coordinate ring of `self` or its quotient.

OUTPUT:

- `True` if `polynomial` is homogeneous and `False` otherwise.

EXAMPLES:

We will use the product of two projective lines with coordinates (x, y) for one and (s, t) for the other:

```
sage: P1xP1.<x,y,s,t> = toric_varieties.P1xP1()
sage: P1xP1.is_homogeneous(x - y)
True
sage: P1xP1.is_homogeneous(x*s + y*t)
True
```

(continues on next page)

(continued from previous page)

```
sage: P1xP1.is_homogeneous(x - t)
False
sage: P1xP1.is_homogeneous(1)
True
```

Note that by homogeneous, we mean well-defined with respect to the homogeneous rescalings of `self`. So a polynomial that you would usually not call homogeneous can be homogeneous if there are no homogeneous rescalings, for example:

```
sage: A1.<z> = toric_varieties.A1()
sage: A1.is_homogeneous(z^3 + z^7)
True
```

Finally, the degree group is really the Chow group $A_{d-1}(X)$ and can contain torsion. For example, take $\mathbf{C}^2/\mathbf{Z}_2$. Here, the Chow group is $A_{d-1}(\mathbf{C}^2/\mathbf{Z}_2) = \mathbf{Z}_2$ and distinguishes even-degree homogeneous polynomials from odd-degree homogeneous polynomials:

```
sage: A2_Z2.<x,y> = toric_varieties.A2_Z2()
sage: A2_Z2.is_homogeneous(x + y + x^3 + y^5 + x^3*y^4)
True
sage: A2_Z2.is_homogeneous(x^2 + x*y + y^4 + (x*y)^5 + x^4*y^4)
True
sage: A2_Z2.is_homogeneous(x + y^2)
False
```

is_isomorphic(*another*)

Check if `self` is isomorphic to `another`.

INPUT:

- `another` - *toric variety*.

OUTPUT:

- `True` if `self` and `another` are isomorphic, `False` otherwise.

EXAMPLES:

```
sage: TV1 = toric_varieties.P1xA1()
sage: TV2 = toric_varieties.P1xP1()
```

Only the most trivial case is implemented so far:

```
sage: TV1.is_isomorphic(TV1)
True
sage: TV1.is_isomorphic(TV2)
Traceback (most recent call last):
...
NotImplementedError: isomorphism check is not yet implemented
```

is_orbifold()

Check if `self` has only quotient singularities.

A toric variety with at most orbifold singularities (in this sense) is often called a simplicial toric variety. In this package, we generally try to avoid this term since it mixes up differential geometry and cone terminology.

OUTPUT:

- `True` if `self` has at most quotient singularities by finite groups, `False` otherwise.

EXAMPLES:

```
sage: fan1 = FaceFan(lattice_polytope.cross_polytope(2))
sage: P1xP1 = ToricVariety(fan1)
sage: P1xP1.is_orbifold()
True
sage: fan2 = NormalFan(lattice_polytope.cross_polytope(3))
sage: TV = ToricVariety(fan2)
sage: TV.is_orbifold()
False
```

is_smooth()

Check if `self` is smooth.

OUTPUT:

- `True` if `self` is smooth and `False` otherwise.

EXAMPLES:

```
sage: fan1 = FaceFan(lattice_polytope.cross_polytope(2))
sage: P1xP1 = ToricVariety(fan1)
sage: P1xP1.is_smooth()
True
sage: fan2 = NormalFan(lattice_polytope.cross_polytope(2))
sage: TV = ToricVariety(fan2)
sage: TV.is_smooth()
False
```

linear_equivalence_ideal()

Return the ideal generated by linear relations

OUTPUT:

- The ideal generated by the linear relations of the rays in the polynomial ring over $\mathbf{Q}$ generated by the homogeneous coordinates.

EXAMPLES:

```
sage: fan = Fan([[0,1,3], [3,4], [2,0], [1,2,4]],
....:           [(-3, -2, 1), (0, 0, 1), (3, -2, 1), (-1, -1, 1), (1, -1, 1)])
sage: X = ToricVariety(fan, coordinate_names='A B C D E', base_field=GF(5))
sage: lin = X.linear_equivalence_ideal(); lin
Ideal (-3*A + 3*C - D + E, -2*A - 2*C - D - E, A + B + C + D + E) of
 Multivariate Polynomial Ring in A, B, C, D, E over Rational Field
```

orbit_closure(*cone*)

Return the orbit closure of `cone`.

The cones σ of a fan Σ are in one-to-one correspondence with the torus orbits $O(\sigma)$ of the corresponding toric variety X_Σ. Each orbit is isomorphic to a lower dimensional torus (of dimension equal to the codimension of σ). Just like the toric variety X_Σ itself, these orbits are (partially) compactified by lower-dimensional orbits. In particular, one can define the closure $V(\sigma)$ of the torus orbit $O(\sigma)$ in the ambient toric variety X_Σ, which is again a toric variety.

See Proposition 3.2.7 of [CLS2011] for more details.

INPUT:

- cone – a cone of the fan.

OUTPUT:

- a torus orbit closure associated to cone as a *toric variety*.

EXAMPLES:

```
sage: P1xP1 = toric_varieties.P1xP1()
sage: H = P1xP1.fan(1)[0]
sage: V = P1xP1.orbit_closure(H);  V
1-d toric variety covered by 2 affine patches
sage: V.embedding_morphism()
Scheme morphism:
  From: 1-d toric variety covered by 2 affine patches
  To:   2-d CPR-Fano toric variety covered by 4 affine patches
  Defn: Defined by embedding the torus closure associated to the 1-d
        cone of Rational polyhedral fan in 2-d lattice N.
sage: V.embedding_morphism().as_polynomial_map()
Scheme morphism:
  From: 1-d toric variety covered by 2 affine patches
  To:   2-d CPR-Fano toric variety covered by 4 affine patches
  Defn: Defined on coordinates by sending [z0 : z1] to [0 : 1 : z1 : z0]
```

plot(***options*)

Plot self, i.e. the corresponding fan.

INPUT:

- any options for toric plots (see toric_plotter.options), none are mandatory.

OUTPUT: A plot.

Note: The difference between X.plot() and X.fan().plot() is that in the first case default ray labels correspond to variables of X.

EXAMPLES:

```
sage: X = toric_varieties.Cube_deformation(4)
sage: X.plot()                                                          #␣
↪needs sage.plot
Graphics3d Object
```

rational_class_group()

Return the rational divisor class group of self.

Let X be a toric variety.

The **Weil divisor class group** $Cl(X)$ is a finitely generated abelian group and can contain torsion. Its rank equals the number of rays in the fan of X minus the dimension of X.

The **rational divisor class group** is $Cl(X) \otimes_{\mathbf{Z}} \mathbf{Q}$ and never includes torsion. If X is *smooth*, this equals the **Picard group** of X, whose elements are the isomorphism classes of line bundles on X. The group law

(which we write as addition) is the tensor product of the line bundles. The Picard group of a toric variety is always torsion-free.

OUTPUT: *rational divisor class group*.

Note:

- Coordinates correspond to the rows of `self.fan().gale_transform()`.
- *Kaehler_cone()* yields a cone in this group.

EXAMPLES:

```
sage: P1xA1 = toric_varieties.P1xA1()
sage: P1xA1.rational_class_group()
The toric rational divisor class group
of a 2-d toric variety covered by 2 affine patches
```

resolve(***kwds*)

Construct a toric variety whose fan subdivides the fan of `self`.

The name of this function reflects the fact that usually such subdivisions are done for resolving singularities of the original variety.

INPUT:

This function accepts only keyword arguments, none of which are mandatory.

- `coordinate_names` – names for coordinates of the new variety. If not given, will be constructed from the coordinate names of `self` and necessary indexed ones. See *normalize_names()* for the description of acceptable formats;
- `coordinate_indices` – coordinate indices which should be used for indexed variables of the new variety;
- all other arguments will be passed to `subdivide()` method of the underlying `rational polyhedral fan`, see its documentation for the available options.

OUTPUT: A *toric variety*.

EXAMPLES:

First we will "manually" resolve a simple orbifold singularity:

```
sage: cone = Cone([(1,1), (-1,1)])
sage: fan = Fan([cone])
sage: TV = ToricVariety(fan)
sage: TV.is_smooth()
False
sage: TV_res = TV.resolve(new_rays=[(0,1)])
sage: TV_res.is_smooth()
True
sage: TV_res.fan().rays()
N( 1, 1),
N(-1, 1),
N( 0, 1)
in 2-d lattice N
sage: [cone.ambient_ray_indices() for cone in TV_res.fan()]
[(0, 2), (1, 2)]
```

Now let's "automatically" partially resolve a more complicated fan:

```
sage: fan = NormalFan(lattice_polytope.cross_polytope(3))
sage: TV = ToricVariety(fan)
sage: TV.is_smooth()
False
sage: TV.is_orbifold()
False
sage: TV.fan().nrays()
8
sage: TV.fan().ngenerating_cones()
6
sage: TV_res = TV.resolve(make_simplicial=True)
sage: TV_res.is_smooth()
False
sage: TV_res.is_orbifold()
True
sage: TV_res.fan().nrays()
8
sage: TV_res.fan().ngenerating_cones()
12
sage: TV.gens()
(z0, z1, z2, z3, z4, z5, z6, z7)
sage: TV_res.gens()
(z0, z1, z2, z3, z4, z5, z6, z7)
sage: TV_res = TV.resolve(coordinate_names="x+",
....:                     make_simplicial=True)
sage: TV_res.gens()
(x0, x1, x2, x3, x4, x5, x6, x7)
```

resolve_to_orbifold(***kwds*)

Construct an orbifold whose fan subdivides the fan of `self`.

It is a synonym for *resolve()* with `make_simplicial=True` option.

INPUT:

- this function accepts only keyword arguments. See *resolve()* for documentation.

OUTPUT: A *toric variety*.

EXAMPLES:

```
sage: fan = NormalFan(lattice_polytope.cross_polytope(3))
sage: TV = ToricVariety(fan)
sage: TV.is_orbifold()
False
sage: TV.fan().nrays()
8
sage: TV.fan().ngenerating_cones()
6
sage: TV_res = TV.resolve_to_orbifold()
sage: TV_res.is_orbifold()
True
sage: TV_res.fan().nrays()
8
```

(continues on next page)

(continued from previous page)

```
sage: TV_res.fan().ngenerating_cones()
12
```

property sheaves

Return the factory object for sheaves on the toric variety.

See *sage.schemes.toric.sheaf.constructor.SheafLibrary* for details.

EXAMPLES:

```
sage: dP6 = toric_varieties.dP6()
sage: dP6.sheaves
Sheaf constructor on 2-d CPR-Fano toric variety covered by 6 affine patches
sage: dP6.sheaves.trivial_bundle()
Rank 1 bundle on 2-d CPR-Fano toric variety covered by 6 affine patches.
```

subscheme(*polynomials*)

Return the subscheme of `self` defined by `polynomials`.

INPUT:

- `polynomials` – list of polynomials in the coordinate ring of `self`.

OUTPUT: A *subscheme of a toric variety*.

EXAMPLES:

We will construct a subscheme of the product of two projective lines with coordinates (x, y) for one and (s, t) for the other:

```
sage: P1xP1.<x,y,s,t> = toric_varieties.P1xP1()
sage: X = P1xP1.subscheme([x*s + y*t, x^3 + y^3])
sage: X
Closed subscheme of 2-d CPR-Fano toric variety
covered by 4 affine patches defined by:
  x*s + y*t,
  x^3 + y^3
sage: X.defining_polynomials()
(x*s + y*t, x^3 + y^3)
sage: X.defining_ideal()
Ideal (x*s + y*t, x^3 + y^3) of
 Multivariate Polynomial Ring in x, y, s, t over Rational Field
sage: X.base_ring()
Rational Field
sage: X.base_scheme()
Spectrum of Rational Field
sage: X.structure_morphism()
Scheme morphism:
  From: Closed subscheme of 2-d CPR-Fano toric variety
        covered by 4 affine patches defined by: x*s + y*t, x^3 + y^3
  To:   Spectrum of Rational Field
  Defn: Structure map
```

toric_divisor_group(*base_ring=Integer Ring*)

Return the group of toric (T-Weil) divisors.

INPUT:

- `base_ring` – the coefficient ring, usually `ZZ` (default) or `QQ`.

OUTPUT:

The free Abelian agroup of toric Weil divisors, that is, formal `base_ring`-linear combinations of codimension-one toric subvarieties. The output will be an instance of *sage.schemes.toric.divisor.ToricDivisorGroup*.

The i-th generator of the divisor group is the divisor where the i-th homogeneous coordinate vanishes, $\{z_i = 0\}$.

EXAMPLES:

```
sage: dP6 = toric_varieties.dP6()
sage: TDiv = dP6.toric_divisor_group(); TDiv
Group of toric ZZ-Weil divisors on 2-d CPR-Fano toric variety
covered by 6 affine patches
sage: TDiv == dP6.toric_divisor_group()
True
sage: TDiv.gens()
(V(x), V(u), V(y), V(v), V(z), V(w))
sage: dP6.coordinate_ring()
Multivariate Polynomial Ring in x, u, y, v, z, w over Rational Field
```

volume_class()

Return the cohomology class of the volume form on the toric variety.

Note that we are using cohomology with compact supports. If the variety is non-compact this is dual to homology without any support condition. In particular, for non-compact varieties the volume form $\mathrm{dVol} = \wedge_i (dx_i \wedge dy_i)$ does not define a (non-zero) cohomology class.

OUTPUT:

A *CohomologyClass*. If it exists, it is the class of the (properly normalized) volume form, that is, it is the Poincaré dual of a single point. If it does not exist, a `ValueError` is raised.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: P2.volume_class()                                                 #␣
↪needs sage.libs.singular
[z^2]

sage: A2_Z2 = toric_varieties.A2_Z2()
sage: A2_Z2.volume_class()                                              #␣
↪needs sage.libs.singular
Traceback (most recent call last):
...
ValueError: volume class does not exist
```

If none of the maximal cones is smooth things get more tricky. In this case no torus-fixed point is smooth. If we want to count an ordinary point as 1, then a G-orbifold point needs to count as $\frac{1}{|G|}$. For example, take $\mathbb{P}^1 \times \mathbb{P}^1$ with inhomogeneous coordinates (t, y). Take the quotient by the action $(t, y) \mapsto (-t, -y)$. The $\mathbf{Z}_2$-invariant Weil divisors $\{t = 0\}$ and $\{y = 0\}$ intersect in a $\mathbf{Z}_2$-fixed point, so they ought to have intersection number $\frac{1}{2}$. This means that the cohomology class $[t] \cap [y]$ should be $\frac{1}{2}$ times the volume class. Note that this is different from the volume normalization chosen in [KS]:

```
sage: P1xP1_Z2 = toric_varieties.P1xP1_Z2()
sage: Dt = P1xP1_Z2.divisor(1);  Dt
V(t)
sage: Dy = P1xP1_Z2.divisor(3);  Dy
V(y)
sage: P1xP1_Z2.volume_class()                                                  #␣
↪needs sage.libs.singular
[2*t*y]

sage: HH = P1xP1_Z2.cohomology_ring()
sage: HH(Dt) * HH(Dy) == 1/2 * P1xP1_Z2.volume_class()                         #␣
↪needs sage.libs.singular
True
```

The fractional coefficients are also necessary to match the normalization in the rational Chow group for simplicial toric varieties:

```
sage: A = P1xP1_Z2.Chow_group(QQ)
sage: A(Dt).intersection_with_divisor(Dy).count_points()
1/2
```

sage.schemes.toric.variety.**is_CohomologyClass**(*x*)

Check whether `x` is a cohomology class of a toric variety.

INPUT:

- `x` – anything.

OUTPUT:

`True` or `False` depending on whether `x` is an instance of *CohomologyClass*

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: HH = P2.cohomology_ring()
sage: from sage.schemes.toric.variety import is_CohomologyClass
sage: is_CohomologyClass( HH.one() )                                           #␣
↪needs sage.libs.singular
True
sage: is_CohomologyClass( HH(P2.fan(1)[0]) )                                   #␣
↪needs sage.libs.singular
True
sage: is_CohomologyClass('z')
False
```

sage.schemes.toric.variety.**is_ToricVariety**(*x*)

Check if `x` is a toric variety.

INPUT:

- `x` – anything.

OUTPUT:

- `True` if `x` is a *toric variety* and `False` otherwise.

Note: While projective spaces are toric varieties mathematically, they are not toric varieties in Sage due to efficiency considerations, so this function will return `False`.

EXAMPLES:

```
sage: from sage.schemes.toric.variety import is_ToricVariety
sage: is_ToricVariety(1)
False
sage: fan = FaceFan(lattice_polytope.cross_polytope(2))
sage: P = ToricVariety(fan)
sage: P
2-d toric variety covered by 4 affine patches
sage: is_ToricVariety(P)
True
sage: is_ToricVariety(ProjectiveSpace(2))
False
```

sage.schemes.toric.variety.**normalize_names**(*names=None*, *ngens=None*, *prefix=None*, *indices=None*, *return_prefix=False*)

Return a list of names in the standard form.

INPUT:

All input parameters are optional.

- `names` – names given either as a single string (with individual names separated by commas or spaces) or a list of strings with each string specifying a name. If the last name ends with the plus sign, "+", this name will be used as `prefix` (even if `prefix` was given explicitly);
- `ngens` – number of names to be returned;
- `prefix` – prefix for the indexed names given as a string;
- `indices` – list of integers (default: `range(ngens)`) used as indices for names with `prefix`. If given, must be of length `ngens`;
- `return_prefix` – if `True`, the last element of the returned list will contain the prefix determined from `names` or given as the parameter `prefix`. This is useful if you may need more names in the future.

OUTPUT:

- list of names given as strings.

These names are constructed in the following way:

1. If necessary, split `names` into separate names.
2. If the last name ends with "+", put it into `prefix`.
3. If `ngens` was given, add to the names obtained so far as many indexed names as necessary to get this number. If the `k`-th name of the *total* list of names is indexed, it is `prefix + str(indices[k])`. If there were already more names than `ngens`, discard "extra" ones.
4. Check if constructed names are valid. See `certify_names()` for details.
5. If the option `return_prefix=True` was given, add `prefix` to the end of the list.

EXAMPLES:

As promised, all parameters are optional:

```
sage: from sage.schemes.toric.variety import normalize_names
sage: normalize_names()
[]
```

One of the most common uses is probably this one:

```
sage: normalize_names("x+", 4)
['x0', 'x1', 'x2', 'x3']
```

Now suppose that you want to enumerate your variables starting with one instead of zero:

```
sage: normalize_names("x+", 4, indices=list(range(1,5)))
['x1', 'x2', 'x3', 'x4']
```

You may actually have an arbitrary enumeration scheme:

```
sage: normalize_names("x+", 4, indices=[1, 10, 100, 1000])
['x1', 'x10', 'x100', 'x1000']
```

Now let's add some "explicit" names:

```
sage: normalize_names("x y z t+", 4)
['x', 'y', 'z', 't3']
```

Note that the "automatic" name is `t3` instead of `t0`. This may seem weird, but the reason for this behaviour is that the fourth name in this list will be the same no matter how many explicit names were given:

```
sage: normalize_names("x y t+", 4)
['x', 'y', 't2', 't3']
```

This is especially useful if you get `names` from a user but want to specify all default names:

```
sage: normalize_names("x, y", 4, prefix="t")
['x', 'y', 't2', 't3']
```

In this format, the user can easily override your choice for automatic names:

```
sage: normalize_names("x y s+", 4, prefix="t")
['x', 'y', 's2', 's3']
```

Let's now use all parameters at once:

```
sage: normalize_names("x, y, s+", 4, prefix="t",
....:     indices=list(range(1,5)), return_prefix=True)
['x', 'y', 's3', 's4', 's']
```

Note that you still need to give indices for all names, even if some of the first ones will be "wasted" because of the explicit names. The reason is the same as before - this ensures consistency of automatically generated names, no matter how many explicit names were given.

The prefix is discarded if `ngens` was not given:

```
sage: normalize_names("alpha, beta, gamma, zeta+")
['alpha', 'beta', 'gamma']
```

Finally, let's take a look at some possible mistakes:

```
sage: normalize_names("123")
Traceback (most recent call last):
...
ValueError: variable name '123' does not start with a letter
```

A more subtle one:

```
sage: normalize_names("x1", 4, prefix="x")
Traceback (most recent call last):
...
ValueError: variable name 'x1' appears more than once
```

16.2 Fano toric varieties

This module provides support for (Crepant Partial Resolutions of) Fano toric varieties, corresponding to crepant subdivisions of face fans of reflexive `lattice polytopes`. The interface is provided via *CPRFanoToricVariety()*.

A careful exposition of different flavours of Fano varieties can be found in the paper by Benjamin Nill [Nil2005]. The main goal of this module is to support work with **Gorenstein weak Fano toric varieties**. Such a variety corresponds to a **coherent crepant refinement of the normal fan of a reflexive polytope** Δ, where crepant means that primitive generators of the refining rays lie on the facets of the polar polytope Δ° and coherent (a.k.a. regular or projective) means that there exists a strictly upper convex piecewise linear function whose domains of linearity are precisely the maximal cones of the subdivision. These varieties are important for string theory in physics, as they serve as ambient spaces for mirror pairs of Calabi-Yau manifolds via constructions due to Victor V. Batyrev [Bat1994] and Lev A. Borisov [Bor1993].

From the combinatorial point of view "crepant" requirement is much more simple and natural to work with than "coherent." For this reason, the code in this module will allow work with arbitrary crepant subdivisions without checking whether they are coherent or not. We refer to corresponding toric varieties as **CPR-Fano toric varieties**.

REFERENCES:

- [Bat1994]
- [Bor1993]
- [CD2007]
- [Nil2005]

AUTHORS:

- Andrey Novoseltsev (2010-05-18): initial version.

EXAMPLES:

Most of the functions available for Fano toric varieties are the same as for general toric varieties, so here we will concentrate only on Calabi-Yau subvarieties, which were the primary goal for creating this module.

For our first example we realize the projective plane as a Fano toric variety:

```
sage: simplex = LatticePolytope([(1,0), (0,1), (-1,-1)])
sage: P2 = CPRFanoToricVariety(Delta_polar=simplex)
```

Its anticanonical "hypersurface" is a one-dimensional Calabi-Yau manifold:

```
sage: P2.anticanonical_hypersurface(monomial_points="all")
Closed subscheme of 2-d CPR-Fano toric variety covered by 3 affine patches defined by:
  a0*z0^3 + a9*z0^2*z1 + a7*z0*z1^2 + a1*z1^3 + a8*z0^2*z2 + a6*z0*z1*z2
  + a4*z1^2*z2 + a5*z0*z2^2 + a3*z1*z2^2 + a2*z2^3
```

In many cases it is sufficient to work with the "simplified polynomial moduli space" of anticanonical hypersurfaces:

```
sage: P2.anticanonical_hypersurface(monomial_points="simplified")
Closed subscheme of 2-d CPR-Fano toric variety covered by 3 affine patches defined by:
  a0*z0^3 + a1*z1^3 + a6*z0*z1*z2 + a2*z2^3
```

The mirror family to these hypersurfaces lives inside the Fano toric variety obtained using `simplex` as `Delta` instead of `Delta_polar`:

```
sage: FTV = CPRFanoToricVariety(Delta=simplex, coordinate_points="all")
sage: FTV.anticanonical_hypersurface(monomial_points="simplified")
Closed subscheme of 2-d CPR-Fano toric variety covered by 9 affine patches defined by:
  a2*z2^3*z3^2*z4*z5^2*z8 + a1*z1^3*z3*z4^2*z7^2*z9
  + a3*z0*z1*z2*z3*z4*z5*z7*z8*z9 + a0*z0^3*z5*z7*z8^2*z9^2
```

Here we have taken the resolved version of the ambient space for the mirror family, but in fact we don't have to resolve singularities corresponding to the interior points of facets - they are singular points which do not lie on a generic anticanonical hypersurface:

```
sage: FTV = CPRFanoToricVariety(Delta=simplex, coordinate_points="all but facets")
sage: FTV.anticanonical_hypersurface(monomial_points="simplified")
Closed subscheme of 2-d CPR-Fano toric variety covered by 3 affine patches defined by:
  a0*z0^3 + a1*z1^3 + a3*z0*z1*z2 + a2*z2^3
```

This looks very similar to our second version of the anticanonical hypersurface of the projective plane, as expected, since all one-dimensional Calabi-Yau manifolds are elliptic curves!

Now let's take a look at a toric realization of M-polarized K3 surfaces studied by Adrian Clingher and Charles F. Doran in [CD2007]:

```
sage: p4318 = ReflexivePolytope(3, 4318)
sage: FTV = CPRFanoToricVariety(Delta_polar=p4318)
sage: FTV.anticanonical_hypersurface()
Closed subscheme of 3-d CPR-Fano toric variety covered by 4 affine patches defined by:
  a0*z2^12 + a4*z2^6*z3^6 + a3*z3^12 + a8*z0*z1*z2*z3 + a2*z1^3 + a1*z0^2
```

Below you will find detailed descriptions of available functions. Current functionality of this module is very basic, but it is under active development and hopefully will improve in future releases of Sage. If there are some particular features that you would like to see implemented ASAP, please consider reporting them to the Sage Development Team or even implementing them on your own as a patch for inclusion!

class sage.schemes.toric.fano_variety.**AnticanonicalHypersurface**(*P_Delta*, *monomial_points=None*, *coefficient_names=None*, *coefficient_name_indices=None*, *coefficients=None*)

Bases: *AlgebraicScheme_subscheme_toric*

Construct an anticanonical hypersurface of a CPR-Fano toric variety.

INPUT:

- P_Delta – *CPR-Fano toric variety* associated to a reflexive polytope Δ;
- see *CPRFanoToricVariety_field.anticanonical_hypersurface()* for documentation on all other acceptable parameters.

OUTPUT:

- *anticanonical hypersurface* of P_Delta (with the extended base field, if necessary).

EXAMPLES:

```
sage: P1xP1 = toric_varieties.P1xP1()
sage: import sage.schemes.toric.fano_variety as ftv
sage: ftv.AnticanonicalHypersurface(P1xP1)
Closed subscheme of 2-d CPR-Fano toric variety
 covered by 4 affine patches defined by:
  a0*s^2*x^2 + a3*t^2*x^2 + a6*s*t*x*y + a1*s^2*y^2 + a2*t^2*y^2
```

See *anticanonical_hypersurface()* for a more elaborate example.

sage.schemes.toric.fano_variety.**CPRFanoToricVariety**(*Delta=None*, *Delta_polar=None*, *coordinate_points=None*, *charts=None*, *coordinate_names=None*, *names=None*, *coordinate_name_indices=None*, *make_simplicial=False*, *base_ring=None*, *base_field=None*, *check=True*)

Construct a CPR-Fano toric variety.

Note: See documentation of the module *fano_variety* for the used definitions and supported varieties.

Due to the large number of available options, it is recommended to always use keyword parameters.

INPUT:

- Delta – reflexive lattice polytope. The fan of the constructed CPR-Fano toric variety will be a crepant subdivision of the *normal fan* of Delta. Either Delta or Delta_polar must be given, but not both at the same time, since one is completely determined by another via polar method;
- Delta_polar – reflexive lattice polytope. The fan of the constructed CPR-Fano toric variety will be a crepant subdivision of the *face fan* of Delta_polar. Either Delta or Delta_polar must be given, but not both at the same time, since one is completely determined by another via polar method;
- coordinate_points – list of integers or string. A list will be interpreted as indices of (boundary) points of Delta_polar which should be used as rays of the underlying fan. It must include all vertices of Delta_polar and no repetitions are allowed. A string must be one of the following descriptions of points of Delta_polar:
  - "vertices" (default),
  - "all" (will not include the origin),
  - "all but facets" (will not include points in the relative interior of facets);
- charts – list of lists of elements from coordinate_points. Each of these lists must define a generating cone of a fan subdividing the normal fan of Delta. Default charts correspond to the normal fan of Delta without subdivision. The fan specified by charts will be subdivided to include all of the requested coordinate_points;
- coordinate_names – names of variables for the coordinate ring, see *normalize_names()* for acceptable formats. If not given, indexed variable names will be created automatically;

- `names` – an alias of `coordinate_names` for internal use. You may specify either `names` or `coordinate_names`, but not both;
- `coordinate_name_indices` – list of integers, indices for indexed variables. If not given, the index of each variable will coincide with the index of the corresponding point of `Delta_polar`;
- `make_simplicial` – if `True`, the underlying fan will be made simplicial (default: `False`);
- `base_ring` – base field of the CPR-Fano toric variety (default: $\mathbf{Q}$);
- `base_field` – alias for `base_ring`. Takes precedence if both are specified.
- `check` – by default the input data will be checked for correctness (e.g. that `charts` do form a subdivision of the normal fan of `Delta`). If you know for sure that the input is valid, you may significantly decrease construction time using `check=False` option.

OUTPUT: *CPR-Fano toric variety*.

EXAMPLES:

We start with the product of two projective lines:

```
sage: diamond = lattice_polytope.cross_polytope(2)
sage: diamond.vertices()
M( 1,  0),        M( 0,  1),
M(-1,  0),        M( 0, -1)
in 2-d lattice M
sage: P1xP1 = CPRFanoToricVariety(Delta_polar=diamond)
sage: P1xP1
2-d CPR-Fano toric variety covered by 4 affine patches
sage: P1xP1.fan()
Rational polyhedral fan in 2-d lattice M
sage: P1xP1.fan().rays()
M( 1,  0),        M( 0,  1),
M(-1,  0),        M( 0, -1)
in 2-d lattice M
```

"Unfortunately," this variety is smooth to start with and we cannot perform any subdivisions of the underlying fan without leaving the category of CPR-Fano toric varieties. Our next example starts with a square:

```
sage: square = diamond.polar()
sage: square.vertices()
N( 1,  1),        N( 1, -1),
N(-1, -1),        N(-1,  1)
in 2-d lattice N
sage: square.points()
N( 1,  1),        N( 1, -1),        N(-1, -1),
N(-1,  1),        N(-1,  0),        N( 0, -1),
N( 0,  0),        N( 0,  1),        N( 1,  0)
in 2-d lattice N
```

We will construct several varieties associated to it:

```
sage: FTV = CPRFanoToricVariety(Delta_polar=square)
sage: FTV.fan().rays()
N( 1,  1),        N( 1, -1),
N(-1, -1),        N(-1,  1)
in 2-d lattice N
```

(continues on next page)

(continued from previous page)

```
sage: FTV.gens()
(z0, z1, z2, z3)

sage: FTV = CPRFanoToricVariety(Delta_polar=square,
....:                           coordinate_points=[0,1,2,3,8])
sage: FTV.fan().rays()
N( 1,  1),        N( 1, -1),        N(-1, -1),
N(-1,  1),        N( 1,  0)
in 2-d lattice N
sage: FTV.gens()
(z0, z1, z2, z3, z8)

sage: FTV = CPRFanoToricVariety(Delta_polar=square,
....:                           coordinate_points=[8,0,2,1,3],
....:                           coordinate_names="x+")
sage: FTV.fan().rays()
N( 1,  0),        N( 1,  1),        N(-1, -1),
N( 1, -1),        N(-1,  1)
in 2-d lattice N
sage: FTV.gens()
(x8, x0, x2, x1, x3)

sage: FTV = CPRFanoToricVariety(Delta_polar=square,
....:                           coordinate_points="all",
....:                           coordinate_names="x y Z+")
sage: FTV.fan().rays()
N( 1,  1),        N( 1, -1),        N(-1, -1),        N(-1,  1),
N(-1,  0),        N( 0, -1),        N( 0,  1),        N( 1,  0)
in 2-d lattice N
sage: FTV.gens()
(x, y, Z2, Z3, Z4, Z5, Z7, Z8)
```

Note that Z6 is "missing". This is due to the fact that the 6-th point of `square` is the origin, and all automatically created names have the same indices as corresponding points of *Delta_polar()*. This is usually very convenient, especially if you have to work with several partial resolutions of the same Fano toric variety. However, you can change it, if you want:

```
sage: FTV = CPRFanoToricVariety(Delta_polar=square,
....:                           coordinate_points="all",
....:                           coordinate_names="x y Z+",
....:                           coordinate_name_indices=list(range(8)))
sage: FTV.gens()
(x, y, Z2, Z3, Z4, Z5, Z6, Z7)
```

Note that you have to provide indices for *all* variables, including those that have "completely custom" names. Again, this is usually convenient, because you can add or remove "custom" variables without disturbing too much "automatic" ones:

```
sage: FTV = CPRFanoToricVariety(Delta_polar=square,
....:                           coordinate_points="all",
....:                           coordinate_names="x Z+",
....:                           coordinate_name_indices=list(range(8)))
```

(continues on next page)

(continued from previous page)

```
sage: FTV.gens()
(x, Z1, Z2, Z3, Z4, Z5, Z6, Z7)
```

If you prefer to always start from zero, you will have to shift indices accordingly:

```
sage: FTV = CPRFanoToricVariety(Delta_polar=square,
....:                           coordinate_points="all",
....:                           coordinate_names="x Z+",
....:                           coordinate_name_indices=[0] + list(range(7)))
sage: FTV.gens()
(x, Z0, Z1, Z2, Z3, Z4, Z5, Z6)

sage: FTV = CPRFanoToricVariety(Delta_polar=square,
....:                           coordinate_points="all",
....:                           coordinate_names="x y Z+",
....:                           coordinate_name_indices=[0]*2 + list(range(6)))
sage: FTV.gens()
(x, y, Z0, Z1, Z2, Z3, Z4, Z5)
```

So you always can get any names you want, somewhat complicated default behaviour was designed with the hope that in most cases you will have no desire to provide different names.

Now we will use the possibility to specify initial charts:

```
sage: charts = [(0,1), (1,2), (2,3), (3,0)]
```

(these charts actually form exactly the face fan of our square)

```
sage: FTV = CPRFanoToricVariety(Delta_polar=square,
....:                           coordinate_points=[0,1,2,3,4],
....:                           charts=charts)
sage: FTV.fan().rays()
N( 1,  1),        N( 1, -1),        N(-1, -1),
N(-1,  1),        N(-1,  0)
in 2-d lattice N
sage: [cone.ambient_ray_indices() for cone in FTV.fan()]
[(0, 1), (1, 2), (2, 4), (3, 4), (0, 3)]
```

If charts are wrong, it should be detected:

```
sage: bad_charts = charts + [(3,0)]
sage: FTV = CPRFanoToricVariety(Delta_polar=square,
....:                           coordinate_points=[0,1,2,3,4],
....:                           charts=bad_charts)
Traceback (most recent call last):
...
ValueError: you have provided 5 cones, but only 4 of them are maximal!
Use discard_faces=True if you indeed need to construct a fan from these cones.
```

These charts are technically correct, they just happened to list one of them twice, but it is assumed that such a situation will not happen. It is especially important when you try to speed up your code:

```
sage: FTV = CPRFanoToricVariety(Delta_polar=square,
....:                           coordinate_points=[0,1,2,3,4],
```

(continues on next page)

(continued from previous page)

```
....:                             charts=bad_charts,
....:                             check=False)
Traceback (most recent call last):
...
IndexError: list assignment index out of range
```

In this case you still get an error message, but it is harder to figure out what is going on. It may also happen that "everything will still work" in the sense of not crashing, but work with such an invalid variety may lead to mathematically wrong results, so use `check=False` carefully!

Here are some other possible mistakes:

```
sage: bad_charts = charts + [(0,2)]
sage: FTV = CPRFanoToricVariety(Delta_polar=square,
....:                             coordinate_points=[0,1,2,3,4],
....:                             charts=bad_charts)
Traceback (most recent call last):
...
ValueError: (0, 2) does not form a chart of a subdivision of
the face fan of 2-d reflexive polytope #14 in 2-d lattice N!

sage: bad_charts = charts[:-1]
sage: FTV = CPRFanoToricVariety(Delta_polar=square,
....:                             coordinate_points=[0,1,2,3,4],
....:                             charts=bad_charts)
Traceback (most recent call last):
...
ValueError: given charts do not form a complete fan!

sage: FTV = CPRFanoToricVariety(Delta_polar=square,
....:                             coordinate_points=[1,2,3,4])
Traceback (most recent call last):
...
ValueError: all 4 vertices of Delta_polar must be used for coordinates!
Got: [1, 2, 3, 4]

sage: FTV = CPRFanoToricVariety(Delta_polar=square,
....:                             coordinate_points=[0,0,1,2,3,4])
Traceback (most recent call last):
...
ValueError: no repetitions are allowed for coordinate points!
Got: [0, 0, 1, 2, 3, 4]

sage: FTV = CPRFanoToricVariety(Delta_polar=square,
....:                             coordinate_points=[0,1,2,3,6])
Traceback (most recent call last):
...
ValueError: the origin (point #6) cannot be used for a coordinate!
Got: [0, 1, 2, 3, 6]
```

Here is a shorthand for defining the toric variety and homogeneous coordinates in one go:

```
sage: P1xP1.<a,b,c,d> = CPRFanoToricVariety(Delta_polar=diamond)
sage: (a^2+b^2) * (c+d)
a^2*c + b^2*c + a^2*d + b^2*d
```

class sage.schemes.toric.fano_variety.**CPRFanoToricVariety_field**(*Delta_polar*, *fan*, *coordinate_points*, *point_to_ray*, *coordinate_names*, *coordinate_name_indices*, *base_field*)

Bases: *ToricVariety_field*

Construct a CPR-Fano toric variety associated to a reflexive polytope.

Warning: This class does not perform any checks of correctness of input and it does assume that the internal structure of the given parameters is coordinated in a certain way. Use *CPRFanoToricVariety()* to construct CPR-Fano toric varieties.

Note: See documentation of the module *fano_variety* for the used definitions and supported varieties.

INPUT:

- `Delta_polar` – reflexive polytope;
- `fan` – rational polyhedral fan subdividing the face fan of `Delta_polar`;
- `coordinate_points` – list of indices of points of `Delta_polar` used for rays of `fan`;
- `point_to_ray` – dictionary mapping the index of a coordinate point to the index of the corresponding ray;
- `coordinate_names` – names of the variables of the coordinate ring in the format accepted by *normalize_names()*;
- `coordinate_name_indices` – indices for indexed variables, if `None`, will be equal to `coordinate_points`;
- `base_field` – base field of the CPR-Fano toric variety.

OUTPUT: *CPR-Fano toric variety*.

Delta()

Return the reflexive polytope associated to `self`.

OUTPUT:

- reflexive `lattice polytope`. The underlying fan of `self` is a coherent subdivision of the *normal fan* of this polytope.

EXAMPLES:

```
sage: diamond = lattice_polytope.cross_polytope(2)
sage: P1xP1 = CPRFanoToricVariety(Delta_polar=diamond)
sage: P1xP1.Delta()
2-d reflexive polytope #14 in 2-d lattice N
sage: P1xP1.Delta() is diamond.polar()
True
```

Delta_polar()

Return polar of *Delta()*.

OUTPUT:

- reflexive lattice polytope. The underlying fan of self is a coherent subdivision of the *face fan* of this polytope.

EXAMPLES:

```
sage: diamond = lattice_polytope.cross_polytope(2)
sage: P1xP1 = CPRFanoToricVariety(Delta_polar=diamond)
sage: P1xP1.Delta_polar()
2-d reflexive polytope #3 in 2-d lattice M
sage: P1xP1.Delta_polar() is diamond
True
sage: P1xP1.Delta_polar() is P1xP1.Delta().polar()
True
```

anticanonical_hypersurface(***kwds*)

Return an anticanonical hypersurface of self.

Note: The returned hypersurface may be actually a subscheme of **another** CPR-Fano toric variety: if the base field of self does not include all of the required names for generic monomial coefficients, it will be automatically extended.

Below Δ is the reflexive polytope corresponding to self, i.e. the fan of self is a refinement of the normal fan of Δ. This function accepts only keyword parameters.

INPUT:

- monomial_points – a list of integers or a string. A list will be interpreted as indices of points of Δ which should be used for monomials of this hypersurface. A string must be one of the following descriptions of points of Δ:
  - "vertices",
  - "vertices+origin",
  - "all",
  - "simplified" (default) – all points of Δ except for the interior points of facets, this choice corresponds to working with the "simplified polynomial moduli space" of anticanonical hypersurfaces;
- coefficient_names – names for the monomial coefficients, see *normalize_names()* for acceptable formats. If not given, indexed coefficient names will be created automatically;
- coefficient_name_indices – a list of integers, indices for indexed coefficients. If not given, the index of each coefficient will coincide with the index of the corresponding point of Δ;
- coefficients – as an alternative to specifying coefficient names and/or indices, you can give the coefficients themselves as arbitrary expressions and/or strings. Using strings allows you to easily add "parameters": the base field of self will be extended to include all necessary names.

OUTPUT:

- an *anticanonical hypersurface* of self (with the extended base field, if necessary).

EXAMPLES:

We realize the projective plane as a Fano toric variety:

```
sage: simplex = LatticePolytope([(1,0), (0,1), (-1,-1)])
sage: P2 = CPRFanoToricVariety(Delta_polar=simplex)
```

Its anticanonical "hypersurface" is a one-dimensional Calabi-Yau manifold:

```
sage: P2.anticanonical_hypersurface(monomial_points="all")
Closed subscheme of 2-d CPR-Fano toric variety
 covered by 3 affine patches defined by:
  a0*z0^3 + a9*z0^2*z1 + a7*z0*z1^2 + a1*z1^3 + a8*z0^2*z2 + a6*z0*z1*z2
  + a4*z1^2*z2 + a5*z0*z2^2 + a3*z1*z2^2 + a2*z2^3
```

In many cases it is sufficient to work with the "simplified polynomial moduli space" of anticanonical hypersurfaces:

```
sage: P2.anticanonical_hypersurface(monomial_points="simplified")
Closed subscheme of 2-d CPR-Fano toric variety
 covered by 3 affine patches defined by:
  a0*z0^3 + a1*z1^3 + a6*z0*z1*z2 + a2*z2^3
```

The mirror family to these hypersurfaces lives inside the Fano toric variety obtained using `simplex` as `Delta` instead of `Delta_polar`:

```
sage: FTV = CPRFanoToricVariety(Delta=simplex,
....:                           coordinate_points="all")
sage: FTV.anticanonical_hypersurface(monomial_points="simplified")
Closed subscheme of 2-d CPR-Fano toric variety
 covered by 9 affine patches defined by:
  a2*z2^3*z3^2*z4*z5^2*z8 + a1*z1^3*z3*z4^2*z7^2*z9
  + a3*z0*z1*z2*z3*z4*z5*z7*z8*z9 + a0*z0^3*z5*z7*z8^2*z9^2
```

Here we have taken the resolved version of the ambient space for the mirror family, but in fact we don't have to resolve singularities corresponding to the interior points of facets - they are singular points which do not lie on a generic anticanonical hypersurface:

```
sage: FTV = CPRFanoToricVariety(Delta=simplex,
....:                           coordinate_points="all but facets")
sage: FTV.anticanonical_hypersurface(monomial_points="simplified")
Closed subscheme of 2-d CPR-Fano toric variety
 covered by 3 affine patches defined by:
  a0*z0^3 + a1*z1^3 + a3*z0*z1*z2 + a2*z2^3
```

This looks very similar to our second anticanonical hypersurface of the projective plane, as expected, since all one-dimensional Calabi-Yau manifolds are elliptic curves!

All anticanonical hypersurfaces constructed above were generic with automatically generated coefficients. If you want, you can specify your own names

```
sage: FTV.anticanonical_hypersurface(coefficient_names="a b c d")
Closed subscheme of 2-d CPR-Fano toric variety
 covered by 3 affine patches defined by:
  a*z0^3 + b*z1^3 + d*z0*z1*z2 + c*z2^3
```

or give concrete coefficients

```
sage: FTV.anticanonical_hypersurface(coefficients=[1, 2, 3, 4])
Closed subscheme of 2-d CPR-Fano toric variety
 covered by 3 affine patches defined by:
  z0^3 + 2*z1^3 + 4*z0*z1*z2 + 3*z2^3
```

or even mix numerical coefficients with some expressions

```
sage: H = FTV.anticanonical_hypersurface(
....:   coefficients=[0, "t", "1/t", "psi/(psi^2 + phi)"])
sage: H
Closed subscheme of 2-d CPR-Fano toric variety
 covered by 3 affine patches defined by:
  t*z1^3 + psi/(phi + psi^2)*z0*z1*z2 + 1/t*z2^3
sage: R = H.ambient_space().base_ring()
sage: R
Fraction Field of
 Multivariate Polynomial Ring in phi, psi, t over Rational Field
```

cartesian_product(*other*, *coordinate_names=None*, *coordinate_indices=None*)

Return the Cartesian product of `self` with `other`.

INPUT:

- `other` – a (possibly *CPR-Fano*) *toric variety*;
- `coordinate_names` – names of variables for the coordinate ring, see *normalize_names()* for acceptable formats. If not given, indexed variable names will be created automatically;
- `coordinate_indices` – list of integers, indices for indexed variables. If not given, the index of each variable will coincide with the index of the corresponding ray of the fan.

OUTPUT:

- a *toric variety*, which is *CPR-Fano* if `other` was.

EXAMPLES:

```
sage: P1 = toric_varieties.P1()
sage: P2 = toric_varieties.P2()
sage: P1xP2 = P1.cartesian_product(P2); P1xP2
3-d CPR-Fano toric variety covered by 6 affine patches
sage: P1xP2.fan().rays()
N+N( 1,  0,  0),        N+N(-1,  0,  0),        N+N( 0,  1,  0),
N+N( 0,  0,  1),        N+N( 0, -1, -1)
in 3-d lattice N+N
sage: P1xP2.Delta_polar()
3-d reflexive polytope in 3-d lattice N+N
```

change_ring(*F*)

Return a CPR-Fano toric variety over field `F`, otherwise the same as `self`.

INPUT:

- `F` – field.

OUTPUT: *CPR-Fano toric variety* over `F`.

Note: There is no need to have any relation between `F` and the base field of `self`. If you do want to have such a relation, use `base_extend()` instead.

EXAMPLES:

```
sage: P1xP1 = toric_varieties.P1xP1()
sage: P1xP1.base_ring()
Rational Field
sage: P1xP1_RR = P1xP1.change_ring(RR)
sage: P1xP1_RR.base_ring()
Real Field with 53 bits of precision
sage: P1xP1_QQ = P1xP1_RR.change_ring(QQ)
sage: P1xP1_QQ.base_ring()
Rational Field
sage: P1xP1_RR.base_extend(QQ)
Traceback (most recent call last):
...
ValueError: no natural map from the base ring
(=Real Field with 53 bits of precision) to R (=Rational Field)!
sage: R = PolynomialRing(QQ, 2, 'a')
sage: P1xP1.change_ring(R)
Traceback (most recent call last):
...
TypeError: need a field to construct a Fano toric variety!
Got Multivariate Polynomial Ring in a0, a1 over Rational Field
```

coordinate_point_to_coordinate(*point*)

Return the variable of the coordinate ring corresponding to `point`.

INPUT:

- `point` – integer from the list of *coordinate_points()*.

OUTPUT:

- the corresponding generator of the coordinate ring of `self`.

EXAMPLES:

```
sage: diamond = lattice_polytope.cross_polytope(2)
sage: FTV = CPRFanoToricVariety(diamond, coordinate_points=[0,1,2,3,8])
sage: FTV.coordinate_points()
(0, 1, 2, 3, 8)
sage: FTV.gens()
(z0, z1, z2, z3, z8)
sage: FTV.coordinate_point_to_coordinate(8)
z8
```

coordinate_points()

Return indices of points of *Delta_polar()* used for coordinates.

OUTPUT: `tuple` of integers.

EXAMPLES:

```
sage: diamond = lattice_polytope.cross_polytope(2)
sage: square = diamond.polar()
sage: FTV = CPRFanoToricVariety(Delta_polar=square,
....:                           coordinate_points=[0,1,2,3,8])
sage: FTV.coordinate_points()
(0, 1, 2, 3, 8)
sage: FTV.gens()
(z0, z1, z2, z3, z8)

sage: FTV = CPRFanoToricVariety(Delta_polar=square,
....:                           coordinate_points="all")
sage: FTV.coordinate_points()
(0, 1, 2, 3, 4, 5, 7, 8)
sage: FTV.gens()
(z0, z1, z2, z3, z4, z5, z7, z8)
```

Note that one point is missing, namely

```
sage: square.origin()
6
```

nef_complete_intersection(*nef_partition*, ***kwds*)

Return a nef complete intersection in `self`.

Note: The returned complete intersection may be actually a subscheme of **another** CPR-Fano toric variety: if the base field of `self` does not include all of the required names for monomial coefficients, it will be automatically extended.

Below Δ is the reflexive polytope corresponding to `self`, i.e. the fan of `self` is a refinement of the normal fan of Δ. Other polytopes are described in the documentation of `nef-partitions` of `reflexive polytopes`.

Except for the first argument, `nef_partition`, this method accepts only keyword parameters.

INPUT:

- `nef_partition` – a k-part `nef-partition` of Δ°, all other parameters (if given) must be lists of length k;
- `monomial_points` – the i-th element of this list is either a list of integers or a string. A list will be interpreted as indices of points of Δ_i which should be used for monomials of the i-th polynomial of this complete intersection. A string must be one of the following descriptions of points of Δ_i:
  - "vertices",
  - "vertices+origin",
  - "all" (default),

  when using this description, it is also OK to pass a single string as `monomial_points` instead of repeating it k times;
- `coefficient_names` – the i-th element of this list specifies names for the monomial coefficients of the i-th polynomial, see *normalize_names()* for acceptable formats. If not given, indexed coefficient names will be created automatically;

- `coefficient_name_indices` – the i-th element of this list specifies indices for indexed coefficients of the i-th polynomial. If not given, the index of each coefficient will coincide with the index of the corresponding point of Δ_i;
- `coefficients` – as an alternative to specifying coefficient names and/or indices, you can give the coefficients themselves as arbitrary expressions and/or strings. Using strings allows you to easily add "parameters": the base field of `self` will be extended to include all necessary names.

OUTPUT:

- a *nef complete intersection* of `self` (with the extended base field, if necessary).

EXAMPLES:

We construct several complete intersections associated to the same nef-partition of the 3-dimensional reflexive polytope #2254:

```
sage: p = ReflexivePolytope(3, 2254)
sage: np = p.nef_partitions()[1]; np
Nef-partition {2, 3, 4, 7, 8} ⊔ {0, 1, 5, 6}
sage: X = CPRFanoToricVariety(Delta_polar=p)
sage: X.nef_complete_intersection(np)
Closed subscheme of 3-d CPR-Fano toric variety
 covered by 10 affine patches defined by:
  a0*z1*z4^2*z5^2*z7^3 + a2*z2*z4*z5*z6*z7^2*z8^2
  + a3*z2*z3*z4*z7*z8 + a1*z0*z2,
  b3*z1*z4*z5^2*z6^2*z7^2*z8^2 + b0*z2*z5*z6^3*z7*z8^4
  + b5*z1*z3*z4*z5*z6*z7*z8 + b2*z2*z3*z6^2*z8^3
  + b1*z1*z3^2*z4 + b4*z0*z1*z5*z6
```

Now we include only monomials associated to vertices of Δ_i:

```
sage: X.nef_complete_intersection(np, monomial_points="vertices")
Closed subscheme of 3-d CPR-Fano toric variety
 covered by 10 affine patches defined by:
  a0*z1*z4^2*z5^2*z7^3 + a2*z2*z4*z5*z6*z7^2*z8^2
  + a3*z2*z3*z4*z7*z8 + a1*z0*z2,
  b3*z1*z4*z5^2*z6^2*z7^2*z8^2 + b0*z2*z5*z6^3*z7*z8^4
  + b2*z2*z3*z6^2*z8^3 + b1*z1*z3^2*z4 + b4*z0*z1*z5*z6
```

(effectively, we set `b5=0`). Next we provide coefficients explicitly instead of using default generic names:

```
sage: X.nef_complete_intersection(np,
....:       monomial_points="vertices",
....:       coefficients=[("a", "a^2", "a/e", "c_i"), list(range(1,6))])
Closed subscheme of 3-d CPR-Fano toric variety
 covered by 10 affine patches defined by:
  a*z1*z4^2*z5^2*z7^3 + a/e*z2*z4*z5*z6*z7^2*z8^2
  + (c_i)*z2*z3*z4*z7*z8 + (a^2)*z0*z2,
  4*z1*z4*z5^2*z6^2*z7^2*z8^2 + z2*z5*z6^3*z7*z8^4
  + 3*z2*z3*z6^2*z8^3 + 2*z1*z3^2*z4 + 5*z0*z1*z5*z6
```

Finally, we take a look at the generic representative of these complete intersections in a completely resolved ambient toric variety:

```
sage: X = CPRFanoToricVariety(Delta_polar=p,
....:                         coordinate_points="all")
```

(continues on next page)

(continued from previous page)

```
sage: X.nef_complete_intersection(np)
Closed subscheme of 3-d CPR-Fano toric variety
 covered by 22 affine patches defined by:
  a2*z2*z4*z5*z6*z7^2*z8^2*z9^2*z10^2*z11*z12*z13
  + a0*z1*z4^2*z5^2*z7^3*z9*z10^2*z12*z13
  + a3*z2*z3*z4*z7*z8*z9*z10*z11*z12 + a1*z0*z2,
  b0*z2*z5*z6^3*z7*z8^4*z9^3*z10^2*z11^2*z12*z13^2
  + b3*z1*z4*z5^2*z6^2*z7^2*z8^2*z9^2*z10^2*z11*z12*z13^2
  + b2*z2*z3*z6^2*z8^3*z9^2*z10*z11^2*z12*z13
  + b5*z1*z3*z4*z5*z6*z7*z8*z9*z10*z11*z12*z13
  + b1*z1*z3^2*z4*z11*z12 + b4*z0*z1*z5*z6*z13
```

resolve(***kwds*)

Construct a toric variety whose fan subdivides the fan of `self`.

This function accepts only keyword arguments, none of which are mandatory.

INPUT:

- `new_points` – list of integers, indices of boundary points of *Delta_polar()*, which should be added as rays to the subdividing fan;
- all other arguments will be passed to *resolve()* method of (general) toric varieties, see its documentation for details.

OUTPUT:

- *CPR-Fano toric variety* if there was no `new_rays` argument and *toric variety* otherwise.

EXAMPLES:

```
sage: diamond = lattice_polytope.cross_polytope(2)
sage: FTV = CPRFanoToricVariety(Delta=diamond)
sage: FTV.coordinate_points()
(0, 1, 2, 3)
sage: FTV.gens()
(z0, z1, z2, z3)
sage: FTV_res = FTV.resolve(new_points=[6,8])
Traceback (most recent call last):
...
ValueError: the origin (point #6)
cannot be used for subdivision!
sage: FTV_res = FTV.resolve(new_points=[8,5]); FTV_res
2-d CPR-Fano toric variety covered by 6 affine patches
sage: FTV_res.coordinate_points()
(0, 1, 2, 3, 8, 5)
sage: FTV_res.gens()
(z0, z1, z2, z3, z8, z5)

sage: TV_res = FTV.resolve(new_rays=[(1,2)]); TV_res
2-d toric variety covered by 5 affine patches
sage: TV_res.gens()
(z0, z1, z2, z3, z4)
```

class sage.schemes.toric.fano_variety.**NefCompleteIntersection**(*P_Delta*, *nef_partition*, *monomial_points='all'*, *coefficient_names=None*, *coefficient_name_indices=None*, *coefficients=None*)

Bases: *AlgebraicScheme_subscheme_toric*

Construct a nef complete intersection in a CPR-Fano toric variety.

INPUT:

- `P_Delta` – a *CPR-Fano toric variety* associated to a reflexive polytope Δ;
- see *CPRFanoToricVariety_field.nef_complete_intersection()* for documentation on all other acceptable parameters.

OUTPUT:

- a *nef complete intersection* of `P_Delta` (with the extended base field, if necessary).

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: np = o.nef_partitions()[0]; np
Nef-partition {0, 1, 3} ⊔ {2, 4, 5}
sage: X = CPRFanoToricVariety(Delta_polar=o)
sage: X.nef_complete_intersection(np)
Closed subscheme of 3-d CPR-Fano toric variety
 covered by 8 affine patches defined by:
  a2*z0^2*z1 + a5*z0*z1*z3 + a1*z1*z3^2 + a3*z0^2*z4 + a4*z0*z3*z4 + a0*z3^2*z4,
  b1*z1*z2^2 + b2*z2^2*z4 + b5*z1*z2*z5 + b4*z2*z4*z5 + b3*z1*z5^2 + b0*z4*z5^2
```

See *CPRFanoToricVariety_field.nef_complete_intersection()* for a more elaborate example.

cohomology_class()

Return the class of `self` in the ambient space cohomology ring.

OUTPUT: A `cohomology class`.

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: np = o.nef_partitions()[0]; np
Nef-partition {0, 1, 3} ⊔ {2, 4, 5}
sage: X = CPRFanoToricVariety(Delta_polar=o)
sage: CI = X.nef_complete_intersection(np); CI
Closed subscheme of 3-d CPR-Fano toric variety
 covered by 8 affine patches defined by:
  a2*z0^2*z1 + a5*z0*z1*z3 + a1*z1*z3^2 + a3*z0^2*z4 + a4*z0*z3*z4 + a0*z3^2*z4,
  b1*z1*z2^2 + b2*z2^2*z4 + b5*z1*z2*z5 + b4*z2*z4*z5 + b3*z1*z5^2 + b0*z4*z5^2
sage: CI.cohomology_class()                                                    #␣
↪needs sage.libs.singular
[2*z3*z4 + 4*z3*z5 + 2*z4*z5]
```

nef_partition()

Return the nef-partition associated to `self`.

OUTPUT: A `nef-partition`.

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: np = o.nef_partitions()[0]; np
Nef-partition {0, 1, 3} ⊔ {2, 4, 5}
sage: X = CPRFanoToricVariety(Delta_polar=o)
sage: CI = X.nef_complete_intersection(np); CI
Closed subscheme of 3-d CPR-Fano toric variety
 covered by 8 affine patches defined by:
  a2*z0^2*z1 + a5*z0*z1*z3 + a1*z1*z3^2 + a3*z0^2*z4 + a4*z0*z3*z4 + a0*z3^2*z4,
  b1*z1*z2^2 + b2*z2^2*z4 + b5*z1*z2*z5 + b4*z2*z4*z5 + b3*z1*z5^2 + b0*z4*z5^2
sage: CI.nef_partition()
Nef-partition {0, 1, 3} ⊔ {2, 4, 5}
sage: CI.nef_partition() is np
True
```

sage.schemes.toric.fano_variety.**add_variables**(*field*, *variables*)

Extend `field` to include all `variables`.

INPUT:

- `field` - a field;
- `variables` - a list of strings.

OUTPUT:

- a fraction field extending the original `field`, which has all `variables` among its generators.

EXAMPLES:

We start with the rational field and slowly add more variables:

```
sage: from sage.schemes.toric.fano_variety import *
sage: F = add_variables(QQ, []); F      # No extension
Rational Field
sage: F = add_variables(QQ, ["a"]); F
Fraction Field of Univariate Polynomial Ring in a over Rational Field
sage: F = add_variables(F, ["a"]); F
Fraction Field of Univariate Polynomial Ring in a over Rational Field
sage: F = add_variables(F, ["b", "c"]); F
Fraction Field of Multivariate Polynomial Ring in a, b, c over Rational Field
sage: F = add_variables(F, ["c", "d", "b", "c", "d"]); F
Fraction Field of Multivariate Polynomial Ring in a, b, c, d over Rational Field
```

sage.schemes.toric.fano_variety.**is_CPRFanoToricVariety**(*x*)

Check if `x` is a CPR-Fano toric variety.

INPUT:

- `x` – anything.

OUTPUT:

- `True` if `x` is a *CPR-Fano toric variety* and `False` otherwise.

Note: While projective spaces are Fano toric varieties mathematically, they are not toric varieties in Sage due to efficiency considerations, so this function will return `False`.

EXAMPLES:

```
sage: from sage.schemes.toric.fano_variety import is_CPRFanoToricVariety
sage: is_CPRFanoToricVariety(1)
False
sage: FTV = toric_varieties.P2()
sage: FTV
2-d CPR-Fano toric variety covered by 3 affine patches
sage: is_CPRFanoToricVariety(FTV)
True
sage: is_CPRFanoToricVariety(ProjectiveSpace(2))
False
```

16.3 Library of toric varieties

This module provides a simple way to construct often-used toric varieties. Please see the help for the individual methods of `toric_varieties` for a more detailed description of which varieties can be constructed.

AUTHORS:

- Volker Braun (2010-07-02): initial version

EXAMPLES:

```
sage: toric_varieties.dP6()
2-d CPR-Fano toric variety covered by 6 affine patches
```

You can assign the homogeneous coordinates to Sage variables either with `inject_variables()` or immediately during assignment like this:

```
sage: P2.<x,y,z> = toric_varieties.P2()
sage: x^2 + y^2 + z^2
x^2 + y^2 + z^2
sage: P2.coordinate_ring()
Multivariate Polynomial Ring in x, y, z over Rational Field
```

class sage.schemes.toric.library.**ToricVarietyFactory**

Bases: SageObject

The methods of this class construct toric varieties.

Warning: You need not create instances of this class. Use the already-provided object `toric_varieties` instead.

A(*n*, *names='z+'*, *base_ring=Rational Field*)

Construct the `n`-dimensional affine space.

INPUT:

- `n` – positive integer. The dimension of the affine space.
- `names` – string. Names for the homogeneous coordinates. See *normalize_names()* for acceptable formats.
- `base_ring` – a ring (default: **Q**). The base ring for the toric variety.

OUTPUT: A *toric variety*.

EXAMPLES:

```
sage: A3 = toric_varieties.A(3); A3
3-d affine toric variety
sage: A3.fan().rays()
N(1, 0, 0),
N(0, 1, 0),
N(0, 0, 1)
in 3-d lattice N
sage: A3.gens()
(z0, z1, z2)
```

A1(*names='z'*, *base_ring=Rational Field*)

Construct the affine line $\mathbb{A}^1$ as a toric variety.

INPUT:

- `names` – string. Names for the homogeneous coordinates. See *normalize_names()* for acceptable formats.
- `base_ring` – a ring (default: $\mathbf{Q}$). The base ring for the toric variety.

OUTPUT: A *toric variety*.

EXAMPLES:

```
sage: A1 = toric_varieties.A1(); A1
1-d affine toric variety
sage: A1.fan().rays()
N(1)
in 1-d lattice N
sage: A1.gens()
(z,)
```

A2(*names='x y'*, *base_ring=Rational Field*)

Construct the affine plane $\mathbb{A}^2$ as a toric variety.

INPUT:

- `names` – string. Names for the homogeneous coordinates. See *normalize_names()* for acceptable formats.
- `base_ring` – a ring (default: $\mathbf{Q}$). The base ring for the toric variety.

OUTPUT: A *toric variety*.

EXAMPLES:

```
sage: A2 = toric_varieties.A2(); A2
2-d affine toric variety
sage: A2.fan().rays()
N(1, 0),
N(0, 1)
in 2-d lattice N
sage: A2.gens()
(x, y)
```

A2_Z2(*names='x y'*, *base_ring=Rational Field*)

Construct the orbifold $\mathbb{A}^2/\mathbf{Z}_2$ as a toric variety.

INPUT:

- `names` – string. Names for the homogeneous coordinates. See *normalize_names()* for acceptable formats.
- `base_ring` – a ring (default: $\mathbf{Q}$). The base ring for the toric variety.

OUTPUT: A *toric variety*.

EXAMPLES:

```
sage: A2_Z2 = toric_varieties.A2_Z2(); A2_Z2
2-d affine toric variety
sage: A2_Z2.fan().rays()
N(1, 0),
N(1, 2)
in 2-d lattice N
sage: A2_Z2.gens()
(x, y)
```

BCdlOG(*names='v1 v2 c1 c2 v4 v5 b e1 e2 e3 f g v6'*, *base_ring=Rational Field*)

Construct the 5-dimensional toric variety studied in [BCdlOG2000], [HLY2002]

INPUT:

- `names` – string. Names for the homogeneous coordinates. See *normalize_names()* for acceptable formats.
- `base_ring` – a ring (default: $\mathbf{Q}$). The base ring for the toric variety.

OUTPUT: A *CPR-Fano toric variety*.

EXAMPLES:

```
sage: X = toric_varieties.BCdlOG(); X
5-d CPR-Fano toric variety covered by 54 affine patches
sage: X.fan().rays()
N(-1,  0,  0,  2,  3),          N( 0, -1,  0,  2,  3),
N( 0,  0, -1,  2,  3),          N( 0,  0, -1,  1,  2),
N( 0,  0,  0, -1,  0),          N( 0,  0,  0,  0, -1),
N( 0,  0,  0,  2,  3),          N( 0,  0,  1,  2,  3),
N( 0,  0,  2,  2,  3),          N( 0,  0,  1,  1,  1),
N( 0,  1,  2,  2,  3),          N( 0,  1,  3,  2,  3),
N( 1,  0,  4,  2,  3)
in 5-d lattice N
sage: X.gens()
(v1, v2, c1, c2, v4, v5, b, e1, e2, e3, f, g, v6)
```

BCdlOG_base(*names='d4 d3 r2 r1 d2 u d1'*, *base_ring=Rational Field*)

Construct the base of the $\mathbb{P}^2(1,2,3)$ fibration *BCdlOG()*.

INPUT:

- `names` – string. Names for the homogeneous coordinates. See *normalize_names()* for acceptable formats.
- `base_ring` – a ring (default: $\mathbf{Q}$). The base ring for the toric variety.

OUTPUT: A *toric variety*.

EXAMPLES:

```
sage: base = toric_varieties.BCdlOG_base(); base
3-d toric variety covered by 10 affine patches
sage: base.fan().rays()
N(-1,  0,  0),    N( 0, -1,  0),    N( 0,  0, -1),    N( 0,  0,  1),
N( 0,  1,  2),    N( 0,  1,  3),    N( 1,  0,  4)
in 3-d lattice N
sage: base.gens()
(d4, d3, r2, r1, d2, u, d1)
```

Conifold(*names='u x y v'*, *base_ring=Rational Field*)

Construct the conifold as a toric variety.

INPUT:

- `names` – string. Names for the homogeneous coordinates. See *normalize_names()* for acceptable formats.
- `base_ring` – a ring (default: $\mathbf{Q}$). The base ring for the toric variety.

OUTPUT: A *toric variety*.

EXAMPLES:

```
sage: Conifold = toric_varieties.Conifold(); Conifold
3-d affine toric variety
sage: Conifold.fan().rays()
N(0, 0, 1),       N(0, 1, 1),
N(1, 0, 1),       N(1, 1, 1)
in 3-d lattice N
sage: Conifold.gens()
(u, x, y, v)
```

Cube_deformation(*k*, *names=None*, *base_ring=Rational Field*)

Construct, for each $k \in \mathbf{Z}_{\geq 0}$, a toric variety with $\mathbf{Z}_k$-torsion in the Chow group.

The fans of this sequence of toric varieties all equal the face fan of a unit cube topologically, but the `(1,1,1)`-vertex is moved to `(1,1,2k+1)`. This example was studied in [FS1994].

INPUT:

- `k` – integer. The case `k=0` is the same as *Cube_face_fan()*.
- `names` – string. Names for the homogeneous coordinates. See *normalize_names()* for acceptable formats.
- `base_ring` – a ring (default: $\mathbf{Q}$). The base ring for the toric variety.

OUTPUT:

A *toric variety* X_k. Its Chow group is $A_1(X_k) = \mathbf{Z}_k$.

EXAMPLES:

```
sage: X_2 = toric_varieties.Cube_deformation(2); X_2
3-d toric variety covered by 6 affine patches
sage: X_2.fan().rays()
```

(continues on next page)

(continued from previous page)

```
N( 1,  1,  5),    N( 1, -1,  1),    N(-1,  1,  1),    N(-1, -1,  1),
N(-1, -1, -1),    N(-1,  1, -1),    N( 1, -1, -1),    N( 1,  1, -1)
in 3-d lattice N
sage: X_2.gens()
(z0, z1, z2, z3, z4, z5, z6, z7)
```

Cube_face_fan(*names='z+'*, *base_ring=Rational Field*)

Construct the toric variety given by the face fan of the 3-dimensional unit lattice cube.

This variety has 6 conifold singularities but the fan is still polyhedral.

INPUT:

- `names` – string. Names for the homogeneous coordinates. See *normalize_names()* for acceptable formats.
- `base_ring` – a ring (default: $\mathbf{Q}$). The base ring for the toric variety.

OUTPUT: A *CPR-Fano toric variety*.

EXAMPLES:

```
sage: Cube_face_fan = toric_varieties.Cube_face_fan(); Cube_face_fan
3-d CPR-Fano toric variety covered by 6 affine patches
sage: Cube_face_fan.fan().rays()
N( 1,  1,  1),    N( 1, -1,  1),    N(-1,  1,  1),    N(-1, -1,  1),
N(-1, -1, -1),    N(-1,  1, -1),    N( 1, -1, -1),    N( 1,  1, -1)
in 3-d lattice N
sage: Cube_face_fan.gens()
(z0, z1, z2, z3, z4, z5, z6, z7)
```

Cube_nonpolyhedral(*names='z+'*, *base_ring=Rational Field*)

Construct the toric variety defined by a fan that is not the face fan of a polyhedron.

This toric variety is defined by a fan that is topologically like the face fan of a 3-dimensional cube, but with a different N-lattice structure.

INPUT:

- `names` – string. Names for the homogeneous coordinates. See *normalize_names()* for acceptable formats.
- `base_ring` – a ring (default: $\mathbf{Q}$). The base ring for the toric variety.

OUTPUT: A *toric variety*.

Note:

- This is an example of an non-polyhedral fan.
- Its Chow group has torsion: $A_2(X) = \mathbf{Z}^5 \oplus \mathbf{Z}_2$

EXAMPLES:

```
sage: Cube_nonpolyhedral = toric_varieties.Cube_nonpolyhedral()
sage: Cube_nonpolyhedral
3-d toric variety covered by 6 affine patches
```

(continues on next page)

(continued from previous page)

```
sage: Cube_nonpolyhedral.fan().rays()
N( 1,  2,  3),    N( 1, -1,  1),    N(-1,  1,  1),    N(-1, -1,  1),
N(-1, -1, -1),    N(-1,  1, -1),    N( 1, -1, -1),    N( 1,  1, -1)
in 3-d lattice N
sage: Cube_nonpolyhedral.gens()
(z0, z1, z2, z3, z4, z5, z6, z7)
```

Cube_sublattice(*names='z+', base_ring=Rational Field*)

Construct the toric variety defined by a face fan over a 3-dimensional cube, but not the unit cube in the N-lattice. See p. 65 of [Ful1993].

Its Chow group is $A_2(X) = \mathbf{Z}^5$, which distinguishes it from the face fan of the unit cube.

INPUT:

- `names` – string. Names for the homogeneous coordinates. See *normalize_names()* for acceptable formats.
- `base_ring` – a ring (default: $\mathbf{Q}$). The base ring for the toric variety.

OUTPUT: A *CPR-Fano toric variety*.

EXAMPLES:

```
sage: Cube_sublattice = toric_varieties.Cube_sublattice(); Cube_sublattice
3-d CPR-Fano toric variety covered by 6 affine patches
sage: Cube_sublattice.fan().rays()
N( 1,  0,  0),    N( 0,  1,  0),    N( 0,  0,  1),    N(-1,  1,  1),
N(-1,  0,  0),    N( 0, -1,  0),    N( 0,  0, -1),    N( 1, -1, -1)
in 3-d lattice N
sage: Cube_sublattice.gens()
(z0, z1, z2, z3, z4, z5, z6, z7)
```

P(*n*, *names='z+', base_ring=Rational Field*)

Construct the `n`-dimensional projective space $\mathbb{P}^n$.

INPUT:

- `n` – positive integer. The dimension of the projective space.
- `names` – string. Names for the homogeneous coordinates. See *normalize_names()* for acceptable formats.
- `base_ring` – a ring (default: $\mathbf{Q}$). The base ring for the toric variety.

OUTPUT: A *CPR-Fano toric variety*.

EXAMPLES:

```
sage: P3 = toric_varieties.P(3); P3
3-d CPR-Fano toric variety covered by 4 affine patches
sage: P3.fan().rays()
N( 1,  0,  0),
N( 0,  1,  0),
N( 0,  0,  1),
N(-1, -1, -1)
in 3-d lattice N
```

(continues on next page)

(continued from previous page)

```
sage: P3.gens()
(z0, z1, z2, z3)
```

P1(*names='s t'*, *base_ring=Rational Field*)

Construct the projective line $\mathbb{P}^1$ as a toric variety.

INPUT:

- `names` – string. Names for the homogeneous coordinates. See *normalize_names()* for acceptable formats.
- `base_ring` – a ring (default: $\mathbf{Q}$). The base ring for the toric variety.

OUTPUT: A *CPR-Fano toric variety*.

EXAMPLES:

```
sage: P1 = toric_varieties.P1(); P1
1-d CPR-Fano toric variety covered by 2 affine patches
sage: P1.fan().rays()
N( 1),
N(-1)
in 1-d lattice N
sage: P1.gens()
(s, t)
```

P1xA1(*names='s t z'*, *base_ring=Rational Field*)

Construct the Cartesian product $\mathbb{P}^1 \times \mathbb{A}^1$ as a toric variety.

INPUT:

- `names` – string. Names for the homogeneous coordinates. See *normalize_names()* for acceptable formats.
- `base_ring` – a ring (default: $\mathbf{Q}$). The base ring for the toric variety.

OUTPUT: A *toric variety*.

EXAMPLES:

```
sage: P1xA1 = toric_varieties.P1xA1(); P1xA1
2-d toric variety covered by 2 affine patches
sage: P1xA1.fan().rays()
N( 1, 0),
N(-1, 0),
N( 0, 1)
in 2-d lattice N
sage: P1xA1.gens()
(s, t, z)
```

P1xP1(*names='s t x y'*, *base_ring=Rational Field*)

Construct the del Pezzo surface $\mathbb{P}^1 \times \mathbb{P}^1$ as a toric variety.

INPUT:

- `names` – string. Names for the homogeneous coordinates. See *normalize_names()* for acceptable formats.
- `base_ring` – a ring (default: $\mathbf{Q}$). The base ring for the toric variety.

OUTPUT: A *CPR-Fano toric variety*.

EXAMPLES:

```
sage: P1xP1 = toric_varieties.P1xP1(); P1xP1
2-d CPR-Fano toric variety covered by 4 affine patches
sage: P1xP1.fan().rays()
N( 1,  0),        N(-1,  0),
N( 0,  1),        N( 0, -1)
in 2-d lattice N
sage: P1xP1.gens()
(s, t, x, y)
```

P1xP1_Z2(*names='s t x y'*, *base_ring=Rational Field*)

Construct the toric $\mathbf{Z}_2$-orbifold of the del Pezzo surface $\mathbb{P}^1 \times \mathbb{P}^1$ as a toric variety.

INPUT:

- `names` – string. Names for the homogeneous coordinates. See *normalize_names()* for acceptable formats.
- `base_ring` – a ring (default: $\mathbf{Q}$). The base ring for the toric variety.

OUTPUT: A *CPR-Fano toric variety*.

EXAMPLES:

```
sage: P1xP1_Z2 = toric_varieties.P1xP1_Z2(); P1xP1_Z2
2-d CPR-Fano toric variety covered by 4 affine patches
sage: P1xP1_Z2.fan().rays()
N( 1,  1),        N(-1, -1),
N(-1,  1),        N( 1, -1)
in 2-d lattice N
sage: P1xP1_Z2.gens()
(s, t, x, y)
sage: P1xP1_Z2.Chow_group().degree(1)
C2 x Z^2
```

P2(*names='x y z'*, *base_ring=Rational Field*)

Construct the projective plane $\mathbb{P}^2$ as a toric variety.

INPUT:

- `names` – string. Names for the homogeneous coordinates. See *normalize_names()* for acceptable formats.
- `base_ring` – a ring (default: $\mathbf{Q}$). The base ring for the toric variety.

OUTPUT: A *CPR-Fano toric variety*.

EXAMPLES:

```
sage: P2 = toric_varieties.P2(); P2
2-d CPR-Fano toric variety covered by 3 affine patches
sage: P2.fan().rays()
N( 1,  0),
N( 0,  1),
N(-1, -1)
in 2-d lattice N
```

(continues on next page)

(continued from previous page)

```
sage: P2.gens()
(x, y, z)
```

P2_112(*names='z+', base_ring=Rational Field*)

Construct the weighted projective space $\mathbb{P}^2(1, 1, 2)$.

INPUT:

- `names` – string. Names for the homogeneous coordinates. See *normalize_names()* for acceptable formats.
- `base_ring` – a ring (default: $\mathbf{Q}$). The base ring for the toric variety.

OUTPUT: A *CPR-Fano toric variety*.

EXAMPLES:

```
sage: P2_112 = toric_varieties.P2_112(); P2_112
2-d CPR-Fano toric variety covered by 3 affine patches
sage: P2_112.fan().rays()
N( 1,  0),
N( 0,  1),
N(-1, -2)
in 2-d lattice N
sage: P2_112.gens()
(z0, z1, z2)
```

P2_123(*names='z+', base_ring=Rational Field*)

Construct the weighted projective space $\mathbb{P}^2(1, 2, 3)$.

INPUT:

- `names` – string. Names for the homogeneous coordinates. See *normalize_names()* for acceptable formats.
- `base_ring` – a ring (default: $\mathbf{Q}$). The base ring for the toric variety.

OUTPUT: A *CPR-Fano toric variety*.

EXAMPLES:

```
sage: P2_123 = toric_varieties.P2_123(); P2_123
2-d CPR-Fano toric variety covered by 3 affine patches
sage: P2_123.fan().rays()
N( 1,  0),
N( 0,  1),
N(-2, -3)
in 2-d lattice N
sage: P2_123.gens()
(z0, z1, z2)
```

P4_11133(*names='z+', base_ring=Rational Field*)

Construct the weighted projective space $\mathbb{P}^4(1, 1, 1, 3, 3)$.

INPUT:

- `names` – string. Names for the homogeneous coordinates. See *normalize_names()* for acceptable formats.

- base_ring – a ring (default: $\mathbf{Q}$). The base ring for the toric variety.

OUTPUT: A *CPR-Fano toric variety*.

EXAMPLES:

```
sage: P4_11133 = toric_varieties.P4_11133(); P4_11133
4-d CPR-Fano toric variety covered by 5 affine patches
sage: P4_11133.fan().rays()
N( 1,  0,  0,  0),      N( 0,  1,  0,  0),      N( 0,  0,  1,  0),
N( 0,  0,  0,  1),      N(-3, -3, -1, -1)
in 4-d lattice N
sage: P4_11133.gens()
(z0, z1, z2, z3, z4)
```

P4_11133_resolved(*names='z+', base_ring=Rational Field*)

Construct the weighted projective space $\mathbb{P}^4(1,1,1,3,3)$.

INPUT:

- names – string. Names for the homogeneous coordinates. See *normalize_names()* for acceptable formats.
- base_ring – a ring (default: $\mathbf{Q}$). The base ring for the toric variety.

OUTPUT: A *CPR-Fano toric variety*.

EXAMPLES:

```
sage: P4_11133_resolved = toric_varieties.P4_11133_resolved()
sage: P4_11133_resolved
4-d CPR-Fano toric variety covered by 9 affine patches
sage: P4_11133_resolved.fan().rays()
N( 1,  0,  0,  0),      N( 0,  1,  0,  0),      N( 0,  0,  1,  0),
N( 0,  0,  0,  1),      N(-3, -3, -1, -1),      N(-1, -1,  0,  0)
in 4-d lattice N
sage: P4_11133_resolved.gens()
(z0, z1, z2, z3, z4, z5)
```

P4_11169(*names='z+', base_ring=Rational Field*)

Construct the weighted projective space $\mathbb{P}^4(1,1,1,6,9)$.

INPUT:

- names – string. Names for the homogeneous coordinates. See *normalize_names()* for acceptable formats.
- base_ring – a ring (default: $\mathbf{Q}$). The base ring for the toric variety.

OUTPUT: A *CPR-Fano toric variety*.

EXAMPLES:

```
sage: P4_11169 = toric_varieties.P4_11169(); P4_11169
4-d CPR-Fano toric variety covered by 5 affine patches
sage: P4_11169.fan().rays()
N( 1,  0,  0,  0),      N( 0,  1,  0,  0),      N( 0,  0,  1,  0),
N( 0,  0,  0,  1),      N(-9, -6, -1, -1)
in 4-d lattice N
```

(continues on next page)

(continued from previous page)

```
sage: P4_11169.gens()
(z0, z1, z2, z3, z4)
```

P4_11169_resolved(*names='z+'*, *base_ring=Rational Field*)

Construct the blow-up of the weighted projective space $\mathbb{P}^4(1,1,1,6,9)$ at its curve of $\mathbf{Z}_3$ orbifold fixed points.

INPUT:

- `names` – string. Names for the homogeneous coordinates. See *normalize_names()* for acceptable formats.
- `base_ring` – a ring (default: $\mathbf{Q}$). The base ring for the toric variety.

OUTPUT: A *CPR-Fano toric variety*.

EXAMPLES:

```
sage: P4_11169_resolved = toric_varieties.P4_11169_resolved()
sage: P4_11169_resolved
4-d CPR-Fano toric variety covered by 9 affine patches
sage: P4_11169_resolved.fan().rays()
N( 1,  0,  0,  0),      N( 0,  1,  0,  0),      N( 0,  0,  1,  0),
N( 0,  0,  0,  1),      N(-9, -6, -1, -1),      N(-3, -2,  0,  0)
in 4-d lattice N
sage: P4_11169_resolved.gens()
(z0, z1, z2, z3, z4, z5)
```

WP(**q*, ***kw*)

Construct weighted projective n-space over a field.

INPUT:

- `q` – a sequence of positive integers relatively prime to one another. The weights `q` can be given either as a list or tuple, or as positional arguments.

Two keyword arguments:

- `base_ring` – a field (default: $\mathbf{Q}$).
- `names` – string or list (tuple) of strings (default 'z+'). See *normalize_names()* for acceptable formats.

OUTPUT:

- A *toric variety*. If $q = (q_0, \ldots, q_n)$, then the output is the weighted projective space $\mathbb{P}(q_0, \ldots, q_n)$ over `base_ring`. `names` are the names of the generators of the homogeneous coordinate ring.

EXAMPLES:

A hyperelliptic curve C of genus 2 as a subscheme of the weighted projective plane $\mathbb{P}(1,3,1)$:

```
sage: X = toric_varieties.WP([1,3,1], names='x y z')
sage: X.inject_variables()
Defining x, y, z
sage: g = y^2 - (x^6-z^6)
sage: C = X.subscheme([g]); C
Closed subscheme of 2-d toric variety covered by 3 affine patches defined by:
  -x^6 + z^6 + y^2
```

dP6(*names='x u y v z w'*, *base_ring=Rational Field*)

Construct the del Pezzo surface of degree 6 ($\mathbb{P}^2$ blown up at 3 points) as a toric variety.

INPUT:

- `names` – string. Names for the homogeneous coordinates. See *normalize_names()* for acceptable formats.
- `base_ring` – a ring (default: $\mathbf{Q}$). The base ring for the toric variety.

OUTPUT: A *CPR-Fano toric variety*.

EXAMPLES:

```
sage: dP6 = toric_varieties.dP6(); dP6
2-d CPR-Fano toric variety covered by 6 affine patches
sage: dP6.fan().rays()
N( 0,  1),        N(-1,  0),        N(-1, -1),
N( 0, -1),        N( 1,  0),        N( 1,  1)
in 2-d lattice N
sage: dP6.gens()
(x, u, y, v, z, w)
```

dP6xdP6(*names='x0 x1 x2 x3 x4 x5 y0 y1 y2 y3 y4 y5'*, *base_ring=Rational Field*)

Construct the product of two del Pezzo surfaces of degree 6 ($\mathbb{P}^2$ blown up at 3 points) as a toric variety.

INPUT:

- `names` – string. Names for the homogeneous coordinates. See *normalize_names()* for acceptable formats.
- `base_ring` – a ring (default: $\mathbf{Q}$). The base ring for the toric variety.

OUTPUT: A *CPR-Fano toric variety*.

EXAMPLES:

```
sage: dP6xdP6 = toric_varieties.dP6xdP6(); dP6xdP6
4-d CPR-Fano toric variety covered by 36 affine patches
sage: dP6xdP6.fan().rays()
N( 0,  1,  0,  0),      N(-1,  0,  0,  0),      N(-1, -1,  0,  0),
N( 0, -1,  0,  0),      N( 1,  0,  0,  0),      N( 1,  1,  0,  0),
N( 0,  0,  0,  1),      N( 0,  0, -1,  0),      N( 0,  0, -1, -1),
N( 0,  0,  0, -1),      N( 0,  0,  1,  0),      N( 0,  0,  1,  1)
in 4-d lattice N
sage: dP6xdP6.gens()
(x0, x1, x2, x3, x4, x5, y0, y1, y2, y3, y4, y5)
```

dP7(*names='x u y v z'*, *base_ring=Rational Field*)

Construct the del Pezzo surface of degree 7 ($\mathbb{P}^2$ blown up at 2 points) as a toric variety.

INPUT:

- `names` – string. Names for the homogeneous coordinates. See *normalize_names()* for acceptable formats.
- `base_ring` – a ring (default: $\mathbf{Q}$). The base ring for the toric variety.

OUTPUT: A *CPR-Fano toric variety*.

EXAMPLES:

```
sage: dP7 = toric_varieties.dP7(); dP7
2-d CPR-Fano toric variety covered by 5 affine patches
sage: dP7.fan().rays()
N( 0,  1),        N(-1,  0),        N(-1, -1),
N( 0, -1),        N( 1,  0)
in 2-d lattice N
sage: dP7.gens()
(x, u, y, v, z)
```

dP8(*names='t x y z'*, *base_ring=Rational Field*)

Construct the del Pezzo surface of degree 8 ($\mathbb{P}^2$ blown up at 1 point) as a toric variety.

INPUT:

- `names` – string. Names for the homogeneous coordinates. See *normalize_names()* for acceptable formats.
- `base_ring` – a ring (default: $\mathbf{Q}$). The base ring for the toric variety.

OUTPUT: A *CPR-Fano toric variety*.

EXAMPLES:

```
sage: dP8 = toric_varieties.dP8(); dP8
2-d CPR-Fano toric variety covered by 4 affine patches
sage: dP8.fan().rays()
N( 1,  1),        N( 0,  1),
N(-1, -1),        N( 1,  0)
in 2-d lattice N
sage: dP8.gens()
(t, x, y, z)
```

torus(*n*, *names='z+'*, *base_ring=Rational Field*)

Construct the n-dimensional algebraic torus $(\mathbb{F}^\times)^n$.

INPUT:

- `n` – non-negative integer. The dimension of the algebraic torus.
- `names` – string. Names for the homogeneous coordinates. See *normalize_names()* for acceptable formats.
- `base_ring` – a ring (default: $\mathbf{Q}$). The base ring for the toric variety.

OUTPUT: A *toric variety*.

EXAMPLES:

```
sage: T3 = toric_varieties.torus(3);  T3
3-d affine toric variety
sage: T3.fan().rays()
Empty collection
in 3-d lattice N
sage: T3.fan().virtual_rays()
N(1, 0, 0),
N(0, 1, 0),
N(0, 0, 1)
in 3-d lattice N
```

(continues on next page)

(continued from previous page)

```
sage: T3.gens()
(z0, z1, z2)
sage: sorted(T3.change_ring(GF(3)).point_set().list())
[[1 : 1 : 1], [1 : 1 : 2], [1 : 2 : 1], [1 : 2 : 2],
 [2 : 1 : 1], [2 : 1 : 2], [2 : 2 : 1], [2 : 2 : 2]]
```

16.4 Toric divisors and divisor classes

Let X be a *toric variety* corresponding to a rational polyhedral fan Σ. A *toric divisor* D is a T-Weil divisor over a given coefficient ring (usually $\mathbf{Z}$ or $\mathbf{Q}$), i.e. a formal linear combination of torus-invariant subvarieties of X of codimension one. In homogeneous coordinates $[z_0 : \cdots : z_k]$, these are the subvarieties $\{z_i = 0\}$. Note that there is a finite number of such subvarieties, one for each ray of Σ. We generally identify

- Toric divisor D,
- Sheaf $\mathcal{O}(D)$ (if D is Cartier, it is a line bundle),
- Support function ϕ_D (if D is $\mathbf{Q}$-Cartier, it is a function linear on each cone of Σ).

EXAMPLES:

We start with an illustration of basic divisor arithmetic:

```
sage: dP6 = toric_varieties.dP6()
sage: Dx,Du,Dy,Dv,Dz,Dw = dP6.toric_divisor_group().gens()
sage: Dx
V(x)
sage: -Dx
-V(x)
sage: 2*Dx
2*V(x)
sage: Dx*2
2*V(x)
sage: (1/2)*Dx + Dy/3 - Dz
1/2*V(x) + 1/3*V(y) - V(z)
sage: Dx.parent()
Group of toric ZZ-Weil divisors
on 2-d CPR-Fano toric variety covered by 6 affine patches
sage: (Dx/2).parent()
Group of toric QQ-Weil divisors
on 2-d CPR-Fano toric variety covered by 6 affine patches
```

Now we create a more complicated variety to demonstrate divisors of different types:

```
sage: F = Fan(cones=[(0,1,2,3), (0,1,4)],
....:         rays=[(1,1,1), (1,-1,1), (1,-1,-1), (1,1,-1), (0,0,1)])
sage: X = ToricVariety(F)
sage: QQ_Cartier = X.divisor([2,2,1,1,1])
sage: Cartier = 2 * QQ_Cartier
sage: Weil = X.divisor([1,1,1,0,0])
sage: QQ_Weil = 1/2 * Weil
sage: [QQ_Weil.is_QQ_Weil(),
....:  QQ_Weil.is_Weil(),
```

(continues on next page)

(continued from previous page)

```
....:  QQ_Weil.is_QQ_Cartier(),
....:  QQ_Weil.is_Cartier()]
[True, False, False, False]
sage: [Weil.is_QQ_Weil(),
....:  Weil.is_Weil(),
....:  Weil.is_QQ_Cartier(),
....:  Weil.is_Cartier()]
[True, True, False, False]
sage: [QQ_Cartier.is_QQ_Weil(),
....:  QQ_Cartier.is_Weil(),
....:  QQ_Cartier.is_QQ_Cartier(),
....:  QQ_Cartier.is_Cartier()]
[True, True, True, False]
sage: [Cartier.is_QQ_Weil(),
....:  Cartier.is_Weil(),
....:  Cartier.is_QQ_Cartier(),
....:  Cartier.is_Cartier()]
[True, True, True, True]
```

The toric (**Q**-Weil) divisors on a toric variety X modulo linear equivalence generate the divisor **class group** $\mathrm{Cl}(X)$, implemented by *ToricRationalDivisorClassGroup*. If X is smooth, this equals the **Picard group** $\mathrm{Pic}(X)$. We continue using del Pezzo surface of degree 6 introduced above:

```
sage: Cl = dP6.rational_class_group(); Cl
The toric rational divisor class group
of a 2-d CPR-Fano toric variety covered by 6 affine patches
sage: Cl.ngens()
4
sage: c0,c1,c2,c3 = Cl.gens()
sage: c = c0 + 2*c1 - c3; c
Divisor class [1, 2, 0, -1]
```

Divisors are mapped to their classes and lifted via:

```
sage: Dx.divisor_class()
Divisor class [1, 0, 0, 0]
sage: Dx.divisor_class() in Cl
True
sage: (-Dw + Dv + Dy).divisor_class()
Divisor class [1, 0, 0, 0]
sage: c0
Divisor class [1, 0, 0, 0]
sage: c0.lift()
V(x)
```

The (rational) divisor class group is where the Kaehler cone lives:

```
sage: Kc = dP6.Kaehler_cone(); Kc
4-d cone in 4-d lattice
sage: Kc.rays()
Divisor class [0, 1, 1, 0],
Divisor class [0, 0, 1, 1],
Divisor class [1, 1, 0, 0],
```

(continues on next page)

(continued from previous page)

```
Divisor class [1, 1, 1, 0],
Divisor class [0, 1, 1, 1]
in Basis lattice of The toric rational divisor class group
of a 2-d CPR-Fano toric variety covered by 6 affine patches
sage: Kc.ray(1).lift()
V(x) + V(w)
```

Given a divisor D, we have an associated line bundle (or a reflexive sheaf, if D is not Cartier) $\mathcal{O}(D)$. Its sections are:

```
sage: P2 = toric_varieties.P2()
sage: H = P2.divisor(0); H
V(x)
sage: H.sections()
(M(-1, 0), M(-1, 1), M(0, 0))
sage: H.sections_monomials()
(z, y, x)
```

Note that the space of sections is always spanned by monomials. Therefore, we can grade the sections (as homogeneous monomials) by their weight under rescaling individual coordinates. This weight data amounts to a point of the dual lattice.

In the same way, we can grade cohomology groups by their cohomological degree and a weight:

```
sage: M = P2.fan().lattice().dual()
sage: H.cohomology(deg=0, weight=M(-1,0))
Vector space of dimension 1 over Rational Field
sage: _.dimension()
1
```

Here is a more complicated example with $h^1(dP_6, \mathcal{O}(D)) = 4$

```
sage: D = dP6.divisor([0, 0, -1, 0, 2, -1])
sage: D.cohomology()
{0: Vector space of dimension 0 over Rational Field,
 1: Vector space of dimension 4 over Rational Field,
 2: Vector space of dimension 0 over Rational Field}
sage: D.cohomology(dim=True)
(0, 4, 0)
```

AUTHORS:

- Volker Braun, Andrey Novoseltsev (2010-09-07): initial version.

sage.schemes.toric.divisor.**ToricDivisor**(*toric_variety*, *arg=None*, *ring=None*, *check=True*, *reduce=True*)

Construct a divisor of `toric_variety`.

INPUT:

- `toric_variety` – a *toric variety*;
- `arg` – one of the following description of the toric divisor to be constructed:
  - `None` or 0 (the trivial divisor);
  - monomial in the homogeneous coordinates;
  - one-dimensional cone of the fan of `toric_variety` or a lattice point generating such a cone;

  - sequence of rational numbers, specifying multiplicities for each of the toric divisors.

- `ring` – usually either **Z** or **Q**. The base ring of the divisor group. If `ring` is not specified, a coefficient ring suitable for `arg` is derived.
- `check` – bool (default: True). Whether to coerce coefficients into base ring. Setting it to `False` can speed up construction.
- `reduce` – reduce (default: True). Whether to combine common terms. Setting it to `False` can speed up construction.

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 valid input data `arg`.

OUTPUT: A *sage.schemes.toric.divisor.ToricDivisor_generic*.

EXAMPLES:

```
sage: from sage.schemes.toric.divisor import ToricDivisor
sage: dP6 = toric_varieties.dP6()
sage: ToricDivisor(dP6, [(1,dP6.gen(2)), (1,dP6.gen(1))])
V(u) + V(y)
sage: ToricDivisor(dP6, (0,1,1,0,0,0), ring=QQ)
V(u) + V(y)
sage: dP6.inject_variables()
Defining x, u, y, v, z, w
sage: ToricDivisor(dP6, u + y)
Traceback (most recent call last):
...
ValueError: u + y is not a monomial
sage: ToricDivisor(dP6, u*y)
V(u) + V(y)
sage: ToricDivisor(dP6, dP6.fan(dim=1)[2] )
V(y)
sage: cone = Cone(dP6.fan(dim=1)[2])
sage: ToricDivisor(dP6, cone)
V(y)
sage: N = dP6.fan().lattice()
sage: ToricDivisor(dP6, N(1,1) )
V(w)
```

We attempt to guess the correct base ring:

```
sage: ToricDivisor(dP6, [(1/2,u)])
1/2*V(u)
sage: _.parent()
Group of toric QQ-Weil divisors on
2-d CPR-Fano toric variety covered by 6 affine patches
sage: ToricDivisor(dP6, [(1/2,u), (1/2,u)])
V(u)
sage: _.parent()
Group of toric ZZ-Weil divisors on
2-d CPR-Fano toric variety covered by 6 affine patches
sage: ToricDivisor(dP6, [(u,u)])
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
TypeError: cannot deduce coefficient ring for [(u, u)]
```

class sage.schemes.toric.divisor.**ToricDivisorGroup**(*toric_variety*, *base_ring*)

Bases: *DivisorGroup_generic*

The group of (Q-T-Weil) divisors on a toric variety.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: P2.toric_divisor_group()
Group of toric ZZ-Weil divisors
on 2-d CPR-Fano toric variety covered by 3 affine patches
```

Element

alias of *ToricDivisor_generic*

base_extend(*R*)

Extend the scalars of `self` to `R`.

INPUT:

- `R` – ring.

OUTPUT:

- toric divisor group.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: DivZZ = P2.toric_divisor_group()
sage: DivQQ = P2.toric_divisor_group(base_ring=QQ)
sage: DivZZ.base_extend(QQ) is DivQQ
True
```

gen(*i*)

Return the `i`-th generator of the divisor group.

INPUT:

- `i` – integer.

OUTPUT:

The divisor $z_i = 0$, where z_i is the i-th homogeneous coordinate.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: TDiv = P2.toric_divisor_group()
sage: TDiv.gen(2)
V(z)
```

gens()

Return the generators of the divisor group.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: TDiv = P2.toric_divisor_group()
sage: TDiv.gens()
(V(x), V(y), V(z))
```

ngens()

Return the number of generators.

OUTPUT:

The number of generators of `self`, which equals the number of rays in the fan of the toric variety.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: TDiv = P2.toric_divisor_group()
sage: TDiv.ngens()
3
```

class sage.schemes.toric.divisor.**ToricDivisor_generic**(*v*, *parent*, *check=True*, *reduce=True*)

Bases: *Divisor_generic*

Construct a *(toric Weil) divisor* on the given toric variety.

INPUT:

- `v` – a list of tuples (multiplicity, coordinate).
- `parent` – *ToricDivisorGroup*. The parent divisor group.
- `check` – boolean. Type-check the entries of `v`, see *DivisorGroup_generic*.
- `reduce` – boolean. Combine coefficients in `v`, see *DivisorGroup_generic*.

Warning: Do not construct *ToricDivisor_generic* objects manually. Instead, use either the function *ToricDivisor()* or the method *divisor()* of toric varieties.

EXAMPLES:

```
sage: dP6 = toric_varieties.dP6()
sage: ray = dP6.fan().ray(0)
sage: ray
N(0, 1)
sage: D = dP6.divisor(ray); D
V(x)
sage: D.parent()
Group of toric ZZ-Weil divisors
on 2-d CPR-Fano toric variety covered by 6 affine patches
```

Chern_character()

Return the Chern character of the sheaf $\mathcal{O}(D)$ defined by the divisor D.

You can also use a shortcut *ch()*.

EXAMPLES:

```
sage: dP6 = toric_varieties.dP6()
sage: N = dP6.fan().lattice()
sage: D3 = dP6.divisor(dP6.fan().cone_containing( N(0,1)   ))
sage: D5 = dP6.divisor(dP6.fan().cone_containing( N(-1,-1) ))
sage: D6 = dP6.divisor(dP6.fan().cone_containing( N(0,-1)  ))
sage: D = -D3 + 2*D5 - D6
sage: D.Chern_character()                                                    #␣
↪needs sage.libs.singular
[5*w^2 + y - 2*v + w + 1]
sage: dP6.integrate(D.ch() * dP6.Td())                                       #␣
↪needs sage.libs.singular
-4
```

Chow_cycle(*ring=Integer Ring*)

Returns the Chow homology class of the divisor.

INPUT:

- `ring` – Either `ZZ` (default) or `QQ`. The base ring of the Chow group.

OUTPUT:

The *ChowCycle* represented by the divisor.

EXAMPLES:

```
sage: dP6 = toric_varieties.dP6()
sage: cone = dP6.fan(1)[5]
sage: D = dP6.divisor(cone); D
V(w)
sage: D.Chow_cycle()
( 0 | -1, 0, 1, 1 | 0 )
sage: dP6.Chow_group()(cone)
( 0 | -1, 0, 1, 1 | 0 )
```

Kodaira_map(*names='z'*)

Return the Kodaira map.

The Kodaira map is the rational map $X_\Sigma \to \mathbb{P}^{n-1}$, where n equals the number of sections. It is defined by the monomial sections of the line bundle.

If the divisor is ample and the toric variety smooth or of dimension 2, then this is an embedding.

INPUT:

- `names` – string (optional; default `'z'`). The variable names for the destination projective space.

EXAMPLES:

```
sage: P1.<u,v> = toric_varieties.P1()
sage: D = -P1.K()
sage: D.Kodaira_map()                                                        #␣
↪needs fpylll sage.libs.singular
Scheme morphism:
  From: 1-d CPR-Fano toric variety covered by 2 affine patches
  To:   Closed subscheme of Projective Space of dimension 2
```

(continues on next page)

(continued from previous page)

```
        over Rational Field defined by: -z1^2 + z0*z2
  Defn: Defined on coordinates by sending [u : v] to (v^2 : u*v : u^2)

sage: dP6 = toric_varieties.dP6()
sage: D = -dP6.K()
sage: D.Kodaira_map(names='x')                                              #␣
↪needs fpylll sage.libs.singular
Scheme morphism:
  From: 2-d CPR-Fano toric variety covered by 6 affine patches
  To:   Closed subscheme of Projective Space of dimension 6
        over Rational Field defined by:
          -x1*x5 + x0*x6,       -x2*x3 + x0*x5,       -x1*x3 + x0*x4,
          x4*x5 - x3*x6,        -x1*x2 + x0*x3,       x3*x5 - x2*x6,
          x3*x4 - x1*x6,        x3^2 - x1*x5,         x2*x4 - x1*x5,
          -x1*x5^2 + x2*x3*x6, -x1*x5^3 + x2^2*x6^2
  Defn: Defined on coordinates by sending [x : u : y : v : z : w] to
        (x*u^2*y^2*v : x^2*u^2*y*w : u*y^2*v^2*z : x*u*y*v*z*w :
         x^2*u*z*w^2 : y*v^2*z^2*w : x*v*z^2*w^2)
```

ch()

Return the Chern character of the sheaf $\mathcal{O}(D)$ defined by the divisor D.

You can also use a shortcut *ch()*.

EXAMPLES:

```
sage: dP6 = toric_varieties.dP6()
sage: N = dP6.fan().lattice()
sage: D3 = dP6.divisor(dP6.fan().cone_containing( N(0,1)   ))
sage: D5 = dP6.divisor(dP6.fan().cone_containing( N(-1,-1) ))
sage: D6 = dP6.divisor(dP6.fan().cone_containing( N(0,-1)  ))
sage: D = -D3 + 2*D5 - D6
sage: D.Chern_character()                                                   #␣
↪needs sage.libs.singular
[5*w^2 + y - 2*v + w + 1]
sage: dP6.integrate(D.ch() * dP6.Td())                                      #␣
↪needs sage.libs.singular
-4
```

coefficient(x)

Return the coefficient of `x`.

INPUT:

- `x` – one of the homogeneous coordinates, either given by the variable or its index.

OUTPUT:

The coefficient of `x`.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: D = P2.divisor((11,12,13)); D
11*V(x) + 12*V(y) + 13*V(z)
```

(continues on next page)

(continued from previous page)

```
sage: D.coefficient(1)
12
sage: P2.inject_variables()
Defining x, y, z
sage: D.coefficient(y)
12
```

cohomology(*weight=None*, *deg=None*, *dim=False*)

Return the cohomology of the line bundle associated to the Cartier divisor or reflexive sheaf associated to the Weil divisor.

Note: The cohomology of a toric line bundle/reflexive sheaf is graded by the usual degree as well as by the M-lattice.

INPUT:

- `weight` – (optional) a point of the M-lattice.
- `deg` – (optional) the degree of the cohomology group.
- `dim` – boolean. If `False` (default), the cohomology groups are returned as vector spaces. If `True`, only the dimension of the vector space(s) is returned.

OUTPUT:

The vector space $H^{\text{deg}}(X, \mathcal{O}(D))$ (if `deg` is specified) or a dictionary `{degree:cohomology(degree)}` of all degrees between 0 and the dimension of the variety.

If `weight` is specified, return only the subspace $H^{\text{deg}}(X, \mathcal{O}(D))_{\text{weight}}$ of the cohomology of the given weight.

If `dim==True`, the dimension of the cohomology vector space is returned instead of actual vector space. Moreover, if `deg` was not specified, a vector whose entries are the dimensions is returned instead of a dictionary.

ALGORITHM:

Roughly, Chech cohomology is used to compute the cohomology. For toric divisors, the local sections can be chosen to be monomials (instead of general homogeneous polynomials), this is the reason for the extra grading by $m \in M$. General references would be [Ful1993], [CLS2011]. Here are some salient features of our implementation:

- First, a finite set of M-lattice points is identified that supports the cohomology. The toric divisor determines a (polyhedral) chamber decomposition of $M_{\mathbf{R}}$, see Section 9.1 and Figure 4 of [CLS2011]. The cohomology vanishes on the non-compact chambers. Hence, the convex hull of the vertices of the chamber decomposition contains all non-vanishing cohomology groups. This is returned by the private method `_sheaf_cohomology_support()`.

  It would be more efficient, but more difficult to implement, to keep track of all of the individual chambers. We leave this for future work.

- For each point $m \in M$, the weight-m part of the cohomology can be rewritten as the cohomology of a simplicial complex, see Exercise 9.1.10 of [CLS2011], [Per2007]. This is returned by the private method `_sheaf_complex()`.

  The simplicial complex is the same for all points in a chamber, but we currently do not make use of this and compute each point $m \in M$ separately.

- Finally, the cohomology (over $\mathbf{Q}$) of this simplicial complex is computed in the private method `_sheaf_cohomology()`. Summing over the supporting points $m \in M$ yields the cohomology of the sheaf $\hat{}$.

EXAMPLES:

Example 9.1.7 of Cox, Little, Schenck: "Toric Varieties" [CLS2011]:

```
sage: F = Fan(cones=[(0,1), (1,2), (2,3), (3,4), (4,5), (5,0)],
....:         rays=[(1,0), (1,1), (0,1), (-1,0), (-1,-1), (0,-1)])
sage: dP6 = ToricVariety(F)
sage: D3 = dP6.divisor(2)
sage: D5 = dP6.divisor(4)
sage: D6 = dP6.divisor(5)
sage: D = -D3 + 2*D5 - D6
sage: D.cohomology()
{0: Vector space of dimension 0 over Rational Field,
 1: Vector space of dimension 4 over Rational Field,
 2: Vector space of dimension 0 over Rational Field}
sage: D.cohomology(deg=1)
Vector space of dimension 4 over Rational Field
sage: M = F.dual_lattice()
sage: D.cohomology( M(0,0) )
{0: Vector space of dimension 0 over Rational Field,
 1: Vector space of dimension 1 over Rational Field,
 2: Vector space of dimension 0 over Rational Field}
sage: D.cohomology( weight=M(0,0), deg=1 )
Vector space of dimension 1 over Rational Field
sage: dP6.integrate(D.ch() * dP6.Td())                                    #␣
↪needs sage.libs.singular
-4
```

Note the different output options:

```
sage: D.cohomology()
{0: Vector space of dimension 0 over Rational Field,
 1: Vector space of dimension 4 over Rational Field,
 2: Vector space of dimension 0 over Rational Field}
sage: D.cohomology(dim=True)
(0, 4, 0)
sage: D.cohomology(weight=M(0,0))
{0: Vector space of dimension 0 over Rational Field,
 1: Vector space of dimension 1 over Rational Field,
 2: Vector space of dimension 0 over Rational Field}
sage: D.cohomology(weight=M(0,0), dim=True)
(0, 1, 0)
sage: D.cohomology(deg=1)
Vector space of dimension 4 over Rational Field
sage: D.cohomology(deg=1, dim=True)
4
sage: D.cohomology(weight=M(0,0), deg=1)
Vector space of dimension 1 over Rational Field
sage: D.cohomology(weight=M(0,0), deg=1, dim=True)
1
```

Here is a Weil (non-Cartier) divisor example:

```
sage: K = toric_varieties.Cube_nonpolyhedral().K()
sage: K.is_Weil()
True
sage: K.is_QQ_Cartier()
False
sage: K.cohomology(dim=True)
(0, 0, 0, 1)
```

cohomology_class()

Return the degree-2 cohomology class associated to the divisor.

OUTPUT:

The corresponding cohomology class as an instance of *CohomologyClass*. The cohomology class is the first Chern class of the associated line bundle $\mathcal{O}(D)$.

EXAMPLES:

```
sage: dP6 = toric_varieties.dP6()
sage: D = dP6.divisor(dP6.fan().ray(0))
sage: D.cohomology_class()                                                  #␣
↪needs sage.libs.singular
[y + v - w]
```

cohomology_support()

Return the weights for which the cohomology groups do not vanish.

OUTPUT:

A tuple of dual lattice points. `self.cohomology(weight=m)` does not vanish if and only if `m` is in the output.

Note: This method is provided for educational purposes and it is not an efficient way of computing the cohomology groups.

EXAMPLES:

```
sage: F = Fan(cones=[(0,1), (1,2), (2,3), (3,4), (4,5), (5,0)],
....:         rays=[(1,0), (1,1), (0,1), (-1,0), (-1,-1), (0,-1)])
sage: dP6 = ToricVariety(F)
sage: D3 = dP6.divisor(2)
sage: D5 = dP6.divisor(4)
sage: D6 = dP6.divisor(5)
sage: D = -D3 + 2*D5 - D6
sage: D.cohomology_support()
(M(0, 0), M(1, 0), M(2, 0), M(1, 1))
```

divisor_class()

Return the linear equivalence class of the divisor.

OUTPUT:

The class of the divisor in $Cl(X) \otimes_{\mathbf{Z}} \mathbf{Q}$ as an instance of *ToricRationalDivisorClassGroup*.

EXAMPLES:

```
sage: dP6 = toric_varieties.dP6()
sage: D = dP6.divisor(0)
sage: D.divisor_class()
Divisor class [1, 0, 0, 0]
```

function_value(*point*)

Return the value of the support function at `point`.

Let X be the ambient toric variety of `self`, Σ the fan associated to X, and N the ambient lattice of Σ.

INPUT:

- `point` – either an integer, interpreted as the index of a ray of Σ, or a point of the lattice N.

OUTPUT: An integer or a rational number.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: D = P2.divisor([11,22,44])   # total degree 77
sage: D.function_value(0)
11
sage: N = P2.fan().lattice()
sage: D.function_value( N(1,1) )
33
sage: D.function_value( P2.fan().ray(0) )
11
```

is_Cartier()

Return whether the divisor is a Cartier-divisor.

Note: The sheaf $\mathcal{O}(D)$ associated to the given divisor D is a line bundle if and only if the divisor is Cartier.

EXAMPLES:

```
sage: X = toric_varieties.P4_11169()
sage: D = X.divisor(3)
sage: D.is_Cartier()
False
sage: D.is_QQ_Cartier()
True
```

is_QQ_Cartier()

Return whether the divisor is a $\mathbf{Q}$-Cartier divisor.

A $\mathbf{Q}$-Cartier divisor is a divisor such that some multiple of it is Cartier.

EXAMPLES:

```
sage: X = toric_varieties.P4_11169()
sage: D = X.divisor(3)
sage: D.is_QQ_Cartier()
True

sage: X = toric_varieties.Cube_face_fan()
```

(continues on next page)

(continued from previous page)

```
sage: D = X.divisor(3)
sage: D.is_QQ_Cartier()
False
```

is_QQ_Weil()

Return whether the divisor is a **Q**-Weil-divisor.

Note: This function returns always `True` since *ToricDivisor* can only describe **Q**-Weil divisors.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: D = P2.divisor([1,2,3])
sage: D.is_QQ_Weil()
True
sage: (D/2).is_QQ_Weil()
True
```

is_Weil()

Return whether the divisor is a Weil-divisor.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: D = P2.divisor([1,2,3])
sage: D.is_Weil()
True
sage: (D/2).is_Weil()
False
```

is_ample()

Return whether a **Q**-Cartier divisor is ample.

OUTPUT:

- `True` if the divisor is in the ample cone, `False` otherwise.

Note:

- For a **Q**-Cartier divisor, some positive integral multiple is Cartier. We return whether this associated divisor is ample, i.e. corresponds to an ample line bundle.
- In the orbifold case, the ample cone is an open and full-dimensional cone in the rational divisor class group *ToricRationalDivisorClassGroup*.
- If the variety has worse than orbifold singularities, the ample cone is a full-dimensional cone within the (not full-dimensional) subspace spanned by the Cartier divisors inside the rational (Weil) divisor class group, that is, *ToricRationalDivisorClassGroup*. The ample cone is then relative open (open in this subspace).
- See also *is_nef()*.
- A toric divisor is ample if and only if its support function is strictly convex.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: K = P2.K()
sage: (+K).is_ample()
False
sage: (0*K).is_ample()
False
sage: (-K).is_ample()
True
```

Example 6.1.3, 6.1.11, 6.1.17 of [CLS2011]:

```
sage: from itertools import product
sage: fan = Fan(cones=[(0,1), (1,2), (2,3), (3,0)],
....:           rays=[(-1,2), (0,1), (1,0), (0,-1)])
sage: F2 = ToricVariety(fan,'u1, u2, u3, u4')
sage: def D(a,b): return a*F2.divisor(2) + b*F2.divisor(3)
sage: [ (a,b) for a,b in product(range(-3,3), repeat=2)
....:         if D(a,b).is_ample() ]
[(1, 1), (1, 2), (2, 1), (2, 2)]
sage: [ (a,b) for a,b in product(range(-3,3), repeat=2)
....:         if D(a,b).is_nef() ]
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
```

A (worse than orbifold) singular Fano threefold:

```
sage: points = [(1,0,0),(0,1,0),(0,0,1),(-2,0,-1),(-2,-1,0),(-3,-1,-1),(1,1,1)]
sage: facets = [[0,1,3],[0,1,6],[0,2,4],[0,2,6],[0,3,5],[0,4,5],[1,2,3,4,5,6]]
sage: X = ToricVariety(Fan(cones=facets, rays=points))
sage: X.rational_class_group().dimension()
4
sage: X.Kaehler_cone().rays()
Divisor class [1, 0, 0, 0]
in Basis lattice of The toric rational divisor class group
of a 3-d toric variety covered by 7 affine patches
sage: antiK = -X.K()
sage: antiK.divisor_class()
Divisor class [2, 0, 0, 0]
sage: antiK.is_ample()
True
```

is_integral()

Return whether the coefficients of the divisor are all integral.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: DZZ = P2.toric_divisor_group(base_ring=ZZ).gen(0); DZZ
V(x)
sage: DQQ = P2.toric_divisor_group(base_ring=QQ).gen(0); DQQ
V(x)
sage: DZZ.is_integral()
True
sage: DQQ.is_integral()
True
```

is_nef()

Return whether a $\mathbf{Q}$-Cartier divisor is nef.

OUTPUT:

- `True` if the divisor is in the closure of the ample cone, `False` otherwise.

Note:

- For a $\mathbf{Q}$-Cartier divisor, some positive integral multiple is Cartier. We return whether this associated divisor is nef.
- The nef cone is the closure of the ample cone.
- See also *is_ample()*.
- A toric divisor is nef if and only if its support function is convex (but not necessarily strictly convex).
- A toric Cartier divisor is nef if and only if its linear system is basepoint free.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: K = P2.K()
sage: (+K).is_nef()
False
sage: (0*K).is_nef()
True
sage: (-K).is_nef()
True
```

Example 6.1.3, 6.1.11, 6.1.17 of [CLS2011]:

```
sage: from itertools import product
sage: fan = Fan(cones=[(0,1), (1,2), (2,3), (3,0)],
....:           rays=[(-1,2), (0,1), (1,0), (0,-1)])
sage: F2 = ToricVariety(fan,'u1, u2, u3, u4')
sage: def D(a,b): return a*F2.divisor(2) + b*F2.divisor(3)
sage: [ (a,b) for a,b in product(range(-3,3), repeat=2)
....:         if D(a,b).is_ample() ]
[(1, 1), (1, 2), (2, 1), (2, 2)]
sage: [ (a,b) for a,b in product(range(-3,3), repeat=2)
....:         if D(a,b).is_nef() ]
[(0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2), (2, 0), (2, 1), (2, 2)]
```

m(*cone*)

Return m_σ representing ϕ_D on `cone`.

Let X be the ambient toric variety of this divisor D associated to the fan Σ in lattice N. Let M be the lattice dual to N. Given the cone $\sigma = \langle v_1, \ldots, v_k \rangle$ in Σ, this method searches for a vector $m_\sigma \in M_{\mathbf{Q}}$ such that $\phi_D(v_i) = \langle m_\sigma, v_i \rangle$ for all $i = 1, \ldots, k$, where ϕ_D is the support function of D.

INPUT:

- `cone` – A cone in the fan of the toric variety.

OUTPUT:

- If possible, a point of lattice M.

- If the dual vector cannot be chosen integral, a rational vector is returned.
- If there is no such vector (i.e. `self` is not even a $\mathbb{Q}$-Cartier divisor), a `ValueError` is raised.

EXAMPLES:

```
sage: F = Fan(cones=[(0,1,2,3), (0,1,4)],
....:         rays=[(1,1,1), (1,-1,1), (1,-1,-1), (1,1,-1), (0,0,1)])
sage: X = ToricVariety(F)
sage: square_cone = X.fan().cone_containing(0,1,2,3)
sage: triangle_cone = X.fan().cone_containing(0,1,4)
sage: ray = X.fan().cone_containing(0)
sage: QQ_Cartier = X.divisor([2,2,1,1,1])
sage: QQ_Cartier.m(ray)
M(0, 2, 0)
sage: QQ_Cartier.m(square_cone)
(3/2, 0, 1/2)
sage: QQ_Cartier.m(triangle_cone)
M(1, 0, 1)
sage: QQ_Cartier.m(Cone(triangle_cone))
M(1, 0, 1)
sage: Weil = X.divisor([1,1,1,0,0])
sage: Weil.m(square_cone)
Traceback (most recent call last):
...
ValueError: V(z0) + V(z1) + V(z2) is not QQ-Cartier,
cannot choose a dual vector on 3-d cone
of Rational polyhedral fan in 3-d lattice N
sage: Weil.m(triangle_cone)
M(1, 0, 0)
```

monomial(*point*)

Return the monomial in the homogeneous coordinate ring associated to the `point` in the dual lattice.

INPUT:

- `point` – a point in `self.variety().fan().dual_lattice()`.

OUTPUT:

For a fixed divisor `D`, the sections are generated by monomials in *ToricVariety.coordinate_ring*. Alternatively, the monomials can be described as M-lattice points in the polyhedron `D.polyhedron()`. This method converts the points $m \in M$ into homogeneous polynomials.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: O3_P2 = -P2.K()
sage: M = P2.fan().dual_lattice()
sage: O3_P2.monomial( M(0,0) )
x*y*z
```

move_away_from(*cone*)

Move the divisor away from the orbit closure of `cone`.

INPUT:

- A `cone` of the fan of the toric variety.

OUTPUT:

A (rationally equivalent) divisor that is moved off the orbit closure of the given cone.

Note: A divisor that is Weil but not Cartier might be impossible to move away. In this case, a `ValueError` is raised.

EXAMPLES:

```
sage: F = Fan(cones=[(0,1,2,3), (0,1,4)],
....:         rays=[(1,1,1), (1,-1,1), (1,-1,-1), (1,1,-1), (0,0,1)])
sage: X = ToricVariety(F)
sage: square_cone = X.fan().cone_containing(0,1,2,3)
sage: triangle_cone = X.fan().cone_containing(0,1,4)
sage: line_cone = square_cone.intersection(triangle_cone)
sage: Cartier = X.divisor([2,2,1,1,1])
sage: Cartier
2*V(z0) + 2*V(z1) + V(z2) + V(z3) + V(z4)
sage: Cartier.move_away_from(line_cone)
3*V(z2) + 3*V(z3) - V(z4)
sage: QQ_Weil = X.divisor([1,0,1,1,0])
sage: QQ_Weil.move_away_from(line_cone)
2*V(z2) + V(z3) - 1/2*V(z4)
```

polyhedron()

Return the polyhedron $P_D \subset M$ associated to a toric divisor D.

OUTPUT:

P_D as an instance of `Polyhedron_base`.

EXAMPLES:

```
sage: dP7 = toric_varieties.dP7()
sage: D = dP7.divisor(2)
sage: P_D = D.polyhedron(); P_D
A 0-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex
sage: P_D.Vrepresentation()
(A vertex at (0, 0),)
sage: D.is_nef()
False
sage: dP7.integrate(D.ch() * dP7.Td())                                    #␣
↪needs sage.libs.singular
1
sage: P_antiK = (-dP7.K()).polyhedron(); P_antiK
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 5 vertices
sage: P_antiK.Vrepresentation()
(A vertex at (1, -1), A vertex at (0, 1), A vertex at (1, 0),
 A vertex at (-1, 1), A vertex at (-1, -1))
sage: P_antiK.integral_points()
((-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 0), (0, 1), (1, -1), (1, 0))
```

Example 6.1.3, 6.1.11, 6.1.17 of [CLS2011]:

```
sage: fan = Fan(cones=[(0,1), (1,2), (2,3), (3,0)],
....:           rays=[(-1,2), (0,1), (1,0), (0,-1)])
sage: F2 = ToricVariety(fan,'u1, u2, u3, u4')
sage: D = F2.divisor(3)
sage: D.polyhedron().Vrepresentation()
(A vertex at (0, 0), A vertex at (2, 1), A vertex at (0, 1))
sage: Dprime = F2.divisor(1) + D
sage: Dprime.polyhedron().Vrepresentation()
(A vertex at (2, 1), A vertex at (0, 1), A vertex at (0, 0))
sage: D.is_ample()
False
sage: D.is_nef()
True
sage: Dprime.is_nef()
False
```

A more complicated example where P_D is not a lattice polytope:

```
sage: X = toric_varieties.BCdlOG_base()
sage: antiK = -X.K()
sage: P_D = antiK.polyhedron()
sage: P_D
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 8 vertices
sage: P_D.Vrepresentation()
(A vertex at (1, -1, 0), A vertex at (1, -3, 1),
 A vertex at (1, 1, 1), A vertex at (-5, 1, 1),
 A vertex at (1, 1, -1/2), A vertex at (1, 1/2, -1/2),
 A vertex at (-1, -1, 0), A vertex at (-5, -3, 1))
sage: P_D.Hrepresentation()
(An inequality (-1, 0, 0) x + 1 >= 0, An inequality (0, -1, 0) x + 1 >= 0,
 An inequality (0, 0, -1) x + 1 >= 0, An inequality (1, 0, 4) x + 1 >= 0,
 An inequality (0, 1, 3) x + 1 >= 0, An inequality (0, 1, 2) x + 1 >= 0)
sage: P_D.integral_points()
((-1, -1, 0), (0, -1, 0), (1, -1, 0), (-1, 0, 0), (0, 0, 0),
 (1, 0, 0), (-1, 1, 0), (0, 1, 0), (1, 1, 0), (-5, -3, 1),
 (-4, -3, 1), (-3, -3, 1), (-2, -3, 1), (-1, -3, 1), (0, -3, 1),
 (1, -3, 1), (-5, -2, 1), (-4, -2, 1), (-3, -2, 1), (-2, -2, 1),
 (-1, -2, 1), (0, -2, 1), (1, -2, 1), (-5, -1, 1), (-4, -1, 1),
 (-3, -1, 1), (-2, -1, 1), (-1, -1, 1), (0, -1, 1), (1, -1, 1),
 (-5, 0, 1), (-4, 0, 1), (-3, 0, 1), (-2, 0, 1), (-1, 0, 1),
 (0, 0, 1), (1, 0, 1), (-5, 1, 1), (-4, 1, 1), (-3, 1, 1),
 (-2, 1, 1), (-1, 1, 1), (0, 1, 1), (1, 1, 1))
```

sections()

Return the global sections (as points of the M-lattice) of the line bundle (or reflexive sheaf) associated to the divisor.

OUTPUT: A `tuple` of points of lattice M.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: P2.fan().nrays()
3
```

(continues on next page)

(continued from previous page)

```
sage: P2.divisor(0).sections()
(M(-1, 0), M(-1, 1), M(0, 0))
sage: P2.divisor(1).sections()
(M(0, -1), M(0, 0), M(1, -1))
sage: P2.divisor(2).sections()
(M(0, 0), M(0, 1), M(1, 0))
```

The divisor can be non-nef yet still have sections:

```
sage: rays = [(1,0,0),(0,1,0),(0,0,1),(-2,0,-1),(-2,-1,0),(-3,-1,-1),(1,1,1),(-
↪1,0,0)]
sage: cones = [[0,1,3],[0,1,6],[0,2,4],[0,2,6],[0,3,5],[0,4,5],[1,3,7],[1,6,7],
↪[2,4,7],[2,6,7],[3,5,7],[4,5,7]]
sage: X = ToricVariety(Fan(rays=rays, cones=cones))
sage: D = X.divisor(2); D
V(z2)
sage: D.is_nef()
False
sage: D.sections()
(M(0, 0, 0),)
sage: D.cohomology(dim=True)
(1, 0, 0, 0)
```

sections_monomials()

Return the global sections of the line bundle associated to the Cartier divisor.

The sections are described as monomials in the generalized homogeneous coordinates.

OUTPUT:

- tuple of monomials in the coordinate ring of `self`.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: P2.fan().nrays()
3
sage: P2.divisor(0).sections_monomials()
(z, y, x)
sage: P2.divisor(1).sections_monomials()
(z, y, x)
sage: P2.divisor(2).sections_monomials()
(z, y, x)
```

From [Cox] page 38:

```
sage: lp = LatticePolytope([(1,0), (1,1), (0,1), (-1,0), (0,-1)])
sage: lp
2-d reflexive polytope #5 in 2-d lattice M
sage: dP7 = ToricVariety(FaceFan(lp), 'x1, x2, x3, x4, x5')
sage: AK = -dP7.K()
sage: AK.sections()
(N(-1, 0), N(-1, 1), N(0, -1), N(0, 0),
 N(0, 1), N(1, -1), N(1, 0), N(1, 1))
```

(continues on next page)

(continued from previous page)

```
sage: AK.sections_monomials()
(x3*x4^2*x5, x2*x3^2*x4^2, x1*x4*x5^2, x1*x2*x3*x4*x5,
 x1*x2^2*x3^2*x4, x1^2*x2*x5^2, x1^2*x2^2*x3*x5, x1^2*x2^3*x3^2)
```

class sage.schemes.toric.divisor.**ToricRationalDivisorClassGroup**(*toric_variety*)

Bases: FreeModule_ambient_field, UniqueRepresentation

The rational divisor class group of a toric variety.

The **T-Weil divisor class group** $Cl(X)$ of a toric variety X is a finitely generated abelian group and can contain torsion. Its rank equals the number of rays in the fan of X minus the dimension of X.

The **rational divisor class group** is $Cl(X) \otimes_{\mathbf{Z}} \mathbf{Q}$ and never includes torsion. If X is *smooth*, this equals the **Picard group** $\mathrm{Pic}(X)$, whose elements are the isomorphism classes of line bundles on X. The group law (which we write as addition) is the tensor product of the line bundles. The Picard group of a toric variety is always torsion-free.

Warning: Do not instantiate this class yourself. Use *rational_class_group()* method of *toric varieties* if you need the divisor class group. Or you can obtain it as the parent of any divisor class constructed, for example, via *ToricDivisor_generic.divisor_class()*.

INPUT:

- toric_variety – toric variety <sage.schemes.toric.variety.ToricVariety_field.

OUTPUT:

- rational divisor class group of a toric variety.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: P2.rational_class_group()
The toric rational divisor class group of a 2-d CPR-Fano
toric variety covered by 3 affine patches
sage: D = P2.divisor(0); D
V(x)
sage: Dclass = D.divisor_class(); Dclass
Divisor class [1]
sage: Dclass.lift()
V(y)
sage: Dclass.parent()
The toric rational divisor class group of a 2-d CPR-Fano
toric variety covered by 3 affine patches
```

Element

alias of *ToricRationalDivisorClass*

class sage.schemes.toric.divisor.**ToricRationalDivisorClassGroup_basis_lattice**(*group*)

Bases: FreeModule_ambient_pid

Construct the basis lattice of the group.

INPUT:

- group – *toric rational divisor class group*.

OUTPUT:

- the basis lattice of `group`.

EXAMPLES:

```
sage: P1xP1 = toric_varieties.P1xP1()
sage: L = P1xP1.Kaehler_cone().lattice()
sage: L
Basis lattice of The toric rational divisor class group of a
2-d CPR-Fano toric variety covered by 4 affine patches
sage: L.basis()
[
Divisor class [1, 0],
Divisor class [0, 1]
]
```

Element

alias of *ToricRationalDivisorClass*

sage.schemes.toric.divisor.**is_ToricDivisor**(*x*)

Test whether `x` is a toric divisor.

INPUT:

- `x` – anything.

OUTPUT:

- `True` if `x` is an instance of *ToricDivisor_generic* and `False` otherwise.

EXAMPLES:

```
sage: from sage.schemes.toric.divisor import is_ToricDivisor
sage: is_ToricDivisor(1)
False
sage: P2 = toric_varieties.P2()
sage: D = P2.divisor(0); D
V(x)
sage: is_ToricDivisor(D)
True
```

16.5 Toric rational divisor classes

This module is a part of the framework for *toric varieties*.

AUTHORS:

- Volker Braun and Andrey Novoseltsev (2010-09-05): initial version.

class sage.schemes.toric.divisor_class.**ToricRationalDivisorClass**

Bases: `Vector_rational_dense`

Create a toric rational divisor class.

Warning: You probably should not construct divisor classes explicitly.

INPUT:

- same as for `Vector_rational_dense`.

OUTPUT:

- toric rational divisor class.

lift()

Return a divisor representing this divisor class.

OUTPUT:

An instance of `ToricDivisor` representing `self`.

EXAMPLES:

```
sage: X = toric_varieties.Cube_nonpolyhedral()
sage: D = X.divisor([0,1,2,3,4,5,6,7]); D
V(z1) + 2*V(z2) + 3*V(z3) + 4*V(z4) + 5*V(z5) + 6*V(z6) + 7*V(z7)
sage: D.divisor_class()
Divisor class [29, 6, 8, 10, 0]
sage: Dequiv = D.divisor_class().lift(); Dequiv
15*V(z1) - 11*V(z2) - 9*V(z5) + 19*V(z6) + 10*V(z7)
sage: Dequiv == D
False
sage: Dequiv.divisor_class() == D.divisor_class()
True
```

sage.schemes.toric.divisor_class.**is_ToricRationalDivisorClass**(*x*)

Check if `x` is a toric rational divisor class.

INPUT:

- `x` – anything.

OUTPUT:

- `True` if `x` is a toric rational divisor class, `False` otherwise.

EXAMPLES:

```
sage: from sage.schemes.toric.divisor_class import is_ToricRationalDivisorClass
sage: is_ToricRationalDivisorClass(1)
False
sage: dP6 = toric_varieties.dP6()
sage: D = dP6.rational_class_group().gen(0); D
Divisor class [1, 0, 0, 0]
sage: is_ToricRationalDivisorClass(D)
True
```

16.6 The Chow group of a toric variety

In general, the Chow group is an algebraic version of a homology theory. That is, the objects are formal linear combinations of submanifolds modulo relations. In particular, the objects of the Chow group are formal linear combinations of algebraic subvarieties and the equivalence relation is rational equivalence. There is no relative version of the Chow group, so it is not a generalized homology theory.

The Chow groups of smooth or mildly singular toric varieties are almost the same as the homology groups:

- For smooth toric varieties, $A_k(X) = H_{2k}(X, \mathbf{Z})$. While they are the same, using the cohomology ring instead of the Chow group will be much faster! The cohomology ring does not try to keep track of torsion and uses Groebner bases to encode the cup product.
- For simplicial toric varieties, $A_k(X) \otimes \mathbf{Q} = H_{2k}(X, \mathbf{Q})$.

Note that in these cases the odd-dimensional (co)homology groups vanish. But for sufficiently singular toric varieties the Chow group differs from the homology groups (and the odd-dimensional homology groups no longer vanish). For singular varieties the Chow group is much easier to compute than the (co)homology groups.

The toric Chow group of a toric variety is the Chow group generated by the toric subvarieties, that is, closures of orbits under the torus action. These are in one-to-one correspondence with the cones of the fan and, therefore, the toric Chow group is a quotient of the free Abelian group generated by the cones. In particular, the toric Chow group has finite rank. One can show [FMSS1995] that the toric Chow groups equal the "full" Chow group of a toric variety, so there is no need to distinguish these in the following.

AUTHORS:

- Volker Braun (2010-08-09): Initial version

REFERENCES:

- Wikipedia article Chow_ring
- [FMSS1995]
- [Ful1993], Chapter 5.1, "Chow Groups"

EXAMPLES:

```
sage: X = toric_varieties.Cube_deformation(7)
sage: X.is_smooth()
False
sage: X.is_orbifold()
False
sage: A = X.Chow_group()
sage: A.degree()
(Z, C7, C2 x C2 x Z^5, Z)
sage: A.degree(2).ngens()
7
sage: a = sum( A.gen(i) * (i+1) for i in range(A.ngens()) )   # an element of A
sage: a  # long time (2s on sage.math, 2011)
( 9 | 1 mod 7 | 1 mod 2, 0 mod 2, 4, 5, 6, 7, 8 | 3 )
```

The Chow group elements are printed as `( a0 | a1 mod 7 | a2 mod 2, a3 mod 2, a4, a5, a6, a7, a8 | a9 )`, which denotes the element of the Chow group in the same basis as `A.degree()`. The `|` separates individual degrees, so the example means:

- The degree-0 part is $3 \in \mathbf{Z}$.
- The degree-1 part is $1 \in \mathbf{Z}_7$.

- The torsion of the degree-2 Chow group is $(0,1) \in \mathbf{Z}_2 \oplus \mathbf{Z}_2$.
- The free part of the degree-2 Chow group is $(4,5,6,7,8) \in \mathbf{Z}^5$.
- The degree-3 part is $9 \in \mathbf{Z}$.

Note that the generators `A.gens()` are not sorted in any way. In fact, they may be of mixed degree. Use `A.gens(degree=d)` to obtain the generators in a fixed degree `d`. See *ChowGroup_class.gens()* for more details.

Cones of toric varieties can determine their own Chow cycle:

```
sage: A = X.Chow_group(); A
Chow group of 3-d toric variety covered by 6 affine patches
sage: cone = X.fan(dim=2)[3]; cone
2-d cone of Rational polyhedral fan in 3-d lattice N
sage: A_cone = A(cone); A_cone
( 0 | 6 mod 7 | 0 mod 2, 0 mod 2, 0, 0, 0, 0, 0 | 0 )
sage: A_cone.degree()
1
sage: 2 * A_cone
( 0 | 5 mod 7 | 0 mod 2, 0 mod 2, 0, 0, 0, 0, 0 | 0 )
sage: A_cone + A.gen(0)
( 0 | 6 mod 7 | 1 mod 2, 0 mod 2, 0, 0, 0, 0, 0 | 0 )
```

Chow cycles can be of mixed degrees:

```
sage: mixed = sum(A.gens()); mixed
( 1 | 4 mod 7 | 1 mod 2, 1 mod 2, 1, 1, 1, 1, 1 | 1 )
sage: mixed.project_to_degree(1)
( 0 | 4 mod 7 | 0 mod 2, 0 mod 2, 0, 0, 0, 0, 0 | 0 )
sage: sum( mixed.project_to_degree(i) for i in range(X.dimension()+1) ) == mixed
True
```

class sage.schemes.toric.chow_group.**ChowCycle**(*parent*, *v*, *check=True*)

Bases: FGP_Element

The elements of the Chow group.

Warning: Do not construct *ChowCycle* objects manually. Instead, use the parent *ChowGroup* to obtain generators or Chow cycles corresponding to cones of the fan.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: A = P2.Chow_group()
sage: A.gens()
(( 0 | 0 | 1 ), ( 0 | 1 | 0 ), ( 1 | 0 | 0 ))
sage: cone = P2.fan(1)[0]
sage: A(cone)
( 0 | 1 | 0 )
sage: A( Cone([(1,0)]) )
( 0 | 1 | 0 )
```

cohomology_class()

Return the (Poincaré-dual) cohomology class.

Consider a simplicial cone of the fan, that is, a d-dimensional cone spanned by d rays. Take the product of the corresponding d homogeneous coordinates. This monomial represents a cohomology classes of the toric variety X, see *cohomology_ring()*. Its cohomological degree is $2d$, which is the same degree as the Poincaré-dual of the (real) $\dim(X) - 2d$-dimensional torus orbit associated to the simplicial cone. By linearity, we can associate a cohomology class to each Chow cycle of a simplicial toric variety.

If the toric variety is compact and smooth, the associated cohomology class actually is the Poincaré dual (over the integers) of the Chow cycle. In particular, integrals of dual cohomology classes perform intersection computations.

If the toric variety is compact and has at most orbifold singularities, the torsion parts in cohomology and the Chow group can differ. But they are still isomorphic as rings over the rationals. Moreover, the normalization of integration (*volume_class*) and *count_points()* are chosen to agree.

OUTPUT:

The *CohomologyClass* which is associated to the Chow cycle.

If the toric variety is not simplicial, that is, has worse than orbifold singularities, there is no way to associate a cohomology class of the correct degree. In this case, *cohomology_class()* raises a `ValueError`.

EXAMPLES:

```
sage: # needs sage.libs.singular
sage: dP6 = toric_varieties.dP6()
sage: cone = dP6.fan().cone_containing(2,3)
sage: HH = dP6.cohomology_ring()
sage: A = dP6.Chow_group()
sage: HH(cone)
[-w^2]
sage: A(cone)
( 1 | 0, 0, 0, 0 | 0 )
sage: A(cone).cohomology_class()
[-w^2]
```

Here is an example of a toric variety with orbifold singularities, where we can also use the isomorphism with the rational cohomology ring:

```
sage: # needs sage.libs.singular
sage: WP4 = toric_varieties.P4_11169()
sage: A = WP4.Chow_group()
sage: HH = WP4.cohomology_ring()
sage: cone3d = Cone([(0,0,1,0), (0,0,0,1), (-9,-6,-1,-1)])
sage: A(cone3d)
( 0 | -1 | 0 | 0 | 0 )
sage: HH(cone3d)
[3*z4^3]
sage: D = -WP4.K()  # the anticanonical divisor
sage: A(D)
( 0 | 0 | 0 | -18 | 0 )
sage: HH(D)
[18*z4]
sage: WP4.integrate( A(cone3d).cohomology_class() * D.cohomology_class() )
1
sage: WP4.integrate( HH(cone3d) * D.cohomology_class() )
1
```

(continues on next page)

(continued from previous page)

```
sage: A(cone3d).intersection_with_divisor(D).count_points()
1
```

count_points()

Return the number of points in the Chow cycle.

OUTPUT:

An element of `self.base_ring()`, which is usually $\mathbf{Z}$. The number of points in the Chow cycle.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: A = P2.Chow_group()
sage: a = 5*A.gen(2) + 7*A.gen(1); a
( 5 | 7 | 0 )
sage: a.count_points()
5
```

In the case of a smooth complete toric variety, the Chow (homology) groups are Poincaré dual to the integral cohomology groups. Here is such a smooth example:

```
sage: D = P2.divisor(1)
sage: a = D.Chow_cycle()
sage: aD = a.intersection_with_divisor(D)
sage: aD.count_points()
1
sage: P2.integrate(aD.cohomology_class())                                    #␣
↪needs sage.libs.singular
1
```

For toric varieties with at most orbifold singularities, the isomorphism only holds over $\mathbf{Q}$. But the normalization of the integral is still chosen such that the intersection numbers (which are potentially rational) computed both ways agree:

```
sage: P1xP1_Z2 = toric_varieties.P1xP1_Z2()
sage: Dt = P1xP1_Z2.divisor(1);  Dt
V(t)
sage: Dy = P1xP1_Z2.divisor(3);  Dy
V(y)
sage: Dt.Chow_cycle(QQ).intersection_with_divisor(Dy).count_points()
1/2
sage: P1xP1_Z2.integrate(Dt.cohomology_class() * Dy.cohomology_class())      #␣
↪needs sage.libs.singular
1/2
```

degree()

The degree of the Chow cycle.

OUTPUT:

Integer. The complex dimension of the subvariety representing the Chow cycle. Raises a `ValueError` if the Chow cycle is a sum of mixed degree cycles.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: A = P2.Chow_group()
sage: [ a.degree() for a in A.gens() ]
[2, 1, 0]
```

intersection_with_divisor(*divisor*)

Intersect the Chow cycle with `divisor`.

See Chapter 5.1 of [Ful1993] for a description of the toric algorithm.

INPUT:

- `divisor` – a *ToricDivisor* that can be moved away from the Chow cycle. For example, any Cartier divisor. See also *ToricDivisor.move_away_from*.

OUTPUT:

A new *ChowCycle*. If the divisor is not Cartier then this method potentially raises a `ValueError`, indicating that the divisor cannot be made transversal to the Chow cycle.

EXAMPLES:

```
sage: dP6 = toric_varieties.dP6()
sage: cone = dP6.fan().cone_containing(2); cone
1-d cone of Rational polyhedral fan in 2-d lattice N
sage: D = dP6.divisor(cone); D
V(y)
sage: A = dP6.Chow_group()
sage: A(cone)
( 0 | 0, 0, 1, 0 | 0 )
sage: intersection = A(cone).intersection_with_divisor(D); intersection
( -1 | 0, 0, 0, 0 | 0 )
sage: intersection.count_points()
-1
```

You can do the same computation over the rational Chow group since there is no torsion in this case:

```
sage: A_QQ = dP6.Chow_group(base_ring=QQ)
sage: A_QQ(cone)
( 0 | 0, 0, 0, 1 | 0 )
sage: intersection_QQ = A_QQ(cone).intersection_with_divisor(D); intersection
( -1 | 0, 0, 0, 0 | 0 )
sage: intersection_QQ.count_points()
-1
sage: type(intersection_QQ.count_points())
<... 'sage.rings.rational.Rational'>
sage: type(intersection.count_points())
<... 'sage.rings.integer.Integer'>
```

project_to_degree(*degree*)

Project a (mixed-degree) Chow cycle to the given `degree`.

INPUT:

- `degree` – integer. The degree to project to.

OUTPUT:

The projection of the Chow class to the given degree as a new *ChowCycle* of the same Chow group.

EXAMPLES:

```
sage: A = toric_varieties.P2().Chow_group()
sage: cycle = 10*A.gen(0) + 11*A.gen(1) + 12*A.gen(2); cycle
( 12 | 11 | 10 )
sage: cycle.project_to_degree(2)
( 0 | 0 | 10 )
```

class sage.schemes.toric.chow_group.**ChowGroupFactory**

Bases: UniqueFactory

Factory for *ChowGroup_class*.

create_key_and_extra_args(*toric_variety*, *base_ring=Integer Ring*, *check=True*)

Create a key that uniquely determines the *ChowGroup_class*.

INPUT:

- `toric_variety` – a toric variety.
- `base_ring` – either $\mathbf{Z}$ (default) or $\mathbf{Q}$. The coefficient ring of the Chow group.
- `check` – boolean (default: `True`).

EXAMPLES:

```
sage: from sage.schemes.toric.chow_group import *
sage: P2 = toric_varieties.P2()
sage: ChowGroup(P2, ZZ, check=True) == ChowGroup(P2, ZZ, check=False)    #␣
↪indirect doctest
True
```

create_object(*version*, *key*, ***extra_args*)

Create a *ChowGroup_class*.

INPUT:

- `version` – object version. Currently not used.
- `key` – a key created by *create_key_and_extra_args()*.
- `**extra_args` – Currently not used.

EXAMPLES:

```
sage: from sage.schemes.toric.chow_group import *
sage: P2 = toric_varieties.P2()
sage: ChowGroup(P2)    # indirect doctest
Chow group of 2-d CPR-Fano toric variety covered by 3 affine patches
```

class sage.schemes.toric.chow_group.**ChowGroup_class**(*toric_variety*, *base_ring*, *check*)

Bases: FGP_Module_class, WithEqualityById

The Chow group of a toric variety.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: from sage.schemes.toric.chow_group import ChowGroup_class
sage: A = ChowGroup_class(P2,ZZ,True);  A
Chow group of 2-d CPR-Fano toric variety covered by 3 affine patches
sage: A.an_element()
( 0 | 0 | 1 )
```

Element

alias of *ChowCycle*

coordinate_vector(*chow_cycle*, *degree=None*, *reduce=True*)

Return the coordinate vector of the `chow_cycle`.

INPUT:

- `chow_cycle` – a *ChowCycle*.
- `degree` – None (default) or an integer.
- `reduce` – boolean (default: `True`). Whether to reduce modulo the invariants.

OUTPUT:

- If `degree is None` (default), the coordinate vector relative to the basis `self.gens()` is returned.
- If some integer `degree=d` is specified, the chow cycle is projected to the given degree and the coordinate vector relative to the basis `self.gens(degree=d)` is returned.

EXAMPLES:

```
sage: A = toric_varieties.P2().Chow_group()
sage: a = A.gen(0) + 2*A.gen(1) + 3*A.gen(2)
sage: A.coordinate_vector(a)
(1, 2, 3)
sage: A.coordinate_vector(a, degree=1)
(2)
```

degree(*k=None*)

Return the degree-k Chow group.

INPUT:

- `k` – an integer or `None` (default). The degree of the Chow group.

OUTPUT:

- if k was specified, the Chow group A_k as an Abelian group.
- if k was not specified, a tuple containing the Chow groups in all degrees.

Note:

- For a smooth toric variety, this is the same as the Poincaré-dual cohomology group $H^{d-2k}(X, \mathbf{Z})$.
- For a simplicial toric variety ("orbifold"), $A_k(X) \otimes \mathbf{Q} = H^{d-2k}(X, \mathbf{Q})$.

EXAMPLES:

Four exercises from page 65 of [Ful1993]. First, an example with $A_1(X) = \mathbf{Z} \oplus \mathbf{Z}/3\mathbf{Z}$:

```
sage: X = ToricVariety(Fan(cones=[[0,1], [1,2], [2,0]],
....:                      rays=[[2,-1], [-1,2], [-1,-1]]))
sage: A = X.Chow_group()
sage: A.degree(1)
C3 x Z
```

Second, an example with $A_2(X) = \mathbf{Z}^2$:

```
sage: points = [[1,0,0], [0,1,0], [0,0,1], [1,-1,1], [-1,0,-1]]
sage: l = LatticePolytope(points)
sage: l.show3d()                                                    #␣
↪needs sage.plot
sage: X = ToricVariety(FaceFan(l))
sage: A = X.Chow_group()
sage: A.degree(2)
Z^2
```

Third, an example with $A_2(X) = \mathbf{Z}^5$:

```
sage: cube = [[ 1,0,0], [0, 1,0], [0,0, 1], [-1, 1, 1],
....:         [-1,0,0], [0,-1,0], [0,0,-1], [ 1,-1,-1]]
sage: lat_cube = LatticePolytope(cube)
sage: X = ToricVariety(FaceFan((LatticePolytope(lat_cube))))
sage: X.Chow_group().degree(2)
Z^5
```

Fourth, a fan that is not the fan over a polytope. Combinatorially, the fan is the same in the third example, only the coordinates of the first point are different. But the resulting fan is not the face fan of a cube, so the variety is "more singular". Its Chow group has torsion, $A_2(X) = \mathbf{Z}^5 \oplus \mathbf{Z}/2$:

```
sage: rays = [[ 1, 2, 3], [ 1,-1, 1], [-1, 1, 1], [-1,-1, 1],
....:         [-1,-1,-1], [-1, 1,-1], [ 1,-1,-1], [ 1, 1,-1]]
sage: cones = [[0,1,2,3], [4,5,6,7], [0,1,7,6],
....:          [4,5,3,2], [0,2,5,7], [4,6,1,3]]
sage: X = ToricVariety(Fan(cones, rays))
sage: X.Chow_group().degree(2)  # long time (2s on sage.math, 2011)
C2 x Z^5
```

Finally, Example 1.3 of [FS1994]:

```
sage: def points_mod(k):
....:     return matrix([[ 1, 1, 2*k+1], [ 1,-1, 1],
....:                    [-1, 1, 1], [-1,-1, 1], [-1,-1,-1],
....:                    [-1, 1,-1], [ 1,-1,-1], [ 1, 1,-1]])
sage: def rays(k):
....:     return matrix([[ 1,  1,  1],
....:                    [ 1, -1,  1],
....:                    [-1,  1,  1]]).solve_left(points_mod(k)).rows()
sage: cones = [[0,1,2,3], [4,5,6,7], [0,1,7,6], [4,5,3,2], [0,2,5,7], [4,6,1,3]]
sage: X_Delta = lambda k: ToricVariety(Fan(cones=cones, rays=rays(k)))
sage: X_Delta(0).Chow_group().degree()  # long time (3s on sage.math, 2011)
(Z, Z, Z^5, Z)
sage: X_Delta(1).Chow_group().degree()  # long time (3s on sage.math, 2011)
```

(continues on next page)

(continued from previous page)

```
(Z, 0, Z^5, Z)
sage: X_Delta(2).Chow_group().degree()  # long time (3s on sage.math, 2011)
(Z, C2, Z^5, Z)
sage: X_Delta(2).Chow_group(base_ring=QQ).degree()  # long time (4s on sage.
↪math, 2011)
(Q, 0, Q^5, Q)
```

gens(*degree=None*)

Return the generators of the Chow group.

INPUT:

- `degree` – integer (optional). The degree of the Chow group.

OUTPUT:

- if no degree is specified, the generators of the whole Chow group. The chosen generators may be of mixed degree.
- if `degree`= k was specified, the generators of the degree-k part A_k of the Chow group.

EXAMPLES:

```
sage: A = toric_varieties.P2().Chow_group()
sage: A.gens()
(( 0 | 0 | 1 ), ( 0 | 1 | 0 ), ( 1 | 0 | 0 ))
sage: A.gens(degree=1)
(( 0 | 1 | 0 ),)
```

relation_gens()

Return the Chow cycles equivalent to zero.

For each $d-k-1$-dimensional cone $\rho \in \Sigma^{(d-k-1)}$, the relations in $A_k(X)$, that is the cycles equivalent to zero, are generated by

$$0 \stackrel{!}{=} \mathrm{div}(u) = \sum_{\rho<\sigma\in\Sigma^{(n-p)}} \langle u, n_{\rho,\sigma}\rangle V(\sigma), \qquad u \in M(\rho)$$

where $n_{\rho,\sigma}$ is a (randomly chosen) lift of the generator of $N_\sigma/N_\rho \simeq \mathbf{Z}$. See also Exercise 12.5.7 of [CLS2011].

See also `relations()` to obtain the relations as submodule of the free module generated by the cones. Or use `self.relations().gens()` to list the relations in the free module.

OUTPUT:

A tuple of Chow cycles, each rationally equivalent to zero, that generates the rational equivalence.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: A = P2.Chow_group()
sage: first = A.relation_gens()[0]; first
( 0 | 0 | 0 )
sage: first.is_zero()
True
sage: first.lift()
(0, 1, 0, -1, 0, 0, 0)
```

scheme()

Return the underlying toric variety.

OUTPUT: A *ToricVariety*.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: A = P2.Chow_group()
sage: A.scheme()
2-d CPR-Fano toric variety covered by 3 affine patches
sage: A.scheme() is P2
True
```

class sage.schemes.toric.chow_group.**ChowGroup_degree_class**(*A*, *d*)

Bases: SageObject

A fixed-degree subgroup of the Chow group of a toric variety.

Warning: Use *degree()* to construct *ChowGroup_degree_class* instances.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: A = P2.Chow_group(); A
Chow group of 2-d CPR-Fano toric variety covered by 3 affine patches
sage: A.degree()
(Z, Z, Z)
sage: A.degree(2)
Z
sage: type(_)
<class 'sage.schemes.toric.chow_group.ChowGroup_degree_class'>
```

gen(*i*)

Return the `i`-th generator of the Chow group of fixed degree.

INPUT:

- `i` – integer. The index of the generator to be returned.

OUTPUT: A Chow cycle.

EXAMPLES:

```
sage: projective_plane = toric_varieties.P2()
sage: A2 = projective_plane.Chow_group().degree(2)
sage: A2.gen(0)
( 0 | 0 | 1 )
```

gens()

Return the generators of the Chow group of fixed degree.

OUTPUT: A tuple of Chow cycles of fixed degree generating *module()*.

EXAMPLES:

```
sage: projective_plane = toric_varieties.P2()
sage: A2 = projective_plane.Chow_group().degree(2)
sage: A2.gens()
(( 0 | 0 | 1 ),)
```

module()

Return the submodule of the toric Chow group generated.

OUTPUT: A sage.modules.fg_pid.fgp_module.FGP_Module_class.

EXAMPLES:

```
sage: projective_plane = toric_varieties.P2()
sage: A2 = projective_plane.Chow_group().degree(2)
sage: A2.module()
Finitely generated module V/W over Integer Ring with invariants (0)
```

ngens()

Return the number of generators.

OUTPUT: An integer.

EXAMPLES:

```
sage: projective_plane = toric_varieties.P2()
sage: A2 = projective_plane.Chow_group().degree(2)
sage: A2.ngens()
1
```

sage.schemes.toric.chow_group.**is_ChowCycle**(*x*)

Return whether x is a *ChowCycle*.

INPUT:

- x – anything.

OUTPUT: True or False.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: A = P2.Chow_group()
sage: from sage.schemes.toric.chow_group import *
sage: is_ChowCycle(A)
False
sage: is_ChowCycle(A.an_element())
True
sage: is_ChowCycle('Victoria')
False
```

sage.schemes.toric.chow_group.**is_ChowGroup**(*x*)

Return whether x is a *ChowGroup_class*.

INPUT:

- x – anything.

OUTPUT: True or False.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: A = P2.Chow_group()
sage: from sage.schemes.toric.chow_group import is_ChowGroup
sage: is_ChowGroup(A)
True
sage: is_ChowGroup('Victoria')
False
```

16.7 Toric ideals

A toric ideal (associated to an integer matrix A) is an ideal of the form

$$I_A = \left\langle x^u - x^v : u, v \in \mathbf{Z}^n_{\geq}, u - v \in \ker(A) \right\rangle$$

In other words, it is an ideal generated by irreducible "binomials", that is, differences of monomials without a common factor. Since the Buchberger algorithm preserves this property, any Groebner basis is then also generated by binomials.

EXAMPLES:

```
sage: A = matrix([[1,1,1], [0,1,2]])
sage: IA = ToricIdeal(A)
sage: IA.ker()
Free module of degree 3 and rank 1 over Integer Ring
User basis matrix:
[-1  2 -1]
sage: IA
Ideal (-z1^2 + z0*z2) of Multivariate Polynomial
Ring in z0, z1, z2 over Rational Field
```

Here, the "naive" ideal generated by $z_0 z_2 - z_1^2$ does already equal the toric ideal. But that is not true in general! For example, this toric ideal ([Stu1997], Example 1.2) is the twisted cubic and cannot be generated by $2 = \dim \ker(A)$ polynomials:

```
sage: A = matrix([[3,2,1,0], [0,1,2,3]])
sage: IA = ToricIdeal(A)
sage: IA.ker()
Free module of degree 4 and rank 2 over Integer Ring
User basis matrix:
[-1  1  1 -1]
[-1  2 -1  0]
sage: IA
Ideal (-z1*z2 + z0*z3, -z1^2 + z0*z2, z2^2 - z1*z3) of
Multivariate Polynomial Ring in z0, z1, z2, z3 over Rational Field
```

The following family of toric ideals is from Example 4.4 of [Stu1997]. One can show that I_d is generated by one quadric and d binomials of degree d:

```
sage: def I(d):
....:     return ToricIdeal(matrix([[1,1,1,1,1], [0,1,1,0,0], [0,0,1,1,d]]))
```

(continues on next page)

(continued from previous page)

```
sage: I(2)
Ideal (-z3^2 + z0*z4,
       z0*z2 - z1*z3,
       z2*z3 - z1*z4) of
Multivariate Polynomial Ring in z0, z1, z2, z3, z4 over Rational Field
sage: I(3)
Ideal (-z3^3 + z0^2*z4,
       z0*z2 - z1*z3,
       z2*z3^2 - z0*z1*z4,
       z2^2*z3 - z1^2*z4) of
Multivariate Polynomial Ring in z0, z1, z2, z3, z4 over Rational Field
sage: I(4)
Ideal (-z3^4 + z0^3*z4,
       z0*z2 - z1*z3,
       z2*z3^3 - z0^2*z1*z4,
       z2^2*z3^2 - z0*z1^2*z4,
       z2^3*z3 - z1^3*z4) of
Multivariate Polynomial Ring in z0, z1, z2, z3, z4 over Rational Field
```

Finally, the example in [SH1995b]

```
sage: A = matrix(ZZ, [ [15,  4, 14, 19,  2,  1, 10, 17],
....:                  [18, 11, 13,  5, 16, 16,  8, 19],
....:                  [11,  7,  8, 19, 15, 18, 14,  6],
....:                  [17, 10, 13, 17, 16, 14, 15, 18] ])
sage: IA = ToricIdeal(A)     # long time
sage: IA.ngens()             # long time
213
```

AUTHORS:

- Volker Braun (2011-01-03): Initial version

class sage.schemes.toric.ideal.**ToricIdeal**(*A*, *names='z'*, *base_ring=Rational Field*, *polynomial_ring=None*, *algorithm='HostenSturmfels'*)

Bases: MPolynomialIdeal

This class represents a toric ideal defined by an integral matrix.

INPUT:

- `A` – integer matrix. The defining matrix of the toric ideal.
- `names` – string (optional). Names for the variables. By default, this is `'z'` and the variables will be named `z0`, `z1`, ...
- `base_ring` – a ring (optional). Default: $\mathbf{Q}$. The base ring of the ideal. A toric ideal uses only coefficients ± 1.
- `polynomial_ring` – a polynomial ring (optional). The polynomial ring to construct the ideal in.

  You may specify the ambient polynomial ring via the `polynomial_ring` parameter or via the `names` and `base_ring` parameter. A `ValueError` is raised if you specify both.
- `algorithm` – string (optional). The algorithm to use. For now, must be `'HostenSturmfels'` which is the algorithm proposed by Hosten and Sturmfels in [SH1995b].

EXAMPLES:

```
sage: A = matrix([[1,1,1], [0,1,2]])
sage: ToricIdeal(A)
Ideal (-z1^2 + z0*z2) of Multivariate Polynomial Ring
in z0, z1, z2 over Rational Field
```

First way of specifying the polynomial ring:

```
sage: ToricIdeal(A, names='x,y,z', base_ring=ZZ)
Ideal (-y^2 + x*z) of Multivariate Polynomial Ring
in x, y, z over Integer Ring
```

Second way of specifying the polynomial ring:

```
sage: R.<x,y,z> = ZZ[]
sage: ToricIdeal(A, polynomial_ring=R)
Ideal (-y^2 + x*z) of Multivariate Polynomial Ring
in x, y, z over Integer Ring
```

It is an error to specify both:

```
sage: ToricIdeal(A, names='x,y,z', polynomial_ring=R)
Traceback (most recent call last):
...
ValueError: you must not specify both variable names and a polynomial ring
```

A()

Return the defining matrix.

OUTPUT: An integer matrix.

EXAMPLES:

```
sage: A = matrix([[1,1,1], [0,1,2]])
sage: IA = ToricIdeal(A)
sage: IA.A()
[1 1 1]
[0 1 2]
```

ker()

Return the kernel of the defining matrix.

OUTPUT: The kernel of `self.A()`.

EXAMPLES:

```
sage: A = matrix([[1,1,1], [0,1,2]])
sage: IA = ToricIdeal(A)
sage: IA.ker()
Free module of degree 3 and rank 1 over Integer Ring
 User basis matrix: [-1  2 -1]
```

nvariables()

Return the number of variables of the ambient polynomial ring.

OUTPUT: An integer. The number of columns of the defining matrix *A()*.

EXAMPLES:

```
sage: A = matrix([[1,1,1], [0,1,2]])
sage: IA = ToricIdeal(A)
sage: IA.nvariables()
3
```

16.8 Morphisms of toric varieties

There are three "obvious" ways to map toric varieties to toric varieties:

1. Polynomial maps in local coordinates, the usual morphisms in algebraic geometry.
2. Polynomial maps in the (global) homogeneous coordinates.
3. Toric morphisms, that is, algebraic morphisms equivariant with respect to the torus action on the toric variety.

Both 2 and 3 are special cases of 1, which is just to say that we always remain within the realm of algebraic geometry. But apart from that, none is included in one of the other cases. In the examples below, we will explore some algebraic maps that can or can not be written as a toric morphism. Often a toric morphism can be written with polynomial maps in homogeneous coordinates, but sometimes it cannot.

The toric morphisms are perhaps the most mysterious at the beginning. Let us quickly review their definition (See Definition 3.3.3 of [CLS2011]). Let Σ_1 be a fan in $N_{1,\mathbf{R}}$ and Σ_2 be a fan in $N_{2,\mathbf{R}}$. A morphism $\phi : X_{\Sigma_1} \to X_{\Sigma_2}$ of the associated toric varieties is toric if ϕ maps the maximal torus $T_{N_1} \subseteq X_{\Sigma_1}$ into $T_{N_2} \subseteq X_{\Sigma_2}$ and $\phi|_{T_N}$ is a group homomorphism.

The data defining a toric morphism is precisely what defines a fan morphism (see `fan_morphism`), extending the more familiar dictionary between toric varieties and fans. Toric geometry is a functor from the category of fans and fan morphisms to the category of toric varieties and toric morphisms.

Note: Do not create the toric morphisms (or any morphism of schemes) directly from the `SchemeMorphism...` classes. Instead, use the `hom()` method common to all algebraic schemes to create new homomorphisms.

EXAMPLES:

First, consider the following embedding of $\mathbb{P}^1$ into $\mathbb{P}^2$

```
sage: P2.<x,y,z> = toric_varieties.P2()
sage: P1.<u,v> = toric_varieties.P1()
sage: P1.hom([0, u^2 + v^2, u*v], P2)
Scheme morphism:
  From: 1-d CPR-Fano toric variety covered by 2 affine patches
  To:   2-d CPR-Fano toric variety covered by 3 affine patches
  Defn: Defined on coordinates by sending [u : v] to [0 : u^2 + v^2 : u*v]
```

This is a well-defined morphism of algebraic varieties because homogeneously rescaled coordinates of a point of $\mathbb{P}^1$ map to the same point in $\mathbb{P}^2$ up to its homogeneous rescalings. It is not equivariant with respect to the torus actions

$$\mathbf{C}^\times \times \mathbb{P}^1, (\mu, [u:v]) \mapsto [u:\mu v] \quad \text{and} \quad \left(\mathbf{C}^\times\right)^2 \times \mathbb{P}^2, ((\alpha,\beta),[x:y:z]) \mapsto [x:\alpha y:\beta z],$$

hence it is not a toric morphism. Clearly, the problem is that the map in homogeneous coordinates contains summands that transform differently under the torus action. However, this is not the only difficulty. For example, consider

```
sage: phi = P1.hom([0,u,v], P2);  phi
Scheme morphism:
  From: 1-d CPR-Fano toric variety covered by 2 affine patches
  To:   2-d CPR-Fano toric variety covered by 3 affine patches
  Defn: Defined on coordinates by sending [u : v] to [0 : u : v]
```

This map is actually the embedding of the *orbit_closure()* associated to one of the rays of the fan of $\mathbb{P}^2$. Now the morphism is equivariant with respect to **some** map $\mathbf{C}^\times \to (\mathbf{C}^\times)^2$ of the maximal tori of $\mathbb{P}^1$ and $\mathbb{P}^2$. But this map of the maximal tori cannot be the same as `phi` defined above. Indeed, the image of `phi` completely misses the maximal torus $T_{\mathbb{P}^2} = \{[x:y:z] | x \neq 0, y \neq 0, z \neq 0\}$ of $\mathbb{P}^2$.

Consider instead the following morphism of fans:

```
sage: fm = FanMorphism(matrix(ZZ, [[1,0]]), P1.fan(), P2.fan());  fm
Fan morphism defined by the matrix
[1 0]
Domain fan: Rational polyhedral fan in 1-d lattice N
Codomain fan: Rational polyhedral fan in 2-d lattice N
```

which also defines a morphism of toric varieties:

```
sage: P1.hom(fm, P2)
Scheme morphism:
  From: 1-d CPR-Fano toric variety covered by 2 affine patches
  To:   2-d CPR-Fano toric variety covered by 3 affine patches
  Defn: Defined by sending Rational polyhedral fan in 1-d lattice N
        to Rational polyhedral fan in 2-d lattice N.
```

The fan morphism map is equivalent to the following polynomial map:

```
sage: _.as_polynomial_map()
Scheme morphism:
  From: 1-d CPR-Fano toric variety covered by 2 affine patches
  To:   2-d CPR-Fano toric variety covered by 3 affine patches
  Defn: Defined on coordinates by sending [u : v] to [u : v : v]
```

Finally, here is an example of a fan morphism that cannot be written using homogeneous polynomials. Consider the blowup $O_{\mathbb{P}^1}(2) \to \mathbf{C}^2/\mathbf{Z}_2$. In terms of toric data, this blowup is:

```
sage: A2_Z2 = toric_varieties.A2_Z2()
sage: A2_Z2.fan().rays()
N(1, 0),
N(1, 2)
in 2-d lattice N
sage: O2_P1 = A2_Z2.resolve(new_rays=[(1,1)])
sage: blowup = O2_P1.hom(identity_matrix(2), A2_Z2)
sage: blowup.as_polynomial_map()
Traceback (most recent call last):
...
TypeError: the fan morphism cannot be written in homogeneous polynomials
```

If we denote the homogeneous coordinates of $O_{\mathbb{P}^1}(2)$ by x, t, y corresponding to the rays $(1,2)$, $(1,1)$, and $(1,0)$ then the blow-up map is [BB2013]:

$$f : O_{\mathbb{P}^1}(2) \to \mathbf{C}^2/\mathbf{Z}_2, \quad (x,t,y) \mapsto \left(x\sqrt{t}, y\sqrt{t}\right)$$

which requires square roots.

16.8.1 Fibrations

If a toric morphism is *dominant*, then all fibers over a fixed torus orbit in the base are isomorphic. Hence, studying the fibers is again a combinatorial question and Sage implements additional methods to study such fibrations that are not available otherwise (however, note that you can always *factor()* to pick out the part that is dominant over the image or its closure).

For example, consider the blow-up restricted to one of the two coordinate charts of $O_{\mathbb{P}^1}(2)$

```
sage: O2_P1_chart = ToricVariety(Fan([O2_P1.fan().generating_cones()[0]]))
sage: single_chart = O2_P1_chart.hom(identity_matrix(2), A2_Z2)
sage: single_chart.is_dominant()
True
sage: single_chart.is_surjective()
False

sage: fiber = single_chart.fiber_generic();  fiber
(0-d affine toric variety, 1)
sage: fiber[0].embedding_morphism().as_polynomial_map()
Scheme morphism:
  From: 0-d affine toric variety
  To:   2-d affine toric variety
  Defn: Defined on coordinates by sending [] to [1 : 1]
```

The fibers are labeled by torus orbits in the base, that is, cones of the codomain fan. In this case, the fibers over lower-dimensional torus orbits are:

```
sage: A2_Z2_cones = flatten(A2_Z2.fan().cones())
sage: table([('cone', 'dim')] +
....:       [(cone.ambient_ray_indices(), single_chart.fiber_dimension(cone))
....:        for cone in A2_Z2_cones], header_row=True)
  cone     dim
+--------+-----+
  ()       0
  (0,)     0
  (1,)     -1
  (0, 1)   1
```

Lets look closer at the one-dimensional fiber. Although not the case in this example, connected components of fibers over higher-dimensional cones (corresponding to lower-dimensional torus orbits) of the base are often not irreducible. The irreducible components are labeled by the `primitive_preimage_cones()`, which are certain cones of the domain fan that map to the cone in the base that defines the torus orbit:

```
sage: table([('base cone', 'primitive preimage cones')] +
....:       [(cone.ambient_ray_indices(),
....:         single_chart.fan_morphism().primitive_preimage_cones(cone))
....:        for cone in A2_Z2_cones], header_row=True)
  base cone   primitive preimage cones
+-----------+-------------------------------------------------------------+
  ()          (0-d cone of Rational polyhedral fan in 2-d lattice N,)
  (0,)        (1-d cone of Rational polyhedral fan in 2-d lattice N,)
```

(continues on next page)

(continued from previous page)

```
  (1,)        ()
  (0, 1)      (1-d cone of Rational polyhedral fan in 2-d lattice N,)
```

The fiber over the trivial cone is the generic fiber that we have already encountered. The interesting fiber is the one over the 2-dimensional cone, which represents the exceptional set of the blow-up in this single coordinate chart. Lets investigate further:

```
sage: fm = single_chart.fan_morphism()
sage: exceptional_cones = fm.primitive_preimage_cones(A2_Z2.fan(2)[0])
sage: exceptional_set = single_chart.fiber_component(exceptional_cones[0])
sage: exceptional_set
1-d affine toric variety
sage: exceptional_set.embedding_morphism().as_polynomial_map()
Scheme morphism:
  From: 1-d affine toric variety
  To:   2-d affine toric variety
  Defn: Defined on coordinates by sending [z0] to [z0 : 0]
```

So we see that the fiber over this point is an affine line. Together with another affine line in the other coordinate patch, this covers the exceptional $\mathbb{P}^1$ of the blowup $O_{\mathbb{P}^1}(2) \to \mathbf{C}^2/\mathbf{Z}_2$.

Here is an example with higher dimensional varieties involved:

```
sage: A3 = toric_varieties.A(3)
sage: P3 = toric_varieties.P(3)
sage: m = matrix([(2,0,0), (1,1,0), (3,1,0)])
sage: phi = A3.hom(m, P3)
sage: phi.as_polynomial_map()
Scheme morphism:
  From: 3-d affine toric variety
  To:   3-d CPR-Fano toric variety covered by 4 affine patches
  Defn: Defined on coordinates by sending [z0 : z1 : z2] to
        [z0^2*z1*z2^3 : z1*z2 : 1 : 1]
sage: phi.fiber_generic()
Traceback (most recent call last):
...
AttributeError: 'SchemeMorphism_fan_toric_variety' object
has no attribute 'fiber_generic'...
```

Let's use factorization mentioned above:

```
sage: phi_i, phi_b, phi_s = phi.factor()
```

It is possible to study fibers of the last two morphisms or their composition:

```
sage: phi_d = phi_b * phi_s
sage: phi_d
Scheme morphism:
  From: 3-d affine toric variety
  To:   2-d toric variety covered by 3 affine patches
  Defn: Defined by sending Rational polyhedral fan in 3-d lattice N to
        Rational polyhedral fan in Sublattice <N(1, 0, 0), N(0, 1, 0)>.
sage: phi_d.as_polynomial_map()
```

(continues on next page)

(continued from previous page)

```
Scheme morphism:
  From: 3-d affine toric variety
  To:   2-d toric variety covered by 3 affine patches
  Defn: Defined on coordinates by sending [z0 : z1 : z2] to
        [1 : z1*z2 : z0^2*z1*z2^3]
sage: phi_d.codomain().fan().rays()
N(-1, -1, 0),
N( 0,  1, 0),
N( 1,  0, 0)
in Sublattice <N(1, 0, 0), N(0, 1, 0)>
sage: for c in phi_d.codomain().fan():
....:     c.ambient_ray_indices()
(0, 1)
(0, 2)
(1, 2)
```

We see that codomain fan of this morphism is a projective plane, which can be verified by

```
sage: phi_d.codomain().fan().is_isomorphic(toric_varieties.P2().fan()) # known bug
True
```

(Unfortunately it cannot be verified correctly until github issue #16012 is fixed.)

We now have access to fiber methods:

```
sage: fiber = phi_d.fiber_generic()
sage: fiber
(1-d affine toric variety, 2)
sage: fiber[0].embedding_morphism()
Scheme morphism:
  From: 1-d affine toric variety
  To:   3-d affine toric variety
  Defn: Defined by sending
        Rational polyhedral fan in Sublattice <N(1, 1, -1)> to
        Rational polyhedral fan in 3-d lattice N.
sage: fiber[0].embedding_morphism().as_polynomial_map()
Traceback (most recent call last):
...
NotImplementedError: polynomial representations for
fans with virtual rays are not implemented yet
sage: fiber[0].fan().rays()
Empty collection
in Sublattice <N(1, 1, -1)>
```

We see that generic fibers of this morphism consist of 2 one-dimensional tori each. To see what happens over boundary points we can look at fiber components corresponding to the cones of the domain fan:

```
sage: fm = phi_d.fan_morphism()
sage: for c in flatten(phi_d.domain().fan().cones()):
....:     fc, m = phi_d.fiber_component(c, multiplicity=True)
....:     print("{} |-> {} ({} rays, multiplicity {}) over {}".format(
....:       c.ambient_ray_indices(), fc, fc.fan().nrays(),
....:       m, fm.image_cone(c).ambient_ray_indices()))
```

(continues on next page)

(continued from previous page)

```
() |-> 1-d affine toric variety (0 rays, multiplicity 2) over ()
(0,) |-> 1-d affine toric variety (0 rays, multiplicity 1) over (2,)
(1,) |-> 2-d affine toric variety (2 rays, multiplicity 1) over (1, 2)
(2,) |-> 2-d affine toric variety (2 rays, multiplicity 1) over (1, 2)
(0, 1) |-> 1-d affine toric variety (1 rays, multiplicity 1) over (1, 2)
(1, 2) |-> 1-d affine toric variety (1 rays, multiplicity 1) over (1, 2)
(0, 2) |-> 1-d affine toric variety (1 rays, multiplicity 1) over (1, 2)
(0, 1, 2) |-> 0-d affine toric variety (0 rays, multiplicity 1) over (1, 2)
```

Now we see that over one of the coordinate lines of the projective plane we also have one-dimensional tori (but only one in each fiber), while over one of the points fixed by torus action we have two affine planes intersecting along an affine line. An alternative perspective is provided by cones of the codomain fan:

```
sage: for c in flatten(phi_d.codomain().fan().cones()):
....:     print("{} connected components over {}, each with {} irreducible components.".
↪format(
....:       fm.index(c), c.ambient_ray_indices(),
....:       len(fm.primitive_preimage_cones(c))))
2 connected components over (), each with 1 irreducible components.
None connected components over (0,), each with 0 irreducible components.
None connected components over (1,), each with 0 irreducible components.
1 connected components over (2,), each with 1 irreducible components.
None connected components over (0, 1), each with 0 irreducible components.
None connected components over (0, 2), each with 0 irreducible components.
1 connected components over (1, 2), each with 2 irreducible components.
```

class sage.schemes.toric.morphism.**SchemeMorphism_fan_fiber_component_toric_variety**(*toric_morphism*, *defining_cone*)

Bases: *SchemeMorphism*

The embedding of a fiber component of a toric morphism.

Note that the embedding map of a fiber component of a toric morphism is itself not a toric morphism!

INPUT:

- `toric_morphism` – a toric morphism. The toric morphism whose fiber component we are describing.
- `defining_cone` – a cone of the fan of the domain of `toric_morphism`. See *fiber_component()* for details.

EXAMPLES:

```
sage: polytope = Polyhedron(
....:     [(-3,0,-1,-1),(-1,2,-1,-1),(0,-1,0,0),(0,0,0,1),(0,0,1,0),
....:      (0,1,0,0),(0,2,-1,-1),(1,0,0,0),(2,0,-1,-1)])
sage: coarse_fan = FaceFan(polytope, lattice=ToricLattice(4))
sage: P2 = toric_varieties.P2()
sage: proj24 = matrix([[0,0],[1,0],[0,0],[0,1]])
sage: fm = FanMorphism(proj24, coarse_fan, P2.fan(), subdivide=True)
sage: fibration = ToricVariety(fm.domain_fan()).hom(fm, P2)
sage: ffm = fibration.fan_morphism()
sage: primitive_cones = ffm.primitive_preimage_cones(P2.fan(1)[0])
sage: primitive_cone = primitive_cones[0]
```

(continues on next page)

(continued from previous page)

```
sage: fiber_component = fibration.fiber_component(primitive_cone)
sage: fiber_component
2-d toric variety covered by 4 affine patches
sage: fiber_component.embedding_morphism()
Scheme morphism:
  From: 2-d toric variety covered by 4 affine patches
  To:   4-d toric variety covered by 23 affine patches
  Defn: Defined by embedding a fiber component corresponding to
        1-d cone of Rational polyhedral fan in 4-d lattice N.
sage: fiber_component.embedding_morphism().as_polynomial_map()
Scheme morphism:
  From: 2-d toric variety covered by 4 affine patches
  To:   4-d toric variety covered by 23 affine patches
  Defn: Defined on coordinates by sending [z0 : z1 : z2 : z3] to
        [1 : 1 : 1 : 1 : z2 : 0 : 1 : z3 : 1 : 1 : 1 : z1 : z0 : 1 : 1]
sage: type(fiber_component.embedding_morphism())
<class 'sage.schemes.toric.morphism.SchemeMorphism_fan_fiber_component_toric_variety
↪'>
```

as_polynomial_map()

Express the embedding morphism via homogeneous polynomials.

OUTPUT:

A *SchemeMorphism_polynomial_toric_variety*. Raises a `ValueError` if the morphism cannot be written in terms of homogeneous polynomials.

EXAMPLES:

```
sage: polytope = Polyhedron(
....:     [(-3,0,-1,-1),(-1,2,-1,-1),(0,-1,0,0),(0,0,0,1),(0,0,1,0),
....:      (0,1,0,0),(0,2,-1,-1),(1,0,0,0),(2,0,-1,-1)])
sage: coarse_fan = FaceFan(polytope, lattice=ToricLattice(4))
sage: P2 = toric_varieties.P2()
sage: proj24 = matrix([[0,0], [1,0], [0,0], [0,1]])
sage: fm = FanMorphism(proj24, coarse_fan, P2.fan(), subdivide=True)
sage: fibration = ToricVariety(fm.domain_fan()).hom(fm, P2)

sage: primitive_cone = Cone([(0, 1, 0, 0)])
sage: f = fibration.fiber_component(primitive_cone).embedding_morphism()
sage: f.as_polynomial_map()
Scheme morphism:
  From: 2-d toric variety covered by 4 affine patches
  To:   4-d toric variety covered by 23 affine patches
  Defn: Defined on coordinates by sending [z0 : z1 : z2 : z3] to
        [1 : 1 : 1 : 1 : z2 : 0 : 1 : z3 : 1 : 1 : 1 : z1 : z0 : 1 : 1]

sage: primitive_cone = Cone([(-1, 2, -1, 0)])
sage: f = fibration.fiber_component(primitive_cone).embedding_morphism()
sage: f.as_polynomial_map()
Traceback (most recent call last):
...
ValueError: the morphism cannot be written using homogeneous polynomials
```

base_cone()

Return the base cone σ.

The fiber is constant over the base orbit closure $V(\sigma)$.

OUTPUT:

A cone of the base of the toric fibration.

EXAMPLES:

```
sage: P1xP1 = toric_varieties.P1xP1()
sage: P1 = toric_varieties.P1()
sage: fc = P1xP1.hom(matrix([[1],[0]]), P1).fiber_component(Cone([(1,0)]))
sage: f = fc.embedding_morphism()
sage: f.defining_cone().rays()
N(1, 0)
in 2-d lattice N
sage: f.base_cone().rays()
N(1)
in 1-d lattice N
```

defining_cone()

Return the cone corresponding to the fiber torus orbit.

OUTPUT:

A cone of the fan of the total space of the toric fibration.

EXAMPLES:

```
sage: P1xP1 = toric_varieties.P1xP1()
sage: P1 = toric_varieties.P1()
sage: fc = P1xP1.hom(matrix([[1],[0]]), P1).fiber_component(Cone([(1,0)]))
sage: f = fc.embedding_morphism()
sage: f.defining_cone().rays()
N(1, 0)
in 2-d lattice N
sage: f.base_cone().rays()
N(1)
in 1-d lattice N
```

pullback_divisor(*divisor*)

Pull back a toric divisor.

INPUT:

- `divisor` – a torus-invariant $\mathbb{Q}$-Cartier divisor on the codomain of the embedding map.

OUTPUT:

A divisor on the domain of the embedding map (irreducible component of a fiber of a toric morphism) that is isomorphic to the pull-back divisor $f^*(D)$ but with possibly different linearization.

EXAMPLES:

```
sage: A1 = toric_varieties.A1()
sage: fan = Fan([(0,1,2)], [(1,1,0),(1,0,1),(1,-1,-1)]).subdivide(new_rays=[(1,
↪0,0)])
```

(continues on next page)

(continued from previous page)

```
sage: f = ToricVariety(fan).hom(matrix([[1],[0],[0]]), A1)
sage: D = f.domain().divisor([1,1,3,4]); D
V(z0) + V(z1) + 3*V(z2) + 4*V(z3)
sage: fc = f.fiber_component(Cone([(1,1,0)]))
sage: fc.embedding_morphism().pullback_divisor(D)
4*V(z0) + V(z1) + 4*V(z2)
sage: fc = f.fiber_component(Cone([(1,0,0)]))
sage: fc.embedding_morphism().pullback_divisor(D)
-V(z0) - 3*V(z1) - 3*V(z2)
```

class sage.schemes.toric.morphism.**SchemeMorphism_fan_toric_variety**(*parent*, *fan_morphism*, *check=True*)

Bases: *SchemeMorphism*, Morphism

Construct a morphism determined by a fan morphism

Warning: You should not create objects of this class directly. Use the `hom()` method of *toric varieties* instead.

INPUT:

- `parent` – Hom-set whose domain and codomain are toric varieties.
- `fan_morphism` – A morphism of fans whose domain and codomain fans equal the fans of the domain and codomain in the `parent` Hom-set.
- `check` – boolean (optional, default:`True`). Whether to check the input for consistency.

Warning: A fibration is a dominant morphism; if you are interested in these then you have to make sure that your fan morphism is dominant. For example, this can be achieved by *factoring the morphism*. See *SchemeMorphism_fan_toric_variety_dominant* for additional functionality for fibrations.

OUTPUT: A *SchemeMorphism_fan_toric_variety*.

EXAMPLES:

```
sage: P1xP1 = toric_varieties.P1xP1()
sage: P1 = toric_varieties.P1()
sage: f = P1.hom(matrix([[1,0]]), P1xP1);  f
Scheme morphism:
  From: 1-d CPR-Fano toric variety covered by 2 affine patches
  To:   2-d CPR-Fano toric variety covered by 4 affine patches
  Defn: Defined by sending Rational polyhedral fan in 1-d lattice N
        to Rational polyhedral fan in 2-d lattice N.
sage: type(f)
<class 'sage.schemes.toric.morphism.SchemeMorphism_fan_toric_variety'>
```

Slightly more explicit construction:

```
sage: P1xP1 = toric_varieties.P1xP1()
sage: P1 = toric_varieties.P1()
sage: hom_set = P1xP1.Hom(P1)
```

(continues on next page)

(continued from previous page)

```
sage: fm = FanMorphism(matrix(ZZ, [[1],[0]]), P1xP1.fan(), P1.fan())
sage: hom_set(fm)
Scheme morphism:
  From: 2-d CPR-Fano toric variety covered by 4 affine patches
  To:   1-d CPR-Fano toric variety covered by 2 affine patches
  Defn: Defined by sending Rational polyhedral fan in 2-d lattice N
        to Rational polyhedral fan in 1-d lattice N.

sage: P1xP1.hom(fm, P1)
Scheme morphism:
  From: 2-d CPR-Fano toric variety covered by 4 affine patches
  To:   1-d CPR-Fano toric variety covered by 2 affine patches
  Defn: Defined by sending Rational polyhedral fan in 2-d lattice N
        to Rational polyhedral fan in 1-d lattice N.
```

as_polynomial_map()

Express the morphism via homogeneous polynomials.

OUTPUT: A *SchemeMorphism_polynomial_toric_variety*.

Raises a `TypeError` if the morphism cannot be written in terms of homogeneous polynomials.

EXAMPLES:

```
sage: A1 = toric_varieties.A1()
sage: square = A1.hom(matrix([[2]]), A1)
sage: square.as_polynomial_map()
Scheme endomorphism of 1-d affine toric variety
  Defn: Defined on coordinates by sending [z] to [z^2]

sage: P1 = toric_varieties.P1()
sage: patch = A1.hom(matrix([[1]]), P1)
sage: patch.as_polynomial_map()
Scheme morphism:
  From: 1-d affine toric variety
  To:   1-d CPR-Fano toric variety covered by 2 affine patches
  Defn: Defined on coordinates by sending [z] to [z : 1]
```

factor()

Factor `self` into injective * birational * surjective morphisms.

OUTPUT:

- a triple of toric morphisms (ϕ_i, ϕ_b, ϕ_s), such that ϕ_s is surjective, ϕ_b is birational, ϕ_i is injective, and `self` is equal to $\phi_i \circ \phi_b \circ \phi_s$.

The intermediate varieties are universal in the following sense. Let `self` map X to X' and let X_s, X_i sit in between, that is,

$$X \twoheadrightarrow X_s \to X_i \hookrightarrow X'.$$

Then any toric morphism from X coinciding with `self` on the maximal torus factors through X_s and any toric morphism into X' coinciding with `self` on the maximal torus factors through X_i. In particular, X_i is the closure of the image of `self` in X'.

See `factor()` for a description of the toric algorithm.

EXAMPLES:

We map an affine plane into a projective 3-space in such a way, that it becomes "a double cover of a chart of the blow up of one of the coordinate planes":

```
sage: A2 = toric_varieties.A2()
sage: P3 = toric_varieties.P(3)
sage: m = matrix([(2,0,0), (1,1,0)])
sage: phi = A2.hom(m, P3)
sage: phi.as_polynomial_map()
Scheme morphism:
  From: 2-d affine toric variety
  To:   3-d CPR-Fano toric variety covered by 4 affine patches
  Defn: Defined on coordinates by sending [x : y] to
        [x^2*y : y : 1 : 1]

sage: phi.is_surjective(), phi.is_birational(), phi.is_injective()
(False, False, False)
sage: phi_i, phi_b, phi_s = phi.factor()
sage: phi_s.is_surjective(), phi_b.is_birational(), phi_i.is_injective()
(True, True, True)
sage: prod(phi.factor()) == phi
True
```

Double cover (surjective):

```
sage: phi_s.as_polynomial_map()
Scheme morphism:
  From: 2-d affine toric variety
  To:   2-d affine toric variety
  Defn: Defined on coordinates by sending [x : y] to [x^2 : y]
```

Blowup chart (birational):

```
sage: phi_b.as_polynomial_map()
Scheme morphism:
  From: 2-d affine toric variety
  To:   2-d toric variety covered by 3 affine patches
  Defn: Defined on coordinates by sending [z0 : z1] to [1 : z1 : z0*z1]
```

Coordinate plane inclusion (injective):

```
sage: phi_i.as_polynomial_map()
Scheme morphism:
  From: 2-d toric variety covered by 3 affine patches
  To:   3-d CPR-Fano toric variety covered by 4 affine patches
  Defn: Defined on coordinates by sending [z0 : z1 : z2] to [z2 : z1 : z0 : z0]
```

fan_morphism()

Return the defining fan morphism.

OUTPUT: A FanMorphism.

EXAMPLES:

```
sage: P1xP1 = toric_varieties.P1xP1()
sage: P1 = toric_varieties.P1()
sage: f = P1xP1.hom(matrix([[1],[0]]), P1)
sage: f.fan_morphism()
Fan morphism defined by the matrix
[1]
[0]
Domain fan: Rational polyhedral fan in 2-d lattice N
Codomain fan: Rational polyhedral fan in 1-d lattice N
```

is_birational()

Check if `self` is birational.

See `is_birational()` for fan morphisms for a description of the toric algorithm.

OUTPUT:

Boolean. Whether `self` is birational.

EXAMPLES:

```
sage: dP8 = toric_varieties.dP8()
sage: P2 = toric_varieties.P2()
sage: dP8.hom(identity_matrix(2), P2).is_birational()
True

sage: X = toric_varieties.A(2)
sage: Y = ToricVariety(Fan([Cone([(1,0), (1,1)])]))
sage: m = identity_matrix(2)
sage: f = Y.hom(m, X)
sage: f.is_birational()
True
```

is_bundle()

Check if `self` is a bundle.

See `is_bundle()` for fan morphisms for details.

OUTPUT:

- `True` if `self` is a bundle, `False` otherwise.

EXAMPLES:

```
sage: P1xP1 = toric_varieties.P1xP1()
sage: P1 = toric_varieties.P1()
sage: P1xP1.hom(matrix([[1],[0]]), P1).is_bundle()
True
```

is_dominant()

Return whether `self` is dominant.

See `is_dominant()` for fan morphisms for a description of the toric algorithm.

OUTPUT:

Boolean. Whether `self` is a dominant scheme morphism.

EXAMPLES:

```
sage: P1 = toric_varieties.P1()
sage: A1 = toric_varieties.A1()
sage: phi = A1.hom(identity_matrix(1), P1);  phi
Scheme morphism:
  From: 1-d affine toric variety
  To:   1-d CPR-Fano toric variety covered by 2 affine patches
  Defn: Defined by sending Rational polyhedral fan in 1-d lattice N
        to Rational polyhedral fan in 1-d lattice N.
sage: phi.is_dominant()
True
sage: phi.is_surjective()
False
```

is_fibration()

Check if `self` is a fibration.

See `is_fibration()` for fan morphisms for details.

OUTPUT:

- `True` if `self` is a fibration, `False` otherwise.

EXAMPLES:

```
sage: P1xP1 = toric_varieties.P1xP1()
sage: P1 = toric_varieties.P1()
sage: P1xP1.hom(matrix([[1],[0]]), P1).is_fibration()
True
```

is_injective()

Check if `self` is injective.

See `is_injective()` for fan morphisms for a description of the toric algorithm.

OUTPUT:

Boolean. Whether `self` is injective.

EXAMPLES:

```
sage: P1xP1 = toric_varieties.P1xP1()
sage: P1 = toric_varieties.P1()
sage: P1xP1.hom(matrix([[1],[0]]), P1).is_injective()
False

sage: X = toric_varieties.A(2)
sage: m = identity_matrix(2)
sage: f = X.hom(m, X)
sage: f.is_injective()
True

sage: Y = ToricVariety(Fan([Cone([(1,0), (1,1)])]))
sage: f = Y.hom(m, X)
sage: f.is_injective()
False
```

is_surjective()

Check if `self` is surjective.

See `is_surjective()` for fan morphisms for a description of the toric algorithm.

OUTPUT:

Boolean. Whether `self` is surjective.

EXAMPLES:

```
sage: P1xP1 = toric_varieties.P1xP1()
sage: P1 = toric_varieties.P1()
sage: P1xP1.hom(matrix([[1],[0]]), P1).is_surjective()
True

sage: X = toric_varieties.A(2)
sage: m = identity_matrix(2)
sage: f = X.hom(m, X)
sage: f.is_surjective()
True

sage: Y = ToricVariety(Fan([Cone([(1,0), (1,1)])]))
sage: f = Y.hom(m, X)
sage: f.is_surjective()
False
```

pullback_divisor(*divisor*)

Pull back a toric divisor.

INPUT:

- `divisor` – a torus-invariant $\mathbb{Q}$-Cartier divisor on the codomain of `self`.

OUTPUT:

The pull-back divisor $f^*(D)$.

EXAMPLES:

```
sage: A2_Z2 = toric_varieties.A2_Z2()
sage: A2 = toric_varieties.A2()
sage: f = A2.hom(matrix([[1,0], [1,2]]), A2_Z2)
sage: f.pullback_divisor(A2_Z2.divisor(0))
V(x)

sage: A1 = toric_varieties.A1()
sage: square = A1.hom(matrix([[2]]), A1)
sage: D = A1.divisor(0);  D
V(z)
sage: square.pullback_divisor(D)
2*V(z)
```

class sage.schemes.toric.morphism.**SchemeMorphism_fan_toric_variety_dominant**(*parent*, *fan_morphism*, *check=True*)

Bases: *SchemeMorphism_fan_toric_variety*

Construct a morphism determined by a dominant fan morphism.

A dominant morphism is one that is surjective onto a dense subset. In the context of toric morphisms, this means that it is onto the big torus orbit.

Warning: You should not create objects of this class directly. Use the `hom()` method of *toric varieties* instead.

INPUT:

See *SchemeMorphism_fan_toric_variety*. The given fan morphism must be dominant.

OUTPUT: A *SchemeMorphism_fan_toric_variety_dominant*.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: dP8 = toric_varieties.dP8()
sage: f = dP8.hom(identity_matrix(2), P2);  f
Scheme morphism:
  From: 2-d CPR-Fano toric variety covered by 4 affine patches
  To:   2-d CPR-Fano toric variety covered by 3 affine patches
  Defn: Defined by sending Rational polyhedral fan in 2-d lattice N
        to Rational polyhedral fan in 2-d lattice N.
sage: type(f)
<class 'sage.schemes.toric.morphism.SchemeMorphism_fan_toric_variety_dominant'>
```

fiber_component(*domain_cone*, *multiplicity=False*)

Return a fiber component corresponding to `domain_cone`.

INPUT:

- `domain_cone` – a cone of the domain fan of `self`.
- `multiplicity` (default: `False`) – whether to return the number of fiber components corresponding to `domain_cone` as well.

OUTPUT:

- either X or a tuple (X, n), where X is a *toric variety* with the embedding morphism into domain of `self` and n is an integer.

Let $\phi : \Sigma \to \Sigma'$ be the fan morphism corresponding to `self`. Let $\sigma \in \Sigma$ and $\sigma' \in \Sigma'$ be the `image_cone()` of σ. The fiber over any point of the torus orbit corresponding to σ' consists of n isomorphic connected components with each component being a union of toric varieties intersecting along their torus invariant subvarieties. The latter correspond to `preimage_cones()` of σ' and X is one of the n components corresponding to σ. The irreducible components correspond to `primitive_preimage_cones()`.

EXAMPLES:

```
sage: polytope = LatticePolytope(
....:     [(-3,0,-1,-1),(-1,2,-1,-1),(0,-1,0,0),(0,0,0,1),(0,0,1,0),
....:      (0,1,0,0),(0,2,-1,-1),(1,0,0,0),(2,0,-1,-1)])
sage: coarse_fan = FaceFan(polytope)
sage: P2 = toric_varieties.P2()
sage: proj24 = matrix([[0,0], [1,0], [0,0], [0,1]])
sage: fm = FanMorphism(proj24, coarse_fan, P2.fan(), subdivide=True)
sage: fibration = ToricVariety(fm.domain_fan()).hom(fm, P2)
sage: ffm = fibration.fan_morphism()
```

(continues on next page)

(continued from previous page)

```
sage: primitive_cones = ffm.primitive_preimage_cones(P2.fan(1)[0])
sage: primitive_cone = primitive_cones[0]
sage: fibration.fiber_component(primitive_cone)
2-d toric variety covered by 4 affine patches
sage: fibration.fiber_component(primitive_cone, True)
(2-d toric variety covered by 4 affine patches, 1)

sage: for primitive_cone in primitive_cones:
....:     print(fibration.fiber_component(primitive_cone))
2-d toric variety covered by 4 affine patches
2-d toric variety covered by 3 affine patches
2-d toric variety covered by 3 affine patches
```

fiber_dimension(*codomain_cone*)

Return the dimension of the fiber over a particular torus orbit in the base.

INPUT:

- `codomain_cone` – a cone σ of the codomain, specifying a torus orbit $O(\sigma)$.

OUTPUT:

An integer. The dimension of the fiber over the torus orbit corresponding to `codomain_cone`. If the fiber is the empty set, `-1` is returned. Note that all fibers over this torus orbit are isomorphic, and therefore have the same dimension.

EXAMPLES:

```
sage: P1xP1 = toric_varieties.P1xP1()
sage: P1 = toric_varieties.P1()
sage: f = P1xP1.hom(matrix([[1],[0]]), P1)
sage: f.fiber_dimension(P1.fan(0)[0])
1
sage: f.fiber_dimension(P1.fan(1)[0])
1
sage: f.fiber_dimension(P1.fan(1)[1])
1
```

Here is a more complicated example that is not a flat fibration:

```
sage: A2_Z2 = toric_varieties.A2_Z2()
sage: O2_P1 = A2_Z2.resolve(new_rays=[(1,1)])
sage: blowup = O2_P1.hom(identity_matrix(2), A2_Z2)
sage: blowup.fiber_dimension(A2_Z2.fan(0)[0])
0
sage: blowup.fiber_dimension(A2_Z2.fan(1)[0])
0
sage: blowup.fiber_dimension(A2_Z2.fan(2)[0])
1
```

This corresponds to the three different fibers:

```
sage: blowup.fiber_generic()
(0-d affine toric variety, 1)
sage: blowup.fiber_component(Cone([(1,0)]))
```

(continues on next page)

(continued from previous page)

```
0-d affine toric variety
sage: blowup.fiber_component(Cone([(1,1)]))
1-d toric variety covered by 2 affine patches
```

fiber_generic()

Return the generic fiber.

OUTPUT:

- a tuple (X, n), where X is a *toric variety* with the embedding morphism into domain of `self` and n is an integer.

The fiber over the base point with homogeneous coordinates $[1 : 1 : \cdots : 1]$ consists of n disjoint toric varieties isomorphic to X. Note that fibers of a dominant toric morphism are isomorphic over all points of a fixed torus orbit of its codomain, in particular over all points of the maximal torus, so it makes sense to talk about "the generic" fiber.

The embedding of X is a toric morphism with the `domain_fan()` being the `kernel_fan()` of the defining fan morphism. By contrast, embeddings of fiber components over lower-dimensional torus orbits of the image are not toric morphisms. Use *fiber_component()* for the latter (non-generic) fibers.

EXAMPLES:

```
sage: P1xP1 = toric_varieties.P1xP1()
sage: P1 = toric_varieties.P1()
sage: fiber = P1xP1.hom(matrix([[1],[0]]), P1).fiber_generic()
sage: fiber
(1-d toric variety covered by 2 affine patches, 1)
sage: f = fiber[0].embedding_morphism();  f
Scheme morphism:
  From: 1-d toric variety covered by 2 affine patches
  To:   2-d CPR-Fano toric variety covered by 4 affine patches
  Defn: Defined by sending Rational polyhedral fan in Sublattice <N(0, 1)> to
        Rational polyhedral fan in 2-d lattice N.
sage: f.as_polynomial_map()
Scheme morphism:
  From: 1-d toric variety covered by 2 affine patches
  To:   2-d CPR-Fano toric variety covered by 4 affine patches
  Defn: Defined on coordinates by sending [z0 : z1] to [1 : 1 : z0 : z1]

sage: A1 = toric_varieties.A1()
sage: fan = Fan([(0,1,2)], [(1,1,0), (1,0,1), (1,-1,-1)])
sage: fan = fan.subdivide(new_rays=[(1,0,0)])
sage: f = ToricVariety(fan).hom(matrix([[1],[0],[0]]), A1)
sage: f.fiber_generic()
(2-d affine toric variety, 1)
sage: _[0].fan().generating_cones()
(0-d cone of Rational polyhedral fan in Sublattice <N(0, 1, 0), N(0, 0, 1)>,)
```

fiber_graph(*codomain_cone*)

Return the fiber over a given torus orbit in the codomain.

INPUT:

- `codomain_cone` – a cone σ of the codomain, specifying a torus orbit $O(\sigma)$.

OUTPUT:

A graph whose nodes are the irreducible components of a connected component of the fiber over a point of $O(\sigma)$. If two irreducible components intersect, the corresponding nodes of the graph are joined by an edge. Note that irreducible components do not have to be of the same dimension.

See also:

fiber_component().

EXAMPLES:

```
sage: polytope = Polyhedron(
....:     [(-3,0,-1,-1),(-1,2,-1,-1),(0,-1,0,0),(0,0,0,1),(0,0,1,0),
....:      (0,1,0,0),(0,2,-1,-1),(1,0,0,0),(2,0,-1,-1)])
sage: coarse_fan = FaceFan(polytope, lattice=ToricLattice(4))

sage: P2 = toric_varieties.P2()
sage: proj34 = block_matrix(2, 1, [zero_matrix(2,2),
....:                              identity_matrix(2)])
sage: fm = FanMorphism(proj34, coarse_fan, P2.fan(), subdivide=True)
sage: fibration = ToricVariety(fm.domain_fan()).hom(fm, P2)

sage: fibration.fiber_graph(P2.fan(0)[0])
Graph on 1 vertex
sage: for c1 in P2.fan(1):
....:     fibration.fiber_graph(c1)
Graph on 1 vertex
Graph on 1 vertex
Graph on 4 vertices

sage: fibration.fiber_graph(P2.fan(1)[2]).get_vertices()
{0: 2-d toric variety covered by 4 affine patches,
 1: 2-d toric variety covered by 3 affine patches,
 2: 2-d toric variety covered by 3 affine patches,
 3: 2-d toric variety covered by 4 affine patches}

sage: fibration
Scheme morphism:
  From: 4-d toric variety covered by 18 affine patches
  To:   2-d CPR-Fano toric variety covered by 3 affine patches
  Defn: Defined by sending Rational polyhedral fan in 4-d lattice N
        to Rational polyhedral fan in 2-d lattice N.
```

class sage.schemes.toric.morphism.**SchemeMorphism_orbit_closure_toric_variety**(*parent*, *defining_cone*, *ray_map*)

Bases: *SchemeMorphism*, Morphism

The embedding of an orbit closure.

INPUT:

- `parent` – the parent homset.
- `defining_cone` – the defining cone.

- ray_map – a dictionary {ambient ray generator: orbit ray generator}. Note that the image of the ambient ray generator is not necessarily primitive.

Warning: You should not create objects of this class directly. Use the *orbit_closure()* method of *toric varieties* instead.

EXAMPLES:

```
sage: P1xP1 = toric_varieties.P1xP1()
sage: H = P1xP1.fan(1)[0]
sage: V = P1xP1.orbit_closure(H)
sage: V.embedding_morphism()
Scheme morphism:
  From: 1-d toric variety covered by 2 affine patches
  To:   2-d CPR-Fano toric variety covered by 4 affine patches
  Defn: Defined by embedding the torus closure associated to the 1-d
        cone of Rational polyhedral fan in 2-d lattice N.
```

as_polynomial_map()

Express the morphism via homogeneous polynomials.

OUTPUT:

A *SchemeMorphism_polynomial_toric_variety*. Raises a TypeError if the morphism cannot be written in terms of homogeneous polynomials.

The defining polynomials are not necessarily unique. There are choices if multiple ambient space ray generators project to the same orbit ray generator, and one such choice is made implicitly. The orbit embedding can be written as a polynomial map if and only if each primitive orbit ray generator is the image of at least one primitive ray generator of the ambient toric variety.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: V = P2.orbit_closure(P2.fan(1)[0]);  V
1-d toric variety covered by 2 affine patches
sage: V.embedding_morphism().as_polynomial_map()
Scheme morphism:
  From: 1-d toric variety covered by 2 affine patches
  To:   2-d CPR-Fano toric variety covered by 3 affine patches
  Defn: Defined on coordinates by sending [z0 : z1] to [0 : z1 : z0]
```

If the toric variety is singular, then some orbit closure embeddings cannot be written with homogeneous polynomials:

```
sage: P2_112 = toric_varieties.P2_112()
sage: P1 = P2_112.orbit_closure(Cone([(1,0)]))
sage: P1.embedding_morphism().as_polynomial_map()
Traceback (most recent call last):
...
TypeError: the embedding cannot be written with homogeneous polynomials
```

defining_cone()

Return the cone corresponding to the torus orbit.

OUTPUT: A cone of the fan of the ambient toric variety.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: cone = P2.fan(1)[0]
sage: P1 = P2.orbit_closure(cone)
sage: P1.embedding_morphism().defining_cone()
1-d cone of Rational polyhedral fan in 2-d lattice N
sage: _ is cone
True
```

pullback_divisor(*divisor*)

Pull back a toric divisor.

INPUT:

- `divisor` – a torus-invariant $\mathbb{Q}$-Cartier divisor on the codomain of the embedding map.

OUTPUT:

A divisor on the domain of the embedding map (the orbit closure) that is isomorphic to the pull-back divisor $f^*(D)$ but with possibly different linearization.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: P1 = P2.orbit_closure(P2.fan(1)[0])
sage: f = P1.embedding_morphism()
sage: D = P2.divisor([1,2,3]); D
V(x) + 2*V(y) + 3*V(z)
sage: f.pullback_divisor(D)
4*V(z0) + 2*V(z1)
```

class sage.schemes.toric.morphism.**SchemeMorphism_point_toric_field**(*X*, *coordinates*, *check=True*)

Bases: *SchemeMorphism_point*, Morphism

A point of a toric variety determined by homogeneous coordinates in a field.

Warning: You should not create objects of this class directly. Use the `hom()` method of *toric varieties* instead.

INPUT:

- `X` – toric variety or subscheme of a toric variety.
- `coordinates` – list of coordinates in the base field of `X`.
- `check` – if `True` (default), the input will be checked for correctness.

OUTPUT: A *SchemeMorphism_point_toric_field*.

class sage.schemes.toric.morphism.**SchemeMorphism_polynomial_toric_variety**(*parent*, *polynomials*, *check=True*)

Bases: *SchemeMorphism_polynomial*, Morphism

A morphism determined by homogeneous polynomials.

Warning: You should not create objects of this class directly. Use the `hom()` method of *toric varieties* instead.

INPUT:

Same as for `SchemeMorphism_polynomial`.

OUTPUT: A *SchemeMorphism_polynomial_toric_variety*.

as_fan_morphism()

Express the morphism as a map defined by a fan morphism.

OUTPUT: A *SchemeMorphism_polynomial_toric_variety*.

Raises a `TypeError` if the morphism cannot be written in such a way.

EXAMPLES:

```
sage: A1.<z> = toric_varieties.A1()
sage: P1 = toric_varieties.P1()
sage: patch = A1.hom([1,z], P1)
sage: patch.as_fan_morphism()
Traceback (most recent call last):
...
NotImplementedError: expressing toric morphisms as fan morphisms is
not implemented yet
```

16.9 Subschemes of toric space

AUTHORS:

- David Kohel (2005): initial version.
- William Stein (2005): initial version.
- Andrey Novoseltsev (2010-05-17): subschemes of toric varieties.

class sage.schemes.toric.toric_subscheme.**AlgebraicScheme_subscheme_affine_toric**(*toric_variety*, *polynomials*)

Bases: *AlgebraicScheme_subscheme_toric*

Construct an algebraic subscheme of an affine toric variety.

Warning: You should not create objects of this class directly. The preferred method to construct such subschemes is to use `subscheme()` method of `toric varieties`.

INPUT:

- `toric_variety` – ambient `affine toric variety`;
- `polynomials` – single polynomial, list, or ideal of defining polynomials in the coordinate ring of `toric_variety`.

OUTPUT:

A *algebraic subscheme of an affine toric variety*.

dimension()

Return the dimension of `self`.

OUTPUT: An integer.

EXAMPLES:

```
sage: # needs sage.libs.singular
sage: P1xP1.<s0,s1,t0,t1> = toric_varieties.P1xP1()
sage: P1 = P1xP1.subscheme(s0 - s1)
sage: P1.dimension()
1
```

A more complicated example where the ambient toric variety is not smooth:

```
sage: # needs sage.libs.singular
sage: X.<x,y> = toric_varieties.A2_Z2()
sage: X.is_smooth()
False
sage: Y = X.subscheme([x*y, x^2]); Y
Closed subscheme of 2-d affine toric variety defined by:
  x*y,
  x^2
sage: Y.dimension()
1
```

is_smooth(*point=None*)

Test whether the algebraic subscheme is smooth.

INPUT:

- `point` – A point or `None` (default). The point to test smoothness at.

OUTPUT:

Boolean. If no point was specified, returns whether the algebraic subscheme is smooth everywhere. Otherwise, smoothness at the specified point is tested.

EXAMPLES:

```
sage: # needs sage.libs.singular
sage: A2.<x,y> = toric_varieties.A2()
sage: cuspidal_curve = A2.subscheme([y^2 - x^3])
sage: cuspidal_curve
Closed subscheme of 2-d affine toric variety defined by:
  -x^3 + y^2
sage: cuspidal_curve.is_smooth([1,1])
True
sage: cuspidal_curve.is_smooth([0,0])
False
sage: cuspidal_curve.is_smooth()
False
sage: circle = A2.subscheme(x^2 + y^2 - 1)
sage: circle.is_smooth([1,0])
True
sage: circle.is_smooth()
True
```

A more complicated example where the ambient toric variety is not smooth:

```
sage: # needs sage.libs.singular
sage: X.<x,y> = toric_varieties.A2_Z2()    # 2-d affine space mod Z/2
sage: X.is_smooth()
False
sage: Y = X.subscheme([x*y, x^2])   # (twice the x=0 curve) mod Z/2
sage: Y
Closed subscheme of 2-d affine toric variety defined by:
  x*y,
  x^2
sage: Y.dimension()   # Y is a Weil divisor but not Cartier
1
sage: Y.is_smooth()
True
sage: Y.is_smooth([0,0])
True
```

class sage.schemes.toric.toric_subscheme.**AlgebraicScheme_subscheme_toric**(*toric_variety*, *polynomials*)

Bases: *AlgebraicScheme_subscheme*

Construct an algebraic subscheme of a toric variety.

Warning: You should not create objects of this class directly. The preferred method to construct such subschemes is to use `subscheme()` method of *toric varieties*.

INPUT:

- `toric_variety` – ambient `toric variety`.
- `polynomials` – single polynomial, list, or ideal of defining polynomials in the coordinate ring of `toric_variety`.

OUTPUT: An *algebraic subscheme of a toric variety*.

affine_algebraic_patch(*cone=None*, *names=None*)

Return the affine patch corresponding to `cone` as an affine algebraic scheme.

INPUT:

- `cone` – a Cone σ of the fan. It can be omitted for an affine toric variety, in which case the single generating cone is used.

OUTPUT:

An *affine algebraic subscheme* corresponding to the patch $Spec(\sigma^\vee \cap M)$ associated to the cone σ.

See also *affine_patch()*, which expresses the patches as subvarieties of affine toric varieties instead.

REFERENCES:

David A. Cox, "The Homogeneous Coordinate Ring of a Toric Variety", Lemma 2.2. arXiv alg-geom/9210008v2

EXAMPLES:

```
sage: P2.<x,y,z> = toric_varieties.P2()
sage: cone = P2.fan().generating_cone(0)
sage: V = P2.subscheme(x^3 + y^3 + z^3)
sage: V.affine_algebraic_patch(cone)
Closed subscheme of Affine Space of dimension 2 over Rational Field defined by:
  z0^3 + z1^3 + 1

sage: # needs fpylll sage.libs.singular
sage: cone = Cone([(0,1), (2,1)])
sage: A2Z2.<x,y> = AffineToricVariety(cone)
sage: A2Z2.affine_algebraic_patch()
Closed subscheme of Affine Space of dimension 3 over Rational Field defined by:
  -z0*z1 + z2^2
sage: V = A2Z2.subscheme(x^2 + y^2 - 1)
sage: patch = V.affine_algebraic_patch();  patch
Closed subscheme of Affine Space of dimension 3 over Rational Field defined by:
  -z0*z1 + z2^2,
  z0 + z1 - 1
sage: nbhd_patch = V.neighborhood([1,0]).affine_algebraic_patch();  nbhd_patch
Closed subscheme of Affine Space of dimension 3 over Rational Field defined by:
  -z0*z1 + z2^2,
  z0 + z1 - 1
sage: nbhd_patch.embedding_center()
(0, 1, 0)
```

Here we got two defining equations. The first one describes the singularity of the ambient space and the second is the pull-back of $x^2 + y^2 - 1$

```
sage: lp = LatticePolytope([(1,0,0), (1,1,0), (1,1,1), (1,0,1), (-2,-1,-1)],
....:                      lattice=ToricLattice(3))
sage: X.<x,y,u,v,t> = CPRFanoToricVariety(Delta_polar=lp)
sage: Y = X.subscheme(x*v + y*u + t)
sage: cone = Cone([(1,0,0), (1,1,0), (1,1,1), (1,0,1)])
sage: Y.affine_algebraic_patch(cone)
Closed subscheme of Affine Space of dimension 4 over Rational Field defined by:
  z0*z2 - z1*z3,
  z1 + z3 + 1
```

affine_patch(i)

Return the `i`-th affine patch of `self` as an affine toric algebraic scheme.

INPUT:

- `i` – integer, index of a generating cone of the fan of the ambient space of `self`.

OUTPUT:

- subscheme of an affine *toric variety* corresponding to the pull-back of `self` by the embedding morphism of the `i`-th *affine patch of the ambient space* of `self`.

The result is cached, so the `i`-th patch is always the same object in memory.

EXAMPLES:

```
sage: P1xP1 = toric_varieties.P1xP1()
sage: patch1 = P1xP1.affine_patch(1)
```

(continues on next page)

(continued from previous page)

```
sage: patch1.embedding_morphism()
Scheme morphism:
  From: 2-d affine toric variety
  To:   2-d CPR-Fano toric variety covered by 4 affine patches
  Defn: Defined on coordinates by sending [t : x] to [1 : t : x : 1]
sage: P1xP1.inject_variables()
Defining s, t, x, y
sage: P1 = P1xP1.subscheme(x - y)
sage: subpatch = P1.affine_patch(1)
sage: subpatch
Closed subscheme of 2-d affine toric variety defined by:
  x - 1
```

dimension()

Return the dimension of `self`.

OUTPUT: An integer. If `self` is empty, -1 is returned.

EXAMPLES:

```
sage: # needs sage.libs.singular
sage: P1xP1 = toric_varieties.P1xP1()
sage: P1xP1.inject_variables()
Defining s, t, x, y
sage: P1 = P1xP1.subscheme(s - t)
sage: P1.dimension()
1
sage: P1xP1.subscheme([s - t, (s-t)^2]).dimension()
1
sage: P1xP1.subscheme([s, t]).dimension()
-1
```

fan()

Return the fan of the ambient space.

OUTPUT: A fan.

EXAMPLES:

```
sage: P2.<x,y,z> = toric_varieties.P(2)
sage: E = P2.subscheme([x^2 + y^2 + z^2])
sage: E.fan()
Rational polyhedral fan in 2-d lattice N
```

is_nondegenerate()

Check if `self` is nondegenerate.

OUTPUT:

Whether the variety is nondegenerate, that is, the intersection with every open torus orbit is smooth and transversal.

EXAMPLES:

```
sage: P2.<x,y,z> = toric_varieties.P2()
sage: P2.subscheme([x^3 + y^3 + z^3]).is_nondegenerate()
True
sage: P2.subscheme([x*y*z]).is_nondegenerate()
False
sage: X = P2.subscheme([(x-y)^2*(x+y) + x*y*z + z^3])
sage: X.is_smooth()
True
sage: X.is_nondegenerate()
False
```

A K3 surface in $\mathbf{P}^1 \times \mathbf{P}^1 \times \mathbf{P}^1$:

```
sage: diamond = lattice_polytope.cross_polytope(3)
sage: fan = FaceFan(diamond)
sage: P1xP1xP1 = ToricVariety(fan)
sage: z0, z1, z2, z3, z4, z5 = P1xP1xP1.gens()
sage: t = 5
sage: F = z0^2*z1^2*z2^2 + z1^2*z2^2*z3^2 + z0^2*z2^2*z4^2\
....: + z2^2*z3^2*z4^2 + t*z0*z1*z2*z3*z4*z5 + z0^2*z1^2*z5^2\
....: + z1^2*z3^2*z5^2 + z0^2*z4^2*z5^2 + z3^2*z4^2*z5^2
sage: X = P1xP1xP1.subscheme([F])
sage: X.is_smooth()
True
sage: X.is_nondegenerate()
False
```

Taking a random change of variables breaks the symmetry, but makes the surface nondegenerate:

```
sage: F1 = F.subs(z0=1*z0 + 1*z3, z3=1*z0 + 2*z3,
....:             z1=-2*z1 + -1*z4, z4=1*z1 + 2*z4,
....:             z2=-3*z2 + -1*z5, z5=-3*z2 + 2*z5)
sage: Y = P1xP1xP1.subscheme([F1])
sage: Y.is_smooth()
True
sage: Y.is_nondegenerate()
True
```

This example is from Hamm, arXiv 1106.1826v1. It addresses an issue raised at github issue #15239:

```
sage: X = toric_varieties.WP([1,4,2,3], names='z0 z1 z2 z3')
sage: X.inject_variables()
Defining z0, z1, z2, z3
sage: g0 = z1^3 + z2^6 + z3^4
sage: g = g0 - 2*z3^2*z0^6 + z2*z0^10 + z0^12
sage: Y = X.subscheme([g])
sage: Y.is_nondegenerate()
False
```

It handles nonzero characteristic:

```
sage: P2.<x,y,z> = toric_varieties.P2()
sage: f = x^5 + 2*x*y^4 + y^5 - 2*y^3*z^2 + x*z^4 - 2*z^5
```

(continues on next page)

(continued from previous page)

```
sage: P2.change_ring(GF(5)).subscheme([f]).is_nondegenerate()
True
sage: P2.change_ring(GF(7)).subscheme([f]).is_nondegenerate()
False
```

is_schon()

Check if `self` is schon (nondegenerate).

See *is_nondegenerate()* for further documentation.

EXAMPLES:

```
sage: # needs sage.libs.singular
sage: P2.<x,y,z> = toric_varieties.P2()
sage: X = P2.subscheme([(x-y)^2*(x+y) + x*y*z + z^3])
sage: X.is_smooth()
True
sage: X.is_schon()
False
```

is_smooth(*point=None*)

Test whether the algebraic subscheme is smooth.

INPUT:

- `point` – A point or `None` (default). The point to test smoothness at.

OUTPUT:

Boolean. If no point was specified, returns whether the algebraic subscheme is smooth everywhere. Otherwise, smoothness at the specified point is tested.

EXAMPLES:

```
sage: # needs sage.libs.singular
sage: P2.<x,y,z> = toric_varieties.P2()
sage: cuspidal_curve = P2.subscheme([y^2*z - x^3])
sage: cuspidal_curve
Closed subscheme of 2-d CPR-Fano toric variety covered by 3 affine patches␣
↪defined by:
  -x^3 + y^2*z
sage: cuspidal_curve.is_smooth([1,1,1])
True
sage: cuspidal_curve.is_smooth([0,0,1])
False
sage: cuspidal_curve.is_smooth()
False
```

Any sufficiently generic cubic hypersurface is smooth:

```
sage: P2.subscheme([y^2*z-x^3+z^3+1/10*x*y*z]).is_smooth()                    #␣
↪needs sage.libs.singular
True
```

A more complicated example:

```
sage: # needs sage.libs.singular
sage: dP6.<x0,x1,x2,x3,x4,x5> = toric_varieties.dP6()
sage: disjointP1s = dP6.subscheme(x0*x3)
sage: disjointP1s.is_smooth()
True
sage: intersectingP1s = dP6.subscheme(x0*x1)
sage: intersectingP1s.is_smooth()
False
```

A smooth hypersurface in a compact singular toric variety:

```
sage: # needs sage.libs.singular
sage: lp = LatticePolytope([(1,0,0), (1,1,0), (1,1,1), (1,0,1), (-2,-1,-1)],
....:                      lattice=ToricLattice(3))
sage: X.<x,y,u,v,t> = CPRFanoToricVariety(Delta_polar=lp)
sage: Y = X.subscheme(x*v + y*u + t)
sage: cone = Cone([(1,0,0), (1,1,0), (1,1,1), (1,0,1)])
sage: Y.is_smooth()
True
```

neighborhood(*point*)

Return an toric algebraic scheme isomorphic to neighborhood of the `point`.

INPUT:

- `point` – a point of the toric algebraic scheme.

OUTPUT:

An affine toric algebraic scheme (polynomial equations in an affine toric variety) with fixed `embedding_morphism()` and `embedding_center()`.

EXAMPLES:

```
sage: # needs sage.libs.singular
sage: P.<x,y,z> = toric_varieties.P2()
sage: S = P.subscheme(x + 2*y + 3*z)
sage: s = S.point([0,-3,2]); s
[0 : -3 : 2]
sage: patch = S.neighborhood(s); patch
Closed subscheme of 2-d affine toric variety defined by:
  x + 2*y + 6
sage: patch.embedding_morphism()
Scheme morphism:
  From: Closed subscheme of 2-d affine toric variety defined by: x + 2*y + 6
  To:   Closed subscheme of 2-d CPR-Fano toric variety
        covered by 3 affine patches defined by: x + 2*y + 3*z
  Defn: Defined on coordinates by sending [x : y] to [-2*y - 6 : y : 2]
sage: patch.embedding_center()
[0 : -3]
sage: patch.embedding_morphism()(patch.embedding_center())
[0 : -3 : 2]
```

A more complicated example:

```
sage: # needs sage.libs.singular
sage: dP6.<x0,x1,x2,x3,x4,x5> = toric_varieties.dP6()
sage: twoP1 = dP6.subscheme(x0*x3)
sage: patch = twoP1.neighborhood([0,1,2, 3,4,5]); patch
Closed subscheme of 2-d affine toric variety defined by:
  3*x0
sage: patch.embedding_morphism()
Scheme morphism:
  From: Closed subscheme of 2-d affine toric variety defined by: 3*x0
  To:   Closed subscheme of 2-d CPR-Fano toric variety
        covered by 6 affine patches defined by: x0*x3
  Defn: Defined on coordinates by sending [x0 : x1] to [0 : x1 : 2 : 3 : 4 : 5]
sage: patch.embedding_center()
[0 : 1]
sage: patch.embedding_morphism()(patch.embedding_center())
[0 : 1 : 2 : 3 : 4 : 5]
```

16.10 Weierstrass form of a toric elliptic curve

There are 16 reflexive polygons in the plane, see `ReflexivePolytopes()`. Each of them defines a toric Fano variety. And each of them has a unique crepant resolution to a smooth toric surface (Section 10.4 in [CLS2011]) by subdividing the face fan. An anticanonical hypersurface defines an elliptic curve in this ambient space, which we call a toric elliptic curve. The purpose of this module is to write an anticanonical hypersurface equation in the short Weierstrass form $y^2 = x^3 + fx + g$. This works over any base ring as long as its characteristic $\neq 2, 3$.

For an analogous treatment of elliptic curves defined as complete intersection in higher dimensional toric varieties, see the module *weierstrass_higher*.

Technically, this module computes the Weierstrass form of the Jacobian of the elliptic curve. This is why you will never have to specify the origin (or zero section) in the following.

It turns out [Bra2011] that the anticanonical hypersurface equation of any one of the above 16 toric surfaces is a specialization (that is, set one or more of the coefficients to zero) of the following three cases. In inhomogeneous coordinates, they are

- Cubic in $\mathbb{P}^2$:

$$p(x,y) = a_{30}x^3 + a_{21}x^2y + a_{12}xy^2 + a_{03}y^3 + a_{20}x^2 + a_{11}xy + a_{02}y^2 + a_{10}x + a_{01}y + a_{00}$$

- Biquadric in $\mathbb{P}^1 \times \mathbb{P}^1$:

$$p(x,y) = a_{22}x^2y^2 + a_{21}x^2y + a_{20}x^2 + a_{12}xy^2 + a_{11}xy + xa_{10} + y^2a_{02} + ya_{01} + a_{00}$$

- Anticanonical hypersurface in weighted projective space $\mathbb{P}^2[1,1,2]$:

$$p(x,y) = a_{40}x^4 + a_{30}x^3 + a_{21}x^2y + a_{20}x^2 + a_{11}xy + a_{02}y^2 + a_{10}x + a_{01}y + a_{00}$$

EXAMPLES:

The main functionality is provided by *WeierstrassForm()*, which brings each of the above hypersurface equations into Weierstrass form:

```
sage: R.<x,y> = QQ[]
sage: cubic = x^3 + y^3 + 1
sage: WeierstrassForm(cubic)
(0, -27/4)
sage: WeierstrassForm(x^4 + y^2 + 1)
(-4, 0)
sage: WeierstrassForm(x^2*y^2 + x^2 + y^2 + 1)
(-16/3, 128/27)
```

Only the affine span of the Newton polytope of the polynomial matters. For example:

```
sage: R.<x,y,z> = QQ[]
sage: WeierstrassForm(x^3 + y^3 + z^3)
(0, -27/4)
sage: WeierstrassForm(x * cubic)
(0, -27/4)
```

This allows you to work with either homogeneous or inhomogeneous variables. For example, here is the del Pezzo surface of degree 8:

```
sage: dP8 = toric_varieties.dP8()
sage: dP8.inject_variables()
Defining t, x, y, z
sage: WeierstrassForm(x*y^2 + y^2*z + t^2*x^3 + t^2*z^3)
(-3, -2)
sage: WeierstrassForm(x*y^2 + y^2 + x^3 + 1)
(-3, -2)
```

By specifying only certain variables we can compute the Weierstrass form over the polynomial ring generated by the remaining variables. For example, here is a cubic over $\mathbf{Q}[a]$

```
sage: R.<a, x, y, z> = QQ[]
sage: cubic = x^3 + a*y^3 + a^2*z^3
sage: WeierstrassForm(cubic, variables=[x,y,z])
(0, -27/4*a^6)
```

REFERENCES:

- [Bra2011]
- [Du2010]
- [ARVT2005]
- [CLS2011]

sage.schemes.toric.weierstrass.**Discriminant**(*polynomial*, *variables=None*)

The discriminant of the elliptic curve.

INPUT:

See *WeierstrassForm()* for how to specify the input polynomial(s) and variables.

OUTPUT: The discriminant of the elliptic curve.

EXAMPLES:

```
sage: from sage.schemes.toric.weierstrass import Discriminant
sage: R.<x, y, z> = QQ[]
sage: Discriminant(x^3 + y^3 + z^3)
19683/16
sage: Discriminant(x*y*z)
0
sage: R.<w,x,y,z> = QQ[]
sage: quadratic1 = w^2 + x^2 + y^2
sage: quadratic2 = z^2 + w*x
sage: Discriminant([quadratic1, quadratic2])
-1/16
```

sage.schemes.toric.weierstrass.**Newton_polygon_embedded**(*polynomial*, *variables*)

Embed the Newton polytope of the polynomial in one of the three maximal reflexive polygons.

This function is a helper for *WeierstrassForm()*

INPUT:

Same as *WeierstrassForm()* with only a single polynomial passed.

OUTPUT:

A tuple $(\Delta, P, (x, y))$ where

- Δ is the Newton polytope of `polynomial`.
- $P(x, y)$ equals the input `polynomial` but with redefined variables such that its Newton polytope is Δ.

EXAMPLES:

```
sage: from sage.schemes.toric.weierstrass import Newton_polygon_embedded
sage: R.<x,y,z> = QQ[]
sage: cubic = x^3 + y^3 + z^3
sage: Newton_polygon_embedded(cubic, [x,y,z])
(A 2-dimensional lattice polytope in ZZ^3 with 3 vertices,
 x^3 + y^3 + 1,
 (x, y))

sage: R.<a, x,y,z> = QQ[]
sage: cubic = x^3 + a*y^3 + a^2*z^3
sage: Newton_polygon_embedded(cubic, variables=[x,y,z])
(A 2-dimensional lattice polytope in ZZ^3 with 3 vertices,
 a^2*x^3 + y^3 + a,
 (x, y))

sage: R.<s,t,x,y> = QQ[]
sage: biquadric = (s+t)^2 * (x+y)^2
sage: Newton_polygon_embedded(biquadric, [s,t,x,y])
(A 2-dimensional lattice polytope in ZZ^4 with 4 vertices,
 s^2*t^2 + 2*s^2*t + 2*s*t^2 + s^2 + 4*s*t + t^2 + 2*s + 2*t + 1,
 (s, t))
```

sage.schemes.toric.weierstrass.**Newton_polytope_vars_coeffs**(*polynomial*, *variables*)

Return the Newton polytope in the given variables.

INPUT:

See *WeierstrassForm()* for how to specify the input polynomial and variables.

OUTPUT:

A dictionary with keys the integral values of the Newton polytope and values the corresponding coefficient of `polynomial`.

EXAMPLES:

```
sage: from sage.schemes.toric.weierstrass import Newton_polytope_vars_coeffs
sage: R.<x,y,z,a30,a21,a12,a03,a20,a11,a02,a10,a01,a00> = QQ[]
sage: p = (a30*x^3 + a21*x^2*y + a12*x*y^2 + a03*y^3 + a20*x^2*z +
....:      a11*x*y*z + a02*y^2*z + a10*x*z^2 + a01*y*z^2 + a00*z^3)
sage: p_data = Newton_polytope_vars_coeffs(p, [x,y,z]);  p_data
{(0, 0, 3): a00,
 (0, 1, 2): a01,
 (0, 2, 1): a02,
 (0, 3, 0): a03,
 (1, 0, 2): a10,
 (1, 1, 1): a11,
 (1, 2, 0): a12,
 (2, 0, 1): a20,
 (2, 1, 0): a21,
 (3, 0, 0): a30}

sage: from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_PPL
sage: polytope = LatticePolytope_PPL(list(p_data));  polytope
A 2-dimensional lattice polytope in ZZ^3 with 3 vertices
sage: polytope.vertices()
((0, 0, 3), (3, 0, 0), (0, 3, 0))
sage: polytope.embed_in_reflexive_polytope()
The map A*x+b with A=
[-1 -1]
[ 0  1]
[ 1  0]
b =
(3, 0, 0)
```

sage.schemes.toric.weierstrass.**WeierstrassForm**(*polynomial*, *variables=None*, *transformation=False*)

Return the Weierstrass form of an elliptic curve inside either inside a toric surface or $\mathbb{P}^3$.

INPUT:

- `polynomial` – either a polynomial or a list of polynomials defining the elliptic curve. A single polynomial can be either a cubic, a biquadric, or the hypersurface in $\mathbb{P}^2[1,1,2]$. In this case the equation need not be in any standard form, only its Newton polyhedron is used. If two polynomials are passed, they must both be quadratics in $\mathbb{P}^3$.
- `variables` – a list of variables of the parent polynomial ring or `None` (default). In the latter case, all variables are taken to be polynomial ring variables. If a subset of polynomial ring variables are given, the Weierstrass form is determined over the function field generated by the remaining variables.
- `transformation` – boolean (default: `False`). Whether to return the new variables that bring `polynomial` into Weierstrass form.

OUTPUT:

The pair of coefficients (f, g) of the Weierstrass form $y^2 = x^3 + fx + g$ of the hypersurface equation.

If `transformation=True`, a triple (X, Y, Z) of polynomials defining a rational map of the toric hypersurface or complete intersection in $\mathbb{P}^3$ to its Weierstrass form in $\mathbb{P}^2[2, 3, 1]$ is returned. That is, the triple satisfies

$$Y^2 = X^3 + fXZ^4 + gZ^6$$

when restricted to the toric hypersurface or complete intersection.

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: cubic = x^3 + y^3 + z^3
sage: f, g = WeierstrassForm(cubic);  (f, g)
(0, -27/4)
```

Same in inhomogeneous coordinates:

```
sage: R.<x,y> = QQ[]
sage: cubic = x^3 + y^3 + 1
sage: f, g = WeierstrassForm(cubic);  (f, g)
(0, -27/4)

sage: X,Y,Z = WeierstrassForm(cubic, transformation=True);  (X,Y,Z)
(-x^3*y^3 - x^3 - y^3,
 1/2*x^6*y^3 - 1/2*x^3*y^6 - 1/2*x^6 + 1/2*y^6 + 1/2*x^3 - 1/2*y^3,
 x*y)
```

Note that plugging in $[X : Y : Z]$ to the Weierstrass equation is a complicated polynomial, but contains the hypersurface equation as a factor:

```
sage: -Y^2 + X^3 + f*X*Z^4 + g*Z^6
-1/4*x^12*y^6 - 1/2*x^9*y^9 - 1/4*x^6*y^12 + 1/2*x^12*y^3
- 7/2*x^9*y^6 - 7/2*x^6*y^9 + 1/2*x^3*y^12 - 1/4*x^12 - 7/2*x^9*y^3
- 45/4*x^6*y^6 - 7/2*x^3*y^9 - 1/4*y^12 - 1/2*x^9 - 7/2*x^6*y^3
- 7/2*x^3*y^6 - 1/2*y^9 - 1/4*x^6 + 1/2*x^3*y^3 - 1/4*y^6
sage: cubic.divides(-Y^2 + X^3 + f*X*Z^4 + g*Z^6)
True
```

Only the affine span of the Newton polytope of the polynomial matters. For example:

```
sage: R.<x,y,z> = QQ[]
sage: cubic = x^3 + y^3 + z^3
sage: WeierstrassForm(cubic.subs(z=1))
(0, -27/4)
sage: WeierstrassForm(x * cubic)
(0, -27/4)
```

This allows you to work with either homogeneous or inhomogeneous variables. For example, here is the del Pezzo surface of degree 8:

```
sage: dP8 = toric_varieties.dP8()
sage: dP8.inject_variables()
Defining t, x, y, z
sage: WeierstrassForm(x*y^2 + y^2*z + t^2*x^3 + t^2*z^3)
(-3, -2)
sage: WeierstrassForm(x*y^2 + y^2 + x^3 + 1)
(-3, -2)
```

By specifying only certain variables we can compute the Weierstrass form over the function field generated by the remaining variables. For example, here is a cubic over $\mathbf{Q}[a]$

```
sage: R.<a, x,y,z> = QQ[]
sage: cubic = x^3 + a*y^3 + a^2*z^3
sage: WeierstrassForm(cubic, variables=[x,y,z])
(0, -27/4*a^6)
```

sage.schemes.toric.weierstrass.**WeierstrassForm_P1xP1**(*biquadric*, *variables=None*)

Bring a biquadric into Weierstrass form

Input/output is the same as *WeierstrassForm()*, except that the input polynomial must be a standard biquadric in $\mathbb{P}^2$,

$$p(x,y) = a_{40}x^4 + a_{30}x^3 + a_{21}x^2y + a_{20}x^2 + a_{11}xy + a_{02}y^2 + a_{10}x + a_{01}y + a_{00}$$

EXAMPLES:

```
sage: from sage.schemes.toric.weierstrass import WeierstrassForm_P1xP1
sage: R.<x0,x1,y0,y1> = QQ[]
sage: biquadric = (x0^2*y0^2 + x0*x1*y0^2*2 + x1^2*y0^2*3
....:     + x0^2*y0*y1*4 + x0*x1*y0*y1*5 + x1^2*y0*y1*6
....:     + x0^2*y1^2*7 + x0*x1*y1^2*8)
sage: WeierstrassForm_P1xP1(biquadric, [x0, x1, y0, y1])
(1581/16, -3529/32)
```

Since there is no $x_1^2y_1^2$ term in `biquadric`, we can dehomogenize it and get a cubic:

```
sage: from sage.schemes.toric.weierstrass import WeierstrassForm_P2
sage: WeierstrassForm_P2(biquadric(x0=1,y0=1))
(1581/16, -3529/32)
```

sage.schemes.toric.weierstrass.**WeierstrassForm_P2**(*polynomial*, *variables=None*)

Bring a cubic into Weierstrass form.

Input/output is the same as *WeierstrassForm()*, except that the input polynomial must be a standard cubic in $\mathbb{P}^2$,

$$p(x,y) = a_{30}x^3 + a_{21}x^2y + a_{12}xy^2 + a_{03}y^3 + a_{20}x^2 + a_{11}xy + a_{02}y^2 + a_{10}x + a_{01}y + a_{00}$$

EXAMPLES:

```
sage: from sage.schemes.toric.weierstrass import WeierstrassForm_P2
sage: R.<x,y,z> = QQ[]
sage: WeierstrassForm_P2(x^3 + y^3 + z^3)
(0, -27/4)

sage: R.<x,y,z, a,b> = QQ[]
sage: WeierstrassForm_P2(-y^2*z + x^3 + a*x*z^2 + b*z^3, [x,y,z])
(a, b)
```

sage.schemes.toric.weierstrass.**WeierstrassForm_P2_112**(*polynomial*, *variables=None*)

Bring an anticanonical hypersurface in $\mathbb{P}^2[1,1,2]$ into Weierstrass form.

Input/output is the same as *WeierstrassForm()*, except that the input polynomial must be a standard anticanonical hypersurface in weighted projective space $\mathbb{P}^2[1,1,2]$:

$$p(x,y) = a_{40}x^4 + a_{30}x^3 + a_{21}x^2y + a_{20}x^2 + a_{11}xy + a_{02}y^2 + a_{10}x + a_{01}y + a_{00}$$

EXAMPLES:

```
sage: from sage.schemes.toric.weierstrass import WeierstrassForm_P2_112
sage: fan = Fan(rays=[(1,0),(0,1),(-1,-2),(0,-1)],
....:           cones=[[0,1],[1,2],[2,3],[3,0]])
sage: P112.<x,y,z,t> = ToricVariety(fan)
sage: (-P112.K()).sections_monomials()
(z^4*t^2, x*z^3*t^2, x^2*z^2*t^2, x^3*z*t^2,
 x^4*t^2, y*z^2*t, x*y*z*t, x^2*y*t, y^2)
sage: WeierstrassForm_P2_112(sum(_), [x,y,z,t])
(-97/48, 17/864)
```

sage.schemes.toric.weierstrass.**j_invariant**(*polynomial*, *variables=None*)

Return the j-invariant of the elliptic curve.

INPUT:

See *WeierstrassForm()* for how to specify the input polynomial(s) and variables.

OUTPUT:

The j-invariant of the (irreducible) cubic. Notable special values:

- The Fermat cubic: $j(x^3+y^3+z^3) = 0$
- A nodal cubic: $j(-y^2+x^2+x^3) = \infty$
- A cuspidal cubic $y^2 = x^3$ has undefined j-invariant. In this case, a `ValueError` is raised.

EXAMPLES:

```
sage: from sage.schemes.toric.weierstrass import j_invariant
sage: R.<x,y,z> = QQ[]
sage: j_invariant(x^3 + y^3 + z^3)
0
sage: j_invariant(-y^2 + x^2 + x^3)
+Infinity
sage: R.<x,y,z, a,b> = QQ[]
sage: j_invariant( -y^2*z + x^3 + a*x*z^2, [x,y,z])
1728
```

16.11 Map to the Weierstrass form of a toric elliptic curve

There are 16 reflexive polygons in 2-d. Each defines a toric Fano variety, which (since it is 2-d) has a unique crepant resolution to a smooth toric surface. An anticanonical hypersurface defines a genus one curve C in this ambient space, with Jacobian elliptic curve $J(C)$ which can be defined by the Weierstrass model $y^2 = x^3 + fx + g$. The coefficients f and g can be computed with the *weierstrass* module. The purpose of this model is to give an explicit rational map $C \to J(C)$. This is an n^2-cover, where n is the minimal multi-section of C.

Since it is technically often easier to deal with polynomials than with fractions, we return the rational map in terms of homogeneous coordinates. That is, the ambient space for the Weierstrass model is the weighted projective space

$\mathbb{P}^2[2,3,1]$ with homogeneous coordinates $[X:Y:Z]=[\lambda^2 X, \lambda^3 Y, \lambda Z]$. The homogenized Weierstrass equation is

$$Y^2 = X^3 + fXZ^4 + gZ^6$$

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: cubic = x^3 + y^3 + 1
sage: f, g = WeierstrassForm(cubic);  (f,g)
(0, -27/4)
```

That is, this hypersurface $C \in \mathbb{P}^2$ has a Weierstrass equation $Y^2 = X^3 + 0 \cdot XZ^4 - \frac{27}{4}Z^6$ where $[X:Y:Z]$ are projective coordinates on $\mathbb{P}^2[2,3,1]$. The form of the map $C \to J(C)$ is:

```
sage: X,Y,Z = WeierstrassForm(cubic, transformation=True);  (X,Y,Z)
(-x^3*y^3 - x^3 - y^3,
 1/2*x^6*y^3 - 1/2*x^3*y^6 - 1/2*x^6 + 1/2*y^6 + 1/2*x^3 - 1/2*y^3,
 x*y)
```

Note that plugging in $[X:Y:Z]$ to the Weierstrass equation is a complicated polynomial, but contains the hypersurface equation as a factor:

```
sage: -Y^2 + X^3 + f*X*Z^4 + g*Z^6
-1/4*x^12*y^6 - 1/2*x^9*y^9 - 1/4*x^6*y^12 + 1/2*x^12*y^3
- 7/2*x^9*y^6 - 7/2*x^6*y^9 + 1/2*x^3*y^12 - 1/4*x^12 - 7/2*x^9*y^3
- 45/4*x^6*y^6 - 7/2*x^3*y^9 - 1/4*y^12 - 1/2*x^9 - 7/2*x^6*y^3
- 7/2*x^3*y^6 - 1/2*y^9 - 1/4*x^6 + 1/2*x^3*y^3 - 1/4*y^6
sage: cubic.divides(-Y^2 + X^3 + f*X*Z^4 + g*Z^6)
True
```

If you prefer you can also use homogeneous coordinates for $C \in \mathbb{P}^2$

```
sage: R.<x,y,z> = QQ[]
sage: cubic = x^3 + y^3 + z^3
sage: f, g = WeierstrassForm(cubic);  (f,g)
(0, -27/4)
sage: X,Y,Z = WeierstrassForm(cubic, transformation=True)
sage: cubic.divides(-Y^2 + X^3 + f*X*Z^4 + g*Z^6)
True
```

The 16 toric surfaces corresponding to the 16 reflexive polygons can all be blown down to $\mathbb{P}^2$, $\mathbb{P}^1 \times \mathbb{P}^1$, or $\mathbb{P}^2[1,1,2]$. Their (and hence in all 16 cases) anticanonical hypersurface can equally be brought into Weierstrass form. For example, here is an anticanonical hypersurface in $\mathbb{P}^2[1,1,2]$

```
sage: P2_112 = toric_varieties.P2_112()
sage: C = P2_112.anticanonical_hypersurface(coefficients=[1]*4);  C
Closed subscheme of 2-d CPR-Fano toric variety
covered by 3 affine patches defined by:
  z0^4 + z2^4 + z0*z1*z2 + z1^2
sage: eq = C.defining_polynomials()[0]
sage: f, g = WeierstrassForm(eq)
sage: X,Y,Z = WeierstrassForm(eq, transformation=True)
sage: (-Y^2 + X^3 + f*X*Z^4 + g*Z^6).reduce(C.defining_ideal())
0
```

Finally, you sometimes have to manually specify the variables to use. This is either because the equation is degenerate or because it contains additional variables that you want to treat as coefficients:

```
sage: R.<a, x,y,z> = QQ[]
sage: cubic = x^3 + y^3 + z^3 + a*x*y*z
sage: f, g = WeierstrassForm(cubic, variables=[x,y,z])
sage: X,Y,Z = WeierstrassForm(cubic, variables=[x,y,z], transformation=True)
sage: cubic.divides(-Y^2 + X^3 + f*X*Z^4 + g*Z^6)
True
```

REFERENCES:

- [AKMMMP2002]

sage.schemes.toric.weierstrass_covering.**WeierstrassMap**(*polynomial*, *variables=None*)

Return the Weierstrass form of an anticanonical hypersurface.

You should use *sage.schemes.toric.weierstrass.WeierstrassForm()* with `transformation=True` to get the transformation. This function is only for internal use.

INPUT:

- `polynomial` – a polynomial. The toric hypersurface equation. Can be either a cubic, a biquadric, or the hypersurface in $\mathbb{P}^2[1,1,2]$. The equation need not be in any standard form, only its Newton polyhedron is used.
- `variables` – a list of variables of the parent polynomial ring or `None` (default). In the latter case, all variables are taken to be polynomial ring variables. If a subset of polynomial ring variables are given, the Weierstrass form is determined over the function field generated by the remaining variables.

OUTPUT:

A triple (X, Y, Z) of polynomials defining a rational map of the toric hypersurface to its Weierstrass form in $\mathbb{P}^2[2,3,1]$. That is, the triple satisfies

$$Y^2 = X^3 + fXZ^4 + gZ^6$$

when restricted to the toric hypersurface.

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: cubic = x^3 + y^3 + z^3
sage: X,Y,Z = WeierstrassForm(cubic, transformation=True);  (X,Y,Z)
(-x^3*y^3 - x^3*z^3 - y^3*z^3,
 1/2*x^6*y^3 - 1/2*x^3*y^6 - 1/2*x^6*z^3 + 1/2*y^6*z^3
     + 1/2*x^3*z^6 - 1/2*y^3*z^6,
 x*y*z)
sage: f, g = WeierstrassForm(cubic);  (f,g)
(0, -27/4)
sage: cubic.divides(-Y^2 + X^3 + f*X*Z^4 + g*Z^6)
True
```

Only the affine span of the Newton polytope of the polynomial matters. For example:

```
sage: WeierstrassForm(cubic.subs(z=1), transformation=True)
(-x^3*y^3 - x^3 - y^3,
 1/2*x^6*y^3 - 1/2*x^3*y^6 - 1/2*x^6
```

(continues on next page)

(continued from previous page)

```
     + 1/2*y^6 + 1/2*x^3 - 1/2*y^3,
 x*y)
sage: WeierstrassForm(x * cubic, transformation=True)
(-x^3*y^3 - x^3*z^3 - y^3*z^3,
 1/2*x^6*y^3 - 1/2*x^3*y^6 - 1/2*x^6*z^3 + 1/2*y^6*z^3
     + 1/2*x^3*z^6 - 1/2*y^3*z^6,
 x*y*z)
```

This allows you to work with either homogeneous or inhomogeneous variables. For example, here is the del Pezzo surface of degree 8:

```
sage: dP8 = toric_varieties.dP8()
sage: dP8.inject_variables()
Defining t, x, y, z
sage: WeierstrassForm(x*y^2 + y^2*z + t^2*x^3 + t^2*z^3, transformation=True)
(-1/27*t^4*x^6 - 2/27*t^4*x^5*z - 5/27*t^4*x^4*z^2
     - 8/27*t^4*x^3*z^3 - 5/27*t^4*x^2*z^4 - 2/27*t^4*x*z^5
     - 1/27*t^4*z^6 - 4/81*t^2*x^4*y^2 - 4/81*t^2*x^3*y^2*z
     - 4/81*t^2*x*y^2*z^3 - 4/81*t^2*y^2*z^4 - 2/81*x^2*y^4
     - 4/81*x*y^4*z - 2/81*y^4*z^2,
 0,
 1/3*t^2*x^2*z + 1/3*t^2*x*z^2 - 1/9*x*y^2 - 1/9*y^2*z)
sage: WeierstrassForm(x*y^2 + y^2 + x^3 + 1, transformation=True)
(-1/27*x^6 - 4/81*x^4*y^2 - 2/81*x^2*y^4 - 2/27*x^5
     - 4/81*x^3*y^2 - 4/81*x*y^4 - 5/27*x^4 - 2/81*y^4 - 8/27*x^3
     - 4/81*x*y^2 - 5/27*x^2 - 4/81*y^2 - 2/27*x - 1/27,
 0,
 -1/9*x*y^2 + 1/3*x^2 - 1/9*y^2 + 1/3*x)
```

By specifying only certain variables we can compute the Weierstrass form over the function field generated by the remaining variables. For example, here is a cubic over $\mathbf{Q}[a]$

```
sage: R.<a, x,y,z> = QQ[]
sage: cubic = x^3 + a*y^3 + a^2*z^3
sage: WeierstrassForm(cubic, variables=[x,y,z], transformation=True)
(-a^9*y^3*z^3 - a^8*x^3*z^3 - a^7*x^3*y^3,
 -1/2*a^14*y^3*z^6 + 1/2*a^13*y^6*z^3 + 1/2*a^13*x^3*z^6
     - 1/2*a^11*x^3*y^6 - 1/2*a^11*x^6*z^3 + 1/2*a^10*x^6*y^3,
 a^3*x*y*z)
```

sage.schemes.toric.weierstrass_covering.**WeierstrassMap_P1xP1**(*polynomial*, *variables=None*)

Map an anticanonical hypersurface in $\mathbb{P}^1 \times \mathbb{P}^1$ into Weierstrass form.

Input/output is the same as *WeierstrassMap()*, except that the input polynomial must be a standard anticanonical hypersurface in the toric surface $\mathbb{P}^1 \times \mathbb{P}^1$:

EXAMPLES:

```
sage: from sage.schemes.toric.weierstrass_covering import WeierstrassMap_P1xP1
sage: from sage.schemes.toric.weierstrass import WeierstrassForm_P1xP1
sage: R.<x0,x1,y0,y1,a> = QQ[]
sage: biquadric = (x0^2*y0^2 + x1^2*y0^2 + x0^2*y1^2 + x1^2*y1^2 +
....:     a * x0*x1*y0*y1*5)
```

(continues on next page)

(continued from previous page)

```
sage: f, g = WeierstrassForm_P1xP1(biquadric, [x0, x1, y0, y1]);  (f,g)
(-625/48*a^4 + 25/3*a^2 - 16/3, 15625/864*a^6 - 625/36*a^4 - 100/9*a^2 + 128/27)
sage: X, Y, Z = WeierstrassMap_P1xP1(biquadric, [x0, x1, y0, y1])
sage: (-Y^2 + X^3 + f*X*Z^4 + g*Z^6).reduce(R.ideal(biquadric))                #␣
↪needs sage.libs.singular
0

sage: R = PolynomialRing(QQ, 'x,y,s,t', order='lex')
sage: R.inject_variables()
Defining x, y, s, t
sage: equation = (s^2*(x^2+2*x*y+3*y^2) + s*t*(4*x^2+5*x*y+6*y^2)
....:             + t^2*(7*x^2+8*x*y+9*y^2))
sage: X, Y, Z = WeierstrassMap_P1xP1(equation, [x,y,s,t])
sage: f, g = WeierstrassForm_P1xP1(equation, variables=[x,y,s,t])
sage: (-Y^2 + X^3 + f*X*Z^4 + g*Z^6).reduce(R.ideal(equation))                 #␣
↪needs sage.libs.singular
0

sage: R = PolynomialRing(QQ, 'x,s', order='lex')
sage: R.inject_variables()
Defining x, s
sage: equation = s^2*(x^2+2*x+3) + s*(4*x^2+5*x+6) + (7*x^2+8*x+9)
sage: X, Y, Z = WeierstrassMap_P1xP1(equation)
sage: f, g = WeierstrassForm_P1xP1(equation)
sage: (-Y^2 + X^3 + f*X*Z^4 + g*Z^6).reduce(R.ideal(equation))                 #␣
↪needs sage.libs.singular
0
```

sage.schemes.toric.weierstrass_covering.**WeierstrassMap_P2**(*polynomial*, *variables=None*)

Map a cubic to its Weierstrass form.

Input/output is the same as *WeierstrassMap()*, except that the input polynomial must be a cubic in $\mathbb{P}^2$,

$$p(x,y) = a_{30}x^3 + a_{21}x^2y + a_{12}xy^2 + a_{03}y^3 + a_{20}x^2 + a_{11}xy + a_{02}y^2 + a_{10}x + a_{01}y + a_{00}$$

EXAMPLES:

```
sage: from sage.schemes.toric.weierstrass import WeierstrassForm_P2
sage: from sage.schemes.toric.weierstrass_covering import WeierstrassMap_P2
sage: R.<x,y,z> = QQ[]
sage: equation =  x^3 + y^3 + z^3 + x*y*z
sage: f, g = WeierstrassForm_P2(equation)
sage: X,Y,Z = WeierstrassMap_P2(equation)
sage: equation.divides(-Y^2 + X^3 + f*X*Z^4 + g*Z^6)
True

sage: from sage.schemes.toric.weierstrass import WeierstrassForm_P2
sage: from sage.schemes.toric.weierstrass_covering import WeierstrassMap_P2
sage: R.<x,y> = QQ[]
sage: equation =  x^3 + y^3 + 1
sage: f, g = WeierstrassForm_P2(equation)
sage: X,Y,Z = WeierstrassMap_P2(equation)
```

(continues on next page)

(continued from previous page)

```
sage: equation.divides(-Y^2 + X^3 + f*X*Z^4 + g*Z^6)
True
```

sage.schemes.toric.weierstrass_covering.**WeierstrassMap_P2_112**(*polynomial*, *variables=None*)

Map an anticanonical hypersurface in $\mathbb{P}^2[1,1,2]$ into Weierstrass form.

Input/output is the same as *WeierstrassMap()*, except that the input polynomial must be a standard anticanonical hypersurface in weighted projective space $\mathbb{P}^2[1,1,2]$:

$$p(x,y) = a_{40}x^4 + a_{30}x^3 + a_{21}x^2y + a_{20}x^2 + a_{11}xy + a_{02}y^2 + a_{10}x + a_{01}y + a_{00}$$

EXAMPLES:

```
sage: from sage.schemes.toric.weierstrass_covering import WeierstrassMap_P2_112
sage: from sage.schemes.toric.weierstrass import WeierstrassForm_P2_112
sage: R = PolynomialRing(QQ, 'x,y,a0,a1,a2,a3,a4', order='lex')
sage: R.inject_variables()
Defining x, y, a0, a1, a2, a3, a4
sage: equation = y^2 + a0*x^4 + 4*a1*x^3 + 6*a2*x^2 + 4*a3*x + a4
sage: X, Y, Z = WeierstrassMap_P2_112(equation, [x,y])
sage: f, g = WeierstrassForm_P2_112(equation, variables=[x,y])
sage: (-Y^2 + X^3 + f*X*Z^4 + g*Z^6).reduce(R.ideal(equation))                #␣
↪needs sage.libs.singular
0
```

Another example, this time in homogeneous coordinates:

```
sage: fan = Fan(rays=[(1,0),(0,1),(-1,-2),(0,-1)],
....:           cones=[[0,1],[1,2],[2,3],[3,0]])
sage: P112.<x,y,z,t> = ToricVariety(fan)
sage: (-P112.K()).sections_monomials()
(z^4*t^2, x*z^3*t^2, x^2*z^2*t^2, x^3*z*t^2,
 x^4*t^2, y*z^2*t, x*y*z*t, x^2*y*t, y^2)
sage: C_eqn = sum(_)
sage: C = P112.subscheme(C_eqn)
sage: WeierstrassForm_P2_112(C_eqn, [x,y,z,t])
(-97/48, 17/864)
sage: X, Y, Z = WeierstrassMap_P2_112(C_eqn, [x,y,z,t])
sage: (-Y^2 + X^3 - 97/48*X*Z^4 + 17/864*Z^6).reduce(C.defining_ideal())     #␣
↪needs sage.libs.singular
0
```

16.12 Weierstrass for elliptic curves in higher codimension

The *weierstrass* module lets you transform a genus-one curve, given as a hypersurface in a toric surface, into Weierstrass form. The purpose of this module is to extend this to higher codimension subschemes of toric varieties. In general, this is an unsolved problem. However, for certain special cases this is known.

The simplest codimension-two case is the complete intersection of two quadratic equations in $\mathbb{P}^3$

```
sage: R.<w,x,y,z> = QQ[]
sage: quadratic1 = w^2 + x^2 + y^2
sage: quadratic2 = z^2 + w*x
sage: WeierstrassForm([quadratic1, quadratic2])
(-1/4, 0)
```

Hence, the Weierstrass form of this complete intersection is $Y^2 = X^3 - \frac{1}{4}XZ^4$.

sage.schemes.toric.weierstrass_higher.**WeierstrassForm2**(*polynomial*, *variables=None*, *transformation=False*)

Helper function for *WeierstrassForm()*

Currently, only the case of the complete intersection of two quadratic equations in $\mathbb{P}^3$ is supported.

INPUT / OUTPUT:

See *WeierstrassForm()*

sage.schemes.toric.weierstrass_higher.**WeierstrassForm_P3**(*quadratic1*, *quadratic2*, *variables=None*)

Bring a complete intersection of two quadratics into Weierstrass form.

Input/output is the same as *sage.schemes.toric.weierstrass.WeierstrassForm()*, except that the two input polynomials must be quadratic polynomials in $\mathbb{P}^3$.

EXAMPLES:

```
sage: from sage.schemes.toric.weierstrass_higher import WeierstrassForm_P3
sage: R.<w,x,y,z> = QQ[]
sage: quadratic1 = w^2 + x^2 + y^2
sage: quadratic2 = z^2 + w*x
sage: WeierstrassForm_P3(quadratic1, quadratic2)
(-1/4, 0)
```

sage.schemes.toric.weierstrass_higher.**WeierstrassMap_P3**(*quadratic1*, *quadratic2*, *variables=None*)

Bring a complete intersection of two quadratics into Weierstrass form.

Input/output is the same as *sage.schemes.toric.weierstrass.WeierstrassForm()*, except that the two input polynomials must be quadratic polynomials in $\mathbb{P}^3$.

EXAMPLES:

```
sage: from sage.schemes.toric.weierstrass_higher import \
....:     WeierstrassMap_P3, WeierstrassForm_P3
sage: R.<w,x,y,z> = QQ[]
sage: quadratic1 = w^2 + x^2 + y^2
sage: quadratic2 = z^2 + w*x
sage: X, Y, Z = WeierstrassMap_P3(quadratic1, quadratic2)
sage: X
1/1024*w^8 + 3/256*w^6*x^2 + 19/512*w^4*x^4 + 3/256*w^2*x^6 + 1/1024*x^8
sage: Y
1/32768*w^12 - 7/16384*w^10*x^2 - 145/32768*w^8*x^4 - 49/8192*w^6*x^6
- 145/32768*w^4*x^8 - 7/16384*w^2*x^10 + 1/32768*x^12
sage: Z
-1/8*w^2*y*z + 1/8*x^2*y*z

sage: a, b = WeierstrassForm_P3(quadratic1, quadratic2);  a, b
(-1/4, 0)
```

(continues on next page)

(continued from previous page)

```
sage: ideal = R.ideal(quadratic1, quadratic2)
sage: (-Y^2 + X^3 + a*X*Z^4 + b*Z^6).reduce(ideal)                              #␣
↪needs sage.libs.singular
0
```

16.13 Set of homomorphisms between two toric varieties

For schemes X and Y, this module implements the set of morphisms $Hom(X, Y)$. This is done by *SchemeHomset_generic*.

As a special case, the Hom-sets can also represent the points of a scheme. Recall that the K-rational points of a scheme X over k can be identified with the set of morphisms $Spec(K) \to X$. In Sage, the rational points are implemented by such scheme morphisms. This is done by *SchemeHomset_points* and its subclasses.

Note: You should not create the Hom-sets manually. Instead, use the `Hom()` method that is inherited by all schemes.

AUTHORS:

- Volker Braun (2012-02-18): Initial version

EXAMPLES:

Here is a simple example, the projection of $\mathbb{P}^1 \times \mathbb{P}^1 \to \mathbb{P}^1$

```
sage: P1xP1 = toric_varieties.P1xP1()
sage: P1 = toric_varieties.P1()
sage: hom_set = P1xP1.Hom(P1);  hom_set
Set of morphisms
  From: 2-d CPR-Fano toric variety covered by 4 affine patches
  To:   1-d CPR-Fano toric variety covered by 2 affine patches
```

In terms of the fan, we can define this morphism by the projection onto the first coordinate. The Hom-set can construct the morphism from the projection matrix alone:

```
sage: hom_set(matrix([[1],[0]]))
Scheme morphism:
  From: 2-d CPR-Fano toric variety covered by 4 affine patches
  To:   1-d CPR-Fano toric variety covered by 2 affine patches
  Defn: Defined by sending Rational polyhedral fan in 2-d lattice N
        to Rational polyhedral fan in 1-d lattice N.
sage: _.as_polynomial_map()
Scheme morphism:
  From: 2-d CPR-Fano toric variety covered by 4 affine patches
  To:   1-d CPR-Fano toric variety covered by 2 affine patches
  Defn: Defined on coordinates by sending [s : t : x : y] to [s : t]
```

In the case of toric algebraic schemes (defined by polynomials in toric varieties), this module defines the underlying morphism of the ambient toric varieties:

```
sage: P1xP1.inject_variables()
Defining s, t, x, y
sage: S = P1xP1.subscheme([s*x - t*y])
sage: type(S.Hom(S))
<class 'sage.schemes.toric.homset.SchemeHomset_toric_variety_with_category'>
```

Finally, you can have morphisms defined through homogeneous coordinates where the codomain is not implemented as a toric variety:

```
sage: P2_toric.<x,y,z> = toric_varieties.P2()
sage: P2_native.<u,v,w> = ProjectiveSpace(QQ, 2)
sage: toric_to_native = P2_toric.Hom(P2_native);  toric_to_native
Set of morphisms
  From: 2-d CPR-Fano toric variety covered by 3 affine patches
  To:   Projective Space of dimension 2 over Rational Field
sage: type(toric_to_native)
<class 'sage.schemes.toric.homset.SchemeHomset_toric_variety_with_category'>
sage: toric_to_native([x^2, y^2, z^2])
Scheme morphism:
  From: 2-d CPR-Fano toric variety covered by 3 affine patches
  To:   Projective Space of dimension 2 over Rational Field
  Defn: Defined on coordinates by sending [x : y : z] to (x^2 : y^2 : z^2)

sage: native_to_toric = P2_native.Hom(P2_toric);  native_to_toric
Set of morphisms
  From: Projective Space of dimension 2 over Rational Field
  To:   2-d CPR-Fano toric variety covered by 3 affine patches
sage: type(native_to_toric)
<class 'sage.schemes.projective.projective_homset.SchemeHomset_polynomial_projective_
↪space_with_category'>
sage: native_to_toric([u^2, v^2, w^2])
Scheme morphism:
  From: Projective Space of dimension 2 over Rational Field
  To:   2-d CPR-Fano toric variety covered by 3 affine patches
  Defn: Defined on coordinates by sending (u : v : w) to [u^2 : v^2 : w^2]
```

class sage.schemes.toric.homset.**SchemeHomset_points_subscheme_toric_field**(*X*, *Y*, *category=None*, *check=True*, *base=Integer Ring*)

Bases: *SchemeHomset_points_toric_base*

cardinality()

Return the number of points of the toric variety.

OUTPUT:

An integer or infinity. The cardinality of the set of points.

EXAMPLES:

```
sage: # needs sage.libs.singular
sage: P2.<x,y,z> = toric_varieties.P2(base_ring=GF(5))
sage: cubic = P2.subscheme([x^3 + y^3 + z^3])
```

(continues on next page)

(continued from previous page)

```
sage: list(cubic.point_set())
[[0 : 1 : 4], [1 : 0 : 4], [1 : 4 : 0], [1 : 1 : 2], [1 : 2 : 1], [1 : 3 : 3]]
sage: cubic.point_set().cardinality()
6
```

class sage.schemes.toric.homset.**SchemeHomset_points_toric_base**(*X*, *Y*, *category=None*, *check=True*, *base=Integer Ring*)

Bases: *SchemeHomset_points*

Base class for homsets with toric ambient spaces.

INPUT:

- same as for *SchemeHomset_points*.

OUTPUT: A scheme morphism of type *SchemeHomset_points_toric_base*.

EXAMPLES:

```
sage: P1xP1 = toric_varieties.P1xP1()
sage: P1xP1(QQ)
Set of rational points of 2-d CPR-Fano toric variety
covered by 4 affine patches
```

is_finite()

Return whether there are finitely many points.

OUTPUT: A boolean.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: P2.point_set().is_finite()
False
sage: P2.change_ring(GF(7)).point_set().is_finite()
True
```

class sage.schemes.toric.homset.**SchemeHomset_points_toric_field**(*X*, *Y*, *category=None*, *check=True*, *base=Integer Ring*)

Bases: *SchemeHomset_points_toric_base*

Set of rational points of a toric variety.

You should not use this class directly. Instead, use the *point_set()* method to construct the point set of a toric variety.

INPUT:

- same as for *SchemeHomset_points*.

OUTPUT: A scheme morphism of type *SchemeHomset_points_toric_field*.

EXAMPLES:

```
sage: P1xP1 = toric_varieties.P1xP1()
sage: P1xP1.point_set()
Set of rational points of 2-d CPR-Fano toric variety
covered by 4 affine patches
```

(continues on next page)

(continued from previous page)

```
sage: P1xP1(QQ)
Set of rational points of 2-d CPR-Fano toric variety
covered by 4 affine patches
```

The quotient $\mathbb{P}^2/\mathbf{Z}_3$ over $GF(7)$ by the diagonal action. This is tricky because the base field has a 3-rd root of unity:

```
sage: fan = NormalFan(ReflexivePolytope(2, 0))
sage: X = ToricVariety(fan, base_field=GF(7))
sage: point_set = X.point_set()
sage: point_set.cardinality()
21
sage: sorted(X.point_set().list())
[[0 : 0 : 1], [0 : 1 : 0], [0 : 1 : 1], [0 : 1 : 3],
 [1 : 0 : 0], [1 : 0 : 1], [1 : 0 : 3], [1 : 1 : 0],
 [1 : 1 : 1], [1 : 1 : 2], [1 : 1 : 3], [1 : 1 : 4],
 [1 : 1 : 5], [1 : 1 : 6], [1 : 3 : 0], [1 : 3 : 1],
 [1 : 3 : 2], [1 : 3 : 3], [1 : 3 : 4], [1 : 3 : 5],
 [1 : 3 : 6]]
```

As for a non-compact example, the blow-up of the plane is the line bundle $O_{\mathbf{P}^1}(-1)$. Its point set is the Cartesian product of the points on the base $\mathbf{P}^1$ with the points on the fiber:

```
sage: fan = Fan([Cone([(1,0), (1,1)]), Cone([(1,1), (0,1)])])
sage: blowup_plane = ToricVariety(fan, base_ring=GF(3))
sage: point_set = blowup_plane.point_set()
sage: sorted(point_set.list())
[[0 : 1 : 0], [0 : 1 : 1], [0 : 1 : 2],
 [1 : 0 : 0], [1 : 0 : 1], [1 : 0 : 2],
 [1 : 1 : 0], [1 : 1 : 1], [1 : 1 : 2],
 [1 : 2 : 0], [1 : 2 : 1], [1 : 2 : 2]]
```

Toric varieties with torus factors (that is, where the fan is not full-dimensional) also work:

```
sage: F_times_Fstar = ToricVariety(Fan([Cone([(1,0)])]), base_field=GF(3))
sage: sorted(F_times_Fstar.point_set().list())
[[0 : 1], [0 : 2], [1 : 1], [1 : 2], [2 : 1], [2 : 2]]
```

cardinality()

Return the number of points of the toric variety.

OUTPUT:

An integer or infinity. The cardinality of the set of points.

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: V = ToricVariety(FaceFan(o))
sage: V.change_ring(GF(2)).point_set().cardinality()
27
sage: V.change_ring(GF(8, "a")).point_set().cardinality()                    #␣
↪needs sage.rings.finite_rings
729
```

(continues on next page)

(continued from previous page)

```
sage: V.change_ring(GF(101)).point_set().cardinality()
1061208
```

For non-smooth varieties over finite fields, the homogeneous rescalings are solved. This is somewhat slower:

```
sage: fan = NormalFan(ReflexivePolytope(2, 0))
sage: X = ToricVariety(fan, base_field=GF(7))
sage: X.point_set().cardinality()
21
```

Fulton's formula does not apply since the variety is not smooth. And, indeed, naive application gives a different result:

```
sage: q = X.base_ring().order()
sage: n = X.dimension()
sage: d = map(len, fan().cones())
sage: sum(dk * (q-1)**(n-k) for k, dk in enumerate(d))
57
```

Over infinite fields the number of points is not very tricky:

```
sage: V.count_points()
+Infinity
```

ALGORITHM:

Uses the formula in Fulton [Ful1993], section 4.5.

AUTHORS:

- Beth Malmskog (2013-07-14)
- Adriana Salerno (2013-07-14)
- Yiwei She (2013-07-14)
- Christelle Vincent (2013-07-14)
- Ursula Whitcher (2013-07-14)

class sage.schemes.toric.homset.**SchemeHomset_toric_variety**(*X*, *Y*, *category=None*, *check=True*, *base=Integer Ring*)

Bases: *SchemeHomset_generic*

Set of homomorphisms between two toric varieties.

EXAMPLES:

```
sage: P1xP1 = toric_varieties.P1xP1()
sage: P1 = toric_varieties.P1()
sage: hom_set = P1xP1.Hom(P1);  hom_set
Set of morphisms
  From: 2-d CPR-Fano toric variety covered by 4 affine patches
  To:   1-d CPR-Fano toric variety covered by 2 affine patches
sage: type(hom_set)
<class 'sage.schemes.toric.homset.SchemeHomset_toric_variety_with_category'>
```

(continues on next page)

(continued from previous page)

```
sage: hom_set(matrix([[1],[0]]))
Scheme morphism:
  From: 2-d CPR-Fano toric variety covered by 4 affine patches
  To:   1-d CPR-Fano toric variety covered by 2 affine patches
  Defn: Defined by sending Rational polyhedral fan in 2-d lattice N
        to Rational polyhedral fan in 1-d lattice N.
```

16.14 Enumerate points of a toric variety

The classes here are not meant to be instantiated manually. Instead, you should always use the methods of the *point set* of the variety.

In this module, points are always represented by tuples instead of Sage's class for points of the toric variety. All Sage library code must then convert it to proper point objects before returning it to the user.

EXAMPLES:

```
sage: P2 = toric_varieties.P2(base_ring=GF(3))
sage: point_set = P2.point_set()
sage: point_set.cardinality()
13
sage: next(iter(point_set))
[0 : 0 : 1]
sage: list(point_set)[0:5]
[[0 : 0 : 1], [1 : 0 : 0], [0 : 1 : 0], [0 : 1 : 1], [0 : 1 : 2]]
```

class sage.schemes.toric.points.**FiniteFieldPointEnumerator**(*fan*, *ring*)

Bases: *NaiveFinitePointEnumerator*

cardinality()

Return the cardinality of the point set.

OUTPUT: An integer. The number of points.

EXAMPLES:

```
sage: fan = NormalFan(ReflexivePolytope(2, 0))
sage: X = ToricVariety(fan, base_ring=GF(7))
sage: point_set = X.point_set()
sage: ffe = point_set._finite_field_enumerator()
sage: ffe.cardinality()
21
```

cone_points_iter()

Iterate over the open torus orbits and yield distinct points.

OUTPUT:

For each open torus orbit (cone): A triple consisting of the cone, the nonzero homogeneous coordinates in that orbit (list of integers), and the nonzero log coordinates of distinct points as a cokernel.

EXAMPLES:

```
sage: fan = NormalFan(ReflexivePolytope(2, 0))
sage: X = ToricVariety(fan, base_ring=GF(7))
sage: point_set = X.point_set()
sage: ffe = point_set._finite_field_enumerator()
sage: cpi = ffe.cone_points_iter()
sage: cone, nonzero_points, cokernel = list(cpi)[5]
sage: cone
1-d cone of Rational polyhedral fan in 2-d lattice N
sage: cone.ambient_ray_indices()
(2,)
sage: nonzero_points
[0, 1]
sage: cokernel
Finitely generated module V/W over Integer Ring with invariants (2)
sage: list(cokernel)
[(0), (1)]
sage: [p.lift() for p in cokernel]
[(0, 0), (0, 1)]
```

exp(*powers*)

Return the component-wise exp of `z`

INPUT:

- `powers` – a list/tuple/iterable of integers.

OUTPUT:

Tuple of finite field elements. The powers of the *multiplicative_generator()*.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: F.<a> = GF(5^2)
sage: point_set = toric_varieties.P2_123(base_ring=F).point_set()
sage: ffe = point_set._finite_field_enumerator()
sage: powers = list(range(24))
sage: ffe.exp(powers)
(1, a, a + 3, 4*a + 3, 2*a + 2, 4*a + 1, 2, 2*a, 2*a + 1, 3*a + 1,
 4*a + 4, 3*a + 2, 4, 4*a, 4*a + 2, a + 2, 3*a + 3, a + 4, 3, 3*a,
 3*a + 4, 2*a + 4, a + 1, 2*a + 3)
sage: ffe.log(ffe.exp(powers)) == tuple(powers)
True
```

log(*z*)

Return the component-wise log of `z`

INPUT:

- `z` – a list/tuple/iterable of non-zero finite field elements.

OUTPUT:

Tuple of integers. The logarithm with base the *multiplicative_generator()*.

EXAMPLES:

```
sage: # needs sage.rings.finite_rings
sage: F.<a> = GF(5^2)
sage: point_set = toric_varieties.P2_123(base_ring=F).point_set()
sage: ffe = point_set._finite_field_enumerator()
sage: z = tuple(a^i for i in range(25));  z
(1, a, a + 3, 4*a + 3, 2*a + 2, 4*a + 1, 2, 2*a, 2*a + 1, 3*a + 1,
 4*a + 4, 3*a + 2, 4, 4*a, 4*a + 2, a + 2, 3*a + 3, a + 4, 3, 3*a,
 3*a + 4, 2*a + 4, a + 1, 2*a + 3, 1)
sage: ffe.log(z)
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
 17, 18, 19, 20, 21, 22, 23, 0)
sage: ffe.exp(ffe.log(z)) == z
True
sage: ffe.log(ffe.exp(range(24))) == tuple(range(24))
True
```

multiplicative_generator()

Return the multiplicative generator of the finite field.

OUTPUT: A finite field element.

EXAMPLES:

```
sage: point_set = toric_varieties.P2(base_ring=GF(5^2, 'a')).point_set()    #␣
↪needs sage.rings.finite_rings
sage: ffe = point_set._finite_field_enumerator()                            #␣
↪needs sage.rings.finite_rings
sage: ffe.multiplicative_generator()                                        #␣
↪needs sage.rings.finite_rings
a
```

multiplicative_group_order()

EXAMPLES:

```
sage: class Foo:
....:     def __init__(self, x):
....:         self._x = x
....:     @cached_method
....:     def f(self):
....:         return self._x^2
sage: a = Foo(2)
sage: print(a.f.cache)
None
sage: a.f()
4
sage: a.f.cache
4
```

rescaling_log_generators()

Return the log generators of `rescalings()`.

OUTPUT:

A tuple containing the logarithms (see *log()*) of the generators of the multiplicative group of `rescalings()`.

EXAMPLES:

```
sage: point_set = toric_varieties.P2_123(base_ring=GF(5)).point_set()
sage: ffe = point_set._finite_field_enumerator()
sage: ffe.rescalings()
((1, 1, 1), (1, 4, 4), (4, 2, 3), (4, 3, 2))
sage: list(map(ffe.log, ffe.rescalings()))
[(0, 0, 0), (0, 2, 2), (2, 1, 3), (2, 3, 1)]
sage: ffe.rescaling_log_generators()
((2, 3, 1),)
```

root_generator(*n*)

Return a generator for `roots()`.

INPUT:

- `n` integer.

OUTPUT: A multiplicative generator for `roots()`.

EXAMPLES:

```
sage: point_set = toric_varieties.P2(base_ring=GF(5)).point_set()
sage: ffe = point_set._finite_field_enumerator()
sage: ffe.root_generator(2)
4
sage: ffe.root_generator(3)
1
sage: ffe.root_generator(4)
2
```

class sage.schemes.toric.points.**FiniteFieldSubschemePointEnumerator**(*polynomials*, *ambient*)

Bases: *NaiveSubschemePointEnumerator*

cardinality()

Return the cardinality of the point set.

OUTPUT: An integer. The number of points.

EXAMPLES:

```
sage: fan = NormalFan(ReflexivePolytope(2, 0))
sage: X.<u,v,w> = ToricVariety(fan, base_ring=GF(7))
sage: Y = X.subscheme(u^3 + v^3 + w^3 + u*v*w)
sage: point_set = Y.point_set()
sage: list(point_set)                                                    #␣
↪needs fpylll
[[0 : 1 : 3],
 [1 : 0 : 3],
 [1 : 3 : 0],
 [1 : 1 : 6],
 [1 : 1 : 4],
 [1 : 3 : 2],
 [1 : 3 : 5]]
sage: ffe = point_set._enumerator()                                      #␣
↪needs fpylll
sage: ffe.cardinality()                                                  #␣
```

(continues on next page)

(continued from previous page)

```
↪needs fpylll
7
```

homogeneous_coordinates(*log_t*, *nonzero_coordinates*, *cokernel*)

Convert the log of inhomogeneous coordinates back to homogeneous coordinates

INPUT:

- `log_t` – log of inhomogeneous coordinates of a point.
- `nonzero_coordinates` – the nonzero homogeneous coordinates in the patch.
- `cokernel` – the logs of the nonzero coordinates of all distinct points as a cokernel. See *FiniteFieldPointEnumerator.cone_points_iter()*.

OUTPUT:

The same point, but as a tuple of homogeneous coordinates.

EXAMPLES:

```
sage: P2.<x,y,z> = toric_varieties.P2(base_ring=GF(7))
sage: X = P2.subscheme([x^3 + 2*y^3 + 3*z^3, x*y*z + x*y^2])
sage: point_set = X.point_set()
sage: ffe = point_set._enumerator()
sage: cone, nonzero_coordinates, cokernel = list(ffe.ambient.cone_points_
↪iter())[5]
sage: cone.ambient_ray_indices(), nonzero_coordinates
((2,), [0, 1])
sage: ffe.homogeneous_coordinates([0], nonzero_coordinates, cokernel)
(1, 1, 0)
sage: ffe.homogeneous_coordinates([1], nonzero_coordinates, cokernel)
(1, 3, 0)
sage: ffe.homogeneous_coordinates([2], nonzero_coordinates, cokernel)
(1, 2, 0)
```

inhomogeneous_equations(*ring*, *nonzero_coordinates*, *cokernel*)

Inhomogenize the defining polynomials

INPUT:

- `ring` – the polynomial ring for inhomogeneous coordinates.
- `nonzero_coordinates` – list of integers. The indices of the non-zero homogeneous coordinates in the patch.
- `cokernel` – the logs of the nonzero coordinates of all distinct points as a cokernel. See *FiniteFieldPointEnumerator.cone_points_iter()*.

EXAMPLES:

```
sage: R.<s> = QQ[]
sage: P2.<x,y,z> = toric_varieties.P2(base_ring=GF(7))
sage: X = P2.subscheme([x^3 + 2*y^3 + 3*z^3, x*y*z + x*y^2])
sage: point_set = X.point_set()
sage: ffe = point_set._enumerator()
sage: cone, nonzero_coordinates, cokernel = list(ffe.ambient.cone_points_
↪iter())[5]
```

(continues on next page)

(continued from previous page)

```
sage: cone.ambient_ray_indices(), nonzero_coordinates
((2,), [0, 1])
sage: ffe.inhomogeneous_equations(R, nonzero_coordinates, cokernel)
[2*s^3 + 1, s^2]
```

solutions(*inhomogeneous_equations*, *log_range*)

Parallel version of *solutions_serial()*

INPUT/OUTPUT:

Same as *solutions_serial()*, except that the output points are in random order. Order depends on the number of processors and relative speed of separate processes.

EXAMPLES:

```
sage: R.<s> = GF(7)[]
sage: P2.<x,y,z> = toric_varieties.P2(base_ring=GF(7))
sage: X = P2.subscheme(1)
sage: point_set = X.point_set()
sage: ffe = point_set._enumerator()
sage: ffe.solutions([s^2 - 1, s^6 - s^2], [range(6)])
<generator object ...solutions at 0x...>
sage: sorted(_)
[(0,), (3,)]
```

solutions_serial(*inhomogeneous_equations*, *log_range*)

Iterate over solutions in a range.

INPUT:

- `inhomogeneous_equations` – list/tuple/iterable of inhomogeneous equations (i.e. output from *inhomogeneous_equations()*).
- `log_range` – list/tuple/iterable of integer ranges. One for each inhomogeneous coordinate. The logarithms of the homogeneous coordinates.

OUTPUT:

All solutions (as tuple of log inhomogeneous coordinates) in the Cartesian product of the ranges.

EXAMPLES:

```
sage: R.<s> = GF(7)[]
sage: P2.<x,y,z> = toric_varieties.P2(base_ring=GF(7))
sage: X = P2.subscheme(1)
sage: point_set = X.point_set()
sage: ffe = point_set._enumerator()
sage: ffe.solutions_serial([s^2 - 1, s^6 - s^2], [range(6)])
<generator object ...solutions_serial at 0x...>
sage: list(_)
[(0,), (3,)]
```

class sage.schemes.toric.points.**InfinitePointEnumerator**(*fan*, *ring*)

Bases: `object`

Point enumerator for infinite fields.

INPUT:

- fan – fan of the toric variety.
- ring – infinite base ring over which to enumerate points.

class sage.schemes.toric.points.**NaiveFinitePointEnumerator**(*fan*, *ring*)

Bases: object

The naive point enumerator.

This is very slow.

INPUT:

- fan – fan of the toric variety.
- ring – finite base ring over which to enumerate points.

EXAMPLES:

```
sage: from sage.schemes.toric.points import NaiveFinitePointEnumerator
sage: fan = toric_varieties.P2().fan()
sage: n = NaiveFinitePointEnumerator(fan, GF(3))
sage: next(iter(n))
(0, 0, 1)
```

cone_iter()

Iterate over all cones of the fan

OUTPUT:

Iterator over the cones, starting with the high-dimensional ones.

EXAMPLES:

```
sage: dP6 = toric_varieties.dP6(base_ring=GF(11))
sage: ne = dP6.point_set()._naive_enumerator()
sage: for cone in ne.cone_iter():
....:     print(cone.ambient_ray_indices())
(0, 1)
(1, 2)
(2, 3)
(3, 4)
(4, 5)
(0, 5)
(0,)
(1,)
(2,)
(3,)
(4,)
(5,)
()
```

coordinate_iter()

Iterate over all distinct homogeneous coordinates.

This method does NOT identify homogeneous coordinates that are equivalent by a homogeneous rescaling.

OUTPUT: An iterator over the points.

EXAMPLES:

```
sage: P2 = toric_varieties.P2(base_ring=GF(2))
sage: ni = P2.point_set()._naive_enumerator()
sage: list(ni.coordinate_iter())
[(0, 0, 1), (1, 0, 0), (0, 1, 0), (0, 1, 1),
 (1, 0, 1), (1, 1, 0), (1, 1, 1)]

sage: P1xP1 = toric_varieties.P1xP1(base_ring=GF(2))
sage: ni = P1xP1.point_set()._naive_enumerator()
sage: list(ni.coordinate_iter())
[(0, 1, 0, 1), (1, 0, 0, 1), (1, 0, 1, 0),
 (0, 1, 1, 0), (0, 1, 1, 1), (1, 0, 1, 1),
 (1, 1, 0, 1), (1, 1, 1, 0), (1, 1, 1, 1)]
```

orbit(*point*)

Return the orbit of homogeneous coordinates under rescalings.

OUTPUT:

The set of all homogeneous coordinates that are equivalent to `point`.

EXAMPLES:

```
sage: P2_123 = toric_varieties.P2_123(base_ring=GF(7))
sage: ne = P2_123.point_set()._naive_enumerator()
sage: sorted(ne.orbit([1, 0, 0]))
[(1, 0, 0), (2, 0, 0), (4, 0, 0)]
sage: sorted(ne.orbit([0, 1, 0]))
[(0, 1, 0), (0, 6, 0)]
sage: sorted(ne.orbit([0, 0, 1]))
[(0, 0, 1), (0, 0, 2), (0, 0, 3), (0, 0, 4), (0, 0, 5), (0, 0, 6)]
sage: sorted(ne.orbit([1, 1, 0]))
[(1, 1, 0), (1, 6, 0), (2, 1, 0), (2, 6, 0), (4, 1, 0), (4, 6, 0)]
```

rays()

Return all rays (real and virtual).

OUTPUT: Tuple of rays of the fan.

EXAMPLES:

```
sage: from sage.schemes.toric.points import NaiveFinitePointEnumerator
sage: fan = toric_varieties.torus(2).fan()
sage: fan.rays()
Empty collection
in 2-d lattice N
sage: n = NaiveFinitePointEnumerator(fan, GF(3))
sage: n.rays()
N(1, 0),
N(0, 1)
in 2-d lattice N
```

rescalings()

Return the rescalings of homogeneous coordinates.

OUTPUT:

A tuple containing all points that are equivalent to $[1 : 1 : \cdots : 1]$, the distinguished point of the big torus orbit.

EXAMPLES:

```
sage: P2_123 = toric_varieties.P2_123(base_ring=GF(5))
sage: ni = P2_123.point_set()._naive_enumerator()
sage: ni.rescalings()
((1, 1, 1), (1, 4, 4), (4, 2, 3), (4, 3, 2))

sage: dP8 = toric_varieties.dP8(base_ring=GF(3))
sage: ni = dP8.point_set()._naive_enumerator()
sage: ni.rescalings()
((1, 1, 1, 1), (1, 2, 2, 2), (2, 1, 2, 1), (2, 2, 1, 2))

sage: P1xP1 = toric_varieties.P1xP1(base_ring=GF(3))
sage: ni = P1xP1.point_set()._naive_enumerator()
sage: ni.rescalings()
((1, 1, 1, 1), (1, 1, 2, 2), (2, 2, 1, 1), (2, 2, 2, 2))
```

roots(*n*)

Return the n-th roots in the base field

INPUT:

- n integer.

OUTPUT:

Tuple containing all n-th roots (not only the primitive ones). In particular, 1 is included.

EXAMPLES:

```
sage: P2 = toric_varieties.P2(base_ring=GF(5))
sage: ne = P2.point_set()._naive_enumerator()
sage: ne.roots(2)
(1, 4)
sage: ne.roots(3)
(1,)
sage: ne.roots(4)
(1, 2, 3, 4)
```

units()

Return the units in the base field.

EXAMPLES:

```
sage: P2 = toric_varieties.P2(base_ring=GF(5))
sage: ne = P2.point_set()._naive_enumerator()
sage: ne.units()
(1, 2, 3, 4)
```

class sage.schemes.toric.points.**NaiveSubschemePointEnumerator**(*polynomials*, *ambient*)

Bases: object

Point enumerator for algebraic subschemes of toric varieties.

INPUT:

- `polynomials` – list/tuple/iterable of polynomials. The defining polynomials.
- `ambient` – enumerator for ambient space points.

16.15 Construct sheaves on toric varieties

A toric vector bundle (on a toric variety) is a vector bundle that is equivariant with respect to the algebraic torus action.

`sage.schemes.toric.sheaf.constructor.`**CotangentBundle**(*X*)

Construct the cotangent bundle of a toric variety.

INPUT:

- `X` – a toric variety. The base space of the bundle.

OUTPUT:

The cotangent bundle as a Klyachko bundle.

EXAMPLES:

```
sage: dP7 = toric_varieties.dP7()
sage: from sage.schemes.toric.sheaf.constructor import CotangentBundle
sage: CotangentBundle(dP7)
Rank 2 bundle on 2-d CPR-Fano toric variety covered by 5 affine patches.
```

`sage.schemes.toric.sheaf.constructor.`**LineBundle**(*X*, *D*)

Construct the rank-1 bundle $O(D)$.

INPUT:

- `X` – a toric variety. The base space of the bundle.
- `D` – a toric divisor.

OUTPUT:

The line bundle $O(D)$ as a Klyachko bundle of rank 1.

EXAMPLES:

```
sage: X = toric_varieties.dP8()
sage: D = X.divisor(0)
sage: from sage.schemes.toric.sheaf.constructor import LineBundle
sage: O_D = LineBundle(X, D)
sage: O_D.cohomology(dim=True, weight=(0,0))
(1, 0, 0)
```

class `sage.schemes.toric.sheaf.constructor.`**SheafLibrary**(*toric_variety*)

Bases: `object`

Utility object to construct sheaves on toric varieties.

Warning: You should never construct instances manually. Can be accessed from a toric variety via the *sage.schemes.toric.variety.ToricVariety_field.sheaves* attribute.

EXAMPLES:

```
sage: type(toric_varieties.P2().sheaves)
<class 'sage.schemes.toric.sheaf.constructor.SheafLibrary'>
```

Klyachko(*multi_filtration*)

Construct a Klyachko bundle (sheaf) from filtration data.

INPUT:

- `multi_filtration` – a multi-filtered vectors space with multiple filtrations being indexed by the rays of the fan. Either an instance of `MultiFilteredVectorSpace()` or something (like a dictionary of ordinary filtered vector spaces).

OUTPUT:

The Klyachko bundle defined by the filtrations, one for each ray, of a vector space.

EXAMPLES:

```
sage: P1 = toric_varieties.P1()
sage: v1, v2, v3 = [(1,0,0), (0,1,0), (0,0,1)]
sage: F1 = FilteredVectorSpace({1: [v1, v2, v3], 3: [v1]})
sage: F2 = FilteredVectorSpace({0: [v1, v2, v3], 2: [v2, v3]})
sage: P1 = toric_varieties.P1()
sage: r1, r2 = P1.fan().rays()
sage: F = MultiFilteredVectorSpace({r1:F1, r2:F2});  F
Filtrations
    N(-1): QQ^3 >= QQ^2 >= QQ^2 >=  0   >= 0
     N(1): QQ^3 >= QQ^3 >= QQ^1 >= QQ^1 >= 0
sage: P1.sheaves.Klyachko(F)
Rank 3 bundle on 1-d CPR-Fano toric variety covered by 2 affine patches.
```

cotangent_bundle()

Return the cotangent bundle of the toric variety.

OUTPUT:

The cotangent bundle as a Klyachko bundle.

EXAMPLES:

```
sage: dP6 = toric_varieties.dP6()
sage: TX = dP6.sheaves.tangent_bundle()
sage: TXdual = dP6.sheaves.cotangent_bundle()
sage: TXdual == TX.dual()
True
```

divisor(**args*, ***kwds*)

Return a toric divisor.

INPUT:

This is just an alias for *sage.schemes.toric.variety.ToricVariety_field.divisor()*, see there for details.

By abuse of notation, you can usually use the divisor D interchangeably with the line bundle $O(D)$.

OUTPUT:

A toric divisor.

EXAMPLES:

```
sage: dP6 = toric_varieties.dP6()
sage: dP6.inject_variables()
Defining x, u, y, v, z, w
sage: D = dP6.sheaves.divisor(x*u^3);  D
V(x) + 3*V(u)
sage: D == dP6.divisor(x*u^3)
True
```

line_bundle(*divisor*)

Construct the rank-1 bundle $O(D)$.

INPUT:

- `divisor` – a toric divisor.

OUTPUT:

The line bundle $O(D)$ for the given divisor as a Klyachko bundle of rank 1.

EXAMPLES:

```
sage: X = toric_varieties.dP8()
sage: D = X.divisor(0)
sage: O_D = X.sheaves.line_bundle(D)
sage: O_D.cohomology(dim=True, weight=(0,0))
(1, 0, 0)
```

tangent_bundle()

Return the tangent bundle of the toric variety.

OUTPUT:

The tangent bundle as a Klyachko bundle.

EXAMPLES:

```
sage: toric_varieties.dP6().sheaves.tangent_bundle()
Rank 2 bundle on 2-d CPR-Fano toric variety covered by 6 affine patches.
```

trivial_bundle(*rank=1*)

Return the trivial bundle of rank `r`.

INPUT:

- `rank` – integer (optional; default: 1). The rank of the bundle.

OUTPUT:

The trivial bundle as a Klyachko bundle.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: I3 = P2.sheaves.trivial_bundle(3);  I3
Rank 3 bundle on 2-d CPR-Fano toric variety covered by 3 affine patches.
sage: I3.cohomology(weight=(0,0), dim=True)
(3, 0, 0)
```

sage.schemes.toric.sheaf.constructor.**TangentBundle**(*X*)

Construct the tangent bundle of a toric variety.

INPUT:

- `X` – a toric variety. The base space of the bundle.

OUTPUT:

The tangent bundle as a Klyachko bundle.

EXAMPLES:

```
sage: dP7 = toric_varieties.dP7()
sage: from sage.schemes.toric.sheaf.constructor import TangentBundle
sage: TangentBundle(dP7)
Rank 2 bundle on 2-d CPR-Fano toric variety covered by 5 affine patches.
```

sage.schemes.toric.sheaf.constructor.**TrivialBundle**(*X*, *rank=1*)

Return the trivial bundle of rank `r`.

INPUT:

- `X` – a toric variety. The base space of the bundle.
- `rank` – the rank of the bundle.

OUTPUT:

The trivial bundle as a Klyachko bundle.

EXAMPLES:

```
sage: P2 = toric_varieties.P2()
sage: from sage.schemes.toric.sheaf.constructor import TrivialBundle
sage: I3 = TrivialBundle(P2, 3);  I3
Rank 3 bundle on 2-d CPR-Fano toric variety covered by 3 affine patches.
sage: I3.cohomology(weight=(0,0), dim=True)
(3, 0, 0)
```

16.16 Klyachko bundles and sheaves

Klyachko bundles are torus-equivariant bundles on toric varieties. That is, the action of the maximal torus on the toric variety lifts to an action on the bundle. There is an equivalence of categories between Klyachko bundles [Kly1990] and multiple filtrations (one for each ray of the fan) of a vector space. The multi-filtrations are implemented in `sage.modules.multi_filtered_vector_space`.

EXAMPLES:

```
sage: X = toric_varieties.dP6xdP6()
sage: TX = X.sheaves.tangent_bundle()
sage: Alt2TX = TX.exterior_power(2);  Alt2TX
Rank 6 bundle on 4-d CPR-Fano toric variety covered by 36 affine patches.

sage: K = X.sheaves.line_bundle(X.K())
sage: antiK = X.sheaves.line_bundle(-X.K())
sage: (Alt2TX * K).cohomology(dim=True, weight=(0,0,0,0))  # long time
```

(continues on next page)

(continued from previous page)

```
(0, 0, 18, 0, 0)

sage: G_sum = TX + X.sheaves.trivial_bundle(2)
sage: V_sum = G_sum.wedge(2) * K                          # long time
sage: V_sum.cohomology(dim=True, weight=(0,0,0,0))    # long time
(0, 0, 18, 16, 1)
sage: Gtilde = G_sum.random_deformation()
sage: V = Gtilde.wedge(2) * K                        # long time
sage: V.cohomology(dim=True, weight=(0,0,0,0))    # long time  # random failure (see
↪#32773)
(0, 0, 3, 0, 0)
```

REFERENCES:

- [Kly1990]
- [BIP]

sage.schemes.toric.sheaf.klyachko.**Bundle**(*toric_variety*, *multi_filtration*, *check=True*)

Construct a Klyacho bundle

INPUT:

- `toric_variety` – a toric variety. The base space of the bundle.
- `multi_filtration` – a multi-filtered vectors space with multiple filtrations being indexed by the one-dimensional cones of the fan. Either an instance of `MultiFilteredVectorSpace()` or something (like a dictionary of ordinary filtered vector spaces).

EXAMPLES:

```
sage: P1 = toric_varieties.P1()
sage: v1, v2, v3 = [(1,0,0), (0,1,0), (0,0,1)]
sage: F1 = FilteredVectorSpace({1: [v1, v2, v3], 3: [v1]})
sage: F2 = FilteredVectorSpace({0: [v1, v2, v3], 2: [v2, v3]})
sage: P1 = toric_varieties.P1()
sage: r1, r2 = P1.fan().rays()
sage: F = MultiFilteredVectorSpace({r1: F1, r2: F2});  F
Filtrations
    N(-1): QQ^3 >= QQ^2 >= QQ^2 >=  0   >= 0
     N(1): QQ^3 >= QQ^3 >= QQ^1 >= QQ^1 >= 0
```

You should use the *Klyachko()* method to construct instances:

```
sage: P1.sheaves.Klyachko(F)
Rank 3 bundle on 1-d CPR-Fano toric variety covered by 2 affine patches.

sage: P1.sheaves.Klyachko({r1: F1, r2: F2})   # alternative
Rank 3 bundle on 1-d CPR-Fano toric variety covered by 2 affine patches.
```

The above is just a shorthand for:

```
sage: from sage.schemes.toric.sheaf.klyachko import Bundle
sage: Bundle(P1, F)
Rank 3 bundle on 1-d CPR-Fano toric variety covered by 2 affine patches.
```

class sage.schemes.toric.sheaf.klyachko.**KlyachkoBundle_class**(*toric_variety*, *multi_filtration*, *check=True*)

Bases: SageObject

A toric bundle using Klyachko's representation.

Warning: You should always use the *Bundle()* factory function to construct instances.

INPUT:

- toric_variety – a toric variety. The base space of the bundle.
- multi_filtration – a MultiFilteredVectorSpace() with index set the rays of the fan.
- check – boolean (default: True). Whether to perform consistency checks.

EXAMPLES:

```
sage: P1 = toric_varieties.P1()
sage: r1, r2 = P1.fan().rays()
sage: F = MultiFilteredVectorSpace({
....:      r1: FilteredVectorSpace(3,1),
....:      r2: FilteredVectorSpace(3,0)});  F
Filtrations
    N(-1): QQ^3 >=  0   >= 0
     N(1): QQ^3 >= QQ^3 >= 0
sage: from sage.schemes.toric.sheaf.klyachko import Bundle
sage: Bundle(P1, F)
Rank 3 bundle on 1-d CPR-Fano toric variety covered by 2 affine patches.
```

E_degree(*alpha*, *m*)

Return the vector subspace $E^\alpha(m)$.

INPUT:

- alpha – a ray of the fan. Can be specified by its index (an integer), a one-dimensional cone, or a N-lattice point.
- m – tuple of integers or M-lattice point. A point in the dual lattice of the fan.

OUTPUT:

The subspace $E^\alpha(\alpha m)$ of the filtration indexed by the ray α and at the filtration degree $\alpha * m$

EXAMPLES:

```
sage: X = toric_varieties.P2()
sage: M = X.fan().dual_lattice()
sage: V = X.sheaves.tangent_bundle()
sage: V.E_degree(X.fan().ray(0), (1,0))
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix: [1 0]
sage: V.E_degree(X.fan(1)[0], (1,0))
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix: [1 0]
sage: V.E_degree(0, (1,0))
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix: [1 0]
```

E_intersection(*sigma*, *m*)

Return the vector subspace $E^\sigma(m)$.

See [Kly1990], equation 4.1.

INPUT:

- `sigma` – a cone of the fan of the base toric variety.
- `m` – tuple of integers or M-lattice point. A point in the dual lattice of the fan. Must be immutable.

OUTPUT: The subspace $E^\sigma(m)$.

EXAMPLES:

```
sage: X = toric_varieties.P2()
sage: fan = X.fan()
sage: V = X.sheaves.tangent_bundle()
sage: V.E_intersection(fan(1)[0], (1,0))
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix: [1 0]
sage: V.E_intersection(fan(2)[0], (-1,1))
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix: [0 1]
```

For the empty cone, this is always the whole vector space:

```
sage: V.E_intersection(fan(0)[0], (1,0))
Vector space of dimension 2 over Rational Field
```

E_quotient(*sigma*, *m*)

Return the vector space quotient $E_\sigma(m)$.

See [Kly1990], equation 4.1.

INPUT:

- `sigma` – a cone of the fan of the base toric variety.
- `m` – tuple of integers or M-lattice point. A point in the dual lattice of the fan. Must be immutable.

OUTPUT: The subspace $E_\sigma(m)$.

EXAMPLES:

```
sage: X = toric_varieties.P2()
sage: fan = X.fan()
sage: M = fan.dual_lattice()
sage: cone = fan(1)[0]
sage: V = X.sheaves.tangent_bundle()
sage: m = M(1, 0)
sage: m.set_immutable()
sage: V.E_quotient(cone, m)
Vector space quotient V/W of dimension 1 over Rational Field where
 V: Vector space of dimension 2 over Rational Field
 W: Vector space of degree 2 and dimension 1 over Rational Field
    Basis matrix: [1 0]
sage: V.E_quotient(fan(2)[0], (-1,1))
Vector space quotient V/W of dimension 0 over Rational Field where
```

(continues on next page)

(continued from previous page)

```
 V: Vector space of dimension 2 over Rational Field
 W: Vector space of degree 2 and dimension 2 over Rational Field
    Basis matrix:
    [1 0]
    [0 1]
```

E_quotient_projection(*sigma*, *tau*, *m*)

Return the projection map $E_\sigma(m) \to E_\tau(m)$ where σ is a face of τ.

INPUT:

- `sigma` – a cone of the fan of the base toric variety.
- `tau` – a cone of the fan containing `sigma`.
- `m` – tuple of integers or M-lattice point. A point in the dual lattice of the fan. Must be immutable.

OUTPUT:

The restriction map

$$E_\sigma(m) \to E_\tau(m)$$

EXAMPLES:

```
sage: P3 = toric_varieties.P(3)
sage: rays = [(1,0,0), (0,1,0), (0,0,1)]
sage: F1 = FilteredVectorSpace(rays, {0: [0], 1: [2], 2: [1]})
sage: F2 = FilteredVectorSpace(3, 0)
sage: r = P3.fan().rays()
sage: V = P3.sheaves.Klyachko({r[0]: F1, r[1]: F2, r[2]: F2, r[3]: F2})
sage: tau = Cone([(1,0,0), (0,1,0)])
sage: sigma = Cone([(1,0,0)])
sage: M = P3.fan().dual_lattice()
sage: m = M(2,1,0)
sage: m.set_immutable()
sage: V.E_quotient(sigma, m)
Vector space quotient V/W of dimension 2 over Rational Field where
 V: Vector space of dimension 3 over Rational Field
 W: Vector space of degree 3 and dimension 1 over Rational Field
    Basis matrix: [0 1 0]
sage: V.E_quotient(tau, m)
Vector space quotient V/W of dimension 2 over Rational Field where
 V: Vector space of dimension 3 over Rational Field
 W: Vector space of degree 3 and dimension 1 over Rational Field
    Basis matrix: [0 1 0]
sage: V.E_quotient_projection(sigma, tau, m)
Vector space morphism represented by the matrix:
 [1 0]
 [0 1]
 Domain:   Vector space quotient V/W of dimension 2 over Rational Field where
           V: Vector space of dimension 3 over Rational Field
           W: Vector space of degree 3 and dimension 1 over Rational Field
              Basis matrix: [0 1 0]
 Codomain: Vector space quotient V/W of dimension 2 over Rational Field where
```

(continues on next page)

(continued from previous page)

```
          V: Vector space of dimension 3 over Rational Field
          W: Vector space of degree 3 and dimension 1 over Rational Field
             Basis matrix: [0 1 0]
```

base_ring()

Return the base field.

OUTPUT: A field.

EXAMPLES:

```
sage: T_P2 = toric_varieties.P2().sheaves.tangent_bundle()
sage: T_P2.base_ring()
Rational Field
```

cohomology(*degree=None*, *weight=None*, *dim=False*)

Return the bundle cohomology groups.

INPUT:

- `degree` – `None` (default) or an integer. The degree of the cohomology group.
- `weight` – `None` (default) or a tuple of integers or a M-lattice point. A point in the dual lattice of the fan defining a torus character. The weight of the cohomology group.
- `dim` – Boolean (default: `False`). Whether to return vector spaces or only their dimension.

OUTPUT:

The cohomology group of given cohomological `degree` and torus `weight`.

- If no `weight` is specified, the unweighted group (sum over all weights) is returned.
- If no `degree` is specified, a dictionary whose keys are integers and whose values are the cohomology groups is returned. If, in addition, `dim=True`, then an integral vector of the dimensions is returned.

EXAMPLES:

```
sage: V = toric_varieties.P2().sheaves.tangent_bundle()
sage: V.cohomology(degree=0, weight=(0,0))
Vector space of dimension 2 over Rational Field
sage: V.cohomology(weight=(0,0), dim=True)
(2, 0, 0)
sage: for i,j in cartesian_product((list(range(-2,3)), list(range(-2,3)))):
....:       HH = V.cohomology(weight=(i,j), dim=True)
....:       if HH.is_zero(): continue
....:       print('H^*i(P^2, TP^2)_M({}, {}) = {}'.format(i,j,HH))
H^*i(P^2, TP^2)_M(-1, 0) = (1, 0, 0)
H^*i(P^2, TP^2)_M(-1, 1) = (1, 0, 0)
H^*i(P^2, TP^2)_M(0, -1) = (1, 0, 0)
H^*i(P^2, TP^2)_M(0, 0) = (2, 0, 0)
H^*i(P^2, TP^2)_M(0, 1) = (1, 0, 0)
H^*i(P^2, TP^2)_M(1, -1) = (1, 0, 0)
H^*i(P^2, TP^2)_M(1, 0) = (1, 0, 0)
```

cohomology_complex(m)

Return the "cohomology complex" $C^*(m)$

See [Kly1990], equation 4.2.

INPUT:

- `m` – tuple of integers or M-lattice point. A point in the dual lattice of the fan. Must be immutable.

OUTPUT:

The "cohomology complex" as a chain complex over the *base_ring()*.

EXAMPLES:

```
sage: P3 = toric_varieties.P(3)
sage: rays = [(1,0,0), (0,1,0), (0,0,1)]
sage: F1 = FilteredVectorSpace(rays, {0: [0], 1: [2], 2: [1]})
sage: F2 = FilteredVectorSpace(rays, {0: [1,2], 1: [0]})
sage: r = P3.fan().rays()
sage: V = P3.sheaves.Klyachko({r[0]: F1, r[1]: F2, r[2]: F2, r[3]: F2})
sage: tau = Cone([(1,0,0), (0,1,0)])
sage: sigma = Cone([(1, 0, 0)])
sage: M = P3.fan().dual_lattice()
sage: m = M(1, 1, 0); m.set_immutable()
sage: V.cohomology_complex(m)
Chain complex with at most 2 nonzero terms over Rational Field

sage: F = CyclotomicField(3)
sage: P3 = toric_varieties.P(3).change_ring(F)
sage: V = P3.sheaves.Klyachko({r[0]: F1, r[1]: F2, r[2]: F2, r[3]: F2})
sage: V.cohomology_complex(m)
Chain complex with at most 2 nonzero terms over Cyclotomic
Field of order 3 and degree 2
```

direct_sum(*other*)

Return the sum of two vector bundles.

INPUT:

- `other` – a Klyachko bundle over the same base.

OUTPUT: The direct sum as a new Klyachko bundle.

EXAMPLES:

```
sage: X = toric_varieties.P2()
sage: V1 = X.sheaves.trivial_bundle(1)
sage: V2 = X.sheaves.trivial_bundle(2)
sage: V2.direct_sum(V1)
Rank 3 bundle on 2-d CPR-Fano toric variety covered by 3 affine patches.

sage: V1 = X.sheaves.trivial_bundle(1)
sage: V2 = X.sheaves.trivial_bundle(2)
sage: V2 == V1 + V1
True
```

dual()

Return the dual bundle.

OUTPUT: The dual bundle as a new Klyachko bundle.

EXAMPLES:

```
sage: P1 = toric_varieties.P1()
sage: H = P1.divisor(0)
sage: L = P1.sheaves.line_bundle(H)
sage: L.dual()
Rank 1 bundle on 1-d CPR-Fano toric variety covered by 2 affine patches.
sage: L.dual() == P1.sheaves.line_bundle(-H)
True
```

exterior_power(n)

Return the n-th exterior power.

INPUT:

- `n` – integer.

OUTPUT:

The n-th exterior power $\wedge^n_{i=1} V$ of the bundle V as a new Klyachko bundle.

EXAMPLES:

```
sage: X = toric_varieties.P2_123()
sage: TX = X.sheaves.tangent_bundle()
sage: antiK = X.sheaves.line_bundle(-X.K())
sage: TX.exterior_power(2) == antiK
True
sage: TX.wedge(2) == antiK   # alias
True
```

fiber()

Return the generic fiber of the vector bundle.

OUTPUT: A vector space over *base_ring()*.

EXAMPLES:

```
sage: T_P2 = toric_varieties.P2().sheaves.tangent_bundle()
sage: T_P2.fiber()
Vector space of dimension 2 over Rational Field
```

filtration_intersection(*sigma*, *i*)

Return the intersection of the filtered subspaces.

INPUT:

- `sigma` – a cone of the fan of the base toric variety.
- `i` – integer. The filtration degree.

OUTPUT:

Let the cone be spanned by the rays $\sigma = \langle r_1, \ldots, r_k \rangle$. This method returns the intersection

$$\bigcap_{r \in \{r_1,\ldots,r_k\}} E^r(i)$$

EXAMPLES:

```
sage: X = toric_varieties.P2()
sage: fan = X.fan()
sage: V = X.sheaves.tangent_bundle()
sage: V.filtration_intersection(fan(1)[0], 1)
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix: [1 0]
sage: V.filtration_intersection(fan(2)[0], 1)
Vector space of degree 2 and dimension 0 over Rational Field
Basis matrix: []
```

get_degree(*ray*, *i*)

Return the vector subspace `E^\alpha(i)`.

- `ray` – Integer, a N-lattice point, a one-dimensional cone, or `None` (default). Specifies a ray of the fan of the toric variety, either via its index or its generator.
- `i` – integer. The filtration degree.

OUTPUT:

A subspace of the *fiber()* vector space. The defining data of a Klyachko bundle.

EXAMPLES:

```
sage: TX = toric_varieties.dP6().sheaves.tangent_bundle()
sage: TX.get_degree(0, 1)
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix: [0 1]
```

get_filtration(*ray=None*)

Return the filtration associated to the `ray`.

INPUT:

- `ray` – Integer, a N-lattice point, a one-dimensional cone, or `None` (default). Specifies a ray of the fan of the toric variety, either via its index or its generator.

OUTPUT:

The filtered vector space associated to the given `ray`. If no ray is specified, all filtrations are returned.

EXAMPLES:

```
sage: TX = toric_varieties.dP6().sheaves.tangent_bundle()
sage: TX.get_filtration(0)
QQ^2 >= QQ^1 >= 0
sage: TX.get_filtration([-1, -1])
QQ^2 >= QQ^1 >= 0
sage: TX.get_filtration(TX.variety().fan(1)[0])
QQ^2 >= QQ^1 >= 0
sage: TX.get_filtration()
Filtrations
    N(-1, -1): QQ^2 >= QQ^1 >= 0
     N(-1, 0): QQ^2 >= QQ^1 >= 0
     N(0, -1): QQ^2 >= QQ^1 >= 0
      N(0, 1): QQ^2 >= QQ^1 >= 0
      N(1, 0): QQ^2 >= QQ^1 >= 0
      N(1, 1): QQ^2 >= QQ^1 >= 0
```

is_isomorphic(*other*)

Test whether two bundles are isomorphic.

INPUT:

- `other` – anything.

OUTPUT: A boolean.

EXAMPLES:

```
sage: X = toric_varieties.P2()
sage: T_X = X.sheaves.tangent_bundle()
sage: O_X = X.sheaves.trivial_bundle(1)
sage: T_X + O_X == O_X + T_X
False
sage: (T_X + O_X).is_isomorphic(O_X + T_X)
Traceback (most recent call last):
...
NotImplementedError
```

random_deformation(*epsilon=None*)

Return a generic torus-equivariant deformation of the bundle.

INPUT:

- `epsilon` – an element of the base ring. Scales the random deformation.

OUTPUT:

A new Klyachko bundle with randomly perturbed moduli. In particular, the same Chern classes.

EXAMPLES:

```
sage: P1 = toric_varieties.P1()
sage: H = P1.divisor(0)
sage: V = P1.sheaves.line_bundle(H) + P1.sheaves.line_bundle(-H)
sage: V.cohomology(dim=True, weight=(0,))
(1, 0)
sage: Vtilde = V.random_deformation()
sage: Vtilde.cohomology(dim=True, weight=(0,))  # random failure (see #32773)
(1, 0)
```

rank()

Return the rank of the vector bundle.

OUTPUT: An integer.

EXAMPLES:

```
sage: T_P2 = toric_varieties.P2().sheaves.tangent_bundle()
sage: T_P2.rank()
2
```

symmetric_power(*n*)

Return the n-th symmetric power.

INPUT:

- `n` – integer.

OUTPUT: The n-th symmetric power as a new Klyachko bundle.

EXAMPLES:

```
sage: P1 = toric_varieties.P1()
sage: H = P1.divisor(0)
sage: L = P1.sheaves.line_bundle(H)
sage: (L + L).symmetric_power(2)
Rank 3 bundle on 1-d CPR-Fano toric variety covered by 2 affine patches.
sage: (L + L).symmetric_power(2) == L*L + L*L + L*L
True
```

tensor_product(*other*)

Return the sum of two vector bundles.

INPUT:

- `other` – a Klyachko bundle over the same base.

OUTPUT: The tensor product as a new Klyachko bundle.

EXAMPLES:

```
sage: X = toric_varieties.P2()
sage: OX = X.sheaves.trivial_bundle(1)
sage: X.sheaves.tangent_bundle().tensor_product(OX)
Rank 2 bundle on 2-d CPR-Fano toric variety covered by 3 affine patches.
sage: OX == OX * OX
True
```

variety()

Return the base toric variety.

OUTPUT: A toric variety.

EXAMPLES:

```
sage: X = toric_varieties.P2()
sage: V = X.sheaves.tangent_bundle(); V
Rank 2 bundle on 2-d CPR-Fano toric variety covered by 3 affine patches.
sage: V.variety() is X
True
```

wedge(n)

Return the n-th exterior power.

INPUT:

- `n` – integer.

OUTPUT:

The n-th exterior power $\wedge^n_{i=1} V$ of the bundle V as a new Klyachko bundle.

EXAMPLES:

```
sage: X = toric_varieties.P2_123()
sage: TX = X.sheaves.tangent_bundle()
sage: antiK = X.sheaves.line_bundle(-X.K())
```

(continues on next page)

(continued from previous page)

```
sage: TX.exterior_power(2) == antiK
True
sage: TX.wedge(2) == antiK   # alias
True
```

sage.schemes.toric.sheaf.klyachko.**is_KlyachkoBundle**(*X*)

Test whether `X` is a Klyachko bundle

INPUT:

- `X` – anything.

OUTPUT: A boolean.

EXAMPLES:

```
sage: from sage.schemes.toric.sheaf.klyachko import is_KlyachkoBundle
sage: is_KlyachkoBundle('test')
False
```

CHAPTER

SEVENTEEN

CYCLIC COVERS

17.1 Cyclic covers over a finite field

The most interesting feature is computation of Frobenius matrix on Monsky-Washnitzer cohomology and the Frobenius polynomial.

REFERENCES:

- [ABCMT2019]

EXAMPLES:

```
sage: p = 13
sage: x = PolynomialRing(GF(p),"x").gen()
sage: C = CyclicCover(4, x^4 + 1)
sage: C.frobenius_polynomial()
x^6 - 6*x^5 + 3*x^4 + 60*x^3 + 39*x^2 - 1014*x + 2197
sage: R.<t> = PowerSeriesRing(Integers())
sage: C.projective_closure().zeta_series(2,t)
1 + 8*t + 102*t^2 + O(t^3)
sage: C.frobenius_polynomial().reverse()(t)/((1-t)*(1-p*t)) + O(t^5)
1 + 8*t + 102*t^2 + 1384*t^3 + 18089*t^4 + O(t^5)

sage: p = 49999
sage: x = PolynomialRing(GF(p),"x").gen()
sage: CyclicCover(5, x^5 + x).frobenius_polynomial()  # long time
x^12 + 299994*x^10 + 37498500015*x^8 + 2499850002999980*x^6 + 937425002249970000015*x^4
 + 18748125074998500001499994*x^2 + 1562312509374750003749970000 1
sage: CyclicCover(5, 2*x^5 + x).frobenius_polynomial()  # long time
x^12 + 299994*x^10 + 37498500015*x^8 + 2499850002999980*x^6 + 937425002249970000015*x^4
 + 18748125074998500001499994*x^2 + 1562312509374750003749970000 1

sage: p = 107
sage: x = PolynomialRing(GF(p),"x").gen()
sage: CyclicCover(2, x^5 + x).frobenius_matrix()
[              O(107^2)      89*107 + O(107^2)               O(107^2)              ␣
↪O(107^2)]
[     89*107 + O(107^2)               O(107^2)               O(107^2)              ␣
↪O(107^2)]
[              O(107^2)               O(107^2)               O(107^2) 105 + 5*107 +␣
↪O(107^2)]
[              O(107^2)               O(107^2) 89 + 53*107 + O(107^2)              ␣
```

(continues on next page)

(continued from previous page)

```
↪O(107^2)]
sage: CyclicCover(2, 3*x^5 + x).frobenius_matrix()
[              O(107^2)       14*107 + O(107^2)                O(107^2)              ␣
↪O(107^2)]
[     69*107 + O(107^2)                O(107^2)                O(107^2)              ␣
↪O(107^2)]
[              O(107^2)                O(107^2)                O(107^2) 61 + 58*107 +␣
↪O(107^2)]
[              O(107^2)                O(107^2) 69 + 53*107 + O(107^2)              ␣
↪O(107^2)]
sage: CyclicCover(3, x^3 + x).frobenius_matrix()
[          0           0       O(107)       O(107)]
[          0           0 52 + O(107)       O(107)]
[     O(107) 35 + O(107)            0            0]
[44 + O(107)      O(107)            0            0]
sage: CyclicCover(3, 3*x^3 + x).frobenius_matrix()
[          0           0       O(107)       O(107)]
[          0           0 79 + O(107)       O(107)]
[     O(107) 42 + O(107)            0            0]
[30 + O(107)      O(107)            0            0]
```

class sage.schemes.cyclic_covers.cycliccover_finite_field.**CyclicCover_finite_field**(*AA*, *r*, *f*, *names=None*, *verbose=0*)

Bases: *CyclicCover_generic*

EXAMPLES:

```
sage: p = 13
sage: x = PolynomialRing(GF(p), "x").gen()
sage: C = CyclicCover(4, x^4 + 1)
sage: C.frobenius_polynomial()
x^6 - 6*x^5 + 3*x^4 + 60*x^3 + 39*x^2 - 1014*x + 2197
sage: R.<t> = PowerSeriesRing(Integers())
sage: C.projective_closure().zeta_series(2,t)
1 + 8*t + 102*t^2 + O(t^3)
sage: C.frobenius_polynomial().reverse()(t)/((1-t)*(1-p*t)) + O(t^5)
1 + 8*t + 102*t^2 + 1384*t^3 + 18089*t^4 + O(t^5)
```

frobenius_matrix(*N=None*)

Compute p-adic Frobenius matrix to precision p^N.

If *N* not supplied, a default value is selected, which is the minimum needed to recover the charpoly unambiguously.

EXAMPLES:

```
sage: p = 107
sage: x = PolynomialRing(GF(p), "x").gen()
sage: CyclicCover(2, x^5 + x).frobenius_matrix()
[              O(107^2)       89*107 + O(107^2)                O(107^2)              ␣
↪    O(107^2)]
```

(continues on next page)

(continued from previous page)

```
[     89*107 + O(107^2)               O(107^2)               O(107^2)          ␣
↪    O(107^2)]
[             O(107^2)               O(107^2)               O(107^2) 105 +␣
↪5*107 + O(107^2)]
[             O(107^2)               O(107^2) 89 + 53*107 + O(107^2)          ␣
↪    O(107^2)]
sage: CyclicCover(2, 3*x^5 + x).frobenius_matrix()
[             O(107^2)      14*107 + O(107^2)               O(107^2)          ␣
↪    O(107^2)]
[     69*107 + O(107^2)               O(107^2)               O(107^2)          ␣
↪    O(107^2)]
[             O(107^2)               O(107^2)               O(107^2) 61 +␣
↪58*107 + O(107^2)]
[             O(107^2)               O(107^2) 69 + 53*107 + O(107^2)          ␣
↪    O(107^2)]
sage: CyclicCover(3, x^3 + x).frobenius_matrix()
[          0           0      O(107)      O(107)]
[          0           0 52 + O(107)      O(107)]
[     O(107) 35 + O(107)           0           0]
[44 + O(107)      O(107)           0           0]
sage: CyclicCover(3, 3*x^3 + x).frobenius_matrix()
[          0           0      O(107)      O(107)]
[          0           0 79 + O(107)      O(107)]
[     O(107) 42 + O(107)           0           0]
[30 + O(107)      O(107)           0           0]
```

frobenius_polynomial()

Return the characteristic polynomial of Frobenius.

EXAMPLES:

Hyperelliptic curves:

```
sage: p = 11
sage: x = PolynomialRing(GF(p), "x").gen()
sage: f = x^7 + 4*x^2 + 10*x + 4
sage: CyclicCover(2, f).frobenius_polynomial() == \
....: HyperellipticCurve(f).frobenius_polynomial()
True
sage: f = 2*x^5 + 4*x^3 + x^2 + 2*x + 1
sage: CyclicCover(2, f).frobenius_polynomial() == \
....: HyperellipticCurve(f).frobenius_polynomial()
True
sage: f = 2*x^6 + 4*x^4 + x^3 + 2*x^2 + x
sage: CyclicCover(2, f).frobenius_polynomial() == \
....: HyperellipticCurve(f).frobenius_polynomial()
True
sage: p = 1117
sage: x = PolynomialRing(GF(p), "x").gen()
sage: f = x^9 + 4*x^2 + 10*x + 4
sage: P1 = CyclicCover(2, f).frobenius_polynomial()
sage: P2 = HyperellipticCurve(f).frobenius_polynomial()
sage: P1 == P2  # long time
```

(continues on next page)

(continued from previous page)

```
True
sage: f = 2*x^5 + 4*x^3 + x^2 + 2*x + 1
sage: CyclicCover(2, f).frobenius_polynomial() == \
....: HyperellipticCurve(f).frobenius_polynomial()
True
```

Superelliptic curves:

```
sage: p = 11
sage: x = PolynomialRing(GF(p), "x").gen()
sage: CyclicCover(3, x^4 + 4*x^3 + 9*x^2 + 3*x + 1).frobenius_polynomial()
x^6 + 21*x^4 + 231*x^2 + 1331
sage: CyclicCover(4, x^3 + x + 1).frobenius_polynomial()
x^6 + 2*x^5 + 11*x^4 + 121*x^2 + 242*x + 1331
sage: p = 4999
sage: x = PolynomialRing(GF(p),"x").gen()
sage: CyclicCover(4, x^3  - 1).frobenius_polynomial() == \
....: CyclicCover(3, x^4  + 1).frobenius_polynomial()
True
sage: CyclicCover(3, x^4 + 4*x^3 + 9*x^2 + 3*x + 1).frobenius_polynomial()
x^6 + 180*x^5 + 20988*x^4 + 1854349*x^3 + 104919012*x^2 + 4498200180*x +␣
↪124925014999
sage: CyclicCover(4, x^5 + x + 1).frobenius_polynomial()
x^12 - 64*x^11 + 5018*x^10 - 488640*x^9 + 28119583*x^8 - 641791616*x^7
 + 124245485932*x^6 - 3208316288384*x^5 + 702708407289583*x^4 -␣
↪6104335932911360*x^3
 + 3133741752599645018*x^2 - 1998000799840001599936*x + 1560625937250037497000 1

sage: h = PolynomialRing(GF(1129), 'x')([-1] + [0]*(5-1) + [1])
sage: CyclicCover(11, h).frobenius_polynomial()  # long time
x^40 + 7337188909826596*x^30 + 20187877911930897108199045855206*x^20
 + 24687045654725446027864774006541463602997309796*x^10
 + 113208448496396499516088099735897769332031367650269635532584012

sage: h = PolynomialRing(GF(1009^2), 'x')([-1] + [0]*(5-1) + [1])
sage: CyclicCover(3, h).frobenius_polynomial()  # long time
x^8 + 532*x^7 - 2877542*x^6 - 242628176*x^5 + 4390163797795*x^4 -␣
↪247015136050256*x^3
 - 2982540407204025062*x^2 + 561382189105547134612*x + 1074309286591662654798721
```

A non-monic example checking that github issue #29015 is fixed:

```
sage: a = 3
sage: K.<s> = GF(83^3);
sage: R.<x> = PolynomialRing(K)
sage: h = s*x^4 + x^3 + 8
sage: C = CyclicCover(a, h)
sage: C.frobenius_polynomial()
x^6 + 1563486*x^4 + 893980969482*x^2 + 186940255267540403
```

Non-superelliptic curves:

```
sage: p = 13
sage: x = PolynomialRing(GF(p), "x").gen()
sage: C = CyclicCover(4, x^4 + 1)
sage: C.frobenius_polynomial()
x^6 - 6*x^5 + 3*x^4 + 60*x^3 + 39*x^2 - 1014*x + 2197
sage: R.<t> = PowerSeriesRing(Integers())
sage: C.projective_closure().zeta_series(2, t)
1 + 8*t + 102*t^2 + O(t^3)
sage: C.frobenius_polynomial().reverse()(t)/((1-t)*(1-p*t)) + O(t^5)
1 + 8*t + 102*t^2 + 1384*t^3 + 18089*t^4 + O(t^5)

sage: x = PolynomialRing(GF(11), "x").gen()
sage: CyclicCover(4, x^6 - 11*x^3 + 70*x^2 - x + 961).frobenius_polynomial()  #␣
↪long time
x^14 + 14*x^12 + 287*x^10 + 3025*x^8 + 33275*x^6 + 381997*x^4 + 2254714*x^2 +␣
↪19487171
sage: x = PolynomialRing(GF(4999), "x").gen()
sage: CyclicCover(4, x^6 - 11*x^3 + 70*x^2 - x + 961).frobenius_polynomial()  #␣
↪long time
x^14 - 4*x^13 - 2822*x^12 - 30032*x^11 + 37164411*x^10 - 152369520*x^9
 + 54217349361*x^8 - 1021791160888*x^7 + 271032529455639*x^6 -␣
↪3807714457169520*x^5
 + 4642764601604000589*x^4 - 18754988504199390032*x^3 -␣
↪880993477679457054717 8*x^2
 - 624250374900014998800004*x + 78015690603129374475034999

sage: p = 11
sage: x = PolynomialRing(GF(p), "x").gen()
sage: CyclicCover(3, 5*x^3 - 5*x + 13).frobenius_polynomial()
x^2 + 11
sage: CyclicCover(3, x^6 + x^4 - x^3 + 2*x^2 - x - 1).frobenius_polynomial()
x^8 + 32*x^6 + 462*x^4 + 3872*x^2 + 14641
sage: p = 4999
sage: x = PolynomialRing(GF(p), "x").gen()
sage: CyclicCover(3, 5*x^3 - 5*x + 13).frobenius_polynomial()
x^2 - 47*x + 4999
sage: CyclicCover(3, x^6 + x^4 - x^3 + 2*x^2 - x - 1).frobenius_polynomial()
x^8 + 122*x^7 + 4594*x^6 - 639110*x^5 - 82959649*x^4 - 3194910890*x^3
 + 114804064594*x^2 + 15240851829878*x + 624500149980001

sage: p = 11
sage: x = PolynomialRing(GF(p), "x").gen()
sage: CyclicCover(5, x^5 + x).frobenius_polynomial()  # long time
x^12 + 4*x^11 + 22*x^10 + 108*x^9 + 503*x^8 + 1848*x^7 + 5588*x^6 + 20328*x^5
 + 60863*x^4 + 143748*x^3 + 322102*x^2 + 644204*x + 1771561
sage: CyclicCover(5, 2*x^5 + x).frobenius_polynomial()  # long time
x^12 - 9*x^11 + 42*x^10 - 108*x^9 - 47*x^8 + 1782*x^7 - 8327*x^6 + 19602*x^5
 - 5687*x^4 - 143748*x^3 + 614922*x^2 - 1449459*x + 1771561
sage: p = 49999
sage: x = PolynomialRing(GF(p), "x").gen()
sage: CyclicCover(5, x^5 + x).frobenius_polynomial()  # long time
x^12 + 299994*x^10 + 37498500015*x^8 + 2499850002999980*x^6
 + 937425002249970000 15*x^4 + 187481250749985000014999 94*x^2
```

(continues on next page)

(continued from previous page)

```
 + 15623125093747500037499700001
sage: CyclicCover(5, 2*x^5 + x).frobenius_polynomial() # long time
x^12 + 299994*x^10 + 37498500015*x^8 + 2499850002999980*x^6
 + 937425002249970000015*x^4 + 187481250749985000014999994*x^2
 + 15623125093747500037499700001
```

17.2 Cyclic covers curves over a general ring

EXAMPLES:

```
sage: ZZx.<x> = ZZ[]
sage: C = CyclicCover(5, x^5 + x + 1); C
Cyclic Cover of P^1 over Integer Ring defined by y^5 = x^5 + x + 1
sage: C.genus()
6
sage: D = C.projective_closure(); D
Projective Plane Curve over Integer Ring defined by x0^5 + x0^4*x1 + x1^5 - x2^5
sage: D.change_ring(QQ).genus()
6
sage: C.change_ring(GF(5))
Traceback (most recent call last):
...
ValueError: As the characteristic divides the order of the cover, this model is not␣
↪smooth.

sage: GF7x.<x> = GF(7)[]
sage: C = CyclicCover(3, x^9 + x + 1)
sage: C
Cyclic Cover of P^1 over Finite Field of size 7 defined by y^3 = x^9 + x + 1
sage: C.genus()
7
sage: C.projective_closure()
Traceback (most recent call last):
...
NotImplementedError: Weighted Projective Space is not implemented
```

class sage.schemes.cyclic_covers.cycliccover_generic.**CyclicCover_generic**(*AA*, *r*, *f*, *names=None*)

Bases: AffinePlaneCurve

Cyclic covers over a general ring

INPUT:

- A - ambient affine space
- r - degree of the cover
- f - univariate polynomial
- names (default: ["x","y"]) - names for the coordinate functions

base_extend(*R*)

Return this CyclicCover over a new base ring R.

EXAMPLES:

```
sage: ZZx.<x> = ZZ[]
sage: C = CyclicCover(5, x^5 + x + 1)
sage: C.change_ring(GF(5))
Traceback (most recent call last):
...
ValueError: As the characteristic divides the order of the cover,
this model is not smooth.
sage: C.change_ring(GF(3))
Traceback (most recent call last):
...
ValueError: Not a smooth Cyclic Cover of P^1: singularity in the
provided affine patch.
sage: C.change_ring(GF(17))
Cyclic Cover of P^1 over Finite Field of size 17 defined by y^5 = x^5 + x + 1
```

change_ring(*R*)

Return this CyclicCover over a new base ring R.

EXAMPLES:

```
sage: ZZx.<x> = ZZ[]
sage: C = CyclicCover(5, x^5 + x + 1)
sage: C.change_ring(GF(5))
Traceback (most recent call last):
...
ValueError: As the characteristic divides the order of the cover,
this model is not smooth.
sage: C.change_ring(GF(3))
Traceback (most recent call last):
...
ValueError: Not a smooth Cyclic Cover of P^1: singularity in the
provided affine patch.
sage: C.change_ring(GF(17))
Cyclic Cover of P^1 over Finite Field of size 17 defined by y^5 = x^5 + x + 1
```

cover_polynomial(*K=None*, *var='x'*)

Return the polynomial defining the cyclic cover.

EXAMPLES:

```
sage: ZZx.<x> = ZZ[]; CyclicCover(5, x^5 + x + 1).cover_polynomial()
x^5 + x + 1
```

genus()

The geometric genus of the curve.

EXAMPLES:

```
sage: ZZx.<x> = ZZ[]
sage: CyclicCover(5, x^5 + x + 1).genus()
```

(continues on next page)

(continued from previous page)

```
6
sage: CyclicCover(3, x^5 + x + 1).genus()
4
```

is_singular()

Return if this curve is singular or not.

This just checks that the characteristic of the ring does not divide the order of the cover and that the defining polynomial of the cover is square free.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: CyclicCover(3, x^5 + x + 1).is_singular()
False
sage: CyclicCover(3, (x^5 + x + 1)^2, check_smooth=False).is_singular()
True
```

is_smooth()

Return if this curve is smooth or not.

This just checks that the characteristic of the ring does not divide the order of the cover and that the defining polynomial of the cover is square free.

EXAMPLES:

```
sage: R.<x> = QQ[]
sage: CyclicCover(3, x^5 + x + 1).is_smooth()
True
sage: CyclicCover(3, (x^5 + x + 1)^2, check_smooth=False).is_smooth()
False
```

order()

The order of the cover.

EXAMPLES:

```
sage: ZZx.<x> = ZZ[]
sage: CyclicCover(5, x^5 + x + 1).order()
5
sage: CyclicCover(3, x^5 + x + 1).order()
3
```

projective_closure(***kwds*)

Return the projective closure of this affine curve.

EXAMPLES:

```
sage: GF7x.<x> = GF(7)[]
sage: CyclicCover(3, x^9 + x + 1).projective_closure()
Traceback (most recent call last):
...
NotImplementedError: Weighted Projective Space is not implemented

sage: ZZx.<x> = ZZ[]
```

(continues on next page)

(continued from previous page)

```
sage: CyclicCover(5, x^5 + x + 1).projective_closure()
Projective Plane Curve over Integer Ring defined by x0^5 + x0^4*x1 + x1^5 - x2^5
```

17.3 Computation of the Frobenius polynomial using Newton's identities

sage.schemes.cyclic_covers.charpoly_frobenius.**charpoly_frobenius**(*frob_matrix*, *charpoly_prec*, *p*, *weight*, *a=1*, *known_factor=[1]*)

Return the characteristic polynomial of the given Frobenius matrix.

INPUT:

- `frob_matrix` – a matrix representing the Frobenius matrix up to some precision
- `charpoly_prec` – a vector ai, such that, `frob_matrix.change_ring(ZZ).charpoly()[i]` will be correct mod $p^a i$, this can be easily deduced from the Hodge numbers and knowing the q-adic precision of `frob_matrix`
- `p` – prime p
- `weight` – weight of the motive
- `a` – $p = q^a$
- `known_factor` – the list of coefficients of the known factor

OUTPUT:

A list of integers corresponding to the characteristic polynomial of the Frobenius action.

EXAMPLES:

```
sage: from sage.schemes.cyclic_covers.charpoly_frobenius import charpoly_frobenius
sage: M = Matrix([[O(17), 8 + O(17)], [O(17), 15 + O(17)]])
sage: charpoly_frobenius(M, [2, 1, 1], 17, 1, 1)
[17, 2, 1]

sage: R = Zq(17**2, names=('a',))
sage: M = Matrix(R, [[8*17 + 16*17**2 + O(17**3), 8 + 11*17 + O(17**2)],
....:                [7*17**2 + O(17**3), 15 + 8*17 + O(17**2)]])
sage: charpoly_frobenius(M*M, [3, 2, 2], 17, 1, 2)
[289, 30, 1]

sage: M = Matrix([[8*31 + 8*31**2 + O(31**3), O(31**3), O(31**3), O(31**3)],
....:             [O(31**3), 23*31 + 22*31**2 + O(31**3), O(31**3), O(31**3)],
....:             [O(31**3), O(31**3), 27 + 7*31 + O(31**3), O(31**3)],
....:             [O(31**3), O(31**3), O(31**3), 4 + 23*31 + O(31**3)]])
sage: charpoly_frobenius(M, [4, 3, 2, 2, 2], 31, 1, 1)
[961, 0, 46, 0, 1]

sage: M = Matrix([(4*43^2 + O(43^3), 17*43 + 11*43^2 + O(43^3), O(43^3), O(43^3),␣
↪17 + 37*43 + O(43^3), O(43^3)),
....:             (30*43 + 23*43^2 + O(43^3), 5*43 + O(43^3), O(43^3), O(43^3), 3 +␣
```

(continues on next page)

(continued from previous page)

```
↪38*43 + O(43^3), O(43^3)),
....:              (O(43^3), O(43^3), 9*43 + 32*43^2 + O(43^3), 13 + 25*43 + O(43^3),
↪ O(43^3), 17 + 18*43 + O(43^3)),
....:              (O(43^3), O(43^3), 22*43 + 25*43^2 + O(43^3), 11 + 24*43 + O(43^
↪3), O(43^3), 36 + 5*43 + O(43^3)),
....:              (42*43 + 15*43^2 + O(43^3), 22*43 + 8*43^2 + O(43^3), O(43^3),␣
↪O(43^3), 29 + 4*43 + O(43^3), O(43^3)),
....:              (O(43^3), O(43^3), 6*43 + 19*43^2 + O(43^3), 8 + 24*43 + O(43^3),␣
↪O(43^3), 31 + 42*43 + O(43^3))])
sage: charpoly_frobenius(M, [5, 4, 3, 2, 2, 2, 2], 43, 1, 1)
    [79507, 27735, 6579, 1258, 153, 15, 1]

sage: M = Matrix([(1 + O(4999), O(4999), 0, 0),
....:              (O(4999), 4860 + O(4999), 0, 0),
....:              (0, 0, O(4999), O(4999)),
....:              (0, 0, O(4999), 1 + O(4999))])
sage: charpoly_frobenius(M, [2, 1, 1], 4999, 1, 1, [1, -2, 1])
[4999, 139, 1]
```

17.4 Cyclic cover curve constructor

sage.schemes.cyclic_covers.constructor.**CyclicCover**(*r*, *f*, *names=None*, *check_smooth=True*)

Return the cyclic cover of the projective line given by $y^r = f$, for a univariate polynomial f.

INPUT:

- `r` - the order of the cover
- `f` - univariate polynomial if not given, then it defaults to 0.
- `names` (default: `["x","y"]`) - names for the coordinate functions
- `check_squarefree` (default: `True`) - test if the input defines a unramified cover of the projective line.

Warning: When setting `check_smooth=False` or using a base ring that is not a field, the output curves are not to be trusted. For example, the output of `is_singular` or `is_smooth` only tests smoothness over the field of fractions.

Note: The words "cyclic cover" are usually used for covers of degree greater than two. We usually refer to smooth double covers of the projective line as "hyperelliptic curves" or "elliptic curves" if the genus is one. We allow such cases in this implementation, but we highly recommend to use the more specific constructors/classes HyperellipticCurve and EllipticCurve for a wider range of tools.

EXAMPLES:

Basic examples:

```
sage: R.<x> = QQ[]
sage: CyclicCover(2, x^5 + x + 1)
Cyclic Cover of P^1 over Rational Field defined by y^2 = x^5 + x + 1
```

(continues on next page)

(continued from previous page)

```
sage: CyclicCover(3, x^5 + x + 1)
Cyclic Cover of P^1 over Rational Field defined by y^3 = x^5 + x + 1
sage: CyclicCover(5, x^5 + x + 1)
Cyclic Cover of P^1 over Rational Field defined by y^5 = x^5 + x + 1
sage: CyclicCover(15, x^9 + x + 1)
Cyclic Cover of P^1 over Rational Field defined by y^15 = x^9 + x + 1

sage: k.<a> = GF(9); R.<x> = k[]                                                #␣
↪needs sage.rings.finite_rings
sage: CyclicCover(5, x^9 + x + 1)                                               #␣
↪needs sage.rings.finite_rings
Cyclic Cover of P^1 over Finite Field in a of size 3^2
 defined by y^5 = x^9 + x + 1
sage: CyclicCover(15, x^9 + x + 1)                                              #␣
↪needs sage.rings.finite_rings
Traceback (most recent call last):
...
ValueError: As the characteristic divides the order of the cover,
this model is not smooth.
```

We can change the names of the variables in the output:

```
sage: k.<a> = GF(9); R.<x> = k[]                                                #␣
↪needs sage.rings.finite_rings
sage: CyclicCover(5, x^9 + x + 1, names=["A","B"])                              #␣
↪needs sage.rings.finite_rings
Cyclic Cover of P^1 over Finite Field in a of size 3^2
 defined by B^5 = A^9 + A + 1
```

Double roots:

```
sage: P.<x> = GF(7)[]
sage: CyclicCover(2, (x^3-x+2)^2*(x^6-1))
Traceback (most recent call last):
...
ValueError: Not a smooth Cyclic Cover of P^1: singularity in the provided affine␣
↪patch.

sage: CyclicCover(2, (x^3-x+2)^2*(x^6-1), check_smooth=False)                   #␣
↪needs sage.rings.finite_rings
Cyclic Cover of P^1 over Finite Field of size 7
 defined by y^2 = x^12 - 2*x^10 - 3*x^9 + x^8 + 3*x^7 + 3*x^6
                  + 2*x^4 + 3*x^3 - x^2 - 3*x + 3
```

Input with integer coefficients creates objects with the integers as base ring, but only checks smoothness over $\mathbf{Q}$, not over Spec($\mathbf{Z}$). In other words, it is checked that the discriminant is non-zero, but it is not checked whether the discriminant is a unit in $\mathbf{Z}^*$:

```
sage: R.<x> = ZZ[]
sage: CyclicCover(5, (x^3-x+2)*(x^6-1))
Cyclic Cover of P^1 over Integer Ring defined by y^5 = x^9 - x^7 + 2*x^6 - x^3 + x -
↪ 2
```

CHAPTER

EIGHTEEN

BERKOVICH ANALYTIC SPACE

18.1 Elements of Berkovich space.

Berkovich_Element is an abstract parent class for elements of any Berkovich space.

Berkovich_Element_Cp_Affine and *Berkovich_Element_Cp_Projective* implement elements of Berkovich space over $\mathbf{C}_p$ and $P^1(\mathbf{C}_p)$. Elements are determined by specific data and fall into one of the four following types:

- Type I points are represented by a center.
- Type II points are represented by a center and a rational power of p.
- Type III points are represented by a center and a non-negative real radius.
- Type IV points are represented by a finite list of centers and a finite list of non-negative radii.

For an exposition of Berkovich space over $\mathbf{C}_p$, see Chapter 6 of [Ben2019]. For a more involved exposition, see Chapter 1 and 2 of [BR2010].

AUTHORS:

- Alexander Galarraga (2020-06-22): initial implementation

class sage.schemes.berkovich.berkovich_cp_element.**Berkovich_Element**

Bases: Element

The parent class for any element of a Berkovich space.

class sage.schemes.berkovich.berkovich_cp_element.**Berkovich_Element_Cp**(*parent*, *center*, *radius=None*, *power=None*, *prec=20*, *space_type=None*, *error_check=True*)

Bases: *Berkovich_Element*

The abstract parent class for any element of Berkovich space over $\mathbf{C}_p$.

This class should never be instantiated, instead use *Berkovich_Element_Cp_Affine* or *Berkovich_Element_Cp_Projective*.

EXAMPLES:

```
sage: B = Berkovich_Cp_Affine(3)
sage: B(2)
Type I point centered at 2 + O(3^20)
```

```
sage: B(0, 1)
Type II point centered at 0 of radius 3^0
```

Hsia_kernel(*other*, *basepoint*)

The Hsia kernel of this point and `other`, with basepoint `basepoint`.

The Hsia kernel with arbitrary basepoint is a generalization of the Hsia kernel at infinity.

INPUT:

- `other` – A point of the same Berkovich space as this point.
- `basepoint` – A point of the same Berkovich space as this point.

OUTPUT: A finite or infinite real number.

EXAMPLES:

```
sage: B = Berkovich_Cp_Projective(3)
sage: Q1 = B(2, 9)
sage: Q2 = B(1/27, 1/27)
sage: Q3 = B(1, 1/3)
sage: Q1.Hsia_kernel(Q2, Q3)
0.111111111111111
```

```
sage: B = Berkovich_Cp_Projective(3)
sage: Q1 = B(2, 9)
sage: Q2 = B(1/2)
sage: Q3 = B(1/2)
sage: Q1.Hsia_kernel(Q2, Q3)
+infinity
```

Hsia_kernel_infinity(*other*)

Return the Hsia kernel at infinity of this point with `other`.

The Hsia kernel at infinity is the natural extension of the absolute value on $\mathbf{C}_p$ to Berkovich space.

INPUT:

- `other` – A point of the same Berkovich space as this point.

OUTPUT: A real number.

EXAMPLES:

```
sage: B = Berkovich_Cp_Affine(Qp(3))
sage: Q1 = B(1/4, 4)
sage: Q2 = B(1/4, 6)
sage: Q1.Hsia_kernel_infinity(Q2)
6.00000000000000
```

```
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: A.<a> = NumberField(x^3 + 20)
sage: ideal = A.ideal(-1/2*a^2 + a - 3)
sage: B = Berkovich_Cp_Projective(A, ideal)
sage: Q1 = B(4)
```

(continues on next page)

(continued from previous page)

```
sage: Q2 = B(0, 1.5)
sage: Q1.Hsia_kernel_infinity(Q2)
1.50000000000000
```

big_metric(*other*)

Return the path distance metric distance between this point and `other`.

Also referred to as the hyperbolic metric, or the big metric.

On the set of type II, III and IV points, the path distance metric is a metric. Following Baker and Rumely, we extend the path distance metric to type I points x, y by $\rho(x, x) = 0$ and $\rho(x, y) = \infty$. See [BR2010].

INPUT:

- `other` – A point of the same Berkovich space as this point.

OUTPUT: A finite or infinite real number.

EXAMPLES:

```
sage: B = Berkovich_Cp_Affine(3)
sage: Q1 = B(1/4, 4)
sage: Q2 = B(1/4, 6)
sage: Q1.path_distance_metric(Q2)
0.369070246428542
```

```
sage: Q3 = B(1)
sage: Q3.path_distance_metric(Q1)
+infinity
```

```
sage: Q3.path_distance_metric(Q3)
0
```

center()

Return the center of the corresponding disk (or sequence of disks) in $\mathbf{C}_p$.

OUTPUT: An element of the `base` of the parent Berkovich space.

EXAMPLES:

```
sage: B = Berkovich_Cp_Affine(3)
sage: B(3, 1).center()
3 + O(3^21)
```

```
sage: C = Berkovich_Cp_Projective(3)
sage: C(3, 1).center()
(3 + O(3^21) : 1 + O(3^20))
```

```
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: A.<a> = NumberField(x^3 + 20)
sage: ideal = A.ideal(-1/2*a^2 + a - 3)
sage: B = Berkovich_Cp_Projective(A, ideal)
sage: B(a^2 + 4).center()
(a^2 + 4 : 1)
```

center_function()

Return the function defining the centers of disks in the approximation.

Not defined unless this point is a type IV point created by using a univariate function to compute centers.

OUTPUT: A univariate function.

EXAMPLES:

```
sage: B = Berkovich_Cp_Projective(5)
sage: L.<t> = PolynomialRing(Qp(5))
sage: T = FractionField(L)
sage: f = T(1/t)
sage: R.<x> = RR[]
sage: Y = FractionField(R)
sage: g = (40*pi)/x                                                    #␣
↪needs sage.symbolic
sage: Q1 = B(f, g)                                                     #␣
↪needs sage.symbolic
sage: Q1.center_function()                                             #␣
↪needs sage.symbolic
(1 + O(5^20))/((1 + O(5^20))*t)
```

diameter(*basepoint=+Infinity*)

Generalized diameter function on Berkovich space.

If the basepoint is infinity, the diameter is equal to the limit of the radii of the corresponding disks in $\mathbf{C}_p$.

If the basepoint is not infinity, the diameter is the Hsia kernel of this point with itself at basepoint `basepoint`.

INPUT:

- `basepoint` – (default = Infinity) A point of the same Berkovich space as this point.

OUTPUT: A real number.

EXAMPLES:

```
sage: B = Berkovich_Cp_Affine(3)
sage: Q1 = B(3)
sage: Q1.diameter()
0
```

```
sage: Q2 = B(1/2, 9)
sage: Q2.diameter()
9.00000000000000
```

The diameter of a type IV point is the limit of the radii:

```
sage: R.<x> = PolynomialRing(Qp(3))
sage: f = R(2)
sage: S.<y> = PolynomialRing(RR)
sage: S = FractionField(S)
sage: g = (y+1)/y
sage: B(f,g).diameter()
1.0
```

```
sage: B = Berkovich_Cp_Affine(3)
sage: Q1 = B(1/81, 1)
sage: Q2 = B(1/3)
sage: Q1.diameter(Q2)
0.00137174211248285
```

```
sage: Q2.diameter(Q2)
+infinity
```

hyperbolic_metric(*other*)

Return the path distance metric distance between this point and `other`.

Also referred to as the hyperbolic metric, or the big metric.

On the set of type II, III and IV points, the path distance metric is a metric. Following Baker and Rumely, we extend the path distance metric to type I points x, y by $\rho(x, x) = 0$ and $\rho(x, y) = \infty$. See [BR2010].

INPUT:

- `other` – A point of the same Berkovich space as this point.

OUTPUT: A finite or infinite real number.

EXAMPLES:

```
sage: B = Berkovich_Cp_Affine(3)
sage: Q1 = B(1/4, 4)
sage: Q2 = B(1/4, 6)
sage: Q1.path_distance_metric(Q2)
0.369070246428542
```

```
sage: Q3 = B(1)
sage: Q3.path_distance_metric(Q1)
+infinity
```

```
sage: Q3.path_distance_metric(Q3)
0
```

ideal()

The ideal which defines an embedding of the `base_ring` into $\mathbf{C}_p$.

If this Berkovich space is backed by a p-adic field, then an embedding is already specified, and this returns `None`.

EXAMPLES:

```
sage: B = Berkovich_Cp_Projective(QQ, 3)
sage: B(0).ideal()
3

::

sage: B = Berkovich_Cp_Projective(3)
sage: B(0).ideal()
```

path_distance_metric(*other*)

Return the path distance metric distance between this point and `other`.

Also referred to as the hyperbolic metric, or the big metric.

On the set of type II, III and IV points, the path distance metric is a metric. Following Baker and Rumely, we extend the path distance metric to type I points x, y by $\rho(x, x) = 0$ and $\rho(x, y) = \infty$. See [BR2010].

INPUT:

- `other` – A point of the same Berkovich space as this point.

OUTPUT: A finite or infinite real number.

EXAMPLES:

```
sage: B = Berkovich_Cp_Affine(3)
sage: Q1 = B(1/4, 4)
sage: Q2 = B(1/4, 6)
sage: Q1.path_distance_metric(Q2)
0.369070246428542
```

```
sage: Q3 = B(1)
sage: Q3.path_distance_metric(Q1)
+infinity
```

```
sage: Q3.path_distance_metric(Q3)
0
```

potential_kernel(*other*, *basepoint*)

The potential kernel of this point with `other`, with basepoint `basepoint`.

The potential kernel is the hyperbolic distance between `basepoint` and the join of this point with `other` relative to `basepoint`.

INPUT:

- `other` – A point of the same Berkovich space as this point.
- `basepoint` – A point of the same Berkovich space as this point.

OUTPUT: A finite or infinite real number.

EXAMPLES:

```
sage: B = Berkovich_Cp_Projective(3)
sage: Q1 = B(27, 1)
sage: Q2 = B(1/3, 2)
sage: Q3 = B(1/9, 1/2)
sage: Q3.potential_kernel(Q1, Q2)
0.369070246428543
```

```
sage: B = Berkovich_Cp_Affine(3)
sage: Q1 = B(27, 1)
sage: Q2 = B(1/3, 2)
sage: Q3 = B(1/9, 1/2)
sage: Q3.potential_kernel(Q1, Q2)
0.369070246428543
```

power()

The power of p such that $p^{\text{power}} = \text{radius}$.

For type II points, always in $\mathbf{Q}$. For type III points, a real number. Not defined for type I or IV points.

OUTPUT:

- A rational for type II points.
- A real number for type III points.

EXAMPLES:

```
sage: B = Berkovich_Cp_Affine(3)
sage: Q1 = B(1, 9)
sage: Q1.power()
2
```

```
sage: Q2 = B(1, 4)
sage: Q2.power()
1.26185950714291
```

prec()

Return the precision of a type IV point.

This integer is the number of disks used in the approximation of the type IV point. Not defined for type I, II, or III points.

OUTPUT: An integer.

EXAMPLES:

```
sage: B = Berkovich_Cp_Affine(Qp(3))
sage: d = B([2, 2, 2], [1.761, 1.123, 1.112])
sage: d.precision()
3
```

precision()

Return the precision of a type IV point.

This integer is the number of disks used in the approximation of the type IV point. Not defined for type I, II, or III points.

OUTPUT: An integer.

EXAMPLES:

```
sage: B = Berkovich_Cp_Affine(Qp(3))
sage: d = B([2, 2, 2], [1.761, 1.123, 1.112])
sage: d.precision()
3
```

prime()

The residue characteristic of the parent.

OUTPUT: A prime integer.

EXAMPLES:

```
sage: B = Berkovich_Cp_Affine(3)
sage: B(1).prime()
3
```

radius()

Radius of the corresponding disk (or sequence of disks) in $\mathbf{C}_p$.

OUTPUT:

- A non-negative real number for type I, II, or III points.
- A list of non-negative real numbers for type IV points.

EXAMPLES:

```
sage: B = Berkovich_Cp_Affine(3)
sage: Q1 = B(1, 2/5)
sage: Q1.radius()
0.400000000000000
```

```
sage: d = B([2, 2, 2], [1.761, 1.123, 1.112])
sage: d.radius()
[1.76100000000000, 1.12300000000000, 1.11200000000000]
```

radius_function()

Return the function defining the radii of disks in the approximation.

Not defined unless this point is a type IV point created by using a univariate function to compute radii.

OUTPUT: A univariate function.

EXAMPLES:

```
sage: B = Berkovich_Cp_Projective(5)
sage: L.<t> = PolynomialRing(Qp(5))
sage: T = FractionField(L)
sage: f = T(1/t)
sage: R.<x> = RR[]
sage: Y = FractionField(R)
sage: g = (40*pi)/x                                                    #␣
↪needs sage.symbolic
sage: Q1 = B(f, g)                                                     #␣
↪needs sage.symbolic
sage: Q1.radius_function()                                             #␣
↪needs sage.symbolic
40.0000000000000*pi/x
```

small_metric(*other*)

Return the small metric distance between this point and `other`.

The small metric is an extension of twice the spherical distance on $P^1(\mathbf{C}_p)$.

INPUT:

- `other` – A point of the same Berkovich space as this point.

OUTPUT: A real number.

EXAMPLES:

```
sage: B = Berkovich_Cp_Affine(3)
sage: Q1 = B(1/4, 4)
sage: Q2 = B(1/4, 6)
sage: Q1.small_metric(Q2)
0.0833333333333333
```

```
sage: B = Berkovich_Cp_Projective(QQ, 5)
sage: Q1 = B(0, 1)
sage: Q2 = B(99)
sage: Q1.small_metric(Q2)
1.00000000000000
```

```
sage: Q3 = B(1/4, 4)
sage: Q3.small_metric(Q2)
1.75000000000000
```

```
sage: Q2.small_metric(Q3)
1.75000000000000
```

spherical_kernel(*other*)

The spherical kernel of this point with `other`.

The spherical kernel is one possible extension of the spherical distance on $P^1(\mathbf{C}_p)$ to the projective Berkovich line. See [BR2010] for details.

INPUT:

- `other` – A point of the same Berkovich space as this point.

OUTPUT: A real number.

EXAMPLES:

```
sage: B = Berkovich_Cp_Projective(3)
sage: Q1 = B(2, 2)
sage: Q2 = B(1/9, 1)
sage: Q1.spherical_kernel(Q2)
0.500000000000000
```

```
sage: Q3 = B(2)
sage: Q3.spherical_kernel(Q3)
0
```

type_of_point()

Return the type of this point of Berkovich space over $\mathbf{C}_p$.

OUTPUT: An integer between 1 and 4 inclusive.

EXAMPLES:

```
sage: B = Berkovich_Cp_Affine(3)
sage: B(1).type_of_point()
1
```

```
sage: B(0, 1).type_of_point()
2
```

class sage.schemes.berkovich.berkovich_cp_element.**Berkovich_Element_Cp_Affine**(*parent*, *center*, *radius=None*, *power=None*, *prec=20*, *error_check=True*)

Bases: *Berkovich_Element_Cp*

Element class of the Berkovich affine line over $\mathbf{C}_p$.

Elements are categorized into four types, represented by specific data:

- Type I points are represented by a center in the `base` of the parent Berkovich space, which is $\mathbf{Q}_p$, a finite extension of $\mathbf{Q}_p$, or a number field.
- Type II points are represented by a center in the `base` of the parent Berkovich space, and a rational power of p.
- Type III points are represented by a center in the `base` of the parent Berkovich space, and a radius, a real number in $[0, \infty)$.
- Type IV points are represented by a finite list of centers in the `base` of the parent Berkovich space and a finite list of radii in $[0, \infty)$. Type IV points can be created from univariate functions, allowing for arbitrary precision.

INPUT:

- `center` – For type I, II, and III points, the center of the corresponding disk in $\mathbf{C}_p$. If the parent Berkovich space was created using a number field K, then `center` must be an element of K. Otherwise, `center` must be an element of a p-adic field. For type IV points, can be a list of centers used to approximate the point or a univariate function that computes the centers (computation starts at 1).
- `radius` – (optional) For type I, II, and III points, the radius of the corresponding disk in `Cp`. Must coerce into the real numbers. For type IV points, can be a list of radii used to approximate the point or a univariate function that computes the radii (computation starts at 1).
- `power` – (optional) Rational number. Used for constructing type II points; specifies the power of `p` such that p^{power} = radius.
- `prec` – (default: 20) The number of disks to be used to approximate a type IV point.
- `error_check` – (default: True) If error checking should be run on input. If input is correctly formatted, can be set to `False` for better performance. WARNING: with error check set to `False`, any error in the input will lead to incorrect results.

EXAMPLES:

Type I points can be created by specifying the corresponding point of `Cp`:

```
sage: B = Berkovich_Cp_Affine(Qp(3))
sage: B(4)
Type I point centered at 1 + 3 + O(3^20)
```

The center of a point can be an element of a finite extension of `Qp`:

```
sage: A.<t> = Qq(27)
sage: B(1 + t)
Type I point centered at (t + 1) + O(3^20)
```

Type II and III points can be created by specifying a center and a radius:

```
sage: B(2, 3**(1/2))                                                    #␣
↪needs sage.symbolic
Type II point centered at 2 + O(3^20) of radius 3^1/2
```

```
sage: B(2, 1.6)
Type III point centered at 2 + O(3^20) of radius 1.60000000000000
```

Some type II points may be mistaken for type III points:

```
sage: B(3, 3**0.5)                        # not tested
Type III point centered at 3 + O(3^21) of radius 1.73205080756888
```

To avoid these errors, specify the power instead of the radius:

```
sage: B(3, power=RR(1/100000))
Type II point centered at 3 + O(3^21) of radius 3^1/100000
```

Type IV points can be constructed in a number of ways, the first being from a list of centers and radii used to approximate the point:

```
sage: B([Qp(3)(2), Qp(3)(2), Qp(3)(2)], [1.761, 1.123, 1.112])
Type IV point of precision 3, approximated by disks centered at
[2 + O(3^20), 2 + O(3^20)] ... with radii [1.76100000000000, 1.12300000000000] ...
```

Type IV points can be constructed from univariate functions, with arbitrary precision:

```
sage: A.<t> = Qq(27)
sage: R.<x> = PolynomialRing(A)
sage: f = (1 + t)^2*x
sage: S.<y> = PolynomialRing(RR)
sage: S = FractionField(S)
sage: g = (y + 1)/y
sage: d = B(f, g, prec=100); d
Type IV point of precision 100 with centers given by
((t^2 + 2*t + 1) + O(3^20))*x and radii given by (y + 1.00000000000000)/y
```

For increased performance, `error_check` can be set to `False`. WARNING: with error check set to `False`, any error in the input will lead to incorrect results:

```
sage: B(f, g, prec=100, error_check=False)
Type IV point of precision 100 with centers given by
((t^2 + 2*t + 1) + O(3^20))*x and radii given by (y + 1.00000000000000)/y
```

When creating a Berkovich space backed by a number field, points can be created similarly:

```
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: A.<a> = NumberField(x^3 + 20)
sage: ideal = A.prime_above(3)
sage: B = Berkovich_Cp_Projective(A, ideal)
sage: Q1 = B(a); Q1
Type I point centered at (a : 1)
```

```
sage: B(a + 1, 3)                                                          #␣
↪needs sage.rings.number_field
Type II point centered at (a + 1 : 1) of radius 3^1
```

as_projective_point()

Return the corresponding point of projective Berkovich space.

We identify affine Berkovich space with the subset $P^1_{\text{Berk}}(C_p) - \{(1:0)\}$.

EXAMPLES:

```
sage: B = Berkovich_Cp_Affine(5)
sage: B(5).as_projective_point()
Type I point centered at (5 + O(5^21) : 1 + O(5^20))
```

```
sage: B(0, 1).as_projective_point()
Type II point centered at (0 : 1 + O(5^20)) of radius 5^0
```

```
sage: L.<t> = PolynomialRing(Qp(5))
sage: T = FractionField(L)
sage: f = T(1/t)
sage: R.<x> = RR[]
sage: Y = FractionField(R)
sage: g = (40*pi)/x                                                        #␣
↪needs sage.symbolic
sage: Q2 = B(f, g)                                                         #␣
↪needs sage.symbolic
sage: Q2.as_projective_point()                                             #␣
↪needs sage.symbolic
Type IV point of precision 20 with centers given by (1 + O(5^20))/((1 + O(5^
↪20))*t)
and radii given by 40.0000000000000*pi/x
```

contained_in_interval(*start*, *end*)

Check if this point is an element of the interval [`start`, `end`].

INPUT:

- `start` – A point of the same Berkovich space as this point.
- `end` – A point of the same Berkovich space as this point.

OUTPUT:

- `True` if this point is an element of [`start`, `end`].
- `False` otherwise.

EXAMPLES:

```
sage: B = Berkovich_Cp_Projective((3))
sage: Q1 = B(2, 1)
sage: Q2 = B(2, 4)
sage: Q3 = B(1/3)
sage: Q2.contained_in_interval(Q1, Q3.join(Q1))
False
```

```
sage: Q4 = B(1/81, 1)
sage: Q2.contained_in_interval(Q1, Q4.join(Q1))
True
```

gt(*other*)

Return `True` if this point is strictly greater than `other` in the standard partial order.

Roughly, the partial order corresponds to containment of the corresponding disks in $\mathbf{C}_p$.

For example, let x and y be points of type II or III. If x has center c_1 and radius r_1 and y has center c_2 and radius r_2, $x < y$ if and only if $D(c_1, r_1)$ is a subset of $D(c_2, r_2)$ in $\mathbf{C}_p$.

INPUT:

- `other` – A point of the same Berkovich space as this point.

OUTPUT:

- `True` – If this point is greater than `other` in the standard partial order.
- `False` – Otherwise.

EXAMPLES:

```
sage: B = Berkovich_Cp_Affine(QQ, 3)
sage: Q1 = B(5, 3)
sage: Q2 = B(5, 1)
sage: Q1.gt(Q2)
True
```

```
sage: Q3 = B(1/27)
sage: Q1.gt(Q3)
False
```

involution_map()

Return the image of this point under the involution map.

The involution map is the extension of the map `z |-> 1/z` on $\mathbf{C}_p$ to Berkovich space.

For affine Berkovich space, not defined for the type I point centered at 0.

If zero is contained in every disk approximating a type IV point, then the image under the involution map is not defined. To avoid this error, increase precision.

OUTPUT: A point of the same Berkovich space.

EXAMPLES:

The involution map is 1/z on type I points:

```
sage: B = Berkovich_Cp_Affine(3)
sage: Q1 = B(1/2)
sage: Q1.involution_map()
Type I point centered at 2 + O(3^20)
```

```
sage: Q2 = B(0, 1/3)
sage: Q2.involution_map()
Type II point centered at 0 of radius 3^1
```

```
sage: Q3 = B(1/3, 1/3)
sage: Q3.involution_map()
Type II point centered at 3 + O(3^21) of radius 3^-3
```

join(*other*, *basepoint=+Infinity*)

Compute the join of this point and `other` with respect to `basepoint`.

The join is first point that lies on the intersection of the path from this point to `basepoint` and the path from `other` to `basepoint`.

INPUT:

- `other` – A point of the same Berkovich space as this point.
- `basepoint` – (default: Infinity) A point of the same Berkovich space as this point or Infinity.

OUTPUT: A point of the same Berkovich space.

EXAMPLES:

```
sage: B = Berkovich_Cp_Affine(3)
sage: Q1 = B(2, 1)
sage: Q2 = B(2, 2)
sage: Q1.join(Q2)
Type III point centered at 2 + O(3^20) of radius 2.000000000000000
```

```
sage: Q3 = B(5)
sage: Q3.join(Q1)
Type II point centered at 2 + 3 + O(3^20) of radius 3^0
```

```
sage: Q3.join(Q1, basepoint=Q2)
Type II point centered at 2 + O(3^20) of radius 3^0
```

lt(*other*)

Return `True` if this point is strictly less than `other` in the standard partial order.

Roughly, the partial order corresponds to containment of the corresponding disks in `Cp`.

For example, let x and y be points of type II or III. If x has center c_1 and radius r_1 and y has center c_2 and radius r_2, $x < y$ if and only if $D(c_1, r_1)$ is a subset of $D(c_2, r_2)$ in $\mathbf{C}_p$.

INPUT:

- `other` – A point of the same Berkovich space as this point.

OUTPUT:

- `True` – If this point is less than `other` in the standard partial order.
- `False` – Otherwise.

EXAMPLES:

```
sage: B = Berkovich_Cp_Projective(3)
sage: Q1 = B(5, 0.5)
sage: Q2 = B(5, 1)
sage: Q1.lt(Q2)
True
```

```
sage: Q3 = B(1)
sage: Q1.lt(Q3)
False
```

class sage.schemes.berkovich.berkovich_cp_element.**Berkovich_Element_Cp_Projective**(*parent*, *center*, *radius=None*, *power=None*, *prec=20*, *error_check=True*)

Bases: *Berkovich_Element_Cp*

Element class of the Berkovich projective line over $\mathbf{C}_p$.

Elements are categorized into four types, represented by specific data:

- Type I points are represented by a center in the `base` of the parent Berkovich space, which is projective space of dimension 1 over either $\mathbf{Q}_p$, a finite extension of $\mathbf{Q}_p$, or a number field.
- Type II points are represented by a center in the `base` of the parent Berkovich space, and a rational power of p.
- Type III points are represented by a center in the `base` of the parent Berkovich space, and by a radius, a real number, in $[0, \infty)$.
- Type IV points are represented by a finite list of centers in the `base` of the parent Berkovich space and a finite list of radii in $[0, \infty)$.

The projective Berkovich line is viewed as the one-point compactification of the affine Berkovich line. The projective Berkovich line therefore contains every point of the affine Berkovich line, along with a type I point centered at infinity.

INPUT:

- `center` – For type I, II, and III points, the center of the corresponding disk in $P^1(\mathbf{C}_p)$. If the parent Berkovich space was created using a number field K, then `center` can be an element of $P^1(K)$. Otherwise, `center` must be an element of a projective space of dimension 1 over a padic field. For type IV points, can be a list of centers used to approximate the point or a univariate function that computes the centers (computation starts at 1).
- `radius` – (optional) For type I, II, and III points, the radius of the corresponding disk in $\mathbf{C}_p$. Must coerce into the real numbers. For type IV points, can be a list of radii used to approximate the point or a univariate function that computes the radii (computation starts at 1).
- `power` – (optional) Rational number. Used for constructing type II points; specifies the power of `p` such that p^{power} = radius
- `prec` – (default: 20) The number of disks to be used to approximate a type IV point
- `error_check` – (default: True) If error checking should be run on input. If input is correctly formatted, can be set to `False` for better performance. WARNING: with error check set to `False`, any error in the input will lead to incorrect results.

EXAMPLES:

Type I points can be created by specifying the corresponding point of $P^1(\mathbf{C}_p)$:

```
sage: S = ProjectiveSpace(Qp(5), 1)
sage: P = Berkovich_Cp_Projective(S); P
Projective Berkovich line over Cp(5) of precision 20
```

```
sage: a = S(0, 1)
sage: Q1 = P(a); Q1
Type I point centered at (0 : 1 + O(5^20))
```

```
sage: Q2 = P((1,0)); Q2
Type I point centered at (1 + O(5^20) : 0)
```

Type II and III points can be created by specifying a center and a radius:

```
sage: Q3 = P((0,5), 5**(3/2)); Q3                                              #␣
↪needs sage.symbolic
Type II point centered at (0 : 1 + O(5^20)) of radius 5^3/2
```

```
sage: Q4 = P(0, 3**(3/2)); Q4                                                  #␣
↪needs sage.symbolic
Type III point centered at (0 : 1 + O(5^20)) of radius 5.19615242270663
```

Type IV points can be created from lists of centers and radii:

```
sage: b = S((3,2))  # create centers
sage: c = S((4,3))
sage: d = S((2,3))
sage: L = [b, c, d]
sage: R = [1.761, 1.123, 1.112]
sage: Q5 = P(L, R); Q5
Type IV point of precision 3, approximated by disks centered at
[(4 + 2*5 + 2*5^2 + 2*5^3 + 2*5^4 + 2*5^5 + 2*5^6 + 2*5^7 + 2*5^8 + 2*5^9 + 2*5^10 +
 2*5^11 + 2*5^12 + 2*5^13 + 2*5^14 + 2*5^15 + 2*5^16 + 2*5^17 + 2*5^18 + 2*5^19 +␣
↪O(5^20)) :
 1 + O(5^20)), (3 + 3*5 + 5^2 + 3*5^3 + 5^4 + 3*5^5 + 5^6 + 3*5^7 + 5^8 + 3*5^9 +
 5^10 + 3*5^11 + 5^12 + 3*5^13 + 5^14 + 3*5^15 + 5^16 + 3*5^17 + 5^18 + 3*5^19 +␣
↪O(5^20)) :
 1 + O(5^20))] ... with radii [1.76100000000000, 1.12300000000000] ...
```

Type IV points can also be created from univariate functions. Since the centers of the sequence of disks can not be the point at infinity in $P^1(\mathbf{C}_p)$, only functions into $\mathbf{C}_p$ are supported:

```
sage: L.<t> = PolynomialRing(Qp(5))
sage: T = FractionField(L)
sage: f = T(1/t)
sage: R.<x> = RR[]
sage: Y = FractionField(R)
sage: g = (40*pi)/x                                                            #␣
↪needs sage.symbolic
sage: Q6 = P(f, g); Q6                                                         #␣
↪needs sage.symbolic
Type IV point of precision 20 with centers given by (1 + O(5^20))/((1 + O(5^20))*t)
 and radii given by 40.0000000000000*pi/x
```

as_affine_point()

Return the corresponding affine point after dehomogenizing at infinity.

OUTPUT: A point of affine Berkovich space.

EXAMPLES:

```
sage: B = Berkovich_Cp_Projective(5)
sage: B(5).as_affine_point()
Type I point centered at 5 + O(5^21)
```

```
sage: Q = B(0, 1).as_affine_point(); Q
Type II point centered at 0 of radius 5^0
sage: Q.parent()
Affine Berkovich line over Cp(5) of precision 20
```

```
sage: L.<t> = PolynomialRing(Qp(5))
sage: T = FractionField(L)
sage: f = T(1/t)
sage: R.<x> = RR[]
sage: Y = FractionField(R)
sage: g = (40*pi)/x                                                    #␣
↪needs sage.symbolic
sage: Q2 = B(f, g)                                                     #␣
↪needs sage.symbolic
sage: Q2.as_affine_point()                                             #␣
↪needs sage.symbolic
Type IV point of precision 20 with centers given by (1 + O(5^20))/((1 + O(5^
↪20))*t)
and radii given by 40.0000000000000*pi/x
```

contained_in_interval(*start*, *end*)

Check if this point is an element of the interval [start, end].

INPUT:

- start – A point of the same Berkovich space as this point.
- end – A point of the same Berkovich space as this point.

OUTPUT:

- True if this point is an element of [start, end].
- False otherwise.

EXAMPLES:

```
sage: B = Berkovich_Cp_Projective(3)
sage: Q1 = B(2, 1)
sage: Q2 = B(2, 4)
sage: Q3 = B(1/3)
sage: Q2.contained_in_interval(Q1, Q3.join(Q1))
False
```

```
sage: Q4 = B(1/81, 1)
sage: Q2.contained_in_interval(Q1, Q4.join(Q1))
True
```

gt(*other*)

Return `True` if this point is strictly greater than `other` in the standard partial order.

Roughly, the partial order corresponds to containment of the corresponding disks in $\mathbf{C}_p$.

For example, let x and y be points of type II or III. If x has center c_1 and radius r_1 and y has center c_2 and radius r_2, $x < y$ if and only if $D(c_1, r_1)$ is a subset of $D(c_2, r_2)$ in $\mathbf{C}_p$.

INPUT:

- `other` – A point of the same Berkovich space as this point.

OUTPUT:

- `True` – If this point is greater than `other` in the standard partial order.
- `False` – Otherwise.

EXAMPLES:

```
sage: B = Berkovich_Cp_Projective(QQ, 3)
sage: Q1 = B(5, 3)
sage: Q2 = B(5, 1)
sage: Q1.gt(Q2)
True
```

```
sage: Q3 = B(1/27)
sage: Q1.gt(Q3)
False
```

involution_map()

Return the image of this point under the involution map.

The involution map is the extension of the map `z |-> 1/z` on $P^1(\mathbf{C}_p)$ to Berkovich space.

If zero is contained in every disk approximating a type IV point, then the image under the involution map is not defined. To avoid this error, increase precision.

OUTPUT: A point of the same Berkovich space.

EXAMPLES:

The involution map is 1/z on type I points:

```
sage: B = Berkovich_Cp_Projective(3)
sage: Q1 = B(1/2)
sage: Q1.involution_map()
Type I point centered at (2 + O(3^20) : 1 + O(3^20))
```

```
sage: Q2 = B(0, 1/3)
sage: Q2.involution_map()
Type II point centered at (0 : 1 + O(3^20)) of radius 3^1
```

```
sage: Q3 = B(1/3, 1/3)
sage: Q3.involution_map()
Type II point centered at (3 + O(3^21) : 1 + O(3^20)) of radius 3^-3
```

join(*other*, *basepoint=+Infinity*)

Compute the join of this point and `other`, with respect to `basepoint`.

The join is first point that lies on the intersection of the path from this point to `basepoint` and the path from `other` to `basepoint`.

INPUT:

- `other` – A point of the same Berkovich space as this point.
- `basepoint` – (default: Infinity) A point of the same Berkovich space as this point, or infinity.

OUTPUT: A point of the same Berkovich space.

EXAMPLES:

```
sage: B = Berkovich_Cp_Projective(3)
sage: Q1 = B(2, 1)
sage: Q2 = B(2, 2)
sage: Q1.join(Q2)
Type III point centered at (2 + O(3^20) : 1 + O(3^20)) of radius 2.
↪00000000000000
```

```
sage: Q3 = B(5)
sage: Q3.join(Q1)
Type II point centered at (2 + 3 + O(3^20) : 1 + O(3^20)) of radius 3^0
```

```
sage: Q3.join(Q1, basepoint=Q2)
Type II point centered at (2 + O(3^20) : 1 + O(3^20)) of radius 3^0
```

lt(*other*)

Return `True` if this point is strictly less than `other` in the standard partial order.

Roughly, the partial order corresponds to containment of the corresponding disks in $\mathbf{C}_p$.

For example, let x and y be points of type II or III. If x has center c_1 and radius r_1 and y has center c_2 and radius r_2, $x < y$ if and only if $D(c_1, r_1)$ is a subset of $D(c_2, r_2)$ in $\mathbf{C}_p$.

INPUT:

- `other` – A point of the same Berkovich space as this point.

OUTPUT:

- `True` – If this point is less than `other` in the standard partial order.
- `False` – Otherwise.

EXAMPLES:

```
sage: B = Berkovich_Cp_Projective(3)
sage: Q1 = B(5, 0.5)
sage: Q2 = B(5, 1)
sage: Q1.lt(Q2)
True
```

```
sage: Q3 = B(1)
sage: Q1.lt(Q3)
False
```

18.2 Berkovich Space over $\mathbb{C}_p$

The Berkovich affine line is the set of seminorms on $\mathbf{C}_p[x]$, with the weakest topology that makes the map $|\cdot| \to |f|$ continuous for all $f \in \mathbf{C}_p[x]$. The Berkovich projective line is the one-point compactification of the Berkovich affine line.

The two main classes are *Berkovich_Cp_Affine* and *Berkovich_Cp_Projective*, which implement the affine and projective lines, respectively.

Berkovich_Cp_Affine and *Berkovich_Cp_Projective* take as input one of the following: the prime p, a finite extension of $\mathbf{Q}_p$, or a number field and a place.

For an exposition of Berkovich space over $\mathbf{C}_p$, see Chapter 6 of [Ben2019]. For a more involved exposition, see Chapter 1 and 2 of [BR2010].

AUTHORS:

- Alexander Galarraga (2020-06-22): initial implementation

class sage.schemes.berkovich.berkovich_space.**Berkovich**

Bases: UniqueRepresentation, Parent

The parent class for any Berkovich space

class sage.schemes.berkovich.berkovich_space.**Berkovich_Cp**

Bases: *Berkovich*

Abstract parent class for Berkovich space over Cp.

ideal()

The ideal which defines an embedding of the base_ring into $\mathbf{C}_p$.

If this Berkovich space is backed by a p-adic field, then an embedding is already specified, and this returns None.

OUTPUT:

- An ideal of a base_ring if base_ring is a number field.
- A prime of $\mathbf{Q}$ if base_ring is $\mathbf{Q}$.
- None if base_ring is a p-adic field.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: R.<z> = QQ[]
sage: A.<a> = NumberField(z^2 + 1)
sage: ideal = A.prime_above(5)
sage: B = Berkovich_Cp_Projective(A, ideal)
sage: B.ideal()
Fractional ideal (-a - 2)
```

```
sage: B = Berkovich_Cp_Projective(QQ, 3)
sage: B.ideal()
3
```

```
sage: B = Berkovich_Cp_Projective(Qp(3))
sage: B.ideal() is None
True
```

is_number_field_base()

Return True if this Berkovich space is backed by a number field.

OUTPUT:

- True if this Berkovich space was created with a number field.
- False otherwise.

EXAMPLES:

```
sage: B = Berkovich_Cp_Affine(Qp(3))
sage: B.is_number_field_base()
False
```

```
sage: B = Berkovich_Cp_Affine(QQ, 3)
sage: B.is_number_field_base()
True
```

is_padic_base()

Return True if this Berkovich space is backed by a p-adic field.

OUTPUT:

- True if this Berkovich space was created with a p-adic field.
- False otherwise.

EXAMPLES:

```
sage: B = Berkovich_Cp_Affine(Qp(3))
sage: B.is_padic_base()
True
```

```
sage: B = Berkovich_Cp_Affine(QQ, 3)
sage: B.is_padic_base()
False
```

prime()

The residue characteristic of the base.

EXAMPLES:

```
sage: B = Berkovich_Cp_Projective(3)
sage: B.prime()
3
```

```
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: A.<a> = NumberField(x^3 + 20)
sage: ideal = A.ideal(-1/2*a^2 + a - 3)
sage: B = Berkovich_Cp_Affine(A, ideal)
sage: B.residue_characteristic()
7
```

residue_characteristic()

The residue characteristic of the base.

EXAMPLES:

```
sage: B = Berkovich_Cp_Projective(3)
sage: B.prime()
3
```

```
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: A.<a> = NumberField(x^3 + 20)
sage: ideal = A.ideal(-1/2*a^2 + a - 3)
sage: B = Berkovich_Cp_Affine(A, ideal)
sage: B.residue_characteristic()
7
```

class sage.schemes.berkovich.berkovich_space.**Berkovich_Cp_Affine**(*base*, *ideal=None*)

Bases: *Berkovich_Cp*

The Berkovich affine line over $\mathbf{C}_p$.

The Berkovich affine line is the set of seminorms on $\mathbf{C}_p[x]$, with the weakest topology such that the map $|\cdot| \to |f|$ is continuous for all $f \in \mathbf{C}_p[x]$.

We can represent the Berkovich affine line in two separate ways: either using a p-adic field to represent elements or using a number field to represent elements while storing an ideal of the ring of integers of the number field, which specifies an embedding of the number field into $\mathbf{C}_p$. See the examples.

INPUT:

- base – Three cases:
  - a prime number p. Centers of elements are then represented as points of $\mathbf{Q}_p$.
  - $\mathbf{Q}_p$ or a finite extension of $\mathbf{Q}_p$. Centers of elements are then represented as points of base.
  - A number field K. Centers of elements are then represented as points of K.
- ideal – (optional) a prime ideal of base. Must be specified if a number field is passed to base, otherwise it is ignored.

EXAMPLES:

```
sage: B = Berkovich_Cp_Affine(3); B
Affine Berkovich line over Cp(3) of precision 20
```

We can create elements:

```
sage: B(-2)
Type I point centered at 1 + 2*3 + 2*3^2 + 2*3^3 + 2*3^4 + 2*3^5
+ 2*3^6 + 2*3^7 + 2*3^8 + 2*3^9 + 2*3^10 + 2*3^11 + 2*3^12 + 2*3^13
+ 2*3^14 + 2*3^15 + 2*3^16 + 2*3^17 + 2*3^18 + 2*3^19 + O(3^20)
```

```
sage: B(1, 2)
Type III point centered at 1 + O(3^20) of radius 2.00000000000000
```

For details on element creation, see the documentation of *Berkovich_Element_Cp_Affine*. Initializing by passing in $\mathbf{Q}_p$ looks the same:

```
sage: B = Berkovich_Cp_Affine(Qp(3)); B
Affine Berkovich line over Cp(3) of precision 20
```

However, this method allows for more control over behind-the-scenes conversion:

```
sage: B = Berkovich_Cp_Affine(Qp(3, 1)); B
Affine Berkovich line over Cp(3) of precision 1

sage: B(1/2)
Type I point centered at 2 + O(3)
```

Note that this point has very low precision, as `B` was initialized with a p-adic field of capped-relative precision one. For high precision, pass in a high precision p-adic field:

```
sage: B = Berkovich_Cp_Affine(Qp(3, 1000)); B
Affine Berkovich line over Cp(3) of precision 1000
```

Points of Berkovich space can be created from points of extensions of $\mathbf{Q}_p$:

```
sage: B = Berkovich_Cp_Affine(3)
sage: A.<a> = Qp(3).extension(x^3 - 3)
sage: B(a)
Type I point centered at a + O(a^61)
```

For exact computation, a number field can be used:

```
sage: R.<x> = QQ[]
sage: A.<a> = NumberField(x^3 + 20)                                    #␣
↪needs sage.rings.number_field
sage: ideal = A.prime_above(3)                                         #␣
↪needs sage.rings.number_field
sage: B = Berkovich_Cp_Affine(A, ideal); B                             #␣
↪needs sage.rings.number_field
Affine Berkovich line over Cp(3), with base
 Number Field in a with defining polynomial x^3 + 20
```

Number fields have a major advantage of exact computation.

Number fields also have added functionality. Arbitrary extensions of $\mathbf{Q}$ are supported, while there is currently limited functionality for extensions of $\mathbf{Q}_p$. As seen above, constructing a Berkovich space backed by a number field requires specifying an ideal of the ring of integers of the number field. Specifying the ideal uniquely specifies an embedding of the number field into $\mathbf{C}_p$.

Unlike in the case where Berkovich space is backed by a p-adic field, any point of a Berkovich space backed by a number field must be centered at a point of that number field:

```
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: A.<a> = NumberField(x^3 + 20)
sage: ideal = A.prime_above(3)
sage: B = Berkovich_Cp_Affine(A, ideal)
sage: C.<c> = NumberField(x^2 + 1)
sage: B(c)
Traceback (most recent call last):
...
ValueError: could not convert c to Number Field in a
with defining polynomial x^3 + 20
```

Element

alias of *Berkovich_Element_Cp_Affine*

class sage.schemes.berkovich.berkovich_space.**Berkovich_Cp_Projective**(*base*, *ideal=None*)

Bases: *Berkovich_Cp*

The Berkovich projective line over $\mathbf{C}_p$.

The Berkovich projective line is the one-point compactification of the Berkovich affine line.

We can represent the Berkovich projective line in two separate ways: either using a p-adic field to represent elements or using a number field to represent elements while storing an ideal of the ring of integers of the number field, which specifies an embedding of the number field into $\mathbf{C}_p$. See the examples.

INPUT:

- `base` – Three cases:
  - a prime number p. Centers of elements are then represented as points of projective space of dimension 1 over $\mathbf{Q}_p$.
  - $\mathbf{Q}_p$ or a finite extension of $\mathbf{Q}_p$. Centers of elements are then represented as points of projective space of dimension 1 over `base`.
  - A number field K. Centers of elements are then represented as points of projective space of dimension 1 over `base`.
- `ideal` – (optional) a prime ideal of `base`. Must be specified if a number field is passed to `base`, otherwise it is ignored.

EXAMPLES:

```
sage: B = Berkovich_Cp_Projective(3); B
Projective Berkovich line over Cp(3) of precision 20
```

Elements can be constructed:

```
sage: B(1/2)
Type I point centered at (2 + 3 + 3^2 + 3^3 + 3^4 + 3^5
+ 3^6 + 3^7 + 3^8 + 3^9 + 3^10 + 3^11 + 3^12 + 3^13 + 3^14
+ 3^15 + 3^16 + 3^17 + 3^18 + 3^19 + O(3^20) : 1 + O(3^20))
```

```
sage: B(2, 1)
Type II point centered at (2 + O(3^20) : 1 + O(3^20)) of radius 3^0
```

For details about element construction, see the documentation of *Berkovich_Element_Cp_Projective*. Initializing a Berkovich projective line by passing in a p-adic space looks the same:

```
sage: B = Berkovich_Cp_Projective(Qp(3)); B
Projective Berkovich line over Cp(3) of precision 20
```

However, this method allows for more control over behind-the-scenes conversion:

```
sage: S = Qp(3, 1)
sage: B = Berkovich_Cp_Projective(S); B
Projective Berkovich line over Cp(3) of precision 1

sage: Q1 = B(1/2); Q1
Type I point centered at (2 + O(3) : 1 + O(3))
```

Note that this point has very low precision, as S has low precision cap. Berkovich space can also be created over a number field, as long as an ideal is specified:

```
sage: R.<x> = QQ[]
sage: A.<a> = NumberField(x^2 + 1)                                    #␣
↪needs sage.rings.number_field
sage: ideal = A.prime_above(2)                                        #␣
↪needs sage.rings.number_field
sage: B = Berkovich_Cp_Projective(A, ideal); B                        #␣
↪needs sage.rings.number_field
Projective Berkovich line over Cp(2), with base
 Number Field in a with defining polynomial x^2 + 1
```

Number fields have the benefit that computation is exact, but lack support for all of $\mathbf{C}_p$.

Number fields also have the advantage of added functionality, as arbitrary extensions of $\mathbf{Q}$ can be constructed while there is currently limited functionality for extensions of $\mathbf{Q}_p$. As seen above, constructing a Berkovich space backed by a number field requires specifying an ideal of the ring of integers of the number field. Specifying the ideal uniquely specifies an embedding of the number field into $\mathbf{C}_p$.

Unlike in the case where Berkovich space is backed by a p-adic field, any point of a Berkovich space backed by a number field must be centered at a point of that number field:

```
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: A.<a> = NumberField(x^3 + 20)
sage: ideal = A.prime_above(3)
sage: B = Berkovich_Cp_Projective(A, ideal)
sage: C.<c> = NumberField(x^2 + 1)
sage: B(c)
Traceback (most recent call last):
...
TypeError: could not convert c to Projective Space
of dimension 1 over Number Field in a with defining polynomial x^3 + 20
```

Element

alias of *Berkovich_Element_Cp_Projective*

base_ring()

The base ring of this Berkovich Space.

OUTPUT: A field.

EXAMPLES:

```
sage: B = Berkovich_Cp_Projective(3)
sage: B.base_ring()
3-adic Field with capped relative precision 20
```

```
sage: C = Berkovich_Cp_Projective(ProjectiveSpace(Qp(3, 1), 1))
sage: C.base_ring()
3-adic Field with capped relative precision 1
```

```
sage: # needs sage.rings.number_field
sage: R.<x> = QQ[]
sage: A.<a> = NumberField(x^3 + 20)
sage: ideal = A.prime_above(3)
sage: D = Berkovich_Cp_Projective(A, ideal)
sage: D.base_ring()
Number Field in a with defining polynomial x^3 + 20
```

sage.schemes.berkovich.berkovich_space.**is_Berkovich**(*space*)

Check if space is a Berkovich space.

OUTPUT:

- True if space is a Berkovich space.
- False otherwise.

EXAMPLES:

```
sage: B = Berkovich_Cp_Projective(3)
sage: from sage.schemes.berkovich.berkovich_space import is_Berkovich
sage: is_Berkovich(B)
True
```

sage.schemes.berkovich.berkovich_space.**is_Berkovich_Cp**(*space*)

Check if space is a Berkovich space over Cp.

OUTPUT:

- True if space is a Berkovich space over Cp.
- False otherwise.

EXAMPLES:

```
sage: B = Berkovich_Cp_Projective(3)
sage: from sage.schemes.berkovich.berkovich_space import is_Berkovich
sage: is_Berkovich(B)
True
```

CHAPTER

NINETEEN

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

S

INDEX

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X

Y

Z