Combinatorial and Discrete Geometry

Release 10.6

The Sage Development Team

Apr 01, 2025

CONTENTS

Sage includes classes for hyperplane arrangements, polyhedra, toric varieties (including polyhedral cones and fans), triangulations and some other helper classes and functions.

CHAPTER

ONE

HYPERPLANE ARRANGEMENTS

1.1 Hyperplane Arrangements

Before talking about hyperplane arrangements, let us start with individual hyperplanes. This package uses certain linear expressions to represent hyperplanes, that is, a linear expression $3x + 3y - 5z - 7$ stands for the hyperplane with the equation $3x + 3y - 5z = 7$. To create it in Sage, you first have to create a `HyperplaneArrangements` object to define the variables x, y, z:

```
sage: H.<x,y,z> = HyperplaneArrangements(QQ)
sage: h = 3*x + 2*y - 5*z - 7;  h
Hyperplane 3*x + 2*y - 5*z - 7
sage: h.normal()
(3, 2, -5)
sage: h.constant_term()
-7
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,) = H._first_
↪ngens(3)
>>> h = Integer(3)*x + Integer(2)*y - Integer(5)*z - Integer(7);  h
Hyperplane 3*x + 2*y - 5*z - 7
>>> h.normal()
(3, 2, -5)
>>> h.constant_term()
-7
```

The individual hyperplanes behave like the linear expression with regard to addition and scalar multiplication, which is why you can do linear combinations of the coordinates:

```
sage: -2*h
Hyperplane -6*x - 4*y + 10*z + 14
sage: x, y, z
(Hyperplane x + 0*y + 0*z + 0,
 Hyperplane 0*x + y + 0*z + 0,
 Hyperplane 0*x + 0*y + z + 0)
```

```
>>> from sage.all import *
>>> -Integer(2)*h
Hyperplane -6*x - 4*y + 10*z + 14
>>> x, y, z
(Hyperplane x + 0*y + 0*z + 0,
```

(continues on next page)

(continued from previous page)

```
Hyperplane 0*x + y + 0*z + 0,
Hyperplane 0*x + 0*y + z + 0)
```

See *sage.geometry.hyperplane_arrangement.hyperplane* for more functionality of the individual hyperplanes.

1.1.1 Arrangements

There are several ways to create hyperplane arrangements:

Notation (i): by passing individual hyperplanes to the *HyperplaneArrangements* object:

```
sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: box = x | y | x-1 | y-1;  box
Arrangement <y - 1 | y | x - 1 | x>
sage: box == H(x, y, x-1, y-1)     # alternative syntax
True
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = H._first_ngens(2)
>>> box = x | y | x-Integer(1) | y-Integer(1);  box
Arrangement <y - 1 | y | x - 1 | x>
>>> box == H(x, y, x-Integer(1), y-Integer(1))     # alternative syntax
True
```

Notation (ii): by passing anything that defines a hyperplane, for example a coefficient vector and constant term:

```
sage: H = HyperplaneArrangements(QQ, ('x', 'y'))
sage: triangle = H([(1, 0), 0], [(0, 1), 0], [(1,1), -1]);  triangle
Arrangement <y | x | x + y - 1>

sage: H.inject_variables()
Defining x, y
sage: triangle == x | y | x+y-1
True
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, ('x', 'y'))
>>> triangle = H([(Integer(1), Integer(0)), Integer(0)], [(Integer(0), Integer(1)),
↪Integer(0)], [(Integer(1),Integer(1)), -Integer(1)]);  triangle
Arrangement <y | x | x + y - 1>

>>> H.inject_variables()
Defining x, y
>>> triangle == x | y | x+y-Integer(1)
True
```

The default base field is $\mathbf{Q}$, the rational numbers. Finite fields are also supported:

```
sage: H.<x,y,z> = HyperplaneArrangements(GF(5))
sage: a = H([(1,2,3), 4], [(5,6,7), 8]);  a
Arrangement <y + 2*z + 3 | x + 2*y + 3*z + 4>
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(GF(Integer(5)), names=('x', 'y', 'z',)); (x, y, z,) =␣
↪H._first_ngens(3)
>>> a = H([(Integer(1),Integer(2),Integer(3)), Integer(4)], [(Integer(5),Integer(6),
↪Integer(7)), Integer(8)]);  a
Arrangement <y + 2*z + 3 | x + 2*y + 3*z + 4>
```

Number fields are also possible:

```
sage: # needs sage.rings.number_field
sage: x = polygen(QQ, 'x')
sage: NF.<a> = NumberField(x**4 - 5*x**2 + 5, embedding=1.90)
sage: H.<y,z> = HyperplaneArrangements(NF)
sage: A = H([[(-a**3 + 3*a, -a**2 + 4), 1], [(a**3 - 4*a, -1), 1],
....:        [(0, 2*a**2 - 6), 1], [(-a**3 + 4*a, -1), 1],
....:        [(a**3 - 3*a, -a**2 + 4), 1]])
sage: A
Arrangement of 5 hyperplanes of dimension 2 and rank 2
sage: A.base_ring()
Number Field in a with defining polynomial x^4 - 5*x^2 + 5
 with a = 1.902113032590308?
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> x = polygen(QQ, 'x')
>>> NF = NumberField(x**Integer(4) - Integer(5)*x**Integer(2) + Integer(5),␣
↪embedding=RealNumber('1.90'), names=('a',)); (a,) = NF._first_ngens(1)
>>> H = HyperplaneArrangements(NF, names=('y', 'z',)); (y, z,) = H._first_ngens(2)
>>> A = H([[(-a**Integer(3) + Integer(3)*a, -a**Integer(2) + Integer(4)), Integer(1)],
↪ [(a**Integer(3) - Integer(4)*a, -Integer(1)), Integer(1)],
...        [(Integer(0), Integer(2)*a**Integer(2) - Integer(6)), Integer(1)], [(-
↪a**Integer(3) + Integer(4)*a, -Integer(1)), Integer(1)],
...        [(a**Integer(3) - Integer(3)*a, -a**Integer(2) + Integer(4)), Integer(1)]])
>>> A
Arrangement of 5 hyperplanes of dimension 2 and rank 2
>>> A.base_ring()
Number Field in a with defining polynomial x^4 - 5*x^2 + 5
 with a = 1.902113032590308?
```

Notation (iii): a list or tuple of hyperplanes:

```
sage: H.<x,y,z> = HyperplaneArrangements(GF(5))
sage: k = [x+i for i in range(4)];  k
[Hyperplane x + 0*y + 0*z + 0, Hyperplane x + 0*y + 0*z + 1,
 Hyperplane x + 0*y + 0*z + 2, Hyperplane x + 0*y + 0*z + 3]
sage: H(k)
Arrangement <x | x + 1 | x + 2 | x + 3>
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(GF(Integer(5)), names=('x', 'y', 'z',)); (x, y, z,) =␣
↪H._first_ngens(3)
>>> k = [x+i for i in range(Integer(4))];  k
[Hyperplane x + 0*y + 0*z + 0, Hyperplane x + 0*y + 0*z + 1,
```

(continues on next page)

(continued from previous page)

```
 Hyperplane x + 0*y + 0*z + 2, Hyperplane x + 0*y + 0*z + 3]
>>> H(k)
Arrangement <x | x + 1 | x + 2 | x + 3>
```

Notation (iv): using the library of arrangements:

```
sage: hyperplane_arrangements.braid(4)                                      #␣
↪needs sage.graphs
Arrangement of 6 hyperplanes of dimension 4 and rank 3
sage: hyperplane_arrangements.semiorder(3)
Arrangement of 6 hyperplanes of dimension 3 and rank 2
sage: hyperplane_arrangements.graphical(graphs.PetersenGraph())             #␣
↪needs sage.graphs
Arrangement of 15 hyperplanes of dimension 10 and rank 9
sage: hyperplane_arrangements.Ish(5)
Arrangement of 20 hyperplanes of dimension 5 and rank 4
```

```
>>> from sage.all import *
>>> hyperplane_arrangements.braid(Integer(4))                                ␣
↪     # needs sage.graphs
Arrangement of 6 hyperplanes of dimension 4 and rank 3
>>> hyperplane_arrangements.semiorder(Integer(3))
Arrangement of 6 hyperplanes of dimension 3 and rank 2
>>> hyperplane_arrangements.graphical(graphs.PetersenGraph())              #␣
↪needs sage.graphs
Arrangement of 15 hyperplanes of dimension 10 and rank 9
>>> hyperplane_arrangements.Ish(Integer(5))
Arrangement of 20 hyperplanes of dimension 5 and rank 4
```

Notation (v): from the bounding hyperplanes of a polyhedron:

```
sage: a = polytopes.cube().hyperplane_arrangement();  a
Arrangement of 6 hyperplanes of dimension 3 and rank 3
sage: a.n_regions()
27
```

```
>>> from sage.all import *
>>> a = polytopes.cube().hyperplane_arrangement();  a
Arrangement of 6 hyperplanes of dimension 3 and rank 3
>>> a.n_regions()
27
```

New arrangements from old:

```
sage: # needs sage.graphs
sage: a = hyperplane_arrangements.braid(3)
sage: b = a.add_hyperplane([4, 1, 2, 3])
sage: b
Arrangement <t1 - t2 | t0 - t1 | t0 - t2 | t0 + 2*t1 + 3*t2 + 4>
sage: c = b.deletion([4, 1, 2, 3])
sage: a == c
True
```

(continues on next page)

(continued from previous page)

```
sage: # needs sage.combinat sage.graphs
sage: a = hyperplane_arrangements.braid(3)
sage: b = a.union(hyperplane_arrangements.semiorder(3))
sage: b == a | hyperplane_arrangements.semiorder(3)     # alternate syntax
True
sage: b == hyperplane_arrangements.Catalan(3)
True
sage: a
Arrangement <t1 - t2 | t0 - t1 | t0 - t2>

sage: a = hyperplane_arrangements.coordinate(4)
sage: h = a.hyperplanes()[0]
sage: b = a.restriction(h)
sage: b == hyperplane_arrangements.coordinate(3)
True
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> a = hyperplane_arrangements.braid(Integer(3))
>>> b = a.add_hyperplane([Integer(4), Integer(1), Integer(2), Integer(3)])
>>> b
Arrangement <t1 - t2 | t0 - t1 | t0 - t2 | t0 + 2*t1 + 3*t2 + 4>
>>> c = b.deletion([Integer(4), Integer(1), Integer(2), Integer(3)])
>>> a == c
True

>>> # needs sage.combinat sage.graphs
>>> a = hyperplane_arrangements.braid(Integer(3))
>>> b = a.union(hyperplane_arrangements.semiorder(Integer(3)))
>>> b == a | hyperplane_arrangements.semiorder(Integer(3))     # alternate syntax
True
>>> b == hyperplane_arrangements.Catalan(Integer(3))
True
>>> a
Arrangement <t1 - t2 | t0 - t1 | t0 - t2>

>>> a = hyperplane_arrangements.coordinate(Integer(4))
>>> h = a.hyperplanes()[Integer(0)]
>>> b = a.restriction(h)
>>> b == hyperplane_arrangements.coordinate(Integer(3))
True
```

1.1.2 Properties of Arrangements

A hyperplane arrangement is *essential* if the normals to its hyperplanes span the ambient space. Otherwise, it is *inessential*. The essentialization is formed by intersecting the hyperplanes by this normal space (actually, it is a bit more complicated over finite fields):

```
sage: # needs sage.graphs
sage: a = hyperplane_arrangements.braid(4);  a
```

(continues on next page)

(continued from previous page)

```
Arrangement of 6 hyperplanes of dimension 4 and rank 3
sage: a.is_essential()
False
sage: a.rank() < a.dimension()  # double-check
True
sage: a.essentialization()
Arrangement of 6 hyperplanes of dimension 3 and rank 3
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> a = hyperplane_arrangements.braid(Integer(4));  a
Arrangement of 6 hyperplanes of dimension 4 and rank 3
>>> a.is_essential()
False
>>> a.rank() < a.dimension()  # double-check
True
>>> a.essentialization()
Arrangement of 6 hyperplanes of dimension 3 and rank 3
```

The connected components of the complement of the hyperplanes of an arrangement in $\mathbf{R}^n$ are called the *regions* of the arrangement:

```
sage: a = hyperplane_arrangements.semiorder(3)
sage: b = a.essentialization();   b
Arrangement of 6 hyperplanes of dimension 2 and rank 2
sage: b.n_regions()
19
sage: b.regions()
(A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 6 vertices,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices and 1␣
↪ray,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices and 1␣
↪ray,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices and 1␣
↪ray,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices and 1␣
↪ray,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices and 1␣
↪ray,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices and 1␣
↪ray,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2 rays,
```

(continues on next page)

(continued from previous page)

```
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2 rays)
sage: b.bounded_regions()
(A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 6 vertices,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices)
sage: b.n_bounded_regions()
7
sage: a.unbounded_regions()
(A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 1 vertex, 2 rays, 1␣
↪line,
 A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 3 vertices, 1 ray,␣
↪1 line,
 A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 1 vertex, 2 rays, 1␣
↪line,
 A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 3 vertices, 1 ray,␣
↪1 line,
 A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 1 vertex, 2 rays, 1␣
↪line,
 A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 3 vertices, 1 ray,␣
↪1 line,
 A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 3 vertices, 1 ray,␣
↪1 line,
 A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 1 vertex, 2 rays, 1␣
↪line,
 A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 3 vertices, 1 ray,␣
↪1 line,
 A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 1 vertex, 2 rays, 1␣
↪line,
 A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 3 vertices, 1 ray,␣
↪1 line,
 A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 1 vertex, 2 rays, 1␣
↪line)
```

```
>>> from sage.all import *
>>> a = hyperplane_arrangements.semiorder(Integer(3))
>>> b = a.essentialization();   b
Arrangement of 6 hyperplanes of dimension 2 and rank 2
>>> b.n_regions()
19
>>> b.regions()
(A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 6 vertices,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices and 1␣
↪ray,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices and 1␣
↪ray,
```

(continues on next page)

(continued from previous page)

```
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices and 1␣
↪ray,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices and 1␣
↪ray,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices and 1␣
↪ray,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices and 1␣
↪ray,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2 rays)
>>> b.bounded_regions()
(A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 6 vertices,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
 A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices)
>>> b.n_bounded_regions()
7
>>> a.unbounded_regions()
(A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 1 vertex, 2 rays, 1␣
↪line,
 A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 3 vertices, 1 ray,␣
↪1 line,
 A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 1 vertex, 2 rays, 1␣
↪line,
 A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 3 vertices, 1 ray,␣
↪1 line,
 A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 1 vertex, 2 rays, 1␣
↪line,
 A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 3 vertices, 1 ray,␣
↪1 line,
 A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 3 vertices, 1 ray,␣
↪1 line,
 A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 1 vertex, 2 rays, 1␣
↪line,
 A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 3 vertices, 1 ray,␣
↪1 line,
 A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 1 vertex, 2 rays, 1␣
↪line,
 A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 3 vertices, 1 ray,␣
↪1 line,
 A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 1 vertex, 2 rays, 1␣
↪line)
```

The distance between regions is defined as the number of hyperplanes separating them. For example:

```
sage: # needs sage.combinat
sage: r1 = b.regions()[0]
sage: r2 = b.regions()[1]
sage: b.distance_between_regions(r1, r2)
1
sage: [hyp for hyp in b if b.is_separating_hyperplane(r1, r2, hyp)]
[Hyperplane 2*t1 + t2 + 1]
sage: b.distance_enumerator(r1)  # generating function for distances from r1
6*x^3 + 6*x^2 + 6*x + 1
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> r1 = b.regions()[Integer(0)]
>>> r2 = b.regions()[Integer(1)]
>>> b.distance_between_regions(r1, r2)
1
>>> [hyp for hyp in b if b.is_separating_hyperplane(r1, r2, hyp)]
[Hyperplane 2*t1 + t2 + 1]
>>> b.distance_enumerator(r1)  # generating function for distances from r1
6*x^3 + 6*x^2 + 6*x + 1
```

Note

bounded region really mean *relatively bounded* here. A region is relatively bounded if its intersection with space spanned by the normals of the hyperplanes in the arrangement is bounded.

The intersection poset of a hyperplane arrangement is the collection of all nonempty intersections of hyperplanes in the arrangement, ordered by reverse inclusion. It includes the ambient space of the arrangement (as the intersection over the empty set):

```
sage: # needs sage.graphs
sage: a = hyperplane_arrangements.braid(3)
sage: p = a.intersection_poset()
sage: p.is_ranked()
True
sage: p.order_polytope()
A 5-dimensional polyhedron in ZZ^5 defined as the convex hull of 10 vertices
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> a = hyperplane_arrangements.braid(Integer(3))
>>> p = a.intersection_poset()
>>> p.is_ranked()
True
>>> p.order_polytope()
A 5-dimensional polyhedron in ZZ^5 defined as the convex hull of 10 vertices
```

The characteristic polynomial is a basic invariant of a hyperplane arrangement. It is defined as

$$\chi(x) := \sum_{w \in P} \mu(w) x^{dim(w)}$$

where P is the `intersection_poset()` of the arrangement and μ is the Möbius function of P:

```
sage: # long time
sage: a = hyperplane_arrangements.semiorder(5)
sage: a.characteristic_polynomial()                # about a second on Core i7
x^5 - 20*x^4 + 180*x^3 - 790*x^2 + 1380*x
sage: a.poincare_polynomial()
1380*x^4 + 790*x^3 + 180*x^2 + 20*x + 1
sage: a.n_regions()
2371
sage: charpoly = a.characteristic_polynomial()
sage: charpoly(-1)
-2371
sage: a.n_bounded_regions()
751
sage: charpoly(1)
751
```

```
>>> from sage.all import *
>>> # long time
>>> a = hyperplane_arrangements.semiorder(Integer(5))
>>> a.characteristic_polynomial()                # about a second on Core i7
x^5 - 20*x^4 + 180*x^3 - 790*x^2 + 1380*x
>>> a.poincare_polynomial()
1380*x^4 + 790*x^3 + 180*x^2 + 20*x + 1
>>> a.n_regions()
2371
>>> charpoly = a.characteristic_polynomial()
>>> charpoly(-Integer(1))
-2371
>>> a.n_bounded_regions()
751
>>> charpoly(Integer(1))
751
```

For finer invariants derived from the intersection poset, see `whitney_number()` and `doubly_indexed_whitney_number()`.

Miscellaneous methods (see documentation for an explanation):

```
sage: a = hyperplane_arrangements.semiorder(3)
sage: a.has_good_reduction(5)                                              #␣
↪needs sage.rings.finite_rings
True
sage: b = a.change_ring(GF(5))
sage: pa = a.intersection_poset()                                          #␣
↪needs sage.graphs
sage: pb = b.intersection_poset()                                          #␣
↪needs sage.rings.finite_rings
sage: pa.is_isomorphic(pb)                                                 #␣
↪needs sage.graphs sage.rings.finite_rings
True
sage: a.face_vector()                                                      #␣
↪needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
(0, 12, 30, 19)
sage: a.face_vector()                                                          #␣
↪needs sage.graphs
(0, 12, 30, 19)
sage: a.is_central()
False
sage: a.is_linear()
False
sage: a.sign_vector((1,1,1))
(-1, 1, -1, 1, -1, 1)
sage: a.varchenko_matrix()[:6, :6]
[          1          h2       h2*h4       h2*h3    h2*h3*h4 h2*h3*h4*h5]
[         h2           1          h4          h3       h3*h4    h3*h4*h5]
[      h2*h4          h4           1       h3*h4          h3       h3*h5]
[      h2*h3          h3       h3*h4           1          h4       h4*h5]
[   h2*h3*h4       h3*h4          h3          h4           1          h5]
[h2*h3*h4*h5    h3*h4*h5       h3*h5       h4*h5          h5           1]
```

```
>>> from sage.all import *
>>> a = hyperplane_arrangements.semiorder(Integer(3))
>>> a.has_good_reduction(Integer(5))                                          ␣
↪     # needs sage.rings.finite_rings
True
>>> b = a.change_ring(GF(Integer(5)))
>>> pa = a.intersection_poset()                                               #␣
↪needs sage.graphs
>>> pb = b.intersection_poset()                                               #␣
↪needs sage.rings.finite_rings
>>> pa.is_isomorphic(pb)                                                      #␣
↪needs sage.graphs sage.rings.finite_rings
True
>>> a.face_vector()                                                           #␣
↪needs sage.graphs
(0, 12, 30, 19)
>>> a.face_vector()                                                           #␣
↪needs sage.graphs
(0, 12, 30, 19)
>>> a.is_central()
False
>>> a.is_linear()
False
>>> a.sign_vector((Integer(1),Integer(1),Integer(1)))
(-1, 1, -1, 1, -1, 1)
>>> a.varchenko_matrix()[:Integer(6), :Integer(6)]
[          1          h2       h2*h4       h2*h3    h2*h3*h4 h2*h3*h4*h5]
[         h2           1          h4          h3       h3*h4    h3*h4*h5]
[      h2*h4          h4           1       h3*h4          h3       h3*h5]
[      h2*h3          h3       h3*h4           1          h4       h4*h5]
[   h2*h3*h4       h3*h4          h3          h4           1          h5]
[h2*h3*h4*h5    h3*h4*h5       h3*h5       h4*h5          h5           1]
```

There are extensive methods for visualizing hyperplane arrangements in low dimensions. See `plot()` for details.

AUTHORS:

- David Perkinson (2013-06): initial version
- Qiaoyu Yang (2013-07)
- Kuai Yu (2013-07)
- Volker Braun (2013-10): Better Sage integration, major code refactoring.

This module implements hyperplane arrangements defined over the rationals or over finite fields. The original motivation was to make a companion to Richard Stanley's notes [Sta2007] on hyperplane arrangements.

class sage.geometry.hyperplane_arrangement.arrangement.**HyperplaneArrangementElement**(*parent*, *hyperplanes*, *check=True*, *backend=None*)

Bases: `Element`

A hyperplane arrangement.

> **Warning**
>
> You should never create *HyperplaneArrangementElement* instances directly, always use the parent.

add_hyperplane(*other*)

The union of `self` with `other`.

INPUT:

- `other` – a hyperplane arrangement or something that can be converted into a hyperplane arrangement

OUTPUT: a new hyperplane arrangement

EXAMPLES:

```
sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: A = H([1,2,3], [0,1,1], [0,1,-1], [1,-1,0], [1,1,0])
sage: B = H([1,1,1], [1,-1,1], [1,0,-1])
sage: C = A.union(B); C
Arrangement of 8 hyperplanes of dimension 2 and rank 2
sage: C == A | B   # syntactic sugar
True
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = H._first_
↪ngens(2)
>>> A = H([Integer(1),Integer(2),Integer(3)], [Integer(0),Integer(1),
↪Integer(1)], [Integer(0),Integer(1),-Integer(1)], [Integer(1),-Integer(1),
↪Integer(0)], [Integer(1),Integer(1),Integer(0)])
>>> B = H([Integer(1),Integer(1),Integer(1)], [Integer(1),-Integer(1),
↪Integer(1)], [Integer(1),Integer(0),-Integer(1)])
>>> C = A.union(B); C
```

(continues on next page)

(continued from previous page)

```
Arrangement of 8 hyperplanes of dimension 2 and rank 2
>>> C == A | B    # syntactic sugar
True
```

A single hyperplane is coerced into a hyperplane arrangement if necessary:

```
sage: A.union(x+y-1)
Arrangement of 6 hyperplanes of dimension 2 and rank 2
sage: A.add_hyperplane(x+y-1)     # alias
Arrangement of 6 hyperplanes of dimension 2 and rank 2

sage: P.<x,y> = HyperplaneArrangements(RR)
sage: C = P(2*x + 4*y + 5)
sage: C.union(A)
Arrangement of 6 hyperplanes of dimension 2 and rank 2
```

```
>>> from sage.all import *
>>> A.union(x+y-Integer(1))
Arrangement of 6 hyperplanes of dimension 2 and rank 2
>>> A.add_hyperplane(x+y-Integer(1))     # alias
Arrangement of 6 hyperplanes of dimension 2 and rank 2

>>> P = HyperplaneArrangements(RR, names=('x', 'y',)); (x, y,) = P._first_
↪ngens(2)
>>> C = P(Integer(2)*x + Integer(4)*y + Integer(5))
>>> C.union(A)
Arrangement of 6 hyperplanes of dimension 2 and rank 2
```

backend()

Return the backend used for polyhedral objects.

OUTPUT: string giving the backend or `None` if none is specified

EXAMPLES:

By default, no backend is specified:

```
sage: H = HyperplaneArrangements(QQ)
sage: A = H()
sage: A.backend()
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ)
>>> A = H()
>>> A.backend()
```

Otherwise, one may specify a polyhedral backend:

```
sage: A = H(backend='ppl')
sage: A.backend()
'ppl'
sage: A = H(backend='normaliz')
sage: A.backend()
'normaliz'
```

```
>>> from sage.all import *
>>> A = H(backend='ppl')
>>> A.backend()
'ppl'
>>> A = H(backend='normaliz')
>>> A.backend()
'normaliz'
```

bounded_regions()

Return the relatively bounded regions of the arrangement.

A region is relatively bounded if its intersection with the space spanned by the normals to the hyperplanes is bounded. This is the same as being bounded in the case that the hyperplane arrangement is essential. It is assumed that the arrangement is defined over the rationals.

OUTPUT:

Tuple of polyhedra. The relatively bounded regions of the arrangement.

See also

unbounded_regions()

EXAMPLES:

```
sage: # needs sage.combinat
sage: A = hyperplane_arrangements.semiorder(3)
sage: A.bounded_regions()
(A 3-dimensional polyhedron in QQ^3 defined
     as the convex hull of 3 vertices and 1 line,
 A 3-dimensional polyhedron in QQ^3 defined
     as the convex hull of 3 vertices and 1 line,
 A 3-dimensional polyhedron in QQ^3 defined
     as the convex hull of 3 vertices and 1 line,
 A 3-dimensional polyhedron in QQ^3 defined
     as the convex hull of 6 vertices and 1 line,
 A 3-dimensional polyhedron in QQ^3 defined
     as the convex hull of 3 vertices and 1 line,
 A 3-dimensional polyhedron in QQ^3 defined
     as the convex hull of 3 vertices and 1 line,
 A 3-dimensional polyhedron in QQ^3 defined
     as the convex hull of 3 vertices and 1 line)
sage: A.bounded_regions()[0].is_compact()     # the regions are only
↪*relatively* bounded
False
sage: A.is_essential()
False
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> A = hyperplane_arrangements.semiorder(Integer(3))
>>> A.bounded_regions()
(A 3-dimensional polyhedron in QQ^3 defined
```

(continues on next page)

(continued from previous page)

```
    as the convex hull of 3 vertices and 1 line,
 A 3-dimensional polyhedron in QQ^3 defined
    as the convex hull of 3 vertices and 1 line,
 A 3-dimensional polyhedron in QQ^3 defined
    as the convex hull of 3 vertices and 1 line,
 A 3-dimensional polyhedron in QQ^3 defined
    as the convex hull of 6 vertices and 1 line,
 A 3-dimensional polyhedron in QQ^3 defined
    as the convex hull of 3 vertices and 1 line,
 A 3-dimensional polyhedron in QQ^3 defined
    as the convex hull of 3 vertices and 1 line,
 A 3-dimensional polyhedron in QQ^3 defined
    as the convex hull of 3 vertices and 1 line)
>>> A.bounded_regions()[Integer(0)].is_compact()    # the regions are only␣
↪*relatively* bounded
False
>>> A.is_essential()
False
```

center()

Return the center of the hyperplane arrangement.

The polyhedron defined to be the set of all points in the ambient space of the arrangement that lie on all of the hyperplanes.

OUTPUT: a polyhedron

EXAMPLES:

The empty hyperplane arrangement has the entire ambient space as its center:

```
sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: A = H()
sage: A.center()
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and␣
↪2 lines
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = H._first_
↪ngens(2)
>>> A = H()
>>> A.center()
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and␣
↪2 lines
```

The Shi arrangement in dimension 3 has an empty center:

```
sage: A = hyperplane_arrangements.Shi(3)
sage: A.center()
The empty polyhedron in QQ^3
```

```
>>> from sage.all import *
>>> A = hyperplane_arrangements.Shi(Integer(3))
```

(continues on next page)

(continued from previous page)

```
>>> A.center()
The empty polyhedron in QQ^3
```

The Braid arrangement in dimension 3 has a center that is neither empty nor full-dimensional:

```
sage: A = hyperplane_arrangements.braid(3)                                  #␣
↪needs sage.combinat
sage: A.center()                                                            #␣
↪needs sage.combinat
A 1-dimensional polyhedron in QQ^3 defined as the convex hull of 1 vertex and␣
↪1 line
```

```
>>> from sage.all import *
>>> A = hyperplane_arrangements.braid(Integer(3))                          ␣
↪      # needs sage.combinat
>>> A.center()                                                            #␣
↪needs sage.combinat
A 1-dimensional polyhedron in QQ^3 defined as the convex hull of 1 vertex and␣
↪1 line
```

change_ring(*base_ring*)

Return hyperplane arrangement over the new base ring.

INPUT:

- `base_ring` – the new base ring; must be a field for hyperplane arrangements

OUTPUT:

The hyperplane arrangement obtained by changing the base field, as a new hyperplane arrangement.

Warning: While there is often a one-to-one correspondence between the hyperplanes of `self` and those of `self.change_ring(base_ring)`, there is no guarantee that the order in which they appear in `self.hyperplanes()` will match the order in which their counterparts in `self.cone()` will appear in `self.change_ring(base_ring).hyperplanes()`!

EXAMPLES:

```
sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: A = H([(1,1), 0], [(2,3), -1])
sage: A.change_ring(FiniteField(2))
Arrangement <y + 1 | x + y>
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = H._first_
↪ngens(2)
>>> A = H([(Integer(1),Integer(1)), Integer(0)], [(Integer(2),Integer(3)), -
↪Integer(1)])
>>> A.change_ring(FiniteField(Integer(2)))
Arrangement <y + 1 | x + y>
```

characteristic_polynomial()

Return the characteristic polynomial of the hyperplane arrangement.

OUTPUT: the characteristic polynomial in $\mathbf{Q}[x]$

EXAMPLES:

```
sage: a = hyperplane_arrangements.coordinate(2)
sage: a.characteristic_polynomial()
x^2 - 2*x + 1
```

```
>>> from sage.all import *
>>> a = hyperplane_arrangements.coordinate(Integer(2))
>>> a.characteristic_polynomial()
x^2 - 2*x + 1
```

closed_faces(*labelled=True*)

Return the closed faces of the hyperplane arrangement `self` (provided that `self` is defined over a totally ordered field).

Let $\mathcal{A}$ be a hyperplane arrangement in the vector space K^n, whose hyperplanes are the zero sets of the affine-linear functions $u_1, u_2, \ldots, u_N$. (We consider these functions $u_1, u_2, \ldots, u_N$, and not just the hyperplanes, as given. We also assume the field K to be totally ordered.) For any point $x \in K^n$, we define the *sign vector* of x to be the vector $(v_1, v_2, \ldots, v_N) \in \{-1, 0, 1\}^N$ such that (for each i) the number v_i is the sign of $u_i(x)$. For any $v \in \{-1, 0, 1\}^N$, we let F_v be the set of all $x \in K^n$ which have sign vector v. The nonempty ones among all these subsets F_v are called the *open faces* of $\mathcal{A}$. They form a partition of the set K^n.

Furthermore, for any $v = (v_1, v_2, \ldots, v_N) \in \{-1, 0, 1\}^N$, we let G_v be the set of all $x \in K^n$ such that, for every i, the sign of $u_i(x)$ is either 0 or v_i. Then, G_v is a polyhedron. The nonempty ones among all these polyhedra G_v are called the *closed faces* of $\mathcal{A}$. While several sign vectors v can lead to one and the same closed face G_v, we can assign to every closed face a canonical choice of a sign vector: Namely, if G is a closed face of $\mathcal{A}$, then the *sign vector* of G is defined to be the vector $(v_1, v_2, \ldots, v_N) \in \{-1, 0, 1\}^N$ where x is any point in the relative interior of G and where, for each i, the number v_i is the sign of $u_i(x)$. (This does not depend on the choice of x.)

There is a one-to-one correspondence between the closed faces and the open faces of $\mathcal{A}$. It sends a closed face G to the open face F_v, where v is the sign vector of G; this F_v is also the relative interior of G_v. The inverse map sends any open face O to the closure of O.

INPUT:

- `labelled` – boolean (default: `True`); if `True`, then this method returns not the faces itself but rather pairs (v, F) where F is a closed face and v is its sign vector (here, the order and the orientation of the $u_1, u_2, \ldots, u_N$ is as given by `self.hyperplanes()`).

OUTPUT:

A tuple containing the closed faces as polyhedra, or (if `labelled` is set to `True`) the pairs of sign vectors and corresponding closed faces.

Todo

Should the output rather be a dictionary where the keys are the sign vectors and the values are the faces?

EXAMPLES:

```
sage: # needs sage.graphs
sage: a = hyperplane_arrangements.braid(2)
sage: a.hyperplanes()
(Hyperplane t0 - t1 + 0,)
sage: a.closed_faces()
(((0,),  A 1-dimensional polyhedron in QQ^2 defined
         as the convex hull of 1 vertex and 1 line),
 ((1,),  A 2-dimensional polyhedron in QQ^2 defined
         as the convex hull of 1 vertex, 1 ray, 1 line),
 ((-1,), A 2-dimensional polyhedron in QQ^2 defined
         as the convex hull of 1 vertex, 1 ray, 1 line))
sage: a.closed_faces(labelled=False)
(A 1-dimensional polyhedron in QQ^2 defined
     as the convex hull of 1 vertex and 1 line,
 A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 1 vertex, 1 ray, 1 line,
 A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 1 vertex, 1 ray, 1 line)
sage: [(v, F, F.representative_point()) for v, F in a.closed_faces()]
[((0,),  A 1-dimensional polyhedron in QQ^2 defined
         as the convex hull of 1 vertex and 1 line,      (0, 0)),
 ((1,),  A 2-dimensional polyhedron in QQ^2 defined
         as the convex hull of 1 vertex, 1 ray, 1 line,  (0, -1)),
 ((-1,), A 2-dimensional polyhedron in QQ^2 defined
         as the convex hull of 1 vertex, 1 ray, 1 line,  (-1, 0))]

sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: a = H(x, y+1)
sage: a.hyperplanes()
(Hyperplane 0*x + y + 1, Hyperplane x + 0*y + 0)
sage: [(v, F, F.representative_point()) for v, F in a.closed_faces()]
[((0, 0),   A 0-dimensional polyhedron in QQ^2 defined
            as the convex hull of 1 vertex,             (0, -1)),
 ((0, 1),   A 1-dimensional polyhedron in QQ^2 defined
            as the convex hull of 1 vertex and 1 ray,   (1, -1)),
 ((0, -1),  A 1-dimensional polyhedron in QQ^2 defined
            as the convex hull of 1 vertex and 1 ray,   (-1, -1)),
 ((1, 0),   A 1-dimensional polyhedron in QQ^2 defined
            as the convex hull of 1 vertex and 1 ray,   (0, 0)),
 ((1, 1),   A 2-dimensional polyhedron in QQ^2 defined
            as the convex hull of 1 vertex and 2 rays,  (1, 0)),
 ((1, -1),  A 2-dimensional polyhedron in QQ^2 defined
            as the convex hull of 1 vertex and 2 rays,  (-1, 0)),
 ((-1, 0),  A 1-dimensional polyhedron in QQ^2 defined
            as the convex hull of 1 vertex and 1 ray,   (0, -2)),
 ((-1, 1),  A 2-dimensional polyhedron in QQ^2 defined
            as the convex hull of 1 vertex and 2 rays,  (1, -2)),
 ((-1, -1), A 2-dimensional polyhedron in QQ^2 defined
            as the convex hull of 1 vertex and 2 rays,  (-1, -2))]

sage: # needs sage.graphs
sage: a = hyperplane_arrangements.braid(3)
sage: a.hyperplanes()
```

(continues on next page)

(continued from previous page)

```
(Hyperplane 0*t0 + t1 - t2 + 0,
 Hyperplane t0 - t1 + 0*t2 + 0,
 Hyperplane t0 + 0*t1 - t2 + 0)
sage: [(v, F, F.representative_point()) for v, F in a.closed_faces()]
[((0, 0, 0),     A 1-dimensional polyhedron in QQ^3 defined
                 as the convex hull of 1 vertex and 1 line,      (0, 0, 0)),
 ((0, 1, 1),     A 2-dimensional polyhedron in QQ^3 defined
                 as the convex hull of 1 vertex, 1 ray, 1 line,  (0, -1, -1)),
 ((0, -1, -1),   A 2-dimensional polyhedron in QQ^3 defined
                 as the convex hull of 1 vertex, 1 ray, 1 line,  (-1, 0, 0)),
 ((1, 0, 1),     A 2-dimensional polyhedron in QQ^3 defined
                 as the convex hull of 1 vertex, 1 ray, 1 line,  (1, 1, 0)),
 ((1, 1, 1),     A 3-dimensional polyhedron in QQ^3 defined
                 as the convex hull of 1 vertex, 2 rays, 1 line, (0, -1, -2)),
 ((1, -1, 0),    A 2-dimensional polyhedron in QQ^3 defined
                 as the convex hull of 1 vertex, 1 ray, 1 line,  (-1, 0, -1)),
 ((1, -1, 1),    A 3-dimensional polyhedron in QQ^3 defined
                 as the convex hull of 1 vertex, 2 rays, 1 line, (1, 2, 0)),
 ((1, -1, -1),   A 3-dimensional polyhedron in QQ^3 defined
                 as the convex hull of 1 vertex, 2 rays, 1 line, (-2, 0, -1)),
 ((-1, 0, -1),   A 2-dimensional polyhedron in QQ^3 defined
                 as the convex hull of 1 vertex, 1 ray, 1 line,  (0, 0, 1)),
 ((-1, 1, 0),    A 2-dimensional polyhedron in QQ^3 defined
                 as the convex hull of 1 vertex, 1 ray, 1 line,  (1, 0, 1)),
 ((-1, 1, 1),    A 3-dimensional polyhedron in QQ^3 defined
                 as the convex hull of 1 vertex, 2 rays, 1 line, (0, -2, -1)),
 ((-1, 1, -1),   A 3-dimensional polyhedron in QQ^3 defined
                 as the convex hull of 1 vertex, 2 rays, 1 line, (1, 0, 2)),
 ((-1, -1, -1), A 3-dimensional polyhedron in QQ^3 defined
                 as the convex hull of 1 vertex, 2 rays, 1 line, (-1, 0, 1))]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> a = hyperplane_arrangements.braid(Integer(2))
>>> a.hyperplanes()
(Hyperplane t0 - t1 + 0,)
>>> a.closed_faces()
(((0,),  A 1-dimensional polyhedron in QQ^2 defined
         as the convex hull of 1 vertex and 1 line),
 ((1,),  A 2-dimensional polyhedron in QQ^2 defined
         as the convex hull of 1 vertex, 1 ray, 1 line),
 ((-1,), A 2-dimensional polyhedron in QQ^2 defined
         as the convex hull of 1 vertex, 1 ray, 1 line))
>>> a.closed_faces(labelled=False)
(A 1-dimensional polyhedron in QQ^2 defined
     as the convex hull of 1 vertex and 1 line,
 A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 1 vertex, 1 ray, 1 line,
 A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 1 vertex, 1 ray, 1 line)
>>> [(v, F, F.representative_point()) for v, F in a.closed_faces()]
[((0,),  A 1-dimensional polyhedron in QQ^2 defined
```

(continues on next page)

(continued from previous page)

```
        as the convex hull of 1 vertex and 1 line,      (0, 0)),
 ((1,), A 2-dimensional polyhedron in QQ^2 defined
        as the convex hull of 1 vertex, 1 ray, 1 line,  (0, -1)),
 ((-1,), A 2-dimensional polyhedron in QQ^2 defined
        as the convex hull of 1 vertex, 1 ray, 1 line,  (-1, 0))]

>>> H = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = H._first_
↪ngens(2)
>>> a = H(x, y+Integer(1))
>>> a.hyperplanes()
(Hyperplane 0*x + y + 1, Hyperplane x + 0*y + 0)
>>> [(v, F, F.representative_point()) for v, F in a.closed_faces()]
[((0, 0),   A 0-dimensional polyhedron in QQ^2 defined
            as the convex hull of 1 vertex,             (0, -1)),
 ((0, 1),   A 1-dimensional polyhedron in QQ^2 defined
            as the convex hull of 1 vertex and 1 ray,   (1, -1)),
 ((0, -1),  A 1-dimensional polyhedron in QQ^2 defined
            as the convex hull of 1 vertex and 1 ray,   (-1, -1)),
 ((1, 0),   A 1-dimensional polyhedron in QQ^2 defined
            as the convex hull of 1 vertex and 1 ray,   (0, 0)),
 ((1, 1),   A 2-dimensional polyhedron in QQ^2 defined
            as the convex hull of 1 vertex and 2 rays,  (1, 0)),
 ((1, -1),  A 2-dimensional polyhedron in QQ^2 defined
            as the convex hull of 1 vertex and 2 rays,  (-1, 0)),
 ((-1, 0),  A 1-dimensional polyhedron in QQ^2 defined
            as the convex hull of 1 vertex and 1 ray,   (0, -2)),
 ((-1, 1),  A 2-dimensional polyhedron in QQ^2 defined
            as the convex hull of 1 vertex and 2 rays,  (1, -2)),
 ((-1, -1), A 2-dimensional polyhedron in QQ^2 defined
            as the convex hull of 1 vertex and 2 rays,  (-1, -2))]

>>> # needs sage.graphs
>>> a = hyperplane_arrangements.braid(Integer(3))
>>> a.hyperplanes()
(Hyperplane 0*t0 + t1 - t2 + 0,
 Hyperplane t0 - t1 + 0*t2 + 0,
 Hyperplane t0 + 0*t1 - t2 + 0)
>>> [(v, F, F.representative_point()) for v, F in a.closed_faces()]
[((0, 0, 0),    A 1-dimensional polyhedron in QQ^3 defined
                as the convex hull of 1 vertex and 1 line,      (0, 0, 0)),
 ((0, 1, 1),    A 2-dimensional polyhedron in QQ^3 defined
                as the convex hull of 1 vertex, 1 ray, 1 line,  (0, -1, -1)),
 ((0, -1, -1),  A 2-dimensional polyhedron in QQ^3 defined
                as the convex hull of 1 vertex, 1 ray, 1 line,  (-1, 0, 0)),
 ((1, 0, 1),    A 2-dimensional polyhedron in QQ^3 defined
                as the convex hull of 1 vertex, 1 ray, 1 line,  (1, 1, 0)),
 ((1, 1, 1),    A 3-dimensional polyhedron in QQ^3 defined
                as the convex hull of 1 vertex, 2 rays, 1 line, (0, -1, -2)),
 ((1, -1, 0),   A 2-dimensional polyhedron in QQ^3 defined
                as the convex hull of 1 vertex, 1 ray, 1 line,  (-1, 0, -1)),
 ((1, -1, 1),   A 3-dimensional polyhedron in QQ^3 defined
                as the convex hull of 1 vertex, 2 rays, 1 line, (1, 2, 0)),
```

(continues on next page)

(continued from previous page)

```
 ((1, -1, -1),  A 3-dimensional polyhedron in QQ^3 defined
                as the convex hull of 1 vertex, 2 rays, 1 line, (-2, 0, -1)),
 ((-1, 0, -1),  A 2-dimensional polyhedron in QQ^3 defined
                as the convex hull of 1 vertex, 1 ray, 1 line,  (0, 0, 1)),
 ((-1, 1, 0),   A 2-dimensional polyhedron in QQ^3 defined
                as the convex hull of 1 vertex, 1 ray, 1 line,  (1, 0, 1)),
 ((-1, 1, 1),   A 3-dimensional polyhedron in QQ^3 defined
                as the convex hull of 1 vertex, 2 rays, 1 line, (0, -2, -1)),
 ((-1, 1, -1),  A 3-dimensional polyhedron in QQ^3 defined
                as the convex hull of 1 vertex, 2 rays, 1 line, (1, 0, 2)),
 ((-1, -1, -1), A 3-dimensional polyhedron in QQ^3 defined
                as the convex hull of 1 vertex, 2 rays, 1 line, (-1, 0, 1))]
```

Let us check that the number of closed faces with a given dimension computed using `self.closed_faces()` equals the one computed using *face_vector()*:

```
sage: def test_number(a):
....:     Qx = PolynomialRing(QQ, 'x'); x = Qx.gen()
....:     RHS = Qx.sum(vi * x ** i for i, vi in enumerate(a.face_vector()))
....:     LHS = Qx.sum(x ** F[1].dim() for F in a.closed_faces())
....:     return LHS == RHS
sage: a = hyperplane_arrangements.Catalan(2)
sage: test_number(a)                                                    #␣
↪needs sage.combinat
True
sage: a = hyperplane_arrangements.Shi(3)
sage: test_number(a)                          # long time               #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> def test_number(a):
...     Qx = PolynomialRing(QQ, 'x'); x = Qx.gen()
...     RHS = Qx.sum(vi * x ** i for i, vi in enumerate(a.face_vector()))
...     LHS = Qx.sum(x ** F[Integer(1)].dim() for F in a.closed_faces())
...     return LHS == RHS
>>> a = hyperplane_arrangements.Catalan(Integer(2))
>>> test_number(a)                                                    #␣
↪needs sage.combinat
True
>>> a = hyperplane_arrangements.Shi(Integer(3))
>>> test_number(a)                          # long time               #␣
↪needs sage.combinat
True
```

cocharacteristic_polynomial()

Return the cocharacteristic polynomial of `self`.

The cocharacteristic polynomial of a hyperplane arrangement A is defined by

$$\Psi_A(z) := \sum_{X \in L} |\mu(B, X)| z^{\dim X},$$

where L is the intersection poset of A, B is the minimal element of L (here, the 0 dimensional subspace), and μ is the Möbius function of L.

OUTPUT: the cocharacteristic polynomial in $\mathbf{Z}[z]$

EXAMPLES:

```
sage: A = hyperplane_arrangements.coordinate(2)
sage: A.cocharacteristic_polynomial()                                   #␣
↪needs sage.graphs
z^2 + 2*z + 1
sage: B = hyperplane_arrangements.braid(3)
sage: B.cocharacteristic_polynomial()                                   #␣
↪needs sage.graphs
2*z^3 + 3*z^2 + z
```

```
>>> from sage.all import *
>>> A = hyperplane_arrangements.coordinate(Integer(2))
>>> A.cocharacteristic_polynomial()                                    #␣
↪needs sage.graphs
z^2 + 2*z + 1
>>> B = hyperplane_arrangements.braid(Integer(3))
>>> B.cocharacteristic_polynomial()                                    #␣
↪needs sage.graphs
2*z^3 + 3*z^2 + z
```

cone(*variable='t'*)

Return the cone over the hyperplane arrangement.

INPUT:

- `variable` – string; the name of the additional variable

OUTPUT:

A new hyperplane arrangement L. Its equations consist of $[0, -d, a_1, \ldots, a_n]$ for each $[d, a_1, \ldots, a_n]$ in the original arrangement and the equation $[0, 1, 0, \ldots, 0]$ (maybe not in this order).

> **Warning**
>
> While there is an almost-one-to-one correspondence between the hyperplanes of `self` and those of `self.cone()`, there is no guarantee that the order in which they appear in `self.hyperplanes()` will match the order in which their counterparts in `self.cone()` will appear in `self.cone().hyperplanes()`! This warning does not apply to ordered hyperplane arrangements.

EXAMPLES:

```
sage: # needs sage.combinat
sage: a.<x,y,z> = hyperplane_arrangements.semiorder(3)
sage: b = a.cone()
sage: a.characteristic_polynomial().factor()
x * (x^2 - 6*x + 12)
sage: b.characteristic_polynomial().factor()
(x - 1) * x * (x^2 - 6*x + 12)
sage: a.hyperplanes()
(Hyperplane 0*x + y - z - 1,
```

(continues on next page)

(continued from previous page)

```
 Hyperplane 0*x + y - z + 1,
 Hyperplane x - y + 0*z - 1,
 Hyperplane x - y + 0*z + 1,
 Hyperplane x + 0*y - z - 1,
 Hyperplane x + 0*y - z + 1)
sage: b.hyperplanes()
(Hyperplane -t + 0*x + y - z + 0,
 Hyperplane -t + x - y + 0*z + 0,
 Hyperplane -t + x + 0*y - z + 0,
 Hyperplane t + 0*x + 0*y + 0*z + 0,
 Hyperplane t + 0*x + y - z + 0,
 Hyperplane t + x - y + 0*z + 0,
 Hyperplane t + x + 0*y - z + 0)
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> a = hyperplane_arrangements.semiorder(Integer(3), names=('x', 'y', 'z',));
↪ (x, y, z,) = a._first_ngens(3)
>>> b = a.cone()
>>> a.characteristic_polynomial().factor()
x * (x^2 - 6*x + 12)
>>> b.characteristic_polynomial().factor()
(x - 1) * x * (x^2 - 6*x + 12)
>>> a.hyperplanes()
(Hyperplane 0*x + y - z - 1,
 Hyperplane 0*x + y - z + 1,
 Hyperplane x - y + 0*z - 1,
 Hyperplane x - y + 0*z + 1,
 Hyperplane x + 0*y - z - 1,
 Hyperplane x + 0*y - z + 1)
>>> b.hyperplanes()
(Hyperplane -t + 0*x + y - z + 0,
 Hyperplane -t + x - y + 0*z + 0,
 Hyperplane -t + x + 0*y - z + 0,
 Hyperplane t + 0*x + 0*y + 0*z + 0,
 Hyperplane t + 0*x + y - z + 0,
 Hyperplane t + x - y + 0*z + 0,
 Hyperplane t + x + 0*y - z + 0)
```

defining_polynomial()

Return the defining polynomial of `A`.

Let $A = (H_i)_i$ be a hyperplane arrangement in a vector space V corresponding to the null spaces of $\alpha_{H_i} \in V^*$. Then the *defining polynomial* of A is given by

$$Q(A) = \prod_i \alpha_{H_i} \in S(V^*).$$

EXAMPLES:

```
sage: H.<x,y,z> = HyperplaneArrangements(QQ)
sage: A = H([2*x + y - z, -x - 2*y + z])
sage: p = A.defining_polynomial(); p
```

(continues on next page)

(continued from previous page)

```
-2*x^2 - 5*x*y - 2*y^2 + 3*x*z + 3*y*z - z^2
sage: p.factor()
(-1) * (x + 2*y - z) * (2*x + y - z)
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,) = H._
↪first_ngens(3)
>>> A = H([Integer(2)*x + y - z, -x - Integer(2)*y + z])
>>> p = A.defining_polynomial(); p
-2*x^2 - 5*x*y - 2*y^2 + 3*x*z + 3*y*z - z^2
>>> p.factor()
(-1) * (x + 2*y - z) * (2*x + y - z)
```

deletion(*hyperplanes*)

Return the hyperplane arrangement obtained by removing `h`.

INPUT:

- `h` – a hyperplane or hyperplane arrangement

OUTPUT:

A new hyperplane arrangement with the given hyperplane(s) `h` removed.

See also

restriction()

EXAMPLES:

```
sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: A = H([0,1,0], [1,0,1], [-1,0,1], [0,1,-1], [0,1,1]);  A
Arrangement of 5 hyperplanes of dimension 2 and rank 2
sage: A.deletion(x)
Arrangement <y - 1 | y + 1 | x - y | x + y>
sage: h = H([0,1,0], [0,1,1])
sage: A.deletion(h)
Arrangement <y - 1 | y + 1 | x - y>
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = H._first_
↪ngens(2)
>>> A = H([Integer(0),Integer(1),Integer(0)], [Integer(1),Integer(0),
↪Integer(1)], [-Integer(1),Integer(0),Integer(1)], [Integer(0),Integer(1),-
↪Integer(1)], [Integer(0),Integer(1),Integer(1)]);  A
Arrangement of 5 hyperplanes of dimension 2 and rank 2
>>> A.deletion(x)
Arrangement <y - 1 | y + 1 | x - y | x + y>
>>> h = H([Integer(0),Integer(1),Integer(0)], [Integer(0),Integer(1),
↪Integer(1)])
>>> A.deletion(h)
Arrangement <y - 1 | y + 1 | x - y>
```

derivation_module_basis(*algorithm='singular'*)

Return a basis for the derivation module of `self` if one exists, otherwise return `None`.

See also

derivation_module_free_chain(), *is_free()*

INPUT:

- `algorithm` – (default: `'singular'`) can be one of the following:
  - `'singular'` – use Singular's minimal free resolution
  - `'BC'` – use the algorithm given by Barakat and Cuntz in [BC2012] (much slower than using Singular)

OUTPUT:

A basis for the derivation module (over S, the *symmetric space*) as vectors of a free module over S.

ALGORITHM:

Singular

This gets the reduced syzygy module of the Jacobian ideal of the defining polynomial f of `self`. It then checks Saito's criterion that the determinant of the basis matrix is a scalar multiple of f. If the basis matrix is not square or it fails Saito's criterion, then we check if the arrangement is free. If it is free, then we fall back to the Barakat-Cuntz algorithm.

BC

Return the product of the derivation module free chain matrices. See Section 6 of [BC2012].

EXAMPLES:

```
sage: # needs sage.combinat sage.groups
sage: W = WeylGroup(['A', 2], prefix='s')
sage: A = W.long_element().inversion_arrangement()
sage: A.derivation_module_basis()
[(a1, a2), (0, a1*a2 + a2^2)]
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.groups
>>> W = WeylGroup(['A', Integer(2)], prefix='s')
>>> A = W.long_element().inversion_arrangement()
>>> A.derivation_module_basis()
[(a1, a2), (0, a1*a2 + a2^2)]
```

derivation_module_free_chain()

Return a free chain for the derivation module if one exists, otherwise return `None`.

See also

is_free()

EXAMPLES:

```
sage: # needs sage.combinat sage.groups
sage: W = WeylGroup(['A',3], prefix='s')
sage: A = W.long_element().inversion_arrangement()
sage: for M in A.derivation_module_free_chain(): print("%s\n"%M)
[ 1  0  0]
[ 0  1  0]
[ 0  0 a3]

[ 1  0  0]
[ 0  0  1]
[ 0 a2  0]

[  1   0   0]
[  0  -1  -1]
[  0  a2 -a3]

[ 0  1  0]
[ 0  0  1]
[a1  0  0]

[ 1  0 -1]
[a3 -1  0]
[a1  0 a2]

[       1         0         0]
[      a3        -1        -1]
[       0        a1 -a2 - a3]
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.groups
>>> W = WeylGroup(['A',Integer(3)], prefix='s')
>>> A = W.long_element().inversion_arrangement()
>>> for M in A.derivation_module_free_chain(): print("%s\n"%M)
[ 1  0  0]
[ 0  1  0]
[ 0  0 a3]
<BLANKLINE>
[ 1  0  0]
[ 0  0  1]
[ 0 a2  0]
<BLANKLINE>
[  1   0   0]
[  0  -1  -1]
[  0  a2 -a3]
<BLANKLINE>
[ 0  1  0]
[ 0  0  1]
[a1  0  0]
<BLANKLINE>
[ 1  0 -1]
[a3 -1  0]
[a1  0 a2]
```

(continues on next page)

(continued from previous page)

```
<BLANKLINE>
[       1        0        0]
[      a3       -1       -1]
[       0       a1 -a2 - a3]
<BLANKLINE>
```

dimension()

Return the ambient space dimension of the arrangement.

OUTPUT: integer

EXAMPLES:

```
sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: (x | x-1 | x+1).dimension()
2
sage: H(x).dimension()
2
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = H._first_
→ngens(2)
>>> (x | x-Integer(1) | x+Integer(1)).dimension()
2
>>> H(x).dimension()
2
```

distance_between_regions(*region1*, *region2*)

Return the number of hyperplanes separating the two regions.

INPUT:

- `region1`, `region2` – regions of the arrangement or representative points of regions

OUTPUT: integer; the number of hyperplanes separating the two regions

EXAMPLES:

```
sage: c = hyperplane_arrangements.coordinate(2)
sage: r = c.region_containing_point([-1, -1])
sage: s = c.region_containing_point([1, 1])
sage: c.distance_between_regions(r, s)
2
sage: c.distance_between_regions(s, s)
0
```

```
>>> from sage.all import *
>>> c = hyperplane_arrangements.coordinate(Integer(2))
>>> r = c.region_containing_point([-Integer(1), -Integer(1)])
>>> s = c.region_containing_point([Integer(1), Integer(1)])
>>> c.distance_between_regions(r, s)
2
>>> c.distance_between_regions(s, s)
0
```

distance_enumerator(*base_region*)

Return the generating function for the number of hyperplanes at given distance.

INPUT:

- `base_region` – region of arrangement or point in region

OUTPUT:

A polynomial $f(x)$ for which the coefficient of x^i is the number of hyperplanes of distance i from `base_region`, i.e., the number of hyperplanes separated by i hyperplanes from `base_region`.

EXAMPLES:

```
sage: c = hyperplane_arrangements.coordinate(3)
sage: c.distance_enumerator(c.region_containing_point([1,1,1]))
x^3 + 3*x^2 + 3*x + 1
```

```
>>> from sage.all import *
>>> c = hyperplane_arrangements.coordinate(Integer(3))
>>> c.distance_enumerator(c.region_containing_point([Integer(1),Integer(1),
↪Integer(1)]))
x^3 + 3*x^2 + 3*x + 1
```

doubly_indexed_whitney_number(*i*, *j*, *kind=1*)

Return the i, j-th doubly-indexed Whitney number.

If `kind=1`, this number is obtained by adding the Möbius function values $mu(x, y)$ over all x, y in the intersection poset with $\mathrm{rank}(x) = i$ and $\mathrm{rank}(y) = j$.

If $kind = 2$, this number is the number of elements x, y in the intersection poset such that $x \leq y$ with ranks i and j, respectively.

INPUT:

- `i`, `j` – integers
- `kind` – (default: 1) 1 or 2

OUTPUT:

Integer. The (i, j)-th entry of the `kind` Whitney number.

See also

`whitney_number()`, *`whitney_data()`*

EXAMPLES:

```
sage: # needs sage.combinat
sage: A = hyperplane_arrangements.Shi(3)
sage: A.doubly_indexed_whitney_number(0, 2)
9
sage: A.whitney_number(2)
9
sage: A.doubly_indexed_whitney_number(1, 2)
-15
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> A = hyperplane_arrangements.Shi(Integer(3))
>>> A.doubly_indexed_whitney_number(Integer(0), Integer(2))
9
>>> A.whitney_number(Integer(2))
9
>>> A.doubly_indexed_whitney_number(Integer(1), Integer(2))
-15
```

REFERENCES:

- [GZ1983]

essentialization()

Return the essentialization of the hyperplane arrangement.

The essentialization of a hyperplane arrangement whose base field has characteristic 0 is obtained by intersecting the hyperplanes by the space spanned by their normal vectors.

OUTPUT:

The essentialization $\mathcal{A}'$ of $\mathcal{A}$ as a new hyperplane arrangement.

EXAMPLES:

```
sage: a = hyperplane_arrangements.braid(3)                                 #␣
↪needs sage.graphs
sage: a.is_essential()                                                     #␣
↪needs sage.graphs
False
sage: a.essentialization()                                                 #␣
↪needs sage.graphs
Arrangement <t1 - t2 | t1 + 2*t2 | 2*t1 + t2>

sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: B = H([(1,0),1], [(1,0),-1])
sage: B.is_essential()
False
sage: B.essentialization()
Arrangement <-x + 1 | x + 1>
sage: B.essentialization().parent()
Hyperplane arrangements in 1-dimensional linear space over
Rational Field with coordinate x

sage: H.<x,y> = HyperplaneArrangements(GF(2))
sage: C = H([(1,1),1], [(1,1),0])
sage: C.essentialization()
Arrangement <y | y + 1>

sage: h = hyperplane_arrangements.semiorder(4)
sage: h.essentialization()
Arrangement of 12 hyperplanes of dimension 3 and rank 3
```

```
>>> from sage.all import *
>>> a = hyperplane_arrangements.braid(Integer(3))                              ␣
↪     # needs sage.graphs
>>> a.is_essential()                                                          #␣
↪needs sage.graphs
False
>>> a.essentialization()                                                      #␣
↪needs sage.graphs
Arrangement <t1 - t2 | t1 + 2*t2 | 2*t1 + t2>

>>> H = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = H._first_
↪ngens(2)
>>> B = H([(Integer(1),Integer(0)),Integer(1)], [(Integer(1),Integer(0)),-
↪Integer(1)])
>>> B.is_essential()
False
>>> B.essentialization()
Arrangement <-x + 1 | x + 1>
>>> B.essentialization().parent()
Hyperplane arrangements in 1-dimensional linear space over
Rational Field with coordinate x

>>> H = HyperplaneArrangements(GF(Integer(2)), names=('x', 'y',)); (x, y,) =␣
↪H._first_ngens(2)
>>> C = H([(Integer(1),Integer(1)),Integer(1)], [(Integer(1),Integer(1)),
↪Integer(0)])
>>> C.essentialization()
Arrangement <y | y + 1>

>>> h = hyperplane_arrangements.semiorder(Integer(4))
>>> h.essentialization()
Arrangement of 12 hyperplanes of dimension 3 and rank 3
```

face_product(*F*, *G*, *normalize=True*)

Return the product FG in the face semigroup of `self`, where F and G are two closed faces of `self`.

The face semigroup of a hyperplane arrangement $\mathcal{A}$ is defined as follows: As a set, it is the set of all open faces of `self` (see `closed_faces()`). Its product is defined by the following rule: If F and G are two open faces of $\mathcal{A}$, then FG is an open face of $\mathcal{A}$, and for every hyperplane $H \in \mathcal{A}$, the open face FG lies on the same side of H as F unless $F \subseteq H$, in which case FG lies on the same side of H as G. Alternatively, FG can be defined as follows: If f and g are two points in F and G, respectively, then FG is the face that contains the point $(f + \varepsilon g)/(1 + \varepsilon)$ for any sufficiently small positive ε.

In our implementation, the face semigroup consists of closed faces rather than open faces (thanks to the 1-to-1 correspondence between open faces and closed faces, this is not really a different semigroup); these closed faces are given as polyhedra.

The face semigroup of a hyperplane arrangement is always a left-regular band (i.e., a semigroup satisfying the identities $x^2 = x$ and $xyx = xy$). When the arrangement is central, then this semigroup is a monoid. See [Br2000] (Appendix A in particular) for further properties.

INPUT:

- `F`, `G` – two faces of `self` (as polyhedra)
- `normalize` – boolean (default: `True`); if `True`, then this method returns the precise instance of FG in

the list returned by `self.closed_faces()`, rather than creating a new instance

EXAMPLES:

```
sage: # needs sage.graphs
sage: a = hyperplane_arrangements.braid(3)
sage: a.hyperplanes()
(Hyperplane 0*t0 + t1 - t2 + 0,
 Hyperplane t0 - t1 + 0*t2 + 0,
 Hyperplane t0 + 0*t1 - t2 + 0)
sage: faces = {F0: F1 for F0, F1 in a.closed_faces()}
sage: xGyEz = faces[(0, 1, 1)]   # closed face x >= y = z
sage: xGyEz.representative_point()
(0, -1, -1)
sage: xGyEz = faces[(0, 1, 1)]   # closed face x >= y = z
sage: xGyEz.representative_point()
(0, -1, -1)
sage: yGxGz = faces[(1, -1, 1)]  # closed face y >= x >= z
sage: xGyGz = faces[(1, 1, 1)]   # closed face x >= y >= z
sage: a.face_product(xGyEz, yGxGz) == xGyGz
True
sage: a.face_product(yGxGz, xGyEz) == yGxGz
True
sage: xEzGy = faces[(-1, 1, 0)]  # closed face x = z >= y
sage: xGzGy = faces[(-1, 1, 1)]  # closed face x >= z >= y
sage: a.face_product(xEzGy, yGxGz) == xGzGy
True
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> a = hyperplane_arrangements.braid(Integer(3))
>>> a.hyperplanes()
(Hyperplane 0*t0 + t1 - t2 + 0,
 Hyperplane t0 - t1 + 0*t2 + 0,
 Hyperplane t0 + 0*t1 - t2 + 0)
>>> faces = {F0: F1 for F0, F1 in a.closed_faces()}
>>> xGyEz = faces[(Integer(0), Integer(1), Integer(1))]   # closed face x >=
↪y = z
>>> xGyEz.representative_point()
(0, -1, -1)
>>> xGyEz = faces[(Integer(0), Integer(1), Integer(1))]   # closed face x >=
↪y = z
>>> xGyEz.representative_point()
(0, -1, -1)
>>> yGxGz = faces[(Integer(1), -Integer(1), Integer(1))]  # closed face y >=
↪x >= z
>>> xGyGz = faces[(Integer(1), Integer(1), Integer(1))]   # closed face x >=
↪y >= z
>>> a.face_product(xGyEz, yGxGz) == xGyGz
True
>>> a.face_product(yGxGz, xGyEz) == yGxGz
True
>>> xEzGy = faces[(-Integer(1), Integer(1), Integer(0))]  # closed face x = z
↪>= y
```

(continues on next page)

(continued from previous page)

```
>>> xGzGy = faces[(-Integer(1), Integer(1), Integer(1))]  # closed face x >=␣
↪z >= y
>>> a.face_product(xEzGy, yGxGz) == xGzGy
True
```

face_semigroup_algebra(*field=None*, *names='e'*)

Return the face semigroup algebra of `self`.

This is the semigroup algebra of the face semigroup of `self` (see *face_product()* for the definition of the semigroup).

Due to limitations of the current Sage codebase (e.g., semigroup algebras do not profit from the functionality of the `FiniteDimensionalAlgebra` class), this is implemented not as a semigroup algebra, but as a `FiniteDimensionalAlgebra`. The closed faces of `self` (in the order in which the *closed_faces()* method outputs them) are identified with the vectors $(0, 0, \ldots, 0, 1, 0, 0, \ldots, 0)$ (with the 1 moving from left to right).

INPUT:

- `field` – a field (default: $\mathbf{Q}$), to be used as the base ring for the algebra (can also be a commutative ring, but then certain representation-theoretical methods might misbehave)
- `names` – (default: `'e'`) string; names for the basis elements of the algebra

Todo

Also implement it as an actual semigroup algebra?

EXAMPLES:

```
sage: # needs sage.graphs
sage: a = hyperplane_arrangements.braid(3)
sage: [(i, F[0]) for i, F in enumerate(a.closed_faces())]
[(0, (0, 0, 0)),
 (1, (0, 1, 1)),
 (2, (0, -1, -1)),
 (3, (1, 0, 1)),
 (4, (1, 1, 1)),
 (5, (1, -1, 0)),
 (6, (1, -1, 1)),
 (7, (1, -1, -1)),
 (8, (-1, 0, -1)),
 (9, (-1, 1, 0)),
 (10, (-1, 1, 1)),
 (11, (-1, 1, -1)),
 (12, (-1, -1, -1))]
sage: U = a.face_semigroup_algebra(); U
Finite-dimensional algebra of degree 13 over Rational Field
sage: e0, e1, e2, e3, e4, e5, e6, e7, e8, e9, e10, e11, e12 = U.basis()
sage: e0 * e1
e1
sage: e0 * e5
e5
```

(continues on next page)

(continued from previous page)

```
sage: e5 * e0
e5
sage: e3 * e2
e6
sage: e7 * e12
e7
sage: e3 * e12
e6
sage: e4 * e8
e4
sage: e8 * e4
e11
sage: e8 * e1
e11
sage: e5 * e12
e7
sage: (e3 + 2*e4) * (e1 - e7)
e4 - e6

sage: U3 = a.face_semigroup_algebra(field=GF(3)); U3                        #␣
↪needs sage.graphs sage.rings.finite_rings
Finite-dimensional algebra of degree 13 over Finite Field of size 3
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> a = hyperplane_arrangements.braid(Integer(3))
>>> [(i, F[Integer(0)]) for i, F in enumerate(a.closed_faces())]
[(0, (0, 0, 0)),
 (1, (0, 1, 1)),
 (2, (0, -1, -1)),
 (3, (1, 0, 1)),
 (4, (1, 1, 1)),
 (5, (1, -1, 0)),
 (6, (1, -1, 1)),
 (7, (1, -1, -1)),
 (8, (-1, 0, -1)),
 (9, (-1, 1, 0)),
 (10, (-1, 1, 1)),
 (11, (-1, 1, -1)),
 (12, (-1, -1, -1))]
>>> U = a.face_semigroup_algebra(); U
Finite-dimensional algebra of degree 13 over Rational Field
>>> e0, e1, e2, e3, e4, e5, e6, e7, e8, e9, e10, e11, e12 = U.basis()
>>> e0 * e1
e1
>>> e0 * e5
e5
>>> e5 * e0
e5
>>> e3 * e2
e6
>>> e7 * e12
```

(continues on next page)

(continued from previous page)

```
e7
>>> e3 * e12
e6
>>> e4 * e8
e4
>>> e8 * e4
e11
>>> e8 * e1
e11
>>> e5 * e12
e7
>>> (e3 + Integer(2)*e4) * (e1 - e7)
e4 - e6

>>> U3 = a.face_semigroup_algebra(field=GF(Integer(3))); U3
↪     # needs sage.graphs sage.rings.finite_rings
Finite-dimensional algebra of degree 13 over Finite Field of size 3
```

face_vector()

Return the face vector.

OUTPUT: a vector of integers

The d-th entry is the number of faces of dimension d. A *face* is the intersection of a region with a hyperplane of the arrangement.

EXAMPLES:

```
sage: A = hyperplane_arrangements.Shi(3)
sage: A.face_vector()                                                    #
↪needs sage.combinat
(0, 6, 21, 16)
```

```
>>> from sage.all import *
>>> A = hyperplane_arrangements.Shi(Integer(3))
>>> A.face_vector()                                                    #
↪needs sage.combinat
(0, 6, 21, 16)
```

has_good_reduction(p)

Return whether the hyperplane arrangement has good reduction mod p.

Let A be a hyperplane arrangement with equations defined over the integers, and let B be the hyperplane arrangement defined by reducing these equations modulo a prime p. Then A has good reduction modulo p if the intersection posets of A and B are isomorphic.

INPUT:

- p – prime number

OUTPUT: boolean

EXAMPLES:

```
sage: # needs sage.combinat
sage: a = hyperplane_arrangements.semiorder(3)
```

(continues on next page)

(continued from previous page)

```
sage: a.has_good_reduction(5)
True
sage: a.has_good_reduction(3)
False
sage: b = a.change_ring(GF(3))
sage: a.characteristic_polynomial()
x^3 - 6*x^2 + 12*x
sage: b.characteristic_polynomial()  # not equal to that for a
x^3 - 6*x^2 + 10*x
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> a = hyperplane_arrangements.semiorder(Integer(3))
>>> a.has_good_reduction(Integer(5))
True
>>> a.has_good_reduction(Integer(3))
False
>>> b = a.change_ring(GF(Integer(3)))
>>> a.characteristic_polynomial()
x^3 - 6*x^2 + 12*x
>>> b.characteristic_polynomial()  # not equal to that for a
x^3 - 6*x^2 + 10*x
```

hyperplanes()

Return the hyperplanes in the arrangement as a tuple.

OUTPUT: a tuple

EXAMPLES:

```
sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: A = H([1,1,0], [2,3,-1], [4,5,3])
sage: A.hyperplanes()
(Hyperplane x + 0*y + 1, Hyperplane 3*x - y + 2, Hyperplane 5*x + 3*y + 4)
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = H._first_
↪ngens(2)
>>> A = H([Integer(1),Integer(1),Integer(0)], [Integer(2),Integer(3),-
↪Integer(1)], [Integer(4),Integer(5),Integer(3)])
>>> A.hyperplanes()
(Hyperplane x + 0*y + 1, Hyperplane 3*x - y + 2, Hyperplane 5*x + 3*y + 4)
```

Note that the hyperplanes can be indexed as if they were a list:

```
sage: A[0]
Hyperplane x + 0*y + 1
```

```
>>> from sage.all import *
>>> A[Integer(0)]
Hyperplane x + 0*y + 1
```

intersection_poset(*element_label='int'*)

Return the intersection poset of the hyperplane arrangement.

INPUT:

- `element_label` – (default: `'int'`) specify how an intersection should be represented; must be one of the following:
  - `'subspace'` – as a subspace
  - `'subset'` – as a subset of the defining hyperplanes
  - `'int'` – as an integer

OUTPUT:

The poset of non-empty intersections of hyperplanes, with intersections represented by integers, subsets of integers or subspaces (see the examples for more details).

EXAMPLES:

By default, the elements of the poset are the integers from 0 through the cardinality of the poset *minus one*. The element labelled 0 always corresponds to the ambient vector space, and the hyperplanes themselves are labelled $1, 2, \ldots, n$, where n is the number of hyperplanes of the arrangement.

```
sage: A = hyperplane_arrangements.coordinate(2)
sage: L = A.intersection_poset(); L                                         #␣
↪needs sage.combinat
Finite poset containing 4 elements
sage: sorted(L)                                                             #␣
↪needs sage.combinat
[0, 1, 2, 3]
sage: L.level_sets()                                                        #␣
↪needs sage.combinat
[[0], [1, 2], [3]]
```

```
>>> from sage.all import *
>>> A = hyperplane_arrangements.coordinate(Integer(2))
>>> L = A.intersection_poset(); L                                         #␣
↪needs sage.combinat
Finite poset containing 4 elements
>>> sorted(L)                                                             #␣
↪needs sage.combinat
[0, 1, 2, 3]
>>> L.level_sets()                                                        #␣
↪needs sage.combinat
[[0], [1, 2], [3]]
```

```
sage: # needs sage.combinat
sage: A = hyperplane_arrangements.semiorder(3)
sage: L = A.intersection_poset(); L
Finite poset containing 19 elements
sage: sorted(L)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]
sage: [sorted(level_set) for level_set in L.level_sets()]
[[0], [1, 2, 3, 4, 5, 6], [7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]]
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> A = hyperplane_arrangements.semiorder(Integer(3))
>>> L = A.intersection_poset(); L
Finite poset containing 19 elements
>>> sorted(L)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]
>>> [sorted(level_set) for level_set in L.level_sets()]
[[0], [1, 2, 3, 4, 5, 6], [7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18]]
```

By passing the argument `element_label="subset"`, each element of the intersection poset is labelled by the set of indices of the hyperplanes whose intersection is said element. The index of a hyperplane is its index in `self.hyperplanes()`.

```
sage: A = hyperplane_arrangements.semiorder(3)
sage: L = A.intersection_poset(element_label='subset')                        #␣
↪needs sage.combinat
sage: [sorted(level, key=sorted) for level in L.level_sets()]                 #␣
↪needs sage.combinat
[[{}],
 [{0}, {1}, {2}, {3}, {4}, {5}],
 [{0, 2}, {0, 3}, {0, 4}, {0, 5}, {1, 2}, {1, 3}, {1, 4}, {1, 5}, {2, 4}, {2,␣
↪5}, {3, 4}, {3, 5}]]
```

```
>>> from sage.all import *
>>> A = hyperplane_arrangements.semiorder(Integer(3))
>>> L = A.intersection_poset(element_label='subset')                        #␣
↪needs sage.combinat
>>> [sorted(level, key=sorted) for level in L.level_sets()]                 #␣
↪needs sage.combinat
[[{}],
 [{0}, {1}, {2}, {3}, {4}, {5}],
 [{0, 2}, {0, 3}, {0, 4}, {0, 5}, {1, 2}, {1, 3}, {1, 4}, {1, 5}, {2, 4}, {2,␣
↪5}, {3, 4}, {3, 5}]]
```

```
sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: A = H((y, y-1, y+1, x-y, x+y))
sage: L = A.intersection_poset(element_label='subset')                        #␣
↪needs sage.combinat
sage: sorted(L, key=sorted)                                                   #␣
↪needs sage.combinat
[{}, {0}, {0, 3}, {0, 4}, {1}, {1, 3, 4}, {2}, {2, 3}, {2, 4}, {3}, {4}]
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = H._first_
↪ngens(2)
>>> A = H((y, y-Integer(1), y+Integer(1), x-y, x+y))
>>> L = A.intersection_poset(element_label='subset')                        #␣
↪needs sage.combinat
>>> sorted(L, key=sorted)                                                   #␣
↪needs sage.combinat
[{}, {0}, {0, 3}, {0, 4}, {1}, {1, 3, 4}, {2}, {2, 3}, {2, 4}, {3}, {4}]
```

One can instead use affine subspaces as elements, which is what is used to compute the poset in the first place:

```
sage: A = hyperplane_arrangements.coordinate(2)
sage: L = A.intersection_poset(element_label='subspace'); L                    #␣
↪needs sage.combinat
Finite poset containing 4 elements
sage: sorted(L, key=lambda S: (S.dimension(),                                  #␣
↪needs sage.combinat
....:                          S.linear_part().basis_matrix()))
[Affine space p + W where:
   p = (0, 0)
   W = Vector space of degree 2 and dimension 0 over Rational Field
       Basis matrix: [],
 Affine space p + W where:
   p = (0, 0)
   W = Vector space of degree 2 and dimension 1 over Rational Field
       Basis matrix: [0 1],
 Affine space p + W where:
   p = (0, 0)
   W = Vector space of degree 2 and dimension 1 over Rational Field
       Basis matrix: [1 0],
 Affine space p + W where:
   p = (0, 0)
   W = Vector space of dimension 2 over Rational Field]
```

```
>>> from sage.all import *
>>> A = hyperplane_arrangements.coordinate(Integer(2))
>>> L = A.intersection_poset(element_label='subspace'); L                    #␣
↪needs sage.combinat
Finite poset containing 4 elements
>>> sorted(L, key=lambda S: (S.dimension(),                                  #␣
↪needs sage.combinat
...                          S.linear_part().basis_matrix()))
[Affine space p + W where:
   p = (0, 0)
   W = Vector space of degree 2 and dimension 0 over Rational Field
       Basis matrix: [],
 Affine space p + W where:
   p = (0, 0)
   W = Vector space of degree 2 and dimension 1 over Rational Field
       Basis matrix: [0 1],
 Affine space p + W where:
   p = (0, 0)
   W = Vector space of degree 2 and dimension 1 over Rational Field
       Basis matrix: [1 0],
 Affine space p + W where:
   p = (0, 0)
   W = Vector space of dimension 2 over Rational Field]
```

is_central(*certificate=False*)

Test whether the intersection of all the hyperplanes is nonempty.

A hyperplane arrangement is central if the intersection of all the hyperplanes in the arrangement is nonempty.

INPUT:

- `certificate` – boolean (default: `False`); specifies whether to return the center as a polyhedron (possibly empty) as part of the output

OUTPUT: if `certificate` is `True`, returns a tuple containing:

1. A boolean
2. The polyhedron defined to be the intersection of all the hyperplanes

If `certificate` is `False`, returns a boolean.

EXAMPLES:

```
sage: a = hyperplane_arrangements.braid(2)                                  #␣
↪needs sage.graphs
sage: a.is_central()                                                        #␣
↪needs sage.graphs
True
```

```
>>> from sage.all import *
>>> a = hyperplane_arrangements.braid(Integer(2))                            ␣
↪       # needs sage.graphs
>>> a.is_central()                                                         #␣
↪needs sage.graphs
True
```

The Catalan arrangement in dimension 3 is not central:

```
sage: b = hyperplane_arrangements.Catalan(3)
sage: b.is_central(certificate=True)
(False, The empty polyhedron in QQ^3)
```

```
>>> from sage.all import *
>>> b = hyperplane_arrangements.Catalan(Integer(3))
>>> b.is_central(certificate=True)
(False, The empty polyhedron in QQ^3)
```

The empty arrangement in dimension 5 is central:

```
sage: H = HyperplaneArrangements(QQ, names=tuple(['x'+str(i) for i in␣
↪range(7)]))
sage: c = H()
sage: c.is_central(certificate=True)
(True, A 7-dimensional polyhedron in QQ^7 defined
       as the convex hull of 1 vertex and 7 lines)
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=tuple(['x'+str(i) for i in␣
↪range(Integer(7))]))
>>> c = H()
>>> c.is_central(certificate=True)
(True, A 7-dimensional polyhedron in QQ^7 defined
       as the convex hull of 1 vertex and 7 lines)
```

is_essential()

Test whether the hyperplane arrangement is essential.

A hyperplane arrangement is essential if the span of the normals of its hyperplanes spans the ambient space.

See also

essentialization()

OUTPUT: boolean

EXAMPLES:

```
sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: H(x, x+1).is_essential()
False
sage: H(x, y).is_essential()
True
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = H._first_
↪ngens(2)
>>> H(x, x+Integer(1)).is_essential()
False
>>> H(x, y).is_essential()
True
```

is_formal()

Return if `self` is formal.

A hyperplane arrangement is *formal* if it is 3-generated [Yuz1993], where k-generated is defined in *minimal_generated_number()*.

EXAMPLES:

```
sage: P.<x,y,z> = HyperplaneArrangements(QQ)
sage: A = P(x, y, z, x+y+z, 2*x+y+z, 2*x+3*y+z, 2*x+3*y+4*z, 3*x+5*z,␣
↪3*x+4*y+5*z)
sage: B = P(x, y, z, x+y+z, 2*x+y+z, 2*x+3*y+z, 2*x+3*y+4*z, x+3*z, x+2*y+3*z)
sage: A.is_formal()
True
sage: B.is_formal()
False
```

```
>>> from sage.all import *
>>> P = HyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,) = P._
↪first_ngens(3)
>>> A = P(x, y, z, x+y+z, Integer(2)*x+y+z, Integer(2)*x+Integer(3)*y+z,␣
↪Integer(2)*x+Integer(3)*y+Integer(4)*z, Integer(3)*x+Integer(5)*z,␣
↪Integer(3)*x+Integer(4)*y+Integer(5)*z)
>>> B = P(x, y, z, x+y+z, Integer(2)*x+y+z, Integer(2)*x+Integer(3)*y+z,␣
↪Integer(2)*x+Integer(3)*y+Integer(4)*z, x+Integer(3)*z,␣
↪x+Integer(2)*y+Integer(3)*z)
>>> A.is_formal()
True
>>> B.is_formal()
False
```

is_free(*algorithm='singular'*)

Return if `self` is free.

A hyperplane arrangement A is free if the module of derivations $\mathrm{Der}(A)$ is a free S-module, where S is the corresponding symmetric space.

INPUT:

- `algorithm` – (default: `'singular'`) can be one of the following:
  - `'singular'` – use Singular's minimal free resolution
  - `'BC'` – use the algorithm given by Barakat and Cuntz in [BC2012] (much slower than using Singular)

ALGORITHM:

singular

Check that the minimal free resolution has length at most 2 by using Singular.

BC

This implementation follows [BC2012] by constructing a chain of free modules

$$D(A) = D(A_n) < D(A_{n-1}) < \cdots < D(A_1) < D(A_0)$$

corresponding to some ordering of the arrangements $A_0 \subset A_1 \subset \cdots \subset A_{n-1} \subset A_n = A$. Such a chain is found by using a backtracking algorithm.

EXAMPLES:

For type A arrangements, chordality is equivalent to freeness. We verify that in type A_3:

```
sage: W = WeylGroup(['A', 3], prefix='s')                                    #␣
↪needs sage.combinat sage.groups
sage: for x in W:                                                            #␣
↪needs sage.combinat sage.groups
....:     A = x.inversion_arrangement()
....:     assert A.matroid().is_chordal() == A.is_free()
```

```
>>> from sage.all import *
>>> W = WeylGroup(['A', Integer(3)], prefix='s')                             ␣
↪     # needs sage.combinat sage.groups
>>> for x in W:                                                              #␣
↪needs sage.combinat sage.groups
...     A = x.inversion_arrangement()
...     assert A.matroid().is_chordal() == A.is_free()
```

is_linear()

Test whether all hyperplanes pass through the origin.

OUTPUT: boolean

EXAMPLES:

```
sage: a = hyperplane_arrangements.semiorder(3)
sage: a.is_linear()
False
```

(continues on next page)

(continued from previous page)

```
sage: b = hyperplane_arrangements.braid(3)                                    #␣
↪needs sage.graphs
sage: b.is_linear()                                                           #␣
↪needs sage.graphs
True

sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: c = H(x+1, y+1)
sage: c.is_linear()
False
sage: c.is_central()
True
```

```
>>> from sage.all import *
>>> a = hyperplane_arrangements.semiorder(Integer(3))
>>> a.is_linear()
False
>>> b = hyperplane_arrangements.braid(Integer(3))                              ␣
↪     # needs sage.graphs
>>> b.is_linear()                                                           #␣
↪needs sage.graphs
True

>>> H = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = H._first_
↪ngens(2)
>>> c = H(x+Integer(1), y+Integer(1))
>>> c.is_linear()
False
>>> c.is_central()
True
```

is_separating_hyperplane(*region1*, *region2*, *hyperplane*)

Test whether the `hyperplane` separates the given regions.

INPUT:

- `region1`, `region2` – polyhedra or list/tuple/iterable of coordinates which are regions of the arrangement or an interior point of a region
- `hyperplane` – a hyperplane

OUTPUT: boolean; whether the hyperplane `hyperplane` separate the given regions

EXAMPLES:

```
sage: A.<x,y> = hyperplane_arrangements.coordinate(2)
sage: A.is_separating_hyperplane([1,1], [2,1], y)
False
sage: A.is_separating_hyperplane([1,1], [-1,1], x)
True
sage: r = A.region_containing_point([1,1])
sage: s = A.region_containing_point([-1,1])
sage: A.is_separating_hyperplane(r, s, x)
True
```

```
>>> from sage.all import *
>>> A = hyperplane_arrangements.coordinate(Integer(2), names=('x', 'y',)); (x,
↪ y,) = A._first_ngens(2)
>>> A.is_separating_hyperplane([Integer(1),Integer(1)], [Integer(2),
↪Integer(1)], y)
False
>>> A.is_separating_hyperplane([Integer(1),Integer(1)], [-Integer(1),
↪Integer(1)], x)
True
>>> r = A.region_containing_point([Integer(1),Integer(1)])
>>> s = A.region_containing_point([-Integer(1),Integer(1)])
>>> A.is_separating_hyperplane(r, s, x)
True
```

is_simplicial()

Test whether the arrangement is simplicial.

A region is simplicial if the normal vectors of its bounding hyperplanes are linearly independent. A hyperplane arrangement is said to be simplicial if every region is simplicial.

OUTPUT: boolean; whether the hyperplane arrangement is simplicial

EXAMPLES:

```
sage: H.<x,y,z> = HyperplaneArrangements(QQ)
sage: A = H([[0,1,1,1], [0,1,2,3]])
sage: A.is_simplicial()
True
sage: A = H([[0,1,1,1], [0,1,2,3], [0,1,3,2]])
sage: A.is_simplicial()
True
sage: A = H([[0,1,1,1], [0,1,2,3], [0,1,3,2], [0,2,1,3]])
sage: A.is_simplicial()
False
sage: hyperplane_arrangements.braid(3).is_simplicial()                    #␣
↪needs sage.graphs
True
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,) = H._
↪first_ngens(3)
>>> A = H([[Integer(0),Integer(1),Integer(1),Integer(1)], [Integer(0),
↪Integer(1),Integer(2),Integer(3)]])
>>> A.is_simplicial()
True
>>> A = H([[Integer(0),Integer(1),Integer(1),Integer(1)], [Integer(0),
↪Integer(1),Integer(2),Integer(3)], [Integer(0),Integer(1),Integer(3),
↪Integer(2)]])
>>> A.is_simplicial()
True
>>> A = H([[Integer(0),Integer(1),Integer(1),Integer(1)], [Integer(0),
↪Integer(1),Integer(2),Integer(3)], [Integer(0),Integer(1),Integer(3),
↪Integer(2)], [Integer(0),Integer(2),Integer(1),Integer(3)]])
>>> A.is_simplicial()
```

(continues on next page)

(continued from previous page)

```
False
>>> hyperplane_arrangements.braid(Integer(3)).is_simplicial()                    ␣
↪     # needs sage.graphs
True
```

matroid()

Return the matroid associated to `self`.

Let A denote a central hyperplane arrangement and n_H the normal vector of some hyperplane $H \in A$. We define a matroid M_A as the linear matroid spanned by $\{n_H | H \in A\}$. The matroid M_A is such that the lattice of flats of M is isomorphic to the intersection lattice of A (Proposition 3.6 in [Sta2007]).

EXAMPLES:

```
sage: P.<x,y,z> = HyperplaneArrangements(QQ)
sage: A = P(x, y, z, x+y+z, 2*x+y+z, 2*x+3*y+z, 2*x+3*y+4*z)
sage: M = A.matroid(); M
Linear matroid of rank 3 on 7 elements represented over the Rational Field
```

```
>>> from sage.all import *
>>> P = HyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,) = P._
↪first_ngens(3)
>>> A = P(x, y, z, x+y+z, Integer(2)*x+y+z, Integer(2)*x+Integer(3)*y+z,␣
↪Integer(2)*x+Integer(3)*y+Integer(4)*z)
>>> M = A.matroid(); M
Linear matroid of rank 3 on 7 elements represented over the Rational Field
```

We check the lattice of flats is isomorphic to the intersection lattice:

```
sage: f = sum([list(M.flats(i)) for i in range(M.rank() + 1)], [])
sage: PF = Poset([f, lambda x, y: x < y])                                        #␣
↪needs sage.combinat
sage: PF.is_isomorphic(A.intersection_poset())                                   #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> f = sum([list(M.flats(i)) for i in range(M.rank() + Integer(1))], [])
>>> PF = Poset([f, lambda x, y: x < y])                                         #␣
↪needs sage.combinat
>>> PF.is_isomorphic(A.intersection_poset())                                    #␣
↪needs sage.combinat
True
```

minimal_generated_number()

Return the minimum k such that `self` is k-generated.

Let A be a central hyperplane arrangement. Let W_k denote the solution space of the linear system corresponding to the linear dependencies among the hyperplanes of A of length at most k. We say A is k-*generated* if $\dim W_k = \operatorname{rank} A$.

Equivalently this says all dependencies forming the Orlik-Terao ideal are generated by at most k hyperplanes.

EXAMPLES:

We construct Example 2.2 from [Yuz1993]:

```
sage: P.<x,y,z> = HyperplaneArrangements(QQ)
sage: A = P(x, y, z, x+y+z, 2*x+y+z, 2*x+3*y+z, 2*x+3*y+4*z, 3*x+5*z,␣
↪3*x+4*y+5*z)
sage: B = P(x, y, z, x+y+z, 2*x+y+z, 2*x+3*y+z, 2*x+3*y+4*z, x+3*z, x+2*y+3*z)
sage: A.minimal_generated_number()
3
sage: B.minimal_generated_number()
4
```

```
>>> from sage.all import *
>>> P = HyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,) = P._
↪first_ngens(3)
>>> A = P(x, y, z, x+y+z, Integer(2)*x+y+z, Integer(2)*x+Integer(3)*y+z,␣
↪Integer(2)*x+Integer(3)*y+Integer(4)*z, Integer(3)*x+Integer(5)*z,␣
↪Integer(3)*x+Integer(4)*y+Integer(5)*z)
>>> B = P(x, y, z, x+y+z, Integer(2)*x+y+z, Integer(2)*x+Integer(3)*y+z,␣
↪Integer(2)*x+Integer(3)*y+Integer(4)*z, x+Integer(3)*z,␣
↪x+Integer(2)*y+Integer(3)*z)
>>> A.minimal_generated_number()
3
>>> B.minimal_generated_number()
4
```

n_bounded_regions()

Return the number of (relatively) bounded regions.

OUTPUT:

An integer. The number of relatively bounded regions of the hyperplane arrangement.

EXAMPLES:

```
sage: A = hyperplane_arrangements.semiorder(3)
sage: A.n_bounded_regions()
7
```

```
>>> from sage.all import *
>>> A = hyperplane_arrangements.semiorder(Integer(3))
>>> A.n_bounded_regions()
7
```

n_hyperplanes()

Return the number of hyperplanes in the arrangement.

OUTPUT: integer

EXAMPLES:

```
sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: A = H([1,1,0], [2,3,-1], [4,5,3])
sage: A.n_hyperplanes()
3
sage: len(A)     # equivalent
3
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = H._first_
↪ngens(2)
>>> A = H([Integer(1),Integer(1),Integer(0)], [Integer(2),Integer(3),-
↪Integer(1)], [Integer(4),Integer(5),Integer(3)])
>>> A.n_hyperplanes()
3
>>> len(A)     # equivalent
3
```

n_regions()

The number of regions of the hyperplane arrangement.

OUTPUT: integer

EXAMPLES:

```
sage: A = hyperplane_arrangements.semiorder(3)
sage: A.n_regions()
19
```

```
>>> from sage.all import *
>>> A = hyperplane_arrangements.semiorder(Integer(3))
>>> A.n_regions()
19
```

orlik_solomon_algebra(*base_ring=None*, *ordering=None*, ***kwds*)

Return the Orlik-Solomon algebra of `self`.

INPUT:

- `base_ring` – (default: the base field of `self`) the ring over which the Orlik-Solomon algebra will be defined
- `ordering` – (optional) an ordering of the ground set

EXAMPLES:

```
sage: P.<x,y,z> = HyperplaneArrangements(QQ)
sage: A = P(x, y, z, x+y+z, 2*x+y+z, 2*x+3*y+z, 2*x+3*y+4*z)
sage: A.orlik_solomon_algebra()
Orlik-Solomon algebra of Linear matroid of rank 3 on 7 elements
 represented over the Rational Field
sage: A.orlik_solomon_algebra(base_ring=ZZ)
Orlik-Solomon algebra of Linear matroid of rank 3 on 7 elements
 represented over the Rational Field
```

```
>>> from sage.all import *
>>> P = HyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,) = P._
↪first_ngens(3)
>>> A = P(x, y, z, x+y+z, Integer(2)*x+y+z, Integer(2)*x+Integer(3)*y+z,␣
↪Integer(2)*x+Integer(3)*y+Integer(4)*z)
>>> A.orlik_solomon_algebra()
Orlik-Solomon algebra of Linear matroid of rank 3 on 7 elements
 represented over the Rational Field
```

(continues on next page)

(continued from previous page)

```
>>> A.orlik_solomon_algebra(base_ring=ZZ)
Orlik-Solomon algebra of Linear matroid of rank 3 on 7 elements
 represented over the Rational Field
```

orlik_terao_algebra(*base_ring=None*, *ordering=None*, ***kwds*)

Return the Orlik-Terao algebra of `self`.

INPUT:

- `base_ring` – (default: the base field of `self`) the ring over which the Orlik-Terao algebra will be defined
- `ordering` – (optional) an ordering of the ground set

EXAMPLES:

```
sage: P.<x,y,z> = HyperplaneArrangements(QQ)
sage: A = P(x, y, z, x+y+z, 2*x+y+z, 2*x+3*y+z, 2*x+3*y+4*z)
sage: A.orlik_terao_algebra()
Orlik-Terao algebra of Linear matroid of rank 3 on 7 elements
 represented over the Rational Field over Rational Field
sage: A.orlik_terao_algebra(base_ring=QQ['t'])
Orlik-Terao algebra of Linear matroid of rank 3 on 7 elements
 represented over the Rational Field
 over Univariate Polynomial Ring in t over Rational Field
```

```
>>> from sage.all import *
>>> P = HyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,) = P._
↪first_ngens(3)
>>> A = P(x, y, z, x+y+z, Integer(2)*x+y+z, Integer(2)*x+Integer(3)*y+z,␣
↪Integer(2)*x+Integer(3)*y+Integer(4)*z)
>>> A.orlik_terao_algebra()
Orlik-Terao algebra of Linear matroid of rank 3 on 7 elements
 represented over the Rational Field over Rational Field
>>> A.orlik_terao_algebra(base_ring=QQ['t'])
Orlik-Terao algebra of Linear matroid of rank 3 on 7 elements
 represented over the Rational Field
 over Univariate Polynomial Ring in t over Rational Field
```

plot(***kwds*)

Plot the hyperplane arrangement.

OUTPUT: a graphics object

EXAMPLES:

```
sage: L.<x, y> = HyperplaneArrangements(QQ)
sage: L(x, y, x+y-2).plot()                                                #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
```

```
>>> from sage.all import *
>>> L = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = L._first_
↪ngens(2)
```

(continues on next page)

(continued from previous page)

```
>>> L(x, y, x+y-Integer(2)).plot()                                          ␣
↪     # needs sage.plot
Graphics object consisting of 3 graphics primitives
```

poincare_polynomial()

Return the Poincaré polynomial of the hyperplane arrangement.

OUTPUT: the Poincaré polynomial in $\mathbf{Q}[x]$

EXAMPLES:

```
sage: a = hyperplane_arrangements.coordinate(2)
sage: a.poincare_polynomial()
x^2 + 2*x + 1
```

```
>>> from sage.all import *
>>> a = hyperplane_arrangements.coordinate(Integer(2))
>>> a.poincare_polynomial()
x^2 + 2*x + 1
```

poset_of_regions(*B=None*, *numbered_labels=True*)

Return the poset of regions for a central hyperplane arrangement.

The poset of regions is a partial order on the set of regions where the regions are ordered by $R \leq R'$ if and only if $S(R) \subseteq S(R')$ where $S(R)$ is the set of hyperplanes which separate the region R from the base region B.

INPUT:

- `B` – a region (optional); if `None`, then an arbitrary region is chosen as the base region
- `numbered_labels` – boolean (default: `True`); if `True`, then the elements of the poset are numbered. Else they are labelled with the regions themselves.

OUTPUT: a Poset object containing the poset of regions

EXAMPLES:

```
sage: H.<x,y,z> = HyperplaneArrangements(QQ)
sage: A = H([[0,1,1,1], [0,1,2,3]])
sage: A.poset_of_regions()                                                 #␣
↪needs sage.combinat
Finite poset containing 4 elements

sage: # needs sage.combinat sage.graphs
sage: A = hyperplane_arrangements.braid(3)
sage: A.poset_of_regions()
Finite poset containing 6 elements
sage: A.poset_of_regions(numbered_labels=False)
Finite poset containing 6 elements
sage: A = hyperplane_arrangements.braid(4)
sage: A.poset_of_regions()
Finite poset containing 24 elements

sage: H.<x,y,z> = HyperplaneArrangements(QQ)
sage: A = H([[0,1,1,1], [0,1,2,3], [0,1,3,2], [0,2,1,3]])
```

(continues on next page)

(continued from previous page)

```
sage: R = A.regions()
sage: base_region = R[3]
sage: A.poset_of_regions(B=base_region)                                      #␣
↪needs sage.combinat
Finite poset containing 14 elements
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,) = H._
↪first_ngens(3)
>>> A = H([[Integer(0),Integer(1),Integer(1),Integer(1)], [Integer(0),
↪Integer(1),Integer(2),Integer(3)]])
>>> A.poset_of_regions()                                                    #␣
↪needs sage.combinat
Finite poset containing 4 elements

>>> # needs sage.combinat sage.graphs
>>> A = hyperplane_arrangements.braid(Integer(3))
>>> A.poset_of_regions()
Finite poset containing 6 elements
>>> A.poset_of_regions(numbered_labels=False)
Finite poset containing 6 elements
>>> A = hyperplane_arrangements.braid(Integer(4))
>>> A.poset_of_regions()
Finite poset containing 24 elements

>>> H = HyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,) = H._
↪first_ngens(3)
>>> A = H([[Integer(0),Integer(1),Integer(1),Integer(1)], [Integer(0),
↪Integer(1),Integer(2),Integer(3)], [Integer(0),Integer(1),Integer(3),
↪Integer(2)], [Integer(0),Integer(2),Integer(1),Integer(3)]])
>>> R = A.regions()
>>> base_region = R[Integer(3)]
>>> A.poset_of_regions(B=base_region)                                       #␣
↪needs sage.combinat
Finite poset containing 14 elements
```

primitive_eulerian_polynomial()

Return the primitive Eulerian polynomial of `self`.

The primitive Eulerian polynomial of a hyperplane arrangement A is defined [BHS2023] by

$$P_A(z) := \sum_{X \in L} |\mu(B, X)|(z-1)^{\operatorname{codim} X},$$

where L is the intersection poset of A, B is the minimal element of L (here, the 0 dimensional subspace), and μ is the Möbius function of L.

OUTPUT: the primitive Eulerian polynomial in $\mathbf{Z}[z]$

EXAMPLES:

```
sage: A = hyperplane_arrangements.coordinate(2)
sage: A.primitive_eulerian_polynomial()                                      #␣
↪needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
z^2
sage: B = hyperplane_arrangements.braid(3)
sage: B.primitive_eulerian_polynomial()                                     #␣
↪needs sage.graphs
z^2 + z

sage: H = hyperplane_arrangements.Shi(['B',2]).cone()
sage: H.is_simplicial()
False
sage: H.primitive_eulerian_polynomial()                                     #␣
↪needs sage.graphs
z^3 + 11*z^2 + 4*z

sage: H = hyperplane_arrangements.graphical(graphs.CycleGraph(4))
sage: H.primitive_eulerian_polynomial()                                     #␣
↪needs sage.graphs
z^3 + 3*z^2 - z
```

```
>>> from sage.all import *
>>> A = hyperplane_arrangements.coordinate(Integer(2))
>>> A.primitive_eulerian_polynomial()                                     #␣
↪needs sage.graphs
z^2
>>> B = hyperplane_arrangements.braid(Integer(3))
>>> B.primitive_eulerian_polynomial()                                     #␣
↪needs sage.graphs
z^2 + z

>>> H = hyperplane_arrangements.Shi(['B',Integer(2)]).cone()
>>> H.is_simplicial()
False
>>> H.primitive_eulerian_polynomial()                                     #␣
↪needs sage.graphs
z^3 + 11*z^2 + 4*z

>>> H = hyperplane_arrangements.graphical(graphs.CycleGraph(Integer(4)))
>>> H.primitive_eulerian_polynomial()                                     #␣
↪needs sage.graphs
z^3 + 3*z^2 - z
```

We verify Example 2.4 in [BHS2023] for $k = 2, 3, 4, 5$:

```
sage: R.<x,y> = HyperplaneArrangements(QQ)
sage: for k in range(2,6):                                                  #␣
↪needs sage.graphs
....:     H = R([x+j*y for j in range(k)])
....:     H.primitive_eulerian_polynomial()
z^2
z^2 + z
z^2 + 2*z
z^2 + 3*z
```

```
>>> from sage.all import *
>>> R = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = R._first_
↪ngens(2)
>>> for k in range(Integer(2),Integer(6)):                                   ␣
↪             # needs sage.graphs
...     H = R([x+j*y for j in range(k)])
...     H.primitive_eulerian_polynomial()
z^2
z^2 + z
z^2 + 2*z
z^2 + 3*z
```

We verify Equation (4) in [BHS2023] on some examples:

```
sage: # needs sage.graphs
sage: R.<x> = ZZ[]
sage: Arr = [hyperplane_arrangements.braid(n) for n in range(2,6)]
sage: all(R(A.cocharacteristic_polynomial()(1/(x-1)) * (x-1)^A.dimension())
....:     == R(A.primitive_eulerian_polynomial()) for A in Arr)
True
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> R = ZZ['x']; (x,) = R._first_ngens(1)
>>> Arr = [hyperplane_arrangements.braid(n) for n in range(Integer(2),
↪Integer(6))]
>>> all(R(A.cocharacteristic_polynomial()(Integer(1)/(x-Integer(1))) * (x-
↪Integer(1))**A.dimension())
...     == R(A.primitive_eulerian_polynomial()) for A in Arr)
True
```

We compute types H_3 and F_4 in Table 1 of [BHS2023]:

```
sage: # needs sage.libs.gap
sage: W = CoxeterGroup(['H',3], implementation='matrix')
sage: A = HyperplaneArrangements(W.base_ring(), tuple(f'x{s}' for s in␣
↪range(W.rank())))
sage: H = A([[0] + list(r) for r in W.positive_roots()])
sage: H.is_simplicial()                                                    #␣
↪needs sage.graphs
True
sage: H.primitive_eulerian_polynomial()
z^3 + 28*z^2 + 16*z

sage: W = CoxeterGroup(['F',4], implementation='permutation')
sage: A = HyperplaneArrangements(QQ, tuple(f'x{s}' for s in range(W.rank())))
sage: H = A([[0] + list(r) for r in W.positive_roots()])
sage: H.primitive_eulerian_polynomial()      # long time                   #␣
↪needs sage.graphs
z^4 + 116*z^3 + 220*z^2 + 48*z
```

```
>>> from sage.all import *
>>> # needs sage.libs.gap
```

(continues on next page)

(continued from previous page)

```
>>> W = CoxeterGroup(['H',Integer(3)], implementation='matrix')
>>> A = HyperplaneArrangements(W.base_ring(), tuple(f'x{s}' for s in range(W.
↪rank())))
>>> H = A([[Integer(0)] + list(r) for r in W.positive_roots()])
>>> H.is_simplicial()                                                    #␣
↪needs sage.graphs
True
>>> H.primitive_eulerian_polynomial()
z^3 + 28*z^2 + 16*z

>>> W = CoxeterGroup(['F',Integer(4)], implementation='permutation')
>>> A = HyperplaneArrangements(QQ, tuple(f'x{s}' for s in range(W.rank())))
>>> H = A([[Integer(0)] + list(r) for r in W.positive_roots()])
>>> H.primitive_eulerian_polynomial()      # long time                   #␣
↪needs sage.graphs
z^4 + 116*z^3 + 220*z^2 + 48*z
```

We verify Proposition 2.5 in [BHS2023] on the braid arrangement B_k for $k = 2, 3, 4, 5$:

```
sage: B = [hyperplane_arrangements.braid(k) for k in range(2,6)]
sage: all(H.is_simplicial() for H in B)
True
sage: all(c > 0 for H in B                                                 #␣
↪needs sage.graphs
....:     for c in H.primitive_eulerian_polynomial().coefficients())
True
```

```
>>> from sage.all import *
>>> B = [hyperplane_arrangements.braid(k) for k in range(Integer(2),
↪Integer(6))]
>>> all(H.is_simplicial() for H in B)
True
>>> all(c > Integer(0) for H in B                                          ␣
↪      # needs sage.graphs
...     for c in H.primitive_eulerian_polynomial().coefficients())
True
```

We verify Example 9.4 in [BHS2023] showing a hyperplane arrangement whose primitive Eulerian polynomial does not have real roots (in general, the graphical arrangement of a cycle graph corresponds to the arrangements in Example 9.4):

```
sage: # needs sage.graphs
sage: H = hyperplane_arrangements.graphical(graphs.CycleGraph(5))
sage: pep = H.primitive_eulerian_polynomial(); pep
z^4 + 6*z^3 - 4*z^2 + z
sage: pep.roots(QQbar)
[(-6.626418492719221?, 1),
 (0, 1),
 (0.3132092463596102? - 0.2298065541510677?*I, 1),
 (0.3132092463596102? + 0.2298065541510677?*I, 1)]
sage: pep.roots(AA)
[(-6.626418492719221?, 1), (0, 1)]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> H = hyperplane_arrangements.graphical(graphs.CycleGraph(Integer(5)))
>>> pep = H.primitive_eulerian_polynomial(); pep
z^4 + 6*z^3 - 4*z^2 + z
>>> pep.roots(QQbar)
[(-6.626418492719221?, 1),
 (0, 1),
 (0.3132092463596102? - 0.2298065541510677?*I, 1),
 (0.3132092463596102? + 0.2298065541510677?*I, 1)]
>>> pep.roots(AA)
[(-6.626418492719221?, 1), (0, 1)]
```

rank()

Return the rank.

OUTPUT:

The dimension of the span of the normals to the hyperplanes in the arrangement.

EXAMPLES:

```
sage: H.<x,y,z> = HyperplaneArrangements(QQ)
sage: A = H([[0, 1, 2, 3],[-3, 4, 5, 6]])
sage: A.dimension()
3
sage: A.rank()
2

sage: # needs sage.graphs
sage: B = hyperplane_arrangements.braid(3)
sage: B.hyperplanes()
(Hyperplane 0*t0 + t1 - t2 + 0,
 Hyperplane t0 - t1 + 0*t2 + 0,
 Hyperplane t0 + 0*t1 - t2 + 0)
sage: B.dimension()
3
sage: B.rank()
2

sage: p = polytopes.simplex(5, project=True)
sage: H = p.hyperplane_arrangement()
sage: H.rank()
5
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,) = H._
↪first_ngens(3)
>>> A = H([[Integer(0), Integer(1), Integer(2), Integer(3)],[-Integer(3),␣
↪Integer(4), Integer(5), Integer(6)]])
>>> A.dimension()
3
>>> A.rank()
2
```

(continues on next page)

(continued from previous page)

```
>>> # needs sage.graphs
>>> B = hyperplane_arrangements.braid(Integer(3))
>>> B.hyperplanes()
(Hyperplane 0*t0 + t1 - t2 + 0,
 Hyperplane t0 - t1 + 0*t2 + 0,
 Hyperplane t0 + 0*t1 - t2 + 0)
>>> B.dimension()
3
>>> B.rank()
2

>>> p = polytopes.simplex(Integer(5), project=True)
>>> H = p.hyperplane_arrangement()
>>> H.rank()
5
```

region_containing_point(*p*)

The region in the hyperplane arrangement containing a given point.

The base field must have characteristic zero.

INPUT:

- `p` – point

OUTPUT:

A polyhedron. A `ValueError` is raised if the point is not interior to a region, that is, sits on a hyperplane.

EXAMPLES:

```
sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: A = H([(1,0), 0], [(0,1), 1], [(0,1), -1], [(1,-1), 0], [(1,1), 0])
sage: A.region_containing_point([1,2])
A 2-dimensional polyhedron in QQ^2 defined
as the convex hull of 2 vertices and 2 rays
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = H._first_
↪ngens(2)
>>> A = H([(Integer(1),Integer(0)), Integer(0)], [(Integer(0),Integer(1)),␣
↪Integer(1)], [(Integer(0),Integer(1)), -Integer(1)], [(Integer(1),-
↪Integer(1)), Integer(0)], [(Integer(1),Integer(1)), Integer(0)])
>>> A.region_containing_point([Integer(1),Integer(2)])
A 2-dimensional polyhedron in QQ^2 defined
as the convex hull of 2 vertices and 2 rays
```

regions()

Return the regions of the hyperplane arrangement.

The base field must have characteristic zero.

OUTPUT: a tuple containing the regions as polyhedra

The regions are the connected components of the complement of the union of the hyperplanes as a subset of $\mathbf{R}^n$.

EXAMPLES:

```
sage: a = hyperplane_arrangements.braid(2)                                    #␣
↪needs sage.graphs
sage: a.regions()                                                             #␣
↪needs sage.graphs
(A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 1 vertex, 1 ray, 1 line,
 A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 1 vertex, 1 ray, 1 line)

sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: A = H(x, y+1)
sage: A.regions()
(A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 1 vertex and 2 rays)

sage: chessboard = []
sage: N = 8
sage: for x0 in range(N + 1):
....:     for y0 in range(N + 1):
....:         chessboard.extend([x-x0, y-y0])
sage: chessboard = H(chessboard)
sage: len(chessboard.bounded_regions())    # long time, 359 ms on a Core i7
64
```

```
>>> from sage.all import *
>>> a = hyperplane_arrangements.braid(Integer(2))                              ␣
↪     # needs sage.graphs
>>> a.regions()                                                              #␣
↪needs sage.graphs
(A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 1 vertex, 1 ray, 1 line,
 A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 1 vertex, 1 ray, 1 line)

>>> H = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = H._first_
↪ngens(2)
>>> A = H(x, y+Integer(1))
>>> A.regions()
(A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in QQ^2 defined
```

(continues on next page)

(continued from previous page)

```
    as the convex hull of 1 vertex and 2 rays)

>>> chessboard = []
>>> N = Integer(8)
>>> for x0 in range(N + Integer(1)):
...     for y0 in range(N + Integer(1)):
...         chessboard.extend([x-x0, y-y0])
>>> chessboard = H(chessboard)
>>> len(chessboard.bounded_regions())    # long time, 359 ms on a Core i7
64
```

Example 6 of [KP2020]:

```
sage: from itertools import product
sage: def zero_one(d):
....:     for x in product([0,1], repeat=d):
....:         if any(x):
....:             yield [0] + list(x)

sage: K.<x,y> = HyperplaneArrangements(QQ)
sage: A = K(*zero_one(2))
sage: len(A.regions())
6
sage: K.<x,y,z> = HyperplaneArrangements(QQ)
sage: A = K(*zero_one(3))
sage: len(A.regions())
32
sage: K.<x,y,z,w> = HyperplaneArrangements(QQ)
sage: A = K(*zero_one(4))
sage: len(A.regions())
370
sage: K.<x,y,z,w,r> = HyperplaneArrangements(QQ)
sage: A = K(*zero_one(5))
sage: len(A.regions())             # not tested (~25s)
11292
```

```
>>> from sage.all import *
>>> from itertools import product
>>> def zero_one(d):
...     for x in product([Integer(0),Integer(1)], repeat=d):
...         if any(x):
...             yield [Integer(0)] + list(x)

>>> K = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = K._first_
↪ngens(2)
>>> A = K(*zero_one(Integer(2)))
>>> len(A.regions())
6
>>> K = HyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,) = K._
↪first_ngens(3)
>>> A = K(*zero_one(Integer(3)))
>>> len(A.regions())
```

(continues on next page)

(continued from previous page)

```
32
>>> K = HyperplaneArrangements(QQ, names=('x', 'y', 'z', 'w',)); (x, y, z, w,
↪) = K._first_ngens(4)
>>> A = K(*zero_one(Integer(4)))
>>> len(A.regions())
370
>>> K = HyperplaneArrangements(QQ, names=('x', 'y', 'z', 'w', 'r',)); (x, y,␣
↪z, w, r,) = K._first_ngens(5)
>>> A = K(*zero_one(Integer(5)))
>>> len(A.regions())            # not tested (~25s)
11292
```

It is possible to specify the backend:

```
sage: # needs sage.rings.number_field
sage: K.<q> = CyclotomicField(9)
sage: L.<r9> = NumberField((q + q**(-1)).minpoly(),
....:                       embedding=AA(q + q**-1))
sage: norms = [[1, 1/3*(-2*r9**2-r9+1), 0],
....:          [1, -r9**2 - r9, 0],
....:          [1, -r9**2 + 1, 0],
....:          [1, -r9**2, 0],
....:          [1, r9**2 - 4, -r9**2+3]]
sage: H.<x,y,z> = HyperplaneArrangements(L)
sage: A = H(backend='normaliz')
sage: for v in norms:
....:     a,b,c = v
....:     A = A.add_hyperplane(a*x + b*y + c*z)
sage: R = A.regions()                                       # optional -␣
↪pynormaliz
sage: R[0].backend()                                        # optional -␣
↪pynormaliz
'normaliz'
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> K = CyclotomicField(Integer(9), names=('q',)); (q,) = K._first_ngens(1)
>>> L = NumberField((q + q**(-Integer(1))).minpoly(),
...                     embedding=AA(q + q**-Integer(1)), names=('r9',));␣
↪(r9,) = L._first_ngens(1)
>>> norms = [[Integer(1), Integer(1)/Integer(3)*(-Integer(2)*r9**Integer(2)-
↪r9+Integer(1)), Integer(0)],
...           [Integer(1), -r9**Integer(2) - r9, Integer(0)],
...           [Integer(1), -r9**Integer(2) + Integer(1), Integer(0)],
...           [Integer(1), -r9**Integer(2), Integer(0)],
...           [Integer(1), r9**Integer(2) - Integer(4), -
↪r9**Integer(2)+Integer(3)]]
>>> H = HyperplaneArrangements(L, names=('x', 'y', 'z',)); (x, y, z,) = H._
↪first_ngens(3)
>>> A = H(backend='normaliz')
>>> for v in norms:
...     a,b,c = v
```

(continues on next page)

(continued from previous page)

```
...     A = A.add_hyperplane(a*x + b*y + c*z)
>>> R = A.regions()                                    # optional -␣
↪pynormaliz
>>> R[Integer(0)].backend()                                # optional␣
↪- pynormaliz
'normaliz'
```

restriction(*hyperplane*, *repetitions=False*)

Return the restriction to a hyperplane.

INPUT:

- `hyperplane` – a hyperplane of the hyperplane arrangement
- `repetitions` – boolean (default: `False`); eliminate repetitions for ordered arrangements

OUTPUT:

The restriction $\mathcal{A}_H$ of the hyperplane arrangement $\mathcal{A}$ to the given `hyperplane` H.

EXAMPLES:

```
sage: # needs sage.graphs
sage: A.<u,x,y,z> = hyperplane_arrangements.braid(4);  A
Arrangement of 6 hyperplanes of dimension 4 and rank 3
sage: H = A[0];  H
Hyperplane 0*u + 0*x + y - z + 0
sage: R = A.restriction(H); R
Arrangement <x - z | u - x | u - z>
sage: A.add_hyperplane(z).restriction(z)
Arrangement of 6 hyperplanes of dimension 3 and rank 3
sage: A.add_hyperplane(u).restriction(u)
Arrangement of 6 hyperplanes of dimension 3 and rank 3
sage: D = A.deletion(H);  D
Arrangement of 5 hyperplanes of dimension 4 and rank 3
sage: ca = A.characteristic_polynomial()
sage: cr = R.characteristic_polynomial()
sage: cd = D.characteristic_polynomial()
sage: ca
x^4 - 6*x^3 + 11*x^2 - 6*x
sage: cd - cr
x^4 - 6*x^3 + 11*x^2 - 6*x
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> A = hyperplane_arrangements.braid(Integer(4), names=('u', 'x', 'y', 'z',
↪)); (u, x, y, z,) = A._first_ngens(4);  A
Arrangement of 6 hyperplanes of dimension 4 and rank 3
>>> H = A[Integer(0)];  H
Hyperplane 0*u + 0*x + y - z + 0
>>> R = A.restriction(H); R
Arrangement <x - z | u - x | u - z>
>>> A.add_hyperplane(z).restriction(z)
Arrangement of 6 hyperplanes of dimension 3 and rank 3
>>> A.add_hyperplane(u).restriction(u)
```

(continues on next page)

(continued from previous page)

```
Arrangement of 6 hyperplanes of dimension 3 and rank 3
>>> D = A.deletion(H);  D
Arrangement of 5 hyperplanes of dimension 4 and rank 3
>>> ca = A.characteristic_polynomial()
>>> cr = R.characteristic_polynomial()
>>> cd = D.characteristic_polynomial()
>>> ca
x^4 - 6*x^3 + 11*x^2 - 6*x
>>> cd - cr
x^4 - 6*x^3 + 11*x^2 - 6*x
```

See also

deletion()

sign_vector(*p*)

Indicates on which side of each hyperplane the given point p lies.

The base field must have characteristic zero.

INPUT:

- `p` – point as a list/tuple/iterable

OUTPUT:

A vector whose entries are in $[-1, 0, +1]$.

EXAMPLES:

```
sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: A = H([(1,0), 0], [(0,1), 1]);  A
Arrangement <y + 1 | x>
sage: A.sign_vector([2, -2])
(-1, 1)
sage: A.sign_vector((-1, -1))
(0, -1)
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = H._first_
↪ngens(2)
>>> A = H([(Integer(1),Integer(0)), Integer(0)], [(Integer(0),Integer(1)),␣
↪Integer(1)]);  A
Arrangement <y + 1 | x>
>>> A.sign_vector([Integer(2), -Integer(2)])
(-1, 1)
>>> A.sign_vector((-Integer(1), -Integer(1)))
(0, -1)
```

unbounded_regions()

Return the relatively bounded regions of the arrangement.

OUTPUT:

Tuple of polyhedra. The regions of the arrangement that are not relatively bounded. It is assumed that the arrangement is defined over the rationals.

See also

bounded_regions()

EXAMPLES:

```
sage: # needs sage.combinat
sage: A = hyperplane_arrangements.semiorder(3)
sage: B = A.essentialization()
sage: B.n_regions() - B.n_bounded_regions()
12
sage: B.unbounded_regions()
(A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 3 vertices and 1 ray,
 A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 3 vertices and 1 ray,
 A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 3 vertices and 1 ray,
 A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 3 vertices and 1 ray,
 A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in QQ^2 defined
    as the convex hull of 3 vertices and 1 ray,
 A 2-dimensional polyhedron in QQ^2 defined
    as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 3 vertices and 1 ray,
 A 2-dimensional polyhedron in QQ^2 defined
    as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 1 vertex and 2 rays)
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> A = hyperplane_arrangements.semiorder(Integer(3))
>>> B = A.essentialization()
>>> B.n_regions() - B.n_bounded_regions()
12
>>> B.unbounded_regions()
(A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 3 vertices and 1 ray,
 A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 3 vertices and 1 ray,
 A 2-dimensional polyhedron in QQ^2 defined
```

(continues on next page)

(continued from previous page)

```
     as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 3 vertices and 1 ray,
 A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 3 vertices and 1 ray,
 A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in QQ^2 defined
    as the convex hull of 3 vertices and 1 ray,
 A 2-dimensional polyhedron in QQ^2 defined
    as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 3 vertices and 1 ray,
 A 2-dimensional polyhedron in QQ^2 defined
    as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in QQ^2 defined
     as the convex hull of 1 vertex and 2 rays)
```

union(*other*)

The union of `self` with `other`.

INPUT:

- `other` – a hyperplane arrangement or something that can be converted into a hyperplane arrangement

OUTPUT: a new hyperplane arrangement

EXAMPLES:

```
sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: A = H([1,2,3], [0,1,1], [0,1,-1], [1,-1,0], [1,1,0])
sage: B = H([1,1,1], [1,-1,1], [1,0,-1])
sage: C = A.union(B); C
Arrangement of 8 hyperplanes of dimension 2 and rank 2
sage: C == A | B   # syntactic sugar
True
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = H._first_
↪ngens(2)
>>> A = H([Integer(1),Integer(2),Integer(3)], [Integer(0),Integer(1),
↪Integer(1)], [Integer(0),Integer(1),-Integer(1)], [Integer(1),-Integer(1),
↪Integer(0)], [Integer(1),Integer(1),Integer(0)])
>>> B = H([Integer(1),Integer(1),Integer(1)], [Integer(1),-Integer(1),
↪Integer(1)], [Integer(1),Integer(0),-Integer(1)])
>>> C = A.union(B); C
Arrangement of 8 hyperplanes of dimension 2 and rank 2
>>> C == A | B   # syntactic sugar
True
```

A single hyperplane is coerced into a hyperplane arrangement if necessary:

```
sage: A.union(x+y-1)
Arrangement of 6 hyperplanes of dimension 2 and rank 2
sage: A.add_hyperplane(x+y-1)     # alias
Arrangement of 6 hyperplanes of dimension 2 and rank 2

sage: P.<x,y> = HyperplaneArrangements(RR)
sage: C = P(2*x + 4*y + 5)
sage: C.union(A)
Arrangement of 6 hyperplanes of dimension 2 and rank 2
```

```
>>> from sage.all import *
>>> A.union(x+y-Integer(1))
Arrangement of 6 hyperplanes of dimension 2 and rank 2
>>> A.add_hyperplane(x+y-Integer(1))     # alias
Arrangement of 6 hyperplanes of dimension 2 and rank 2

>>> P = HyperplaneArrangements(RR, names=('x', 'y',)); (x, y,) = P._first_
↪ngens(2)
>>> C = P(Integer(2)*x + Integer(4)*y + Integer(5))
>>> C.union(A)
Arrangement of 6 hyperplanes of dimension 2 and rank 2
```

varchenko_matrix(*names='h'*)

Return the Varchenko matrix of the arrangement.

Let $H_1, \dots, H_s$ and $R_1, \dots, R_t$ denote the hyperplanes and regions, respectively, of the arrangement. Let $S = \mathbf{Q}[h_1, \dots, h_s]$, a polynomial ring with indeterminate h_i corresponding to hyperplane H_i. The Varchenko matrix is the $t \times t$ matrix with i, j-th entry the product of those h_k such that H_k separates R_i and R_j.

INPUT:

- `names` – string or list/tuple/iterable of strings. The variable names for the polynomial ring S

OUTPUT: the Varchenko matrix

EXAMPLES:

```
sage: a = hyperplane_arrangements.coordinate(3)
sage: v = a.varchenko_matrix();  v
[       1       h2       h1    h1*h2 h0*h1*h2    h0*h1    h0*h2       h0]
[      h2        1    h1*h2       h1    h0*h1 h0*h1*h2       h0    h0*h2]
[      h1    h1*h2        1       h2    h0*h2       h0 h0*h1*h2    h0*h1]
[   h1*h2       h1       h2        1       h0    h0*h2    h0*h1 h0*h1*h2]
[h0*h1*h2    h0*h1    h0*h2       h0        1       h2       h1    h1*h2]
[   h0*h1 h0*h1*h2       h0    h0*h2       h2        1    h1*h2       h1]
[   h0*h2       h0 h0*h1*h2    h0*h1       h1    h1*h2        1       h2]
[      h0    h0*h2    h0*h1 h0*h1*h2    h1*h2       h1       h2        1]
sage: factor(det(v))
(h2 - 1)^4 * (h2 + 1)^4 * (h1 - 1)^4 * (h1 + 1)^4 * (h0 - 1)^4 * (h0 + 1)^4
```

```
>>> from sage.all import *
>>> a = hyperplane_arrangements.coordinate(Integer(3))
>>> v = a.varchenko_matrix();  v
[       1       h2       h1    h1*h2 h0*h1*h2    h0*h1    h0*h2       h0]
[      h2        1    h1*h2       h1    h0*h1 h0*h1*h2       h0    h0*h2]
```

(continues on next page)

(continued from previous page)

```
[      h1     h1*h2         1        h2     h0*h2        h0  h0*h1*h2     h0*h1]
[   h1*h2        h1        h2         1        h0     h0*h2     h0*h1  h0*h1*h2]
[h0*h1*h2     h0*h1     h0*h2        h0         1        h2        h1     h1*h2]
[   h0*h1  h0*h1*h2        h0     h0*h2        h2         1     h1*h2        h1]
[   h0*h2        h0  h0*h1*h2     h0*h1        h1     h1*h2         1        h2]
[      h0     h0*h2     h0*h1  h0*h1*h2     h1*h2        h1        h2         1]
>>> factor(det(v))
(h2 - 1)^4 * (h2 + 1)^4 * (h1 - 1)^4 * (h1 + 1)^4 * (h0 - 1)^4 * (h0 + 1)^4
```

vertices(*exclude_sandwiched=False*)

Return the vertices.

The vertices are the zero-dimensional faces, see `face_vector()`.

INPUT:

- `exclude_sandwiched` – boolean (default: `False`). Whether to exclude hyperplanes that are sandwiched between parallel hyperplanes. Useful if you only need the convex hull.

OUTPUT:

The vertices in a sorted tuple. Each vertex is returned as a vector in the ambient vector space.

EXAMPLES:

```
sage: # needs sage.combinat
sage: A = hyperplane_arrangements.Shi(3).essentialization()
sage: A.dimension()
2
sage: A.face_vector()
(6, 21, 16)
sage: A.vertices()
((-2/3, 1/3), (-1/3, -1/3), (0, -1), (0, 0), (1/3, -2/3), (2/3, -1/3))
sage: point2d(A.vertices(), size=20) + A.plot()                              #␣
↪needs sage.plot
Graphics object consisting of 7 graphics primitives

sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: chessboard = []
sage: N = 8
sage: for x0 in range(N + 1):
....:     for y0 in range(N + 1):
....:         chessboard.extend([x-x0, y-y0])
sage: chessboard = H(chessboard)
sage: len(chessboard.vertices())
81
sage: chessboard.vertices(exclude_sandwiched=True)
((0, 0), (0, 8), (8, 0), (8, 8))
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> A = hyperplane_arrangements.Shi(Integer(3)).essentialization()
>>> A.dimension()
2
>>> A.face_vector()
```

(continues on next page)

(continued from previous page)

```
(6, 21, 16)
>>> A.vertices()
((-2/3, 1/3), (-1/3, -1/3), (0, -1), (0, 0), (1/3, -2/3), (2/3, -1/3))
>>> point2d(A.vertices(), size=Integer(20)) + A.plot()                    ␣
↪     # needs sage.plot
Graphics object consisting of 7 graphics primitives

>>> H = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = H._first_
↪ngens(2)
>>> chessboard = []
>>> N = Integer(8)
>>> for x0 in range(N + Integer(1)):
...     for y0 in range(N + Integer(1)):
...         chessboard.extend([x-x0, y-y0])
>>> chessboard = H(chessboard)
>>> len(chessboard.vertices())
81
>>> chessboard.vertices(exclude_sandwiched=True)
((0, 0), (0, 8), (8, 0), (8, 8))
```

whitney_data()

Return the Whitney numbers.

See also

whitney_number(), *doubly_indexed_whitney_number()*

OUTPUT:

A pair of integer matrices. The two matrices are the doubly-indexed Whitney numbers of the first or second kind, respectively. The i, j-th entry is the i, j-th doubly-indexed Whitney number.

EXAMPLES:

```
sage: # needs sage.combinat
sage: A = hyperplane_arrangements.Shi(3)
sage: A.whitney_data()
(
[  1  -6   9]  [ 1  6  6]
[  0   6 -15]  [ 0  6 15]
[  0   0   6], [ 0  0  6]
)
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> A = hyperplane_arrangements.Shi(Integer(3))
>>> A.whitney_data()
(
[  1  -6   9]  [ 1  6  6]
[  0   6 -15]  [ 0  6 15]
[  0   0   6], [ 0  0  6]
)
```

whitney_number(*k*, *kind=1*)

Return the k-th Whitney number.

If kind=1, this number is obtained by summing the Möbius function values $mu(0, x)$ over all x in the intersection poset with $\mathrm{rank}(x) = k$.

If kind=2, this number is the number of elements x, y in the intersection poset such that $x \leq y$ with ranks i and j, respectively.

See [GZ1983] for more details.

INPUT:

- k – integer
- kind – 1 or 2 (default: 1)

OUTPUT:

Integer. The k-th Whitney number.

See also

doubly_indexed_whitney_number() whitney_data()

EXAMPLES:

```
sage: # needs sage.combinat
sage: A = hyperplane_arrangements.Shi(3)
sage: A.whitney_number(0)
1
sage: A.whitney_number(1)
-6
sage: A.whitney_number(2)
9
sage: A.characteristic_polynomial()
x^3 - 6*x^2 + 9*x
sage: A.whitney_number(1, kind=2)
6
sage: p = A.intersection_poset()
sage: r = p.rank_function()
sage: len([i for i in p if r(i) == 1])
6
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> A = hyperplane_arrangements.Shi(Integer(3))
>>> A.whitney_number(Integer(0))
1
>>> A.whitney_number(Integer(1))
-6
>>> A.whitney_number(Integer(2))
9
>>> A.characteristic_polynomial()
x^3 - 6*x^2 + 9*x
>>> A.whitney_number(Integer(1), kind=Integer(2))
```

(continues on next page)

(continued from previous page)

```
6
>>> p = A.intersection_poset()
>>> r = p.rank_function()
>>> len([i for i in p if r(i) == Integer(1)])
6
```

class sage.geometry.hyperplane_arrangement.arrangement.**HyperplaneArrangements**(*base_ring*, *names=()*)

Bases: `Parent`, `UniqueRepresentation`

Hyperplane arrangements.

For more information on hyperplane arrangements, see *sage.geometry.hyperplane_arrangement.arrangement*.

INPUT:

- `base_ring` – ring; the base ring
- `names` – tuple of strings; the variable names

EXAMPLES:

```
sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: x
Hyperplane x + 0*y + 0
sage: x + y
Hyperplane x + y + 0
sage: H(x, y, x-1, y-1)
Arrangement <y - 1 | y | x - 1 | x>
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = H._first_ngens(2)
>>> x
Hyperplane x + 0*y + 0
>>> x + y
Hyperplane x + y + 0
>>> H(x, y, x-Integer(1), y-Integer(1))
Arrangement <y - 1 | y | x - 1 | x>
```

Element

alias of *HyperplaneArrangementElement*

ambient_space()

Return the ambient space.

The ambient space is the parent of hyperplanes. That is, new hyperplanes are always constructed internally from the ambient space instance.

EXAMPLES:

```
sage: L.<x, y> = HyperplaneArrangements(QQ)
sage: L.ambient_space()([(1,0), 0])
Hyperplane x + 0*y + 0
sage: L.ambient_space()([(1,0), 0]) == x
True
```

```
>>> from sage.all import *
>>> L = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = L._first_
↪ngens(2)
>>> L.ambient_space()([(Integer(1),Integer(0)), Integer(0)])
Hyperplane x + 0*y + 0
>>> L.ambient_space()([(Integer(1),Integer(0)), Integer(0)]) == x
True
```

base_ring()

Return the base ring.

OUTPUT: the base ring of the hyperplane arrangement

EXAMPLES:

```
sage: L.<x,y> = HyperplaneArrangements(QQ)
sage: L.base_ring()
Rational Field
```

```
>>> from sage.all import *
>>> L = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = L._first_
↪ngens(2)
>>> L.base_ring()
Rational Field
```

change_ring(*base_ring*)

Return hyperplane arrangements over a different base ring.

INPUT:

- `base_ring` – a ring; the new base ring

OUTPUT:

A new *HyperplaneArrangements* instance over the new base ring.

EXAMPLES:

```
sage: L.<x,y> = HyperplaneArrangements(QQ)
sage: L.gen(0)
Hyperplane x + 0*y + 0
sage: L.change_ring(RR).gen(0)
Hyperplane 1.00000000000000*x + 0.000000000000000*y + 0.000000000000000
```

```
>>> from sage.all import *
>>> L = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = L._first_
↪ngens(2)
>>> L.gen(Integer(0))
Hyperplane x + 0*y + 0
>>> L.change_ring(RR).gen(Integer(0))
Hyperplane 1.00000000000000*x + 0.000000000000000*y + 0.000000000000000
```

gen(i)

Return the i-th coordinate hyperplane.

INPUT:

- `i` – integer

OUTPUT: a linear expression

EXAMPLES:

```
sage: L.<x, y, z> = HyperplaneArrangements(QQ);  L
Hyperplane arrangements in
 3-dimensional linear space over Rational Field with coordinates x, y, z
sage: L.gen(0)
Hyperplane x + 0*y + 0*z + 0
```

```
>>> from sage.all import *
>>> L = HyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,) = L._
↪first_ngens(3);  L
Hyperplane arrangements in
 3-dimensional linear space over Rational Field with coordinates x, y, z
>>> L.gen(Integer(0))
Hyperplane x + 0*y + 0*z + 0
```

gens()

Return the coordinate hyperplanes.

OUTPUT: a tuple of linear expressions, one for each linear variable

EXAMPLES:

```
sage: L = HyperplaneArrangements(QQ, ('x', 'y', 'z'))
sage: L.gens()
(Hyperplane x + 0*y + 0*z + 0,
 Hyperplane 0*x + y + 0*z + 0,
 Hyperplane 0*x + 0*y + z + 0)
```

```
>>> from sage.all import *
>>> L = HyperplaneArrangements(QQ, ('x', 'y', 'z'))
>>> L.gens()
(Hyperplane x + 0*y + 0*z + 0,
 Hyperplane 0*x + y + 0*z + 0,
 Hyperplane 0*x + 0*y + z + 0)
```

ngens()

Return the number of linear variables.

OUTPUT: integer

EXAMPLES:

```
sage: L.<x, y, z> = HyperplaneArrangements(QQ);  L
Hyperplane arrangements in 3-dimensional linear space
 over Rational Field with coordinates x, y, z
sage: L.ngens()
3
```

```
>>> from sage.all import *
>>> L = HyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,) = L._
```

(continues on next page)

(continued from previous page)

```
↪first_ngens(3);   L
Hyperplane arrangements in 3-dimensional linear space
 over Rational Field with coordinates x, y, z
>>> L.ngens()
3
```

1.2 Ordered Hyperplane Arrangements

The *HyperplaneArrangements* orders the hyperplanes in a arrangement independently of the way the hyperplanes are introduced. The class *OrderedHyperplaneArrangements* fixes an order specified by the user. This can be needed for certain properties, e.g., fundamental group with information about meridians, braid monodromy with information about the strands; in the future, it may be useful for combinatorial properties. There are no other differences with usual hyperplane arrangements.

An ordered arrangement is an arrangement where the hyperplanes are sorted by the user:

```
sage: H0.<t0, t1, t2> = HyperplaneArrangements(QQ)
sage: H0(t0 - t1, t1 - t2, t0 - t2)
Arrangement <t1 - t2 | t0 - t1 | t0 - t2>
sage: H.<t0, t1, t2> = OrderedHyperplaneArrangements(QQ)
sage: H(t0 - t1, t1 - t2, t0 - t2)
Arrangement <t0 - t1 | t1 - t2 | t0 - t2>
```

```
>>> from sage.all import *
>>> H0 = HyperplaneArrangements(QQ, names=('t0', 't1', 't2',)); (t0, t1, t2,) = H0._
↪first_ngens(3)
>>> H0(t0 - t1, t1 - t2, t0 - t2)
Arrangement <t1 - t2 | t0 - t1 | t0 - t2>
>>> H = OrderedHyperplaneArrangements(QQ, names=('t0', 't1', 't2',)); (t0, t1, t2,) =␣
↪H._first_ngens(3)
>>> H(t0 - t1, t1 - t2, t0 - t2)
Arrangement <t0 - t1 | t1 - t2 | t0 - t2>
```

Some methods are adapted, e.g., *hyperplanes()*, and some new ones are created, regarding hyperplane sections and fundamental groups:

```
sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: H1.<x,y> = OrderedHyperplaneArrangements(QQ)
sage: A1 = H1(x, y); A = H(A1)
sage: A.hyperplanes()
(Hyperplane 0*x + y + 0, Hyperplane x + 0*y + 0)
sage: A1.hyperplanes()
(Hyperplane x + 0*y + 0, Hyperplane 0*x + y + 0)
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = H._first_ngens(2)
>>> H1 = OrderedHyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = H1._first_
↪ngens(2)
>>> A1 = H1(x, y); A = H(A1)
>>> A.hyperplanes()
(Hyperplane 0*x + y + 0, Hyperplane x + 0*y + 0)
```

(continues on next page)

(continued from previous page)

```
>>> A1.hyperplanes()
(Hyperplane x + 0*y + 0, Hyperplane 0*x + y + 0)
```

We see the differences in `union()`:

```
sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: H1.<x,y> = OrderedHyperplaneArrangements(QQ)
sage: A = H([1,2,3], [0,1,1], [0,1,-1], [1,-1,0], [1,1,0])
sage: B = H([1,1,1], [1,-1,1], [1,0,-1])
sage: C = A.union(B)
sage: A1 = H1(A); B1 = H1(B); C1 = A1.union(B1)
sage: [C1.hyperplanes().index(h) for h in C.hyperplanes()]
[0, 5, 6, 1, 2, 3, 7, 4]
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = H._first_ngens(2)
>>> H1 = OrderedHyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = H1._first_
↪ngens(2)
>>> A = H([Integer(1),Integer(2),Integer(3)], [Integer(0),Integer(1),Integer(1)],
↪[Integer(0),Integer(1),-Integer(1)], [Integer(1),-Integer(1),Integer(0)],
↪[Integer(1),Integer(1),Integer(0)])
>>> B = H([Integer(1),Integer(1),Integer(1)], [Integer(1),-Integer(1),Integer(1)],
↪[Integer(1),Integer(0),-Integer(1)])
>>> C = A.union(B)
>>> A1 = H1(A); B1 = H1(B); C1 = A1.union(B1)
>>> [C1.hyperplanes().index(h) for h in C.hyperplanes()]
[0, 5, 6, 1, 2, 3, 7, 4]
```

Also in meth: $sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement.cone$:

```
sage: # needs sage.combinat
sage: a.<x,y,z> = hyperplane_arrangements.semiorder(3)
sage: H.<x,y,z> = OrderedHyperplaneArrangements(QQ)
sage: a1 = H(a)
sage: b = a.cone(); b1 = a1.cone()
sage: [b1.hyperplanes().index(h) for h in b.hyperplanes()]
[0, 2, 4, 6, 1, 3, 5]
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> a = hyperplane_arrangements.semiorder(Integer(3), names=('x', 'y', 'z',)); (x, y,
↪z,) = a._first_ngens(3)
>>> H = OrderedHyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,) = H._
↪first_ngens(3)
>>> a1 = H(a)
>>> b = a.cone(); b1 = a1.cone()
>>> [b1.hyperplanes().index(h) for h in b.hyperplanes()]
[0, 2, 4, 6, 1, 3, 5]
```

And in `restriction()`:

```
sage: # needs sage.graphs
sage: A.<u, x, y, z> = hyperplane_arrangements.braid(4)
sage: L.<u, x, y, z> = OrderedHyperplaneArrangements(QQ)
sage: A1 = L(A)
sage: H = A[0]; H
Hyperplane 0*u + 0*x + y - z + 0
sage: A.restriction(H)
Arrangement <x - z | u - x | u - z>
sage: A1.restriction(H)
Arrangement <x - z | u - x | u - z>
sage: A1.restriction(H, repetitions=True)
Arrangement of 5 hyperplanes of dimension 3 and rank 2
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> A = hyperplane_arrangements.braid(Integer(4), names=('u', 'x', 'y', 'z',)); (u, x,
↪ y, z,) = A._first_ngens(4)
>>> L = OrderedHyperplaneArrangements(QQ, names=('u', 'x', 'y', 'z',)); (u, x, y, z,)␣
↪= L._first_ngens(4)
>>> A1 = L(A)
>>> H = A[Integer(0)]; H
Hyperplane 0*u + 0*x + y - z + 0
>>> A.restriction(H)
Arrangement <x - z | u - x | u - z>
>>> A1.restriction(H)
Arrangement <x - z | u - x | u - z>
>>> A1.restriction(H, repetitions=True)
Arrangement of 5 hyperplanes of dimension 3 and rank 2
```

AUTHORS:

- Enrique Artal (2023-12): initial version

This module adds some features to the *unordered* one for some properties which depend on the order.

class sage.geometry.hyperplane_arrangement.ordered_arrangement.**OrderedHyperplaneArrangementElement**(*pa en hy pe pl ch ba en*

Bases: *HyperplaneArrangementElement*

An ordered hyperplane arrangement.

> **Warning**
>
> You should never create *OrderedHyperplaneArrangementElement* instances directly, always use the parent.

affine_fundamental_group()

Return the fundamental group of the complement of an affine hyperplane arrangement in $\mathbf{C}^n$ whose equations have coefficients in a subfield of $\overline{\mathbf{Q}}$.

OUTPUT: a finitely presented fundamental group

Note: This functionality requires the `sirocco` package to be installed.

EXAMPLES:

```
sage: # needs sirocco
sage: A.<x, y> = OrderedHyperplaneArrangements(QQ)
sage: L = [y + x, y + x - 1]
sage: H = A(L)
sage: H.affine_fundamental_group()
Finitely presented group < x0, x1 |  >
sage: L = [x, y, x + 1, y + 1, x - y]
sage: A(L).affine_fundamental_group()
Finitely presented group
< x0, x1, x2, x3, x4 | x4*x0*x4^-1*x0^-1,
                       x0*x2*x3*x2^-1*x0^-1*x3^-1,
                       x1*x2*x4*x2^-1*x1^-1*x4^-1,
                       x2*x3*x0*x2^-1*x0^-1*x3^-1,
                       x2*x4*x1*x2^-1*x1^-1*x4^-1,
                       x4*x1*x4^-1*x3^-1*x2^-1*x1^-1*x2*x3 >
sage: H = A(x, y, x + y)
sage: H.affine_fundamental_group()
Finitely presented group
< x0, x1, x2 | x0*x1*x2*x1^-1*x0^-1*x2^-1, x1*x2*x0*x1^-1*x0^-1*x2^-1 >
sage: H.affine_fundamental_group()  # repeat to use the attribute
Finitely presented group
< x0, x1, x2 | x0*x1*x2*x1^-1*x0^-1*x2^-1, x1*x2*x0*x1^-1*x0^-1*x2^-1 >
sage: T.<t> = QQ[]
sage: K.<a> = NumberField(t^3 + t + 1)
sage: L.<x, y> = OrderedHyperplaneArrangements(K)
sage: H = L(a*x + y -1, x + a*y + 1, x - 1, y - 1)
sage: H.affine_fundamental_group()
Traceback (most recent call last):
...
TypeError: the base field is not in QQbar
sage: L.<t> = OrderedHyperplaneArrangements(QQ)
sage: L([t - j for j in range(4)]).affine_fundamental_group()
Finitely presented group < x0, x1, x2, x3 |  >
sage: L.<x, y, z> = OrderedHyperplaneArrangements(QQ)
sage: L(L.gens() + (x + y + z + 1,)).affine_fundamental_group().sorted_
→presentation()
Finitely presented group
< x0, x1, x2, x3 | x3^-1*x2^-1*x3*x2, x3^-1*x1^-1*x3*x1,
                   x3^-1*x0^-1*x3*x0, x2^-1*x1^-1*x2*x1,
                   x2^-1*x0^-1*x2*x0, x1^-1*x0^-1*x1*x0 >
sage: A = OrderedHyperplaneArrangements(QQ, names=())
sage: H = A(); H
Empty hyperplane arrangement of dimension 0
```

(continues on next page)

(continued from previous page)

```
sage: H.affine_fundamental_group()
Finitely presented group <  |  >
```

```
>>> from sage.all import *
>>> # needs sirocco
>>> A = OrderedHyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = A._
↪first_ngens(2)
>>> L = [y + x, y + x - Integer(1)]
>>> H = A(L)
>>> H.affine_fundamental_group()
Finitely presented group < x0, x1 |  >
>>> L = [x, y, x + Integer(1), y + Integer(1), x - y]
>>> A(L).affine_fundamental_group()
Finitely presented group
< x0, x1, x2, x3, x4 | x4*x0*x4^-1*x0^-1,
                       x0*x2*x3*x2^-1*x0^-1*x3^-1,
                       x1*x2*x4*x2^-1*x1^-1*x4^-1,
                       x2*x3*x0*x2^-1*x0^-1*x3^-1,
                       x2*x4*x1*x2^-1*x1^-1*x4^-1,
                       x4*x1*x4^-1*x3^-1*x2^-1*x1^-1*x2*x3 >
>>> H = A(x, y, x + y)
>>> H.affine_fundamental_group()
Finitely presented group
< x0, x1, x2 | x0*x1*x2*x1^-1*x0^-1*x2^-1, x1*x2*x0*x1^-1*x0^-1*x2^-1 >
>>> H.affine_fundamental_group()   # repeat to use the attribute
Finitely presented group
< x0, x1, x2 | x0*x1*x2*x1^-1*x0^-1*x2^-1, x1*x2*x0*x1^-1*x0^-1*x2^-1 >
>>> T = QQ['t']; (t,) = T._first_ngens(1)
>>> K = NumberField(t**Integer(3) + t + Integer(1), names=('a',)); (a,) = K._
↪first_ngens(1)
>>> L = OrderedHyperplaneArrangements(K, names=('x', 'y',)); (x, y,) = L._
↪first_ngens(2)
>>> H = L(a*x + y -Integer(1), x + a*y + Integer(1), x - Integer(1), y -␣
↪Integer(1))
>>> H.affine_fundamental_group()
Traceback (most recent call last):
...
TypeError: the base field is not in QQbar
>>> L = OrderedHyperplaneArrangements(QQ, names=('t',)); (t,) = L._first_
↪ngens(1)
>>> L([t - j for j in range(Integer(4))]).affine_fundamental_group()
Finitely presented group < x0, x1, x2, x3 |  >
>>> L = OrderedHyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,)␣
↪= L._first_ngens(3)
>>> L(L.gens() + (x + y + z + Integer(1),)).affine_fundamental_group().sorted_
↪presentation()
Finitely presented group
< x0, x1, x2, x3 | x3^-1*x2^-1*x3*x2, x3^-1*x1^-1*x3*x1,
                   x3^-1*x0^-1*x3*x0, x2^-1*x1^-1*x2*x1,
                   x2^-1*x0^-1*x2*x0, x1^-1*x0^-1*x1*x0 >
>>> A = OrderedHyperplaneArrangements(QQ, names=())
>>> H = A(); H
```

(continues on next page)

(continued from previous page)

```
Empty hyperplane arrangement of dimension 0
>>> H.affine_fundamental_group()
Finitely presented group <  |  >
```

affine_meridians()

Return the meridians of each hyperplane (including the one at infinity).

OUTPUT: a dictionary

> **Note**
>
> This functionality requires the `sirocco` package to be installed.

EXAMPLES:

```
sage: # needs sirocco
sage: A.<x, y> = OrderedHyperplaneArrangements(QQ)
sage: L = [y + x, y + x - 1]
sage: H = A(L)
sage: g = H.affine_fundamental_group()
sage: g
Finitely presented group < x0, x1 |  >
sage: H.affine_meridians()
{0: [x0], 1: [x1], 2: [x1^-1*x0^-1]}
sage: H1 = H.add_hyperplane(y - x)
sage: H1.affine_meridians()
{0: [x0], 1: [x1], 2: [x2], 3: [x2^-1*x1^-1*x0^-1]}
```

```
>>> from sage.all import *
>>> # needs sirocco
>>> A = OrderedHyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = A._
↪first_ngens(2)
>>> L = [y + x, y + x - Integer(1)]
>>> H = A(L)
>>> g = H.affine_fundamental_group()
>>> g
Finitely presented group < x0, x1 |  >
>>> H.affine_meridians()
{0: [x0], 1: [x1], 2: [x1^-1*x0^-1]}
>>> H1 = H.add_hyperplane(y - x)
>>> H1.affine_meridians()
{0: [x0], 1: [x1], 2: [x2], 3: [x2^-1*x1^-1*x0^-1]}
```

hyperplane_section(*proj=True*)

Compute a generic hyperplane section of `self`.

INPUT:

- `proj` – (default: `True`) if the ambient space is affine or projective

OUTPUT:

An arrangement $\mathcal{A}$ obtained by intersecting with a generic hyperplane

EXAMPLES:

```
sage: L.<x, y, z> = OrderedHyperplaneArrangements(QQ)
sage: L(x, y - 1, z).hyperplane_section()
Traceback (most recent call last):
...
TypeError: the arrangement is not projective

sage: # needs sage.graphs
sage: A0.<u,x,y,z> = hyperplane_arrangements.braid(4); A0
Arrangement of 6 hyperplanes of dimension 4 and rank 3
sage: L.<u,x,y,z> = OrderedHyperplaneArrangements(QQ)
sage: A = L(A0)
sage: M = A.matroid()
sage: A1 = A.hyperplane_section()
sage: A1
Arrangement of 6 hyperplanes of dimension 3 and rank 3
sage: M1 = A1.matroid()
sage: A2 = A1.hyperplane_section(); A2
Arrangement of 6 hyperplanes of dimension 2 and rank 2
sage: M2 = A2.matroid()
sage: T1 = M1.truncation()
sage: T1.is_isomorphic(M2)
True
sage: T1.isomorphism(M2)
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5}

sage: # needs sage.combinat
sage: a0 = hyperplane_arrangements.semiorder(3); a0
Arrangement of 6 hyperplanes of dimension 3 and rank 2
sage: L.<t0, t1, t2> = OrderedHyperplaneArrangements(QQ)
sage: a = L(a0)
sage: ca = a.cone()
sage: m = ca.matroid()
sage: a1 = a.hyperplane_section(proj=False)
sage: a1
Arrangement of 6 hyperplanes of dimension 2 and rank 2
sage: ca1 = a1.cone()
sage: m1 = ca1.matroid()
sage: m.isomorphism(m1)
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6}
sage: p0 = hyperplane_arrangements.Shi(4)
sage: L.<t0, t1, t2, t3> = OrderedHyperplaneArrangements(QQ)
sage: p = L(p0)
sage: a = p.hyperplane_section(proj=False); a
Arrangement of 12 hyperplanes of dimension 3 and rank 3
sage: ca = a.cone()
sage: m = ca.matroid().truncation()
sage: a1 = a.hyperplane_section(proj=False); a1
Arrangement of 12 hyperplanes of dimension 2 and rank 2
sage: ca1 = a1.cone()
sage: m1 = ca1.matroid()
sage: m1.is_isomorphism(m, {j: j for j in range(13)})
True
```

```
>>> from sage.all import *
>>> L = OrderedHyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,)␣
↪= L._first_ngens(3)
>>> L(x, y - Integer(1), z).hyperplane_section()
Traceback (most recent call last):
...
TypeError: the arrangement is not projective

>>> # needs sage.graphs
>>> A0 = hyperplane_arrangements.braid(Integer(4), names=('u', 'x', 'y', 'z',
↪)); (u, x, y, z,) = A0._first_ngens(4); A0
Arrangement of 6 hyperplanes of dimension 4 and rank 3
>>> L = OrderedHyperplaneArrangements(QQ, names=('u', 'x', 'y', 'z',)); (u, x,
↪ y, z,) = L._first_ngens(4)
>>> A = L(A0)
>>> M = A.matroid()
>>> A1 = A.hyperplane_section()
>>> A1
Arrangement of 6 hyperplanes of dimension 3 and rank 3
>>> M1 = A1.matroid()
>>> A2 = A1.hyperplane_section(); A2
Arrangement of 6 hyperplanes of dimension 2 and rank 2
>>> M2 = A2.matroid()
>>> T1 = M1.truncation()
>>> T1.is_isomorphic(M2)
True
>>> T1.isomorphism(M2)
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5}

>>> # needs sage.combinat
>>> a0 = hyperplane_arrangements.semiorder(Integer(3)); a0
Arrangement of 6 hyperplanes of dimension 3 and rank 2
>>> L = OrderedHyperplaneArrangements(QQ, names=('t0', 't1', 't2',)); (t0, t1,
↪ t2,) = L._first_ngens(3)
>>> a = L(a0)
>>> ca = a.cone()
>>> m = ca.matroid()
>>> a1 = a.hyperplane_section(proj=False)
>>> a1
Arrangement of 6 hyperplanes of dimension 2 and rank 2
>>> ca1 = a1.cone()
>>> m1 = ca1.matroid()
>>> m.isomorphism(m1)
{0: 0, 1: 1, 2: 2, 3: 3, 4: 4, 5: 5, 6: 6}
>>> p0 = hyperplane_arrangements.Shi(Integer(4))
>>> L = OrderedHyperplaneArrangements(QQ, names=('t0', 't1', 't2', 't3',));␣
↪(t0, t1, t2, t3,) = L._first_ngens(4)
>>> p = L(p0)
>>> a = p.hyperplane_section(proj=False); a
Arrangement of 12 hyperplanes of dimension 3 and rank 3
>>> ca = a.cone()
>>> m = ca.matroid().truncation()
>>> a1 = a.hyperplane_section(proj=False); a1
```

(continues on next page)

(continued from previous page)

```
Arrangement of 12 hyperplanes of dimension 2 and rank 2
>>> ca1 = a1.cone()
>>> m1 = ca1.matroid()
>>> m1.is_isomorphism(m, {j: j for j in range(Integer(13))})
True
```

projective_fundamental_group()

Return the fundamental group of the complement of a projective hyperplane arrangement.

OUTPUT:

The finitely presented group of the complement in the projective space whose equations have coefficients in a subfield of $\overline{\mathbf{Q}}$.

Note: This functionality requires the `sirocco` package to be installed.

EXAMPLES:

```
sage: # needs sirocco
sage: A.<x, y> = OrderedHyperplaneArrangements(QQ)
sage: H = A(x, y, x + y)
sage: H.projective_fundamental_group()
Finitely presented group < x0, x1 |  >

sage: # needs sirocco sage.graphs
sage: A3.<x, y, z> = OrderedHyperplaneArrangements(QQ)
sage: H = A3(hyperplane_arrangements.braid(4).essentialization())
sage: G3 = H.projective_fundamental_group(); G3.sorted_presentation()
Finitely presented group
< x0, x1, x2, x3, x4 | x4^-1*x3^-1*x2^-1*x3*x4*x0*x2*x0^-1,
                       x4^-1*x2^-1*x4*x2, x4^-1*x1^-1*x0^-1*x1*x4*x0,
                       x4^-1*x1^-1*x0^-1*x4*x0*x1,
                       x3^-1*x2^-1*x1^-1*x0^-1*x3*x0*x1*x2,
                       x3^-1*x1^-1*x3*x1 >
sage: G3.abelian_invariants()
(0, 0, 0, 0, 0)
sage: A4.<t1, t2, t3, t4> = OrderedHyperplaneArrangements(QQ)
sage: H = A4(hyperplane_arrangements.braid(4))
sage: G4 = H.projective_fundamental_group(); G4.sorted_presentation()
Finitely presented group
< x0, x1, x2, x3, x4 | x4^-1*x3^-1*x2^-1*x3*x4*x0*x2*x0^-1,
                       x4^-1*x2^-1*x4*x2, x4^-1*x1^-1*x0^-1*x1*x4*x0,
                       x4^-1*x1^-1*x0^-1*x4*x0*x1,
                       x3^-1*x2^-1*x1^-1*x0^-1*x3*x0*x1*x2, x3^-1*x1^-1*x3*x1
↪>
sage: G4.abelian_invariants()
(0, 0, 0, 0, 0)

sage: # needs sirocco
sage: L.<t0, t1, t2, t3, t4> = OrderedHyperplaneArrangements(QQ)
```

(continues on next page)

(continued from previous page)

```
sage: H = hyperplane_arrangements.coordinate(5)
sage: H = L(H)
sage: g = H.projective_fundamental_group()
sage: g.is_abelian(), g.abelian_invariants()
(True, (0, 0, 0, 0))
sage: L(t0, t1, t2, t3, t4, t0 - 1).projective_fundamental_group()
Traceback (most recent call last):
...
TypeError: the arrangement is not projective
sage: T.<t> = QQ[]
sage: K.<a> = NumberField(t^3 + t + 1)
sage: L.<x, y, z> = OrderedHyperplaneArrangements(K)
sage: H = L(a*x + y - z, x + a*y + z, x - z, y - z)
sage: H.projective_fundamental_group()
Traceback (most recent call last):
...
TypeError: the base field is not in QQbar
sage: A.<x> = OrderedHyperplaneArrangements(QQ)
sage: H = A(); H
Empty hyperplane arrangement of dimension 1
sage: H.projective_fundamental_group()
Finitely presented group <  |  >
```

```
>>> from sage.all import *
>>> # needs sirocco
>>> A = OrderedHyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = A._
↪first_ngens(2)
>>> H = A(x, y, x + y)
>>> H.projective_fundamental_group()
Finitely presented group < x0, x1 |  >

>>> # needs sirocco sage.graphs
>>> A3 = OrderedHyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,
↪) = A3._first_ngens(3)
>>> H = A3(hyperplane_arrangements.braid(Integer(4)).essentialization())
>>> G3 = H.projective_fundamental_group(); G3.sorted_presentation()
Finitely presented group
< x0, x1, x2, x3, x4 | x4^-1*x3^-1*x2^-1*x3*x4*x0*x2*x0^-1,
                       x4^-1*x2^-1*x4*x2, x4^-1*x1^-1*x0^-1*x1*x4*x0,
                       x4^-1*x1^-1*x0^-1*x4*x0*x1,
                       x3^-1*x2^-1*x1^-1*x0^-1*x3*x0*x1*x2,
                       x3^-1*x1^-1*x3*x1 >
>>> G3.abelian_invariants()
(0, 0, 0, 0, 0)
>>> A4 = OrderedHyperplaneArrangements(QQ, names=('t1', 't2', 't3', 't4',));␣
↪(t1, t2, t3, t4,) = A4._first_ngens(4)
>>> H = A4(hyperplane_arrangements.braid(Integer(4)))
>>> G4 = H.projective_fundamental_group(); G4.sorted_presentation()
Finitely presented group
< x0, x1, x2, x3, x4 | x4^-1*x3^-1*x2^-1*x3*x4*x0*x2*x0^-1,
                       x4^-1*x2^-1*x4*x2, x4^-1*x1^-1*x0^-1*x1*x4*x0,
                       x4^-1*x1^-1*x0^-1*x4*x0*x1,
```

(continues on next page)

(continued from previous page)

```
                        x3^-1*x2^-1*x1^-1*x0^-1*x3*x0*x1*x2, x3^-1*x1^-1*x3*x1␣
↪>
>>> G4.abelian_invariants()
(0, 0, 0, 0, 0)

>>> # needs sirocco
>>> L = OrderedHyperplaneArrangements(QQ, names=('t0', 't1', 't2', 't3', 't4',
↪)); (t0, t1, t2, t3, t4,) = L._first_ngens(5)
>>> H = hyperplane_arrangements.coordinate(Integer(5))
>>> H = L(H)
>>> g = H.projective_fundamental_group()
>>> g.is_abelian(), g.abelian_invariants()
(True, (0, 0, 0, 0))
>>> L(t0, t1, t2, t3, t4, t0 - Integer(1)).projective_fundamental_group()
Traceback (most recent call last):
...
TypeError: the arrangement is not projective
>>> T = QQ['t']; (t,) = T._first_ngens(1)
>>> K = NumberField(t**Integer(3) + t + Integer(1), names=('a',)); (a,) = K._
↪first_ngens(1)
>>> L = OrderedHyperplaneArrangements(K, names=('x', 'y', 'z',)); (x, y, z,)␣
↪= L._first_ngens(3)
>>> H = L(a*x + y - z, x + a*y + z, x - z, y - z)
>>> H.projective_fundamental_group()
Traceback (most recent call last):
...
TypeError: the base field is not in QQbar
>>> A = OrderedHyperplaneArrangements(QQ, names=('x',)); (x,) = A._first_
↪ngens(1)
>>> H = A(); H
Empty hyperplane arrangement of dimension 1
>>> H.projective_fundamental_group()
Finitely presented group <  |  >
```

projective_meridians()

Return the meridian of each hyperplane.

OUTPUT: a dictionary

> **Note**
>
> This functionality requires the `sirocco` package to be installed.

EXAMPLES:

```
sage: # needs sirocco
sage: A.<x, y> = OrderedHyperplaneArrangements(QQ)
sage: H = A(x, y, x + y)
sage: H.projective_meridians()
{0: x0, 1: x1, 2: [x1^-1*x0^-1]}
```

(continues on next page)

(continued from previous page)

```
sage: # needs sirocco sage.graphs
sage: A3.<x, y, z> = OrderedHyperplaneArrangements(QQ)
sage: H = A3(hyperplane_arrangements.braid(4).essentialization())
sage: H.projective_meridians()
{0: [x2^-1*x0^-1*x4^-1*x3^-1*x1^-1],
 1: [x3], 2: [x4], 3: [x1], 4: [x2], 5: [x0]}
sage: A4.<t1, t2, t3, t4> = OrderedHyperplaneArrangements(QQ)
sage: H = A4(hyperplane_arrangements.braid(4))
sage: H.projective_meridians()
{0: [x2^-1*x0^-1*x4^-1*x3^-1*x1^-1], 1: [x3],
 2: [x4], 3: [x0], 4: [x2], 5: [x1]}

sage: # needs sirocco
sage: L.<t0, t1, t2, t3, t4> = OrderedHyperplaneArrangements(QQ)
sage: H = hyperplane_arrangements.coordinate(5)
sage: H = L(H)
sage: H.projective_meridians()
{0: [x2], 1: [x3], 2: [x0], 3: [x3^-1*x2^-1*x1^-1*x0^-1], 4: [x1]}
```

```
>>> from sage.all import *
>>> # needs sirocco
>>> A = OrderedHyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = A._
↪first_ngens(2)
>>> H = A(x, y, x + y)
>>> H.projective_meridians()
{0: x0, 1: x1, 2: [x1^-1*x0^-1]}

>>> # needs sirocco sage.graphs
>>> A3 = OrderedHyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,
↪) = A3._first_ngens(3)
>>> H = A3(hyperplane_arrangements.braid(Integer(4)).essentialization())
>>> H.projective_meridians()
{0: [x2^-1*x0^-1*x4^-1*x3^-1*x1^-1],
 1: [x3], 2: [x4], 3: [x1], 4: [x2], 5: [x0]}
>>> A4 = OrderedHyperplaneArrangements(QQ, names=('t1', 't2', 't3', 't4',));␣
↪(t1, t2, t3, t4,) = A4._first_ngens(4)
>>> H = A4(hyperplane_arrangements.braid(Integer(4)))
>>> H.projective_meridians()
{0: [x2^-1*x0^-1*x4^-1*x3^-1*x1^-1], 1: [x3],
 2: [x4], 3: [x0], 4: [x2], 5: [x1]}

>>> # needs sirocco
>>> L = OrderedHyperplaneArrangements(QQ, names=('t0', 't1', 't2', 't3', 't4',
↪)); (t0, t1, t2, t3, t4,) = L._first_ngens(5)
>>> H = hyperplane_arrangements.coordinate(Integer(5))
>>> H = L(H)
>>> H.projective_meridians()
{0: [x2], 1: [x3], 2: [x0], 3: [x3^-1*x2^-1*x1^-1*x0^-1], 4: [x1]}
```

class sage.geometry.hyperplane_arrangement.ordered_arrangement.**OrderedHyperplaneArrangements**(*base_ring*, *names=()*)

Bases: *HyperplaneArrangements*

Ordered Hyperplane arrangements.

For more information on hyperplane arrangements, see *sage.geometry.hyperplane_arrangement.arrangement*.

INPUT:

- `base_ring` – ring; the base ring
- `names` – tuple of strings; the variable names

EXAMPLES:

```
sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: x
Hyperplane x + 0*y + 0
sage: x + y
Hyperplane x + y + 0
sage: H(x, y, x-1, y-1)
Arrangement <y - 1 | y | x - 1 | x>
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = H._first_ngens(2)
>>> x
Hyperplane x + 0*y + 0
>>> x + y
Hyperplane x + y + 0
>>> H(x, y, x-Integer(1), y-Integer(1))
Arrangement <y - 1 | y | x - 1 | x>
```

Element

alias of *OrderedHyperplaneArrangementElement*

1.3 Library of Hyperplane Arrangements

A collection of useful or interesting hyperplane arrangements. See *sage.geometry.hyperplane_arrangement.arrangement* for details about how to construct your own hyperplane arrangements.

class sage.geometry.hyperplane_arrangement.library.**HyperplaneArrangementLibrary**

Bases: `object`

The library of hyperplane arrangements.

Catalan(*n*, *K=Rational Field*, *names=None*)

Return the Catalan arrangement.

INPUT:

- `n` – integer
- `K` – field (default: $\mathbf{Q}$)
- `names` – tuple of strings or `None` (default); the variable names for the ambient space

OUTPUT:

The arrangement of $3n(n-1)/2$ hyperplanes $\{x_i - x_j = -1, 0, 1 : 1 \leq i \leq j \leq n\}$.

EXAMPLES:

```
sage: hyperplane_arrangements.Catalan(5)
Arrangement of 30 hyperplanes of dimension 5 and rank 4
```

```
>>> from sage.all import *
>>> hyperplane_arrangements.Catalan(Integer(5))
Arrangement of 30 hyperplanes of dimension 5 and rank 4
```

Coxeter(*data*, *K=Rational Field*, *names=None*)

Return the Coxeter arrangement.

This generalizes the braid arrangements to crystallographic root systems.

INPUT:

- `data` – either an integer or a Cartan type (or coercible into; see "CartanType")
- `K` – field (default: `QQ`)
- `names` – tuple of strings or `None` (default); the variable names for the ambient space

OUTPUT:

- If `data` is an integer n, return the braid arrangement in dimension n, i.e. the set of $n(n-1)$ hyperplanes: $\{x_i - x_j = 0, 1 : 1 \leq i \leq j \leq n\}$. This corresponds to the Coxeter arrangement of Cartan type A_{n-1}.
- If `data` is a Cartan type, return the Coxeter arrangement of given type.

The Coxeter arrangement of a given crystallographic Cartan type is defined by the inner products $\langle a, x\rangle = 0$ where $a \in \Phi^+$ runs over positive roots of the root system Φ.

EXAMPLES:

```
sage: # needs sage.combinat
sage: hyperplane_arrangements.Coxeter(4)
Arrangement of 6 hyperplanes of dimension 4 and rank 3
sage: hyperplane_arrangements.Coxeter("B4")
Arrangement of 16 hyperplanes of dimension 4 and rank 4
sage: hyperplane_arrangements.Coxeter("A3")
Arrangement of 6 hyperplanes of dimension 4 and rank 3
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> hyperplane_arrangements.Coxeter(Integer(4))
Arrangement of 6 hyperplanes of dimension 4 and rank 3
>>> hyperplane_arrangements.Coxeter("B4")
Arrangement of 16 hyperplanes of dimension 4 and rank 4
>>> hyperplane_arrangements.Coxeter("A3")
Arrangement of 6 hyperplanes of dimension 4 and rank 3
```

If the Cartan type is not crystallographic, the Coxeter arrangement is not implemented yet:

```
sage: hyperplane_arrangements.Coxeter("H3")                                    #␣
↪needs sage.libs.gap
Traceback (most recent call last):
...
NotImplementedError: Coxeter arrangements are not implemented
for non crystallographic Cartan types
```

```
>>> from sage.all import *
>>> hyperplane_arrangements.Coxeter("H3")                              #␣
↪needs sage.libs.gap
Traceback (most recent call last):
...
NotImplementedError: Coxeter arrangements are not implemented
for non crystallographic Cartan types
```

The characteristic polynomial is pre-computed using the results of Terao, see [Ath2000]:

```
sage: # needs sage.combinat
sage: hyperplane_arrangements.Coxeter("A3").characteristic_polynomial()
x^3 - 6*x^2 + 11*x - 6
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> hyperplane_arrangements.Coxeter("A3").characteristic_polynomial()
x^3 - 6*x^2 + 11*x - 6
```

G_Shi(*G*, *K=Rational Field*, *names=None*)

Return the Shi hyperplane arrangement of a graph G.

INPUT:

- `G` – graph
- `K` – field (default: $\mathbf{Q}$)
- `names` – tuple of strings or `None` (default); the variable names for the ambient space

OUTPUT: the Shi hyperplane arrangement of the given graph `G`

EXAMPLES:

```
sage: # needs sage.graphs
sage: G = graphs.CompleteGraph(5)
sage: hyperplane_arrangements.G_Shi(G)
Arrangement of 20 hyperplanes of dimension 5 and rank 4
sage: g = graphs.HouseGraph()
sage: hyperplane_arrangements.G_Shi(g)
Arrangement of 12 hyperplanes of dimension 5 and rank 4
sage: a = hyperplane_arrangements.G_Shi(graphs.WheelGraph(4)); a
Arrangement of 12 hyperplanes of dimension 4 and rank 3
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> G = graphs.CompleteGraph(Integer(5))
>>> hyperplane_arrangements.G_Shi(G)
Arrangement of 20 hyperplanes of dimension 5 and rank 4
>>> g = graphs.HouseGraph()
>>> hyperplane_arrangements.G_Shi(g)
Arrangement of 12 hyperplanes of dimension 5 and rank 4
>>> a = hyperplane_arrangements.G_Shi(graphs.WheelGraph(Integer(4))); a
Arrangement of 12 hyperplanes of dimension 4 and rank 3
```

G_semiorder(*G*, *K=Rational Field*, *names=None*)

Return the semiorder hyperplane arrangement of a graph.

INPUT:

- `G` – graph
- `K` – field (default: **Q**)
- `names` – tuple of strings or `None` (default); the variable names for the ambient space

OUTPUT:

The semiorder hyperplane arrangement of a graph G is the arrangement $\{x_i - x_j = -1, 1\}$ where ij is an edge of `G`.

EXAMPLES:

```
sage: # needs sage.graphs
sage: G = graphs.CompleteGraph(5)
sage: hyperplane_arrangements.G_semiorder(G)
Arrangement of 20 hyperplanes of dimension 5 and rank 4
sage: g = graphs.HouseGraph()
sage: hyperplane_arrangements.G_semiorder(g)
Arrangement of 12 hyperplanes of dimension 5 and rank 4
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> G = graphs.CompleteGraph(Integer(5))
>>> hyperplane_arrangements.G_semiorder(G)
Arrangement of 20 hyperplanes of dimension 5 and rank 4
>>> g = graphs.HouseGraph()
>>> hyperplane_arrangements.G_semiorder(g)
Arrangement of 12 hyperplanes of dimension 5 and rank 4
```

Ish(*n*, *K=Rational Field*, *names=None*)

Return the Ish arrangement.

INPUT:

- `n` – integer
- `K` – field (default: `QQ`)
- `names` – tuple of strings or `None` (default); the variable names for the ambient space

OUTPUT:

The Ish arrangement, which is the set of $n(n-1)$ hyperplanes.

$$\{x_i - x_j = 0 : 1 \leq i \leq j \leq n\} \cup \{x_1 - x_j = i : 1 \leq i \leq j \leq n\}.$$

EXAMPLES:

```
sage: # needs sage.combinat
sage: a = hyperplane_arrangements.Ish(3); a
Arrangement of 6 hyperplanes of dimension 3 and rank 2
sage: a.characteristic_polynomial()
x^3 - 6*x^2 + 9*x
```

(continues on next page)

(continued from previous page)

```
sage: b = hyperplane_arrangements.Shi(3)
sage: b.characteristic_polynomial()
x^3 - 6*x^2 + 9*x
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> a = hyperplane_arrangements.Ish(Integer(3)); a
Arrangement of 6 hyperplanes of dimension 3 and rank 2
>>> a.characteristic_polynomial()
x^3 - 6*x^2 + 9*x
>>> b = hyperplane_arrangements.Shi(Integer(3))
>>> b.characteristic_polynomial()
x^3 - 6*x^2 + 9*x
```

REFERENCES:

- [AR2012]

IshB(*n*, *K=Rational Field*, *names=None*)

Return the type B Ish arrangement.

INPUT:

- `n` – integer
- `K` – field (default: `QQ`)
- `names` – tuple of strings or `None` (default); the variable names for the ambient space

OUTPUT: the type B Ish arrangement, which is the set of $2n^2$ hyperplanes

$$\{x_i \pm x_j = 0 : 1 \leq i < j \leq n\} \cup \{x_i = a : 1 \leq i \leq n, \quad i - n \leq a \leq n - i + 1\}.$$

EXAMPLES:

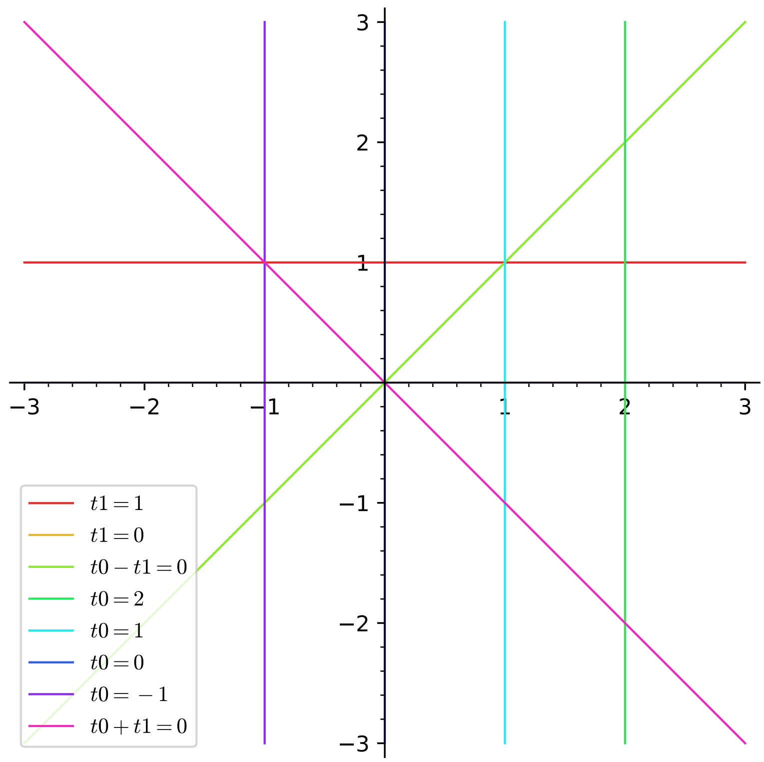
```
sage: a = hyperplane_arrangements.IshB(2)
sage: a
Arrangement of 8 hyperplanes of dimension 2 and rank 2
sage: a.hyperplanes()
(Hyperplane 0*t0 + t1 - 1,
 Hyperplane 0*t0 + t1 + 0,
 Hyperplane t0 - t1 + 0,
 Hyperplane t0 + 0*t1 - 2,
 Hyperplane t0 + 0*t1 - 1,
 Hyperplane t0 + 0*t1 + 0,
 Hyperplane t0 + 0*t1 + 1,
 Hyperplane t0 + t1 + 0)
sage: a.cone().is_free()                                                  #␣
↪needs sage.libs.singular
True
```

```
>>> from sage.all import *
>>> a = hyperplane_arrangements.IshB(Integer(2))
>>> a
Arrangement of 8 hyperplanes of dimension 2 and rank 2
```

(continues on next page)

(continued from previous page)

```
>>> a.hyperplanes()
(Hyperplane 0*t0 + t1 - 1,
 Hyperplane 0*t0 + t1 + 0,
 Hyperplane t0 - t1 + 0,
 Hyperplane t0 + 0*t1 - 2,
 Hyperplane t0 + 0*t1 - 1,
 Hyperplane t0 + 0*t1 + 0,
 Hyperplane t0 + 0*t1 + 1,
 Hyperplane t0 + t1 + 0)
>>> a.cone().is_free()                                                  #␣
↪needs sage.libs.singular
True
```




```
sage: a = hyperplane_arrangements.IshB(3); a
Arrangement of 18 hyperplanes of dimension 3 and rank 3
sage: a.characteristic_polynomial()
x^3 - 18*x^2 + 108*x - 216
sage: b = hyperplane_arrangements.Shi(['B', 3])
sage: b.characteristic_polynomial()
x^3 - 18*x^2 + 108*x - 216
```

```
>>> from sage.all import *
>>> a = hyperplane_arrangements.IshB(Integer(3)); a
Arrangement of 18 hyperplanes of dimension 3 and rank 3
>>> a.characteristic_polynomial()
x^3 - 18*x^2 + 108*x - 216
>>> b = hyperplane_arrangements.Shi(['B', Integer(3)])
```

(continues on next page)

(continued from previous page)

```
>>> b.characteristic_polynomial()
x^3 - 18*x^2 + 108*x - 216
```

REFERENCES:

- [TT2023]

Shi(*data*, *K=Rational Field*, *names=None*, *m=1*)

Return the Shi arrangement.

INPUT:

- `data` – either an integer or a Cartan type (or coercible into; see "CartanType")
- `K` – field (default: `QQ`)
- `names` – tuple of strings or `None` (default); the variable names for the ambient space
- `m` – integer (default: 1)

OUTPUT:

- If `data` is an integer n, return the Shi arrangement in dimension n, i.e. the set of $n(n-1)$ hyperplanes: $\{x_i - x_j = 0, 1 : 1 \leq i \leq j \leq n\}$. This corresponds to the Shi arrangement of Cartan type A_{n-1}.
- If `data` is a Cartan type, return the Shi arrangement of given type.
- If $m > 1$, return the m-extended Shi arrangement of given type.

The m-extended Shi arrangement of a given crystallographic Cartan type is defined by the inner product $\langle a, x\rangle = k$ for $-m < k \leq m$ and $a \in \Phi^+$ is a positive root of the root system Φ.

EXAMPLES:

```
sage: # needs sage.combinat
sage: hyperplane_arrangements.Shi(4)
Arrangement of 12 hyperplanes of dimension 4 and rank 3
sage: hyperplane_arrangements.Shi("A3")
Arrangement of 12 hyperplanes of dimension 4 and rank 3
sage: hyperplane_arrangements.Shi("A3", m=2)
Arrangement of 24 hyperplanes of dimension 4 and rank 3
sage: hyperplane_arrangements.Shi("B4")
Arrangement of 32 hyperplanes of dimension 4 and rank 4
sage: hyperplane_arrangements.Shi("B4", m=3)
Arrangement of 96 hyperplanes of dimension 4 and rank 4
sage: hyperplane_arrangements.Shi("C3")
Arrangement of 18 hyperplanes of dimension 3 and rank 3
sage: hyperplane_arrangements.Shi("D4", m=3)
Arrangement of 72 hyperplanes of dimension 4 and rank 4
sage: hyperplane_arrangements.Shi("E6")
Arrangement of 72 hyperplanes of dimension 8 and rank 6
sage: hyperplane_arrangements.Shi("E6", m=2)
Arrangement of 144 hyperplanes of dimension 8 and rank 6
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> hyperplane_arrangements.Shi(Integer(4))
Arrangement of 12 hyperplanes of dimension 4 and rank 3
```

(continues on next page)

(continued from previous page)

```
>>> hyperplane_arrangements.Shi("A3")
Arrangement of 12 hyperplanes of dimension 4 and rank 3
>>> hyperplane_arrangements.Shi("A3", m=Integer(2))
Arrangement of 24 hyperplanes of dimension 4 and rank 3
>>> hyperplane_arrangements.Shi("B4")
Arrangement of 32 hyperplanes of dimension 4 and rank 4
>>> hyperplane_arrangements.Shi("B4", m=Integer(3))
Arrangement of 96 hyperplanes of dimension 4 and rank 4
>>> hyperplane_arrangements.Shi("C3")
Arrangement of 18 hyperplanes of dimension 3 and rank 3
>>> hyperplane_arrangements.Shi("D4", m=Integer(3))
Arrangement of 72 hyperplanes of dimension 4 and rank 4
>>> hyperplane_arrangements.Shi("E6")
Arrangement of 72 hyperplanes of dimension 8 and rank 6
>>> hyperplane_arrangements.Shi("E6", m=Integer(2))
Arrangement of 144 hyperplanes of dimension 8 and rank 6
```

If the Cartan type is not crystallographic, the Shi arrangement is not defined:

```
sage: hyperplane_arrangements.Shi("H4")
Traceback (most recent call last):
...
NotImplementedError: Shi arrangements are not defined for non␣
↪crystallographic Cartan types
```

```
>>> from sage.all import *
>>> hyperplane_arrangements.Shi("H4")
Traceback (most recent call last):
...
NotImplementedError: Shi arrangements are not defined for non␣
↪crystallographic Cartan types
```

The characteristic polynomial is pre-computed using the results of [Ath1996]:

```
sage: # needs sage.combinat
sage: hyperplane_arrangements.Shi("A3").characteristic_polynomial()
x^4 - 12*x^3 + 48*x^2 - 64*x
sage: hyperplane_arrangements.Shi("A3", m=2).characteristic_polynomial()
x^4 - 24*x^3 + 192*x^2 - 512*x
sage: hyperplane_arrangements.Shi("C3").characteristic_polynomial()
x^3 - 18*x^2 + 108*x - 216
sage: hyperplane_arrangements.Shi("E6").characteristic_polynomial()
x^8 - 72*x^7 + 2160*x^6 - 34560*x^5 + 311040*x^4 - 1492992*x^3 + 2985984*x^2
sage: hyperplane_arrangements.Shi("B4", m=3).characteristic_polynomial()
x^4 - 96*x^3 + 3456*x^2 - 55296*x + 331776
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> hyperplane_arrangements.Shi("A3").characteristic_polynomial()
x^4 - 12*x^3 + 48*x^2 - 64*x
>>> hyperplane_arrangements.Shi("A3", m=Integer(2)).characteristic_
↪polynomial()
```

(continues on next page)

(continued from previous page)

```
x^4 - 24*x^3 + 192*x^2 - 512*x
>>> hyperplane_arrangements.Shi("C3").characteristic_polynomial()
x^3 - 18*x^2 + 108*x - 216
>>> hyperplane_arrangements.Shi("E6").characteristic_polynomial()
x^8 - 72*x^7 + 2160*x^6 - 34560*x^5 + 311040*x^4 - 1492992*x^3 + 2985984*x^2
>>> hyperplane_arrangements.Shi("B4", m=Integer(3)).characteristic_
↪polynomial()
x^4 - 96*x^3 + 3456*x^2 - 55296*x + 331776
```

bigraphical(*G*, *A=None*, *K=Rational Field*, *names=None*)

Return a bigraphical hyperplane arrangement.

INPUT:

- `G` – graph
- `A` – list, matrix, dictionary (default: `None` gives semiorder), or the string 'generic'
- `K` – field (default: $\mathbf{Q}$)
- `names` – tuple of strings or `None` (default); the variable names for the ambient space

OUTPUT:

The hyperplane arrangement with hyperplanes $x_i - x_j = A[i, j]$ and $x_j - x_i = A[j, i]$ for each edge v_i, v_j of `G`. The indices i, j are the indices of elements of `G.vertices()`.

EXAMPLES:

```
sage: # needs sage.graphs
sage: G = graphs.CycleGraph(4)
sage: G.edges(sort=True)
[(0, 1, None), (0, 3, None), (1, 2, None), (2, 3, None)]
sage: G.edges(sort=True, labels=False)
[(0, 1), (0, 3), (1, 2), (2, 3)]
sage: A = {0:{1:1, 3:2}, 1:{0:3, 2:0}, 2:{1:2, 3:1}, 3:{2:0, 0:2}}
sage: HA = hyperplane_arrangements.bigraphical(G, A)
sage: HA.n_regions()
63
sage: hyperplane_arrangements.bigraphical(G, # random
....:    'generic').n_regions()
65
sage: hyperplane_arrangements.bigraphical(G).n_regions()
59
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> G = graphs.CycleGraph(Integer(4))
>>> G.edges(sort=True)
[(0, 1, None), (0, 3, None), (1, 2, None), (2, 3, None)]
>>> G.edges(sort=True, labels=False)
[(0, 1), (0, 3), (1, 2), (2, 3)]
>>> A = {Integer(0):{Integer(1):Integer(1), Integer(3):Integer(2)},␣
↪Integer(1):{Integer(0):Integer(3), Integer(2):Integer(0)}, Integer(2):
↪{Integer(1):Integer(2), Integer(3):Integer(1)}, Integer(3):
↪{Integer(2):Integer(0), Integer(0):Integer(2)}}
```

(continues on next page)

(continued from previous page)

```
>>> HA = hyperplane_arrangements.bigraphical(G, A)
>>> HA.n_regions()
63
>>> hyperplane_arrangements.bigraphical(G, # random
...     'generic').n_regions()
65
>>> hyperplane_arrangements.bigraphical(G).n_regions()
59
```

REFERENCES:

- [HP2016]

braid(*n*, *K=Rational Field*, *names=None*)

The braid arrangement.

INPUT:

- `n` – integer
- `K` – field (default: `QQ`)
- `names` – tuple of strings or `None` (default); the variable names for the ambient space

OUTPUT:

The hyperplane arrangement consisting of the $n(n-1)/2$ hyperplanes $\{x_i - x_j = 0 : 1 \leq i \leq j \leq n\}$.

EXAMPLES:

```
sage: hyperplane_arrangements.braid(4)                                    #␣
↪needs sage.graphs
Arrangement of 6 hyperplanes of dimension 4 and rank 3
```

```
>>> from sage.all import *
>>> hyperplane_arrangements.braid(Integer(4))                             ␣
↪     # needs sage.graphs
Arrangement of 6 hyperplanes of dimension 4 and rank 3
```

coordinate(*n*, *K=Rational Field*, *names=None*)

Return the coordinate hyperplane arrangement.

INPUT:

- `n` – integer
- `K` – field (default: $\mathbf{Q}$)
- `names` – tuple of strings or `None` (default); the variable names for the ambient space

OUTPUT:

The coordinate hyperplane arrangement, which is the central hyperplane arrangement consisting of the coordinate hyperplanes $x_i = 0$.

EXAMPLES:

```
sage: hyperplane_arrangements.coordinate(5)
Arrangement of 5 hyperplanes of dimension 5 and rank 5
```

```
>>> from sage.all import *
>>> hyperplane_arrangements.coordinate(Integer(5))
Arrangement of 5 hyperplanes of dimension 5 and rank 5
```

graphical(*G*, *K=Rational Field*, *names=None*)

Return the graphical hyperplane arrangement of a graph `G`.

INPUT:

- `G` – graph
- `K` – field (default: $\mathbf{Q}$)
- `names` – tuple of strings or `None` (default); the variable names for the ambient space

OUTPUT:

The graphical hyperplane arrangement of a graph G, which is the arrangement $\{x_i - x_j = 0\}$ for all edges ij of the graph `G`.

EXAMPLES:

```
sage: # needs sage.graphs
sage: G = graphs.CompleteGraph(5)
sage: hyperplane_arrangements.graphical(G)
Arrangement of 10 hyperplanes of dimension 5 and rank 4
sage: g = graphs.HouseGraph()
sage: hyperplane_arrangements.graphical(g)
Arrangement of 6 hyperplanes of dimension 5 and rank 4
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> G = graphs.CompleteGraph(Integer(5))
>>> hyperplane_arrangements.graphical(G)
Arrangement of 10 hyperplanes of dimension 5 and rank 4
>>> g = graphs.HouseGraph()
>>> hyperplane_arrangements.graphical(g)
Arrangement of 6 hyperplanes of dimension 5 and rank 4
```

linial(*n*, *K=Rational Field*, *names=None*)

Return the linial hyperplane arrangement.

INPUT:

- `n` – integer
- `K` – field (default: $\mathbf{Q}$)
- `names` – tuple of strings or `None` (default); the variable names for the ambient space

OUTPUT:

The linial hyperplane arrangement is the set of hyperplanes $\{x_i - x_j = 1 : 1 \leq i < j \leq n\}$.

EXAMPLES:

```
sage: a = hyperplane_arrangements.linial(4);  a
Arrangement of 6 hyperplanes of dimension 4 and rank 3
sage: a.characteristic_polynomial()
x^4 - 6*x^3 + 15*x^2 - 14*x
```

```
>>> from sage.all import *
>>> a = hyperplane_arrangements.linial(Integer(4));  a
Arrangement of 6 hyperplanes of dimension 4 and rank 3
>>> a.characteristic_polynomial()
x^4 - 6*x^3 + 15*x^2 - 14*x
```

semiorder(*n*, *K=Rational Field*, *names=None*)

Return the semiorder arrangement.

INPUT:

- `n` – integer
- `K` – field (default: $\mathbf{Q}$)
- `names` – tuple of strings or `None` (default); the variable names for the ambient space

OUTPUT:

The semiorder arrangement, which is the set of $n(n-1)$ hyperplanes $\{x_i - x_j = -1, 1 : 1 \leq i \leq j \leq n\}$.

EXAMPLES:

```
sage: hyperplane_arrangements.semiorder(4)
Arrangement of 12 hyperplanes of dimension 4 and rank 3
```

```
>>> from sage.all import *
>>> hyperplane_arrangements.semiorder(Integer(4))
Arrangement of 12 hyperplanes of dimension 4 and rank 3
```

sage.geometry.hyperplane_arrangement.library.**make_parent**(*base_ring*, *dimension*, *names=None*)

Construct the parent for the hyperplane arrangements.

For internal use only.

INPUT:

- `base_ring` – a ring
- `dimension` – integer
- `names` – `None` (default) or a list/tuple/iterable of strings

OUTPUT:

A new *HyperplaneArrangements* instance.

EXAMPLES:

```
sage: from sage.geometry.hyperplane_arrangement.library import make_parent
sage: make_parent(QQ, 3)
Hyperplane arrangements in 3-dimensional linear space over
Rational Field with coordinates t0, t1, t2
```

```
>>> from sage.all import *
>>> from sage.geometry.hyperplane_arrangement.library import make_parent
>>> make_parent(QQ, Integer(3))
Hyperplane arrangements in 3-dimensional linear space over
Rational Field with coordinates t0, t1, t2
```

1.4 Hyperplanes

Note: If you want to learn about Sage's hyperplane arrangements then you should start with `sage.geometry.hyperplane_arrangement.arrangement`. This module is used to represent the individual hyperplanes, but you should never construct the classes from this module directly (but only via the `HyperplaneArrangements`.

A linear expression, for example, $3x + 3y - 5z - 7$ stands for the hyperplane with the equation $x + 3y - 5z = 7$. To create it in Sage, you first have to create a `HyperplaneArrangements` object to define the variables x, y, z:

```
sage: H.<x,y,z> = HyperplaneArrangements(QQ)
sage: h = 3*x + 2*y - 5*z - 7;  h
Hyperplane 3*x + 2*y - 5*z - 7
sage: h.coefficients()
[-7, 3, 2, -5]
sage: h.normal()
(3, 2, -5)
sage: h.constant_term()
-7
sage: h.change_ring(GF(3))
Hyperplane 0*x + 2*y + z + 2
sage: h.point()
(21/38, 7/19, -35/38)
sage: h.linear_part()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[   1   0 3/5]
[   0   1 2/5]
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,) = H._first_
↪ngens(3)
>>> h = Integer(3)*x + Integer(2)*y - Integer(5)*z - Integer(7);  h
Hyperplane 3*x + 2*y - 5*z - 7
>>> h.coefficients()
[-7, 3, 2, -5]
>>> h.normal()
(3, 2, -5)
>>> h.constant_term()
-7
>>> h.change_ring(GF(Integer(3)))
Hyperplane 0*x + 2*y + z + 2
>>> h.point()
(21/38, 7/19, -35/38)
>>> h.linear_part()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[   1   0 3/5]
[   0   1 2/5]
```

Another syntax to create hyperplanes is to specify coefficients and a constant term:

```
sage: V = H.ambient_space();  V
3-dimensional linear space over Rational Field with coordinates x, y, z
sage: h in V
True
sage: V([3, 2, -5], -7)
Hyperplane 3*x + 2*y - 5*z - 7
```

```
>>> from sage.all import *
>>> V = H.ambient_space();  V
3-dimensional linear space over Rational Field with coordinates x, y, z
>>> h in V
True
>>> V([Integer(3), Integer(2), -Integer(5)], -Integer(7))
Hyperplane 3*x + 2*y - 5*z - 7
```

Or constant term and coefficients together in one list/tuple/iterable:

```
sage: V([-7, 3, 2, -5])
Hyperplane 3*x + 2*y - 5*z - 7
sage: v = vector([-7, 3, 2, -5]);  v
(-7, 3, 2, -5)
sage: V(v)
Hyperplane 3*x + 2*y - 5*z - 7
```

```
>>> from sage.all import *
>>> V([-Integer(7), Integer(3), Integer(2), -Integer(5)])
Hyperplane 3*x + 2*y - 5*z - 7
>>> v = vector([-Integer(7), Integer(3), Integer(2), -Integer(5)]);  v
(-7, 3, 2, -5)
>>> V(v)
Hyperplane 3*x + 2*y - 5*z - 7
```

Note that the constant term comes first, which matches the notation for Sage's `Polyhedron()`

```
sage: Polyhedron(ieqs=[(4,1,2,3)]).Hrepresentation()
(An inequality (1, 2, 3) x + 4 >= 0,)
```

```
>>> from sage.all import *
>>> Polyhedron(ieqs=[(Integer(4),Integer(1),Integer(2),Integer(3))]).Hrepresentation()
(An inequality (1, 2, 3) x + 4 >= 0,)
```

The difference between hyperplanes as implemented in this module and hyperplane arrangements is that:

- hyperplane arrangements contain multiple hyperplanes (of course),
- linear expressions are a module over the base ring, and these module structure is inherited by the hyperplanes.

The latter means that you can add and multiply by a scalar:

```
sage: h = 3*x + 2*y - 5*z - 7;  h
Hyperplane 3*x + 2*y - 5*z - 7
sage: -h
Hyperplane -3*x - 2*y + 5*z + 7
sage: h + x
```

(continues on next page)

(continued from previous page)

```
Hyperplane 4*x + 2*y - 5*z - 7
sage: h + 7
Hyperplane 3*x + 2*y - 5*z + 0
sage: 3*h
Hyperplane 9*x + 6*y - 15*z - 21
sage: h * RDF(3)
Hyperplane 9.0*x + 6.0*y - 15.0*z - 21.0
```

```
>>> from sage.all import *
>>> h = Integer(3)*x + Integer(2)*y - Integer(5)*z - Integer(7);  h
Hyperplane 3*x + 2*y - 5*z - 7
>>> -h
Hyperplane -3*x - 2*y + 5*z + 7
>>> h + x
Hyperplane 4*x + 2*y - 5*z - 7
>>> h + Integer(7)
Hyperplane 3*x + 2*y - 5*z + 0
>>> Integer(3)*h
Hyperplane 9*x + 6*y - 15*z - 21
>>> h * RDF(Integer(3))
Hyperplane 9.0*x + 6.0*y - 15.0*z - 21.0
```

Which you can't do with hyperplane arrangements:

```
sage: arrangement = H(h, x, y, x+y-1);  arrangement
Arrangement <y | x | x + y - 1 | 3*x + 2*y - 5*z - 7>
sage: arrangement + x
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +:
'Hyperplane arrangements in 3-dimensional linear space
     over Rational Field with coordinates x, y, z' and
'Hyperplane arrangements in 3-dimensional linear space
     over Rational Field with coordinates x, y, z'
```

```
>>> from sage.all import *
>>> arrangement = H(h, x, y, x+y-Integer(1));  arrangement
Arrangement <y | x | x + y - 1 | 3*x + 2*y - 5*z - 7>
>>> arrangement + x
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +:
'Hyperplane arrangements in 3-dimensional linear space
     over Rational Field with coordinates x, y, z' and
'Hyperplane arrangements in 3-dimensional linear space
     over Rational Field with coordinates x, y, z'
```

class sage.geometry.hyperplane_arrangement.hyperplane.**AmbientVectorSpace**(*base_ring*, *names=()*)

Bases: *LinearExpressionModule*

The ambient space for hyperplanes.

This class is the parent for the *Hyperplane* instances.

Element

alias of *Hyperplane*

change_ring(*base_ring*)

Return a ambient vector space with a changed base ring.

INPUT:

- `base_ring` – a ring; the new base ring

OUTPUT: a new *AmbientVectorSpace*

EXAMPLES:

```
sage: M.<y> = HyperplaneArrangements(QQ)
sage: V = M.ambient_space()
sage: V.change_ring(RR)
1-dimensional linear space over Real Field with 53 bits of precision with␣
↪coordinate y
```

```
>>> from sage.all import *
>>> M = HyperplaneArrangements(QQ, names=('y',)); (y,) = M._first_ngens(1)
>>> V = M.ambient_space()
>>> V.change_ring(RR)
1-dimensional linear space over Real Field with 53 bits of precision with␣
↪coordinate y
```

dimension()

Return the ambient space dimension.

OUTPUT: integer

EXAMPLES:

```
sage: M.<x,y> = HyperplaneArrangements(QQ)
sage: x.parent().dimension()
2
sage: x.parent() is M.ambient_space()
True
sage: x.dimension()
1
```

```
>>> from sage.all import *
>>> M = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = M._first_
↪ngens(2)
>>> x.parent().dimension()
2
>>> x.parent() is M.ambient_space()
True
>>> x.dimension()
1
```

symmetric_space()

Construct the symmetric space of `self`.

Consider a hyperplane arrangement A in the vector space $V = k^n$, for some field k. The symmetric space is the symmetric algebra $S(V^*)$ as the polynomial ring $k[x_1, x_2, \ldots, x_n]$ where $(x_1, x_2, \ldots, x_n)$ is a basis for V.

EXAMPLES:

```
sage: H.<x,y,z> = HyperplaneArrangements(QQ)
sage: A = H.ambient_space()
sage: A.symmetric_space()
Multivariate Polynomial Ring in x, y, z over Rational Field
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,) = H._
↪first_ngens(3)
>>> A = H.ambient_space()
>>> A.symmetric_space()
Multivariate Polynomial Ring in x, y, z over Rational Field
```

class sage.geometry.hyperplane_arrangement.hyperplane.**Hyperplane**(*parent*, *coefficients*, *constant*)

Bases: *LinearExpression*

A hyperplane.

You should always use *AmbientVectorSpace* to construct instances of this class.

INPUT:

- parent – the parent *AmbientVectorSpace*
- coefficients – a vector of coefficients of the linear variables
- constant – the constant term for the linear expression

EXAMPLES:

```
sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: x+y-1
Hyperplane x + y - 1

sage: ambient = H.ambient_space()
sage: ambient._element_constructor_(x+y-1)
Hyperplane x + y - 1
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = H._first_ngens(2)
>>> x+y-Integer(1)
Hyperplane x + y - 1

>>> ambient = H.ambient_space()
>>> ambient._element_constructor_(x+y-Integer(1))
Hyperplane x + y - 1
```

For technical reasons, we must allow the degenerate cases of an empty space and of a full space:

```
sage: 0*x
Hyperplane 0*x + 0*y + 0
sage: 0*x + 1
```

(continues on next page)

(continued from previous page)

```
Hyperplane 0*x + 0*y + 1
sage: x + 0 == x + ambient(0)     # because coercion requires them
True
```

```
>>> from sage.all import *
>>> Integer(0)*x
Hyperplane 0*x + 0*y + 0
>>> Integer(0)*x + Integer(1)
Hyperplane 0*x + 0*y + 1
>>> x + Integer(0) == x + ambient(Integer(0))     # because coercion requires them
True
```

dimension()

The dimension of the hyperplane.

OUTPUT: integer

EXAMPLES:

```
sage: H.<x,y,z> = HyperplaneArrangements(QQ)
sage: h = x + y + z - 1
sage: h.dimension()
2
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,) = H._
↪first_ngens(3)
>>> h = x + y + z - Integer(1)
>>> h.dimension()
2
```

intersection(*other*)

The intersection of `self` with `other`.

INPUT:

- `other` – a hyperplane, a polyhedron, or something that defines a polyhedron

OUTPUT: a polyhedron

EXAMPLES:

```
sage: H.<x,y,z> = HyperplaneArrangements(QQ)
sage: h = x + y + z - 1
sage: h.intersection(x - y)
A 1-dimensional polyhedron in QQ^3 defined as the convex hull of 1 vertex and␣
↪1 line
sage: h.intersection(polytopes.cube())
A 2-dimensional polyhedron in QQ^3 defined as the convex hull of 3 vertices
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,) = H._
↪first_ngens(3)
>>> h = x + y + z - Integer(1)
```

(continues on next page)

(continued from previous page)

```
>>> h.intersection(x - y)
A 1-dimensional polyhedron in QQ^3 defined as the convex hull of 1 vertex and␣
↪1 line
>>> h.intersection(polytopes.cube())
A 2-dimensional polyhedron in QQ^3 defined as the convex hull of 3 vertices
```

linear_part()

The linear part of the affine space.

OUTPUT:

Vector subspace of the ambient vector space, parallel to the hyperplane.

EXAMPLES:

```
sage: H.<x,y,z> = HyperplaneArrangements(QQ)
sage: h = x + 2*y + 3*z - 1
sage: h.linear_part()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[   1    0 -1/3]
[   0    1 -2/3]
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,) = H._
↪first_ngens(3)
>>> h = x + Integer(2)*y + Integer(3)*z - Integer(1)
>>> h.linear_part()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[   1    0 -1/3]
[   0    1 -2/3]
```

linear_part_projection(*point*)

Orthogonal projection onto the linear part.

INPUT:

- `point` – vector of the ambient space, or anything that can be converted into one; not necessarily on the hyperplane

OUTPUT:

Coordinate vector of the projection of `point` with respect to the basis of *linear_part()*. In particular, the length of this vector is one less than the ambient space dimension.

EXAMPLES:

```
sage: H.<x,y,z> = HyperplaneArrangements(QQ)
sage: h = x + 2*y + 3*z - 4
sage: h.linear_part()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[   1    0 -1/3]
[   0    1 -2/3]
sage: p1 = h.linear_part_projection(0);  p1
```

(continues on next page)

(continued from previous page)

```
(0, 0)
sage: p2 = h.linear_part_projection([3,4,5]);  p2
(8/7, 2/7)
sage: h.linear_part().basis()
[(1, 0, -1/3), (0, 1, -2/3)]
sage: p3 = h.linear_part_projection([1,1,1]);  p3
(4/7, 1/7)
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,) = H._
↪first_ngens(3)
>>> h = x + Integer(2)*y + Integer(3)*z - Integer(4)
>>> h.linear_part()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[   1    0 -1/3]
[   0    1 -2/3]
>>> p1 = h.linear_part_projection(Integer(0));  p1
(0, 0)
>>> p2 = h.linear_part_projection([Integer(3),Integer(4),Integer(5)]);  p2
(8/7, 2/7)
>>> h.linear_part().basis()
[(1, 0, -1/3), (0, 1, -2/3)]
>>> p3 = h.linear_part_projection([Integer(1),Integer(1),Integer(1)]);  p3
(4/7, 1/7)
```

normal()

Return the normal vector.

OUTPUT: a vector over the base ring

EXAMPLES:

```
sage: H.<x, y, z> = HyperplaneArrangements(QQ)
sage: x.normal()
(1, 0, 0)
sage: x.A(), x.b()
((1, 0, 0), 0)
sage: (x + 2*y + 3*z + 4).normal()
(1, 2, 3)
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,) = H._
↪first_ngens(3)
>>> x.normal()
(1, 0, 0)
>>> x.A(), x.b()
((1, 0, 0), 0)
>>> (x + Integer(2)*y + Integer(3)*z + Integer(4)).normal()
(1, 2, 3)
```

orthogonal_projection(*point*)

Return the orthogonal projection of a point.

INPUT:

- `point` – vector of the ambient space, or anything that can be converted into one; not necessarily on the hyperplane

OUTPUT:

A vector in the ambient vector space that lies on the hyperplane.

In finite characteristic, a `ValueError` is raised if the the norm of the hyperplane normal is zero.

EXAMPLES:

```
sage: H.<x,y,z> = HyperplaneArrangements(QQ)
sage: h = x + 2*y + 3*z - 4
sage: p1 = h.orthogonal_projection(0);  p1
(2/7, 4/7, 6/7)
sage: p1 in h
True
sage: p2 = h.orthogonal_projection([3,4,5]);  p2
(10/7, 6/7, 2/7)
sage: p1 in h
True
sage: p3 = h.orthogonal_projection([1,1,1]);  p3
(6/7, 5/7, 4/7)
sage: p3 in h
True
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,) = H._
↪first_ngens(3)
>>> h = x + Integer(2)*y + Integer(3)*z - Integer(4)
>>> p1 = h.orthogonal_projection(Integer(0));  p1
(2/7, 4/7, 6/7)
>>> p1 in h
True
>>> p2 = h.orthogonal_projection([Integer(3),Integer(4),Integer(5)]);  p2
(10/7, 6/7, 2/7)
>>> p1 in h
True
>>> p3 = h.orthogonal_projection([Integer(1),Integer(1),Integer(1)]);  p3
(6/7, 5/7, 4/7)
>>> p3 in h
True
```

plot(***kwds*)

Plot the hyperplane.

OUTPUT: a graphics object

EXAMPLES:

```
sage: L.<x, y> = HyperplaneArrangements(QQ)
sage: (x + y - 2).plot()                                                    #␣
↪needs sage.plot
Graphics object consisting of 2 graphics primitives
```

```
>>> from sage.all import *
>>> L = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = L._first_
↪ngens(2)
>>> (x + y - Integer(2)).plot()                                                ␣
↪      # needs sage.plot
Graphics object consisting of 2 graphics primitives
```

point()

Return the point closest to the origin.

OUTPUT:

A vector of the ambient vector space. The closest point to the origin in the L^2-norm.

In finite characteristic a random point will be returned if the norm of the hyperplane normal vector is zero.

EXAMPLES:

```
sage: H.<x,y,z> = HyperplaneArrangements(QQ)
sage: h = x + 2*y + 3*z - 4
sage: h.point()
(2/7, 4/7, 6/7)
sage: h.point() in h
True

sage: # needs sage.rings.finite_rings
sage: H.<x,y,z> = HyperplaneArrangements(GF(3))
sage: h = 2*x + y + z + 1
sage: h.point()
(1, 0, 0)
sage: h.point().base_ring()
Finite Field of size 3

sage: H.<x,y,z> = HyperplaneArrangements(GF(3))
sage: h = x + y + z + 1
sage: h.point()
(2, 0, 0)
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,) = H._
↪first_ngens(3)
>>> h = x + Integer(2)*y + Integer(3)*z - Integer(4)
>>> h.point()
(2/7, 4/7, 6/7)
>>> h.point() in h
True

>>> # needs sage.rings.finite_rings
>>> H = HyperplaneArrangements(GF(Integer(3)), names=('x', 'y', 'z',)); (x, y,
↪ z,) = H._first_ngens(3)
>>> h = Integer(2)*x + y + z + Integer(1)
>>> h.point()
(1, 0, 0)
>>> h.point().base_ring()
```

(continues on next page)

(continued from previous page)

```
Finite Field of size 3

>>> H = HyperplaneArrangements(GF(Integer(3)), names=('x', 'y', 'z',)); (x, y,
↪ z,) = H._first_ngens(3)
>>> h = x + y + z + Integer(1)
>>> h.point()
(2, 0, 0)
```

polyhedron(***kwds*)

Return the hyperplane as a polyhedron.

OUTPUT: a *Polyhedron()* instance

EXAMPLES:

```
sage: H.<x,y,z> = HyperplaneArrangements(QQ)
sage: h = x + 2*y + 3*z - 4
sage: P = h.polyhedron();  P
A 2-dimensional polyhedron in QQ^3 defined as the convex hull of 1 vertex and␣
↪2 lines
sage: P.Hrepresentation()
(An equation (1, 2, 3) x - 4 == 0,)
sage: P.Vrepresentation()
(A line in the direction (0, 3, -2),
 A line in the direction (3, 0, -1),
 A vertex at (0, 0, 4/3))
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,) = H._
↪first_ngens(3)
>>> h = x + Integer(2)*y + Integer(3)*z - Integer(4)
>>> P = h.polyhedron();  P
A 2-dimensional polyhedron in QQ^3 defined as the convex hull of 1 vertex and␣
↪2 lines
>>> P.Hrepresentation()
(An equation (1, 2, 3) x - 4 == 0,)
>>> P.Vrepresentation()
(A line in the direction (0, 3, -2),
 A line in the direction (3, 0, -1),
 A vertex at (0, 0, 4/3))
```

primitive(*signed=True*)

Return hyperplane defined by primitive equation.

INPUT:

- `signed` – boolean (default: `True`); whether to preserve the overall sign

OUTPUT:

Hyperplane whose linear expression has common factors and denominators cleared. That is, the same hyperplane (with the same sign) but defined by a rescaled equation. Note that different linear expressions must define different hyperplanes as comparison is used in caching.

If `signed`, the overall rescaling is by a positive constant only.

EXAMPLES:

```
sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: h = -1/3*x + 1/2*y - 1;  h
Hyperplane -1/3*x + 1/2*y - 1
sage: h.primitive()
Hyperplane -2*x + 3*y - 6
sage: h == h.primitive()
False
sage: (4*x + 8).primitive()
Hyperplane x + 0*y + 2

sage: (4*x - y - 8).primitive(signed=True)    # default
Hyperplane 4*x - y - 8
sage: (4*x - y - 8).primitive(signed=False)
Hyperplane -4*x + y + 8
```

```
>>> from sage.all import *
>>> H = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = H._first_
↪ngens(2)
>>> h = -Integer(1)/Integer(3)*x + Integer(1)/Integer(2)*y - Integer(1);  h
Hyperplane -1/3*x + 1/2*y - 1
>>> h.primitive()
Hyperplane -2*x + 3*y - 6
>>> h == h.primitive()
False
>>> (Integer(4)*x + Integer(8)).primitive()
Hyperplane x + 0*y + 2

>>> (Integer(4)*x - y - Integer(8)).primitive(signed=True)    # default
Hyperplane 4*x - y - 8
>>> (Integer(4)*x - y - Integer(8)).primitive(signed=False)
Hyperplane -4*x + y + 8
```

to_symmetric_space()

Return `self` considered as an element in the corresponding symmetric space.

EXAMPLES:

```
sage: L.<x, y> = HyperplaneArrangements(QQ)
sage: h = -1/3*x + 1/2*y
sage: h.to_symmetric_space()
-1/3*x + 1/2*y

sage: hp = -1/3*x + 1/2*y - 1
sage: hp.to_symmetric_space()
Traceback (most recent call last):
...
ValueError: the hyperplane must pass through the origin
```

```
>>> from sage.all import *
>>> L = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = L._first_
↪ngens(2)
```

(continues on next page)

(continued from previous page)

```
>>> h = -Integer(1)/Integer(3)*x + Integer(1)/Integer(2)*y
>>> h.to_symmetric_space()
-1/3*x + 1/2*y

>>> hp = -Integer(1)/Integer(3)*x + Integer(1)/Integer(2)*y - Integer(1)
>>> hp.to_symmetric_space()
Traceback (most recent call last):
...
ValueError: the hyperplane must pass through the origin
```

1.5 Affine Subspaces of a Vector Space

An affine subspace of a vector space is a translation of a linear subspace. The affine subspaces here are only used internally in hyperplane arrangements. You should not use them for interactive work or return them to the user.

EXAMPLES:

```
sage: from sage.geometry.hyperplane_arrangement.affine_subspace import AffineSubspace
sage: a = AffineSubspace([1,0,0,0], QQ^4)
sage: a.dimension()
4
sage: a.point()
(1, 0, 0, 0)
sage: a.linear_part()
Vector space of dimension 4 over Rational Field
sage: a
Affine space p + W where:
  p = (1, 0, 0, 0)
  W = Vector space of dimension 4 over Rational Field
sage: b = AffineSubspace((1,0,0,0), matrix(QQ, [[1,2,3,4]]).right_kernel())
sage: c = AffineSubspace((0,2,0,0), matrix(QQ, [[0,0,1,2]]).right_kernel())
sage: b.intersection(c)
Affine space p + W where:
  p = (-3, 2, 0, 0)
  W = Vector space of degree 4 and dimension 2 over Rational Field
Basis matrix:
[  1   0  -1 1/2]
[  0   1  -2   1]
sage: b < a
True
sage: c < b
False
sage: A = AffineSubspace([8,38,21,250], VectorSpace(GF(19),4))
sage: A
Affine space p + W where:
   p = (8, 0, 2, 3)
   W = Vector space of dimension 4 over Finite Field of size 19
```

```
>>> from sage.all import *
>>> from sage.geometry.hyperplane_arrangement.affine_subspace import AffineSubspace
>>> a = AffineSubspace([Integer(1),Integer(0),Integer(0),Integer(0)], QQ**Integer(4))
```

(continues on next page)

(continued from previous page)

```
>>> a.dimension()
4
>>> a.point()
(1, 0, 0, 0)
>>> a.linear_part()
Vector space of dimension 4 over Rational Field
>>> a
Affine space p + W where:
  p = (1, 0, 0, 0)
  W = Vector space of dimension 4 over Rational Field
>>> b = AffineSubspace((Integer(1),Integer(0),Integer(0),Integer(0)), matrix(QQ,␣
↪[[Integer(1),Integer(2),Integer(3),Integer(4)]]).right_kernel())
>>> c = AffineSubspace((Integer(0),Integer(2),Integer(0),Integer(0)), matrix(QQ,␣
↪[[Integer(0),Integer(0),Integer(1),Integer(2)]]).right_kernel())
>>> b.intersection(c)
Affine space p + W where:
  p = (-3, 2, 0, 0)
  W = Vector space of degree 4 and dimension 2 over Rational Field
Basis matrix:
[   1    0   -1 1/2]
[   0    1   -2    1]
>>> b < a
True
>>> c < b
False
>>> A = AffineSubspace([Integer(8),Integer(38),Integer(21),Integer(250)],␣
↪VectorSpace(GF(Integer(19)),Integer(4)))
>>> A
Affine space p + W where:
   p = (8, 0, 2, 3)
   W = Vector space of dimension 4 over Finite Field of size 19
```

class sage.geometry.hyperplane_arrangement.affine_subspace.**AffineSubspace**(p, V)

Bases: SageObject

An affine subspace.

INPUT:

- p – list/tuple/iterable representing a point on the affine space
- V – vector subspace

OUTPUT: affine subspace parallel to V and passing through p

EXAMPLES:

```
sage: from sage.geometry.hyperplane_arrangement.affine_subspace import␣
↪AffineSubspace
sage: a = AffineSubspace([1,0,0,0], VectorSpace(QQ,4))
sage: a
Affine space p + W where:
  p = (1, 0, 0, 0)
  W = Vector space of dimension 4 over Rational Field
```

```
>>> from sage.all import *
>>> from sage.geometry.hyperplane_arrangement.affine_subspace import␣
↪AffineSubspace
>>> a = AffineSubspace([Integer(1),Integer(0),Integer(0),Integer(0)],␣
↪VectorSpace(QQ,Integer(4)))
>>> a
Affine space p + W where:
  p = (1, 0, 0, 0)
  W = Vector space of dimension 4 over Rational Field
```

dimension()

Return the dimension of the affine space.

OUTPUT: integer

EXAMPLES:

```
sage: from sage.geometry.hyperplane_arrangement.affine_subspace import␣
↪AffineSubspace
sage: a = AffineSubspace([1,0,0,0],VectorSpace(QQ,4))
sage: a.dimension()
4
```

```
>>> from sage.all import *
>>> from sage.geometry.hyperplane_arrangement.affine_subspace import␣
↪AffineSubspace
>>> a = AffineSubspace([Integer(1),Integer(0),Integer(0),Integer(0)],
↪VectorSpace(QQ,Integer(4)))
>>> a.dimension()
4
```

intersection(*other*)

Return the intersection of `self` with `other`.

INPUT:

- `other` – an *AffineSubspace*

OUTPUT:

A new affine subspace, (or `None` if the intersection is empty).

EXAMPLES:

```
sage: from sage.geometry.hyperplane_arrangement.affine_subspace import␣
↪AffineSubspace
sage: V = VectorSpace(QQ,3)
sage: U = V.subspace([(1,0,0), (0,1,0)])
sage: W = V.subspace([(0,1,0), (0,0,1)])
sage: A = AffineSubspace((0,0,0), U)
sage: B = AffineSubspace((1,1,1), W)
sage: A.intersection(B)
Affine space p + W where:
  p = (1, 1, 0)
  W = Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
```

(continues on next page)

(continued from previous page)

```
[0 1 0]
sage: C = AffineSubspace((0,0,1), U)
sage: A.intersection(C)
sage: C = AffineSubspace((7,8,9), U.complement())
sage: A.intersection(C)
Affine space p + W where:
  p = (7, 8, 0)
  W = Vector space of degree 3 and dimension 0 over Rational Field
Basis matrix:
[]
sage: A.intersection(C).intersection(B)

sage: D = AffineSubspace([1,2,3], VectorSpace(GF(5),3))
sage: E = AffineSubspace([3,4,5], VectorSpace(GF(5),3))
sage: D.intersection(E)
Affine space p + W where:
  p = (3, 4, 0)
  W = Vector space of dimension 3 over Finite Field of size 5
```

```
>>> from sage.all import *
>>> from sage.geometry.hyperplane_arrangement.affine_subspace import␣
↪AffineSubspace
>>> V = VectorSpace(QQ,Integer(3))
>>> U = V.subspace([(Integer(1),Integer(0),Integer(0)), (Integer(0),
↪Integer(1),Integer(0))])
>>> W = V.subspace([(Integer(0),Integer(1),Integer(0)), (Integer(0),
↪Integer(0),Integer(1))])
>>> A = AffineSubspace((Integer(0),Integer(0),Integer(0)), U)
>>> B = AffineSubspace((Integer(1),Integer(1),Integer(1)), W)
>>> A.intersection(B)
Affine space p + W where:
  p = (1, 1, 0)
  W = Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[0 1 0]
>>> C = AffineSubspace((Integer(0),Integer(0),Integer(1)), U)
>>> A.intersection(C)
>>> C = AffineSubspace((Integer(7),Integer(8),Integer(9)), U.complement())
>>> A.intersection(C)
Affine space p + W where:
  p = (7, 8, 0)
  W = Vector space of degree 3 and dimension 0 over Rational Field
Basis matrix:
[]
>>> A.intersection(C).intersection(B)

>>> D = AffineSubspace([Integer(1),Integer(2),Integer(3)],␣
↪VectorSpace(GF(Integer(5)),Integer(3)))
>>> E = AffineSubspace([Integer(3),Integer(4),Integer(5)],␣
↪VectorSpace(GF(Integer(5)),Integer(3)))
>>> D.intersection(E)
Affine space p + W where:
```

(continues on next page)

(continued from previous page)

```
p = (3, 4, 0)
W = Vector space of dimension 3 over Finite Field of size 5
```

linear_part()

Return the linear part of the affine space.

OUTPUT: a vector subspace of the ambient space

EXAMPLES:

```
sage: from sage.geometry.hyperplane_arrangement.affine_subspace import␣
↪AffineSubspace
sage: A = AffineSubspace([2,3,1], matrix(QQ, [[1,2,3]]).right_kernel())
sage: A.linear_part()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[   1    0 -1/3]
[   0    1 -2/3]
sage: A.linear_part().ambient_vector_space()
Vector space of dimension 3 over Rational Field
```

```
>>> from sage.all import *
>>> from sage.geometry.hyperplane_arrangement.affine_subspace import␣
↪AffineSubspace
>>> A = AffineSubspace([Integer(2),Integer(3),Integer(1)], matrix(QQ,␣
↪[[Integer(1),Integer(2),Integer(3)]]).right_kernel())
>>> A.linear_part()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[   1    0 -1/3]
[   0    1 -2/3]
>>> A.linear_part().ambient_vector_space()
Vector space of dimension 3 over Rational Field
```

point()

Return a point `p` in the affine space.

OUTPUT: a point of the affine space as a vector in the ambient space

EXAMPLES:

```
sage: from sage.geometry.hyperplane_arrangement.affine_subspace import␣
↪AffineSubspace
sage: A = AffineSubspace([2,3,1], VectorSpace(QQ,3))
sage: A.point()
(2, 3, 1)
```

```
>>> from sage.all import *
>>> from sage.geometry.hyperplane_arrangement.affine_subspace import␣
↪AffineSubspace
>>> A = AffineSubspace([Integer(2),Integer(3),Integer(1)], VectorSpace(QQ,
↪Integer(3)))
>>> A.point()
(2, 3, 1)
```

1.6 Plotting of Hyperplane Arrangements

PLOT OPTIONS:

Beside the usual plot options (enter `plot?`), the plot command for hyperplane arrangements includes the following:

- `hyperplane_colors` – color or list of colors, one for each hyperplane (default: equally spread range of hues)
- `hyperplane_labels` – boolean, `'short'`, `'long'` (default: `False`). If `False`, no labels are shown; if 'short' or 'long', the hyperplanes are given short or long labels, respectively. If `True`, the hyperplanes are given long labels.
- `label_colors` – color or list of colors, one for each hyperplane (default: black)
- `label_fontsize` – size for hyperplane_label font (default: `14`); this does not work for 3d plots
- `label_offsets` – amount be which labels are offset from h.point() for each hyperplane h. The format is different for each dimension: if the hyperplanes have dimension 0, the offset can be a single number or a list of numbers, one for each hyperplane; if the hyperplanes have dimension 1, the offset can be a single 2-tuple, or a list of 2-tuples, one for each hyperplane; if the hyperplanes have dimension 2, the offset can be a single 3-tuple or a list of 3-tuples, one for each hyperplane. (Defaults: 0-dim: `0.1`, 1-dim: `(0,1)`, 2-dim: `(0,0,0.2)`).
- `hyperplane_legend` – boolean, `'short'`, `'long'` (default: `'long'`; in 3-d: `False`). If `False`, no legend is shown; if `True`, `'short'`, or `'long'`, the legend is shown with the default, long, or short labeling, respectively. (For arrangements of lines or planes, only.)
- `hyperplane_opacities` – a number or list of numbers, one for each hyperplane, between 0 and 1; only applies to 3d plots
- `point_sizes` – number or list of numbers, one for each hyperplane giving the sizes of points in a zero-dimensional arrangement (default: `50`)
- `ranges` – range for the parameters or a list of ranges of parameters, one for each hyperplane, for the parametric plots of the hyperplanes. If a single positive number r is given for `ranges`, then all parameters run from -r to r. Otherwise, for a line in the plane, the range has the form `[a,b]` (default: `[-3,3]`), and for a plane in 3-space, the range has the form `[[a,b],[c,d]]` (default: `[[-3,3],[-3,3]]`). The ranges are centered around `hyperplane_arrangement.point()`.

EXAMPLES:

```
sage: H3.<x,y,z> = HyperplaneArrangements(QQ)
sage: A = H3([(1,0,0), 0], [(0,0,1), 5])
sage: A.plot(hyperplane_opacities=0.5, hyperplane_labels=True,                #␣
↪needs sage.plot
....:        hyperplane_legend=False)
Graphics3d Object

sage: c = H3([(1,0,0),0], [(0,0,1),5])
sage: c.plot(ranges=10)                                                       #␣
↪needs sage.plot
Graphics3d Object
sage: c.plot(ranges=[[9.5,10], [-3,3]])                                       #␣
↪needs sage.plot
Graphics3d Object
sage: c.plot(ranges=[[[9.5,10], [-3,3]], [[-6,6], [-5,5]]])                   #␣
↪needs sage.plot
Graphics3d Object


sage: H2.<s,t> = HyperplaneArrangements(QQ)
```

(continues on next page)

(continued from previous page)

```
sage: h = H2([(1,1),0], [(1,-1),0], [(0,1),2])
sage: h.plot(ranges=20)                                                     #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
sage: h.plot(ranges=[-1, 10])                                               #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
sage: h.plot(ranges=[[-1, 1], [-5, 5], [-1, 10]])                           #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives

sage: a = hyperplane_arrangements.coordinate(3)
sage: opts = {'hyperplane_colors':['yellow', 'green', 'blue']}
sage: opts['hyperplane_labels'] = True
sage: opts['label_offsets'] = [(0,2,2), (2,0,2), (2,2,0)]
sage: opts['hyperplane_legend'] = False
sage: opts['hyperplane_opacities'] = 0.7
sage: a.plot(**opts)                                                        #␣
↪needs sage.plot
Graphics3d Object
sage: opts['hyperplane_labels'] = 'short'
sage: a.plot(**opts)                                                        #␣
↪needs sage.plot
Graphics3d Object

sage: H.<u> = HyperplaneArrangements(QQ)
sage: pts = H(3*u+4, 2*u+5, 7*u+1)
sage: pts.plot(hyperplane_colors=['yellow','black','blue'])                 #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
sage: pts.plot(point_sizes=[50,100,200], hyperplane_colors='blue')          #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives

sage: H.<x,y,z> = HyperplaneArrangements(QQ)
sage: a = H(x, y+1, y+2)
sage: a.plot(hyperplane_labels=True, label_colors='blue', label_fontsize=18)    #␣
↪needs sage.plot
Graphics3d Object
sage: a.plot(hyperplane_labels=True, label_colors=['red','green','black'])  #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> H3 = HyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,) = H3._first_
↪ngens(3)
>>> A = H3([(Integer(1),Integer(0),Integer(0)), Integer(0)], [(Integer(0),Integer(0),
↪Integer(1)), Integer(5)])
>>> A.plot(hyperplane_opacities=RealNumber('0.5'), hyperplane_labels=True,       ␣
↪           # needs sage.plot
...         hyperplane_legend=False)
Graphics3d Object
```

(continues on next page)

(continued from previous page)

```
>>> c = H3([(Integer(1),Integer(0),Integer(0)),Integer(0)], [(Integer(0),Integer(0),
↪Integer(1)),Integer(5)])
>>> c.plot(ranges=Integer(10))                                                      ␣
↪     # needs sage.plot
Graphics3d Object
>>> c.plot(ranges=[[RealNumber('9.5'),Integer(10)], [-Integer(3),Integer(3)]])      ␣
↪                                  # needs sage.plot
Graphics3d Object
>>> c.plot(ranges=[[[RealNumber('9.5'),Integer(10)], [-Integer(3),Integer(3)]], [[-
↪Integer(6),Integer(6)], [-Integer(5),Integer(5)]]])                        # needs␣
↪sage.plot
Graphics3d Object


>>> H2 = HyperplaneArrangements(QQ, names=('s', 't',)); (s, t,) = H2._first_ngens(2)
>>> h = H2([(Integer(1),Integer(1)),Integer(0)], [(Integer(1),-Integer(1)),
↪Integer(0)], [(Integer(0),Integer(1)),Integer(2)])
>>> h.plot(ranges=Integer(20))                                                      ␣
↪     # needs sage.plot
Graphics object consisting of 3 graphics primitives
>>> h.plot(ranges=[-Integer(1), Integer(10)])                                       ␣
↪              # needs sage.plot
Graphics object consisting of 3 graphics primitives
>>> h.plot(ranges=[[-Integer(1), Integer(1)], [-Integer(5), Integer(5)], [-Integer(1),
↪ Integer(10)]])                                # needs sage.plot
Graphics object consisting of 3 graphics primitives

>>> a = hyperplane_arrangements.coordinate(Integer(3))
>>> opts = {'hyperplane_colors':['yellow', 'green', 'blue']}
>>> opts['hyperplane_labels'] = True
>>> opts['label_offsets'] = [(Integer(0),Integer(2),Integer(2)), (Integer(2),
↪Integer(0),Integer(2)), (Integer(2),Integer(2),Integer(0))]
>>> opts['hyperplane_legend'] = False
>>> opts['hyperplane_opacities'] = RealNumber('0.7')
>>> a.plot(**opts)                                                                #␣
↪needs sage.plot
Graphics3d Object
>>> opts['hyperplane_labels'] = 'short'
>>> a.plot(**opts)                                                                #␣
↪needs sage.plot
Graphics3d Object

>>> H = HyperplaneArrangements(QQ, names=('u',)); (u,) = H._first_ngens(1)
>>> pts = H(Integer(3)*u+Integer(4), Integer(2)*u+Integer(5), Integer(7)*u+Integer(1))
>>> pts.plot(hyperplane_colors=['yellow','black','blue'])                         #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
>>> pts.plot(point_sizes=[Integer(50),Integer(100),Integer(200)], hyperplane_colors=
↪'blue')                    # needs sage.plot
Graphics object consisting of 3 graphics primitives
```

(continues on next page)

(continued from previous page)

```
>>> H = HyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,) = H._first_
↪ngens(3)
>>> a = H(x, y+Integer(1), y+Integer(2))
>>> a.plot(hyperplane_labels=True, label_colors='blue', label_fontsize=Integer(18))  ␣
↪     # needs sage.plot
Graphics3d Object
>>> a.plot(hyperplane_labels=True, label_colors=['red','green','black'])          #␣
↪needs sage.plot
Graphics3d Object
```

sage.geometry.hyperplane_arrangement.plot.**legend_3d**(*hyperplane_arrangement*, *hyperplane_colors*, *length*)

Create plot of a 3d legend for an arrangement of planes in 3-space.

The `length` parameter determines whether short or long labels are used in the legend.

INPUT:

- `hyperplane_arrangement` – a hyperplane arrangement
- `hyperplane_colors` – list of colors
- `length` – either `'short'` or `'long'`

OUTPUT: a graphics object

EXAMPLES:

```
sage: a = hyperplane_arrangements.semiorder(3)
sage: from sage.geometry.hyperplane_arrangement.plot import legend_3d
sage: legend_3d(a, list(colors.values())[:6], length='long')                      #␣
↪needs sage.combinat sage.plot
Graphics object consisting of 6 graphics primitives

sage: b = hyperplane_arrangements.semiorder(4)
sage: c = b.essentialization()
sage: legend_3d(c, list(colors.values())[:12], length='long')                     #␣
↪needs sage.combinat sage.plot
Graphics object consisting of 12 graphics primitives

sage: legend_3d(c, list(colors.values())[:12], length='short')                    #␣
↪needs sage.combinat sage.plot
Graphics object consisting of 12 graphics primitives

sage: p = legend_3d(c, list(colors.values())[:12], length='short')                #␣
↪needs sage.combinat sage.plot
sage: p.set_legend_options(ncol=4)                                                 #␣
↪needs sage.combinat sage.plot
sage: type(p)                                                                      #␣
↪needs sage.combinat sage.plot
<class 'sage.plot.graphics.Graphics'>
```

```
>>> from sage.all import *
>>> a = hyperplane_arrangements.semiorder(Integer(3))
>>> from sage.geometry.hyperplane_arrangement.plot import legend_3d
```

(continues on next page)

(continued from previous page)

```
>>> legend_3d(a, list(colors.values())[:Integer(6)], length='long')
↪     # needs sage.combinat sage.plot
Graphics object consisting of 6 graphics primitives

>>> b = hyperplane_arrangements.semiorder(Integer(4))
>>> c = b.essentialization()
>>> legend_3d(c, list(colors.values())[:Integer(12)], length='long')
↪     # needs sage.combinat sage.plot
Graphics object consisting of 12 graphics primitives

>>> legend_3d(c, list(colors.values())[:Integer(12)], length='short')
↪     # needs sage.combinat sage.plot
Graphics object consisting of 12 graphics primitives

>>> p = legend_3d(c, list(colors.values())[:Integer(12)], length='short')
↪     # needs sage.combinat sage.plot
>>> p.set_legend_options(ncol=Integer(4))
↪     # needs sage.combinat sage.plot
>>> type(p)                                                              #
↪needs sage.combinat sage.plot
<class 'sage.plot.graphics.Graphics'>
```

`sage.geometry.hyperplane_arrangement.plot.plot`(*hyperplane_arrangement*, ***kwds*)

Return a plot of the hyperplane arrangement.

If the arrangement is in 4 dimensions but inessential, a plot of the essentialization is returned.

Note

This function is available as the `plot()` method of hyperplane arrangements. You should not call this function directly, only through the method.

INPUT:

- `hyperplane_arrangement` – the hyperplane arrangement to plot
- `**kwds` – plot options: see `sage.geometry.hyperplane_arrangement.plot`

OUTPUT: a graphics object of the plot

EXAMPLES:

```
sage: B = hyperplane_arrangements.semiorder(4)
sage: B.plot()                                                           #
↪needs sage.combinat sage.plot
Displaying the essentialization.
Graphics3d Object
```

```
>>> from sage.all import *
>>> B = hyperplane_arrangements.semiorder(Integer(4))
>>> B.plot()                                                             #
↪needs sage.combinat sage.plot
```

(continues on next page)

(continued from previous page)

```
Displaying the essentialization.
Graphics3d Object
```

`sage.geometry.hyperplane_arrangement.plot.`**`plot_hyperplane`**`(`*hyperplane*, ***kwds*`)`

Return the plot of a single hyperplane.

INPUT:

- `**kwds` – plot options: see below

OUTPUT: a graphics object of the plot

Plot Options

Beside the usual plot options (enter `plot?`), the plot command for hyperplanes includes the following:

- `hyperplane_label` – boolean value or string (default: `True`); if `True`, the hyperplane is labeled with its equation, if a string, it is labeled by that string, otherwise it is not labeled
- `label_color` – (default: `'black'`) color for hyperplane_label
- `label_fontsize` – size for `hyperplane_label` font (default: 14) (does not work in 3d, yet)
- `label_offset` – (default: 0-dim: 0.1, 1-dim: (0,1), 2-dim: (0,0,0.2)); amount by which label is offset from `hyperplane.point()`
- `point_size` – (default: 50) size of points in a zero-dimensional arrangement or of an arrangement over a finite field
- `ranges` – range for the parameters for the parametric plot of the hyperplane. If a single positive number `r` is given for the value of `ranges`, then the ranges for all parameters are set to $[-r, r]$. Otherwise, for a line in the plane, `ranges` has the form `[a, b]` (default: [-3,3]), and for a plane in 3-space, the `ranges` has the form `[[a, b], [c, d]]` (default: [[-3,3],[-3,3]]). (The ranges are centered around `hyperplane.point()`.)

EXAMPLES:

```
sage: H1.<x> = HyperplaneArrangements(QQ)
sage: a = 3*x + 4
sage: a.plot()     # indirect doctest                                          #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
sage: a.plot(point_size=100, hyperplane_label='hello')                         #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives

sage: H2.<x,y> = HyperplaneArrangements(QQ)
sage: b = 3*x + 4*y + 5
sage: b.plot()                                                                 #␣
↪needs sage.plot
Graphics object consisting of 2 graphics primitives
sage: b.plot(ranges=(1,5), label_offset=(2,-1))                                #␣
↪needs sage.plot
Graphics object consisting of 2 graphics primitives
sage: opts = {'hyperplane_label': True, 'label_color': 'green',
....:         'label_fontsize': 24, 'label_offset': (0,1.5)}
sage: b.plot(**opts)                                                           #␣
↪needs sage.plot
```

(continues on next page)

(continued from previous page)

```
Graphics object consisting of 2 graphics primitives

sage: # needs sage.plot
sage: H3.<x,y,z> = HyperplaneArrangements(QQ)
sage: c = 2*x + 3*y + 4*z + 5
sage: c.plot()
Graphics3d Object
sage: c.plot(label_offset=(1,0,1), color='green', label_color='red',
....:        frame=False)
Graphics3d Object
sage: d = -3*x + 2*y + 2*z + 3
sage: d.plot(opacity=0.8)
Graphics3d Object
sage: e = 4*x + 2*z + 3
sage: e.plot(ranges=[[-1,1],[0,8]], label_offset=(2,2,1), aspect_ratio=1)
Graphics3d Object
```

```
>>> from sage.all import *
>>> H1 = HyperplaneArrangements(QQ, names=('x',)); (x,) = H1._first_ngens(1)
>>> a = Integer(3)*x + Integer(4)
>>> a.plot()     # indirect doctest                                          #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
>>> a.plot(point_size=Integer(100), hyperplane_label='hello')               ␣
↪     # needs sage.plot
Graphics object consisting of 3 graphics primitives

>>> H2 = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = H2._first_
↪ngens(2)
>>> b = Integer(3)*x + Integer(4)*y + Integer(5)
>>> b.plot()                                                                 #␣
↪needs sage.plot
Graphics object consisting of 2 graphics primitives
>>> b.plot(ranges=(Integer(1),Integer(5)), label_offset=(Integer(2),-Integer(1)))␣
↪                                 # needs sage.plot
Graphics object consisting of 2 graphics primitives
>>> opts = {'hyperplane_label': True, 'label_color': 'green',
...         'label_fontsize': Integer(24), 'label_offset': (Integer(0),RealNumber(
↪'1.5'))}
>>> b.plot(**opts)                                                           #␣
↪needs sage.plot
Graphics object consisting of 2 graphics primitives

>>> # needs sage.plot
>>> H3 = HyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,) = H3._
↪first_ngens(3)
>>> c = Integer(2)*x + Integer(3)*y + Integer(4)*z + Integer(5)
>>> c.plot()
Graphics3d Object
>>> c.plot(label_offset=(Integer(1),Integer(0),Integer(1)), color='green', label_
↪color='red',
...        frame=False)
```

(continues on next page)

(continued from previous page)

```
Graphics3d Object
>>> d = -Integer(3)*x + Integer(2)*y + Integer(2)*z + Integer(3)
>>> d.plot(opacity=RealNumber('0.8'))
Graphics3d Object
>>> e = Integer(4)*x + Integer(2)*z + Integer(3)
>>> e.plot(ranges=[[-Integer(1),Integer(1)],[Integer(0),Integer(8)]], label_
↪offset=(Integer(2),Integer(2),Integer(1)), aspect_ratio=Integer(1))
Graphics3d Object
```

CHAPTER

TWO

POLYHEDRAL COMPUTATIONS

2.1 Polyhedra

2.1.1 Library of commonly used, famous, or interesting polytopes

This module gathers several constructors of polytopes that can be reached through `polytopes.<tab>`. For example, here is the hypercube in dimension 5:

```
sage: polytopes.hypercube(5)
A 5-dimensional polyhedron in ZZ^5 defined as the convex hull of 32 vertices
```

```
>>> from sage.all import *
>>> polytopes.hypercube(Integer(5))
A 5-dimensional polyhedron in ZZ^5 defined as the convex hull of 32 vertices
```

The following constructions are available

Birkhoff_polytope()
associahedron()
bitruncated_six_hundred_cell()
buckyball()
cantellated_one_hundred_twenty_cell()
cantellated_six_hundred_cell()
cantitruncated_one_hundred_twenty_cell()
cantitruncated_six_hundred_cell()
cross_polytope()
cube()
cuboctahedron()
cyclic_polytope()
dodecahedron()
flow_polytope()
Gosset_3_21()
grand_antiprism()
great_rhombicuboctahedron()
hypercube()
hypersimplex()
icosahedron()
icosidodecahedron()
Kirkman_icosahedron()
octahedron()

continues on next page

Table 1 – continued from previous page

omnitruncated_one_hundred_twenty_cell()
omnitruncated_six_hundred_cell()
one_hundred_twenty_cell()
parallelotope()
pentakis_dodecahedron()
permutahedron()
generalized_permutahedron()
rectified_one_hundred_twenty_cell()
rectified_six_hundred_cell()
regular_polygon()
rhombic_dodecahedron()
rhombicosidodecahedron()
runcinated_one_hundred_twenty_cell()
runcitruncated_one_hundred_twenty_cell()
runcitruncated_six_hundred_cell()
simplex()
six_hundred_cell()
small_rhombicuboctahedron()
snub_cube()
snub_dodecahedron()
tetrahedron()
truncated_cube()
truncated_dodecahedron()
truncated_icosidodecahedron()
truncated_tetrahedron()
truncated_octahedron()
truncated_one_hundred_twenty_cell()
truncated_six_hundred_cell()
twenty_four_cell()

class `sage.geometry.polyhedron.library.`**`Polytopes`**

Bases: `object`

A class of constructors for commonly used, famous, or interesting polytopes.

`Birkhoff_polytope`(*n*, *backend=None*)

Return the Birkhoff polytope with $n!$ vertices.

The vertices of this polyhedron are the (flattened) n by n permutation matrices. So the ambient vector space has dimension n^2 but the dimension of the polyhedron is $(n-1)^2$.

INPUT:

- `n` – positive integer giving the size of the permutation matrices
- `backend` – the backend to use to create the polytope

See also

`sage.matrix.matrix2.Matrix.as_sum_of_permutations()` – return the current matrix as a sum of permutation matrices

EXAMPLES:

```
sage: b3 = polytopes.Birkhoff_polytope(3)
sage: b3.f_vector()
(1, 6, 15, 18, 9, 1)
sage: b3.ambient_dim(), b3.dim()
(9, 4)
sage: b3.is_lattice_polytope()
True
sage: p3 = b3.ehrhart_polynomial()      # optional - latte_int
sage: p3                                # optional - latte_int
1/8*t^4 + 3/4*t^3 + 15/8*t^2 + 9/4*t + 1
sage: [p3(i) for i in [1,2,3,4]]        # optional - latte_int
[6, 21, 55, 120]
sage: [len((i*b3).integral_points()) for i in [1,2,3,4]]
[6, 21, 55, 120]

sage: b4 = polytopes.Birkhoff_polytope(4)
sage: b4.n_vertices(), b4.ambient_dim(), b4.dim()
(24, 16, 9)
```

```
>>> from sage.all import *
>>> b3 = polytopes.Birkhoff_polytope(Integer(3))
>>> b3.f_vector()
(1, 6, 15, 18, 9, 1)
>>> b3.ambient_dim(), b3.dim()
(9, 4)
>>> b3.is_lattice_polytope()
True
>>> p3 = b3.ehrhart_polynomial()      # optional - latte_int
>>> p3                                # optional - latte_int
1/8*t^4 + 3/4*t^3 + 15/8*t^2 + 9/4*t + 1
>>> [p3(i) for i in [Integer(1),Integer(2),Integer(3),Integer(4)]]        #␣
↪optional - latte_int
[6, 21, 55, 120]
>>> [len((i*b3).integral_points()) for i in [Integer(1),Integer(2),Integer(3),
↪Integer(4)]]
[6, 21, 55, 120]

>>> b4 = polytopes.Birkhoff_polytope(Integer(4))
>>> b4.n_vertices(), b4.ambient_dim(), b4.dim()
(24, 16, 9)
```

Gosset_3_21(*backend=None*)

Return the Gosset 3_{21} polytope.

The Gosset 3_{21} polytope is a uniform 7-polytope. It has 56 vertices, and 702 facets: 126 3_{11} and 576 6-simplex. For more information, see the Wikipedia article 3_21_polytope.

INPUT:

- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: g = polytopes.Gosset_3_21(); g
A 7-dimensional polyhedron in ZZ^8 defined as the convex hull of 56 vertices
sage: g.f_vector() # not tested (~16s)
(1, 56, 756, 4032, 10080, 12096, 6048, 702, 1)
```

```
>>> from sage.all import *
>>> g = polytopes.Gosset_3_21(); g
A 7-dimensional polyhedron in ZZ^8 defined as the convex hull of 56 vertices
>>> g.f_vector() # not tested (~16s)
(1, 56, 756, 4032, 10080, 12096, 6048, 702, 1)
```

Kirkman_icosahedron(*backend=None*)

Return the Kirkman icosahedron.

The Kirkman icosahedron is a 3-polytope with integer coordinates: $(\pm 9, \pm 6, \pm 6)$, $(\pm 12, \pm 4, 0)$, $(0, \pm 12, \pm 8)$, $(\pm 6, 0, \pm 12)$. See [Fe2012] for more information.

INPUT:

- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: ki = polytopes.Kirkman_icosahedron()
sage: ki.f_vector()
(1, 20, 38, 20, 1)

sage: ki.volume()
6528

sage: vertices = ki.vertices()
sage: edges = [[vector(edge[0]),vector(edge[1])] for edge in ki.bounded_
↪edges()]
sage: edge_lengths = [norm(edge[0]-edge[1]) for edge in edges]
sage: sorted(set(edge_lengths))
[7, 8, 9, 11, 12, 14, 16]
```

```
>>> from sage.all import *
>>> ki = polytopes.Kirkman_icosahedron()
>>> ki.f_vector()
(1, 20, 38, 20, 1)

>>> ki.volume()
6528

>>> vertices = ki.vertices()
>>> edges = [[vector(edge[Integer(0)]),vector(edge[Integer(1)])] for edge in␣
↪ki.bounded_edges()]
>>> edge_lengths = [norm(edge[Integer(0)]-edge[Integer(1)]) for edge in edges]
>>> sorted(set(edge_lengths))
[7, 8, 9, 11, 12, 14, 16]
```

bitruncated_six_hundred_cell(*exact=True*, *backend=None*)

Return the bitruncated 600-cell.

The bitruncated 600-cell is a 4-dimensional 4-uniform polytope in the H_4 family. It has 3600 vertices. For more information see Wikipedia article Bitruncated 600-cell.

> **Warning**
>
> The coordinates are exact by default. The computation with inexact coordinates (using the backend `'cdd'`) returns a numerical inconsistency error, and thus cannot be computed.

INPUT:

- `exact` – boolean (default: `True`); if `True` use exact coordinates instead of floating point approximations
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: polytopes.runcinated_six_hundred_cell(exact=True,              # not␣
↪tested - very long time
....:                                         backend='normaliz')
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 3600 vertices
```

```
>>> from sage.all import *
>>> polytopes.runcinated_six_hundred_cell(exact=True,              # not␣
↪tested - very long time
...                                         backend='normaliz')
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 3600 vertices
```

buckyball(*exact=True*, *base_ring=None*, *backend=None*)

Return the bucky ball.

The bucky ball, also known as the truncated icosahedron is an Archimedean solid. It has 32 faces and 60 vertices.

> **See also**
>
> *icosahedron()*

INPUT:

- `exact` – boolean (default: `True`); if `False` use an approximate ring for the coordinates
- `base_ring` – the ring in which the coordinates will belong to. If it is not provided and `exact=True` it will be a the number field $\mathbf{Q}[\phi]$ where ϕ is the golden ratio and if `exact=False` it will be the real double field.
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: bb = polytopes.buckyball()     # long time                          #␣
↪needs sage.groups sage.rings.number_field
sage: bb.f_vector()                  # long time                          #␣
↪needs sage.groups sage.rings.number_field
(1, 60, 90, 32, 1)
sage: bb.base_ring()                 # long time                          #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.groups sage.rings.number_field
Number Field in sqrt5 with defining polynomial x^2 - 5
 with sqrt5 = 2.236067977499790?
```

```
>>> from sage.all import *
>>> bb = polytopes.buckyball()      # long time                             #␣
↪needs sage.groups sage.rings.number_field
>>> bb.f_vector()                   # long time                             #␣
↪needs sage.groups sage.rings.number_field
(1, 60, 90, 32, 1)
>>> bb.base_ring()                  # long time                             #␣
↪needs sage.groups sage.rings.number_field
Number Field in sqrt5 with defining polynomial x^2 - 5
 with sqrt5 = 2.236067977499790?
```

A much faster implementation using floating point approximations:

```
sage: bb = polytopes.buckyball(exact=False)                                   #␣
↪needs sage.groups
sage: bb.f_vector()                                                           #␣
↪needs sage.groups
(1, 60, 90, 32, 1)
sage: bb.base_ring()                                                          #␣
↪needs sage.groups
Real Double Field
```

```
>>> from sage.all import *
>>> bb = polytopes.buckyball(exact=False)                                   #␣
↪needs sage.groups
>>> bb.f_vector()                                                           #␣
↪needs sage.groups
(1, 60, 90, 32, 1)
>>> bb.base_ring()                                                          #␣
↪needs sage.groups
Real Double Field
```

Its facets are 5 regular pentagons and 6 regular hexagons:

```
sage: sum(1 for f in bb.facets() if len(f.vertices()) == 5)                   #␣
↪needs sage.groups
12
sage: sum(1 for f in bb.facets() if len(f.vertices()) == 6)                   #␣
↪needs sage.groups
20
```

```
>>> from sage.all import *
>>> sum(Integer(1) for f in bb.facets() if len(f.vertices()) == Integer(5))  ␣
↪               # needs sage.groups
12
>>> sum(Integer(1) for f in bb.facets() if len(f.vertices()) == Integer(6))  ␣
↪               # needs sage.groups
20
```

cantellated_one_hundred_twenty_cell(*exact=True*, *backend=None*)

Return the cantellated 120-cell.

The cantellated 120-cell is a 4-dimensional 4-uniform polytope in the H_4 family. It has 3600 vertices. For more information see Wikipedia article Cantellated 120-cell.

> **Warning**
>
> The coordinates are exact by default. The computation with inexact coordinates (using the backend `'cdd'`) returns a numerical inconsistency error, and thus cannot be computed.

INPUT:

- `exact` – boolean (default: `True`); if `True` use exact coordinates instead of floating point approximations
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: polytopes.cantellated_one_hundred_twenty_cell(backend='normaliz')  #␣
↪not tested - long time
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 3600 vertices
```

```
>>> from sage.all import *
>>> polytopes.cantellated_one_hundred_twenty_cell(backend='normaliz')  # not␣
↪tested - long time
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 3600 vertices
```

cantellated_six_hundred_cell(*exact=False*, *backend=None*)

Return the cantellated 600-cell.

The cantellated 600-cell is a 4-dimensional 4-uniform polytope in the H_4 family. It has 3600 vertices. For more information see Wikipedia article Cantellated 600-cell.

> **Warning**
>
> The coordinates are inexact by default. The computation with inexact coordinates (using the backend `'cdd'`) issues a UserWarning on inconsistencies.

INPUT:

- `exact` – boolean (default: `False`); if `True` use exact coordinates instead of floating point approximations
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: polytopes.cantellated_six_hundred_cell()      # not tested - very long␣
↪time
doctest:warning
...
UserWarning: This polyhedron data is numerically complicated; cdd
could not convert between the inexact V and H representation
without loss of data. The resulting object might show
```

(continues on next page)

(continued from previous page)

```
inconsistencies.
A 4-dimensional polyhedron in RDF^4 defined as the convex hull of 3600␣
↪vertices
```

```
>>> from sage.all import *
>>> polytopes.cantellated_six_hundred_cell()        # not tested - very long␣
↪time
doctest:warning
...
UserWarning: This polyhedron data is numerically complicated; cdd
could not convert between the inexact V and H representation
without loss of data. The resulting object might show
inconsistencies.
A 4-dimensional polyhedron in RDF^4 defined as the convex hull of 3600␣
↪vertices
```

It is possible to use the backend 'normaliz' to get an exact representation:

```
sage: polytopes.cantellated_six_hundred_cell(exact=True,  # not tested - long␣
↪time
....:                                             backend='normaliz')
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 3600 vertices
```

```
>>> from sage.all import *
>>> polytopes.cantellated_six_hundred_cell(exact=True,  # not tested - long␣
↪time
...                                           backend='normaliz')
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 3600 vertices
```

cantitruncated_one_hundred_twenty_cell(*exact=True*, *backend=None*)

Return the cantitruncated 120-cell.

The cantitruncated 120-cell is a 4-dimensional 4-uniform polytope in the H_4 family. It has 7200 vertices. For more information see Wikipedia article Cantitruncated 120-cell.

Warning

The coordinates are exact by default. The computation with inexact coordinates (using the backend 'cdd') returns a numerical inconsistency error, and thus cannot be computed.

INPUT:

- exact – boolean (default: True); if True use exact coordinates instead of floating point approximations
- backend – the backend to use to create the polytope

EXAMPLES:

```
sage: polytopes.cantitruncated_one_hundred_twenty_cell(exact=True, backend=
↪'normaliz')  # not tested - very long time
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 7200 vertices
```

```
>>> from sage.all import *
>>> polytopes.cantitruncated_one_hundred_twenty_cell(exact=True, backend=
↪'normaliz')   # not tested - very long time
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 7200 vertices
```

cantitruncated_six_hundred_cell(*exact=True*, *backend=None*)

Return the cantitruncated 600-cell.

The cantitruncated 600-cell is a 4-dimensional 4-uniform polytope in the H_4 family. It has 7200 vertices. For more information see Wikipedia article Cantitruncated 600-cell.

> **Warning**
>
> The coordinates are exact by default. The computation with inexact coordinates (using the backend `'cdd'`) returns a numerical inconsistency error, and thus cannot be computed.

INPUT:

- `exact` – boolean (default: `True`); if `True` use exact coordinates instead of floating point approximations
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: polytopes.cantitruncated_six_hundred_cell(exact=True,          # not␣
↪tested - very long time
....:                                             backend='normaliz')
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 7200 vertices
```

```
>>> from sage.all import *
>>> polytopes.cantitruncated_six_hundred_cell(exact=True,          # not␣
↪tested - very long time
...                                             backend='normaliz')
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 7200 vertices
```

cross_polytope(*dim*, *backend=None*)

Return a cross-polytope in dimension `dim`.

A cross-polytope is a higher dimensional generalization of the octahedron. It is the convex hull of the $2d$ points $(\pm 1, 0, \ldots, 0)$, $(0, \pm 1, \ldots, 0)$, ldots, $(0, 0, \ldots, \pm 1)$. See the Wikipedia article Cross-polytope for more information.

INPUT:

- `dim` – integer; the dimension of the cross-polytope
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: four_cross = polytopes.cross_polytope(4)
sage: four_cross.f_vector()
(1, 8, 24, 32, 16, 1)
sage: four_cross.is_simple()
False
```

```
>>> from sage.all import *
>>> four_cross = polytopes.cross_polytope(Integer(4))
>>> four_cross.f_vector()
(1, 8, 24, 32, 16, 1)
>>> four_cross.is_simple()
False
```

cube (*intervals=None*, *backend=None*)

Return the cube.

The cube is the Platonic solid that is obtained as the convex hull of the eight ± 1 vectors of length 3 (by default). Alternatively, the cube is the product of three intervals from `intervals`.

See also:

hypercube()

INPUT:

- `intervals` – list (default=None). It takes the following possible inputs:
  - If the input is `None` (the default), returns the convex hull of the eight ± 1 vectors of length three.
  - `'zero_one'` – string; return the $0/1$-cube
  - a list of 3 lists of length 2. The cube will be a product of these three intervals.
- `backend` – the backend to use to create the polytope

OUTPUT: a cube as a polyhedron object

EXAMPLES:

Return the ± 1-cube:

```
sage: c = polytopes.cube()
sage: c
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 8 vertices
sage: c.f_vector()
(1, 8, 12, 6, 1)
sage: c.volume()
8
sage: c.plot()                                                              #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> c = polytopes.cube()
>>> c
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 8 vertices
>>> c.f_vector()
(1, 8, 12, 6, 1)
>>> c.volume()
8
>>> c.plot()                                                              #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.plot
Graphics3d Object
```

Return the $0/1$-cube:

```
sage: cc = polytopes.cube(intervals ='zero_one')
sage: cc.vertices_list()
[[1, 0, 0],
 [1, 1, 0],
 [1, 1, 1],
 [1, 0, 1],
 [0, 0, 1],
 [0, 0, 0],
 [0, 1, 0],
 [0, 1, 1]]
```

```
>>> from sage.all import *
>>> cc = polytopes.cube(intervals ='zero_one')
>>> cc.vertices_list()
[[1, 0, 0],
 [1, 1, 0],
 [1, 1, 1],
 [1, 0, 1],
 [0, 0, 1],
 [0, 0, 0],
 [0, 1, 0],
 [0, 1, 1]]
```

cuboctahedron(*backend=None*)

Return the cuboctahedron.

The cuboctahedron is an Archimedean solid with 12 vertices and 14 faces dual to the rhombic dodecahedron. It can be defined as the convex hull of the twelve vertices $(0, \pm 1, \pm 1)$, $(\pm 1, 0, \pm 1)$ and $(\pm 1, \pm 1, 0)$. For more information, see the Wikipedia article Cuboctahedron.

INPUT:

- `backend` – the backend to use to create the polytope

See also

rhombic_dodecahedron()

EXAMPLES:

```
sage: co = polytopes.cuboctahedron()
sage: co.f_vector()
(1, 12, 24, 14, 1)
```

```
>>> from sage.all import *
>>> co = polytopes.cuboctahedron()
>>> co.f_vector()
(1, 12, 24, 14, 1)
```

Its facets are 8 triangles and 6 squares:

```
sage: sum(1 for f in co.facets() if len(f.vertices()) == 3)
8
sage: sum(1 for f in co.facets() if len(f.vertices()) == 4)
6
```

```
>>> from sage.all import *
>>> sum(Integer(1) for f in co.facets() if len(f.vertices()) == Integer(3))
8
>>> sum(Integer(1) for f in co.facets() if len(f.vertices()) == Integer(4))
6
```

Some more computation:

```
sage: co.volume()
20/3
sage: co.ehrhart_polynomial()      # optional - latte_int
20/3*t^3 + 8*t^2 + 10/3*t + 1
```

```
>>> from sage.all import *
>>> co.volume()
20/3
>>> co.ehrhart_polynomial()      # optional - latte_int
20/3*t^3 + 8*t^2 + 10/3*t + 1
```

cyclic_polytope(*dim*, *n*, *base_ring=Rational Field*, *backend=None*)

Return a cyclic polytope.

A cyclic polytope of dimension `dim` with `n` vertices is the convex hull of the points `(t,t^2,...,t^dim)` with $t \in \{0, 1, ..., n-1\}$. For more information, see the Wikipedia article Cyclic_polytope.

INPUT:

- `dim` – positive integer; the dimension of the polytope
- `n` – positive integer; the number of vertices
- `base_ring` – either `QQ` (default) or `RDF`
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: c = polytopes.cyclic_polytope(4,10)
sage: c.f_vector()
(1, 10, 45, 70, 35, 1)
```

```
>>> from sage.all import *
>>> c = polytopes.cyclic_polytope(Integer(4),Integer(10))
>>> c.f_vector()
(1, 10, 45, 70, 35, 1)
```

dodecahedron(*exact=True*, *base_ring=None*, *backend=None*)

Return a dodecahedron.

The dodecahedron is the Platonic solid dual to the *icosahedron()*.

INPUT:

- `exact` – boolean (default: `True`); if `False` use an approximate ring for the coordinates
- `base_ring` – (optional) the ring in which the coordinates will belong to. Note that this ring must contain $\sqrt(5)$. If it is not provided and `exact=True` it will be the number field $\mathbf{Q}[\sqrt(5)]$ and if `exact=False` it will be the real double field.
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: # needs sage.groups sage.rings.number_field
sage: d12 = polytopes.dodecahedron()
sage: d12.f_vector()
(1, 20, 30, 12, 1)
sage: d12.volume()
-176*sqrt5 + 400
sage: numerical_approx(_)
6.45203596003699

sage: d12 = polytopes.dodecahedron(exact=False)                          #␣
↪needs sage.groups
sage: d12.base_ring()                                                    #␣
↪needs sage.groups
Real Double Field
```

```
>>> from sage.all import *
>>> # needs sage.groups sage.rings.number_field
>>> d12 = polytopes.dodecahedron()
>>> d12.f_vector()
(1, 20, 30, 12, 1)
>>> d12.volume()
-176*sqrt5 + 400
>>> numerical_approx(_)
6.45203596003699

>>> d12 = polytopes.dodecahedron(exact=False)                          #␣
↪needs sage.groups
>>> d12.base_ring()                                                    #␣
↪needs sage.groups
Real Double Field
```

Here is an error with a field that does not contain $\sqrt(5)$:

```
sage: polytopes.dodecahedron(base_ring=QQ)                               #␣
↪needs sage.groups sage.symbolic
Traceback (most recent call last):
...
TypeError: unable to convert 1/4*sqrt(5) + 1/4 to a rational
```

```
>>> from sage.all import *
>>> polytopes.dodecahedron(base_ring=QQ)                               #␣
↪needs sage.groups sage.symbolic
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
TypeError: unable to convert 1/4*sqrt(5) + 1/4 to a rational
```

static edge_polytope(*backend=None*)

Return the edge polytope of `self`.

The edge polytope (EP) of a Graph on n vertices is the polytope in $\mathbf{Z}^n$ defined as the convex hull of $e_i + e_j$ for each edge (i, j). Here $e_1, \dots, e_n$ denotes the standard basis.

INPUT:

- `backend` – string or `None` (default); the backend to use; see *sage.geometry.polyhedron.constructor.Polyhedron()*

EXAMPLES:

The EP of a 4-cycle is a square:

```
sage: G = graphs.CycleGraph(4)
sage: P = G.edge_polytope(); P                                          #␣
↪needs sage.geometry.polyhedron
A 2-dimensional polyhedron in ZZ^4 defined as the convex hull of 4 vertices
```

```
>>> from sage.all import *
>>> G = graphs.CycleGraph(Integer(4))
>>> P = G.edge_polytope(); P                                          #␣
↪needs sage.geometry.polyhedron
A 2-dimensional polyhedron in ZZ^4 defined as the convex hull of 4 vertices
```

The EP of a complete graph on 4 vertices is cross polytope:

```
sage: G = graphs.CompleteGraph(4)
sage: P = G.edge_polytope(); P                                          #␣
↪needs sage.geometry.polyhedron
A 3-dimensional polyhedron in ZZ^4 defined as the convex hull of 6 vertices
sage: P.is_combinatorially_isomorphic(polytopes.cross_polytope(3))      #␣
↪needs sage.geometry.polyhedron
True
```

```
>>> from sage.all import *
>>> G = graphs.CompleteGraph(Integer(4))
>>> P = G.edge_polytope(); P                                          #␣
↪needs sage.geometry.polyhedron
A 3-dimensional polyhedron in ZZ^4 defined as the convex hull of 6 vertices
>>> P.is_combinatorially_isomorphic(polytopes.cross_polytope(Integer(3)))    ␣
↪     # needs sage.geometry.polyhedron
True
```

The EP of a graph is isomorphic to the subdirect sum of its connected components EPs:

```
sage: n = randint(3, 6)
sage: G1 = graphs.RandomGNP(n, 0.2)                                     #␣
↪needs networkx
sage: n = randint(3, 6)
```

(continues on next page)

(continued from previous page)

```
sage: G2 = graphs.RandomGNP(n, 0.2)                                          #␣
↪needs networkx
sage: G = G1.disjoint_union(G2)                                              #␣
↪needs networkx
sage: P = G.edge_polytope()                                                  #␣
↪needs networkx sage.geometry.polyhedron
sage: P1 = G1.edge_polytope()                                                #␣
↪needs networkx sage.geometry.polyhedron
sage: P2 = G2.edge_polytope()                                                #␣
↪needs networkx sage.geometry.polyhedron
sage: P.is_combinatorially_isomorphic(P1.subdirect_sum(P2))                  #␣
↪needs networkx sage.geometry.polyhedron
True
```

```
>>> from sage.all import *
>>> n = randint(Integer(3), Integer(6))
>>> G1 = graphs.RandomGNP(n, RealNumber('0.2'))                               ␣
↪          # needs networkx
>>> n = randint(Integer(3), Integer(6))
>>> G2 = graphs.RandomGNP(n, RealNumber('0.2'))                               ␣
↪          # needs networkx
>>> G = G1.disjoint_union(G2)                                               #␣
↪needs networkx
>>> P = G.edge_polytope()                                                   #␣
↪needs networkx sage.geometry.polyhedron
>>> P1 = G1.edge_polytope()                                                 #␣
↪needs networkx sage.geometry.polyhedron
>>> P2 = G2.edge_polytope()                                                 #␣
↪needs networkx sage.geometry.polyhedron
>>> P.is_combinatorially_isomorphic(P1.subdirect_sum(P2))                   #␣
↪needs networkx sage.geometry.polyhedron
True
```

All trees on n vertices have isomorphic EPs:

```
sage: n = randint(4, 10)
sage: G1 = graphs.RandomTree(n)
sage: G2 = graphs.RandomTree(n)
sage: P1 = G1.edge_polytope()                                                #␣
↪needs sage.geometry.polyhedron
sage: P2 = G2.edge_polytope()                                                #␣
↪needs sage.geometry.polyhedron
sage: P1.is_combinatorially_isomorphic(P2)                                   #␣
↪needs sage.geometry.polyhedron
True
```

```
>>> from sage.all import *
>>> n = randint(Integer(4), Integer(10))
>>> G1 = graphs.RandomTree(n)
>>> G2 = graphs.RandomTree(n)
>>> P1 = G1.edge_polytope()                                                 #␣
↪needs sage.geometry.polyhedron
```

(continues on next page)

(continued from previous page)

```
>>> P2 = G2.edge_polytope()                                                  #␣
↪needs sage.geometry.polyhedron
>>> P1.is_combinatorially_isomorphic(P2)                                     #␣
↪needs sage.geometry.polyhedron
True
```

However, there are still many different EPs:

```
sage: len(list(graphs(5)))
34
sage: polys = []
sage: for G in graphs(5):                                                     #␣
↪needs sage.geometry.polyhedron
....:     P = G.edge_polytope()
....:     for P1 in polys:
....:         if P.is_combinatorially_isomorphic(P1):
....:             break
....:     else:
....:         polys.append(P)
sage: len(polys)                                                              #␣
↪needs sage.geometry.polyhedron
19
```

```
>>> from sage.all import *
>>> len(list(graphs(Integer(5))))
34
>>> polys = []
>>> for G in graphs(Integer(5)):                                              ␣
↪    # needs sage.geometry.polyhedron
...     P = G.edge_polytope()
...     for P1 in polys:
...         if P.is_combinatorially_isomorphic(P1):
...             break
...     else:
...         polys.append(P)
>>> len(polys)                                                               #␣
↪needs sage.geometry.polyhedron
19
```

static flow_polytope(*edges=None*, *ends=None*, *backend=None*)

Return the flow polytope of a digraph.

The flow polytope of a directed graph is the polytope consisting of all nonnegative flows on the graph with a given set S of sources and a given set T of sinks.

A *flow* on a directed graph G with a given set S of sources and a given set T of sinks means an assignment of a nonnegative real to each edge of G such that the flow is conserved in each vertex outside of S and T, and there is a unit of flow entering each vertex in S and a unit of flow leaving each vertex in T. These flows clearly form a polytope in the space of all assignments of reals to the edges of G.

The polytope is empty unless the sets S and T are equinumerous.

By default, S is taken to be the set of all sources (i.e., vertices of indegree 0) of G, and T is taken to be the set of all sinks (i.e., vertices of outdegree 0) of G. If a different choice of S and T is desired, it can be specified using the optional `ends` parameter.

The polytope is returned as a polytope in $\mathbf{R}^m$, where m is the number of edges of the digraph `self`. The k-th coordinate of a point in the polytope is the real assigned to the k-th edge of `self`. The order of the edges is the one returned by `self.edges(sort=True)`. If a different order is desired, it can be specified using the optional `edges` parameter.

The faces and volume of these polytopes are of interest. Examples of these polytopes are the Chan-Robbins-Yuen polytope and the Pitman-Stanley polytope [PS2002].

INPUT:

- `edges` – list (default: `None`); a list of edges of `self`. If not specified, the list of all edges of `self` is used with the default ordering of `self.edges(sort=True)`. This determines which coordinate of a point in the polytope will correspond to which edge of `self`. It is also possible to specify a list which contains not all edges of `self`; this results in a polytope corresponding to the flows which are 0 on all remaining edges. Notice that the edges entered here must be in the precisely same format as outputted by `self.edges()`; so, if `self.edges()` outputs an edge in the form `(1, 3, None)`, then `(1, 3)` will not do!
- `ends` – (default: `(self.sources(), self.sinks())`) a pair (S, T) of an iterable S and an iterable T
- `backend` – string or `None` (default); the backend to use; see `sage.geometry.polyhedron.constructor.Polyhedron()`

Note

Flow polytopes can also be built through the `polytopes.<tab>` object:

```
sage: polytopes.flow_polytope(digraphs.Path(5))                          #␣
↪needs sage.geometry.polyhedron
A 0-dimensional polyhedron in QQ^4 defined as the convex hull of 1 vertex
```

```
>>> from sage.all import *
>>> polytopes.flow_polytope(digraphs.Path(Integer(5)))                   ␣
↪    # needs sage.geometry.polyhedron
A 0-dimensional polyhedron in QQ^4 defined as the convex hull of 1 vertex
```

EXAMPLES:

A commutative square:

```
sage: G = DiGraph({1: [2, 3], 2: [4], 3: [4]})
sage: fl = G.flow_polytope(); fl                                          #␣
↪needs sage.geometry.polyhedron
A 1-dimensional polyhedron in QQ^4 defined as the convex hull
of 2 vertices
sage: fl.vertices()                                                       #␣
↪needs sage.geometry.polyhedron
(A vertex at (0, 1, 0, 1), A vertex at (1, 0, 1, 0))
```

```
>>> from sage.all import *
>>> G = DiGraph({Integer(1): [Integer(2), Integer(3)], Integer(2):␣
↪[Integer(4)], Integer(3): [Integer(4)]})
>>> fl = G.flow_polytope(); fl                                           #␣
↪needs sage.geometry.polyhedron
```

(continues on next page)

(continued from previous page)

```
A 1-dimensional polyhedron in QQ^4 defined as the convex hull
of 2 vertices
>>> fl.vertices()                                                    #␣
↪needs sage.geometry.polyhedron
(A vertex at (0, 1, 0, 1), A vertex at (1, 0, 1, 0))
```

Using a different order for the edges of the graph:

```
sage: ordered_edges = G.edges(sort=True, key=lambda x: x[0] - x[1])
sage: fl = G.flow_polytope(edges=ordered_edges); fl                   #␣
↪needs sage.geometry.polyhedron
A 1-dimensional polyhedron in QQ^4 defined as the convex hull of 2 vertices
sage: fl.vertices()                                                   #␣
↪needs sage.geometry.polyhedron
(A vertex at (0, 1, 1, 0), A vertex at (1, 0, 0, 1))
```

```
>>> from sage.all import *
>>> ordered_edges = G.edges(sort=True, key=lambda x: x[Integer(0)] -␣
↪x[Integer(1)])
>>> fl = G.flow_polytope(edges=ordered_edges); fl                    #␣
↪needs sage.geometry.polyhedron
A 1-dimensional polyhedron in QQ^4 defined as the convex hull of 2 vertices
>>> fl.vertices()                                                    #␣
↪needs sage.geometry.polyhedron
(A vertex at (0, 1, 1, 0), A vertex at (1, 0, 0, 1))
```

A tournament on 4 vertices:

```
sage: H = digraphs.TransitiveTournament(4)
sage: fl = H.flow_polytope(); fl                                      #␣
↪needs sage.geometry.polyhedron
A 3-dimensional polyhedron in QQ^6 defined as the convex hull
of 4 vertices
sage: fl.vertices()                                                   #␣
↪needs sage.geometry.polyhedron
(A vertex at (0, 0, 1, 0, 0, 0),
 A vertex at (0, 1, 0, 0, 0, 1),
 A vertex at (1, 0, 0, 0, 1, 0),
 A vertex at (1, 0, 0, 1, 0, 1))
```

```
>>> from sage.all import *
>>> H = digraphs.TransitiveTournament(Integer(4))
>>> fl = H.flow_polytope(); fl                                       #␣
↪needs sage.geometry.polyhedron
A 3-dimensional polyhedron in QQ^6 defined as the convex hull
of 4 vertices
>>> fl.vertices()                                                    #␣
↪needs sage.geometry.polyhedron
(A vertex at (0, 0, 1, 0, 0, 0),
 A vertex at (0, 1, 0, 0, 0, 1),
 A vertex at (1, 0, 0, 0, 1, 0),
 A vertex at (1, 0, 0, 1, 0, 1))
```

Restricting to a subset of the edges:

```
sage: fl = H.flow_polytope(edges=[(0, 1, None), (1, 2, None),                #␣
↪needs sage.geometry.polyhedron
....:                              (2, 3, None), (0, 3, None)]); fl
A 1-dimensional polyhedron in QQ^4 defined as the convex hull
of 2 vertices
sage: fl.vertices()                                                          #␣
↪needs sage.geometry.polyhedron
(A vertex at (0, 0, 0, 1), A vertex at (1, 1, 1, 0))
```

```
>>> from sage.all import *
>>> fl = H.flow_polytope(edges=[(Integer(0), Integer(1), None), (Integer(1),␣
↪Integer(2), None),                # needs sage.geometry.polyhedron
...                              (Integer(2), Integer(3), None), (Integer(0),␣
↪Integer(3), None)]); fl
A 1-dimensional polyhedron in QQ^4 defined as the convex hull
of 2 vertices
>>> fl.vertices()                                                          #␣
↪needs sage.geometry.polyhedron
(A vertex at (0, 0, 0, 1), A vertex at (1, 1, 1, 0))
```

Using a different choice of sources and sinks:

```
sage: # needs sage.geometry.polyhedron
sage: fl = H.flow_polytope(ends=([1], [3])); fl
A 1-dimensional polyhedron in QQ^6 defined as the convex hull
of 2 vertices
sage: fl.vertices()
(A vertex at (0, 0, 0, 1, 0, 1), A vertex at (0, 0, 0, 0, 1, 0))
sage: fl = H.flow_polytope(ends=([0, 1], [3])); fl
The empty polyhedron in QQ^6
sage: fl = H.flow_polytope(ends=([3], [0])); fl
The empty polyhedron in QQ^6
sage: fl = H.flow_polytope(ends=([0, 1], [2, 3])); fl
A 3-dimensional polyhedron in QQ^6 defined as the convex hull
of 5 vertices
sage: fl.vertices()
(A vertex at (0, 0, 1, 1, 0, 0),
 A vertex at (0, 1, 0, 0, 1, 0),
 A vertex at (1, 0, 0, 2, 0, 1),
 A vertex at (1, 0, 0, 1, 1, 0),
 A vertex at (0, 1, 0, 1, 0, 1))
sage: fl = H.flow_polytope(edges=[(0, 1, None), (1, 2, None),
....:                              (2, 3, None), (0, 2, None),
....:                              (1, 3, None)],
....:                       ends=([0, 1], [2, 3])); fl
A 2-dimensional polyhedron in QQ^5 defined as the convex hull
of 4 vertices
sage: fl.vertices()
(A vertex at (0, 0, 0, 1, 1),
 A vertex at (1, 2, 1, 0, 0),
 A vertex at (1, 1, 0, 0, 1),
 A vertex at (0, 1, 1, 1, 0))
```

```
>>> from sage.all import *
>>> # needs sage.geometry.polyhedron
>>> fl = H.flow_polytope(ends=([Integer(1)], [Integer(3)])); fl
A 1-dimensional polyhedron in QQ^6 defined as the convex hull
of 2 vertices
>>> fl.vertices()
(A vertex at (0, 0, 0, 1, 0, 1), A vertex at (0, 0, 0, 0, 1, 0))
>>> fl = H.flow_polytope(ends=([Integer(0), Integer(1)], [Integer(3)])); fl
The empty polyhedron in QQ^6
>>> fl = H.flow_polytope(ends=([Integer(3)], [Integer(0)])); fl
The empty polyhedron in QQ^6
>>> fl = H.flow_polytope(ends=([Integer(0), Integer(1)], [Integer(2),␣
↪Integer(3)])); fl
A 3-dimensional polyhedron in QQ^6 defined as the convex hull
of 5 vertices
>>> fl.vertices()
(A vertex at (0, 0, 1, 1, 0, 0),
 A vertex at (0, 1, 0, 0, 1, 0),
 A vertex at (1, 0, 0, 2, 0, 1),
 A vertex at (1, 0, 0, 1, 1, 0),
 A vertex at (0, 1, 0, 1, 0, 1))
>>> fl = H.flow_polytope(edges=[(Integer(0), Integer(1), None), (Integer(1),␣
↪Integer(2), None),
...                                 (Integer(2), Integer(3), None), (Integer(0),␣
↪Integer(2), None),
...                                 (Integer(1), Integer(3), None)],
...                          ends=([Integer(0), Integer(1)], [Integer(2),␣
↪Integer(3)])); fl
A 2-dimensional polyhedron in QQ^5 defined as the convex hull
of 4 vertices
>>> fl.vertices()
(A vertex at (0, 0, 0, 1, 1),
 A vertex at (1, 2, 1, 0, 0),
 A vertex at (1, 1, 0, 0, 1),
 A vertex at (0, 1, 1, 1, 0))
```

A digraph with one source and two sinks:

```
sage: Y = DiGraph({1: [2], 2: [3, 4]})
sage: Y.flow_polytope()                                                     #␣
↪needs sage.geometry.polyhedron
The empty polyhedron in QQ^3
```

```
>>> from sage.all import *
>>> Y = DiGraph({Integer(1): [Integer(2)], Integer(2): [Integer(3),␣
↪Integer(4)]})
>>> Y.flow_polytope()                                                     #␣
↪needs sage.geometry.polyhedron
The empty polyhedron in QQ^3
```

A digraph with one vertex and no edge:

```
sage: Z = DiGraph({1: []})
sage: Z.flow_polytope()                                                  #␣
↪needs sage.geometry.polyhedron
A 0-dimensional polyhedron in QQ^0 defined as the convex hull
of 1 vertex
```

```
>>> from sage.all import *
>>> Z = DiGraph({Integer(1): []})
>>> Z.flow_polytope()                                                   #␣
↪needs sage.geometry.polyhedron
A 0-dimensional polyhedron in QQ^0 defined as the convex hull
of 1 vertex
```

A digraph with multiple edges (Issue #28837):

```
sage: G = DiGraph([(0, 1), (0,1)], multiedges=True); G
Multi-digraph on 2 vertices
sage: P = G.flow_polytope(); P                                           #␣
↪needs sage.geometry.polyhedron
A 1-dimensional polyhedron in QQ^2 defined as the convex hull of 2 vertices
sage: P.vertices()                                                       #␣
↪needs sage.geometry.polyhedron
(A vertex at (1, 0), A vertex at (0, 1))
sage: P.lines()                                                          #␣
↪needs sage.geometry.polyhedron
()
```

```
>>> from sage.all import *
>>> G = DiGraph([(Integer(0), Integer(1)), (Integer(0),Integer(1))],␣
↪multiedges=True); G
Multi-digraph on 2 vertices
>>> P = G.flow_polytope(); P                                            #␣
↪needs sage.geometry.polyhedron
A 1-dimensional polyhedron in QQ^2 defined as the convex hull of 2 vertices
>>> P.vertices()                                                        #␣
↪needs sage.geometry.polyhedron
(A vertex at (1, 0), A vertex at (0, 1))
>>> P.lines()                                                           #␣
↪needs sage.geometry.polyhedron
()
```

generalized_permutahedron(*coxeter_type*, *point=None*, *exact=True*, *regular=False*, *backend=None*)

Return the generalized permutahedron of type `coxeter_type` as the convex hull of the orbit of `point` in the fundamental cone.

This generalized permutahedron lies in the vector space used in the geometric representation, that is, in the default case, the dimension of generalized permutahedron equals the dimension of the space.

INPUT:

- `coxeter_type` – a Coxeter type; given as a pair [type,rank], where type is a letter and rank is the number of generators
- `point` – list (default: `None`); a point given by its coordinates in the weight basis. If `None` is given, the point $(1, 1, 1, \ldots)$ is used.

- `exact` – boolean (default: `True`); if `False` use floating point approximations instead of exact coordinates
- `regular` – boolean (default: `False`); whether to apply a linear transformation making the vertex figures isometric
- `backend` – backend to use to create the polytope; (default: `None`)

EXAMPLES:

```
sage: perm_a3 = polytopes.generalized_permutahedron(['A',3]); perm_a3        #␣
↪needs sage.combinat
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 24 vertices
```

```
>>> from sage.all import *
>>> perm_a3 = polytopes.generalized_permutahedron(['A',Integer(3)]); perm_a3 ␣
↪     # needs sage.combinat
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 24 vertices
```

You can put the starting point along the hyperplane of the first generator:

```
sage: # needs sage.combinat
sage: perm_a3_011 = polytopes.generalized_permutahedron(['A',3], [0,1,1])
sage: perm_a3_011
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 12 vertices
sage: perm_a3_110 = polytopes.generalized_permutahedron(['A',3], [1,1,0])
sage: perm_a3_110
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 12 vertices
sage: perm_a3_110.is_combinatorially_isomorphic(perm_a3_011)
True
sage: perm_a3_101 = polytopes.generalized_permutahedron(['A',3], [1,0,1])
sage: perm_a3_101
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 12 vertices
sage: perm_a3_110.is_combinatorially_isomorphic(perm_a3_101)
False
sage: perm_a3_011.f_vector()
(1, 12, 18, 8, 1)
sage: perm_a3_101.f_vector()
(1, 12, 24, 14, 1)
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> perm_a3_011 = polytopes.generalized_permutahedron(['A',Integer(3)],␣
↪[Integer(0),Integer(1),Integer(1)])
>>> perm_a3_011
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 12 vertices
>>> perm_a3_110 = polytopes.generalized_permutahedron(['A',Integer(3)],␣
↪[Integer(1),Integer(1),Integer(0)])
>>> perm_a3_110
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 12 vertices
>>> perm_a3_110.is_combinatorially_isomorphic(perm_a3_011)
True
>>> perm_a3_101 = polytopes.generalized_permutahedron(['A',Integer(3)],␣
↪[Integer(1),Integer(0),Integer(1)])
>>> perm_a3_101
```

(continues on next page)

(continued from previous page)

```
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 12 vertices
>>> perm_a3_110.is_combinatorially_isomorphic(perm_a3_101)
False
>>> perm_a3_011.f_vector()
(1, 12, 18, 8, 1)
>>> perm_a3_101.f_vector()
(1, 12, 24, 14, 1)
```

The usual output does not necessarily give a polyhedron with isometric vertex figures:

```
sage: perm_a2 = polytopes.generalized_permutahedron(['A',2])                #␣
↪needs sage.combinat
sage: perm_a2.vertices()                                                    #␣
↪needs sage.combinat
(A vertex at (-1, -1),
 A vertex at (-1, 0),
 A vertex at (0, -1),
 A vertex at (0, 1),
 A vertex at (1, 0),
 A vertex at (1, 1))
```

```
>>> from sage.all import *
>>> perm_a2 = polytopes.generalized_permutahedron(['A',Integer(2)])         ␣
↪     # needs sage.combinat
>>> perm_a2.vertices()                                                     #␣
↪needs sage.combinat
(A vertex at (-1, -1),
 A vertex at (-1, 0),
 A vertex at (0, -1),
 A vertex at (0, 1),
 A vertex at (1, 0),
 A vertex at (1, 1))
```

It works also with Coxeter types that lead to non-rational coordinates:

```
sage: perm_b3 = polytopes.generalized_permutahedron(['B',3])        # long␣
↪time, needs sage.combinat sage.rings.number_field
sage: perm_b3                                                       # long␣
↪time, needs sage.combinat sage.rings.number_field
A 3-dimensional polyhedron in
 (Number Field in a with defining polynomial x^2 - 2 with a = 1.
↪414213562373095?)^3
 defined as the convex hull of 48 vertices
```

```
>>> from sage.all import *
>>> perm_b3 = polytopes.generalized_permutahedron(['B',Integer(3)])        #␣
↪long time, needs sage.combinat sage.rings.number_field
>>> perm_b3                                                    # long time,
↪ needs sage.combinat sage.rings.number_field
A 3-dimensional polyhedron in
 (Number Field in a with defining polynomial x^2 - 2 with a = 1.
↪414213562373095?)^3
```

(continues on next page)

(continued from previous page)

```
defined as the convex hull of 48 vertices
```

Setting `regular=True` applies a linear transformation to get isometric vertex figures and the result is inscribed. This cannot be done using rational coordinates. We first do the computations using floating point approximations (`RDF`):

```
sage: perm_a2_inexact = polytopes.generalized_permutahedron(                #␣
↪needs sage.combinat
....:     ['A',2], exact=False)
sage: sorted(perm_a2_inexact.vertices())                                    #␣
↪needs sage.combinat
[A vertex at (-1.0, -1.0),
 A vertex at (-1.0, 0.0),
 A vertex at (0.0, -1.0),
 A vertex at (0.0, 1.0),
 A vertex at (1.0, 0.0),
 A vertex at (1.0, 1.0)]

sage: perm_a2_inexact_reg = polytopes.generalized_permutahedron(            #␣
↪needs sage.combinat
....:     ['A',2], exact=False, regular=True)
sage: sorted(perm_a2_inexact_reg.vertices())                                #␣
↪needs sage.combinat
[A vertex at (-1.0, 0.0),
 A vertex at (-0.5, -0.8660254038),
 A vertex at (-0.5, 0.8660254038),
 A vertex at (0.5, -0.8660254038),
 A vertex at (0.5, 0.8660254038),
 A vertex at (1.0, 0.0)]
```

```
>>> from sage.all import *
>>> perm_a2_inexact = polytopes.generalized_permutahedron(                #␣
↪needs sage.combinat
...     ['A',Integer(2)], exact=False)
>>> sorted(perm_a2_inexact.vertices())                                    #␣
↪needs sage.combinat
[A vertex at (-1.0, -1.0),
 A vertex at (-1.0, 0.0),
 A vertex at (0.0, -1.0),
 A vertex at (0.0, 1.0),
 A vertex at (1.0, 0.0),
 A vertex at (1.0, 1.0)]

>>> perm_a2_inexact_reg = polytopes.generalized_permutahedron(            #␣
↪needs sage.combinat
...     ['A',Integer(2)], exact=False, regular=True)
>>> sorted(perm_a2_inexact_reg.vertices())                                #␣
↪needs sage.combinat
[A vertex at (-1.0, 0.0),
 A vertex at (-0.5, -0.8660254038),
 A vertex at (-0.5, 0.8660254038),
 A vertex at (0.5, -0.8660254038),
```

(continues on next page)

(continued from previous page)

```
 A vertex at (0.5, 0.8660254038),
 A vertex at (1.0, 0.0)]
```

We can do the same computation using exact arithmetic with the field AA:

```
sage: perm_a2_reg = polytopes.generalized_permutahedron(                    #␣
↪needs sage.combinat sage.rings.number_field
....:     ['A',2], regular=True)
sage: V = sorted(perm_a2_reg.vertices()); V  # random                       #␣
↪needs sage.combinat sage.rings.number_field
[A vertex at (-1, 0),
 A vertex at (-1/2, -0.866025403784439?),
 A vertex at (-1/2, 0.866025403784439?),
 A vertex at (1/2, -0.866025403784439?),
 A vertex at (1/2, 0.866025403784439?),
 A vertex at (1.000000000000000?, 0.?e-18)]
```

```
>>> from sage.all import *
>>> perm_a2_reg = polytopes.generalized_permutahedron(                      #␣
↪needs sage.combinat sage.rings.number_field
...     ['A',Integer(2)], regular=True)
>>> V = sorted(perm_a2_reg.vertices()); V  # random                         #␣
↪needs sage.combinat sage.rings.number_field
[A vertex at (-1, 0),
 A vertex at (-1/2, -0.866025403784439?),
 A vertex at (-1/2, 0.866025403784439?),
 A vertex at (1/2, -0.866025403784439?),
 A vertex at (1/2, 0.866025403784439?),
 A vertex at (1.000000000000000?, 0.?e-18)]
```

Even though the numbers look like floating point approximations, the computation is actually exact. We can clean up the display a bit using exactify:

```
sage: for v in V:                                                           #␣
↪needs sage.combinat sage.rings.number_field
....:     for x in v:
....:         x.exactify()
sage: V                                                                     #␣
↪needs sage.combinat sage.rings.number_field
[A vertex at (-1, 0),
 A vertex at (-1/2, -0.866025403784439?),
 A vertex at (-1/2, 0.866025403784439?),
 A vertex at (1/2, -0.866025403784439?),
 A vertex at (1/2, 0.866025403784439?),
 A vertex at (1, 0)]
sage: perm_a2_reg.is_inscribed()                                            #␣
↪needs sage.combinat sage.rings.number_field
True
```

```
>>> from sage.all import *
>>> for v in V:                                                             #␣
↪needs sage.combinat sage.rings.number_field
```

(continues on next page)

(continued from previous page)

```
...     for x in v:
...         x.exactify()
>>> V                                                                    #␣
↪needs sage.combinat sage.rings.number_field
[A vertex at (-1, 0),
 A vertex at (-1/2, -0.866025403784439?),
 A vertex at (-1/2, 0.866025403784439?),
 A vertex at (1/2, -0.866025403784439?),
 A vertex at (1/2, 0.866025403784439?),
 A vertex at (1, 0)]
>>> perm_a2_reg.is_inscribed()                                           #␣
↪needs sage.combinat sage.rings.number_field
True
```

Larger examples take longer:

```
sage: # needs sage.combinat sage.rings.number_field
sage: perm_a3_reg = polytopes.generalized_permutahedron(    # long time
....:     ['A',3], regular=True); perm_a3_reg
A 3-dimensional polyhedron in AA^3 defined as the convex hull of 24 vertices
sage: perm_a3_reg.is_inscribed()     # long time
True
sage: perm_b3_reg = polytopes.generalized_permutahedron(    # long time␣
↪(12sec on 64 bits), not tested
....:     ['B',3], regular=True); perm_b3_reg
A 3-dimensional polyhedron in AA^3 defined as the convex hull of 48 vertices
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.rings.number_field
>>> perm_a3_reg = polytopes.generalized_permutahedron(    # long time
...     ['A',Integer(3)], regular=True); perm_a3_reg
A 3-dimensional polyhedron in AA^3 defined as the convex hull of 24 vertices
>>> perm_a3_reg.is_inscribed()     # long time
True
>>> perm_b3_reg = polytopes.generalized_permutahedron(    # long time (12sec␣
↪on 64 bits), not tested
...     ['B',Integer(3)], regular=True); perm_b3_reg
A 3-dimensional polyhedron in AA^3 defined as the convex hull of 48 vertices
```

It is faster with the backend `'number_field'`, which internally uses an embedded number field instead of doing the computations directly with the base ring (`AA`):

```
sage: # needs sage.combinat sage.rings.number_field
sage: perm_a3_reg_nf = polytopes.generalized_permutahedron(
....:    ['A',3], regular=True, backend='number_field'); perm_a3_reg_nf
A 3-dimensional polyhedron in AA^3 defined as the convex hull of 24 vertices
sage: perm_a3_reg_nf.is_inscribed()
True
sage: perm_b3_reg_nf = polytopes.generalized_permutahedron(         # long␣
↪time
....:    ['B',3], regular=True, backend='number_field'); perm_b3_reg_nf
A 3-dimensional polyhedron in AA^3 defined as the convex hull of 48 vertices
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.rings.number_field
>>> perm_a3_reg_nf = polytopes.generalized_permutahedron(
...     ['A',Integer(3)], regular=True, backend='number_field'); perm_a3_reg_nf
A 3-dimensional polyhedron in AA^3 defined as the convex hull of 24 vertices
>>> perm_a3_reg_nf.is_inscribed()
True
>>> perm_b3_reg_nf = polytopes.generalized_permutahedron(         # long time
...     ['B',Integer(3)], regular=True, backend='number_field'); perm_b3_reg_nf
A 3-dimensional polyhedron in AA^3 defined as the convex hull of 48 vertices
```

It is even faster with the backend `'normaliz'`:

```
sage: # optional - pynormaliz, needs sage.combinat sage.rings.number_field
sage: perm_a3_reg_norm = polytopes.generalized_permutahedron(
....:     ['A',3], regular=True, backend='normaliz'); perm_a3_reg_norm
A 3-dimensional polyhedron in AA^3 defined as the convex hull of 24 vertices
sage: perm_a3_reg_norm.is_inscribed()
True
sage: perm_b3_reg_norm = polytopes.generalized_permutahedron(
....:     ['B',3], regular=True, backend='normaliz'); perm_b3_reg_norm
A 3-dimensional polyhedron in AA^3 defined as the convex hull of 48 vertices
```

```
>>> from sage.all import *
>>> # optional - pynormaliz, needs sage.combinat sage.rings.number_field
>>> perm_a3_reg_norm = polytopes.generalized_permutahedron(
...     ['A',Integer(3)], regular=True, backend='normaliz'); perm_a3_reg_norm
A 3-dimensional polyhedron in AA^3 defined as the convex hull of 24 vertices
>>> perm_a3_reg_norm.is_inscribed()
True
>>> perm_b3_reg_norm = polytopes.generalized_permutahedron(
...     ['B',Integer(3)], regular=True, backend='normaliz'); perm_b3_reg_norm
A 3-dimensional polyhedron in AA^3 defined as the convex hull of 48 vertices
```

The speedups from using backend `'normaliz'` allow us to go even further:

```
sage: # optional - pynormaliz, needs sage.combinat sage.rings.number_field
sage: perm_h3 = polytopes.generalized_permutahedron(
....:      ['H',3], backend='normaliz'); perm_h3
A 3-dimensional polyhedron in
 (Number Field in a with defining polynomial x^2 - 5 with a = 2.
↪236067977499790?)^3
 defined as the convex hull of 120 vertices
sage: perm_f4 = polytopes.generalized_permutahedron(        # long time
....:      ['F',4], backend='normaliz'); perm_f4
A 4-dimensional polyhedron
 in (Number Field in a with defining polynomial x^2 - 2 with a = 1.
↪414213562373095?)^4
 defined as the convex hull of 1152 vertices
```

```
>>> from sage.all import *
>>> # optional - pynormaliz, needs sage.combinat sage.rings.number_field
>>> perm_h3 = polytopes.generalized_permutahedron(
```

(continues on next page)

(continued from previous page)

```
...     ['H',Integer(3)], backend='normaliz'); perm_h3
A 3-dimensional polyhedron in
 (Number Field in a with defining polynomial x^2 - 5 with a = 2.
↪236067977499790?)^3
 defined as the convex hull of 120 vertices
>>> perm_f4 = polytopes.generalized_permutahedron(        # long time
...     ['F',Integer(4)], backend='normaliz'); perm_f4
A 4-dimensional polyhedron
 in (Number Field in a with defining polynomial x^2 - 2 with a = 1.
↪414213562373095?)^4
 defined as the convex hull of 1152 vertices
```

See also

- `permutahedron()`
- `permutahedron()`

grand_antiprism(*exact=True*, *backend=None*, *verbose=False*)

Return the grand antiprism.

The grand antiprism is a 4-dimensional non-Wythoffian uniform polytope. The coordinates were taken from http://eusebeia.dyndns.org/4d/gap. For more information, see the Wikipedia article Grand_antiprism.

Warning

The coordinates are exact by default. The computation with exact coordinates is not as fast as with floating point approximations. If you find this method to be too slow, consider using floating point approximations

INPUT:

- `exact` – boolean (default: `True`); if `False` use floating point approximations instead of exact coordinates
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: gap = polytopes.grand_antiprism()  # not tested - very long time
sage: gap                                # not tested - very long time
A 4-dimensional polyhedron in (Number Field in sqrt5 with defining
polynomial x^2 - 5 with sqrt5 = 2.236067977499790?)^4 defined as
the convex hull of 100 vertices
```

```
>>> from sage.all import *
>>> gap = polytopes.grand_antiprism()  # not tested - very long time
>>> gap                                # not tested - very long time
A 4-dimensional polyhedron in (Number Field in sqrt5 with defining
polynomial x^2 - 5 with sqrt5 = 2.236067977499790?)^4 defined as
the convex hull of 100 vertices
```

Computation with the backend `'normaliz'` is instantaneous:

```
sage: gap_norm = polytopes.grand_antiprism(backend='normaliz')      #␣
↪optional - pynormaliz, needs sage.rings.number_field
sage: gap_norm                                                        #␣
↪optional - pynormaliz, needs sage.rings.number_field
A 4-dimensional polyhedron in (Number Field in sqrt5 with defining
polynomial x^2 - 5 with sqrt5 = 2.236067977499790?)^4 defined as
the convex hull of 100 vertices
```

```
>>> from sage.all import *
>>> gap_norm = polytopes.grand_antiprism(backend='normaliz')      # optional -
↪ pynormaliz, needs sage.rings.number_field
>>> gap_norm                                                        # optional -
↪ pynormaliz, needs sage.rings.number_field
A 4-dimensional polyhedron in (Number Field in sqrt5 with defining
polynomial x^2 - 5 with sqrt5 = 2.236067977499790?)^4 defined as
the convex hull of 100 vertices
```

Computation with approximated coordinates is also faster, but inexact:

```
sage: gap = polytopes.grand_antiprism(exact=False) # random
sage: gap
A 4-dimensional polyhedron in RDF^4 defined as the convex hull of 100 vertices
sage: gap.f_vector()
(1, 100, 500, 720, 320, 1)
sage: len(list(gap.bounded_edges()))
500
```

```
>>> from sage.all import *
>>> gap = polytopes.grand_antiprism(exact=False) # random
>>> gap
A 4-dimensional polyhedron in RDF^4 defined as the convex hull of 100 vertices
>>> gap.f_vector()
(1, 100, 500, 720, 320, 1)
>>> len(list(gap.bounded_edges()))
500
```

great_rhombicuboctahedron(*exact=True*, *base_ring=None*, *backend=None*)

Return the great rhombicuboctahedron.

The great rhombicuboctahedron (or truncated cuboctahedron) is an Archimedean solid with 48 vertices and 26 faces. For more information see the Wikipedia article Truncated_cuboctahedron.

INPUT:

- `exact` – boolean (default: `True`); if `False` use an approximate ring for the coordinates
- `base_ring` – the ring in which the coordinates will belong to. If it is not provided and `exact=True` it will be a the number field $\mathbf{Q}[\phi]$ where ϕ is the golden ratio and if `exact=False` it will be the real double field.
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: gr = polytopes.great_rhombicuboctahedron()    # long time             #␣
↪needs sage.rings.number_field
sage: gr.f_vector()                     # long time                         #␣
↪needs sage.rings.number_field
(1, 48, 72, 26, 1)
```

```
>>> from sage.all import *
>>> gr = polytopes.great_rhombicuboctahedron()    # long time             #␣
↪needs sage.rings.number_field
>>> gr.f_vector()                     # long time                         #␣
↪needs sage.rings.number_field
(1, 48, 72, 26, 1)
```

A faster implementation is obtained by setting `exact=False`:

```
sage: gr = polytopes.great_rhombicuboctahedron(exact=False)
sage: gr.f_vector()
(1, 48, 72, 26, 1)
```

```
>>> from sage.all import *
>>> gr = polytopes.great_rhombicuboctahedron(exact=False)
>>> gr.f_vector()
(1, 48, 72, 26, 1)
```

Its facets are 4 squares, 8 regular hexagons and 6 regular octagons:

```
sage: sum(1 for f in gr.facets() if len(f.vertices()) == 4)
12
sage: sum(1 for f in gr.facets() if len(f.vertices()) == 6)
8
sage: sum(1 for f in gr.facets() if len(f.vertices()) == 8)
6
```

```
>>> from sage.all import *
>>> sum(Integer(1) for f in gr.facets() if len(f.vertices()) == Integer(4))
12
>>> sum(Integer(1) for f in gr.facets() if len(f.vertices()) == Integer(6))
8
>>> sum(Integer(1) for f in gr.facets() if len(f.vertices()) == Integer(8))
6
```

hypercube(*dim*, *intervals=None*, *backend=None*)

Return a hypercube of the given dimension.

The `dim`-dimensional hypercube is by default the convex hull of the 2^{dim} ± 1 vectors of length `dim`. Alternatively, it is the product of `dim` line segments given in the `intervals`. For more information see the wikipedia article Wikipedia article Hypercube.

INPUT:

- `dim` – integer; the dimension of the hypercube
- `intervals` – (default: `None`) it takes the following possible inputs:
  - If `None` (the default), it returns the ± 1-cube of dimension `dim`.

  - 'zero_one' – string; return the $0/1$-cube
  - a list of length dim. Its elements are pairs of numbers (a, b) with $a < b$. The cube will be the product of these intervals.

- backend – the backend to use to create the polytope

EXAMPLES:

Create the ± 1-hypercube of dimension 4:

```
sage: four_cube = polytopes.hypercube(4)
sage: four_cube.is_simple()
True
sage: four_cube.base_ring()
Integer Ring
sage: four_cube.volume()
16
sage: four_cube.ehrhart_polynomial()     # optional - latte_int
16*t^4 + 32*t^3 + 24*t^2 + 8*t + 1
```

```
>>> from sage.all import *
>>> four_cube = polytopes.hypercube(Integer(4))
>>> four_cube.is_simple()
True
>>> four_cube.base_ring()
Integer Ring
>>> four_cube.volume()
16
>>> four_cube.ehrhart_polynomial()     # optional - latte_int
16*t^4 + 32*t^3 + 24*t^2 + 8*t + 1
```

Return the $0/1$-hypercube of dimension 4:

```
sage: z_cube = polytopes.hypercube(4, intervals='zero_one')
sage: z_cube.vertices()[0]
A vertex at (1, 0, 1, 1)
sage: z_cube.is_simple()
True
sage: z_cube.base_ring()
Integer Ring
sage: z_cube.volume()
1
sage: z_cube.ehrhart_polynomial()       # optional - latte_int
t^4 + 4*t^3 + 6*t^2 + 4*t + 1
```

```
>>> from sage.all import *
>>> z_cube = polytopes.hypercube(Integer(4), intervals='zero_one')
>>> z_cube.vertices()[Integer(0)]
A vertex at (1, 0, 1, 1)
>>> z_cube.is_simple()
True
>>> z_cube.base_ring()
Integer Ring
>>> z_cube.volume()
```

(continues on next page)

(continued from previous page)

```
1
>>> z_cube.ehrhart_polynomial()        # optional - latte_int
t^4 + 4*t^3 + 6*t^2 + 4*t + 1
```

Return the 4-dimensional combinatorial cube that is the product of [0,3]^4:

```
sage: t_cube = polytopes.hypercube(4, intervals=[[0,3]]*4)
```

```
>>> from sage.all import *
>>> t_cube = polytopes.hypercube(Integer(4), intervals=[[Integer(0),
↪Integer(3)]]*Integer(4))
```

Checking that t_cube is three times the previous $0/1$-cube:

```
sage: t_cube == 3 * z_cube
True
```

```
>>> from sage.all import *
>>> t_cube == Integer(3) * z_cube
True
```

hypersimplex(*dim*, *k*, *project=False*, *backend=None*)

Return the hypersimplex in dimension `dim` and parameter `k`.

The hypersimplex $\Delta_{d,k}$ is the convex hull of the vertices made of k ones and $d-k$ zeros. It lies in the $d-1$ hyperplane of vectors of sum k. If you want a projected version to $\mathbf{R}^{d-1}$ (with floating point coordinates) then set `project=True` in the options.

See also

simplex()

INPUT:

- `dim` – the dimension
- `n` – the numbers `(1,...,n)` are permuted
- `project` – boolean (default: `False`); if `True`, the polytope is (isometrically) projected to a vector space of dimension `dim-1`. This operation turns the coordinates into floating point approximations and corresponds to the projection given by the matrix from *zero_sum_projection()*.
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: # needs sage.combinat
sage: h_4_2 = polytopes.hypersimplex(4, 2)
sage: h_4_2
A 3-dimensional polyhedron in ZZ^4 defined as the convex hull of 6 vertices
sage: h_4_2.f_vector()
(1, 6, 12, 8, 1)
sage: h_4_2.ehrhart_polynomial()                        # optional -␣
↪latte_int
```

(continues on next page)

(continued from previous page)

```
2/3*t^3 + 2*t^2 + 7/3*t + 1
sage: TestSuite(h_4_2).run()

sage: # needs sage.combinat
sage: h_7_3 = polytopes.hypersimplex(7, 3, project=True)
sage: h_7_3
A 6-dimensional polyhedron in RDF^6 defined as the convex hull of 35 vertices
sage: h_7_3.f_vector()
(1, 35, 210, 350, 245, 84, 14, 1)
sage: TestSuite(h_7_3).run(skip=["_test_pyramid", "_test_lawrence"])
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> h_4_2 = polytopes.hypersimplex(Integer(4), Integer(2))
>>> h_4_2
A 3-dimensional polyhedron in ZZ^4 defined as the convex hull of 6 vertices
>>> h_4_2.f_vector()
(1, 6, 12, 8, 1)
>>> h_4_2.ehrhart_polynomial()                          # optional - latte_
↪int
2/3*t^3 + 2*t^2 + 7/3*t + 1
>>> TestSuite(h_4_2).run()

>>> # needs sage.combinat
>>> h_7_3 = polytopes.hypersimplex(Integer(7), Integer(3), project=True)
>>> h_7_3
A 6-dimensional polyhedron in RDF^6 defined as the convex hull of 35 vertices
>>> h_7_3.f_vector()
(1, 35, 210, 350, 245, 84, 14, 1)
>>> TestSuite(h_7_3).run(skip=["_test_pyramid", "_test_lawrence"])
```

icosahedron(*exact=True*, *base_ring=None*, *backend=None*)

Return an icosahedron with edge length 1.

The icosahedron is one of the Platonic solids. It has 20 faces and is dual to the *dodecahedron()*.

INPUT:

- `exact` – boolean (default: `True`); if `False` use an approximate ring for the coordinates
- `base_ring` – (optional) the ring in which the coordinates will belong to. Note that this ring must contain $\sqrt(5)$. If it is not provided and `exact=True` it will be the number field $\mathbf{Q}[\sqrt(5)]$ and if `exact=False` it will be the real double field.
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: ico = polytopes.icosahedron()                                   #␣
↪needs sage.rings.number_field
sage: ico.f_vector()                                                  #␣
↪needs sage.rings.number_field
(1, 12, 30, 20, 1)
sage: ico.volume()                                                    #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.rings.number_field
5/12*sqrt5 + 5/4
```

```
>>> from sage.all import *
>>> ico = polytopes.icosahedron()                                           #␣
↪needs sage.rings.number_field
>>> ico.f_vector()                                                          #␣
↪needs sage.rings.number_field
(1, 12, 30, 20, 1)
>>> ico.volume()                                                            #␣
↪needs sage.rings.number_field
5/12*sqrt5 + 5/4
```

Its non exact version:

```
sage: ico = polytopes.icosahedron(exact=False)                              #␣
↪needs sage.groups
sage: ico.base_ring()                                                       #␣
↪needs sage.groups
Real Double Field
sage: ico.volume()                  # known bug                             #␣
↪needs sage.groups
2.181694990...
```

```
>>> from sage.all import *
>>> ico = polytopes.icosahedron(exact=False)                                #␣
↪needs sage.groups
>>> ico.base_ring()                                                         #␣
↪needs sage.groups
Real Double Field
>>> ico.volume()                  # known bug                               #␣
↪needs sage.groups
2.181694990...
```

A version using $AA < sage.rings.qqbar.AlgebraicRealField >$:

```
sage: ico = polytopes.icosahedron(base_ring=AA)     # long time             #␣
↪needs sage.groups sage.rings.number_field
sage: ico.base_ring()                               # long time             #␣
↪needs sage.groups sage.rings.number_field
Algebraic Real Field
sage: ico.volume()                                  # long time             #␣
↪needs sage.groups sage.rings.number_field
2.181694990624913?
```

```
>>> from sage.all import *
>>> ico = polytopes.icosahedron(base_ring=AA)     # long time               #␣
↪needs sage.groups sage.rings.number_field
>>> ico.base_ring()                               # long time               #␣
↪needs sage.groups sage.rings.number_field
Algebraic Real Field
>>> ico.volume()                                  # long time               #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.groups sage.rings.number_field
2.181694990624913?
```

Note that if base ring is provided it must contain the square root of 5. Otherwise you will get an error:

```
sage: polytopes.icosahedron(base_ring=QQ)                                    #␣
↪needs sage.symbolic
Traceback (most recent call last):
...
TypeError: unable to convert 1/4*sqrt(5) + 1/4 to a rational
```

```
>>> from sage.all import *
>>> polytopes.icosahedron(base_ring=QQ)                                     #␣
↪needs sage.symbolic
Traceback (most recent call last):
...
TypeError: unable to convert 1/4*sqrt(5) + 1/4 to a rational
```

icosidodecahedron(*exact=True*, *backend=None*)

Return the icosidodecahedron.

The Icosidodecahedron is a polyhedron with twenty triangular faces and twelve pentagonal faces. For more information see the Wikipedia article Icosidodecahedron.

INPUT:

- `exact` – boolean (default: `True`); if `False` use an approximate ring for the coordinates
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: id = polytopes.icosidodecahedron()                                     #␣
↪needs sage.groups sage.rings.number_field
sage: id.f_vector()                                                          #␣
↪needs sage.groups sage.rings.number_field
(1, 30, 60, 32, 1)
```

```
>>> from sage.all import *
>>> id = polytopes.icosidodecahedron()                                      #␣
↪needs sage.groups sage.rings.number_field
>>> id.f_vector()                                                           #␣
↪needs sage.groups sage.rings.number_field
(1, 30, 60, 32, 1)
```

icosidodecahedron_V2(*exact=True*, *base_ring=None*, *backend=None*)

Return the icosidodecahedron.

The icosidodecahedron is an Archimedean solid. It has 32 faces and 30 vertices. For more information, see the Wikipedia article Icosidodecahedron.

INPUT:

- `exact` – boolean (default: `True`); if `False` use an approximate ring for the coordinates

- `base_ring` – the ring in which the coordinates will belong to If it is not provided and `exact=True` it will be a the number field $\mathbf{Q}[\phi]$ where ϕ is the golden ratio and if `exact=False` it will be the real double field.
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: id = polytopes.icosidodecahedron_V2()    # long time (6s)
sage: id.f_vector()                            # long time
(1, 30, 60, 32, 1)
sage: id.base_ring()                           # long time
Number Field in sqrt5 with defining polynomial x^2 - 5
 with sqrt5 = 2.236067977499790?
```

```
>>> from sage.all import *
>>> id = polytopes.icosidodecahedron_V2()    # long time (6s)
>>> id.f_vector()                            # long time
(1, 30, 60, 32, 1)
>>> id.base_ring()                           # long time
Number Field in sqrt5 with defining polynomial x^2 - 5
 with sqrt5 = 2.236067977499790?
```

A much faster implementation using floating point approximations:

```
sage: id = polytopes.icosidodecahedron_V2(exact=False)
sage: id.f_vector()
(1, 30, 60, 32, 1)
sage: id.base_ring()
Real Double Field
```

```
>>> from sage.all import *
>>> id = polytopes.icosidodecahedron_V2(exact=False)
>>> id.f_vector()
(1, 30, 60, 32, 1)
>>> id.base_ring()
Real Double Field
```

Its facets are 20 triangles and 12 regular pentagons:

```
sage: sum(1 for f in id.facets() if len(f.vertices()) == 3)
20
sage: sum(1 for f in id.facets() if len(f.vertices()) == 5)
12
```

```
>>> from sage.all import *
>>> sum(Integer(1) for f in id.facets() if len(f.vertices()) == Integer(3))
20
>>> sum(Integer(1) for f in id.facets() if len(f.vertices()) == Integer(5))
12
```

octahedron(*backend=None*)

Return the octahedron.

The octahedron is a Platonic solid with 6 vertices and 8 faces dual to the cube. It can be defined as the convex hull of the six vertices $(0, 0, \pm 1)$, $(\pm 1, 0, 0)$ and $(0, \pm 1, 0)$. For more information, see the Wikipedia article Octahedron.

INPUT:

- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: co = polytopes.octahedron()
sage: co.f_vector()
(1, 6, 12, 8, 1)
```

```
>>> from sage.all import *
>>> co = polytopes.octahedron()
>>> co.f_vector()
(1, 6, 12, 8, 1)
```

Its facets are 8 triangles:

```
sage: sum(1 for f in co.facets() if len(f.vertices()) == 3)
8
```

```
>>> from sage.all import *
>>> sum(Integer(1) for f in co.facets() if len(f.vertices()) == Integer(3))
8
```

Some more computation:

```
sage: co.volume()
4/3
sage: co.ehrhart_polynomial()      # optional - latte_int
4/3*t^3 + 2*t^2 + 8/3*t + 1
```

```
>>> from sage.all import *
>>> co.volume()
4/3
>>> co.ehrhart_polynomial()      # optional - latte_int
4/3*t^3 + 2*t^2 + 8/3*t + 1
```

omnitruncated_one_hundred_twenty_cell(*exact=True*, *backend=None*)

Return the omnitruncated 120-cell.

The omnitruncated 120-cell is a 4-dimensional 4-uniform polytope in the H_4 family. It has 14400 vertices. For more information see Wikipedia article Omnitruncated 120-cell.

> **Warning**
>
> The coordinates are exact by default. The computation with inexact coordinates (using the backend `'cdd'`) returns a numerical inconsistency error, and thus cannot be computed.

INPUT:

- `exact` – boolean (default: `True`); if `True` use exact coordinates instead of floating point approximations

- backend – the backend to use to create the polytope

EXAMPLES:

```
sage: polytopes.omnitruncated_one_hundred_twenty_cell(backend='normaliz')   #␣
↪not tested - very long time ~10min
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 14400␣
↪vertices
```

```
>>> from sage.all import *
>>> polytopes.omnitruncated_one_hundred_twenty_cell(backend='normaliz')   #␣
↪not tested - very long time ~10min
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 14400␣
↪vertices
```

omnitruncated_six_hundred_cell(*exact=True*, *backend=None*)

Return the omnitruncated 120-cell.

The omnitruncated 120-cell is a 4-dimensional 4-uniform polytope in the H_4 family. It has 14400 vertices. For more information see Wikipedia article Omnitruncated 120-cell.

> **Warning**
>
> The coordinates are exact by default. The computation with inexact coordinates (using the backend 'cdd') returns a numerical inconsistency error, and thus cannot be computed.

INPUT:

- exact – boolean (default: True); if True use exact coordinates instead of floating point approximations
- backend – the backend to use to create the polytope

EXAMPLES:

```
sage: polytopes.omnitruncated_one_hundred_twenty_cell(backend='normaliz')   #␣
↪not tested - very long time ~10min
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 14400␣
↪vertices
```

```
>>> from sage.all import *
>>> polytopes.omnitruncated_one_hundred_twenty_cell(backend='normaliz')   #␣
↪not tested - very long time ~10min
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 14400␣
↪vertices
```

one_hundred_twenty_cell(*exact=True*, *backend=None*, *construction='coxeter'*)

Return the 120-cell.

The 120-cell is a 4-dimensional 4-uniform polytope in the H_4 family. It has 600 vertices and 120 facets. For more information see Wikipedia article 120-cell.

> **Warning**

The coordinates are exact by default. The computation with inexact coordinates (using the backend `'cdd'`) returns a numerical inconsistency error, and thus cannot be computed.

INPUT:

- `exact` – boolean (default: `True`); if `True` use exact coordinates instead of floating point approximations
- `backend` – the backend to use to create the polytope
- `construction` – string (default: `'coxeter'`); the construction to use. The other possibility is 'as_permutahedron'.

EXAMPLES:

The classical construction given by Coxeter in [Cox1969] is given by:

```
sage: polytopes.one_hundred_twenty_cell()                    # not tested,␣
↪long time (~15s)
A 4-dimensional polyhedron in (Number Field in sqrt5 with defining
 polynomial x^2 - 5 with sqrt5 = 2.236067977499790?)^4 defined as
 the convex hull of 600 vertices
```

```
>>> from sage.all import *
>>> polytopes.one_hundred_twenty_cell()                    # not tested, long␣
↪time (~15s)
A 4-dimensional polyhedron in (Number Field in sqrt5 with defining
 polynomial x^2 - 5 with sqrt5 = 2.236067977499790?)^4 defined as
 the convex hull of 600 vertices
```

The `'normaliz'` is faster:

```
sage: P = polytopes.one_hundred_twenty_cell(backend='normaliz'); P   #␣
↪optional - pynormaliz
A 4-dimensional polyhedron in (Number Field in sqrt5 with defining
 polynomial x^2 - 5 with sqrt5 = 2.236067977499790?)^4 defined as
 the convex hull of 600 vertices
```

```
>>> from sage.all import *
>>> P = polytopes.one_hundred_twenty_cell(backend='normaliz'); P   # optional -
↪ pynormaliz
A 4-dimensional polyhedron in (Number Field in sqrt5 with defining
 polynomial x^2 - 5 with sqrt5 = 2.236067977499790?)^4 defined as
 the convex hull of 600 vertices
```

It is also possible to realize it using the generalized permutahedron of type H_4:

```
sage: polytopes.one_hundred_twenty_cell(backend='normaliz',  # not tested -␣
↪long time
....:                                  construction='as_permutahedron')
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 600 vertices
```

```
>>> from sage.all import *
>>> polytopes.one_hundred_twenty_cell(backend='normaliz',  # not tested -␣
↪long time
```

(continues on next page)

(continued from previous page)

```
...                                      construction='as_permutahedron')
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 600 vertices
```

parallelotope (*generators*, *backend=None*)

Return the zonotope, or parallelotope, spanned by the generators.

The parallelotope is the multi-dimensional generalization of a parallelogram (2 generators) and a parallelepiped (3 generators).

INPUT:

- `generators` – list of vectors of same dimension
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: polytopes.parallelotope([ (1,0), (0,1) ])
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices
sage: polytopes.parallelotope([[1,2,3,4], [0,1,0,7], [3,1,0,2], [0,0,1,0]])
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 16 vertices

sage: K = QuadraticField(2, 'sqrt2')                                        #␣
↪needs sage.rings.number_field
sage: sqrt2 = K.gen()                                                       #␣
↪needs sage.rings.number_field
sage: P = polytopes.parallelotope([(1, sqrt2), (1, -1)]); P                 #␣
↪needs sage.rings.number_field
A 2-dimensional polyhedron in (Number Field in sqrt2 with defining
polynomial x^2 - 2 with sqrt2 = 1.414213562373095?)^2 defined as
the convex hull of 4 vertices
```

```
>>> from sage.all import *
>>> polytopes.parallelotope([ (Integer(1),Integer(0)), (Integer(0),
↪Integer(1)) ])
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices
>>> polytopes.parallelotope([[Integer(1),Integer(2),Integer(3),Integer(4)],␣
↪[Integer(0),Integer(1),Integer(0),Integer(7)], [Integer(3),Integer(1),
↪Integer(0),Integer(2)], [Integer(0),Integer(0),Integer(1),Integer(0)]])
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 16 vertices

>>> K = QuadraticField(Integer(2), 'sqrt2')                                ␣
↪     # needs sage.rings.number_field
>>> sqrt2 = K.gen()                                                       #␣
↪needs sage.rings.number_field
>>> P = polytopes.parallelotope([(Integer(1), sqrt2), (Integer(1), -
↪Integer(1))]); P                # needs sage.rings.number_field
A 2-dimensional polyhedron in (Number Field in sqrt2 with defining
polynomial x^2 - 2 with sqrt2 = 1.414213562373095?)^2 defined as
the convex hull of 4 vertices
```

pentakis_dodecahedron (*exact=True*, *base_ring=None*, *backend=None*)

Return the pentakis dodecahedron.

The pentakis dodecahedron (orkisdodecahedron) is a face-regular, vertex-uniform polytope dual to the truncated icosahedron. It has 60 facets and 32 vertices. See the Wikipedia article Pentakis_dodecahedron for more information.

INPUT:

- `exact` – boolean (default: `True`); if `False` use an approximate ring for the coordinates
- `base_ring` – the ring in which the coordinates will belong to. If it is not provided and `exact=True` it will be a the number field $\mathbf{Q}[\phi]$ where ϕ is the golden ratio and if `exact=False` it will be the real double field.
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: pd = polytopes.pentakis_dodecahedron()     # long time (10s)
sage: pd.n_vertices()                            # long time
32
sage: pd.n_inequalities()                        # long time
60
```

```
>>> from sage.all import *
>>> pd = polytopes.pentakis_dodecahedron()     # long time (10s)
>>> pd.n_vertices()                            # long time
32
>>> pd.n_inequalities()                        # long time
60
```

A much faster implementation is obtained when setting `exact=False`:

```
sage: pd = polytopes.pentakis_dodecahedron(exact=False)                     #␣
↪needs sage.groups
sage: pd.n_vertices()                                                       #␣
↪needs sage.groups
32
sage: pd.n_inequalities()                                                   #␣
↪needs sage.groups
60
```

```
>>> from sage.all import *
>>> pd = polytopes.pentakis_dodecahedron(exact=False)                     #␣
↪needs sage.groups
>>> pd.n_vertices()                                                       #␣
↪needs sage.groups
32
>>> pd.n_inequalities()                                                   #␣
↪needs sage.groups
60
```

The 60 are triangles:

```
sage: all(len(f.vertices()) == 3 for f in pd.facets())                      #␣
↪needs sage.groups
True
```

```
>>> from sage.all import *
>>> all(len(f.vertices()) == Integer(3) for f in pd.facets())                    ␣
↪     # needs sage.groups
True
```

permutahedron(*n*, *project=False*, *backend=None*)

Return the standard permutahedron of (1,…,n).

The permutahedron (or permutohedron) is the convex hull of the permutations of $\{1, \ldots, n\}$ seen as vectors. The edges between the permutations correspond to multiplication on the right by an elementary transposition in the `SymmetricGroup`.

If we take the graph in which the vertices correspond to vertices of the polyhedron, and edges to edges, we get the `BubbleSortGraph()`.

INPUT:

- `n` – integer
- `project` – boolean (default: `False`); if `True`, the polytope is (isometrically) projected to a vector space of dimension `dim-1`. This operation turns the coordinates into floating point approximations and corresponds to the projection given by the matrix from `zero_sum_projection()`.
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: perm4 = polytopes.permutahedron(4)
sage: perm4
A 3-dimensional polyhedron in ZZ^4 defined as the convex hull of 24 vertices
sage: perm4.is_lattice_polytope()
True
sage: perm4.ehrhart_polynomial()   # optional - latte_int
16*t^3 + 15*t^2 + 6*t + 1

sage: perm4 = polytopes.permutahedron(4, project=True)
sage: perm4
A 3-dimensional polyhedron in RDF^3 defined as the convex hull of 24 vertices
sage: perm4.plot()                                                              #␣
↪needs sage.plot
Graphics3d Object
sage: perm4.graph().is_isomorphic(graphs.BubbleSortGraph(4))                    #␣
↪needs sage.graphs
True
```

```
>>> from sage.all import *
>>> perm4 = polytopes.permutahedron(Integer(4))
>>> perm4
A 3-dimensional polyhedron in ZZ^4 defined as the convex hull of 24 vertices
>>> perm4.is_lattice_polytope()
True
>>> perm4.ehrhart_polynomial()   # optional - latte_int
16*t^3 + 15*t^2 + 6*t + 1

>>> perm4 = polytopes.permutahedron(Integer(4), project=True)
>>> perm4
```

(continues on next page)

(continued from previous page)

```
A 3-dimensional polyhedron in RDF^3 defined as the convex hull of 24 vertices
>>> perm4.plot()                                                              #␣
↪needs sage.plot
Graphics3d Object
>>> perm4.graph().is_isomorphic(graphs.BubbleSortGraph(Integer(4)))          ␣
↪      # needs sage.graphs
True
```

As both Hrepresentation and Vrepresentation are known, the permutahedron can be set up with both using the backend `field`. The following takes very very long time to recompute, e.g. with backend `ppl`:

```
sage: polytopes.permutahedron(8, backend='field')   # (~1s)
A 7-dimensional polyhedron in QQ^8 defined as the convex hull of 40320␣
↪vertices
sage: polytopes.permutahedron(9, backend='field')   # not tested (memory␣
↪consumption)   # (~5s)
A 8-dimensional polyhedron in QQ^9 defined as the convex hull of 362880␣
↪vertices
```

```
>>> from sage.all import *
>>> polytopes.permutahedron(Integer(8), backend='field')   # (~1s)
A 7-dimensional polyhedron in QQ^8 defined as the convex hull of 40320␣
↪vertices
>>> polytopes.permutahedron(Integer(9), backend='field')   # not tested␣
↪(memory consumption)   # (~5s)
A 8-dimensional polyhedron in QQ^9 defined as the convex hull of 362880␣
↪vertices
```

See also

- `BubbleSortGraph()`

`rectified_one_hundred_twenty_cell`(*exact=True*, *backend=None*)

Return the rectified 120-cell.

The rectified 120-cell is a 4-dimensional 4-uniform polytope in the H_4 family. It has 1200 vertices. For more information see Wikipedia article Rectified 120-cell.

Warning

The coordinates are exact by default. The computation with inexact coordinates (using the backend `'cdd'`) returns a numerical inconsistency error, and thus cannot be computed.

INPUT:

- `exact` – boolean (default: `True`); if `True` use exact coordinates instead of floating point approximations
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: polytopes.rectified_one_hundred_twenty_cell(backend='normaliz')  # not␣
↪tested - long time
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 1200 vertices
```

```
>>> from sage.all import *
>>> polytopes.rectified_one_hundred_twenty_cell(backend='normaliz')  # not␣
↪tested - long time
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 1200 vertices
```

rectified_six_hundred_cell(*exact=True*, *backend=None*)

Return the rectified 600-cell.

The rectified 600-cell is a 4-dimensional 4-uniform polytope in the H_4 family. It has 720 vertices. For more information see Wikipedia article Rectified 600-cell.

> **Warning**
>
> The coordinates are exact by default. The computation with inexact coordinates (using the backend `'cdd'`) returns a numerical inconsistency error, and thus cannot be computed.

INPUT:

- `exact` – boolean (default: `True`); if `True` use exact coordinates instead of floating point approximations
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: polytopes.rectified_six_hundred_cell(backend='normaliz')                #␣
↪not tested, long time (14s)
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 720 vertices
```

```
>>> from sage.all import *
>>> polytopes.rectified_six_hundred_cell(backend='normaliz')                #␣
↪not tested, long time (14s)
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 720 vertices
```

regular_polygon(*n*, *exact=True*, *base_ring=None*, *backend=None*)

Return a regular polygon with n vertices.

INPUT:

- `n` – positive integer; the number of vertices
- `exact` – boolean (default: `True`); if `False` floating point numbers are used for coordinates
- `base_ring` – a ring in which the coordinates will lie. It is `None` by default. If it is not provided and `exact` is `True` then it will be the field of real algebraic number, if `exact` is `False` it will be the real double field.
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: octagon = polytopes.regular_polygon(8)
sage: octagon
A 2-dimensional polyhedron in AA^2 defined as the convex hull of 8 vertices
sage: octagon.n_vertices()
8
sage: v = octagon.volume()
sage: v
2.828427124746190?
sage: v == 2*QQbar(2).sqrt()
True
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> octagon = polytopes.regular_polygon(Integer(8))
>>> octagon
A 2-dimensional polyhedron in AA^2 defined as the convex hull of 8 vertices
>>> octagon.n_vertices()
8
>>> v = octagon.volume()
>>> v
2.828427124746190?
>>> v == Integer(2)*QQbar(Integer(2)).sqrt()
True
```

Its non exact version:

```
sage: polytopes.regular_polygon(3, exact=False).vertices()
(A vertex at (0.0, 1.0),
 A vertex at (0.8660254038, -0.5),
 A vertex at (-0.8660254038, -0.5))
sage: polytopes.regular_polygon(25, exact=False).n_vertices()
25
```

```
>>> from sage.all import *
>>> polytopes.regular_polygon(Integer(3), exact=False).vertices()
(A vertex at (0.0, 1.0),
 A vertex at (0.8660254038, -0.5),
 A vertex at (-0.8660254038, -0.5))
>>> polytopes.regular_polygon(Integer(25), exact=False).n_vertices()
25
```

rhombic_dodecahedron(*backend=None*)

Return the rhombic dodecahedron.

The rhombic dodecahedron is a polytope dual to the cuboctahedron. It has 14 vertices and 12 faces. For more information see the Wikipedia article Rhombic_dodecahedron.

INPUT:

- `backend` – the backend to use to create the polytope

See also

cuboctahedron()

EXAMPLES:

```
sage: rd = polytopes.rhombic_dodecahedron()
sage: rd.f_vector()
(1, 14, 24, 12, 1)
```

```
>>> from sage.all import *
>>> rd = polytopes.rhombic_dodecahedron()
>>> rd.f_vector()
(1, 14, 24, 12, 1)
```

Its facets are 12 quadrilaterals (not all identical):

```
sage: sum(1 for f in rd.facets() if len(f.vertices()) == 4)
12
```

```
>>> from sage.all import *
>>> sum(Integer(1) for f in rd.facets() if len(f.vertices()) == Integer(4))
12
```

Some more computations:

```
sage: p = rd.ehrhart_polynomial()     # optional - latte_int
sage: p                               # optional - latte_int
16*t^3 + 12*t^2 + 4*t + 1
sage: [p(i) for i in [1,2,3,4]]       # optional - latte_int
[33, 185, 553, 1233]
sage: [len((i*rd).integral_points()) for i in [1,2,3,4]]
[33, 185, 553, 1233]
```

```
>>> from sage.all import *
>>> p = rd.ehrhart_polynomial()     # optional - latte_int
>>> p                               # optional - latte_int
16*t^3 + 12*t^2 + 4*t + 1
>>> [p(i) for i in [Integer(1),Integer(2),Integer(3),Integer(4)]]       #␣
↪optional - latte_int
[33, 185, 553, 1233]
>>> [len((i*rd).integral_points()) for i in [Integer(1),Integer(2),Integer(3),
↪Integer(4)]]
[33, 185, 553, 1233]
```

rhombicosidodecahedron(*exact=True*, *base_ring=None*, *backend=None*)

Return the rhombicosidodecahedron.

The rhombicosidodecahedron is an Archimedean solid. It has 62 faces and 60 vertices. For more information, see the Wikipedia article Rhombicosidodecahedron.

INPUT:

- `exact` – boolean (default: `True`); if `False` use an approximate ring for the coordinates

- `base_ring` – the ring in which the coordinates will belong to. If it is not provided and `exact=True` it will be a the number field $\mathbf{Q}[\phi]$ where ϕ is the golden ratio and if `exact=False` it will be the real double field.
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: rid = polytopes.rhombicosidodecahedron()    # long time (6secs)
sage: rid.f_vector()                              # long time
(1, 60, 120, 62, 1)
sage: rid.base_ring()                             # long time
Number Field in sqrt5 with defining polynomial x^2 - 5
 with sqrt5 = 2.236067977499790?
```

```
>>> from sage.all import *
>>> rid = polytopes.rhombicosidodecahedron()    # long time (6secs)
>>> rid.f_vector()                              # long time
(1, 60, 120, 62, 1)
>>> rid.base_ring()                             # long time
Number Field in sqrt5 with defining polynomial x^2 - 5
 with sqrt5 = 2.236067977499790?
```

A much faster implementation using floating point approximations:

```
sage: rid = polytopes.rhombicosidodecahedron(exact=False)
sage: rid.f_vector()
(1, 60, 120, 62, 1)
sage: rid.base_ring()
Real Double Field
```

```
>>> from sage.all import *
>>> rid = polytopes.rhombicosidodecahedron(exact=False)
>>> rid.f_vector()
(1, 60, 120, 62, 1)
>>> rid.base_ring()
Real Double Field
```

Its facets are 20 triangles, 30 squares and 12 pentagons:

```
sage: sum(1 for f in rid.facets() if len(f.vertices()) == 3)
20
sage: sum(1 for f in rid.facets() if len(f.vertices()) == 4)
30
sage: sum(1 for f in rid.facets() if len(f.vertices()) == 5)
12
```

```
>>> from sage.all import *
>>> sum(Integer(1) for f in rid.facets() if len(f.vertices()) == Integer(3))
20
>>> sum(Integer(1) for f in rid.facets() if len(f.vertices()) == Integer(4))
30
>>> sum(Integer(1) for f in rid.facets() if len(f.vertices()) == Integer(5))
12
```

runcinated_one_hundred_twenty_cell(*exact=False*, *backend=None*)

Return the runcinated 120-cell.

The runcinated 120-cell is a 4-dimensional 4-uniform polytope in the H_4 family. It has 2400 vertices. For more information see Wikipedia article Runcinated 120-cell.

Warning

The coordinates are inexact by default. The computation with inexact coordinates (using the backend `'cdd'`) issues a UserWarning on inconsistencies.

INPUT:

- `exact` – boolean (default: `False`); if `True` use exact coordinates instead of floating point approximations
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: polytopes.runcinated_one_hundred_twenty_cell(exact=False)  # not tested␣
↪- very long time
doctest:warning ...  UserWarning: This polyhedron data is
numerically complicated; cdd could not convert between the inexact
V and H representation without loss of data. The resulting object
might show inconsistencies.
A 4-dimensional polyhedron in RDF^4 defined as the convex hull of 2400␣
↪vertices
```

```
>>> from sage.all import *
>>> polytopes.runcinated_one_hundred_twenty_cell(exact=False)  # not tested -␣
↪very long time
doctest:warning ...  UserWarning: This polyhedron data is
numerically complicated; cdd could not convert between the inexact
V and H representation without loss of data. The resulting object
might show inconsistencies.
A 4-dimensional polyhedron in RDF^4 defined as the convex hull of 2400␣
↪vertices
```

It is possible to use the backend `'normaliz'` to get an exact representation:

```
sage: polytopes.runcinated_one_hundred_twenty_cell(exact=True,   # not tested␣
↪- very long time
....:                                                   backend='normaliz')
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 2400 vertices
```

```
>>> from sage.all import *
>>> polytopes.runcinated_one_hundred_twenty_cell(exact=True,   # not tested -␣
↪very long time
...                                                   backend='normaliz')
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 2400 vertices
```

runcitruncated_one_hundred_twenty_cell(*exact=False*, *backend=None*)

Return the runcitruncated 120-cell.

The runcitruncated 120-cell is a 4-dimensional 4-uniform polytope in the H_4 family. It has 7200 vertices. For more information see Wikipedia article Runcitruncated 120-cell.

> **Warning**
>
> The coordinates are inexact by default. The computation with inexact coordinates (using the backend `'cdd'`) issues a UserWarning on inconsistencies.

INPUT:

- `exact` – boolean (default: `False`); if `True` use exact coordinates instead of floating point approximations
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: polytopes.runcitruncated_one_hundred_twenty_cell(exact=False)   # not␣
↪tested - very long time
doctest:warning
...
UserWarning: This polyhedron data is numerically complicated; cdd
could not convert between the inexact V and H representation
without loss of data. The resulting object might show
inconsistencies.
```

```
>>> from sage.all import *
>>> polytopes.runcitruncated_one_hundred_twenty_cell(exact=False)   # not␣
↪tested - very long time
doctest:warning
...
UserWarning: This polyhedron data is numerically complicated; cdd
could not convert between the inexact V and H representation
without loss of data. The resulting object might show
inconsistencies.
```

It is possible to use the backend `'normaliz'` to get an exact representation:

```
sage: polytopes.runcitruncated_one_hundred_twenty_cell(exact=True,    # not␣
↪tested - very long time
....:                                                       backend='normaliz')
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 7200 vertices
```

```
>>> from sage.all import *
>>> polytopes.runcitruncated_one_hundred_twenty_cell(exact=True,    # not␣
↪tested - very long time
...                                                       backend='normaliz')
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 7200 vertices
```

runcitruncated_six_hundred_cell(*exact=True*, *backend=None*)

Return the runcitruncated 600-cell.

The runcitruncated 600-cell is a 4-dimensional 4-uniform polytope in the H_4 family. It has 7200 vertices. For more information see Wikipedia article Runcitruncated 600-cell.

> **Warning**
>
> The coordinates are exact by default. The computation with inexact coordinates (using the backend `'cdd'`) returns a numerical inconsistency error, and thus cannot be computed.

INPUT:

- `exact` – boolean (default: `True`); if `True` use exact coordinates instead of floating point approximations
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: polytopes.runcitruncated_six_hundred_cell(backend='normaliz')   # not␣
↪tested - very long time
A 4-dimensional polyhedron in AA^4 defined as the convex hull of
7200 vertices
```

```
>>> from sage.all import *
>>> polytopes.runcitruncated_six_hundred_cell(backend='normaliz')   # not␣
↪tested - very long time
A 4-dimensional polyhedron in AA^4 defined as the convex hull of
7200 vertices
```

simplex(*dim=3*, *project=False*, *base_ring=None*, *backend=None*)

Return the `dim` dimensional simplex.

The d-simplex is the convex hull in $\mathbf{R}^{d+1}$ of the standard basis $(1, 0, \ldots, 0)$, $(0, 1, \ldots, 0)$, ldots, $(0, 0, \ldots, 1)$. For more information, see the Wikipedia article Simplex.

INPUT:

- `dim` – the dimension of the simplex, a positive integer
- `project` – boolean (default: `False`); if `True`, the polytope is (isometrically) projected to a vector space of dimension `dim-1`. This corresponds to the projection given by the matrix from *zero_sum_projection()*. By default, this operation turns the coordinates into floating point approximations (see `base_ring`).
- `base_ring` – the base ring to use to create the polytope; if `project` is `False`, this defaults to $\mathbf{Z}$. Otherwise, it defaults to `RDF`.
- `backend` – the backend to use to create the polytope

> **See also**
>
> *tetrahedron()*

EXAMPLES:

```
sage: s5 = polytopes.simplex(5)
sage: s5
A 5-dimensional polyhedron in ZZ^6 defined as the convex hull of 6 vertices
sage: s5.f_vector()
(1, 6, 15, 20, 15, 6, 1)
```

(continues on next page)

(continued from previous page)

```
sage: s5 = polytopes.simplex(5, project=True)
sage: s5
A 5-dimensional polyhedron in RDF^5 defined as the convex hull of 6 vertices
```

```
>>> from sage.all import *
>>> s5 = polytopes.simplex(Integer(5))
>>> s5
A 5-dimensional polyhedron in ZZ^6 defined as the convex hull of 6 vertices
>>> s5.f_vector()
(1, 6, 15, 20, 15, 6, 1)

>>> s5 = polytopes.simplex(Integer(5), project=True)
>>> s5
A 5-dimensional polyhedron in RDF^5 defined as the convex hull of 6 vertices
```

Its volume is $\sqrt{d+1}/d!$:

```
sage: s5 = polytopes.simplex(5, project=True)
sage: s5.volume()      # abs tol 1e-10
0.0204124145231931
sage: sqrt(6.) / factorial(5)
0.0204124145231931

sage: s6 = polytopes.simplex(6, project=True)
sage: s6.volume()      # abs tol 1e-10
0.00367465459870082
sage: sqrt(7.) / factorial(6)
0.00367465459870082
```

```
>>> from sage.all import *
>>> s5 = polytopes.simplex(Integer(5), project=True)
>>> s5.volume()      # abs tol 1e-10
0.0204124145231931
>>> sqrt(RealNumber('6.')) / factorial(Integer(5))
0.0204124145231931

>>> s6 = polytopes.simplex(Integer(6), project=True)
>>> s6.volume()      # abs tol 1e-10
0.00367465459870082
>>> sqrt(RealNumber('7.')) / factorial(Integer(6))
0.00367465459870082
```

Computation in algebraic reals:

```
sage: s3 = polytopes.simplex(3, project=True, base_ring=AA)                    #␣
↪needs sage.rings.number_field
sage: s3.volume() == sqrt(3+1) / factorial(3)                                   #␣
↪needs sage.rings.number_field
True
```

```
>>> from sage.all import *
>>> s3 = polytopes.simplex(Integer(3), project=True, base_ring=AA)              ␣
```

(continues on next page)

(continued from previous page)

```
↪     # needs sage.rings.number_field
>>> s3.volume() == sqrt(Integer(3)+Integer(1)) / factorial(Integer(3))            ␣
↪                     # needs sage.rings.number_field
True
```

six_hundred_cell(*exact=False*, *backend=None*)

Return the standard 600-cell polytope.

The 600-cell is a 4-dimensional regular polytope. In many ways this is an analogue of the icosahedron.

> **Warning**
>
> The coordinates are not exact by default. The computation with exact coordinates takes a huge amount of time.

INPUT:

- `exact` – boolean (default: `False`); if `True` use exact coordinates instead of floating point approximations
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: p600 = polytopes.six_hundred_cell(); p600                                   #␣
↪needs sage.groups
A 4-dimensional polyhedron in RDF^4 defined as the convex hull of 120 vertices
sage: p600.f_vector()                 # long time (~2sec)                         #␣
↪needs sage.groups
(1, 120, 720, 1200, 600, 1)
```

```
>>> from sage.all import *
>>> p600 = polytopes.six_hundred_cell(); p600                                   #␣
↪needs sage.groups
A 4-dimensional polyhedron in RDF^4 defined as the convex hull of 120 vertices
>>> p600.f_vector()                 # long time (~2sec)                         #␣
↪needs sage.groups
(1, 120, 720, 1200, 600, 1)
```

Computation with exact coordinates is currently too long to be useful:

```
sage: p600 = polytopes.six_hundred_cell(exact=True)          # long time, not␣
↪tested, needs sage.groups
sage: len(list(p600.bounded_edges()))                        # long time, not␣
↪tested, needs sage.groups
720
```

```
>>> from sage.all import *
>>> p600 = polytopes.six_hundred_cell(exact=True)          # long time, not␣
↪tested, needs sage.groups
>>> len(list(p600.bounded_edges()))                        # long time, not␣
↪tested, needs sage.groups
720
```

small_rhombicuboctahedron(*exact=True*, *base_ring=None*, *backend=None*)

Return the (small) rhombicuboctahedron.

The rhombicuboctahedron is an Archimedean solid with 24 vertices and 26 faces. See the Wikipedia article Rhombicuboctahedron for more information.

INPUT:

- `exact` – boolean (default: `True`); if `False` use an approximate ring for the coordinates
- `base_ring` – the ring in which the coordinates will belong to. If it is not provided and `exact=True` it will be a the number field $\mathbf{Q}[\phi]$ where ϕ is the golden ratio and if `exact=False` it will be the real double field.
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: sr = polytopes.small_rhombicuboctahedron()                          #␣
↪needs sage.rings.number_field
sage: sr.f_vector()                                                       #␣
↪needs sage.rings.number_field
(1, 24, 48, 26, 1)
sage: sr.volume()                                                         #␣
↪needs sage.rings.number_field
80/3*sqrt2 + 32
```

```
>>> from sage.all import *
>>> sr = polytopes.small_rhombicuboctahedron()                          #␣
↪needs sage.rings.number_field
>>> sr.f_vector()                                                       #␣
↪needs sage.rings.number_field
(1, 24, 48, 26, 1)
>>> sr.volume()                                                         #␣
↪needs sage.rings.number_field
80/3*sqrt2 + 32
```

The faces are 8 equilateral triangles and 18 squares:

```
sage: sum(1 for f in sr.facets() if len(f.vertices()) == 3)               #␣
↪needs sage.rings.number_field
8
sage: sum(1 for f in sr.facets() if len(f.vertices()) == 4)               #␣
↪needs sage.rings.number_field
18
```

```
>>> from sage.all import *
>>> sum(Integer(1) for f in sr.facets() if len(f.vertices()) == Integer(3))  ␣
↪                # needs sage.rings.number_field
8
>>> sum(Integer(1) for f in sr.facets() if len(f.vertices()) == Integer(4))  ␣
↪                # needs sage.rings.number_field
18
```

Its non exact version:

```
sage: sr = polytopes.small_rhombicuboctahedron(False)
sage: sr
A 3-dimensional polyhedron in RDF^3 defined as the convex hull of
24 vertices
sage: sr.f_vector()
(1, 24, 48, 26, 1)
```

```
>>> from sage.all import *
>>> sr = polytopes.small_rhombicuboctahedron(False)
>>> sr
A 3-dimensional polyhedron in RDF^3 defined as the convex hull of
24 vertices
>>> sr.f_vector()
(1, 24, 48, 26, 1)
```

snub_cube(*exact=False*, *base_ring=None*, *backend=None*, *verbose=False*)

Return a snub cube.

The snub cube is an Archimedean solid. It has 24 vertices and 38 faces. For more information see the Wikipedia article Snub_cube.

The constant z used in constructing this polytope is the reciprocal of the tribonacci constant, that is, the solution of the equation $x^3 + x^2 + x - 1 = 0$. See Wikipedia article Generalizations_of_Fibonacci_numbers#Tribonacci_numbers.

INPUT:

- `exact` – boolean (default: `False`); if `True` use exact coordinates instead of floating point approximations
- `base_ring` – the field to use; if `None` (the default), construct the exact number field needed (if `exact` is `True`) or default to `RDF` (if `exact` is `True`)
- `backend` – the backend to use to create the polytope; if `None` (the default), the backend will be selected automatically

EXAMPLES:

```
sage: # needs sage.groups
sage: sc_inexact = polytopes.snub_cube(exact=False); sc_inexact
A 3-dimensional polyhedron in RDF^3 defined as the convex hull of 24 vertices
sage: sc_inexact.f_vector()
(1, 24, 60, 38, 1)

sage: # long time, needs sage.groups sage.rings.number_field
sage: sc_exact = polytopes.snub_cube(exact=True)
sage: sc_exact.f_vector()
(1, 24, 60, 38, 1)
sage: sorted(sc_exact.vertices())
[A vertex at (-1, -z, -z^2),
 A vertex at (-1, -z^2, z),
 A vertex at (-1, z^2, -z),
 A vertex at (-1, z, z^2),
 A vertex at (-z, -1, z^2),
 A vertex at (-z, -z^2, -1),
 A vertex at (-z, z^2, 1),
```

(continues on next page)

(continued from previous page)

```
 A vertex at (-z, 1, -z^2),
 A vertex at (-z^2, -1, -z),
 A vertex at (-z^2, -z, 1),
 A vertex at (-z^2, z, -1),
 A vertex at (-z^2, 1, z),
 A vertex at (z^2, -1, z),
 A vertex at (z^2, -z, -1),
 A vertex at (z^2, z, 1),
 A vertex at (z^2, 1, -z),
 A vertex at (z, -1, -z^2),
 A vertex at (z, -z^2, 1),
 A vertex at (z, z^2, -1),
 A vertex at (z, 1, z^2),
 A vertex at (1, -z, z^2),
 A vertex at (1, -z^2, -z),
 A vertex at (1, z^2, z),
 A vertex at (1, z, -z^2)]
sage: sc_exact.is_combinatorially_isomorphic(sc_inexact)
True
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> sc_inexact = polytopes.snub_cube(exact=False); sc_inexact
A 3-dimensional polyhedron in RDF^3 defined as the convex hull of 24 vertices
>>> sc_inexact.f_vector()
(1, 24, 60, 38, 1)

>>> # long time, needs sage.groups sage.rings.number_field
>>> sc_exact = polytopes.snub_cube(exact=True)
>>> sc_exact.f_vector()
(1, 24, 60, 38, 1)
>>> sorted(sc_exact.vertices())
[A vertex at (-1, -z, -z^2),
 A vertex at (-1, -z^2, z),
 A vertex at (-1, z^2, -z),
 A vertex at (-1, z, z^2),
 A vertex at (-z, -1, z^2),
 A vertex at (-z, -z^2, -1),
 A vertex at (-z, z^2, 1),
 A vertex at (-z, 1, -z^2),
 A vertex at (-z^2, -1, -z),
 A vertex at (-z^2, -z, 1),
 A vertex at (-z^2, z, -1),
 A vertex at (-z^2, 1, z),
 A vertex at (z^2, -1, z),
 A vertex at (z^2, -z, -1),
 A vertex at (z^2, z, 1),
 A vertex at (z^2, 1, -z),
 A vertex at (z, -1, -z^2),
 A vertex at (z, -z^2, 1),
 A vertex at (z, z^2, -1),
 A vertex at (z, 1, z^2),
```

(continues on next page)

(continued from previous page)

```
 A vertex at (1, -z, z^2),
 A vertex at (1, -z^2, -z),
 A vertex at (1, z^2, z),
 A vertex at (1, z, -z^2)]
>>> sc_exact.is_combinatorially_isomorphic(sc_inexact)
True
```

snub_dodecahedron(*base_ring=None*, *backend=None*, *verbose=False*)

Return the snub dodecahedron.

The snub dodecahedron is an Archimedean solid. It has 92 faces and 60 vertices. For more information, see the Wikipedia article Snub_dodecahedron.

INPUT:

- `base_ring` – the ring in which the coordinates will belong to; if it is not provided it will be the real double field
- `backend` – the backend to use to create the polytope

EXAMPLES:

Only the backend using the optional normaliz package can construct the snub dodecahedron in reasonable time:

```
sage: sd = polytopes.snub_dodecahedron(base_ring=AA,        # optional -␣
↪pynormaliz, long time
....:                                       backend='normaliz')
sage: sd.f_vector()                                          # optional -␣
↪pynormaliz, long time
(1, 60, 150, 92, 1)
sage: sd.base_ring()                                         # optional -␣
↪pynormaliz, long time
Algebraic Real Field
```

```
>>> from sage.all import *
>>> sd = polytopes.snub_dodecahedron(base_ring=AA,        # optional -␣
↪pynormaliz, long time
...                                       backend='normaliz')
>>> sd.f_vector()                                          # optional -␣
↪pynormaliz, long time
(1, 60, 150, 92, 1)
>>> sd.base_ring()                                         # optional -␣
↪pynormaliz, long time
Algebraic Real Field
```

Its facets are 80 triangles and 12 pentagons:

```
sage: sum(1 for f in sd.facets()                            # optional -␣
↪pynormaliz, long time
....:       if len(f.vertices()) == 3)
80
sage: sum(1 for f in sd.facets()                            # optional -␣
↪pynormaliz, long time
```

(continues on next page)

(continued from previous page)

```
....:         if len(f.vertices()) == 5)
12
```

```
>>> from sage.all import *
>>> sum(Integer(1) for f in sd.facets()                             # optional␣
↪- pynormaliz, long time
...         if len(f.vertices()) == Integer(3))
80
>>> sum(Integer(1) for f in sd.facets()                             # optional␣
↪- pynormaliz, long time
...         if len(f.vertices()) == Integer(5))
12
```

static symmetric_edge_polytope(*backend=None*)

Return the symmetric edge polytope of `self`.

The symmetric edge polytope (SEP) of a Graph on n vertices is the polytope in $\mathbf{Z}^n$ defined as the convex hull of $e_i - e_j$ and $e_j - e_i$ for each edge (i, j). Here $e_1, \dots, e_n$ denotes the standard basis.

INPUT:

- `backend` – string or `None` (default); the backend to use; see *sage.geometry.polyhedron.constructor.Polyhedron()*

EXAMPLES:

The SEP of a 4-cycle is a cube:

```
sage: G = graphs.CycleGraph(4)
sage: P = G.symmetric_edge_polytope(); P                                    #␣
↪needs sage.geometry.polyhedron
A 3-dimensional polyhedron in ZZ^4 defined as the convex hull of 8 vertices
sage: P.is_combinatorially_isomorphic(polytopes.cube())                     #␣
↪needs sage.geometry.polyhedron
True
```

```
>>> from sage.all import *
>>> G = graphs.CycleGraph(Integer(4))
>>> P = G.symmetric_edge_polytope(); P                                    #␣
↪needs sage.geometry.polyhedron
A 3-dimensional polyhedron in ZZ^4 defined as the convex hull of 8 vertices
>>> P.is_combinatorially_isomorphic(polytopes.cube())                     #␣
↪needs sage.geometry.polyhedron
True
```

The SEP of a complete graph on 4 vertices is a cuboctahedron:

```
sage: G = graphs.CompleteGraph(4)
sage: P = G.symmetric_edge_polytope(); P                                    #␣
↪needs sage.geometry.polyhedron
A 3-dimensional polyhedron in ZZ^4 defined as the convex hull of 12 vertices
sage: P.is_combinatorially_isomorphic(polytopes.cuboctahedron())            #␣
↪needs sage.geometry.polyhedron
True
```

```
>>> from sage.all import *
>>> G = graphs.CompleteGraph(Integer(4))
>>> P = G.symmetric_edge_polytope(); P                                      #␣
↪needs sage.geometry.polyhedron
A 3-dimensional polyhedron in ZZ^4 defined as the convex hull of 12 vertices
>>> P.is_combinatorially_isomorphic(polytopes.cuboctahedron())             #␣
↪needs sage.geometry.polyhedron
True
```

The SEP of a graph with edges on n vertices has dimension n minus the number of connected components:

```
sage: n = randint(5, 12)
sage: G = Graph()
sage: while not G.num_edges():                                               #␣
↪needs networkx
....:     G = graphs.RandomGNP(n, 0.2)
sage: P = G.symmetric_edge_polytope()                                        #␣
↪needs networkx sage.geometry.polyhedron
sage: P.ambient_dim() == n                                                   #␣
↪needs networkx sage.geometry.polyhedron
True
sage: P.dim() == n - G.connected_components_number()                         #␣
↪needs networkx sage.geometry.polyhedron
True
```

```
>>> from sage.all import *
>>> n = randint(Integer(5), Integer(12))
>>> G = Graph()
>>> while not G.num_edges():                                                 #␣
↪needs networkx
...     G = graphs.RandomGNP(n, RealNumber('0.2'))
>>> P = G.symmetric_edge_polytope()                                          #␣
↪needs networkx sage.geometry.polyhedron
>>> P.ambient_dim() == n                                                     #␣
↪needs networkx sage.geometry.polyhedron
True
>>> P.dim() == n - G.connected_components_number()                           #␣
↪needs networkx sage.geometry.polyhedron
True
```

The SEP of a graph is isomorphic to the subdirect sum of its connected components SEP's:

```
sage: n = randint(3, 6)
sage: G1 = graphs.RandomGNP(n, 0.2)                                          #␣
↪needs networkx
sage: n = randint(3, 6)
sage: G2 = graphs.RandomGNP(n, 0.2)                                          #␣
↪needs networkx
sage: G = G1.disjoint_union(G2)                                              #␣
↪needs networkx
sage: P = G.symmetric_edge_polytope()                                        #␣
↪needs networkx sage.geometry.polyhedron
sage: P1 = G1.symmetric_edge_polytope()                                      #␣
```

(continues on next page)

(continued from previous page)

```
↪needs networkx sage.geometry.polyhedron
sage: P2 = G2.symmetric_edge_polytope()                                   #␣
↪needs networkx sage.geometry.polyhedron
sage: P.is_combinatorially_isomorphic(P1.subdirect_sum(P2))               #␣
↪needs networkx sage.geometry.polyhedron
True
```

```
>>> from sage.all import *
>>> n = randint(Integer(3), Integer(6))
>>> G1 = graphs.RandomGNP(n, RealNumber('0.2'))                            ␣
↪          # needs networkx
>>> n = randint(Integer(3), Integer(6))
>>> G2 = graphs.RandomGNP(n, RealNumber('0.2'))                            ␣
↪          # needs networkx
>>> G = G1.disjoint_union(G2)                                           #␣
↪needs networkx
>>> P = G.symmetric_edge_polytope()                                     #␣
↪needs networkx sage.geometry.polyhedron
>>> P1 = G1.symmetric_edge_polytope()                                   #␣
↪needs networkx sage.geometry.polyhedron
>>> P2 = G2.symmetric_edge_polytope()                                   #␣
↪needs networkx sage.geometry.polyhedron
>>> P.is_combinatorially_isomorphic(P1.subdirect_sum(P2))               #␣
↪needs networkx sage.geometry.polyhedron
True
```

All trees on n vertices have isomorphic SEPs:

```
sage: n = randint(4, 10)
sage: G1 = graphs.RandomTree(n)
sage: G2 = graphs.RandomTree(n)
sage: P1 = G1.symmetric_edge_polytope()                                   #␣
↪needs sage.geometry.polyhedron
sage: P2 = G2.symmetric_edge_polytope()                                   #␣
↪needs sage.geometry.polyhedron
sage: P1.is_combinatorially_isomorphic(P2)                                #␣
↪needs sage.geometry.polyhedron
True
```

```
>>> from sage.all import *
>>> n = randint(Integer(4), Integer(10))
>>> G1 = graphs.RandomTree(n)
>>> G2 = graphs.RandomTree(n)
>>> P1 = G1.symmetric_edge_polytope()                                   #␣
↪needs sage.geometry.polyhedron
>>> P2 = G2.symmetric_edge_polytope()                                   #␣
↪needs sage.geometry.polyhedron
>>> P1.is_combinatorially_isomorphic(P2)                                #␣
↪needs sage.geometry.polyhedron
True
```

However, there are still many different SEPs:

```
sage: len(list(graphs(5)))
34
sage: polys = []
sage: for G in graphs(5):                                                    #␣
↪needs sage.geometry.polyhedron
....:     P = G.symmetric_edge_polytope()
....:     for P1 in polys:
....:         if P.is_combinatorially_isomorphic(P1):
....:             break
....:     else:
....:         polys.append(P)
sage: len(polys)                                                             #␣
↪needs sage.geometry.polyhedron
25
```

```
>>> from sage.all import *
>>> len(list(graphs(Integer(5))))
34
>>> polys = []
>>> for G in graphs(Integer(5)):                                             ␣
↪     # needs sage.geometry.polyhedron
...     P = G.symmetric_edge_polytope()
...     for P1 in polys:
...         if P.is_combinatorially_isomorphic(P1):
...             break
...     else:
...         polys.append(P)
>>> len(polys)                                                             #␣
↪needs sage.geometry.polyhedron
25
```

A non-trivial example of two graphs with isomorphic SEPs:

```
sage: G1 = graphs.CycleGraph(4)
sage: G1.add_edges([[0, 5], [5, 2], [1, 6], [6, 2]])
sage: G2 = copy(G1)
sage: G1.add_edges([[2, 7], [7, 3]])
sage: G2.add_edges([[0, 7], [7, 3]])
sage: G1.is_isomorphic(G2)
False
sage: P1 = G1.symmetric_edge_polytope()                                      #␣
↪needs sage.geometry.polyhedron
sage: P2 = G2.symmetric_edge_polytope()                                      #␣
↪needs sage.geometry.polyhedron
sage: P1.is_combinatorially_isomorphic(P2)                                   #␣
↪needs sage.geometry.polyhedron
True
```

```
>>> from sage.all import *
>>> G1 = graphs.CycleGraph(Integer(4))
>>> G1.add_edges([[Integer(0), Integer(5)], [Integer(5), Integer(2)],␣
↪[Integer(1), Integer(6)], [Integer(6), Integer(2)]])
```

(continues on next page)

(continued from previous page)

```
>>> G2 = copy(G1)
>>> G1.add_edges([[Integer(2), Integer(7)], [Integer(7), Integer(3)]])
>>> G2.add_edges([[Integer(0), Integer(7)], [Integer(7), Integer(3)]])
>>> G1.is_isomorphic(G2)
False
>>> P1 = G1.symmetric_edge_polytope()                                    #␣
↪needs sage.geometry.polyhedron
>>> P2 = G2.symmetric_edge_polytope()                                    #␣
↪needs sage.geometry.polyhedron
>>> P1.is_combinatorially_isomorphic(P2)                                 #␣
↪needs sage.geometry.polyhedron
True
```

Apparently, glueing two graphs together on a vertex gives isomorphic SEPs:

```
sage: n = randint(3, 7)
sage: g1 = graphs.RandomGNP(n, 0.2)                                        #␣
↪needs networkx
sage: g2 = graphs.RandomGNP(n, 0.2)                                        #␣
↪needs networkx
sage: G = g1.disjoint_union(g2)                                            #␣
↪needs networkx
sage: H = copy(G)                                                          #␣
↪needs networkx
sage: G.merge_vertices(((0, randrange(n)), (1, randrange(n))))             #␣
↪needs networkx
sage: H.merge_vertices(((0, randrange(n)), (1, randrange(n))))             #␣
↪needs networkx
sage: PG = G.symmetric_edge_polytope()                                     #␣
↪needs networkx sage.geometry.polyhedron
sage: PH = H.symmetric_edge_polytope()                                     #␣
↪needs networkx sage.geometry.polyhedron
sage: PG.is_combinatorially_isomorphic(PH)                                 #␣
↪needs networkx sage.geometry.polyhedron
True
```

```
>>> from sage.all import *
>>> n = randint(Integer(3), Integer(7))
>>> g1 = graphs.RandomGNP(n, RealNumber('0.2'))                              ␣
↪           # needs networkx
>>> g2 = graphs.RandomGNP(n, RealNumber('0.2'))                              ␣
↪           # needs networkx
>>> G = g1.disjoint_union(g2)                                            #␣
↪needs networkx
>>> H = copy(G)                                                          #␣
↪needs networkx
>>> G.merge_vertices(((Integer(0), randrange(n)), (Integer(1),␣
↪randrange(n))))                # needs networkx
>>> H.merge_vertices(((Integer(0), randrange(n)), (Integer(1),␣
↪randrange(n))))                # needs networkx
>>> PG = G.symmetric_edge_polytope()                                     #␣
↪needs networkx sage.geometry.polyhedron
```

(continues on next page)

(continued from previous page)

```
>>> PH = H.symmetric_edge_polytope()                                        #␣
↪needs networkx sage.geometry.polyhedron
>>> PG.is_combinatorially_isomorphic(PH)                                    #␣
↪needs networkx sage.geometry.polyhedron
True
```

tetrahedron(*backend=None*)

Return the tetrahedron.

The tetrahedron is a Platonic solid with 4 vertices and 4 faces dual to itself. It can be defined as the convex hull of the 4 vertices $(0, 0, 0)$, $(1, 1, 0)$, $(1, 0, 1)$ and $(0, 1, 1)$. For more information, see the Wikipedia article Tetrahedron.

INPUT:

- `backend` – the backend to use to create the polytope

See also

simplex()

EXAMPLES:

```
sage: co = polytopes.tetrahedron()
sage: co.f_vector()
(1, 4, 6, 4, 1)
```

```
>>> from sage.all import *
>>> co = polytopes.tetrahedron()
>>> co.f_vector()
(1, 4, 6, 4, 1)
```

Its facets are 4 triangles:

```
sage: sum(1 for f in co.facets() if len(f.vertices()) == 3)
4
```

```
>>> from sage.all import *
>>> sum(Integer(1) for f in co.facets() if len(f.vertices()) == Integer(3))
4
```

Some more computation:

```
sage: co.volume()
1/3
sage: co.ehrhart_polynomial()      # optional - latte_int
1/3*t^3 + t^2 + 5/3*t + 1
```

```
>>> from sage.all import *
>>> co.volume()
1/3
>>> co.ehrhart_polynomial()      # optional - latte_int
1/3*t^3 + t^2 + 5/3*t + 1
```

truncated_cube(*exact=True*, *base_ring=None*, *backend=None*)

Return the truncated cube.

The truncated cube is an Archimedean solid with 24 vertices and 14 faces. It can be defined as the convex hull of the 24 vertices $(\pm x, \pm 1, \pm 1), (\pm 1, \pm x, \pm 1), (\pm 1, \pm 1, \pm x)$ where $x = \sqrt(2) - 1$. For more information, see the Wikipedia article Truncated_cube.

INPUT:

- `exact` – boolean (default: `True`); if `False` use an approximate ring for the coordinates
- `base_ring` – the ring in which the coordinates will belong to. If it is not provided and `exact=True` it will be a the number field $\mathbf{Q}[\sqrt{2}]$ and if `exact=False` it will be the real double field.
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: co = polytopes.truncated_cube()                                    #␣
↪needs sage.rings.number_field
sage: co.f_vector()                                                      #␣
↪needs sage.rings.number_field
(1, 24, 36, 14, 1)
```

```
>>> from sage.all import *
>>> co = polytopes.truncated_cube()                                    #␣
↪needs sage.rings.number_field
>>> co.f_vector()                                                      #␣
↪needs sage.rings.number_field
(1, 24, 36, 14, 1)
```

Its facets are 8 triangles and 6 octogons:

```
sage: sum(1 for f in co.facets() if len(f.vertices()) == 3)             #␣
↪needs sage.rings.number_field
8
sage: sum(1 for f in co.facets() if len(f.vertices()) == 8)             #␣
↪needs sage.rings.number_field
6
```

```
>>> from sage.all import *
>>> sum(Integer(1) for f in co.facets() if len(f.vertices()) == Integer(3))  ␣
↪                # needs sage.rings.number_field
8
>>> sum(Integer(1) for f in co.facets() if len(f.vertices()) == Integer(8))  ␣
↪                # needs sage.rings.number_field
6
```

Some more computation:

```
sage: co.volume()                                                        #␣
↪needs sage.rings.number_field
56/3*sqrt2 - 56/3
```

```
>>> from sage.all import *
>>> co.volume()                                                        #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.rings.number_field
56/3*sqrt2 - 56/3
```

truncated_dodecahedron(*exact=True*, *base_ring=None*, *backend=None*)

Return the truncated dodecahedron.

The truncated dodecahedron is an Archimedean solid. It has 32 faces and 60 vertices. For more information, see the Wikipedia article Truncated dodecahedron.

INPUT:

- `exact` – boolean (default: `True`); if `False` use an approximate ring for the coordinates
- `base_ring` – the ring in which the coordinates will belong to. If it is not provided and `exact=True` it will be a the number field $\mathbf{Q}[\phi]$ where ϕ is the golden ratio and if `exact=False` it will be the real double field.
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: td = polytopes.truncated_dodecahedron()                          #␣
↪needs sage.rings.number_field
sage: td.f_vector()                                                     #␣
↪needs sage.rings.number_field
(1, 60, 90, 32, 1)
sage: td.base_ring()                                                    #␣
↪needs sage.rings.number_field
Number Field in sqrt5 with defining polynomial x^2 - 5
 with sqrt5 = 2.236067977499790?
```

```
>>> from sage.all import *
>>> td = polytopes.truncated_dodecahedron()                          #␣
↪needs sage.rings.number_field
>>> td.f_vector()                                                     #␣
↪needs sage.rings.number_field
(1, 60, 90, 32, 1)
>>> td.base_ring()                                                    #␣
↪needs sage.rings.number_field
Number Field in sqrt5 with defining polynomial x^2 - 5
 with sqrt5 = 2.236067977499790?
```

Its facets are 20 triangles and 12 regular decagons:

```
sage: sum(1 for f in td.facets() if len(f.vertices()) == 3)             #␣
↪needs sage.rings.number_field
20
sage: sum(1 for f in td.facets() if len(f.vertices()) == 10)            #␣
↪needs sage.rings.number_field
12
```

```
>>> from sage.all import *
>>> sum(Integer(1) for f in td.facets() if len(f.vertices()) == Integer(3))  ␣
↪            # needs sage.rings.number_field
20
```

(continues on next page)

(continued from previous page)

```
>>> sum(Integer(1) for f in td.facets() if len(f.vertices()) == Integer(10)) ␣
↪             # needs sage.rings.number_field
12
```

The faster implementation using floating point approximations does not fully work unfortunately, see https://github.com/cddlib/cddlib/pull/7 for a detailed discussion of this case:

```
sage: td = polytopes.truncated_dodecahedron(exact=False)  # random
doctest:warning
...
UserWarning: This polyhedron data is numerically complicated; cdd
could not convert between the inexact V and H representation
without loss of data. The resulting object might show
inconsistencies.
sage: td.f_vector()
Traceback (most recent call last):
...
ValueError: not all vertices are intersections of facets
sage: td.base_ring()
Real Double Field
```

```
>>> from sage.all import *
>>> td = polytopes.truncated_dodecahedron(exact=False)  # random
doctest:warning
...
UserWarning: This polyhedron data is numerically complicated; cdd
could not convert between the inexact V and H representation
without loss of data. The resulting object might show
inconsistencies.
>>> td.f_vector()
Traceback (most recent call last):
...
ValueError: not all vertices are intersections of facets
>>> td.base_ring()
Real Double Field
```

truncated_icosidodecahedron(*exact=True*, *base_ring=None*, *backend=None*)

Return the truncated icosidodecahedron.

The truncated icosidodecahedron is an Archimedean solid. It has 62 faces and 120 vertices. For more information, see the Wikipedia article Truncated_icosidodecahedron.

INPUT:

- `exact` – boolean (default: `True`); if `False` use an approximate ring for the coordinates
- `base_ring` – the ring in which the coordinates will belong to. If it is not provided and `exact=True` it will be a the number field $\mathbf{Q}[\phi]$ where ϕ is the golden ratio and if `exact=False` it will be the real double field.
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: ti = polytopes.truncated_icosidodecahedron()  # long time
sage: ti.f_vector()                                  # long time
(1, 120, 180, 62, 1)
sage: ti.base_ring()                                 # long time
Number Field in sqrt5 with defining polynomial x^2 - 5
 with sqrt5 = 2.236067977499790?
```

```
>>> from sage.all import *
>>> ti = polytopes.truncated_icosidodecahedron()  # long time
>>> ti.f_vector()                                  # long time
(1, 120, 180, 62, 1)
>>> ti.base_ring()                                 # long time
Number Field in sqrt5 with defining polynomial x^2 - 5
 with sqrt5 = 2.236067977499790?
```

The implementation using floating point approximations is much faster:

```
sage: ti = polytopes.truncated_icosidodecahedron(exact=False) # random
sage: ti.f_vector()
(1, 120, 180, 62, 1)
sage: ti.base_ring()
Real Double Field
```

```
>>> from sage.all import *
>>> ti = polytopes.truncated_icosidodecahedron(exact=False) # random
>>> ti.f_vector()
(1, 120, 180, 62, 1)
>>> ti.base_ring()
Real Double Field
```

Its facets are 30 squares, 20 hexagons and 12 decagons:

```
sage: sum(1 for f in ti.facets() if len(f.vertices()) == 4)
30
sage: sum(1 for f in ti.facets() if len(f.vertices()) == 6)
20
sage: sum(1 for f in ti.facets() if len(f.vertices()) == 10)
12
```

```
>>> from sage.all import *
>>> sum(Integer(1) for f in ti.facets() if len(f.vertices()) == Integer(4))
30
>>> sum(Integer(1) for f in ti.facets() if len(f.vertices()) == Integer(6))
20
>>> sum(Integer(1) for f in ti.facets() if len(f.vertices()) == Integer(10))
12
```

truncated_octahedron(*backend=None*)

Return the truncated octahedron.

The truncated octahedron is an Archimedean solid with 24 vertices and 14 faces. It can be defined as the convex hull off all the permutations of $(0, \pm 1, \pm 2)$. For more information, see the Wikipedia article Truncated_octahedron.

This is also known as the permutohedron of dimension 3.

INPUT:

- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: co = polytopes.truncated_octahedron()
sage: co.f_vector()
(1, 24, 36, 14, 1)
```

```
>>> from sage.all import *
>>> co = polytopes.truncated_octahedron()
>>> co.f_vector()
(1, 24, 36, 14, 1)
```

Its facets are 6 squares and 8 hexagons:

```
sage: sum(1 for f in co.facets() if len(f.vertices()) == 4)
6
sage: sum(1 for f in co.facets() if len(f.vertices()) == 6)
8
```

```
>>> from sage.all import *
>>> sum(Integer(1) for f in co.facets() if len(f.vertices()) == Integer(4))
6
>>> sum(Integer(1) for f in co.facets() if len(f.vertices()) == Integer(6))
8
```

Some more computation:

```
sage: co.volume()
32
sage: co.ehrhart_polynomial()       # optional - latte_int                    #␣
↪needs sage.combinat
32*t^3 + 18*t^2 + 6*t + 1
```

```
>>> from sage.all import *
>>> co.volume()
32
>>> co.ehrhart_polynomial()       # optional - latte_int                    #␣
↪needs sage.combinat
32*t^3 + 18*t^2 + 6*t + 1
```

truncated_one_hundred_twenty_cell(*exact=True*, *backend=None*)

Return the truncated 120-cell.

The truncated 120-cell is a 4-dimensional 4-uniform polytope in the H_4 family. It has 2400 vertices. For more information see Wikipedia article Truncated 120-cell.

> **Warning**
>
> The coordinates are exact by default. The computation with inexact coordinates (using the backend `'cdd'`) returns a numerical inconsistency error, and thus cannot be computed.

INPUT:

- `exact` – boolean (default: `True`); if `True` use exact coordinates instead of floating point approximations
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: polytopes.truncated_one_hundred_twenty_cell(backend='normaliz')  # not␣
↪tested - long time
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 2400 vertices
```

```
>>> from sage.all import *
>>> polytopes.truncated_one_hundred_twenty_cell(backend='normaliz')  # not␣
↪tested - long time
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 2400 vertices
```

truncated_six_hundred_cell(*exact=False*, *backend=None*)

Return the truncated 600-cell.

The truncated 600-cell is a 4-dimensional 4-uniform polytope in the H_4 family. It has 1440 vertices. For more information see Wikipedia article Truncated 600-cell.

> **Warning**
>
> The coordinates are not exact by default. The computation with exact coordinates takes a huge amount of time.

INPUT:

- `exact` – boolean (default: `False`); if `True` use exact coordinates instead of floating point approximations
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: polytopes.truncated_six_hundred_cell()  # not tested - long time
A 4-dimensional polyhedron in RDF^4 defined as the convex hull of 1440␣
↪vertices
```

```
>>> from sage.all import *
>>> polytopes.truncated_six_hundred_cell()  # not tested - long time
A 4-dimensional polyhedron in RDF^4 defined as the convex hull of 1440␣
↪vertices
```

It is possible to use the backend `'normaliz'` to get an exact representation:

```
sage: polytopes.truncated_six_hundred_cell(exact=True,backend='normaliz')  #␣
↪not tested, long time (16s)
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 1440 vertices
```

```
>>> from sage.all import *
>>> polytopes.truncated_six_hundred_cell(exact=True,backend='normaliz')  #␣
↪not tested, long time (16s)
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 1440 vertices
```

truncated_tetrahedron(*backend=None*)

Return the truncated tetrahedron.

The truncated tetrahedron is an Archimedean solid with 12 vertices and 8 faces. It can be defined as the convex hull off all the permutations of $(\pm 1, \pm 1, \pm 3)$ with an even number of minus signs. For more information, see the Wikipedia article Truncated_tetrahedron.

INPUT:

- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: co = polytopes.truncated_tetrahedron()
sage: co.f_vector()
(1, 12, 18, 8, 1)
```

```
>>> from sage.all import *
>>> co = polytopes.truncated_tetrahedron()
>>> co.f_vector()
(1, 12, 18, 8, 1)
```

Its facets are 4 triangles and 4 hexagons:

```
sage: sum(1 for f in co.facets() if len(f.vertices()) == 3)
4
sage: sum(1 for f in co.facets() if len(f.vertices()) == 6)
4
```

```
>>> from sage.all import *
>>> sum(Integer(1) for f in co.facets() if len(f.vertices()) == Integer(3))
4
>>> sum(Integer(1) for f in co.facets() if len(f.vertices()) == Integer(6))
4
```

Some more computation:

```
sage: co.volume()
184/3
sage: co.ehrhart_polynomial()      # optional - latte_int
184/3*t^3 + 28*t^2 + 26/3*t + 1
```

```
>>> from sage.all import *
>>> co.volume()
184/3
>>> co.ehrhart_polynomial()      # optional - latte_int
184/3*t^3 + 28*t^2 + 26/3*t + 1
```

twenty_four_cell(*backend=None*)

Return the standard 24-cell polytope.

The 24-cell polyhedron (also called icositetrachoron or octaplex) is a regular polyhedron in 4-dimension. For more information see the Wikipedia article 24-cell.

INPUT:

- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: p24 = polytopes.twenty_four_cell()
sage: p24.f_vector()
(1, 24, 96, 96, 24, 1)
sage: v = next(p24.vertex_generator())
sage: for adj in v.neighbors(): print(adj)
A vertex at (-1/2, -1/2, -1/2, 1/2)
A vertex at (-1/2, -1/2, 1/2, -1/2)
A vertex at (-1, 0, 0, 0)
A vertex at (-1/2, 1/2, -1/2, -1/2)
A vertex at (0, -1, 0, 0)
A vertex at (0, 0, -1, 0)
A vertex at (0, 0, 0, -1)
A vertex at (1/2, -1/2, -1/2, -1/2)

sage: p24.volume()
2
```

```
>>> from sage.all import *
>>> p24 = polytopes.twenty_four_cell()
>>> p24.f_vector()
(1, 24, 96, 96, 24, 1)
>>> v = next(p24.vertex_generator())
>>> for adj in v.neighbors(): print(adj)
A vertex at (-1/2, -1/2, -1/2, 1/2)
A vertex at (-1/2, -1/2, 1/2, -1/2)
A vertex at (-1, 0, 0, 0)
A vertex at (-1/2, 1/2, -1/2, -1/2)
A vertex at (0, -1, 0, 0)
A vertex at (0, 0, -1, 0)
A vertex at (0, 0, 0, -1)
A vertex at (1/2, -1/2, -1/2, -1/2)

>>> p24.volume()
2
```

zonotope(*generators*, *backend=None*)

Return the zonotope, or parallelotope, spanned by the generators.

The parallelotope is the multi-dimensional generalization of a parallelogram (2 generators) and a parallelepiped (3 generators).

INPUT:

- `generators` – list of vectors of same dimension
- `backend` – the backend to use to create the polytope

EXAMPLES:

```
sage: polytopes.parallelotope([ (1,0), (0,1) ])
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices
sage: polytopes.parallelotope([[1,2,3,4], [0,1,0,7], [3,1,0,2], [0,0,1,0]])
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 16 vertices
```

(continues on next page)

(continued from previous page)

```
sage: K = QuadraticField(2, 'sqrt2')                                    #␣
↪needs sage.rings.number_field
sage: sqrt2 = K.gen()                                                   #␣
↪needs sage.rings.number_field
sage: P = polytopes.parallelotope([(1, sqrt2), (1, -1)]); P             #␣
↪needs sage.rings.number_field
A 2-dimensional polyhedron in (Number Field in sqrt2 with defining
polynomial x^2 - 2 with sqrt2 = 1.414213562373095?)^2 defined as
the convex hull of 4 vertices
```

```
>>> from sage.all import *
>>> polytopes.parallelotope([ (Integer(1),Integer(0)), (Integer(0),
↪Integer(1)) ])
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices
>>> polytopes.parallelotope([[Integer(1),Integer(2),Integer(3),Integer(4)],␣
↪[Integer(0),Integer(1),Integer(0),Integer(7)], [Integer(3),Integer(1),
↪Integer(0),Integer(2)], [Integer(0),Integer(0),Integer(1),Integer(0)]])
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 16 vertices

>>> K = QuadraticField(Integer(2), 'sqrt2')                             ␣
↪     # needs sage.rings.number_field
>>> sqrt2 = K.gen()                                                   #␣
↪needs sage.rings.number_field
>>> P = polytopes.parallelotope([(Integer(1), sqrt2), (Integer(1), -
↪Integer(1))]); P                # needs sage.rings.number_field
A 2-dimensional polyhedron in (Number Field in sqrt2 with defining
polynomial x^2 - 2 with sqrt2 = 1.414213562373095?)^2 defined as
the convex hull of 4 vertices
```

`sage.geometry.polyhedron.library.`**`gale_transform_to_polytope`**`(`*vectors*, *base_ring=None*, *backend=None*`)`

Return the polytope associated to the list of vectors forming a Gale transform.

This function is the inverse of `gale_transform()` up to projective transformation.

INPUT:

- `vectors` – the vectors of the Gale transform
- `base_ring` – string (default: `None`); the base ring to be used for the construction
- `backend` – string (default: `None`); the backend to use to create the polytope

Note

The order of the input vectors will not be preserved.

If the center of the (input) vectors is the origin, the function is much faster and might give a nicer representation of the polytope.

If this is not the case, the vectors will be scaled (each by a positive scalar) accordingly to obtain the polytope.

See also

:func`~sage.geometry.polyhedron.library.gale_transform_to_primal`.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.library import gale_transform_to_polytope
sage: points = polytopes.octahedron().gale_transform()
sage: points
((0, -1), (-1, 0), (1, 1), (1, 1), (-1, 0), (0, -1))
sage: P = gale_transform_to_polytope(points); P
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 6 vertices
sage: P.vertices()
(A vertex at (-1, 0, 0),
 A vertex at (0, -1, 0),
 A vertex at (0, 0, -1),
 A vertex at (0, 0, 1),
 A vertex at (0, 1, 0),
 A vertex at (1, 0, 0))
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.library import gale_transform_to_polytope
>>> points = polytopes.octahedron().gale_transform()
>>> points
((0, -1), (-1, 0), (1, 1), (1, 1), (-1, 0), (0, -1))
>>> P = gale_transform_to_polytope(points); P
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 6 vertices
>>> P.vertices()
(A vertex at (-1, 0, 0),
 A vertex at (0, -1, 0),
 A vertex at (0, 0, -1),
 A vertex at (0, 0, 1),
 A vertex at (0, 1, 0),
 A vertex at (1, 0, 0))
```

One can specify the base ring:

```
sage: gale_transform_to_polytope(
....:     [(1,1), (-1,-1), (1,0),
....:      (-1,0), (1,-1), (-2,1)]).vertices()
(A vertex at (-25, 0, 0),
 A vertex at (-15, 50, -60),
 A vertex at (0, -25, 0),
 A vertex at (0, 0, -25),
 A vertex at (16, -35, 54),
 A vertex at (24, 10, 31))
sage: gale_transform_to_polytope(
....:     [(1,1), (-1,-1), (1,0),
....:      (-1,0), (1,-1), (-2,1)],
....:     base_ring=RDF).vertices()
(A vertex at (-0.64, 1.4, -2.16),
 A vertex at (-0.96, -0.4, -1.24),
```

(continues on next page)

(continued from previous page)

```
A vertex at (0.6, -2.0, 2.4),
 A vertex at (1.0, 0.0, 0.0),
 A vertex at (0.0, 1.0, 0.0),
 A vertex at (0.0, 0.0, 1.0))
```

```
>>> from sage.all import *
>>> gale_transform_to_polytope(
...     [(Integer(1),Integer(1)), (-Integer(1),-Integer(1)), (Integer(1),
↪Integer(0)),
...      (-Integer(1),Integer(0)), (Integer(1),-Integer(1)), (-Integer(2),
↪Integer(1))]).vertices()
(A vertex at (-25, 0, 0),
 A vertex at (-15, 50, -60),
 A vertex at (0, -25, 0),
 A vertex at (0, 0, -25),
 A vertex at (16, -35, 54),
 A vertex at (24, 10, 31))
>>> gale_transform_to_polytope(
...     [(Integer(1),Integer(1)), (-Integer(1),-Integer(1)), (Integer(1),
↪Integer(0)),
...      (-Integer(1),Integer(0)), (Integer(1),-Integer(1)), (-Integer(2),
↪Integer(1))],
...     base_ring=RDF).vertices()
(A vertex at (-0.64, 1.4, -2.16),
 A vertex at (-0.96, -0.4, -1.24),
 A vertex at (0.6, -2.0, 2.4),
 A vertex at (1.0, 0.0, 0.0),
 A vertex at (0.0, 1.0, 0.0),
 A vertex at (0.0, 0.0, 1.0))
```

One can also specify the backend:

```
sage: gale_transform_to_polytope(
....:     [(1,1), (-1,-1), (1,0),
....:      (-1,0), (1,-1), (-2,1)],
....:     backend='field').backend()
'field'
sage: gale_transform_to_polytope(
....:     [(1,1), (-1,-1), (1,0),
....:      (-1,0), (1,-1), (-2,1)],
....:     backend='cdd', base_ring=RDF).backend()
'cdd'
```

```
>>> from sage.all import *
>>> gale_transform_to_polytope(
...     [(Integer(1),Integer(1)), (-Integer(1),-Integer(1)), (Integer(1),
↪Integer(0)),
...      (-Integer(1),Integer(0)), (Integer(1),-Integer(1)), (-Integer(2),
↪Integer(1))],
...     backend='field').backend()
'field'
>>> gale_transform_to_polytope(
```

(continues on next page)

(continued from previous page)

```
...     [(Integer(1),Integer(1)), (-Integer(1),-Integer(1)), (Integer(1),
↪Integer(0)),
...      (-Integer(1),Integer(0)), (Integer(1),-Integer(1)), (-Integer(2),
↪Integer(1))],
...     backend='cdd', base_ring=RDF).backend()
'cdd'
```

A gale transform corresponds to a polytope if and only if every oriented (linear) hyperplane has at least two vectors on each side. See Theorem 6.19 of [Zie2007]. If this is not the case, one of two errors is raised.

If there is such a hyperplane with no vector on one side, the vectors are not totally cyclic:

```
sage: gale_transform_to_polytope([(0,1), (1,1), (1,0), (-1,0)])
Traceback (most recent call last):
...
ValueError: input vectors not totally cyclic
```

```
>>> from sage.all import *
>>> gale_transform_to_polytope([(Integer(0),Integer(1)), (Integer(1),Integer(1)),␣
↪(Integer(1),Integer(0)), (-Integer(1),Integer(0))])
Traceback (most recent call last):
...
ValueError: input vectors not totally cyclic
```

If every hyperplane has at least one vector on each side, then the gale transform corresponds to a point configuration. It corresponds to a polytope if and only if this point configuration is convex and if and only if every hyperplane contains at least two vectors of the gale transform on each side.

If this is not the case, an error is raised:

```
sage: gale_transform_to_polytope([(0,1), (1,1), (1,0), (-1,-1)])
Traceback (most recent call last):
...
ValueError: the gale transform does not correspond to a polytope
```

```
>>> from sage.all import *
>>> gale_transform_to_polytope([(Integer(0),Integer(1)), (Integer(1),Integer(1)),␣
↪(Integer(1),Integer(0)), (-Integer(1),-Integer(1))])
Traceback (most recent call last):
...
ValueError: the gale transform does not correspond to a polytope
```

`sage.geometry.polyhedron.library.`**`gale_transform_to_primal`**(*vectors*, *base_ring=None*, *backend=None*)

Return a point configuration dual to a totally cyclic vector configuration.

This is the dehomogenized vector configuration dual to the input. The dual vector configuration is acyclic and can therefore be dehomogenized as the input is totally cyclic.

INPUT:

- `vectors` – the ordered vectors of the Gale transform
- `base_ring` – string (default: `None`); the base ring to be used for the construction

- `backend` – string (default: `None`); the backend to be use to construct a polyhedral, used internally in case the center is not the origin, see `Polyhedron()`

OUTPUT: an ordered point configuration as list of vectors

Note: If the center of the (input) vectors is the origin, the function is much faster and might give a nicer representation of the point configuration.

If this is not the case, the vectors will be scaled (each by a positive scalar) accordingly.

ALGORITHM:

Step 1: If the center of the (input) vectors is not the origin, we do an appropriate transformation to make it so.

Step 2: We add a row of ones on top of `Matrix(vectors)`. The right kernel of this larger matrix is the dual configuration space, and a basis of this space provides the dual point configuration.

More concretely, the dual vector configuration (inhomogeneous) is obtained by taking a basis of the right kernel of `Matrix(vectors)`. If the center of the (input) vectors is the origin, there exists a basis of the right kernel of the form `[[1], [V]]`, where `[1]` represents a row of ones. Then, `V` is a dehomogenization and thus the dual point configuration.

To extend `[1]` to a basis of `Matrix(vectors)`, we add a row of ones to `Matrix(vectors)` and calculate a basis of the right kernel of the obtained matrix.

REFERENCES:

For more information, see Section 6.4 of [Zie2007] or Definition 2.5.1 and Definition 4.1.35 of [DLRS2010].

See also: :func`~sage.geometry.polyhedron.library.gale_transform_to_polytope`.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.library import gale_transform_to_primal
sage: points = ((0, -1), (-1, 0), (1, 1), (1, 1), (-1, 0), (0, -1))
sage: gale_transform_to_primal(points)
[(0, 0, 1), (0, 1, 0), (1, 0, 0), (-1, 0, 0), (0, -1, 0), (0, 0, -1)]
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.library import gale_transform_to_primal
>>> points = ((Integer(0), -Integer(1)), (-Integer(1), Integer(0)), (Integer(1),
↪Integer(1)), (Integer(1), Integer(1)), (-Integer(1), Integer(0)), (Integer(0), -
↪Integer(1)))
>>> gale_transform_to_primal(points)
[(0, 0, 1), (0, 1, 0), (1, 0, 0), (-1, 0, 0), (0, -1, 0), (0, 0, -1)]
```

One can specify the base ring:

```
sage: p = [(1,1), (-1,-1), (1,0), (-1,0), (1,-1), (-2,1)]
sage: gtpp = gale_transform_to_primal(p); gtpp
```

(continues on next page)

(continued from previous page)

```
[(16, -35, 54),
 (24, 10, 31),
 (-15, 50, -60),
 (-25, 0, 0),
 (0, -25, 0),
 (0, 0, -25)]
sage: (matrix(RDF, gtpp)/25 +
....:  matrix(gale_transform_to_primal(p, base_ring=RDF))).norm() < 1e-15
True
```

```
>>> from sage.all import *
>>> p = [(Integer(1),Integer(1)), (-Integer(1),-Integer(1)), (Integer(1),
↪Integer(0)), (-Integer(1),Integer(0)), (Integer(1),-Integer(1)), (-Integer(2),
↪Integer(1))]
>>> gtpp = gale_transform_to_primal(p); gtpp
[(16, -35, 54),
 (24, 10, 31),
 (-15, 50, -60),
 (-25, 0, 0),
 (0, -25, 0),
 (0, 0, -25)]
>>> (matrix(RDF, gtpp)/Integer(25) +
...  matrix(gale_transform_to_primal(p, base_ring=RDF))).norm() < RealNumber('1e-
↪15')
True
```

One can also specify the backend to be used internally:

```
sage: gale_transform_to_primal(p, backend='field')
[(48, -71, 88),
 (84, -28, 99),
 (-77, 154, -132),
 (-55, 0, 0),
 (0, -55, 0),
 (0, 0, -55)]
sage: gale_transform_to_primal(p, backend='normaliz')           # optional -
↪pynormaliz
[(16, -35, 54),
 (24, 10, 31),
 (-15, 50, -60),
 (-25, 0, 0),
 (0, -25, 0),
 (0, 0, -25)]
```

```
>>> from sage.all import *
>>> gale_transform_to_primal(p, backend='field')
[(48, -71, 88),
 (84, -28, 99),
 (-77, 154, -132),
 (-55, 0, 0),
 (0, -55, 0),
 (0, 0, -55)]
```

(continues on next page)

(continued from previous page)

```
>>> gale_transform_to_primal(p, backend='normaliz')           # optional -
↪pynormaliz
[(16, -35, 54),
 (24, 10, 31),
 (-15, 50, -60),
 (-25, 0, 0),
 (0, -25, 0),
 (0, 0, -25)]
```

The input vectors should be totally cyclic:

```
sage: gale_transform_to_primal([(0,1), (1,0), (1,1), (-1,0)])
Traceback (most recent call last):
...
ValueError: input vectors not totally cyclic

sage: gale_transform_to_primal(
....:     [(1,1,0), (-1,-1,0), (1,0,0),
....:      (-1,0,0), (1,-1,0), (-2,1,0)], backend='field')
Traceback (most recent call last):
...
ValueError: input vectors not totally cyclic
```

```
>>> from sage.all import *
>>> gale_transform_to_primal([(Integer(0),Integer(1)), (Integer(1),Integer(0)),
↪(Integer(1),Integer(1)), (-Integer(1),Integer(0))])
Traceback (most recent call last):
...
ValueError: input vectors not totally cyclic

>>> gale_transform_to_primal(
...     [(Integer(1),Integer(1),Integer(0)), (-Integer(1),-Integer(1),Integer(0)),
↪ (Integer(1),Integer(0),Integer(0)),
...      (-Integer(1),Integer(0),Integer(0)), (Integer(1),-Integer(1),Integer(0)),
↪ (-Integer(2),Integer(1),Integer(0))], backend='field')
Traceback (most recent call last):
...
ValueError: input vectors not totally cyclic
```

sage.geometry.polyhedron.library.**project_points**(**points*, ***kwds*)

Projects a set of points into a vector space of dimension one less.

INPUT:

- `points...` – the points to project
- `base_ring` – (defaults to `RDF` if keyword is `None` or not provided in `kwds`) the base ring to use

The projection is isometric to the orthogonal projection on the hyperplane made of zero sum vector. Hence, if the set of points have all equal sums, then their projection is isometric (as a set of points).

The projection used is the matrix given by *zero_sum_projection()*.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.library import project_points
sage: project_points([2,-1,3,2])          # abs tol 1e-15
[(2.1213203435596424, -2.041241452319315, -0.577350269189626)]
sage: project_points([1,2,3],[3,3,5])     # abs tol 1e-15
[(-0.7071067811865475, -1.2247448713915892), (0.0, -1.6329931618554523)]
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.library import project_points
>>> project_points([Integer(2),-Integer(1),Integer(3),Integer(2)])          # abs␣
↪tol 1e-15
[(2.1213203435596424, -2.041241452319315, -0.577350269189626)]
>>> project_points([Integer(1),Integer(2),Integer(3)],[Integer(3),Integer(3),
↪Integer(5)])     # abs tol 1e-15
[(-0.7071067811865475, -1.2247448713915892), (0.0, -1.6329931618554523)]
```

These projections are compatible with the restriction. More precisely, given a vector v, the projection of v restricted to the first i coordinates will be equal to the projection of the first $i+1$ coordinates of v:

```
sage: project_points([1,2])    # abs tol 1e-15
[(-0.7071067811865475)]
sage: project_points([1,2,3])  # abs tol 1e-15
[(-0.7071067811865475, -1.2247448713915892)]
sage: project_points([1,2,3,4])     # abs tol 1e-15
[(-0.7071067811865475, -1.2247448713915892, -1.7320508075688776)]
sage: project_points([1,2,3,4,0])   # abs tol 1e-15
[(-0.7071067811865475, -1.2247448713915892, -1.7320508075688776, 2.
↪23606797749979)]
```

```
>>> from sage.all import *
>>> project_points([Integer(1),Integer(2)])    # abs tol 1e-15
[(-0.7071067811865475)]
>>> project_points([Integer(1),Integer(2),Integer(3)])  # abs tol 1e-15
[(-0.7071067811865475, -1.2247448713915892)]
>>> project_points([Integer(1),Integer(2),Integer(3),Integer(4)])     # abs tol␣
↪1e-15
[(-0.7071067811865475, -1.2247448713915892, -1.7320508075688776)]
>>> project_points([Integer(1),Integer(2),Integer(3),Integer(4),Integer(0)])   #␣
↪abs tol 1e-15
[(-0.7071067811865475, -1.2247448713915892, -1.7320508075688776, 2.
↪23606797749979)]
```

Check that it is (almost) an isometry:

```
sage: V = list(map(vector, IntegerVectors(n=5, length=3)))
sage: P = project_points(*V)
sage: for i in range(21):                                                       #␣
↪needs sage.combinat
....:     for j in range(21):
....:         assert abs((V[i]-V[j]).norm() - (P[i]-P[j]).norm()) < 0.00001
```

```
>>> from sage.all import *
>>> V = list(map(vector, IntegerVectors(n=Integer(5), length=Integer(3))))
```

(continues on next page)

(continued from previous page)

```
>>> P = project_points(*V)
>>> for i in range(Integer(21)):                                                      ␣
↪     # needs sage.combinat
...     for j in range(Integer(21)):
...         assert abs((V[i]-V[j]).norm() - (P[i]-P[j]).norm()) < RealNumber('0.
↪00001')
```

Example with exact computation:

```
sage: V = [ vector(v) for v in IntegerVectors(n=4, length=2) ]
sage: P = project_points(*V, base_ring=AA)                                          #␣
↪needs sage.combinat sage.rings.number_field
sage: for i in range(len(V)):                                                        #␣
↪needs sage.combinat sage.rings.number_field
....:     for j in range(len(V)):
....:         assert (V[i]-V[j]).norm() == (P[i]-P[j]).norm()
```

```
>>> from sage.all import *
>>> V = [ vector(v) for v in IntegerVectors(n=Integer(4), length=Integer(2)) ]
>>> P = project_points(*V, base_ring=AA)                                          #␣
↪needs sage.combinat sage.rings.number_field
>>> for i in range(len(V)):                                                        #␣
↪needs sage.combinat sage.rings.number_field
...     for j in range(len(V)):
...         assert (V[i]-V[j]).norm() == (P[i]-P[j]).norm()
```

`sage.geometry.polyhedron.library.`**`zero_sum_projection`**`(`*d*, *base_ring=None*`)`

Return a matrix corresponding to the projection on the orthogonal of $(1, 1, \ldots, 1)$ in dimension d.

The projection maps the orthonormal basis $(1, -1, 0, \ldots, 0)/\sqrt(2)$, $(1, 1, -1, 0, \ldots, 0)/\sqrt(3)$, ldots, $(1, 1, \ldots, 1, -1)/\sqrt(d)$ to the canonical basis in $\mathbf{R}^{d-1}$.

OUTPUT:

A matrix of dimensions $(d-1) \times d$ defined over `base_ring` (default: `RDF`).

EXAMPLES:

```
sage: from sage.geometry.polyhedron.library import zero_sum_projection
sage: zero_sum_projection(2)
[ 0.7071067811865475 -0.7071067811865475]
sage: zero_sum_projection(3)
[ 0.7071067811865475 -0.7071067811865475                  0.0]
[ 0.4082482904638631  0.4082482904638631 -0.8164965809277261]
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.library import zero_sum_projection
>>> zero_sum_projection(Integer(2))
[ 0.7071067811865475 -0.7071067811865475]
>>> zero_sum_projection(Integer(3))
[ 0.7071067811865475 -0.7071067811865475                  0.0]
[ 0.4082482904638631  0.4082482904638631 -0.8164965809277261]
```

Exact computation in `AA`:

```
sage: zero_sum_projection(3, base_ring=AA)                                  #␣
↪needs sage.rings.number_field
[ 0.7071067811865475? -0.7071067811865475?                    0]
[ 0.4082482904638630?  0.4082482904638630? -0.8164965809277260?]
```

```
>>> from sage.all import *
>>> zero_sum_projection(Integer(3), base_ring=AA)                             ␣
↪     # needs sage.rings.number_field
[ 0.7071067811865475? -0.7071067811865475?                    0]
[ 0.4082482904638630?  0.4082482904638630? -0.8164965809277260?]
```

2.1.2 Polyhedra

In this module, a polyhedron is a convex (possibly unbounded) set in Euclidean space cut out by a finite set of linear inequalities and linear equations. Note that the dimension of the polyhedron can be less than the dimension of the ambient space. There are two complementary representations of the same data:

H(alf-space/Hyperplane)-representation

This describes a polyhedron as the common solution set of a finite number of

- linear **inequalities** $A\vec{x} + b \geq 0$, and
- linear **equations** $C\vec{x} + d = 0$.

V(ertex)-representation

The other representation is as the convex hull of vertices (and rays and lines to all for unbounded polyhedra) as generators. The polyhedron is then the Minkowski sum

$$P = \text{conv}\{v_1, \ldots, v_k\} + \sum_{i=1}^{m} \mathbf{R}_+ r_i + \sum_{j=1}^{n} \mathbf{R}\ell_j$$

where

- **vertices** $v_1, \ldots, v_k$ are a finite number of points. Each vertex is specified by an arbitrary vector, and two points are equal if and only if the vector is the same.
- **rays** $r_1, \ldots, r_m$ are a finite number of directions (directions of infinity). Each ray is specified by a nonzero vector, and two rays are equal if and only if the vectors are the same up to rescaling with a positive constant.
- **lines** $\ell_1, \ldots, \ell_n$ are a finite number of unoriented directions. In other words, a line is equivalent to the set $\{r, -r\}$ for a ray r. Each line is specified by a nonzero vector, and two lines are equivalent if and only if the vectors are the same up to rescaling with a nonzero (possibly negative) constant.

When specifying a polyhedron, you can input a non-minimal set of inequalities/equations or generating vertices/rays/lines. The non-minimal generators are usually called points, non-extremal rays, and non-extremal lines, but for our purposes it is more convenient to always talk about vertices/rays/lines. Sage will remove any superfluous representation objects and always return a minimal representation. For example, $(0, 0)$ is a superfluous vertex here:

```
sage: triangle = Polyhedron(vertices=[(0,2), (-1,0), (1,0), (0,0)])
sage: triangle.vertices()
(A vertex at (-1, 0), A vertex at (1, 0), A vertex at (0, 2))
```

```
>>> from sage.all import *
>>> triangle = Polyhedron(vertices=[(Integer(0),Integer(2)), (-Integer(1),Integer(0)),
↪ (Integer(1),Integer(0)), (Integer(0),Integer(0))])
>>> triangle.vertices()
(A vertex at (-1, 0), A vertex at (1, 0), A vertex at (0, 2))
```

See also

If one only needs to keep track of a system of linear system of inequalities, one should also consider the class for mixed integer linear programming.

- Mixed Integer Linear Programming

Unbounded Polyhedra

A polytope is defined as a bounded polyhedron. In this case, the minimal representation is unique and a vertex of the minimal representation is equivalent to a 0-dimensional face of the polytope. This is why one generally does not distinguish vertices and 0-dimensional faces. But for non-bounded polyhedra we have to allow for a more general notion of "vertex" in order to make sense of the Minkowski sum presentation:

```
sage: half_plane = Polyhedron(ieqs=[(0,1,0)])
sage: half_plane.Hrepresentation()
(An inequality (1, 0) x + 0 >= 0,)
sage: half_plane.Vrepresentation()
(A line in the direction (0, 1), A ray in the direction (1, 0), A vertex at (0, 0))
```

```
>>> from sage.all import *
>>> half_plane = Polyhedron(ieqs=[(Integer(0),Integer(1),Integer(0))])
>>> half_plane.Hrepresentation()
(An inequality (1, 0) x + 0 >= 0,)
>>> half_plane.Vrepresentation()
(A line in the direction (0, 1), A ray in the direction (1, 0), A vertex at (0, 0))
```

Note how we need a point in the above example to anchor the ray and line. But any point on the boundary of the half-plane would serve the purpose just as well. Sage picked the origin here, but this choice is not unique. Similarly, the choice of ray is arbitrary but necessary to generate the half-plane.

Finally, note that while rays and lines generate unbounded edges of the polyhedron they are not in a one-to-one correspondence with them. For example, the infinite strip has two infinite edges (1-faces) but only one generating line:

```
sage: strip = Polyhedron(vertices=[(1,0),(-1,0)], lines=[(0,1)])
sage: strip.Hrepresentation()
(An inequality (1, 0) x + 1 >= 0, An inequality (-1, 0) x + 1 >= 0)
sage: strip.lines()
(A line in the direction (0, 1),)
sage: [f.ambient_V_indices() for f in strip.faces(1)]
[(0, 2), (0, 1)]
sage: for face in strip.faces(1):
....:     print(face.ambient_V_indices())
(0, 2)
(0, 1)
sage: for face in strip.faces(1):
....:     print("{} = {}".format(face.ambient_V_indices(), face.as_polyhedron().
↪Vrepresentation()))
(0, 2) = (A line in the direction (0, 1), A vertex at (1, 0))
(0, 1) = (A line in the direction (0, 1), A vertex at (-1, 0))
```

```
>>> from sage.all import *
>>> strip = Polyhedron(vertices=[(Integer(1),Integer(0)),(-Integer(1),Integer(0))],␣
```

(continues on next page)

(continued from previous page)

```
↪lines=[(Integer(0),Integer(1))])
>>> strip.Hrepresentation()
(An inequality (1, 0) x + 1 >= 0, An inequality (-1, 0) x + 1 >= 0)
>>> strip.lines()
(A line in the direction (0, 1),)
>>> [f.ambient_V_indices() for f in strip.faces(Integer(1))]
[(0, 2), (0, 1)]
>>> for face in strip.faces(Integer(1)):
...     print(face.ambient_V_indices())
(0, 2)
(0, 1)
>>> for face in strip.faces(Integer(1)):
...     print("{} = {}".format(face.ambient_V_indices(), face.as_polyhedron().
↪Vrepresentation()))
(0, 2) = (A line in the direction (0, 1), A vertex at (1, 0))
(0, 1) = (A line in the direction (0, 1), A vertex at (-1, 0))
```

EXAMPLES:

```
sage: trunc_quadr = Polyhedron(vertices=[[1,0],[0,1]], rays=[[1,0],[0,1]])
sage: trunc_quadr
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 2 vertices and 2 rays
sage: v = next(trunc_quadr.vertex_generator())  # the first vertex in the internal␣
↪enumeration
sage: v
A vertex at (0, 1)
sage: v.vector()
(0, 1)
sage: list(v)
[0, 1]
sage: len(v)
2
sage: v[0] + v[1]
1
sage: v.is_vertex()
True
sage: type(v)
<class 'sage.geometry.polyhedron.representation.Vertex'>
sage: type( v() )
<class 'sage.modules.vector_rational_dense.Vector_rational_dense'>
sage: v.polyhedron()
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 2 vertices and 2 rays
sage: r = next(trunc_quadr.ray_generator())
sage: r
A ray in the direction (0, 1)
sage: r.vector()
(0, 1)
sage: list( v.neighbors() )
[A ray in the direction (0, 1), A vertex at (1, 0)]
```

```
>>> from sage.all import *
>>> trunc_quadr = Polyhedron(vertices=[[Integer(1),Integer(0)],[Integer(0),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1)]], rays=[[Integer(1),Integer(0)],[Integer(0),Integer(1)]])
>>> trunc_quadr
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 2 vertices and 2 rays
>>> v = next(trunc_quadr.vertex_generator())  # the first vertex in the internal␣
↪enumeration
>>> v
A vertex at (0, 1)
>>> v.vector()
(0, 1)
>>> list(v)
[0, 1]
>>> len(v)
2
>>> v[Integer(0)] + v[Integer(1)]
1
>>> v.is_vertex()
True
>>> type(v)
<class 'sage.geometry.polyhedron.representation.Vertex'>
>>> type( v() )
<class 'sage.modules.vector_rational_dense.Vector_rational_dense'>
>>> v.polyhedron()
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 2 vertices and 2 rays
>>> r = next(trunc_quadr.ray_generator())
>>> r
A ray in the direction (0, 1)
>>> r.vector()
(0, 1)
>>> list( v.neighbors() )
[A ray in the direction (0, 1), A vertex at (1, 0)]
```

Inequalities $A\vec{x} + b \geq 0$ (and, similarly, equations) are specified by a list `[b, A]`:

```
sage: Polyhedron(ieqs=[(0,1,0),(0,0,1),(1,-1,-1)]).Hrepresentation()
(An inequality (-1, -1) x + 1 >= 0,
 An inequality (1, 0) x + 0 >= 0,
 An inequality (0, 1) x + 0 >= 0)
```

```
>>> from sage.all import *
>>> Polyhedron(ieqs=[(Integer(0),Integer(1),Integer(0)),(Integer(0),Integer(0),
↪Integer(1)),(Integer(1),-Integer(1),-Integer(1))]).Hrepresentation()
(An inequality (-1, -1) x + 1 >= 0,
 An inequality (1, 0) x + 0 >= 0,
 An inequality (0, 1) x + 0 >= 0)
```

See *`Polyhedron()`* for a detailed description of all possible ways to construct a polyhedron.

Base Rings

The base ring of the polyhedron can be specified by the `base_ring` optional keyword argument. If not specified, a suitable common base ring for all coordinates and coefficients will be chosen automatically. Important cases are:

- `base_ring=QQ` uses a fast implementation for exact rational numbers.

- base_ring=ZZ is similar to QQ, but the resulting polyhedron object will have extra methods for lattice polyhedra.
- base_ring=RDF uses floating point numbers, this is fast but susceptible to numerical errors.

Polyhedra with symmetries often are defined over some algebraic field extension of the rationals. As a simple example, consider the equilateral triangle whose vertex coordinates involve $\sqrt{3}$. An exact way to work with roots in Sage is the Algebraic Real Field

```
sage: triangle = Polyhedron([(0,0), (1,0), (1/2, sqrt(3)/2)], base_ring=AA)          #␣
↪needs sage.rings.number_field sage.symbolic
sage: triangle.Hrepresentation()                                                      #␣
↪needs sage.rings.number_field sage.symbolic
(An inequality (-1, -0.5773502691896258?) x + 1 >= 0,
 An inequality (1, -0.5773502691896258?) x + 0 >= 0,
 An inequality (0, 1.154700538379252?) x + 0 >= 0)
```

```
>>> from sage.all import *
>>> triangle = Polyhedron([(Integer(0),Integer(0)), (Integer(1),Integer(0)),␣
↪(Integer(1)/Integer(2), sqrt(Integer(3))/Integer(2))], base_ring=AA)          #␣
↪needs sage.rings.number_field sage.symbolic
>>> triangle.Hrepresentation()                                                  #␣
↪needs sage.rings.number_field sage.symbolic
(An inequality (-1, -0.5773502691896258?) x + 1 >= 0,
 An inequality (1, -0.5773502691896258?) x + 0 >= 0,
 An inequality (0, 1.154700538379252?) x + 0 >= 0)
```

Without specifying the base_ring, the sqrt(3) would be a symbolic ring element and, therefore, the polyhedron defined over the symbolic ring. This is currently not supported as SR is not exact:

```
sage: Polyhedron([(0,0), (1,0), (1/2, sqrt(3)/2)])                                    #␣
↪needs sage.symbolic
Traceback (most recent call last):
...
ValueError: no default backend for computations with Symbolic Ring

sage: SR.is_exact()                                                                   #␣
↪needs sage.symbolic
False
```

```
>>> from sage.all import *
>>> Polyhedron([(Integer(0),Integer(0)), (Integer(1),Integer(0)), (Integer(1)/
↪Integer(2), sqrt(Integer(3))/Integer(2))])                                    # needs␣
↪sage.symbolic
Traceback (most recent call last):
...
ValueError: no default backend for computations with Symbolic Ring

>>> SR.is_exact()                                                               #␣
↪needs sage.symbolic
False
```

Even faster than all algebraic real numbers (the field AA) is to take the smallest extension field. For the equilateral triangle, that would be:

```
sage: x = polygen(ZZ, 'x')
sage: K.<sqrt3> = NumberField(x^2 - 3, embedding=AA(3)**(1/2))                #␣
↪needs sage.rings.number_field
sage: Polyhedron([(0,0), (1,0), (1/2, sqrt3/2)])                              #␣
↪needs sage.rings.number_field
A 2-dimensional polyhedron in
 (Number Field in sqrt3 with defining polynomial x^2 - 3 with sqrt3 = 1.
↪732050807568878?)^2
 defined as the convex hull of 3 vertices
```

```
>>> from sage.all import *
>>> x = polygen(ZZ, 'x')
>>> K = NumberField(x**Integer(2) - Integer(3), embedding=AA(Integer(3))**(Integer(1)/
↪Integer(2)), names=('sqrt3',)); (sqrt3,) = K._first_ngens(1)# needs sage.rings.
↪number_field
>>> Polyhedron([(Integer(0),Integer(0)), (Integer(1),Integer(0)), (Integer(1)/
↪Integer(2), sqrt3/Integer(2))])                                # needs sage.
↪rings.number_field
A 2-dimensional polyhedron in
 (Number Field in sqrt3 with defining polynomial x^2 - 3 with sqrt3 = 1.
↪732050807568878?)^2
 defined as the convex hull of 3 vertices
```

Warning

Be careful when you construct polyhedra with floating point numbers. The only available backend for such computation is `cdd` which uses machine floating point numbers which have limited precision. If the input consists of floating point numbers and the `base_ring` is not specified, the base ring is set to be the `RealField` with the precision given by the minimal bit precision of the input. Then, if the obtained minimum is 53 bits of precision, the constructor converts automatically the base ring to `RDF`. Otherwise, it returns an error:

```
sage: Polyhedron(vertices = [[1.12345678901234, 2.12345678901234]])
A 0-dimensional polyhedron in RDF^2 defined as the convex hull of 1 vertex
sage: Polyhedron(vertices = [[1.12345678901234, 2.123456789012345]])
A 0-dimensional polyhedron in RDF^2 defined as the convex hull of 1 vertex
sage: Polyhedron(vertices = [[1.123456789012345, 2.123456789012345]])        #␣
↪needs sage.rings.real_mpfr
Traceback (most recent call last):
...
ValueError: the only allowed inexact ring is 'RDF' with backend 'cdd'
```

```
>>> from sage.all import *
>>> Polyhedron(vertices = [[RealNumber('1.12345678901234'), RealNumber('2.
↪12345678901234')]])
A 0-dimensional polyhedron in RDF^2 defined as the convex hull of 1 vertex
>>> Polyhedron(vertices = [[RealNumber('1.12345678901234'), RealNumber('2.
↪123456789012345')]])
A 0-dimensional polyhedron in RDF^2 defined as the convex hull of 1 vertex
>>> Polyhedron(vertices = [[RealNumber('1.123456789012345'), RealNumber('2.
↪123456789012345')]])           # needs sage.rings.real_mpfr
Traceback (most recent call last):
...
ValueError: the only allowed inexact ring is 'RDF' with backend 'cdd'
```

The strongly suggested method to input floating point numbers is to specify the `base_ring` to be `RDF`:

```
sage: Polyhedron(vertices = [[1.123456789012345, 2.123456789012345]], base_ring=RDF)
A 0-dimensional polyhedron in RDF^2 defined as the convex hull of 1 vertex
```

```
>>> from sage.all import *
>>> Polyhedron(vertices = [[RealNumber('1.123456789012345'), RealNumber('2.
↪123456789012345')]], base_ring=RDF)
A 0-dimensional polyhedron in RDF^2 defined as the convex hull of 1 vertex
```

See also

Parents for polyhedra

Base classes

Depending on the chosen base ring, a specific class is used to represent the polyhedron object.

See also

- *Base class for polyhedra*
- *Base class for polyhedra over integers*
- *Base class for polyhedra over rationals*
- *Base class for polyhedra over RDF*

The most important base class is **Base class for polyhedra** from which other base classes and backends inherit.

Backends

There are different backends available to deal with polyhedron objects.

See also

- *cdd backend for polyhedra*
- *field backend for polyhedra*
- *normaliz backend for polyhedra*
- *ppl backend for polyhedra*

Note

Depending on the backend used, it may occur that different methods are available or not.

Appendix

REFERENCES:

Komei Fukuda's FAQ in Polyhedral Computation

AUTHORS:

- Marshall Hampton: first version, bug fixes, and various improvements, 2008 and 2009
- Arnaud Bergeron: improvements to triangulation and rendering, 2008
- Sebastien Barthelemy: documentation improvements, 2008
- Volker Braun: refactoring, handle non-compact case, 2009 and 2010
- Andrey Novoseltsev: added lattice_from_incidences, 2010
- Volker Braun: rewrite to use PPL instead of cddlib, 2011
- Volker Braun: Add support for arbitrary subfields of the reals

`sage.geometry.polyhedron.constructor.`**`Polyhedron`**`(`*vertices=None*, *rays=None*, *lines=None*, *ieqs=None*, *eqns=None*, *ambient_dim=None*, *base_ring=None*, *minimize=True*, *verbose=False*, *backend=None*, *mutable=False*`)`

Construct a polyhedron object.

You may either define it with vertex/ray/line or inequalities/equations data, but not both. Redundant data will automatically be removed (unless `minimize=False`), and the complementary representation will be computed.

INPUT:

- `vertices` – iterable of points. Each point can be specified as any iterable container of `base_ring` elements. If `rays` or `lines` are specified but no `vertices`, the origin is taken to be the single vertex.

  Instead of vertices, the first argument can also be an object that can be converted to a `Polyhedron()` via an `as_polyhedron()` or `polyhedron()` method. In this case, the following 5 arguments cannot be provided.
- `rays` – list of rays; each ray can be specified as any iterable container of `base_ring` elements
- `lines` – list of lines; each line can be specified as any iterable container of `base_ring` elements
- `ieqs` – list of inequalities; each line can be specified as any iterable container of `base_ring` elements. An entry equal to `[-1,7,3,4]` represents the inequality $7x_1 + 3x_2 + 4x_3 \geq 1$.
- `eqns` – list of equalities; each line can be specified as any iterable container of `base_ring` elements. An entry equal to `[-1,7,3,4]` represents the equality $7x_1 + 3x_2 + 4x_3 = 1$.
- `ambient_dim` – integer; the ambient space dimension. Usually can be figured out automatically from the H/Vrepresentation dimensions.
- `base_ring` – a sub-field of the reals implemented in Sage. The field over which the polyhedron will be defined. For `QQ` and algebraic extensions, exact arithmetic will be used. For `RDF`, floating point numbers will be used. Floating point arithmetic is faster but might give the wrong result for degenerate input.
- `backend` – string or `None` (default). The backend to use. Valid choices are
  - `'cdd'`: use cdd (`backend_cdd`) with $\mathbf{Q}$ or $\mathbf{R}$ coefficients depending on `base_ring`
  - `'normaliz'`: use normaliz (`backend_normaliz`) with $\mathbf{Z}$ or $\mathbf{Q}$ coefficients depending on `base_ring`
  - `'polymake'`: use polymake (`backend_polymake`) with $\mathbf{Q}$, $\mathbf{R}$ or `QuadraticField` coefficients depending on `base_ring`
  - `'ppl'`: use ppl (`backend_ppl`) with $\mathbf{Z}$ or $\mathbf{Q}$ coefficients depending on `base_ring`

   - `'field'`: use python implementation (`backend_field`) for any field

Some backends support further optional arguments:

- `minimize` – boolean (default: `True`); whether to immediately remove redundant H/V-representation data; currently not used.
- `verbose` – boolean (default: `False`); whether to print verbose output for debugging purposes; only supported by the cdd and normaliz backends
- `mutable` – boolean (default: `False`); whether the polyhedron is mutable

OUTPUT: the polyhedron defined by the input data

EXAMPLES:

Construct some polyhedra:

```
sage: square_from_vertices = Polyhedron(vertices = [[1, 1], [1, -1], [-1, 1], [-1,
↪ -1]])
sage: square_from_ieqs = Polyhedron(ieqs = [[1, 0, 1], [1, 1, 0], [1, 0, -1], [1,␣
↪-1, 0]])
sage: list(square_from_ieqs.vertex_generator())
[A vertex at (1, -1),
 A vertex at (1, 1),
 A vertex at (-1, 1),
 A vertex at (-1, -1)]
sage: list(square_from_vertices.inequality_generator())
[An inequality (1, 0) x + 1 >= 0,
 An inequality (0, 1) x + 1 >= 0,
 An inequality (-1, 0) x + 1 >= 0,
 An inequality (0, -1) x + 1 >= 0]
sage: p = Polyhedron(vertices = [[1.1, 2.2], [3.3, 4.4]], base_ring=RDF)
sage: p.n_inequalities()
2
```

```
>>> from sage.all import *
>>> square_from_vertices = Polyhedron(vertices = [[Integer(1), Integer(1)],␣
↪[Integer(1), -Integer(1)], [-Integer(1), Integer(1)], [-Integer(1), -
↪Integer(1)]])
>>> square_from_ieqs = Polyhedron(ieqs = [[Integer(1), Integer(0), Integer(1)],␣
↪[Integer(1), Integer(1), Integer(0)], [Integer(1), Integer(0), -Integer(1)],␣
↪[Integer(1), -Integer(1), Integer(0)]])
>>> list(square_from_ieqs.vertex_generator())
[A vertex at (1, -1),
 A vertex at (1, 1),
 A vertex at (-1, 1),
 A vertex at (-1, -1)]
>>> list(square_from_vertices.inequality_generator())
[An inequality (1, 0) x + 1 >= 0,
 An inequality (0, 1) x + 1 >= 0,
 An inequality (-1, 0) x + 1 >= 0,
 An inequality (0, -1) x + 1 >= 0]
>>> p = Polyhedron(vertices = [[RealNumber('1.1'), RealNumber('2.2')],␣
↪[RealNumber('3.3'), RealNumber('4.4')]], base_ring=RDF)
>>> p.n_inequalities()
2
```

The same polyhedron given in two ways:

```
sage: p = Polyhedron(ieqs = [[0,1,0,0],[0,0,1,0]])
sage: p.Vrepresentation()
(A line in the direction (0, 0, 1),
 A ray in the direction (1, 0, 0),
 A ray in the direction (0, 1, 0),
 A vertex at (0, 0, 0))
sage: q = Polyhedron(vertices=[[0,0,0]], rays=[[1,0,0],[0,1,0]], lines=[[0,0,1]])
sage: q.Hrepresentation()
(An inequality (1, 0, 0) x + 0 >= 0,
 An inequality (0, 1, 0) x + 0 >= 0)
```

```
>>> from sage.all import *
>>> p = Polyhedron(ieqs = [[Integer(0),Integer(1),Integer(0),Integer(0)],
↪[Integer(0),Integer(0),Integer(1),Integer(0)]])
>>> p.Vrepresentation()
(A line in the direction (0, 0, 1),
 A ray in the direction (1, 0, 0),
 A ray in the direction (0, 1, 0),
 A vertex at (0, 0, 0))
>>> q = Polyhedron(vertices=[[Integer(0),Integer(0),Integer(0)]],␣
↪rays=[[Integer(1),Integer(0),Integer(0)],[Integer(0),Integer(1),Integer(0)]],␣
↪lines=[[Integer(0),Integer(0),Integer(1)]])
>>> q.Hrepresentation()
(An inequality (1, 0, 0) x + 0 >= 0,
 An inequality (0, 1, 0) x + 0 >= 0)
```

Finally, a more complicated example. Take $\mathbb{R}^6_{\geq 0}$ with coordinates $a, b, \dots, f$ and

- The inequality $e + b \geq c + d$
- The inequality $e + c \geq b + d$
- The equation $a + b + c + d + e + f = 31$

```
sage: positive_coords = Polyhedron(ieqs=[
....:     [0, 1, 0, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0, 0],
....:     [0, 0, 0, 0, 1, 0, 0], [0, 0, 0, 0, 0, 1, 0], [0, 0, 0, 0, 0, 0, 1]])
sage: P = Polyhedron(ieqs=positive_coords.inequalities() + (
....:     [0,0,1,-1,-1,1,0], [0,0,-1,1,-1,1,0]), eqns=[[-31,1,1,1,1,1,1]])
sage: P
A 5-dimensional polyhedron in QQ^6 defined as the convex hull of 7 vertices
sage: P.dim()
5
sage: P.Vrepresentation()
(A vertex at (31, 0, 0, 0, 0, 0), A vertex at (0, 0, 0, 0, 0, 31),
 A vertex at (0, 0, 0, 0, 31, 0), A vertex at (0, 0, 31/2, 0, 31/2, 0),
 A vertex at (0, 31/2, 31/2, 0, 0, 0), A vertex at (0, 31/2, 0, 0, 31/2, 0),
 A vertex at (0, 0, 0, 31/2, 31/2, 0))
```

```
>>> from sage.all import *
>>> positive_coords = Polyhedron(ieqs=[
...     [Integer(0), Integer(1), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(0)], [Integer(0), Integer(0), Integer(1), Integer(0), Integer(0),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(0), Integer(0)], [Integer(0), Integer(0), Integer(0), Integer(1),␣
↪Integer(0), Integer(0), Integer(0)],
...      [Integer(0), Integer(0), Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(0)], [Integer(0), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(1), Integer(0)], [Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(0), Integer(0), Integer(1)]])
>>> P = Polyhedron(ieqs=positive_coords.inequalities() + (
...      [Integer(0),Integer(0),Integer(1),-Integer(1),-Integer(1),Integer(1),
↪Integer(0)], [Integer(0),Integer(0),-Integer(1),Integer(1),-Integer(1),
↪Integer(1),Integer(0)]), eqns=[[-Integer(31),Integer(1),Integer(1),Integer(1),
↪Integer(1),Integer(1),Integer(1)]])
>>> P
A 5-dimensional polyhedron in QQ^6 defined as the convex hull of 7 vertices
>>> P.dim()
5
>>> P.Vrepresentation()
(A vertex at (31, 0, 0, 0, 0, 0), A vertex at (0, 0, 0, 0, 0, 31),
 A vertex at (0, 0, 0, 0, 31, 0), A vertex at (0, 0, 31/2, 0, 31/2, 0),
 A vertex at (0, 31/2, 31/2, 0, 0, 0), A vertex at (0, 31/2, 0, 0, 31/2, 0),
 A vertex at (0, 0, 0, 31/2, 31/2, 0))
```

Regular icosahedron, centered at 0 with edge length 2, with vertices given by the cyclic shifts of $(0, \pm 1, \pm(1+\sqrt(5))/2)$, cf. Wikipedia article Regular_icosahedron. It needs a number field:

```
sage: # needs sage.rings.number_field
sage: R0.<r0> = QQ[]
sage: R1.<r1> = NumberField(r0^2-5, embedding=AA(5)**(1/2))
sage: gold = (1+r1)/2
sage: v = [[0, 1, gold], [0, 1, -gold], [0, -1, gold], [0, -1, -gold]]
sage: pp = Permutation((1, 2, 3))
sage: icosah = Polyhedron(                                                    #␣
↪needs sage.combinat
....:     [(pp^2).action(w) for w in v] + [pp.action(w) for w in v] + v,
....:     base_ring=R1)
sage: len(icosah.faces(2))                                                    #␣
↪needs sage.combinat
20
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> R0 = QQ['r0']; (r0,) = R0._first_ngens(1)
>>> R1 = NumberField(r0**Integer(2)-Integer(5),␣
↪embedding=AA(Integer(5))**(Integer(1)/Integer(2)), names=('r1',)); (r1,) = R1._
↪first_ngens(1)
>>> gold = (Integer(1)+r1)/Integer(2)
>>> v = [[Integer(0), Integer(1), gold], [Integer(0), Integer(1), -gold],␣
↪[Integer(0), -Integer(1), gold], [Integer(0), -Integer(1), -gold]]
>>> pp = Permutation((Integer(1), Integer(2), Integer(3)))
>>> icosah = Polyhedron(                                                  #␣
↪needs sage.combinat
...     [(pp**Integer(2)).action(w) for w in v] + [pp.action(w) for w in v] + v,
...     base_ring=R1)
```

(continues on next page)

(continued from previous page)

```
>>> len(icosah.faces(Integer(2)))                                              ␣
↪     # needs sage.combinat
20
```

When the input contains elements of a Number Field, they require an embedding:

```
sage: # needs sage.rings.number_field
sage: x = polygen(ZZ, 'x')
sage: K = NumberField(x^2 - 2,'s')
sage: s = K.0
sage: L = NumberField(x^3 - 2,'t')
sage: t = L.0
sage: P = Polyhedron(vertices=[[0,s], [t,0]])
Traceback (most recent call last):
...
ValueError: invalid base ring
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> x = polygen(ZZ, 'x')
>>> K = NumberField(x**Integer(2) - Integer(2),'s')
>>> s = K.gen(0)
>>> L = NumberField(x**Integer(3) - Integer(2),'t')
>>> t = L.gen(0)
>>> P = Polyhedron(vertices=[[Integer(0),s], [t,Integer(0)]])
Traceback (most recent call last):
...
ValueError: invalid base ring
```

Converting from a given polyhedron:

```
sage: cb = polytopes.cube(); cb
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 8 vertices
sage: Polyhedron(cb, base_ring=QQ)
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 8 vertices
```

```
>>> from sage.all import *
>>> cb = polytopes.cube(); cb
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 8 vertices
>>> Polyhedron(cb, base_ring=QQ)
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 8 vertices
```

Converting from other objects to a polyhedron:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: Polyhedron(quadrant)
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex and 2␣
↪rays
sage: Polyhedron(quadrant, base_ring=QQ)
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2␣
↪rays
```

(continues on next page)

(continued from previous page)

```
sage: o = lattice_polytope.cross_polytope(2)
sage: Polyhedron(o)
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices
sage: Polyhedron(o, base_ring=QQ)
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 4 vertices

sage: p = MixedIntegerLinearProgram(solver='PPL')
sage: x, y = p['x'], p['y']
sage: p.add_constraint(x <= 1)
sage: p.add_constraint(x >= -1)
sage: p.add_constraint(y <= 1)
sage: p.add_constraint(y >= -1)
sage: Polyhedron(p, base_ring=ZZ)
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices
sage: Polyhedron(p)
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 4 vertices

sage: # needs sage.combinat
sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: h = x + y - 1; h
Hyperplane x + y - 1
sage: Polyhedron(h, base_ring=ZZ)
A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex and 1␣
↪line
sage: Polyhedron(h)
A 1-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 1␣
↪line
```

```
>>> from sage.all import *
>>> quadrant = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> Polyhedron(quadrant)
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex and 2␣
↪rays
>>> Polyhedron(quadrant, base_ring=QQ)
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2␣
↪rays

>>> o = lattice_polytope.cross_polytope(Integer(2))
>>> Polyhedron(o)
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices
>>> Polyhedron(o, base_ring=QQ)
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 4 vertices

>>> p = MixedIntegerLinearProgram(solver='PPL')
>>> x, y = p['x'], p['y']
>>> p.add_constraint(x <= Integer(1))
>>> p.add_constraint(x >= -Integer(1))
>>> p.add_constraint(y <= Integer(1))
>>> p.add_constraint(y >= -Integer(1))
>>> Polyhedron(p, base_ring=ZZ)
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices
>>> Polyhedron(p)
```

(continues on next page)

(continued from previous page)

```
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 4 vertices

>>> # needs sage.combinat
>>> H = HyperplaneArrangements(QQ, names=('x', 'y',)); (x, y,) = H._first_ngens(2)
>>> h = x + y - Integer(1); h
Hyperplane x + y - 1
>>> Polyhedron(h, base_ring=ZZ)
A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex and 1␣
↪line
>>> Polyhedron(h)
A 1-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 1␣
↪line
```

Note

- Once constructed, a `Polyhedron` object is immutable.
- Although the option `base_ring=RDF` allows numerical data to be used, it might not give the right answer for degenerate input data - the results can depend upon the tolerance setting of cdd.

See also

Library of polytopes

2.1.3 Parents for Polyhedra

`sage.geometry.polyhedron.parent.`**`Polyhedra`**(*ambient_space_or_base_ring*, *ambient_dim*, *backend=None*, *ambient_space=None*, *base_ring=None*)

Construct a suitable parent class for polyhedra.

INPUT:

- `base_ring` – a ring; currently there are backends for **Z**, **Q**, and **R**
- `ambient_dim` – integer; the ambient space dimension
- `ambient_space` – a free module
- `backend` – string. The name of the backend for computations. There are several backends implemented:
  – `backend="ppl"` uses the Parma Polyhedra Library
  – `backend="cdd"` uses CDD
  – `backend="normaliz"` uses normaliz
  – `backend="polymake"` uses polymake
  – `backend="field"` a generic Sage implementation

OUTPUT:

A parent class for polyhedra over the given base ring if the backend supports it. If not, the parent base ring can be larger (for example, **Q** instead of **Z**). If there is no implementation at all, a `ValueError` is raised.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.parent import Polyhedra
sage: Polyhedra(AA, 3)                                                      #␣
↪needs sage.rings.number_field
Polyhedra in AA^3
sage: Polyhedra(ZZ, 3)
Polyhedra in ZZ^3
sage: type(_)
<class 'sage.geometry.polyhedron.parent.Polyhedra_ZZ_ppl_with_category'>
sage: Polyhedra(QQ, 3, backend='cdd')
Polyhedra in QQ^3
sage: type(_)
<class 'sage.geometry.polyhedron.parent.Polyhedra_QQ_cdd_with_category'>
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.parent import Polyhedra
>>> Polyhedra(AA, Integer(3))                                               ␣
↪     # needs sage.rings.number_field
Polyhedra in AA^3
>>> Polyhedra(ZZ, Integer(3))
Polyhedra in ZZ^3
>>> type(_)
<class 'sage.geometry.polyhedron.parent.Polyhedra_ZZ_ppl_with_category'>
>>> Polyhedra(QQ, Integer(3), backend='cdd')
Polyhedra in QQ^3
>>> type(_)
<class 'sage.geometry.polyhedron.parent.Polyhedra_QQ_cdd_with_category'>
```

CDD does not support integer polytopes directly:

```
sage: Polyhedra(ZZ, 3, backend='cdd')
Polyhedra in QQ^3
```

```
>>> from sage.all import *
>>> Polyhedra(ZZ, Integer(3), backend='cdd')
Polyhedra in QQ^3
```

Using a more general form of the constructor:

```
sage: V = VectorSpace(QQ, 3)
sage: Polyhedra(V) is Polyhedra(QQ, 3)
True
sage: Polyhedra(V, backend='field') is Polyhedra(QQ, 3, 'field')
True
sage: Polyhedra(backend='field', ambient_space=V) is Polyhedra(QQ, 3, 'field')
True

sage: M = FreeModule(ZZ, 2)
sage: Polyhedra(M, backend='ppl') is Polyhedra(ZZ, 2, 'ppl')
True
```

```
>>> from sage.all import *
>>> V = VectorSpace(QQ, Integer(3))
```

(continues on next page)

(continued from previous page)

```
>>> Polyhedra(V) is Polyhedra(QQ, Integer(3))
True
>>> Polyhedra(V, backend='field') is Polyhedra(QQ, Integer(3), 'field')
True
>>> Polyhedra(backend='field', ambient_space=V) is Polyhedra(QQ, Integer(3),
↪'field')
True

>>> M = FreeModule(ZZ, Integer(2))
>>> Polyhedra(M, backend='ppl') is Polyhedra(ZZ, Integer(2), 'ppl')
True
```

class sage.geometry.polyhedron.parent.**Polyhedra_QQ_cdd**(*base_ring*, *ambient_dim*, *backend*)

Bases: *Polyhedra_base*

Element

alias of *Polyhedron_QQ_cdd*

class sage.geometry.polyhedron.parent.**Polyhedra_QQ_normaliz**(*base_ring*, *ambient_dim*, *backend*)

Bases: *Polyhedra_base*

Element

alias of *Polyhedron_QQ_normaliz*

class sage.geometry.polyhedron.parent.**Polyhedra_QQ_ppl**(*base_ring*, *ambient_dim*, *backend*)

Bases: *Polyhedra_base*

Element

alias of *Polyhedron_QQ_ppl*

class sage.geometry.polyhedron.parent.**Polyhedra_RDF_cdd**(*base_ring*, *ambient_dim*, *backend*)

Bases: *Polyhedra_base*

Element

alias of *Polyhedron_RDF_cdd*

class sage.geometry.polyhedron.parent.**Polyhedra_ZZ_normaliz**(*base_ring*, *ambient_dim*, *backend*)

Bases: *Polyhedra_base*

Element

alias of *Polyhedron_ZZ_normaliz*

class sage.geometry.polyhedron.parent.**Polyhedra_ZZ_ppl**(*base_ring*, *ambient_dim*, *backend*)

Bases: *Polyhedra_base*

Element

alias of *Polyhedron_ZZ_ppl*

class sage.geometry.polyhedron.parent.**Polyhedra_base**(*base_ring*, *ambient_dim*, *backend*)

Bases: UniqueRepresentation, Parent

Polyhedra in a fixed ambient space.

INPUT:

- base_ring – either ZZ, QQ, or RDF; the base ring of the ambient module/vector space
- ambient_dim – integer; the ambient space dimension

- `backend` – string; the name of the backend for computations. There are several backends implemented:
  - `backend="ppl"` uses the Parma Polyhedra Library
  - `backend="cdd"` uses CDD
  - `backend="normaliz"` uses normaliz
  - `backend="polymake"` uses polymake
  - `backend="field"` a generic Sage implementation

EXAMPLES:

```
sage: from sage.geometry.polyhedron.parent import Polyhedra
sage: Polyhedra(ZZ, 3)
Polyhedra in ZZ^3
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.parent import Polyhedra
>>> Polyhedra(ZZ, Integer(3))
Polyhedra in ZZ^3
```

Hrepresentation_space()

Return the linear space containing the H-representation vectors.

OUTPUT: a free module over the base ring of dimension *ambient_dim()* + 1

EXAMPLES:

```
sage: from sage.geometry.polyhedron.parent import Polyhedra
sage: Polyhedra(ZZ, 2).Hrepresentation_space()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.parent import Polyhedra
>>> Polyhedra(ZZ, Integer(2)).Hrepresentation_space()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

Vrepresentation_space()

Return the ambient vector space.

This is the vector space or module containing the Vrepresentation vectors.

OUTPUT: a free module over the base ring of dimension *ambient_dim()*

EXAMPLES:

```
sage: from sage.geometry.polyhedron.parent import Polyhedra
sage: Polyhedra(QQ, 4).Vrepresentation_space()
Vector space of dimension 4 over Rational Field
sage: Polyhedra(QQ, 4).ambient_space()
Vector space of dimension 4 over Rational Field
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.parent import Polyhedra
>>> Polyhedra(QQ, Integer(4)).Vrepresentation_space()
Vector space of dimension 4 over Rational Field
```

(continues on next page)

(continued from previous page)

```
>>> Polyhedra(QQ, Integer(4)).ambient_space()
Vector space of dimension 4 over Rational Field
```

ambient_dim()

Return the dimension of the ambient space.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.parent import Polyhedra
sage: Polyhedra(QQ, 3).ambient_dim()
3
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.parent import Polyhedra
>>> Polyhedra(QQ, Integer(3)).ambient_dim()
3
```

ambient_space()

Return the ambient vector space.

This is the vector space or module containing the Vrepresentation vectors.

OUTPUT: a free module over the base ring of dimension *ambient_dim()*

EXAMPLES:

```
sage: from sage.geometry.polyhedron.parent import Polyhedra
sage: Polyhedra(QQ, 4).Vrepresentation_space()
Vector space of dimension 4 over Rational Field
sage: Polyhedra(QQ, 4).ambient_space()
Vector space of dimension 4 over Rational Field
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.parent import Polyhedra
>>> Polyhedra(QQ, Integer(4)).Vrepresentation_space()
Vector space of dimension 4 over Rational Field
>>> Polyhedra(QQ, Integer(4)).ambient_space()
Vector space of dimension 4 over Rational Field
```

an_element()

Return a Polyhedron.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.parent import Polyhedra
sage: Polyhedra(QQ, 4).an_element()
A 4-dimensional polyhedron in QQ^4 defined as the convex hull of 5 vertices
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.parent import Polyhedra
>>> Polyhedra(QQ, Integer(4)).an_element()
A 4-dimensional polyhedron in QQ^4 defined as the convex hull of 5 vertices
```

backend()

Return the backend.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.parent import Polyhedra
sage: Polyhedra(QQ, 3).backend()
'ppl'
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.parent import Polyhedra
>>> Polyhedra(QQ, Integer(3)).backend()
'ppl'
```

base_extend(*base_ring*, *backend=None*, *ambient_dim=None*)

Return the base extended parent.

INPUT:

- `base_ring`, `backend` – see *Polyhedron()*
- `ambient_dim` – if not `None` change ambient dimension accordingly

EXAMPLES:

```
sage: from sage.geometry.polyhedron.parent import Polyhedra
sage: Polyhedra(ZZ, 3).base_extend(QQ)
Polyhedra in QQ^3
sage: Polyhedra(ZZ, 3).an_element().base_extend(QQ)
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 4 vertices
sage: Polyhedra(QQ, 2).base_extend(ZZ)
Polyhedra in QQ^2
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.parent import Polyhedra
>>> Polyhedra(ZZ, Integer(3)).base_extend(QQ)
Polyhedra in QQ^3
>>> Polyhedra(ZZ, Integer(3)).an_element().base_extend(QQ)
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 4 vertices
>>> Polyhedra(QQ, Integer(2)).base_extend(ZZ)
Polyhedra in QQ^2
```

change_ring(*base_ring*, *backend=None*, *ambient_dim=None*)

Return the parent with the new base ring.

INPUT:

- `base_ring`, `backend` – see *Polyhedron()*
- `ambient_dim` – if not `None` change ambient dimension accordingly

EXAMPLES:

```
sage: from sage.geometry.polyhedron.parent import Polyhedra
sage: Polyhedra(ZZ, 3).change_ring(QQ)
Polyhedra in QQ^3
sage: Polyhedra(ZZ, 3).an_element().change_ring(QQ)
```

(continues on next page)

(continued from previous page)

```
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 4 vertices

sage: Polyhedra(RDF, 3).change_ring(QQ).backend()
'cdd'
sage: Polyhedra(QQ, 3).change_ring(ZZ, ambient_dim=4)
Polyhedra in ZZ^4
sage: Polyhedra(QQ, 3, backend='cdd').change_ring(QQ, ambient_dim=4).backend()
'cdd'
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.parent import Polyhedra
>>> Polyhedra(ZZ, Integer(3)).change_ring(QQ)
Polyhedra in QQ^3
>>> Polyhedra(ZZ, Integer(3)).an_element().change_ring(QQ)
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 4 vertices

>>> Polyhedra(RDF, Integer(3)).change_ring(QQ).backend()
'cdd'
>>> Polyhedra(QQ, Integer(3)).change_ring(ZZ, ambient_dim=Integer(4))
Polyhedra in ZZ^4
>>> Polyhedra(QQ, Integer(3), backend='cdd').change_ring(QQ, ambient_
↪dim=Integer(4)).backend()
'cdd'
```

empty()

Return the empty polyhedron.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.parent import Polyhedra
sage: P = Polyhedra(QQ, 4)
sage: P.empty()
The empty polyhedron in QQ^4
sage: P.empty().is_empty()
True
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.parent import Polyhedra
>>> P = Polyhedra(QQ, Integer(4))
>>> P.empty()
The empty polyhedron in QQ^4
>>> P.empty().is_empty()
True
```

list()

Return the two polyhedra in ambient dimension 0, raise an error otherwise.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.parent import Polyhedra
sage: P = Polyhedra(QQ, 3)
sage: P.cardinality()
```

(continues on next page)

(continued from previous page)

```
+Infinity

sage: # needs sage.rings.number_field
sage: P = Polyhedra(AA, 0)
sage: P.category()
Category of finite enumerated polyhedral sets over Algebraic Real Field
sage: P.list()
[The empty polyhedron in AA^0,
 A 0-dimensional polyhedron in AA^0 defined as the convex hull of 1 vertex]
sage: P.cardinality()
2
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.parent import Polyhedra
>>> P = Polyhedra(QQ, Integer(3))
>>> P.cardinality()
+Infinity

>>> # needs sage.rings.number_field
>>> P = Polyhedra(AA, Integer(0))
>>> P.category()
Category of finite enumerated polyhedral sets over Algebraic Real Field
>>> P.list()
[The empty polyhedron in AA^0,
 A 0-dimensional polyhedron in AA^0 defined as the convex hull of 1 vertex]
>>> P.cardinality()
2
```

recycle(*polyhedron*)

Recycle the H/V-representation objects of a polyhedron.

This speeds up creation of new polyhedra by reusing objects. After recycling a polyhedron object, it is not in a consistent state any more and neither the polyhedron nor its H/V-representation objects may be used any more.

INPUT:

- `polyhedron` – a polyhedron whose parent is `self`

EXAMPLES:

```
sage: p = Polyhedron([(0,0),(1,0),(0,1)])
sage: p.parent().recycle(p)
```

```
>>> from sage.all import *
>>> p = Polyhedron([(Integer(0),Integer(0)),(Integer(1),Integer(0)),
↪(Integer(0),Integer(1))])
>>> p.parent().recycle(p)
```

some_elements()

Return a list of some elements of the semigroup.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.parent import Polyhedra
sage: Polyhedra(QQ, 4).some_elements()
[A 3-dimensional polyhedron in QQ^4
  defined as the convex hull of 4 vertices,
 A 4-dimensional polyhedron in QQ^4
  defined as the convex hull of 1 vertex and 4 rays,
 A 2-dimensional polyhedron in QQ^4
  defined as the convex hull of 2 vertices and 1 ray,
 The empty polyhedron in QQ^4]
sage: Polyhedra(ZZ, 0).some_elements()
[The empty polyhedron in ZZ^0,
 A 0-dimensional polyhedron in ZZ^0 defined as the convex hull of 1 vertex]
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.parent import Polyhedra
>>> Polyhedra(QQ, Integer(4)).some_elements()
[A 3-dimensional polyhedron in QQ^4
  defined as the convex hull of 4 vertices,
 A 4-dimensional polyhedron in QQ^4
  defined as the convex hull of 1 vertex and 4 rays,
 A 2-dimensional polyhedron in QQ^4
  defined as the convex hull of 2 vertices and 1 ray,
 The empty polyhedron in QQ^4]
>>> Polyhedra(ZZ, Integer(0)).some_elements()
[The empty polyhedron in ZZ^0,
 A 0-dimensional polyhedron in ZZ^0 defined as the convex hull of 1 vertex]
```

universe()

Return the entire ambient space as polyhedron.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.parent import Polyhedra
sage: P = Polyhedra(QQ, 4)
sage: P.universe()
A 4-dimensional polyhedron in QQ^4 defined as
 the convex hull of 1 vertex and 4 lines
sage: P.universe().is_universe()
True
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.parent import Polyhedra
>>> P = Polyhedra(QQ, Integer(4))
>>> P.universe()
A 4-dimensional polyhedron in QQ^4 defined as
 the convex hull of 1 vertex and 4 lines
>>> P.universe().is_universe()
True
```

zero()

Return the polyhedron consisting of the origin, which is the neutral element for Minkowski addition.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.parent import Polyhedra
sage: p = Polyhedra(QQ, 4).zero();  p
A 0-dimensional polyhedron in QQ^4 defined as the convex hull of 1 vertex
sage: p + p == p
True
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.parent import Polyhedra
>>> p = Polyhedra(QQ, Integer(4)).zero();  p
A 0-dimensional polyhedron in QQ^4 defined as the convex hull of 1 vertex
>>> p + p == p
True
```

class sage.geometry.polyhedron.parent.**Polyhedra_field**(*base_ring*, *ambient_dim*, *backend*)

Bases: *Polyhedra_base*

Element

alias of *Polyhedron_field*

class sage.geometry.polyhedron.parent.**Polyhedra_normaliz**(*base_ring*, *ambient_dim*, *backend*)

Bases: *Polyhedra_base*

Element

alias of *Polyhedron_normaliz*

class sage.geometry.polyhedron.parent.**Polyhedra_number_field**(*base_ring*, *ambient_dim*, *backend*)

Bases: *Polyhedra_base*

Element

alias of *Polyhedron_number_field*

class sage.geometry.polyhedron.parent.**Polyhedra_polymake**(*base_ring*, *ambient_dim*, *backend*)

Bases: *Polyhedra_base*

Element

alias of *Polyhedron_polymake*

sage.geometry.polyhedron.parent.**does_backend_handle_base_ring**(*backend*)

Return true, if `backend` can handle `base_ring`.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.parent import does_backend_handle_base_ring
sage: does_backend_handle_base_ring(QQ, 'ppl')
True
sage: does_backend_handle_base_ring(QQ[sqrt(5)], 'ppl')                    #␣
↪needs sage.rings.number_field sage.symbolic
False
sage: does_backend_handle_base_ring(QQ[sqrt(5)], 'field')                  #␣
↪needs sage.rings.number_field sage.symbolic
True
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.parent import does_backend_handle_base_ring
>>> does_backend_handle_base_ring(QQ, 'ppl')
```

(continues on next page)

(continued from previous page)

```
True
>>> does_backend_handle_base_ring(QQ[sqrt(Integer(5))], 'ppl')                    ␣
↪     # needs sage.rings.number_field sage.symbolic
False
>>> does_backend_handle_base_ring(QQ[sqrt(Integer(5))], 'field')                  ␣
↪     # needs sage.rings.number_field sage.symbolic
True
```

2.1.4 H(yperplane) and V(ertex) representation objects for polyhedra

class sage.geometry.polyhedron.representation.**Equation**(*polyhedron_parent*)

Bases: *Hrepresentation*

A linear equation of the polyhedron. That is, the polyhedron is strictly smaller-dimensional than the ambient space, and contained in this hyperplane. Inherits from `Hrepresentation`.

contains(*Vobj*)

Test whether the hyperplane defined by the equation contains the given vertex/ray/line.

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[0,0,0],[1,1,0],[1,2,0]])
sage: v = next(p.vertex_generator())
sage: v
A vertex at (0, 0, 0)
sage: a = next(p.equation_generator())
sage: a
An equation (0, 0, 1) x + 0 == 0
sage: a.contains(v)
True
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices = [[Integer(0),Integer(0),Integer(0)],[Integer(1),
↪Integer(1),Integer(0)],[Integer(1),Integer(2),Integer(0)]])
>>> v = next(p.vertex_generator())
>>> v
A vertex at (0, 0, 0)
>>> a = next(p.equation_generator())
>>> a
An equation (0, 0, 1) x + 0 == 0
>>> a.contains(v)
True
```

interior_contains(*Vobj*)

Test whether the interior of the halfspace (excluding its boundary) defined by the inequality contains the given vertex/ray/line.

> **Note**
>
> Return `False` for any equation.

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[0,0,0],[1,1,0],[1,2,0]])
sage: v = next(p.vertex_generator())
sage: v
A vertex at (0, 0, 0)
sage: a = next(p.equation_generator())
sage: a
An equation (0, 0, 1) x + 0 == 0
sage: a.interior_contains(v)
False
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices = [[Integer(0),Integer(0),Integer(0)],[Integer(1),
↪Integer(1),Integer(0)],[Integer(1),Integer(2),Integer(0)]])
>>> v = next(p.vertex_generator())
>>> v
A vertex at (0, 0, 0)
>>> a = next(p.equation_generator())
>>> a
An equation (0, 0, 1) x + 0 == 0
>>> a.interior_contains(v)
False
```

is_equation()

Test if this object is an equation. By construction, it must be.

type()

Return the type (equation/inequality/vertex/ray/line) as an integer.

OUTPUT:

Integer. One of `PolyhedronRepresentation.INEQUALITY`, `.EQUATION`, `.VERTEX`, `.RAY`, or `.LINE`.

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[0,0,0],[1,1,0],[1,2,0]])
sage: repr_obj = next(p.equation_generator())
sage: repr_obj.type()
1
sage: repr_obj.type() == repr_obj.INEQUALITY
False
sage: repr_obj.type() == repr_obj.EQUATION
True
sage: repr_obj.type() == repr_obj.VERTEX
False
sage: repr_obj.type() == repr_obj.RAY
False
sage: repr_obj.type() == repr_obj.LINE
False
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices = [[Integer(0),Integer(0),Integer(0)],[Integer(1),
↪Integer(1),Integer(0)],[Integer(1),Integer(2),Integer(0)]])
>>> repr_obj = next(p.equation_generator())
>>> repr_obj.type()
```

(continues on next page)

(continued from previous page)

```
1
>>> repr_obj.type() == repr_obj.INEQUALITY
False
>>> repr_obj.type() == repr_obj.EQUATION
True
>>> repr_obj.type() == repr_obj.VERTEX
False
>>> repr_obj.type() == repr_obj.RAY
False
>>> repr_obj.type() == repr_obj.LINE
False
```

class sage.geometry.polyhedron.representation.**Hrepresentation**(*polyhedron_parent*)

Bases: *PolyhedronRepresentation*

The internal base class for H-representation objects of a polyhedron. Inherits from `PolyhedronRepresentation`.

A()

Return the coefficient vector A in $A\vec{x}+b$.

EXAMPLES:

```
sage: p = Polyhedron(ieqs = [[0,1,0],[0,0,1],[1,-1,0,],[1,0,-1]])
sage: pH = p.Hrepresentation(2)
sage: pH.A()
(1, 0)
```

```
>>> from sage.all import *
>>> p = Polyhedron(ieqs = [[Integer(0),Integer(1),Integer(0)],[Integer(0),
↪Integer(0),Integer(1)],[Integer(1),-Integer(1),Integer(0),],[Integer(1),
↪Integer(0),-Integer(1)]])
>>> pH = p.Hrepresentation(Integer(2))
>>> pH.A()
(1, 0)
```

adjacent()

Alias for neighbors().

b()

Return the constant b in $A\vec{x}+b$.

EXAMPLES:

```
sage: p = Polyhedron(ieqs = [[0,1,0],[0,0,1],[1,-1,0,],[1,0,-1]])
sage: pH = p.Hrepresentation(2)
sage: pH.b()
0
```

```
>>> from sage.all import *
>>> p = Polyhedron(ieqs = [[Integer(0),Integer(1),Integer(0)],[Integer(0),
↪Integer(0),Integer(1)],[Integer(1),-Integer(1),Integer(0),],[Integer(1),
↪Integer(0),-Integer(1)]])
```

(continues on next page)

(continued from previous page)

```
>>> pH = p.Hrepresentation(Integer(2))
>>> pH.b()
0
```

eval(*Vobj*)

Evaluate the left hand side $A\vec{x} + b$ on the given vertex/ray/line.

Note

- Evaluating on a vertex returns $A\vec{x} + b$
- Evaluating on a ray returns $A\vec{r}$. Only the sign or whether it is zero is meaningful.
- Evaluating on a line returns $A\vec{l}$. Only whether it is zero or not is meaningful.

EXAMPLES:

```
sage: triangle = Polyhedron(vertices=[[1,0],[0,1],[-1,-1]])
sage: ineq = next(triangle.inequality_generator())
sage: ineq
An inequality (2, -1) x + 1 >= 0
sage: [ ineq.eval(v) for v in triangle.vertex_generator() ]
[0, 0, 3]
sage: [ ineq * v for v in triangle.vertex_generator() ]
[0, 0, 3]
```

```
>>> from sage.all import *
>>> triangle = Polyhedron(vertices=[[Integer(1),Integer(0)],[Integer(0),
↪Integer(1)],[-Integer(1),-Integer(1)]])
>>> ineq = next(triangle.inequality_generator())
>>> ineq
An inequality (2, -1) x + 1 >= 0
>>> [ ineq.eval(v) for v in triangle.vertex_generator() ]
[0, 0, 3]
>>> [ ineq * v for v in triangle.vertex_generator() ]
[0, 0, 3]
```

If you pass a vector, it is assumed to be the coordinate vector of a point:

```
sage: ineq.eval( vector(ZZ, [3,2]) )
5
```

```
>>> from sage.all import *
>>> ineq.eval( vector(ZZ, [Integer(3),Integer(2)]) )
5
```

incident()

Return a generator for the incident H-representation objects, that is, the vertices/rays/lines satisfying the (in)equality.

EXAMPLES:

```
sage: triangle = Polyhedron(vertices=[[1,0],[0,1],[-1,-1]])
sage: ineq = next(triangle.inequality_generator())
sage: ineq
An inequality (2, -1) x + 1 >= 0
sage: [ v for v in ineq.incident()]
[A vertex at (-1, -1), A vertex at (0, 1)]
sage: p = Polyhedron(vertices=[[0,0,0],[0,1,0],[0,0,1]], rays=[[1,-1,-1]])
sage: ineq = p.Hrepresentation(2)
sage: ineq
An inequality (1, 0, 1) x + 0 >= 0
sage: [ x for x in ineq.incident() ]
[A vertex at (0, 0, 0),
 A vertex at (0, 1, 0),
 A ray in the direction (1, -1, -1)]
```

```
>>> from sage.all import *
>>> triangle = Polyhedron(vertices=[[Integer(1),Integer(0)],[Integer(0),
↪Integer(1)],[-Integer(1),-Integer(1)]])
>>> ineq = next(triangle.inequality_generator())
>>> ineq
An inequality (2, -1) x + 1 >= 0
>>> [ v for v in ineq.incident()]
[A vertex at (-1, -1), A vertex at (0, 1)]
>>> p = Polyhedron(vertices=[[Integer(0),Integer(0),Integer(0)],[Integer(0),
↪Integer(1),Integer(0)],[Integer(0),Integer(0),Integer(1)]],␣
↪rays=[[Integer(1),-Integer(1),-Integer(1)]])
>>> ineq = p.Hrepresentation(Integer(2))
>>> ineq
An inequality (1, 0, 1) x + 0 >= 0
>>> [ x for x in ineq.incident() ]
[A vertex at (0, 0, 0),
 A vertex at (0, 1, 0),
 A ray in the direction (1, -1, -1)]
```

is_H()

Return `True` if the object is part of a H-representation (inequality or equation).

EXAMPLES:

```
sage: p = Polyhedron(ieqs = [[0,1,0],[0,0,1],[1,-1,0,],[1,0,-1]])
sage: pH = p.Hrepresentation(0)
sage: pH.is_H()
True
```

```
>>> from sage.all import *
>>> p = Polyhedron(ieqs = [[Integer(0),Integer(1),Integer(0)],[Integer(0),
↪Integer(0),Integer(1)],[Integer(1),-Integer(1),Integer(0),],[Integer(1),
↪Integer(0),-Integer(1)]])
>>> pH = p.Hrepresentation(Integer(0))
>>> pH.is_H()
True
```

is_equation()

Return `True` if the object is an equation of the H-representation.

EXAMPLES:

```
sage: p = Polyhedron(ieqs = [[0,1,0],[0,0,1],[1,-1,0,],[1,0,-1]], eqns = [[1,
↪1,-1]])
sage: pH = p.Hrepresentation(0)
sage: pH.is_equation()
True
```

```
>>> from sage.all import *
>>> p = Polyhedron(ieqs = [[Integer(0),Integer(1),Integer(0)],[Integer(0),
↪Integer(0),Integer(1)],[Integer(1),-Integer(1),Integer(0),],[Integer(1),
↪Integer(0),-Integer(1)]], eqns = [[Integer(1),Integer(1),-Integer(1)]])
>>> pH = p.Hrepresentation(Integer(0))
>>> pH.is_equation()
True
```

is_incident(*Vobj*)

Return whether the incidence matrix element (Vobj,self) == 1.

EXAMPLES:

```
sage: p = Polyhedron(ieqs = [[0,0,0,1],[0,0,1,0,],[0,1,0,0],
....:     [1,-1,0,0],[1,0,-1,0,],[1,0,0,-1]])
sage: pH = p.Hrepresentation(0)
sage: pH.is_incident(p.Vrepresentation(1))
True
sage: pH.is_incident(p.Vrepresentation(5))
False
```

```
>>> from sage.all import *
>>> p = Polyhedron(ieqs = [[Integer(0),Integer(0),Integer(0),Integer(1)],
↪[Integer(0),Integer(0),Integer(1),Integer(0),],[Integer(0),Integer(1),
↪Integer(0),Integer(0)],
...     [Integer(1),-Integer(1),Integer(0),Integer(0)],[Integer(1),Integer(0),
↪-Integer(1),Integer(0),],[Integer(1),Integer(0),Integer(0),-Integer(1)]])
>>> pH = p.Hrepresentation(Integer(0))
>>> pH.is_incident(p.Vrepresentation(Integer(1)))
True
>>> pH.is_incident(p.Vrepresentation(Integer(5)))
False
```

is_inequality()

Return `True` if the object is an inequality of the H-representation.

EXAMPLES:

```
sage: p = Polyhedron(ieqs = [[0,1,0],[0,0,1],[1,-1,0,],[1,0,-1]])
sage: pH = p.Hrepresentation(0)
sage: pH.is_inequality()
True
```

```
>>> from sage.all import *
>>> p = Polyhedron(ieqs = [[Integer(0),Integer(1),Integer(0)],[Integer(0),
↪Integer(0),Integer(1)],[Integer(1),-Integer(1),Integer(0),],[Integer(1),
↪Integer(0),-Integer(1)]])
>>> pH = p.Hrepresentation(Integer(0))
>>> pH.is_inequality()
True
```

neighbors()

Iterate over the adjacent facets (i.e. inequalities).

Only defined for inequalities.

EXAMPLES:

```
sage: p = Polyhedron(ieqs = [[0,0,0,1],[0,0,1,0,],[0,1,0,0],
....:                        [1,-1,0,0],[1,0,-1,0,],[1,0,0,-1]])
sage: pH = p.Hrepresentation(0)
sage: a = list(pH.neighbors())
sage: a[0]
An inequality (0, -1, 0) x + 1 >= 0
sage: list(a[0])
[1, 0, -1, 0]
```

```
>>> from sage.all import *
>>> p = Polyhedron(ieqs = [[Integer(0),Integer(0),Integer(0),Integer(1)],
↪[Integer(0),Integer(0),Integer(1),Integer(0),],[Integer(0),Integer(1),
↪Integer(0),Integer(0)],
...                        [Integer(1),-Integer(1),Integer(0),Integer(0)],
↪[Integer(1),Integer(0),-Integer(1),Integer(0),],[Integer(1),Integer(0),
↪Integer(0),-Integer(1)]])
>>> pH = p.Hrepresentation(Integer(0))
>>> a = list(pH.neighbors())
>>> a[Integer(0)]
An inequality (0, -1, 0) x + 1 >= 0
>>> list(a[Integer(0)])
[1, 0, -1, 0]
```

repr_pretty(***kwds*)

Return a pretty representation of this equality/inequality.

INPUT:

- `prefix` – string
- `indices` – tuple or other iterable
- `latex` – boolean

OUTPUT: string

EXAMPLES:

```
sage: P = Polyhedron(ieqs=[(0, 1, 0, 0), (1, 2, 1, 0)],
....:                eqns=[(1, -1, -1, 1)])
sage: for h in P.Hrepresentation():
```

(continues on next page)

(continued from previous page)

```
....:     print(h.repr_pretty())
x0 + x1 - x2 == 1
x0 >= 0
2*x0 + x1 >= -1
```

```
>>> from sage.all import *
>>> P = Polyhedron(ieqs=[(Integer(0), Integer(1), Integer(0), Integer(0)),
↪(Integer(1), Integer(2), Integer(1), Integer(0))],
...                eqns=[(Integer(1), -Integer(1), -Integer(1), Integer(1))])
>>> for h in P.Hrepresentation():
...     print(h.repr_pretty())
x0 + x1 - x2 == 1
x0 >= 0
2*x0 + x1 >= -1
```

class sage.geometry.polyhedron.representation.**Inequality**(*polyhedron_parent*)

Bases: *Hrepresentation*

A linear inequality (supporting hyperplane) of the polyhedron. Inherits from `Hrepresentation`.

contains(*Vobj*)

Test whether the halfspace (including its boundary) defined by the inequality contains the given vertex/ray/line.

EXAMPLES:

```
sage: p = polytopes.cross_polytope(3)
sage: i1 = next(p.inequality_generator())
sage: [i1.contains(q) for q in p.vertex_generator()]
[True, True, True, True, True, True]
sage: p2 = 3*polytopes.hypercube(3)
sage: [i1.contains(q) for q in p2.vertex_generator()]
[True, True, False, True, False, True, False, False]
```

```
>>> from sage.all import *
>>> p = polytopes.cross_polytope(Integer(3))
>>> i1 = next(p.inequality_generator())
>>> [i1.contains(q) for q in p.vertex_generator()]
[True, True, True, True, True, True]
>>> p2 = Integer(3)*polytopes.hypercube(Integer(3))
>>> [i1.contains(q) for q in p2.vertex_generator()]
[True, True, False, True, False, True, False, False]
```

interior_contains(*Vobj*)

Test whether the interior of the halfspace (excluding its boundary) defined by the inequality contains the given vertex/ray/line.

EXAMPLES:

```
sage: p = polytopes.cross_polytope(3)
sage: i1 = next(p.inequality_generator())
sage: [i1.interior_contains(q) for q in p.vertex_generator()]
[False, True, True, False, False, True]
sage: p2 = 3*polytopes.hypercube(3)
```

(continues on next page)

(continued from previous page)

```
sage: [i1.interior_contains(q) for q in p2.vertex_generator()]
[True, True, False, True, False, True, False, False]
```

```
>>> from sage.all import *
>>> p = polytopes.cross_polytope(Integer(3))
>>> i1 = next(p.inequality_generator())
>>> [i1.interior_contains(q) for q in p.vertex_generator()]
[False, True, True, False, False, True]
>>> p2 = Integer(3)*polytopes.hypercube(Integer(3))
>>> [i1.interior_contains(q) for q in p2.vertex_generator()]
[True, True, False, True, False, True, False, False]
```

If you pass a vector, it is assumed to be the coordinate vector of a point:

```
sage: P = Polyhedron(vertices=[[1,1],[1,-1],[-1,1],[-1,-1]])
sage: p = vector(ZZ, [1,0] )
sage: [ ieq.interior_contains(p) for ieq in P.inequality_generator() ]
[True, True, False, True]
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[[Integer(1),Integer(1)],[Integer(1),-Integer(1)],
↪[-Integer(1),Integer(1)],[-Integer(1),-Integer(1)]])
>>> p = vector(ZZ, [Integer(1),Integer(0)] )
>>> [ ieq.interior_contains(p) for ieq in P.inequality_generator() ]
[True, True, False, True]
```

is_facet_defining_inequality(*other*)

Check if `self` defines a facet of `other`.

INPUT:

- `other` – a polyhedron

> **See also**
>
> `slack_matrix()` `incidence_matrix()`

EXAMPLES:

```
sage: P = Polyhedron(vertices=[[0,0,0],[0,1,0]], rays=[[1,0,0]])
sage: P.inequalities()
(An inequality (1, 0, 0) x + 0 >= 0,
 An inequality (0, 1, 0) x + 0 >= 0,
 An inequality (0, -1, 0) x + 1 >= 0)
sage: Q = Polyhedron(ieqs=[[0,1,0,0]])
sage: Q.inequalities()[0].is_facet_defining_inequality(P)
True
sage: Q = Polyhedron(ieqs=[[0,2,0,3]])
sage: Q.inequalities()[0].is_facet_defining_inequality(P)
True
sage: Q = Polyhedron(ieqs=[[0,AA(2).sqrt(),0,3]])                          #␣
↪needs sage.rings.number_field
```

(continues on next page)

(continued from previous page)

```
sage: Q.inequalities()[0].is_facet_defining_inequality(P)
True
sage: Q = Polyhedron(ieqs=[[1,1,0,0]])
sage: Q.inequalities()[0].is_facet_defining_inequality(P)
False
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[[Integer(0),Integer(0),Integer(0)],[Integer(0),
↪Integer(1),Integer(0)]], rays=[[Integer(1),Integer(0),Integer(0)]])
>>> P.inequalities()
(An inequality (1, 0, 0) x + 0 >= 0,
 An inequality (0, 1, 0) x + 0 >= 0,
 An inequality (0, -1, 0) x + 1 >= 0)
>>> Q = Polyhedron(ieqs=[[Integer(0),Integer(1),Integer(0),Integer(0)]])
>>> Q.inequalities()[Integer(0)].is_facet_defining_inequality(P)
True
>>> Q = Polyhedron(ieqs=[[Integer(0),Integer(2),Integer(0),Integer(3)]])
>>> Q.inequalities()[Integer(0)].is_facet_defining_inequality(P)
True
>>> Q = Polyhedron(ieqs=[[Integer(0),AA(Integer(2)).sqrt(),Integer(0),
↪Integer(3)]])                               # needs sage.rings.number_field
>>> Q.inequalities()[Integer(0)].is_facet_defining_inequality(P)
True
>>> Q = Polyhedron(ieqs=[[Integer(1),Integer(1),Integer(0),Integer(0)]])
>>> Q.inequalities()[Integer(0)].is_facet_defining_inequality(P)
False
```

```
sage: P = Polyhedron(vertices=[[0,0,0],[0,1,0]], lines=[[1,0,0]])
sage: P.inequalities()
(An inequality (0, 1, 0) x + 0 >= 0, An inequality (0, -1, 0) x + 1 >= 0)
sage: Q = Polyhedron(ieqs=[[0,1,0,0]])
sage: Q.inequalities()[0].is_facet_defining_inequality(P)
False
sage: Q = Polyhedron(ieqs=[[0,-1,0,0]])
sage: Q.inequalities()[0].is_facet_defining_inequality(P)
False
sage: Q = Polyhedron(ieqs=[[0,0,1,3]])
sage: Q.inequalities()[0].is_facet_defining_inequality(P)
True
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[[Integer(0),Integer(0),Integer(0)],[Integer(0),
↪Integer(1),Integer(0)]], lines=[[Integer(1),Integer(0),Integer(0)]])
>>> P.inequalities()
(An inequality (0, 1, 0) x + 0 >= 0, An inequality (0, -1, 0) x + 1 >= 0)
>>> Q = Polyhedron(ieqs=[[Integer(0),Integer(1),Integer(0),Integer(0)]])
>>> Q.inequalities()[Integer(0)].is_facet_defining_inequality(P)
False
>>> Q = Polyhedron(ieqs=[[Integer(0),-Integer(1),Integer(0),Integer(0)]])
>>> Q.inequalities()[Integer(0)].is_facet_defining_inequality(P)
False
```

(continues on next page)

(continued from previous page)

```
>>> Q = Polyhedron(ieqs=[[Integer(0),Integer(0),Integer(1),Integer(3)]])
>>> Q.inequalities()[Integer(0)].is_facet_defining_inequality(P)
True
```

is_inequality()

Return `True` since this is, by construction, an inequality.

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[0,0,0],[1,1,0],[1,2,0]])
sage: a = next(p.inequality_generator())
sage: a.is_inequality()
True
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices = [[Integer(0),Integer(0),Integer(0)],[Integer(1),
↪Integer(1),Integer(0)],[Integer(1),Integer(2),Integer(0)]])
>>> a = next(p.inequality_generator())
>>> a.is_inequality()
True
```

outer_normal()

Return the outer normal vector of `self`.

OUTPUT: the normal vector directed away from the interior of the polyhedron

EXAMPLES:

```
sage: p = Polyhedron(vertices=[[0,0,0],[1,1,0],[1,2,0]])
sage: a = next(p.inequality_generator())
sage: a.outer_normal()
(1, -1, 0)
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices=[[Integer(0),Integer(0),Integer(0)],[Integer(1),
↪Integer(1),Integer(0)],[Integer(1),Integer(2),Integer(0)]])
>>> a = next(p.inequality_generator())
>>> a.outer_normal()
(1, -1, 0)
```

type()

Return the type (equation/inequality/vertex/ray/line) as an integer.

OUTPUT:

Integer. One of `PolyhedronRepresentation.INEQUALITY`, `.EQUATION`, `.VERTEX`, `.RAY`, or `.LINE`.

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[0,0,0],[1,1,0],[1,2,0]])
sage: repr_obj = next(p.inequality_generator())
sage: repr_obj.type()
0
sage: repr_obj.type() == repr_obj.INEQUALITY
```

(continues on next page)

(continued from previous page)

```
True
sage: repr_obj.type() == repr_obj.EQUATION
False
sage: repr_obj.type() == repr_obj.VERTEX
False
sage: repr_obj.type() == repr_obj.RAY
False
sage: repr_obj.type() == repr_obj.LINE
False
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices = [[Integer(0),Integer(0),Integer(0)],[Integer(1),
↪Integer(1),Integer(0)],[Integer(1),Integer(2),Integer(0)]])
>>> repr_obj = next(p.inequality_generator())
>>> repr_obj.type()
0
>>> repr_obj.type() == repr_obj.INEQUALITY
True
>>> repr_obj.type() == repr_obj.EQUATION
False
>>> repr_obj.type() == repr_obj.VERTEX
False
>>> repr_obj.type() == repr_obj.RAY
False
>>> repr_obj.type() == repr_obj.LINE
False
```

class sage.geometry.polyhedron.representation.**Line**(*polyhedron_parent*)

Bases: *Vrepresentation*

A line (Minkowski summand $\simeq \mathbf{R}$) of the polyhedron. Inherits from Vrepresentation.

evaluated_on(*Hobj*)

Return $A\vec{\ell}$.

EXAMPLES:

```
sage: p = Polyhedron(ieqs = [[1, 0, 0, 1],[1,1,0,0]])
sage: a = next(p.line_generator())
sage: h = next(p.inequality_generator())
sage: a.evaluated_on(h)
0
```

```
>>> from sage.all import *
>>> p = Polyhedron(ieqs = [[Integer(1), Integer(0), Integer(0), Integer(1)],
↪[Integer(1),Integer(1),Integer(0),Integer(0)]])
>>> a = next(p.line_generator())
>>> h = next(p.inequality_generator())
>>> a.evaluated_on(h)
0
```

homogeneous_vector(*base_ring=None*)

Return homogeneous coordinates for this line.

Since a line is given by a direction, this is the vector with a 0 appended.

INPUT:

- `base_ring` – the base ring of the vector

EXAMPLES:

```
sage: P = Polyhedron(vertices=[(2,0)], rays=[(1,0)], lines=[(3,2)])
sage: P.lines()[0].homogeneous_vector()
(3, 2, 0)
sage: P.lines()[0].homogeneous_vector(RDF)
(3.0, 2.0, 0.0)
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[(Integer(2),Integer(0))], rays=[(Integer(1),
↪Integer(0))], lines=[(Integer(3),Integer(2))])
>>> P.lines()[Integer(0)].homogeneous_vector()
(3, 2, 0)
>>> P.lines()[Integer(0)].homogeneous_vector(RDF)
(3.0, 2.0, 0.0)
```

is_line()

Test if the object is a line. By construction it must be.

type()

Return the type (equation/inequality/vertex/ray/line) as an integer.

OUTPUT:

Integer. One of `PolyhedronRepresentation.INEQUALITY`, `.EQUATION`, `.VERTEX`, `.RAY`, or `.LINE`.

EXAMPLES:

```
sage: p = Polyhedron(ieqs = [[1, 0, 0, 1],[1,1,0,0]])
sage: repr_obj = next(p.line_generator())
sage: repr_obj.type()
4
sage: repr_obj.type() == repr_obj.INEQUALITY
False
sage: repr_obj.type() == repr_obj.EQUATION
False
sage: repr_obj.type() == repr_obj.VERTEX
False
sage: repr_obj.type() == repr_obj.RAY
False
sage: repr_obj.type() == repr_obj.LINE
True
```

```
>>> from sage.all import *
>>> p = Polyhedron(ieqs = [[Integer(1), Integer(0), Integer(0), Integer(1)],
↪[Integer(1),Integer(1),Integer(0),Integer(0)]])
>>> repr_obj = next(p.line_generator())
>>> repr_obj.type()
4
>>> repr_obj.type() == repr_obj.INEQUALITY
```

(continues on next page)

(continued from previous page)

```
False
>>> repr_obj.type() == repr_obj.EQUATION
False
>>> repr_obj.type() == repr_obj.VERTEX
False
>>> repr_obj.type() == repr_obj.RAY
False
>>> repr_obj.type() == repr_obj.LINE
True
```

class sage.geometry.polyhedron.representation.**PolyhedronRepresentation**

Bases: `SageObject`

The internal base class for all representation objects of `Polyhedron` (vertices/rays/lines and inequalities/equations)

Note: You should not (and cannot) instantiate it yourself. You can only obtain them from a Polyhedron() class.

EQUATION = 1

INEQUALITY = 0

LINE = 4

RAY = 3

VERTEX = 2

count(*i*)

Count the number of occurrences of `i` in the coordinates.

INPUT:

- `i` – anything

OUTPUT: integer; the number of occurrences of `i` in the coordinates

EXAMPLES:

```
sage: p = Polyhedron(vertices=[(0,1,1,2,1)])
sage: v = p.Vrepresentation(0); v
A vertex at (0, 1, 1, 2, 1)
sage: v.count(1)
3
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices=[(Integer(0),Integer(1),Integer(1),Integer(2),
↪Integer(1))])
>>> v = p.Vrepresentation(Integer(0)); v
A vertex at (0, 1, 1, 2, 1)
>>> v.count(Integer(1))
3
```

index()

Return an arbitrary but fixed number according to the internal storage order.

> **Note**
>
> H-representation and V-representation objects are enumerated independently. That is, amongst all vertices/rays/lines there will be one with `index()==0`, and amongst all inequalities/equations there will be one with `index()==0`, unless the polyhedron is empty or spans the whole space.

EXAMPLES:

```
sage: s = Polyhedron(vertices=[[1],[-1]])
sage: first_vertex = next(s.vertex_generator())
sage: first_vertex.index()
0
sage: first_vertex == s.Vrepresentation(0)
True
```

```
>>> from sage.all import *
>>> s = Polyhedron(vertices=[[Integer(1)],[-Integer(1)]])
>>> first_vertex = next(s.vertex_generator())
>>> first_vertex.index()
0
>>> first_vertex == s.Vrepresentation(Integer(0))
True
```

polyhedron()

Return the underlying polyhedron.

vector(*base_ring=None*)

Return the vector representation of the H/V-representation object.

INPUT:

- `base_ring` – the base ring of the vector

OUTPUT:

For a V-representation object, a vector of length `ambient_dim()`. For a H-representation object, a vector of length `ambient_dim()` + 1.

EXAMPLES:

```
sage: s = polytopes.cuboctahedron()
sage: v = next(s.vertex_generator())
sage: v
A vertex at (-1, -1, 0)
sage: v.vector()
(-1, -1, 0)
sage: v()
(-1, -1, 0)
sage: type(v())
<class 'sage.modules.vector_integer_dense.Vector_integer_dense'>
```

```
>>> from sage.all import *
>>> s = polytopes.cuboctahedron()
>>> v = next(s.vertex_generator())
>>> v
A vertex at (-1, -1, 0)
>>> v.vector()
(-1, -1, 0)
>>> v()
(-1, -1, 0)
>>> type(v())
<class 'sage.modules.vector_integer_dense.Vector_integer_dense'>
```

Conversion to a different base ring can be forced with the optional argument:

```
sage: v.vector(RDF)
(-1.0, -1.0, 0.0)
sage: vector(RDF, v)
(-1.0, -1.0, 0.0)
```

```
>>> from sage.all import *
>>> v.vector(RDF)
(-1.0, -1.0, 0.0)
>>> vector(RDF, v)
(-1.0, -1.0, 0.0)
```

class sage.geometry.polyhedron.representation.**Ray**(*polyhedron_parent*)

Bases: *Vrepresentation*

A ray of the polyhedron. Inherits from Vrepresentation.

evaluated_on(*Hobj*)

Return $A\vec{r}$.

EXAMPLES:

```
sage: p = Polyhedron(ieqs = [[0,0,1],[0,1,0],[1,-1,0]])
sage: a = next(p.ray_generator())
sage: h = next(p.inequality_generator())
sage: a.evaluated_on(h)
0
```

```
>>> from sage.all import *
>>> p = Polyhedron(ieqs = [[Integer(0),Integer(0),Integer(1)],[Integer(0),
↪Integer(1),Integer(0)],[Integer(1),-Integer(1),Integer(0)]])
>>> a = next(p.ray_generator())
>>> h = next(p.inequality_generator())
>>> a.evaluated_on(h)
0
```

homogeneous_vector(*base_ring=None*)

Return homogeneous coordinates for this ray.

Since a ray is given by a direction, this is the vector with a 0 appended.

INPUT:

- `base_ring` – the base ring of the vector

EXAMPLES:

```
sage: P = Polyhedron(vertices=[(2,0)], rays=[(1,0)], lines=[(3,2)])
sage: P.rays()[0].homogeneous_vector()
(1, 0, 0)
sage: P.rays()[0].homogeneous_vector(RDF)
(1.0, 0.0, 0.0)
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[(Integer(2),Integer(0))], rays=[(Integer(1),
↪Integer(0))], lines=[(Integer(3),Integer(2))])
>>> P.rays()[Integer(0)].homogeneous_vector()
(1, 0, 0)
>>> P.rays()[Integer(0)].homogeneous_vector(RDF)
(1.0, 0.0, 0.0)
```

is_ray()

Test if this object is a ray. Always `True` by construction.

EXAMPLES:

```
sage: p = Polyhedron(ieqs = [[0,0,1],[0,1,0],[1,-1,0]])
sage: a = next(p.ray_generator())
sage: a.is_ray()
True
```

```
>>> from sage.all import *
>>> p = Polyhedron(ieqs = [[Integer(0),Integer(0),Integer(1)],[Integer(0),
↪Integer(1),Integer(0)],[Integer(1),-Integer(1),Integer(0)]])
>>> a = next(p.ray_generator())
>>> a.is_ray()
True
```

type()

Return the type (equation/inequality/vertex/ray/line) as an integer.

OUTPUT:

Integer. One of `PolyhedronRepresentation.INEQUALITY`, `.EQUATION`, `.VERTEX`, `.RAY`, or `.LINE`.

EXAMPLES:

```
sage: p = Polyhedron(ieqs = [[0,0,1],[0,1,0],[1,-1,0]])
sage: repr_obj = next(p.ray_generator())
sage: repr_obj.type()
3
sage: repr_obj.type() == repr_obj.INEQUALITY
False
sage: repr_obj.type() == repr_obj.EQUATION
False
sage: repr_obj.type() == repr_obj.VERTEX
False
sage: repr_obj.type() == repr_obj.RAY
```

(continues on next page)

(continued from previous page)

```
True
sage: repr_obj.type() == repr_obj.LINE
False
```

```
>>> from sage.all import *
>>> p = Polyhedron(ieqs = [[Integer(0),Integer(0),Integer(1)],[Integer(0),
↪Integer(1),Integer(0)],[Integer(1),-Integer(1),Integer(0)]])
>>> repr_obj = next(p.ray_generator())
>>> repr_obj.type()
3
>>> repr_obj.type() == repr_obj.INEQUALITY
False
>>> repr_obj.type() == repr_obj.EQUATION
False
>>> repr_obj.type() == repr_obj.VERTEX
False
>>> repr_obj.type() == repr_obj.RAY
True
>>> repr_obj.type() == repr_obj.LINE
False
```

class sage.geometry.polyhedron.representation.**Vertex**(*polyhedron_parent*)

Bases: *Vrepresentation*

A vertex of the polyhedron. Inherits from Vrepresentation.

evaluated_on(*Hobj*)

Return $A\vec{x} + b$.

EXAMPLES:

```
sage: p = polytopes.hypercube(3)
sage: v = next(p.vertex_generator())
sage: h = next(p.inequality_generator())
sage: v
A vertex at (1, -1, -1)
sage: h
An inequality (-1, 0, 0) x + 1 >= 0
sage: v.evaluated_on(h)
0
```

```
>>> from sage.all import *
>>> p = polytopes.hypercube(Integer(3))
>>> v = next(p.vertex_generator())
>>> h = next(p.inequality_generator())
>>> v
A vertex at (1, -1, -1)
>>> h
An inequality (-1, 0, 0) x + 1 >= 0
>>> v.evaluated_on(h)
0
```

homogeneous_vector(*base_ring=None*)

Return homogeneous coordinates for this vertex.

Since a vertex is given by an affine point, this is the vector with a 1 appended.

INPUT:

- `base_ring` – the base ring of the vector

EXAMPLES:

```
sage: P = Polyhedron(vertices=[(2,0)], rays=[(1,0)], lines=[(3,2)])
sage: P.vertices()[0].homogeneous_vector()
(2, 0, 1)
sage: P.vertices()[0].homogeneous_vector(RDF)
(2.0, 0.0, 1.0)
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[(Integer(2),Integer(0))], rays=[(Integer(1),
↪Integer(0))], lines=[(Integer(3),Integer(2))])
>>> P.vertices()[Integer(0)].homogeneous_vector()
(2, 0, 1)
>>> P.vertices()[Integer(0)].homogeneous_vector(RDF)
(2.0, 0.0, 1.0)
```

is_integral()

Return whether the coordinates of the vertex are all integral.

OUTPUT: boolean

EXAMPLES:

```
sage: p = Polyhedron([(1/2,3,5), (0,0,0), (2,3,7)])
sage: [ v.is_integral() for v in p.vertex_generator() ]
[True, False, True]
```

```
>>> from sage.all import *
>>> p = Polyhedron([(Integer(1)/Integer(2),Integer(3),Integer(5)),␣
↪(Integer(0),Integer(0),Integer(0)), (Integer(2),Integer(3),Integer(7))])
>>> [ v.is_integral() for v in p.vertex_generator() ]
[True, False, True]
```

is_vertex()

Test if this object is a vertex. By construction it always is.

EXAMPLES:

```
sage: p = Polyhedron(ieqs = [[0,0,1],[0,1,0],[1,-1,0]])
sage: a = next(p.vertex_generator())
sage: a.is_vertex()
True
```

```
>>> from sage.all import *
>>> p = Polyhedron(ieqs = [[Integer(0),Integer(0),Integer(1)],[Integer(0),
↪Integer(1),Integer(0)],[Integer(1),-Integer(1),Integer(0)]])
>>> a = next(p.vertex_generator())
```

(continues on next page)

(continued from previous page)

```
>>> a.is_vertex()
True
```

type()

Return the type (equation/inequality/vertex/ray/line) as an integer.

OUTPUT:

Integer. One of `PolyhedronRepresentation.INEQUALITY`, `.EQUATION`, `.VERTEX`, `.RAY`, or `.LINE`.

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[0,0,0],[1,1,0],[1,2,0]])
sage: repr_obj = next(p.vertex_generator())
sage: repr_obj.type()
2
sage: repr_obj.type() == repr_obj.INEQUALITY
False
sage: repr_obj.type() == repr_obj.EQUATION
False
sage: repr_obj.type() == repr_obj.VERTEX
True
sage: repr_obj.type() == repr_obj.RAY
False
sage: repr_obj.type() == repr_obj.LINE
False
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices = [[Integer(0),Integer(0),Integer(0)],[Integer(1),
↪Integer(1),Integer(0)],[Integer(1),Integer(2),Integer(0)]])
>>> repr_obj = next(p.vertex_generator())
>>> repr_obj.type()
2
>>> repr_obj.type() == repr_obj.INEQUALITY
False
>>> repr_obj.type() == repr_obj.EQUATION
False
>>> repr_obj.type() == repr_obj.VERTEX
True
>>> repr_obj.type() == repr_obj.RAY
False
>>> repr_obj.type() == repr_obj.LINE
False
```

class sage.geometry.polyhedron.representation.**Vrepresentation**(*polyhedron_parent*)

Bases: *PolyhedronRepresentation*

The base class for V-representation objects of a polyhedron. Inherits from `PolyhedronRepresentation`.

adjacent()

Alias for neighbors().

incident()

Return a generator for the equations/inequalities that are satisfied on the given vertex/ray/line.

EXAMPLES:

```
sage: triangle = Polyhedron(vertices=[[1,0],[0,1],[-1,-1]])
sage: ineq = next(triangle.inequality_generator())
sage: ineq
An inequality (2, -1) x + 1 >= 0
sage: [ v for v in ineq.incident()]
[A vertex at (-1, -1), A vertex at (0, 1)]
sage: p = Polyhedron(vertices=[[0,0,0],[0,1,0],[0,0,1]], rays=[[1,-1,-1]])
sage: ineq = p.Hrepresentation(2)
sage: ineq
An inequality (1, 0, 1) x + 0 >= 0
sage: [ x for x in ineq.incident() ]
[A vertex at (0, 0, 0),
 A vertex at (0, 1, 0),
 A ray in the direction (1, -1, -1)]
```

```
>>> from sage.all import *
>>> triangle = Polyhedron(vertices=[[Integer(1),Integer(0)],[Integer(0),
↪Integer(1)],[-Integer(1),-Integer(1)]])
>>> ineq = next(triangle.inequality_generator())
>>> ineq
An inequality (2, -1) x + 1 >= 0
>>> [ v for v in ineq.incident()]
[A vertex at (-1, -1), A vertex at (0, 1)]
>>> p = Polyhedron(vertices=[[Integer(0),Integer(0),Integer(0)],[Integer(0),
↪Integer(1),Integer(0)],[Integer(0),Integer(0),Integer(1)]],
↪rays=[[Integer(1),-Integer(1),-Integer(1)]])
>>> ineq = p.Hrepresentation(Integer(2))
>>> ineq
An inequality (1, 0, 1) x + 0 >= 0
>>> [ x for x in ineq.incident() ]
[A vertex at (0, 0, 0),
 A vertex at (0, 1, 0),
 A ray in the direction (1, -1, -1)]
```

is_V()

Return `True` if the object is part of a V-representation (a vertex, ray, or line).

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[0,0],[1,0],[0,3],[1,3]])
sage: v = next(p.vertex_generator())
sage: v.is_V()
True
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices = [[Integer(0),Integer(0)],[Integer(1),
↪Integer(0)],[Integer(0),Integer(3)],[Integer(1),Integer(3)]])
>>> v = next(p.vertex_generator())
>>> v.is_V()
True
```

is_incident(*Hobj*)

Return whether the incidence matrix element (self,Hobj) == 1.

EXAMPLES:

```
sage: p = polytopes.hypercube(3)
sage: h1 = next(p.inequality_generator())
sage: h1
An inequality (-1, 0, 0) x + 1 >= 0
sage: v1 = next(p.vertex_generator())
sage: v1
A vertex at (1, -1, -1)
sage: v1.is_incident(h1)
True
```

```
>>> from sage.all import *
>>> p = polytopes.hypercube(Integer(3))
>>> h1 = next(p.inequality_generator())
>>> h1
An inequality (-1, 0, 0) x + 1 >= 0
>>> v1 = next(p.vertex_generator())
>>> v1
A vertex at (1, -1, -1)
>>> v1.is_incident(h1)
True
```

is_line()

Return `True` if the object is a line of the V-representation. This method is over-ridden by the corresponding method in the derived class Line.

EXAMPLES:

```
sage: p = Polyhedron(ieqs = [[1, 0, 0, 0, 1], [1, 1, 0, 0, 0], [1, 0, 1, 0,
↪0]])
sage: line1 = next(p.line_generator())
sage: line1.is_line()
True
sage: v1 = next(p.vertex_generator())
sage: v1.is_line()
False
```

```
>>> from sage.all import *
>>> p = Polyhedron(ieqs = [[Integer(1), Integer(0), Integer(0), Integer(0),
↪Integer(1)], [Integer(1), Integer(1), Integer(0), Integer(0), Integer(0)],
↪[Integer(1), Integer(0), Integer(1), Integer(0), Integer(0)]])
>>> line1 = next(p.line_generator())
>>> line1.is_line()
True
>>> v1 = next(p.vertex_generator())
>>> v1.is_line()
False
```

is_ray()

Return `True` if the object is a ray of the V-representation. This method is over-ridden by the corresponding method in the derived class Ray.

EXAMPLES:

```
sage: p = Polyhedron(ieqs = [[1, 0, 0, 0, 1], [1, 1, 0, 0, 0], [1, 0, 1, 0,␣
↪0]])
sage: r1 = next(p.ray_generator())
sage: r1.is_ray()
True
sage: v1 = next(p.vertex_generator())
sage: v1
A vertex at (-1, -1, 0, -1)
sage: v1.is_ray()
False
```

```
>>> from sage.all import *
>>> p = Polyhedron(ieqs = [[Integer(1), Integer(0), Integer(0), Integer(0),␣
↪Integer(1)], [Integer(1), Integer(1), Integer(0), Integer(0), Integer(0)],␣
↪[Integer(1), Integer(0), Integer(1), Integer(0), Integer(0)]])
>>> r1 = next(p.ray_generator())
>>> r1.is_ray()
True
>>> v1 = next(p.vertex_generator())
>>> v1
A vertex at (-1, -1, 0, -1)
>>> v1.is_ray()
False
```

is_vertex()

Return `True` if the object is a vertex of the V-representation. This method is over-ridden by the corresponding method in the derived class Vertex.

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[0,0],[1,0],[0,3],[1,3]])
sage: v = next(p.vertex_generator())
sage: v.is_vertex()
True
sage: p = Polyhedron(ieqs = [[1, 0, 0, 0, 1], [1, 1, 0, 0, 0], [1, 0, 1, 0,␣
↪0]])
sage: r1 = next(p.ray_generator())
sage: r1.is_vertex()
False
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices = [[Integer(0),Integer(0)],[Integer(1),
↪Integer(0)],[Integer(0),Integer(3)],[Integer(1),Integer(3)]])
>>> v = next(p.vertex_generator())
>>> v.is_vertex()
True
>>> p = Polyhedron(ieqs = [[Integer(1), Integer(0), Integer(0), Integer(0),␣
↪Integer(1)], [Integer(1), Integer(1), Integer(0), Integer(0), Integer(0)],␣
↪[Integer(1), Integer(0), Integer(1), Integer(0), Integer(0)]])
>>> r1 = next(p.ray_generator())
>>> r1.is_vertex()
False
```

neighbors()

Return a generator for the adjacent vertices/rays/lines.

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[0,0],[1,0],[0,3],[1,4]])
sage: v = next(p.vertex_generator())
sage: next(v.neighbors())
A vertex at (0, 3)
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices = [[Integer(0),Integer(0)],[Integer(1),
↪Integer(0)],[Integer(0),Integer(3)],[Integer(1),Integer(4)]])
>>> v = next(p.vertex_generator())
>>> next(v.neighbors())
A vertex at (0, 3)
```

sage.geometry.polyhedron.representation.**repr_pretty**(*coefficients*, *type*, *prefix='x'*, *indices=None*, *latex=False*, *style='>='*, *split=False*)

Return a pretty representation of equation/inequality represented by the coefficients.

INPUT:

- `coefficients` – tuple or other iterable
- `type` – either `0` (`PolyhedronRepresentation.INEQUALITY`) or `1` (`PolyhedronRepresentation.EQUATION`)
- `prefix` – string (default: `'x'`)
- `indices` – tuple or other iterable
- `latex` – boolean
- `split` – boolean (default: `False`); if set to `True`, the output is split into a 3-tuple containing the left-hand side, the relation, and the right-hand side of the object
- `style` – either `'positive'` (making all coefficients positive), or `'<='` or `'>='`

OUTPUT: a string or 3-tuple of strings (depending on `split`)

EXAMPLES:

```
sage: from sage.geometry.polyhedron.representation import repr_pretty
sage: from sage.geometry.polyhedron.representation import PolyhedronRepresentation
sage: print(repr_pretty((0, 1, 0, 0), PolyhedronRepresentation.INEQUALITY))
x0 >= 0
sage: print(repr_pretty((1, 2, 1, 0), PolyhedronRepresentation.INEQUALITY))
2*x0 + x1 >= -1
sage: print(repr_pretty((1, -1, -1, 1), PolyhedronRepresentation.EQUATION))
-x0 - x1 + x2 == -1
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.representation import repr_pretty
>>> from sage.geometry.polyhedron.representation import PolyhedronRepresentation
>>> print(repr_pretty((Integer(0), Integer(1), Integer(0), Integer(0)),␣
↪PolyhedronRepresentation.INEQUALITY))
x0 >= 0
```

(continues on next page)

(continued from previous page)

```
>>> print(repr_pretty((Integer(1), Integer(2), Integer(1), Integer(0)),␣
↪PolyhedronRepresentation.INEQUALITY))
2*x0 + x1 >= -1
>>> print(repr_pretty((Integer(1), -Integer(1), -Integer(1), Integer(1)),␣
↪PolyhedronRepresentation.EQUATION))
-x0 - x1 + x2 == -1
```

2.1.5 Functions for plotting polyhedra

class sage.geometry.polyhedron.plot.**Projection**(*polyhedron*, *proj=<function projection_func_identity>*)

Bases: SageObject

The projection of a *Polyhedron*.

This class keeps track of the necessary data to plot the input polyhedron.

coord_index_of(*v*)

Convert a coordinate vector to its internal index.

EXAMPLES:

```
sage: p = polytopes.hypercube(3)
sage: proj = p.projection()
sage: proj.coord_index_of(vector((1,1,1)))
2
```

```
>>> from sage.all import *
>>> p = polytopes.hypercube(Integer(3))
>>> proj = p.projection()
>>> proj.coord_index_of(vector((Integer(1),Integer(1),Integer(1))))
2
```

coord_indices_of(*v_list*)

Convert list of coordinate vectors to the corresponding list of internal indices.

EXAMPLES:

```
sage: p = polytopes.hypercube(3)
sage: proj = p.projection()
sage: proj.coord_indices_of([vector((1,1,1)), vector((1,-1,1))])
[2, 3]
```

```
>>> from sage.all import *
>>> p = polytopes.hypercube(Integer(3))
>>> proj = p.projection()
>>> proj.coord_indices_of([vector((Integer(1),Integer(1),Integer(1))),␣
↪vector((Integer(1),-Integer(1),Integer(1)))])
[2, 3]
```

coordinates_of(*coord_index_list*)

Given a list of indices, return the projected coordinates.

EXAMPLES:

```
sage: p = polytopes.simplex(4, project=True).projection()
sage: p.coordinates_of([1])
[[-0.7071067812, 0.4082482905, 0.2886751346, 0.2236067977]]
```

```
>>> from sage.all import *
>>> p = polytopes.simplex(Integer(4), project=True).projection()
>>> p.coordinates_of([Integer(1)])
[[-0.7071067812, 0.4082482905, 0.2886751346, 0.2236067977]]
```

identity()

Return the identity projection of the polyhedron.

EXAMPLES:

```
sage: # needs sage.groups
sage: p = polytopes.icosahedron(exact=False)
sage: from sage.geometry.polyhedron.plot import Projection
sage: pproj = Projection(p)
sage: ppid = pproj.identity()
sage: ppid.dimension
3
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> p = polytopes.icosahedron(exact=False)
>>> from sage.geometry.polyhedron.plot import Projection
>>> pproj = Projection(p)
>>> ppid = pproj.identity()
>>> ppid.dimension
3
```

render_0d(*point_opts=None*, *line_opts=None*, *polygon_opts=None*)

Return 0d rendering of the projection of a polyhedron into 2-dimensional ambient space.

INPUT:

See `plot()`.

OUTPUT: a 2-d graphics object

EXAMPLES:

```
sage: print(Polyhedron([]).projection().render_0d().description())          #␣
↪needs sage.plot

sage: P = Polyhedron(ieqs=[(1,)])
sage: print(P.projection().render_0d().description())                       #␣
↪needs sage.plot
Point set defined by 1 point(s):    [(0.0, 0.0)]
```

```
>>> from sage.all import *
>>> print(Polyhedron([]).projection().render_0d().description())            #␣
↪needs sage.plot
<BLANKLINE>
```

(continues on next page)

(continued from previous page)

```
>>> P = Polyhedron(ieqs=[(Integer(1),)])
>>> print(P.projection().render_0d().description())                         #␣
↪needs sage.plot
Point set defined by 1 point(s):    [(0.0, 0.0)]
```

render_1d(*point_opts=None*, *line_opts=None*, *polygon_opts=None*)

Return 1d rendering of the projection of a polyhedron into 2-dimensional ambient space.

INPUT:

See `plot()`.

OUTPUT: a 2-d graphics object

EXAMPLES:

```
sage: Polyhedron([(0,), (1,)]).projection().render_1d()                      #␣
↪needs sage.plot
Graphics object consisting of 2 graphics primitives
```

```
>>> from sage.all import *
>>> Polyhedron([(Integer(0),), (Integer(1),)]).projection().render_1d()      ␣
↪               # needs sage.plot
Graphics object consisting of 2 graphics primitives
```

render_2d(*point_opts=None*, *line_opts=None*, *polygon_opts=None*)

Return 2d rendering of the projection of a polyhedron into 2-dimensional ambient space.

EXAMPLES:

```
sage: p1 = Polyhedron(vertices=[[1,1]], rays=[[1,1]])
sage: q1 = p1.projection()
sage: p2 = Polyhedron(vertices=[[1,0], [0,1], [0,0]])
sage: q2 = p2.projection()
sage: p3 = Polyhedron(vertices=[[1,2]])
sage: q3 = p3.projection()
sage: p4 = Polyhedron(vertices=[[2,0]], rays=[[1,-1]], lines=[[1,1]])
sage: q4 = p4.projection()
sage: q1.plot() + q2.plot() + q3.plot() + q4.plot()                           #␣
↪needs sage.plot
Graphics object consisting of 18 graphics primitives
```

```
>>> from sage.all import *
>>> p1 = Polyhedron(vertices=[[Integer(1),Integer(1)]], rays=[[Integer(1),
↪Integer(1)]])
>>> q1 = p1.projection()
>>> p2 = Polyhedron(vertices=[[Integer(1),Integer(0)], [Integer(0),
↪Integer(1)], [Integer(0),Integer(0)]])
>>> q2 = p2.projection()
>>> p3 = Polyhedron(vertices=[[Integer(1),Integer(2)]])
>>> q3 = p3.projection()
>>> p4 = Polyhedron(vertices=[[Integer(2),Integer(0)]], rays=[[Integer(1),-
↪Integer(1)]], lines=[[Integer(1),Integer(1)]])
>>> q4 = p4.projection()
```

(continues on next page)

(continued from previous page)

```
>>> q1.plot() + q2.plot() + q3.plot() + q4.plot()                          #␣
↪needs sage.plot
Graphics object consisting of 18 graphics primitives
```

render_3d(*point_opts=None*, *line_opts=None*, *polygon_opts=None*)

Return 3d rendering of a polyhedron projected into 3-dimensional ambient space.

EXAMPLES:

```
sage: p1 = Polyhedron(vertices=[[1,1,1]], rays=[[1,1,1]])
sage: p2 = Polyhedron(vertices=[[2,0,0], [0,2,0], [0,0,2]])
sage: p3 = Polyhedron(vertices=[[1,0,0], [0,1,0], [0,0,1]],
....:                 rays=[[-1,-1,-1]])
sage: (p1.projection().plot() + p2.projection().plot()                     #␣
↪needs sage.plot
....:   + p3.projection().plot())
Graphics3d Object
```

```
>>> from sage.all import *
>>> p1 = Polyhedron(vertices=[[Integer(1),Integer(1),Integer(1)]],␣
↪rays=[[Integer(1),Integer(1),Integer(1)]])
>>> p2 = Polyhedron(vertices=[[Integer(2),Integer(0),Integer(0)], [Integer(0),
↪Integer(2),Integer(0)], [Integer(0),Integer(0),Integer(2)]])
>>> p3 = Polyhedron(vertices=[[Integer(1),Integer(0),Integer(0)], [Integer(0),
↪Integer(1),Integer(0)], [Integer(0),Integer(0),Integer(1)]],
...                 rays=[[-Integer(1),-Integer(1),-Integer(1)]])
>>> (p1.projection().plot() + p2.projection().plot()                       #␣
↪needs sage.plot
...   + p3.projection().plot())
Graphics3d Object
```

It correctly handles various degenerate cases:

```
sage: # needs sage.plot
sage: Polyhedron(lines=[[1,0,0], [0,1,0], [0,0,1]]).plot()  # whole space
Graphics3d Object
sage: Polyhedron(vertices=[[1,1,1]], rays=[[1,0,0]],
....:            lines=[[0,1,0], [0,0,1]]).plot()           # half space
Graphics3d Object
sage: Polyhedron(lines=[[0,1,0], [0,0,1]],
....:            vertices=[[1,1,1]]).plot()      # R^2 in R^3
Graphics3d Object
sage: Polyhedron(rays=[[0,1,0], [0,0,1]],        # quadrant wedge in R^2
....:            lines=[[1,0,0]]).plot()
Graphics3d Object
sage: Polyhedron(rays=[[0,1,0]],                 # upper half plane in R^3
....:            lines=[[1,0,0]]).plot()
Graphics3d Object
sage: Polyhedron(lines=[[1,0,0]]).plot()         # R^1 in R^2
Graphics3d Object
sage: Polyhedron(rays=[[0,1,0]]).plot()          # Half-line in R^3
Graphics3d Object
```

(continues on next page)

(continued from previous page)

```
sage: Polyhedron(vertices=[[1,1,1]]).plot()       # point in R^3
Graphics3d Object
```

```
>>> from sage.all import *
>>> # needs sage.plot
>>> Polyhedron(lines=[[Integer(1),Integer(0),Integer(0)], [Integer(0),
↪Integer(1),Integer(0)], [Integer(0),Integer(0),Integer(1)]]).plot()   #␣
↪whole space
Graphics3d Object
>>> Polyhedron(vertices=[[Integer(1),Integer(1),Integer(1)]],␣
↪rays=[[Integer(1),Integer(0),Integer(0)]],
...             lines=[[Integer(0),Integer(1),Integer(0)], [Integer(0),
↪Integer(0),Integer(1)]]).plot()           # half space
Graphics3d Object
>>> Polyhedron(lines=[[Integer(0),Integer(1),Integer(0)], [Integer(0),
↪Integer(0),Integer(1)]],
...             vertices=[[Integer(1),Integer(1),Integer(1)]]).plot()      # R^
↪2 in R^3
Graphics3d Object
>>> Polyhedron(rays=[[Integer(0),Integer(1),Integer(0)], [Integer(0),
↪Integer(0),Integer(1)]],        # quadrant wedge in R^2
...             lines=[[Integer(1),Integer(0),Integer(0)]]).plot()
Graphics3d Object
>>> Polyhedron(rays=[[Integer(0),Integer(1),Integer(0)]],                  #␣
↪upper half plane in R^3
...             lines=[[Integer(1),Integer(0),Integer(0)]]).plot()
Graphics3d Object
>>> Polyhedron(lines=[[Integer(1),Integer(0),Integer(0)]]).plot()          # R^
↪1 in R^2
Graphics3d Object
>>> Polyhedron(rays=[[Integer(0),Integer(1),Integer(0)]]).plot()           #␣
↪Half-line in R^3
Graphics3d Object
>>> Polyhedron(vertices=[[Integer(1),Integer(1),Integer(1)]]).plot()       #␣
↪point in R^3
Graphics3d Object
```

The origin is not included, if it is not in the polyhedron (Issue #23555):

```
sage: Q = Polyhedron([[100],[101]])
sage: P = Q*Q*Q; P
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 8 vertices
sage: p = P.plot()                                                          #␣
↪needs sage.plot
sage: p.bounding_box()                                                      #␣
↪needs sage.plot
((100.0, 100.0, 100.0), (101.0, 101.0, 101.0))
```

```
>>> from sage.all import *
>>> Q = Polyhedron([[Integer(100)],[Integer(101)]])
>>> P = Q*Q*Q; P
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 8 vertices
```

(continues on next page)

(continued from previous page)

```
>>> p = P.plot()                                                             #␣
↪needs sage.plot
>>> p.bounding_box()                                                         #␣
↪needs sage.plot
((100.0, 100.0, 100.0), (101.0, 101.0, 101.0))
```

Plot 3d polytope with rainbow colors:

```
sage: polytopes.hypercube(3).plot(polygon='rainbow', alpha=0.4)              #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> polytopes.hypercube(Integer(3)).plot(polygon='rainbow', alpha=RealNumber(
↪'0.4'))              # needs sage.plot
Graphics3d Object
```

render_fill_2d(***kwds*)

Return the filled interior (a polygon) of a polyhedron in 2d.

EXAMPLES:

```
sage: cps = [i^3 for i in srange(-2, 2, 1/5)]
sage: p = Polyhedron(vertices=[[(t^2-1)/(t^2+1), 2*t/(t^2+1)] for t in cps])
sage: proj = p.projection()
sage: filled_poly = proj.render_fill_2d()                                    #␣
↪needs sage.plot
sage: filled_poly.axes_width()                                               #␣
↪needs sage.plot
0.8
```

```
>>> from sage.all import *
>>> cps = [i**Integer(3) for i in srange(-Integer(2), Integer(2), Integer(1)/
↪Integer(5))]
>>> p = Polyhedron(vertices=[[(t**Integer(2)-Integer(1))/
↪(t**Integer(2)+Integer(1)), Integer(2)*t/(t**Integer(2)+Integer(1))] for t␣
↪in cps])
>>> proj = p.projection()
>>> filled_poly = proj.render_fill_2d()                                      #␣
↪needs sage.plot
>>> filled_poly.axes_width()                                                 #␣
↪needs sage.plot
0.8
```

render_line_1d(***kwds*)

Return the line of a polyhedron in 1d.

INPUT:

- `**kwds` – options passed through to `line2d()`

OUTPUT: a 2-d graphics object

EXAMPLES:

```
sage: outline = polytopes.hypercube(1).projection().render_line_1d()          #␣
↪needs sage.plot
sage: outline._objects[0]                                                       #␣
↪needs sage.plot
Line defined by 2 points
```

```
>>> from sage.all import *
>>> outline = polytopes.hypercube(Integer(1)).projection().render_line_1d()   ␣
↪     # needs sage.plot
>>> outline._objects[Integer(0)]                                               ␣
↪     # needs sage.plot
Line defined by 2 points
```

render_outline_2d(***kwds*)

Return the outline (edges) of a polyhedron in 2d.

EXAMPLES:

```
sage: penta = polytopes.regular_polygon(5)                                      #␣
↪needs sage.rings.number_field
sage: outline = penta.projection().render_outline_2d()                         #␣
↪needs sage.plot sage.rings.number_field
sage: outline._objects[0]                                                       #␣
↪needs sage.plot sage.rings.number_field
Line defined by 2 points
```

```
>>> from sage.all import *
>>> penta = polytopes.regular_polygon(Integer(5))                              ␣
↪     # needs sage.rings.number_field
>>> outline = penta.projection().render_outline_2d()                          #␣
↪needs sage.plot sage.rings.number_field
>>> outline._objects[Integer(0)]                                               ␣
↪     # needs sage.plot sage.rings.number_field
Line defined by 2 points
```

render_points_1d(***kwds*)

Return the points of a polyhedron in 1d.

INPUT:

- `**kwds` – options passed through to `point2d()`

OUTPUT: a 2-d graphics object

EXAMPLES:

```
sage: cube1 = polytopes.hypercube(1)
sage: proj = cube1.projection()
sage: points = proj.render_points_1d()                                          #␣
↪needs sage.plot
sage: points._objects                                                           #␣
↪needs sage.plot
[Point set defined by 2 point(s)]
```

```
>>> from sage.all import *
>>> cube1 = polytopes.hypercube(Integer(1))
>>> proj = cube1.projection()
>>> points = proj.render_points_1d()                                        #␣
↪needs sage.plot
>>> points._objects                                                         #␣
↪needs sage.plot
[Point set defined by 2 point(s)]
```

render_points_2d(***kwds*)

Return the points of a polyhedron in 2d.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: hex = polytopes.regular_polygon(6)
sage: proj = hex.projection()
sage: hex_points = proj.render_points_2d()                                   #␣
↪needs sage.plot
sage: hex_points._objects                                                    #␣
↪needs sage.plot
[Point set defined by 6 point(s)]
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> hex = polytopes.regular_polygon(Integer(6))
>>> proj = hex.projection()
>>> hex_points = proj.render_points_2d()                                    #␣
↪needs sage.plot
>>> hex_points._objects                                                     #␣
↪needs sage.plot
[Point set defined by 6 point(s)]
```

render_solid_3d(***kwds*)

Return solid 3d rendering of a 3d polytope.

EXAMPLES:

```
sage: p = polytopes.hypercube(3).projection()
sage: p_solid = p.render_solid_3d(opacity=.7)                                #␣
↪needs sage.plot
sage: type(p_solid)                                                          #␣
↪needs sage.plot
<class 'sage.plot.plot3d.index_face_set.IndexFaceSet'>
```

```
>>> from sage.all import *
>>> p = polytopes.hypercube(Integer(3)).projection()
>>> p_solid = p.render_solid_3d(opacity=RealNumber('.7'))                    ␣
↪          # needs sage.plot
>>> type(p_solid)                                                           #␣
↪needs sage.plot
<class 'sage.plot.plot3d.index_face_set.IndexFaceSet'>
```

render_vertices_3d(***kwds*)

Return the 3d rendering of the vertices.

EXAMPLES:

```
sage: p = polytopes.cross_polytope(3)
sage: proj = p.projection()
sage: verts = proj.render_vertices_3d()                                    #␣
↪needs sage.plot
sage: verts.bounding_box()                                                 #␣
↪needs sage.plot
((-1.0, -1.0, -1.0), (1.0, 1.0, 1.0))
```

```
>>> from sage.all import *
>>> p = polytopes.cross_polytope(Integer(3))
>>> proj = p.projection()
>>> verts = proj.render_vertices_3d()                                     #␣
↪needs sage.plot
>>> verts.bounding_box()                                                  #␣
↪needs sage.plot
((-1.0, -1.0, -1.0), (1.0, 1.0, 1.0))
```

render_wireframe_3d(***kwds*)

Return the 3d wireframe rendering.

EXAMPLES:

```
sage: cube = polytopes.hypercube(3)
sage: cube_proj = cube.projection()
sage: wire = cube_proj.render_wireframe_3d()                               #␣
↪needs sage.plot
sage: print(wire.tachyon().split('\n')[77])   # for testing               #␣
↪needs sage.plot
FCylinder base 1.0 1.0 -1.0 apex -1.0 1.0 -1.0 rad 0.005 texture...
```

```
>>> from sage.all import *
>>> cube = polytopes.hypercube(Integer(3))
>>> cube_proj = cube.projection()
>>> wire = cube_proj.render_wireframe_3d()                                #␣
↪needs sage.plot
>>> print(wire.tachyon().split('\n')[Integer(77)])   # for testing           ␣
↪     # needs sage.plot
FCylinder base 1.0 1.0 -1.0 apex -1.0 1.0 -1.0 rad 0.005 texture...
```

schlegel(*facet=None*, *position=None*)

Return the Schlegel projection.

- The facet is orthonormally transformed into its affine hull.
- The position specifies a point coming out of the barycenter of the facet from which the other vertices will be projected into the facet.

INPUT:

- `facet` – a PolyhedronFace; the facet into which the Schlegel diagram is created. The default is the first facet.

- `position` – a positive number. Determines a relative distance from the barycenter of `facet`. A value close to 0 will place the projection point close to the facet and a large value further away. If the given value is too large, an error is returned. If no position is given, it takes the midpoint of the possible point of views along a line spanned by the barycenter of the facet and a valid point outside the facet.

EXAMPLES:

```
sage: cube4 = polytopes.hypercube(4)
sage: from sage.geometry.polyhedron.plot import Projection
sage: Projection(cube4).schlegel()
The projection of a polyhedron into 3 dimensions
sage: _.plot()                                                          #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> cube4 = polytopes.hypercube(Integer(4))
>>> from sage.geometry.polyhedron.plot import Projection
>>> Projection(cube4).schlegel()
The projection of a polyhedron into 3 dimensions
>>> _.plot()                                                          #␣
↪needs sage.plot
Graphics3d Object
```

The 4-cube with a truncated vertex seen into the resulting tetrahedron facet:

```
sage: tcube4 = cube4.face_truncation(cube4.faces(0)[0])
sage: tcube4.facets()[4]
A 3-dimensional face of a Polyhedron in QQ^4 defined as the convex hull of 4␣
↪vertices
sage: into_tetra = Projection(tcube4).schlegel(tcube4.facets()[4])      #␣
↪needs sage.symbolic
sage: into_tetra.plot()                                                 #␣
↪needs sage.plot sage.symbolic
Graphics3d Object
```

```
>>> from sage.all import *
>>> tcube4 = cube4.face_truncation(cube4.faces(Integer(0))[Integer(0)])
>>> tcube4.facets()[Integer(4)]
A 3-dimensional face of a Polyhedron in QQ^4 defined as the convex hull of 4␣
↪vertices
>>> into_tetra = Projection(tcube4).schlegel(tcube4.facets()[Integer(4)])    ␣
↪      # needs sage.symbolic
>>> into_tetra.plot()                                                 #␣
↪needs sage.plot sage.symbolic
Graphics3d Object
```

Taking a larger value for the position changes the image:

```
sage: into_tetra_far = Projection(tcube4).schlegel(tcube4.facets()[4], 4)    #␣
↪needs sage.symbolic
sage: into_tetra_far.plot()                                             #␣
↪needs sage.plot sage.symbolic
Graphics3d Object
```

```
>>> from sage.all import *
>>> into_tetra_far = Projection(tcube4).schlegel(tcube4.facets()[Integer(4)],␣
↪Integer(4))    # needs sage.symbolic
>>> into_tetra_far.plot()                                                    #␣
↪needs sage.plot sage.symbolic
Graphics3d Object
```

A value which is too large or negative give a projection point that sees more than one facet resulting in a error:

```
sage: Projection(tcube4).schlegel(tcube4.facets()[4], 5)
Traceback (most recent call last):
...
ValueError: the chosen position is too large
sage: Projection(tcube4).schlegel(tcube4.facets()[4], -1)
Traceback (most recent call last):
...
ValueError: 'position' should be a positive number
```

```
>>> from sage.all import *
>>> Projection(tcube4).schlegel(tcube4.facets()[Integer(4)], Integer(5))
Traceback (most recent call last):
...
ValueError: the chosen position is too large
>>> Projection(tcube4).schlegel(tcube4.facets()[Integer(4)], -Integer(1))
Traceback (most recent call last):
...
ValueError: 'position' should be a positive number
```

stereographic(*projection_point=None*)

Return the stereographic projection.

INPUT:

- `projection_point` – the projection point. This must be distinct from the polyhedron's vertices. Default is $(1, 0, \dots, 0)$.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.plot import Projection
sage: proj = Projection(polytopes.buckyball()); proj  # long time
The projection of a polyhedron into 3 dimensions
sage: proj.stereographic([5,2,3]).plot()     # long time                      #␣
↪needs sage.plot
Graphics object consisting of 123 graphics primitives
sage: Projection(polytopes.twenty_four_cell()).stereographic([2,0,0,0])
The projection of a polyhedron into 3 dimensions
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.plot import Projection
>>> proj = Projection(polytopes.buckyball()); proj  # long time
The projection of a polyhedron into 3 dimensions
>>> proj.stereographic([Integer(5),Integer(2),Integer(3)]).plot()     # long␣
↪time                        # needs sage.plot
Graphics object consisting of 123 graphics primitives
```

(continues on next page)

(continued from previous page)

```
>>> Projection(polytopes.twenty_four_cell()).stereographic([Integer(2),
↪Integer(0),Integer(0),Integer(0)])
The projection of a polyhedron into 3 dimensions
```

tikz(*view=[0, 0, 1]*, *angle=0*, *scale=1*, *edge_color='blue!95!black'*, *facet_color='blue!95!black'*, *opacity=0.8*, *vertex_color='green'*, *axis=False*, *output_type='TikzPicture'*)

Return a tikz picture of `self` as a string or as a `TikzPicture` according to a projection `view` and an angle `angle` obtained via the threejs viewer.

INPUT:

- `view` – list (default: [0,0,1]) representing the rotation axis (see note below)
- `angle` – integer (default: 0); angle of rotation in degree from 0 to 360 (see note below)
- `scale` – integer (default: 1); the scaling of the tikz picture
- `edge_color` – string (default: `'blue!95!black'`); representing colors which tikz recognizes
- `facet_color` – string (default: `'blue!95!black'`); representing colors which tikz recognizes
- `vertex_color` – string (default: `'green'`); representing colors which tikz recognizes
- `opacity` – real number (default: 0.8) between 0 and 1 giving the opacity of the front facets
- `axis` – boolean (default: `False`); draw the axes at the origin or not
- `output_type` – string (default: `'TikzPicture'`); valid values are `'LatexExpr'` and `'TikzPicture'`, whether to return a `LatexExpr` object (which inherits from Python `str`) or a `TikzPicture` object from module `sage.misc.latex_standalone`

OUTPUT: `LatexExpr` object or `TikzPicture` object

Note

The inputs `view` and `angle` can be obtained by visualizing it using `.show(aspect_ratio=1)`. This will open an interactive view in your default browser, where you can rotate the polytope. Once the desired view angle is found, click on the information icon in the lower right-hand corner and select *Get Viewpoint*. This will copy a string of the form '[x,y,z],angle' to your local clipboard. Go back to Sage and type `Img = P.projection().tikz([x,y,z],angle)`.

The inputs `view` and `angle` can also be obtained from the viewer Jmol:

```
1) Right click on the image
2) Select ``Console``
3) Select the tab ``State``
4) Scroll to the line ``moveto``
```

It reads something like:

```
moveto 0.0 {x y z angle} Scale
```

The `view` is then [x,y,z] and `angle` is angle. The following number is the scale.

Jmol performs a rotation of `angle` degrees along the vector [x,y,z] and show the result from the z-axis.

EXAMPLES:

```
sage: # needs sage.plot sage.rings.number_field
sage: P1 = polytopes.small_rhombicuboctahedron()
sage: Image1 = P1.projection().tikz([1,3,5], 175, scale=4,
....:                               output_type='TikzPicture')
sage: type(Image1)
<class 'sage.misc.latex_standalone.TikzPicture'>
sage: Image1
\documentclass[tikz]{standalone}
\begin{document}
\begin{tikzpicture}%
        [x={(-0.939161cm, 0.244762cm)},
        y={(0.097442cm, -0.482887cm)},
        z={(0.329367cm, 0.840780cm)},
        scale=4.000000,
...
Use print to see the full content.
...
\node[vertex] at (-2.41421, 1.00000, -1.00000)     {};
\node[vertex] at (-2.41421, -1.00000, 1.00000)     {};
%%
%%
\end{tikzpicture}
\end{document}
sage: _ = Image1.tex('polytope-tikz1.tex')          # not tested
sage: _ = Image1.png('polytope-tikz1.png')          # not tested
sage: _ = Image1.pdf('polytope-tikz1.pdf')          # not tested
sage: _ = Image1.svg('polytope-tikz1.svg')          # not tested
```

```
>>> from sage.all import *
>>> # needs sage.plot sage.rings.number_field
>>> P1 = polytopes.small_rhombicuboctahedron()
>>> Image1 = P1.projection().tikz([Integer(1),Integer(3),Integer(5)],␣
↪Integer(175), scale=Integer(4),
...                               output_type='TikzPicture')
>>> type(Image1)
<class 'sage.misc.latex_standalone.TikzPicture'>
>>> Image1
\documentclass[tikz]{standalone}
\begin{document}
\begin{tikzpicture}%
        [x={(-0.939161cm, 0.244762cm)},
        y={(0.097442cm, -0.482887cm)},
        z={(0.329367cm, 0.840780cm)},
        scale=4.000000,
...
Use print to see the full content.
...
\node[vertex] at (-2.41421, 1.00000, -1.00000)     {};
\node[vertex] at (-2.41421, -1.00000, 1.00000)     {};
%%
%%
\end{tikzpicture}
```

(continues on next page)

(continued from previous page)

```
\end{document}
>>> _ = Image1.tex('polytope-tikz1.tex')          # not tested
>>> _ = Image1.png('polytope-tikz1.png')          # not tested
>>> _ = Image1.pdf('polytope-tikz1.pdf')          # not tested
>>> _ = Image1.svg('polytope-tikz1.svg')          # not tested
```

A second example:

```
sage: P2 = Polyhedron(vertices=[[1, 1], [1, 2], [2, 1]])
sage: Image2 = P2.projection().tikz(scale=3, edge_color='blue!95!black',
....:                               facet_color='orange!95!black', opacity=0.
↪4,
....:                               vertex_color='yellow', axis=True,
....:                               output_type='TikzPicture')
sage: Image2
\documentclass[tikz]{standalone}
\begin{document}
\begin{tikzpicture}%
        [scale=3.000000,
        back/.style={loosely dotted, thin},
        edge/.style={color=blue!95!black, thick},
        facet/.style={fill=orange!95!black,fill opacity=0.400000},
...
Use print to see the full content.
...
\node[vertex] at (1.00000, 2.00000)     {};
\node[vertex] at (2.00000, 1.00000)     {};
%%
%%
\end{tikzpicture}
\end{document}
```

```
>>> from sage.all import *
>>> P2 = Polyhedron(vertices=[[Integer(1), Integer(1)], [Integer(1),␣
↪Integer(2)], [Integer(2), Integer(1)]])
>>> Image2 = P2.projection().tikz(scale=Integer(3), edge_color='blue!95!black
↪',
...                               facet_color='orange!95!black',␣
↪opacity=RealNumber('0.4'),
...                               vertex_color='yellow', axis=True,
...                               output_type='TikzPicture')
>>> Image2
\documentclass[tikz]{standalone}
\begin{document}
\begin{tikzpicture}%
        [scale=3.000000,
        back/.style={loosely dotted, thin},
        edge/.style={color=blue!95!black, thick},
        facet/.style={fill=orange!95!black,fill opacity=0.400000},
...
Use print to see the full content.
...
```

(continues on next page)

(continued from previous page)

```
\node[vertex] at (1.00000, 2.00000)     {};
\node[vertex] at (2.00000, 1.00000)     {};
%%
%%
\end{tikzpicture}
\end{document}
```

The second example using a LatexExpr as output type:

```
sage: # needs sage.plot
sage: Image2 = P2.projection().tikz(scale=3, edge_color='blue!95!black',
....:                               facet_color='orange!95!black', opacity=0.
↪4,
....:                               vertex_color='yellow', axis=True,
....:                               output_type='LatexExpr')
sage: type(Image2)
<class 'sage.misc.latex.LatexExpr'>
sage: print('\n'.join(Image2.splitlines()[:4]))
\begin{tikzpicture}%
    [scale=3.000000,
    back/.style={loosely dotted, thin},
    edge/.style={color=blue!95!black, thick},
sage: with open('polytope-tikz2.tex', 'w') as f:    # not tested
....:     _ = f.write(Image2)
```

```
>>> from sage.all import *
>>> # needs sage.plot
>>> Image2 = P2.projection().tikz(scale=Integer(3), edge_color='blue!95!black
↪',
...                               facet_color='orange!95!black',␣
↪opacity=RealNumber('0.4'),
...                               vertex_color='yellow', axis=True,
...                               output_type='LatexExpr')
>>> type(Image2)
<class 'sage.misc.latex.LatexExpr'>
>>> print('\n'.join(Image2.splitlines()[:Integer(4)]))
\begin{tikzpicture}%
    [scale=3.000000,
    back/.style={loosely dotted, thin},
    edge/.style={color=blue!95!black, thick},
>>> with open('polytope-tikz2.tex', 'w') as f:    # not tested
...     _ = f.write(Image2)
```

A third example:

```
sage: # needs sage.plot
sage: P3 = Polyhedron(vertices=[[-1, -1, 2], [-1, 2, -1], [2, -1, -1]]); P3
A 2-dimensional polyhedron in ZZ^3 defined as the convex hull of 3 vertices
sage: Image3 = P3.projection().tikz([0.5, -1, -0.1], 55, scale=3,
....:                               edge_color='blue!95!black',
....:                               facet_color='orange!95!black', opacity=0.
↪7,
```

(continues on next page)

(continued from previous page)

```
....:                               vertex_color='yellow', axis=True)
sage: Image3
\documentclass[tikz]{standalone}
\begin{document}
\begin{tikzpicture}%
        [x={(0.658184cm, -0.242192cm)},
        y={(-0.096240cm, 0.912008cm)},
        z={(-0.746680cm, -0.331036cm)},
        scale=3.000000,
...
Use print to see the full content.
...
\node[vertex] at (-1.00000, 2.00000, -1.00000)     {};
\node[vertex] at (2.00000, -1.00000, -1.00000)     {};
%%
%%
\end{tikzpicture}
\end{document}
sage: _ = Image3.tex('polytope-tikz3.tex')          # not tested
sage: _ = Image3.png('polytope-tikz3.png')          # not tested
sage: _ = Image3.pdf('polytope-tikz3.pdf')          # not tested
sage: _ = Image3.svg('polytope-tikz3.svg')          # not tested
```

```
>>> from sage.all import *
>>> # needs sage.plot
>>> P3 = Polyhedron(vertices=[[-Integer(1), -Integer(1), Integer(2)], [-
↪Integer(1), Integer(2), -Integer(1)], [Integer(2), -Integer(1), -
↪Integer(1)]]); P3
A 2-dimensional polyhedron in ZZ^3 defined as the convex hull of 3 vertices
>>> Image3 = P3.projection().tikz([RealNumber('0.5'), -Integer(1), -
↪RealNumber('0.1')], Integer(55), scale=Integer(3),
...                                   edge_color='blue!95!black',
...                                   facet_color='orange!95!black',␣
↪opacity=RealNumber('0.7'),
...                                   vertex_color='yellow', axis=True)
>>> Image3
\documentclass[tikz]{standalone}
\begin{document}
\begin{tikzpicture}%
        [x={(0.658184cm, -0.242192cm)},
        y={(-0.096240cm, 0.912008cm)},
        z={(-0.746680cm, -0.331036cm)},
        scale=3.000000,
...
Use print to see the full content.
...
\node[vertex] at (-1.00000, 2.00000, -1.00000)     {};
\node[vertex] at (2.00000, -1.00000, -1.00000)     {};
%%
%%
\end{tikzpicture}
\end{document}
```

(continues on next page)

(continued from previous page)

```
>>> _ = Image3.tex('polytope-tikz3.tex')          # not tested
>>> _ = Image3.png('polytope-tikz3.png')          # not tested
>>> _ = Image3.pdf('polytope-tikz3.pdf')          # not tested
>>> _ = Image3.svg('polytope-tikz3.svg')          # not tested
```

A fourth example:

```
sage: P = Polyhedron(vertices=[[1,1,0,0], [1,2,0,0],
....:                          [2,1,0,0], [0,0,1,0], [0,0,0,1]]); P
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 5 vertices
sage: P.projection().tikz(output_type='TikzPicture')
Traceback (most recent call last):
...
NotImplementedError: The polytope has to live in 2 or 3 dimensions.
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[[Integer(1),Integer(1),Integer(0),Integer(0)],
↪[Integer(1),Integer(2),Integer(0),Integer(0)],
...                          [Integer(2),Integer(1),Integer(0),Integer(0)],
↪[Integer(0),Integer(0),Integer(1),Integer(0)], [Integer(0),Integer(0),
↪Integer(0),Integer(1)]]); P
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 5 vertices
>>> P.projection().tikz(output_type='TikzPicture')
Traceback (most recent call last):
...
NotImplementedError: The polytope has to live in 2 or 3 dimensions.
```

Todo

Make it possible to draw Schlegel diagram for 4-polytopes.

```
sage: P = Polyhedron(vertices=[[1,1,0,0], [1,2,0,0],
....:                          [2,1,0,0], [0,0,1,0], [0,0,0,1]]); P
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 5 vertices
sage: P.projection().tikz(output_type='TikzPicture')
Traceback (most recent call last):
...
NotImplementedError: The polytope has to live in 2 or 3 dimensions.
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[[Integer(1),Integer(1),Integer(0),Integer(0)],
↪ [Integer(1),Integer(2),Integer(0),Integer(0)],
...                          [Integer(2),Integer(1),Integer(0),Integer(0)],
↪ [Integer(0),Integer(0),Integer(1),Integer(0)], [Integer(0),Integer(0),
↪Integer(0),Integer(1)]]); P
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 5 vertices
>>> P.projection().tikz(output_type='TikzPicture')
Traceback (most recent call last):
...
NotImplementedError: The polytope has to live in 2 or 3 dimensions.
```

Make it possible to draw 3-polytopes living in higher dimension.

class sage.geometry.polyhedron.plot.**ProjectionFuncSchlegel**(*facet*, *projection_point*)

Bases: object

The Schlegel projection from the given input point.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.plot import ProjectionFuncSchlegel
sage: fcube = polytopes.hypercube(4)
sage: facet = fcube.facets()[0]
sage: proj = ProjectionFuncSchlegel(facet,[0,-1.5,0,0])
sage: proj([0,0,0,0])[0]
1.0
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.plot import ProjectionFuncSchlegel
>>> fcube = polytopes.hypercube(Integer(4))
>>> facet = fcube.facets()[Integer(0)]
>>> proj = ProjectionFuncSchlegel(facet,[Integer(0),-RealNumber('1.5'),Integer(0),
→Integer(0)])
>>> proj([Integer(0),Integer(0),Integer(0),Integer(0)])[Integer(0)]
1.0
```

class sage.geometry.polyhedron.plot.**ProjectionFuncStereographic**(*projection_point*)

Bases: object

The stereographic (or perspective) projection onto a codimension-1 linear subspace with respect to a sphere centered at the origin.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.plot import ProjectionFuncStereographic
sage: cube = polytopes.hypercube(3).vertices()
sage: proj = ProjectionFuncStereographic([1.2, 3.4, 5.6])
sage: ppoints = [proj(vector(x)) for x in cube]
sage: ppoints[5]
(-0.0918273..., -0.036375...)
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.plot import ProjectionFuncStereographic
>>> cube = polytopes.hypercube(Integer(3)).vertices()
>>> proj = ProjectionFuncStereographic([RealNumber('1.2'), RealNumber('3.4'),␣
→RealNumber('5.6')])
>>> ppoints = [proj(vector(x)) for x in cube]
>>> ppoints[Integer(5)]
(-0.0918273..., -0.036375...)
```

sage.geometry.polyhedron.plot.**cyclic_sort_vertices_2d**(*Vlist*)

Return the vertices/rays in cyclic order if possible.

> **Note**
>
> This works if and only if each vertex/ray is adjacent to exactly two others. For example, any 2-dimensional polyhedron satisfies this.

See `vertex_adjacency_matrix()` for a discussion of "adjacent".

EXAMPLES:

```
sage: from sage.geometry.polyhedron.plot import cyclic_sort_vertices_2d
sage: square = Polyhedron([[1,0],[-1,0],[0,1],[0,-1]])
sage: vertices = [v for v in square.vertex_generator()]
sage: vertices
[A vertex at (-1, 0),
 A vertex at (0, -1),
 A vertex at (0, 1),
 A vertex at (1, 0)]
sage: cyclic_sort_vertices_2d(vertices)
[A vertex at (1, 0),
 A vertex at (0, -1),
 A vertex at (-1, 0),
 A vertex at (0, 1)]
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.plot import cyclic_sort_vertices_2d
>>> square = Polyhedron([[Integer(1),Integer(0)],[-Integer(1),Integer(0)],
↪[Integer(0),Integer(1)],[Integer(0),-Integer(1)]])
>>> vertices = [v for v in square.vertex_generator()]
>>> vertices
[A vertex at (-1, 0),
 A vertex at (0, -1),
 A vertex at (0, 1),
 A vertex at (1, 0)]
>>> cyclic_sort_vertices_2d(vertices)
[A vertex at (1, 0),
 A vertex at (0, -1),
 A vertex at (-1, 0),
 A vertex at (0, 1)]
```

Rays are allowed, too:

```
sage: P = Polyhedron(vertices=[(0, 1), (1, 0), (2, 0), (3, 0), (4, 1)], rays=[(0,
↪1)])
sage: P.adjacency_matrix()
[0 1 0 1 0]
[1 0 1 0 0]
[0 1 0 0 1]
[1 0 0 0 1]
[0 0 1 1 0]
sage: cyclic_sort_vertices_2d(P.Vrepresentation())
[A vertex at (3, 0),
 A vertex at (1, 0),
 A vertex at (0, 1),
 A ray in the direction (0, 1),
 A vertex at (4, 1)]

sage: P = Polyhedron(vertices=[(0, 1), (1, 0), (2, 0), (3, 0), (4, 1)], rays=[(0,
↪1), (1,1)])
sage: P.adjacency_matrix()
```

(continues on next page)

(continued from previous page)

```
[0 1 0 0 0]
[1 0 1 0 0]
[0 1 0 0 1]
[0 0 0 0 1]
[0 0 1 1 0]
sage: cyclic_sort_vertices_2d(P.Vrepresentation())
[A ray in the direction (1, 1),
 A vertex at (3, 0),
 A vertex at (1, 0),
 A vertex at (0, 1),
 A ray in the direction (0, 1)]

sage: P = Polyhedron(vertices=[(1,2)], rays=[(0,1)], lines=[(1,0)])
sage: P.adjacency_matrix()
[0 0 1]
[0 0 0]
[1 0 0]
sage: cyclic_sort_vertices_2d(P.Vrepresentation())
[A vertex at (0, 2),
 A line in the direction (1, 0),
 A ray in the direction (0, 1)]
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[(Integer(0), Integer(1)), (Integer(1), Integer(0)),␣
↪(Integer(2), Integer(0)), (Integer(3), Integer(0)), (Integer(4), Integer(1))],␣
↪rays=[(Integer(0),Integer(1))])
>>> P.adjacency_matrix()
[0 1 0 1 0]
[1 0 1 0 0]
[0 1 0 0 1]
[1 0 0 0 1]
[0 0 1 1 0]
>>> cyclic_sort_vertices_2d(P.Vrepresentation())
[A vertex at (3, 0),
 A vertex at (1, 0),
 A vertex at (0, 1),
 A ray in the direction (0, 1),
 A vertex at (4, 1)]

>>> P = Polyhedron(vertices=[(Integer(0), Integer(1)), (Integer(1), Integer(0)),␣
↪(Integer(2), Integer(0)), (Integer(3), Integer(0)), (Integer(4), Integer(1))],␣
↪rays=[(Integer(0),Integer(1)), (Integer(1),Integer(1))])
>>> P.adjacency_matrix()
[0 1 0 0 0]
[1 0 1 0 0]
[0 1 0 0 1]
[0 0 0 0 1]
[0 0 1 1 0]
>>> cyclic_sort_vertices_2d(P.Vrepresentation())
[A ray in the direction (1, 1),
 A vertex at (3, 0),
 A vertex at (1, 0),
```

(continues on next page)

(continued from previous page)

```
 A vertex at (0, 1),
 A ray in the direction (0, 1)]

>>> P = Polyhedron(vertices=[(Integer(1),Integer(2))], rays=[(Integer(0),
↪Integer(1))], lines=[(Integer(1),Integer(0))])
>>> P.adjacency_matrix()
[0 0 1]
[0 0 0]
[1 0 0]
>>> cyclic_sort_vertices_2d(P.Vrepresentation())
[A vertex at (0, 2),
 A line in the direction (1, 0),
 A ray in the direction (0, 1)]
```

sage.geometry.polyhedron.plot.**projection_func_identity**(*x*)

The identity projection.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.plot import projection_func_identity
sage: projection_func_identity((1,2,3))
[1, 2, 3]
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.plot import projection_func_identity
>>> projection_func_identity((Integer(1),Integer(2),Integer(3)))
[1, 2, 3]
```

2.1.6 A class to keep information about faces of a polyhedron

This module gives you a tool to work with the faces of a polyhedron and their relative position. First, you need to find the faces. To get the faces in a particular dimension, use the `face()` method:

```
sage: P = polytopes.cross_polytope(3)
sage: P.faces(3)
(A 3-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 6␣
↪vertices,)
sage: [f.ambient_V_indices() for f in P.facets()]
[(3, 4, 5),
 (2, 4, 5),
 (1, 3, 5),
 (1, 2, 5),
 (0, 3, 4),
 (0, 2, 4),
 (0, 1, 3),
 (0, 1, 2)]
sage: [f.ambient_V_indices() for f in P.faces(1)]
[(4, 5),
 (3, 5),
 (2, 5),
 (1, 5),
 (3, 4),
```

(continues on next page)

(continued from previous page)

```
 (2, 4),
 (0, 4),
 (1, 3),
 (0, 3),
 (1, 2),
 (0, 2),
 (0, 1)]
```

```
>>> from sage.all import *
>>> P = polytopes.cross_polytope(Integer(3))
>>> P.faces(Integer(3))
(A 3-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 6␣
↪vertices,)
>>> [f.ambient_V_indices() for f in P.facets()]
[(3, 4, 5),
 (2, 4, 5),
 (1, 3, 5),
 (1, 2, 5),
 (0, 3, 4),
 (0, 2, 4),
 (0, 1, 3),
 (0, 1, 2)]
>>> [f.ambient_V_indices() for f in P.faces(Integer(1))]
[(4, 5),
 (3, 5),
 (2, 5),
 (1, 5),
 (3, 4),
 (2, 4),
 (0, 4),
 (1, 3),
 (0, 3),
 (1, 2),
 (0, 2),
 (0, 1)]
```

or `face_lattice()` to get the whole face lattice as a poset:

```
sage: P.face_lattice()                                                    #␣
↪needs sage.combinat
Finite lattice containing 28 elements
```

```
>>> from sage.all import *
>>> P.face_lattice()                                                    #␣
↪needs sage.combinat
Finite lattice containing 28 elements
```

The faces are printed in shorthand notation where each integer is the index of a vertex/ray/line in the same order as the containing Polyhedron's `Vrepresentation()`

```
sage: face = P.faces(1)[8];  face
A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2 vertices
```

(continues on next page)

(continued from previous page)

```
sage: face.ambient_V_indices()
(0, 3)
sage: P.Vrepresentation(0)
A vertex at (-1, 0, 0)
sage: P.Vrepresentation(3)
A vertex at (0, 0, 1)
sage: face.vertices()
(A vertex at (-1, 0, 0), A vertex at (0, 0, 1))
```

```
>>> from sage.all import *
>>> face = P.faces(Integer(1))[Integer(8)];  face
A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2 vertices
>>> face.ambient_V_indices()
(0, 3)
>>> P.Vrepresentation(Integer(0))
A vertex at (-1, 0, 0)
>>> P.Vrepresentation(Integer(3))
A vertex at (0, 0, 1)
>>> face.vertices()
(A vertex at (-1, 0, 0), A vertex at (0, 0, 1))
```

The face itself is not represented by Sage's `sage.geometry.polyhedron.constructor.Polyhedron()` class, but by an auxiliary class to keep the information. You can get the face as a polyhedron with the `PolyhedronFace.as_polyhedron()` method:

```
sage: face.as_polyhedron()
A 1-dimensional polyhedron in ZZ^3 defined as the convex hull of 2 vertices
sage: _.equations()
(An equation (0, 1, 0) x + 0 == 0,
 An equation (1, 0, -1) x + 1 == 0)
```

```
>>> from sage.all import *
>>> face.as_polyhedron()
A 1-dimensional polyhedron in ZZ^3 defined as the convex hull of 2 vertices
>>> _.equations()
(An equation (0, 1, 0) x + 0 == 0,
 An equation (1, 0, -1) x + 1 == 0)
```

class sage.geometry.polyhedron.face.**PolyhedronFace**(*polyhedron*, *V_indices*, *H_indices*)

Bases: `ConvexSet_closed`

A face of a polyhedron.

This class is for use in `face_lattice()`.

INPUT:

No checking is performed whether the H/V-representation indices actually determine a face of the polyhedron. You should not manually create `PolyhedronFace` objects unless you know what you are doing.

OUTPUT: a `PolyhedronFace`

EXAMPLES:

```
sage: octahedron = polytopes.cross_polytope(3)
sage: inequality = octahedron.Hrepresentation(2)
sage: face_h = tuple([ inequality ])
sage: face_v = tuple( inequality.incident() )
sage: face_h_indices = [ h.index() for h in face_h ]
sage: face_v_indices = [ v.index() for v in face_v ]
sage: from sage.geometry.polyhedron.face import PolyhedronFace
sage: face = PolyhedronFace(octahedron, face_v_indices, face_h_indices)
sage: face
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3␣
↪vertices
sage: face.dim()
2
sage: face.ambient_V_indices()
(0, 1, 2)
sage: face.ambient_Hrepresentation()
(An inequality (1, 1, 1) x + 1 >= 0,)
sage: face.ambient_Vrepresentation()
(A vertex at (-1, 0, 0), A vertex at (0, -1, 0), A vertex at (0, 0, -1))
```

```
>>> from sage.all import *
>>> octahedron = polytopes.cross_polytope(Integer(3))
>>> inequality = octahedron.Hrepresentation(Integer(2))
>>> face_h = tuple([ inequality ])
>>> face_v = tuple( inequality.incident() )
>>> face_h_indices = [ h.index() for h in face_h ]
>>> face_v_indices = [ v.index() for v in face_v ]
>>> from sage.geometry.polyhedron.face import PolyhedronFace
>>> face = PolyhedronFace(octahedron, face_v_indices, face_h_indices)
>>> face
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3␣
↪vertices
>>> face.dim()
2
>>> face.ambient_V_indices()
(0, 1, 2)
>>> face.ambient_Hrepresentation()
(An inequality (1, 1, 1) x + 1 >= 0,)
>>> face.ambient_Vrepresentation()
(A vertex at (-1, 0, 0), A vertex at (0, -1, 0), A vertex at (0, 0, -1))
```

affine_tangent_cone()

Return the affine tangent cone of `self` as a polyhedron.

It is equal to the sum of `self` and the cone of feasible directions at any point of the relative interior of `self`.

OUTPUT: a polyhedron

EXAMPLES:

```
sage: half_plane_in_space = Polyhedron(ieqs=[(0,1,0,0)], eqns=[(0,0,0,1)])
sage: line = half_plane_in_space.faces(1)[0]; line
A 1-dimensional face of a
 Polyhedron in QQ^3 defined as the convex hull of 1 vertex and 1 line
```

(continues on next page)

(continued from previous page)

```
sage: T_line = line.affine_tangent_cone()
sage: T_line == half_plane_in_space
True

sage: c = polytopes.cube()
sage: edge = min(c.faces(1))
sage: edge.vertices()
(A vertex at (1, -1, -1), A vertex at (1, 1, -1))
sage: T_edge = edge.affine_tangent_cone()
sage: T_edge.Vrepresentation()
(A line in the direction (0, 1, 0),
 A ray in the direction (0, 0, 1),
 A vertex at (1, 0, -1),
 A ray in the direction (-1, 0, 0))
```

```
>>> from sage.all import *
>>> half_plane_in_space = Polyhedron(ieqs=[(Integer(0),Integer(1),Integer(0),
↪Integer(0))], eqns=[(Integer(0),Integer(0),Integer(0),Integer(1))])
>>> line = half_plane_in_space.faces(Integer(1))[Integer(0)]; line
A 1-dimensional face of a
 Polyhedron in QQ^3 defined as the convex hull of 1 vertex and 1 line
>>> T_line = line.affine_tangent_cone()
>>> T_line == half_plane_in_space
True

>>> c = polytopes.cube()
>>> edge = min(c.faces(Integer(1)))
>>> edge.vertices()
(A vertex at (1, -1, -1), A vertex at (1, 1, -1))
>>> T_edge = edge.affine_tangent_cone()
>>> T_edge.Vrepresentation()
(A line in the direction (0, 1, 0),
 A ray in the direction (0, 0, 1),
 A vertex at (1, 0, -1),
 A ray in the direction (-1, 0, 0))
```

ambient()

Return the containing polyhedron.

EXAMPLES:

```
sage: P = polytopes.cross_polytope(3); P
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 6 vertices
sage: face = P.facets()[3]; face
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3␣
↪vertices
sage: face.polyhedron()
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 6 vertices
```

```
>>> from sage.all import *
>>> P = polytopes.cross_polytope(Integer(3)); P
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 6 vertices
```

(continues on next page)

(continued from previous page)

```
>>> face = P.facets()[Integer(3)]; face
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3␣
↪vertices
>>> face.polyhedron()
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 6 vertices
```

ambient_H_indices()

Return the indices of the H-representation objects of the ambient polyhedron that make up the H-representation of `self`.

See also *ambient_Hrepresentation()*.

OUTPUT: tuple of indices

EXAMPLES:

```
sage: Q = polytopes.cross_polytope(3)
sage: F = Q.faces(1)
sage: [f.ambient_H_indices() for f in F]
[(4, 5),
 (5, 6),
 (4, 7),
 (6, 7),
 (0, 5),
 (3, 4),
 (0, 3),
 (1, 6),
 (0, 1),
 (2, 7),
 (2, 3),
 (1, 2)]
```

```
>>> from sage.all import *
>>> Q = polytopes.cross_polytope(Integer(3))
>>> F = Q.faces(Integer(1))
>>> [f.ambient_H_indices() for f in F]
[(4, 5),
 (5, 6),
 (4, 7),
 (6, 7),
 (0, 5),
 (3, 4),
 (0, 3),
 (1, 6),
 (0, 1),
 (2, 7),
 (2, 3),
 (1, 2)]
```

ambient_Hrepresentation(*index=None*)

Return the H-representation objects of the ambient polytope defining the face.

INPUT:

- `index` – integer or `None` (default)

OUTPUT:

If the optional argument is not present, a tuple of H-representation objects. Each entry is either an inequality or an equation.

If the optional integer `index` is specified, the `index`-th element of the tuple is returned.

EXAMPLES:

```
sage: square = polytopes.hypercube(2)
sage: for face in square.face_lattice():                                    #␣
↪needs sage.combinat
....:     print(face.ambient_Hrepresentation())
(An inequality (-1, 0) x + 1 >= 0, An inequality (0, -1) x + 1 >= 0,
 An inequality (1, 0) x + 1 >= 0, An inequality (0, 1) x + 1 >= 0)
(An inequality (-1, 0) x + 1 >= 0, An inequality (0, 1) x + 1 >= 0)
(An inequality (-1, 0) x + 1 >= 0, An inequality (0, -1) x + 1 >= 0)
(An inequality (-1, 0) x + 1 >= 0,)
(An inequality (0, -1) x + 1 >= 0, An inequality (1, 0) x + 1 >= 0)
(An inequality (0, -1) x + 1 >= 0,)
(An inequality (1, 0) x + 1 >= 0, An inequality (0, 1) x + 1 >= 0)
(An inequality (0, 1) x + 1 >= 0,)
(An inequality (1, 0) x + 1 >= 0,)
()
```

```
>>> from sage.all import *
>>> square = polytopes.hypercube(Integer(2))
>>> for face in square.face_lattice():                                    #␣
↪needs sage.combinat
...     print(face.ambient_Hrepresentation())
(An inequality (-1, 0) x + 1 >= 0, An inequality (0, -1) x + 1 >= 0,
 An inequality (1, 0) x + 1 >= 0, An inequality (0, 1) x + 1 >= 0)
(An inequality (-1, 0) x + 1 >= 0, An inequality (0, 1) x + 1 >= 0)
(An inequality (-1, 0) x + 1 >= 0, An inequality (0, -1) x + 1 >= 0)
(An inequality (-1, 0) x + 1 >= 0,)
(An inequality (0, -1) x + 1 >= 0, An inequality (1, 0) x + 1 >= 0)
(An inequality (0, -1) x + 1 >= 0,)
(An inequality (1, 0) x + 1 >= 0, An inequality (0, 1) x + 1 >= 0)
(An inequality (0, 1) x + 1 >= 0,)
(An inequality (1, 0) x + 1 >= 0,)
()
```

ambient_V_indices()

Return the indices of the V-representation objects of the ambient polyhedron that make up the V-representation of `self`.

See also *ambient_Vrepresentation()*.

OUTPUT: tuple of indices

EXAMPLES:

```
sage: P = polytopes.cube()
sage: F = P.faces(2)
sage: [f.ambient_V_indices() for f in F]
[(0, 3, 4, 5),
```

(continues on next page)

(continued from previous page)

```
 (0, 1, 5, 6),
 (4, 5, 6, 7),
 (2, 3, 4, 7),
 (1, 2, 6, 7),
 (0, 1, 2, 3)]
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> F = P.faces(Integer(2))
>>> [f.ambient_V_indices() for f in F]
[(0, 3, 4, 5),
 (0, 1, 5, 6),
 (4, 5, 6, 7),
 (2, 3, 4, 7),
 (1, 2, 6, 7),
 (0, 1, 2, 3)]
```

ambient_Vrepresentation(*index=None*)

Return the V-representation objects of the ambient polytope defining the face.

INPUT:

- `index` – integer or `None` (default)

OUTPUT:

If the optional argument is not present, a tuple of V-representation objects. Each entry is either a vertex, a ray, or a line.

If the optional integer `index` is specified, the `index`-th element of the tuple is returned.

EXAMPLES:

```
sage: square = polytopes.hypercube(2)
sage: for fl in square.face_lattice():                                      #␣
↪needs sage.combinat
....:     print(fl.ambient_Vrepresentation())
()
(A vertex at (1, -1),)
(A vertex at (1, 1),)
(A vertex at (1, -1), A vertex at (1, 1))
(A vertex at (-1, 1),)
(A vertex at (1, 1), A vertex at (-1, 1))
(A vertex at (-1, -1),)
(A vertex at (1, -1), A vertex at (-1, -1))
(A vertex at (-1, 1), A vertex at (-1, -1))
(A vertex at (1, -1), A vertex at (1, 1),
 A vertex at (-1, 1), A vertex at (-1, -1))
```

```
>>> from sage.all import *
>>> square = polytopes.hypercube(Integer(2))
>>> for fl in square.face_lattice():                                        #␣
↪needs sage.combinat
...     print(fl.ambient_Vrepresentation())
```

(continues on next page)

(continued from previous page)

```
()
(A vertex at (1, -1),)
(A vertex at (1, 1),)
(A vertex at (1, -1), A vertex at (1, 1))
(A vertex at (-1, 1),)
(A vertex at (1, 1), A vertex at (-1, 1))
(A vertex at (-1, -1),)
(A vertex at (1, -1), A vertex at (-1, -1))
(A vertex at (-1, 1), A vertex at (-1, -1))
(A vertex at (1, -1), A vertex at (1, 1),
 A vertex at (-1, 1), A vertex at (-1, -1))
```

ambient_dim()

Return the dimension of the containing polyhedron.

EXAMPLES:

```
sage: P = Polyhedron(vertices = [[1,0,0,0],[0,1,0,0]])
sage: face = P.faces(1)[0]
sage: face.ambient_dim()
4
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices = [[Integer(1),Integer(0),Integer(0),Integer(0)],
↪[Integer(0),Integer(1),Integer(0),Integer(0)]])
>>> face = P.faces(Integer(1))[Integer(0)]
>>> face.ambient_dim()
4
```

ambient_vector_space(*base_field=None*)

Return the ambient vector space.

It is the ambient free module of the containing polyhedron tensored with a field.

INPUT:

- `base_field` – a field (default: the fraction field of the base ring)

EXAMPLES:

```
sage: half_plane = Polyhedron(ieqs=[(0,1,0)])
sage: line = half_plane.faces(1)[0]; line
A 1-dimensional face of a
 Polyhedron in QQ^2 defined as the convex hull of 1 vertex and 1 line
sage: line.ambient_vector_space()
Vector space of dimension 2 over Rational Field
sage: line.ambient_vector_space(AA)                                    #␣
↪needs sage.rings.number_field
Vector space of dimension 2 over Algebraic Real Field
```

```
>>> from sage.all import *
>>> half_plane = Polyhedron(ieqs=[(Integer(0),Integer(1),Integer(0))])
>>> line = half_plane.faces(Integer(1))[Integer(0)]; line
A 1-dimensional face of a
```

(continues on next page)

(continued from previous page)

```
 Polyhedron in QQ^2 defined as the convex hull of 1 vertex and 1 line
>>> line.ambient_vector_space()
Vector space of dimension 2 over Rational Field
>>> line.ambient_vector_space(AA)                                        #␣
↪needs sage.rings.number_field
Vector space of dimension 2 over Algebraic Real Field
```

as_polyhedron(***kwds*)

Return the face as an independent polyhedron.

OUTPUT: a polyhedron

EXAMPLES:

```
sage: P = polytopes.cross_polytope(3);  P
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 6 vertices
sage: face = P.faces(2)[3]; face
A 2-dimensional face of a
 Polyhedron in ZZ^3 defined as the convex hull of 3 vertices
sage: face.as_polyhedron()
A 2-dimensional polyhedron in ZZ^3 defined as the convex hull of 3 vertices

sage: P.intersection(face.as_polyhedron()) == face.as_polyhedron()
True
```

```
>>> from sage.all import *
>>> P = polytopes.cross_polytope(Integer(3));  P
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 6 vertices
>>> face = P.faces(Integer(2))[Integer(3)]; face
A 2-dimensional face of a
 Polyhedron in ZZ^3 defined as the convex hull of 3 vertices
>>> face.as_polyhedron()
A 2-dimensional polyhedron in ZZ^3 defined as the convex hull of 3 vertices

>>> P.intersection(face.as_polyhedron()) == face.as_polyhedron()
True
```

contains(*point*)

Test whether the polyhedron contains the given `point`.

INPUT:

- `point` – a point or its coordinates

EXAMPLES:

```
sage: half_plane = Polyhedron(ieqs=[(0,1,0)])
sage: line = half_plane.faces(1)[0]; line
A 1-dimensional face of a
 Polyhedron in QQ^2 defined as the convex hull of 1 vertex and 1 line
sage: line.contains([0, 1])
True
```

```
>>> from sage.all import *
>>> half_plane = Polyhedron(ieqs=[(Integer(0),Integer(1),Integer(0))])
>>> line = half_plane.faces(Integer(1))[Integer(0)]; line
A 1-dimensional face of a
 Polyhedron in QQ^2 defined as the convex hull of 1 vertex and 1 line
>>> line.contains([Integer(0), Integer(1)])
True
```

As a shorthand, one may use the usual `in` operator:

```
sage: [5, 7] in line
False
```

```
>>> from sage.all import *
>>> [Integer(5), Integer(7)] in line
False
```

dim()

Return the dimension of the face.

OUTPUT: integer

EXAMPLES:

```
sage: fl = polytopes.dodecahedron().face_lattice()                          #␣
↪needs sage.combinat sage.rings.number_field
sage: sorted(x.dim() for x in fl)                                           #␣
↪needs sage.combinat sage.rings.number_field
[-1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3]
```

```
>>> from sage.all import *
>>> fl = polytopes.dodecahedron().face_lattice()                           #␣
↪needs sage.combinat sage.rings.number_field
>>> sorted(x.dim() for x in fl)                                            #␣
↪needs sage.combinat sage.rings.number_field
[-1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3]
```

is_compact()

Return whether `self` is compact.

OUTPUT: boolean

EXAMPLES:

```
sage: half_plane = Polyhedron(ieqs=[(0,1,0)])
sage: line = half_plane.faces(1)[0]; line
A 1-dimensional face of a
 Polyhedron in QQ^2 defined as the convex hull of 1 vertex and 1 line
sage: line.is_compact()
False
```

```
>>> from sage.all import *
>>> half_plane = Polyhedron(ieqs=[(Integer(0),Integer(1),Integer(0))])
>>> line = half_plane.faces(Integer(1))[Integer(0)]; line
A 1-dimensional face of a
 Polyhedron in QQ^2 defined as the convex hull of 1 vertex and 1 line
>>> line.is_compact()
False
```

is_relatively_open()

Return whether `self` is relatively open.

OUTPUT: boolean

EXAMPLES:

```
sage: half_plane = Polyhedron(ieqs=[(0,1,0)])
sage: line = half_plane.faces(1)[0]; line
A 1-dimensional face of a
 Polyhedron in QQ^2 defined as the convex hull of 1 vertex and 1 line
sage: line.is_relatively_open()
True
```

```
>>> from sage.all import *
>>> half_plane = Polyhedron(ieqs=[(Integer(0),Integer(1),Integer(0))])
>>> line = half_plane.faces(Integer(1))[Integer(0)]; line
A 1-dimensional face of a
 Polyhedron in QQ^2 defined as the convex hull of 1 vertex and 1 line
>>> line.is_relatively_open()
True
```

line_generator()

Return a generator for the lines of the face.

EXAMPLES:

```
sage: pr = Polyhedron(rays = [[1,0],[-1,0],[0,1]], vertices = [[-1,-1]])
sage: face = pr.faces(1)[0]
sage: next(face.line_generator())
A line in the direction (1, 0)
```

```
>>> from sage.all import *
>>> pr = Polyhedron(rays = [[Integer(1),Integer(0)],[-Integer(1),Integer(0)],
↪[Integer(0),Integer(1)]], vertices = [[-Integer(1),-Integer(1)]])
>>> face = pr.faces(Integer(1))[Integer(0)]
>>> next(face.line_generator())
A line in the direction (1, 0)
```

lines()

Return all lines of the face.

OUTPUT: a tuple of lines

EXAMPLES:

```
sage: p = Polyhedron(rays = [[1,0],[-1,0],[0,1],[1,1]], vertices = [[-2,-2],
↪[2,3]])
sage: p.lines()
(A line in the direction (1, 0),)
```

```
>>> from sage.all import *
>>> p = Polyhedron(rays = [[Integer(1),Integer(0)],[-Integer(1),Integer(0)],
↪[Integer(0),Integer(1)],[Integer(1),Integer(1)]], vertices = [[-Integer(2),-
↪Integer(2)],[Integer(2),Integer(3)]])
>>> p.lines()
(A line in the direction (1, 0),)
```

n_ambient_Hrepresentation()

Return the number of objects that make up the ambient H-representation of the polyhedron.

See also *ambient_Hrepresentation()*.

OUTPUT: integer

EXAMPLES:

```
sage: p = polytopes.cross_polytope(4)
sage: face = p.face_lattice()[5]; face                                     #␣
↪needs sage.combinat
A 1-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 2␣
↪vertices
sage: face.ambient_Hrepresentation()                                      #␣
↪needs sage.combinat
(An inequality (1, -1, 1, -1) x + 1 >= 0,
 An inequality (1, 1, 1, 1) x + 1 >= 0,
 An inequality (1, 1, 1, -1) x + 1 >= 0,
 An inequality (1, -1, 1, 1) x + 1 >= 0)
sage: face.n_ambient_Hrepresentation()                                    #␣
↪needs sage.combinat
4
```

```
>>> from sage.all import *
>>> p = polytopes.cross_polytope(Integer(4))
>>> face = p.face_lattice()[Integer(5)]; face                              ␣
↪     # needs sage.combinat
A 1-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 2␣
↪vertices
>>> face.ambient_Hrepresentation()                                      #␣
↪needs sage.combinat
(An inequality (1, -1, 1, -1) x + 1 >= 0,
 An inequality (1, 1, 1, 1) x + 1 >= 0,
 An inequality (1, 1, 1, -1) x + 1 >= 0,
 An inequality (1, -1, 1, 1) x + 1 >= 0)
>>> face.n_ambient_Hrepresentation()                                    #␣
↪needs sage.combinat
4
```

n_ambient_Vrepresentation()

Return the number of objects that make up the ambient V-representation of the polyhedron.

See also `ambient_Vrepresentation()`.

OUTPUT: integer

EXAMPLES:

```
sage: p = polytopes.cross_polytope(4)
sage: face = p.face_lattice()[5]; face                                  #␣
↪needs sage.combinat
A 1-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 2␣
↪vertices
sage: face.ambient_Vrepresentation()                                    #␣
↪needs sage.combinat
(A vertex at (-1, 0, 0, 0), A vertex at (0, 0, -1, 0))
sage: face.n_ambient_Vrepresentation()                                  #␣
↪needs sage.combinat
2
```

```
>>> from sage.all import *
>>> p = polytopes.cross_polytope(Integer(4))
>>> face = p.face_lattice()[Integer(5)]; face                          ␣
↪     # needs sage.combinat
A 1-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 2␣
↪vertices
>>> face.ambient_Vrepresentation()                                    #␣
↪needs sage.combinat
(A vertex at (-1, 0, 0, 0), A vertex at (0, 0, -1, 0))
>>> face.n_ambient_Vrepresentation()                                  #␣
↪needs sage.combinat
2
```

n_lines()

Return the number of lines of the face.

OUTPUT: integer

EXAMPLES:

```
sage: p = Polyhedron(rays = [[1,0],[-1,0],[0,1],[1,1]], vertices = [[-2,-2],
↪[2,3]])
sage: p.n_lines()
1
```

```
>>> from sage.all import *
>>> p = Polyhedron(rays = [[Integer(1),Integer(0)],[-Integer(1),Integer(0)],
↪[Integer(0),Integer(1)],[Integer(1),Integer(1)]], vertices = [[-Integer(2),-
↪Integer(2)],[Integer(2),Integer(3)]])
>>> p.n_lines()
1
```

n_rays()

Return the number of rays of the face.

OUTPUT: integer

EXAMPLES:

```
sage: p = Polyhedron(ieqs = [[0,0,0,1],[0,0,1,0],[1,1,0,0]])
sage: face = p.faces(2)[0]
sage: face.n_rays()
2
```

```
>>> from sage.all import *
>>> p = Polyhedron(ieqs = [[Integer(0),Integer(0),Integer(0),Integer(1)],
↪[Integer(0),Integer(0),Integer(1),Integer(0)],[Integer(1),Integer(1),
↪Integer(0),Integer(0)]])
>>> face = p.faces(Integer(2))[Integer(0)]
>>> face.n_rays()
2
```

n_vertices()

Return the number of vertices of the face.

OUTPUT: integer

EXAMPLES:

```
sage: Q = polytopes.cross_polytope(3)
sage: face = Q.faces(2)[0]
sage: face.n_vertices()
3
```

```
>>> from sage.all import *
>>> Q = polytopes.cross_polytope(Integer(3))
>>> face = Q.faces(Integer(2))[Integer(0)]
>>> face.n_vertices()
3
```

normal_cone(*direction='outer'*)

Return the polyhedral cone consisting of normal vectors to hyperplanes supporting `self`.

INPUT:

- `direction` – string (default: `'outer'`); the direction in which to consider the normals. The other allowed option is `'inner'`.

OUTPUT: a polyhedron

EXAMPLES:

```
sage: p = Polyhedron(vertices=[[1,2], [2,1], [-2,2], [-2,-2], [2,-2]])
sage: for v in p.face_generator(0):
....:     vect = v.vertices()[0].vector()
....:     nc = v.normal_cone().rays_list()
....:     print("{} has outer normal cone spanned by {}".format(vect,nc))
....:
(2, 1) has outer normal cone spanned by [[1, 0], [1, 1]]
(1, 2) has outer normal cone spanned by [[0, 1], [1, 1]]
(2, -2) has outer normal cone spanned by [[0, -1], [1, 0]]
(-2, -2) has outer normal cone spanned by [[-1, 0], [0, -1]]
(-2, 2) has outer normal cone spanned by [[-1, 0], [0, 1]]
```

(continues on next page)

(continued from previous page)

```
sage: for v in p.face_generator(0):
....:     vect = v.vertices()[0].vector()
....:     nc = v.normal_cone(direction='inner').rays_list()
....:     print("{} has inner normal cone spanned by {}".format(vect,nc))
....:
(2, 1) has inner normal cone spanned by [[-1, -1], [-1, 0]]
(1, 2) has inner normal cone spanned by [[-1, -1], [0, -1]]
(2, -2) has inner normal cone spanned by [[-1, 0], [0, 1]]
(-2, -2) has inner normal cone spanned by [[0, 1], [1, 0]]
(-2, 2) has inner normal cone spanned by [[0, -1], [1, 0]]
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices=[[Integer(1),Integer(2)], [Integer(2),Integer(1)],
↪ [-Integer(2),Integer(2)], [-Integer(2),-Integer(2)], [Integer(2),-
↪Integer(2)]])
>>> for v in p.face_generator(Integer(0)):
...     vect = v.vertices()[Integer(0)].vector()
...     nc = v.normal_cone().rays_list()
...     print("{} has outer normal cone spanned by {}".format(vect,nc))
....:
(2, 1) has outer normal cone spanned by [[1, 0], [1, 1]]
(1, 2) has outer normal cone spanned by [[0, 1], [1, 1]]
(2, -2) has outer normal cone spanned by [[0, -1], [1, 0]]
(-2, -2) has outer normal cone spanned by [[-1, 0], [0, -1]]
(-2, 2) has outer normal cone spanned by [[-1, 0], [0, 1]]

>>> for v in p.face_generator(Integer(0)):
...     vect = v.vertices()[Integer(0)].vector()
...     nc = v.normal_cone(direction='inner').rays_list()
...     print("{} has inner normal cone spanned by {}".format(vect,nc))
....:
(2, 1) has inner normal cone spanned by [[-1, -1], [-1, 0]]
(1, 2) has inner normal cone spanned by [[-1, -1], [0, -1]]
(2, -2) has inner normal cone spanned by [[-1, 0], [0, 1]]
(-2, -2) has inner normal cone spanned by [[0, 1], [1, 0]]
(-2, 2) has inner normal cone spanned by [[0, -1], [1, 0]]
```

The function works for polytopes that are not full-dimensional:

```
sage: p = polytopes.permutahedron(3)
sage: f1 = p.faces(0)[0]
sage: f2 = p.faces(1)[0]
sage: f3 = p.faces(2)[0]
sage: f1.normal_cone()
A 3-dimensional polyhedron in ZZ^3 defined as
 the convex hull of 1 vertex, 2 rays, 1 line
sage: f2.normal_cone()
A 2-dimensional polyhedron in ZZ^3 defined as
 the convex hull of 1 vertex, 1 ray, 1 line
sage: f3.normal_cone()
A 1-dimensional polyhedron in ZZ^3 defined as
 the convex hull of 1 vertex and 1 line
```

```
>>> from sage.all import *
>>> p = polytopes.permutahedron(Integer(3))
>>> f1 = p.faces(Integer(0))[Integer(0)]
>>> f2 = p.faces(Integer(1))[Integer(0)]
>>> f3 = p.faces(Integer(2))[Integer(0)]
>>> f1.normal_cone()
A 3-dimensional polyhedron in ZZ^3 defined as
 the convex hull of 1 vertex, 2 rays, 1 line
>>> f2.normal_cone()
A 2-dimensional polyhedron in ZZ^3 defined as
 the convex hull of 1 vertex, 1 ray, 1 line
>>> f3.normal_cone()
A 1-dimensional polyhedron in ZZ^3 defined as
 the convex hull of 1 vertex and 1 line
```

Normal cones are only defined for non-empty faces:

```
sage: f0 = p.faces(-1)[0]
sage: f0.normal_cone()
Traceback (most recent call last):
...
ValueError: the empty face does not have a normal cone
```

```
>>> from sage.all import *
>>> f0 = p.faces(-Integer(1))[Integer(0)]
>>> f0.normal_cone()
Traceback (most recent call last):
...
ValueError: the empty face does not have a normal cone
```

polyhedron()

Return the containing polyhedron.

EXAMPLES:

```
sage: P = polytopes.cross_polytope(3); P
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 6 vertices
sage: face = P.facets()[3]; face
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3␣
↪vertices
sage: face.polyhedron()
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 6 vertices
```

```
>>> from sage.all import *
>>> P = polytopes.cross_polytope(Integer(3)); P
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 6 vertices
>>> face = P.facets()[Integer(3)]; face
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3␣
↪vertices
>>> face.polyhedron()
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 6 vertices
```

ray_generator()

Return a generator for the rays of the face.

EXAMPLES:

```
sage: pi = Polyhedron(ieqs = [[1,1,0],[1,0,1]])
sage: face = pi.faces(1)[1]
sage: next(face.ray_generator())
A ray in the direction (1, 0)
```

```
>>> from sage.all import *
>>> pi = Polyhedron(ieqs = [[Integer(1),Integer(1),Integer(0)],[Integer(1),
↪Integer(0),Integer(1)]])
>>> face = pi.faces(Integer(1))[Integer(1)]
>>> next(face.ray_generator())
A ray in the direction (1, 0)
```

rays()

Return the rays of the face.

OUTPUT: a tuple of rays

EXAMPLES:

```
sage: p = Polyhedron(ieqs = [[0,0,0,1],[0,0,1,0],[1,1,0,0]])
sage: face = p.faces(2)[2]
sage: face.rays()
(A ray in the direction (1, 0, 0), A ray in the direction (0, 1, 0))
```

```
>>> from sage.all import *
>>> p = Polyhedron(ieqs = [[Integer(0),Integer(0),Integer(0),Integer(1)],
↪[Integer(0),Integer(0),Integer(1),Integer(0)],[Integer(1),Integer(1),
↪Integer(0),Integer(0)]])
>>> face = p.faces(Integer(2))[Integer(2)]
>>> face.rays()
(A ray in the direction (1, 0, 0), A ray in the direction (0, 1, 0))
```

stacking_locus()

Return the polyhedron containing the points that sees every facet containing `self`.

OUTPUT: a polyhedron

EXAMPLES:

```
sage: cp = polytopes.cross_polytope(4)
sage: facet = cp.facets()[0]
sage: facet.stacking_locus().vertices()
(A vertex at (1/2, 1/2, 1/2, 1/2),
 A vertex at (1, 0, 0, 0),
 A vertex at (0, 0, 0, 1),
 A vertex at (0, 0, 1, 0),
 A vertex at (0, 1, 0, 0))
sage: face = cp.faces(2)[0]
sage: face.stacking_locus().vertices()
(A vertex at (0, 1, 0, 0),
 A vertex at (0, 0, 1, 0),
 A vertex at (1, 0, 0, 0),
 A vertex at (1, 1, 1, 0),
```

(continues on next page)

(continued from previous page)

```
 A vertex at (1/2, 1/2, 1/2, 1/2),
 A vertex at (1/2, 1/2, 1/2, -1/2))
```

```
>>> from sage.all import *
>>> cp = polytopes.cross_polytope(Integer(4))
>>> facet = cp.facets()[Integer(0)]
>>> facet.stacking_locus().vertices()
(A vertex at (1/2, 1/2, 1/2, 1/2),
 A vertex at (1, 0, 0, 0),
 A vertex at (0, 0, 0, 1),
 A vertex at (0, 0, 1, 0),
 A vertex at (0, 1, 0, 0))
>>> face = cp.faces(Integer(2))[Integer(0)]
>>> face.stacking_locus().vertices()
(A vertex at (0, 1, 0, 0),
 A vertex at (0, 0, 1, 0),
 A vertex at (1, 0, 0, 0),
 A vertex at (1, 1, 1, 0),
 A vertex at (1/2, 1/2, 1/2, 1/2),
 A vertex at (1/2, 1/2, 1/2, -1/2))
```

vertex_generator()

Return a generator for the vertices of the face.

EXAMPLES:

```
sage: triangle = Polyhedron(vertices=[[1,0],[0,1],[1,1]])
sage: face = triangle.facets()[0]
sage: for v in face.vertex_generator(): print(v)
A vertex at (1, 0)
A vertex at (1, 1)
sage: type(face.vertex_generator())
<... 'generator'>
```

```
>>> from sage.all import *
>>> triangle = Polyhedron(vertices=[[Integer(1),Integer(0)],[Integer(0),
↪Integer(1)],[Integer(1),Integer(1)]])
>>> face = triangle.facets()[Integer(0)]
>>> for v in face.vertex_generator(): print(v)
A vertex at (1, 0)
A vertex at (1, 1)
>>> type(face.vertex_generator())
<... 'generator'>
```

vertices()

Return all vertices of the face.

OUTPUT: a tuple of vertices

EXAMPLES:

```
sage: triangle = Polyhedron(vertices=[[1,0],[0,1],[1,1]])
sage: face = triangle.faces(1)[2]
```

(continues on next page)

(continued from previous page)

```
sage: face.vertices()
(A vertex at (0, 1), A vertex at (1, 0))
```

```
>>> from sage.all import *
>>> triangle = Polyhedron(vertices=[[Integer(1),Integer(0)],[Integer(0),
↪Integer(1)],[Integer(1),Integer(1)]])
>>> face = triangle.faces(Integer(1))[Integer(2)]
>>> face.vertices()
(A vertex at (0, 1), A vertex at (1, 0))
```

sage.geometry.polyhedron.face.**combinatorial_face_to_polyhedral_face**(*polyhedron*, *combinatorial_face*)

Convert a combinatorial face to a face of a polyhedron.

INPUT:

- `polyhedron` – a polyhedron containing `combinatorial_face`
- `combinatorial_face` – a `CombinatorialFace`

OUTPUT: a *PolyhedronFace*

EXAMPLES:

```
sage: from sage.geometry.polyhedron.face import combinatorial_face_to_polyhedral_
↪face
sage: P = polytopes.simplex()
sage: C = P.combinatorial_polyhedron()
sage: it = C.face_iter()
sage: comb_face = next(it)
sage: combinatorial_face_to_polyhedral_face(P, comb_face)
A 2-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 3␣
↪vertices
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.face import combinatorial_face_to_polyhedral_
↪face
>>> P = polytopes.simplex()
>>> C = P.combinatorial_polyhedron()
>>> it = C.face_iter()
>>> comb_face = next(it)
>>> combinatorial_face_to_polyhedral_face(P, comb_face)
A 2-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 3␣
↪vertices
```

2.1.7 Generate cdd `.ext` / `.ine` file format

sage.geometry.polyhedron.cdd_file_format.**cdd_Hrepresentation**(*cdd_type*, *ieqs*, *eqns*, *file_output=None*)

Return a string containing the H-representation in cddlib's ine format.

INPUT:

- `file_output` – string (optional); a filename to which the representation should be written. If set to `None` (default), representation is returned as a string.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.cdd_file_format import cdd_Hrepresentation
sage: cdd_Hrepresentation('rational', None, [[0,1]])
'H-representation\nlinearity 1 1\nbegin\n 1 2 rational\n 0 1\nend\n'
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.cdd_file_format import cdd_Hrepresentation
>>> cdd_Hrepresentation('rational', None, [[Integer(0),Integer(1)]])
'H-representation\nlinearity 1 1\nbegin\n 1 2 rational\n 0 1\nend\n'
```

sage.geometry.polyhedron.cdd_file_format.**cdd_Vrepresentation**(*cdd_type*, *vertices*, *rays*, *lines*, *file_output=None*)

Return a string containing the V-representation in cddlib's ext format.

INPUT:

- `file_output` – string (optional); a filename to which the representation should be written. If set to `None` (default), representation is returned as a string.

Note

If there is no vertex given, then the origin will be implicitly added. You cannot write the empty V-representation (which cdd would refuse to process).

EXAMPLES:

```
sage: from sage.geometry.polyhedron.cdd_file_format import cdd_Vrepresentation
sage: print(cdd_Vrepresentation('rational', [[0,0]], [[1,0]], [[0,1]]))
V-representation
linearity 1 1
begin
  3 3 rational
  0 0 1
  0 1 0
  1 0 0
end
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.cdd_file_format import cdd_Vrepresentation
>>> print(cdd_Vrepresentation('rational', [[Integer(0),Integer(0)]], [[Integer(1),
→Integer(0)]], [[Integer(0),Integer(1)]]))
V-representation
linearity 1 1
begin
  3 3 rational
  0 0 1
  0 1 0
  1 0 0
end
```

2.1.8 Formal modules generated by polyhedra

class sage.geometry.polyhedron.modules.formal_polyhedra_module.**FormalPolyhedraModule**(*base_ring*, *dimension*, *basis*, *category*)

Bases: CombinatorialFreeModule

Class for formal modules generated by polyhedra.

It is formal because it is free – it does not know about linear relations of polyhedra.

A formal polyhedral module is graded by dimension.

INPUT:

- base_ring – base ring of the module; unrelated to the base ring of the polyhedra
- dimension – the ambient dimension of the polyhedra
- basis – the basis

EXAMPLES:

```
sage: from sage.geometry.polyhedron.modules.formal_polyhedra_module import␣
↪FormalPolyhedraModule
sage: def closed_interval(a, b): return Polyhedron(vertices=[[a], [b]])
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.modules.formal_polyhedra_module import␣
↪FormalPolyhedraModule
>>> def closed_interval(a, b): return Polyhedron(vertices=[[a], [b]])
```

A three-dimensional vector space of polyhedra:

```
sage: I01 = closed_interval(0, 1); I01.rename('conv([0], [1])')
sage: I11 = closed_interval(1, 1); I11.rename('{[1]}')
sage: I12 = closed_interval(1, 2); I12.rename('conv([1], [2])')
sage: basis = [I01, I11, I12]
sage: M = FormalPolyhedraModule(QQ, 1, basis=basis); M
Free module generated by {conv([0], [1]), {[1]}, conv([1], [2])} over Rational␣
↪Field
sage: M.get_order()
[conv([0], [1]), {[1]}, conv([1], [2])]
```

```
>>> from sage.all import *
>>> I01 = closed_interval(Integer(0), Integer(1)); I01.rename('conv([0], [1])')
>>> I11 = closed_interval(Integer(1), Integer(1)); I11.rename('{[1]}')
>>> I12 = closed_interval(Integer(1), Integer(2)); I12.rename('conv([1], [2])')
>>> basis = [I01, I11, I12]
>>> M = FormalPolyhedraModule(QQ, Integer(1), basis=basis); M
Free module generated by {conv([0], [1]), {[1]}, conv([1], [2])} over Rational␣
```

(continues on next page)

(continued from previous page)

```
↪Field
>>> M.get_order()
[conv([0], [1]), {[1]}, conv([1], [2])]
```

A one-dimensional subspace; bases of subspaces just use the indexing set $0, \ldots, d-1$, where d is the dimension:

```
sage: M_lower = M.submodule([M(I11)]); M_lower
Free module generated by {0} over Rational Field
sage: M_lower.print_options(prefix='S')
sage: M_lower.is_submodule(M)
True
sage: x = M(I01) - 2*M(I11) + M(I12)
sage: M_lower.reduce(x)
[conv([0], [1])] + [conv([1], [2])]
sage: M_lower.retract.domain() is M
True
sage: y = M_lower.retract(M(I11)); y
S[0]
sage: M_lower.lift(y)
[{[1]}]
```

```
>>> from sage.all import *
>>> M_lower = M.submodule([M(I11)]); M_lower
Free module generated by {0} over Rational Field
>>> M_lower.print_options(prefix='S')
>>> M_lower.is_submodule(M)
True
>>> x = M(I01) - Integer(2)*M(I11) + M(I12)
>>> M_lower.reduce(x)
[conv([0], [1])] + [conv([1], [2])]
>>> M_lower.retract.domain() is M
True
>>> y = M_lower.retract(M(I11)); y
S[0]
>>> M_lower.lift(y)
[{[1]}]
```

Quotient space; bases of quotient space are families indexed by elements of the ambient space:

```
sage: M_mod_lower = M.quotient_module(M_lower); M_mod_lower
Free module generated by {conv([0], [1]), conv([1], [2])} over Rational Field
sage: M_mod_lower.print_options(prefix='Q')
sage: M_mod_lower.retract(x)
Q[conv([0], [1])] + Q[conv([1], [2])]
sage: M_mod_lower.retract(M(I01) - 2*M(I11) + M(I12)) ==  M_mod_lower.
↪retract(M(I01) + M(I12))
True
```

```
>>> from sage.all import *
>>> M_mod_lower = M.quotient_module(M_lower); M_mod_lower
Free module generated by {conv([0], [1]), conv([1], [2])} over Rational Field
>>> M_mod_lower.print_options(prefix='Q')
```

(continues on next page)

(continued from previous page)

```
>>> M_mod_lower.retract(x)
Q[conv([0], [1])] + Q[conv([1], [2])]
>>> M_mod_lower.retract(M(I01) - Integer(2)*M(I11) + M(I12)) ==  M_mod_lower.
↪retract(M(I01) + M(I12))
True
```

degree_on_basis(*m*)

The degree of an element of the basis is defined as the dimension of the polyhedron.

INPUT:

- `m` – an element of the basis (a polyhedron)

EXAMPLES:

```
sage: from sage.geometry.polyhedron.modules.formal_polyhedra_module import␣
↪FormalPolyhedraModule
sage: def closed_interval(a, b): return Polyhedron(vertices=[[a], [b]])
sage: I01 = closed_interval(0, 1); I01.rename('conv([0], [1])')
sage: I11 = closed_interval(1, 1); I11.rename('{[1]}')
sage: I12 = closed_interval(1, 2); I12.rename('conv([1], [2])')
sage: I02 = closed_interval(0, 2); I02.rename('conv([0], [2])')
sage: M = FormalPolyhedraModule(QQ, 1, basis=[I01, I11, I12, I02])
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.modules.formal_polyhedra_module import␣
↪FormalPolyhedraModule
>>> def closed_interval(a, b): return Polyhedron(vertices=[[a], [b]])
>>> I01 = closed_interval(Integer(0), Integer(1)); I01.rename('conv([0], [1])
↪')
>>> I11 = closed_interval(Integer(1), Integer(1)); I11.rename('{[1]}')
>>> I12 = closed_interval(Integer(1), Integer(2)); I12.rename('conv([1], [2])
↪')
>>> I02 = closed_interval(Integer(0), Integer(2)); I02.rename('conv([0], [2])
↪')
>>> M = FormalPolyhedraModule(QQ, Integer(1), basis=[I01, I11, I12, I02])
```

We can extract homogeneous components:

```
sage: O = M(I01) + M(I11) + M(I12)
sage: O.homogeneous_component(0)
[{[1]}]
sage: O.homogeneous_component(1)
[conv([0], [1])] + [conv([1], [2])]
```

```
>>> from sage.all import *
>>> O = M(I01) + M(I11) + M(I12)
>>> O.homogeneous_component(Integer(0))
[{[1]}]
>>> O.homogeneous_component(Integer(1))
[conv([0], [1])] + [conv([1], [2])]
```

We note that modulo the linear relations of polyhedra, this would only be a filtration, not a grading, as the following example shows:

```
sage: X = M(I01) + M(I12) - M(I02)
sage: X.degree()
1

sage: Y = M(I11)
sage: Y.degree()
0
```

```
>>> from sage.all import *
>>> X = M(I01) + M(I12) - M(I02)
>>> X.degree()
1

>>> Y = M(I11)
>>> Y.degree()
0
```

2.2 Lattice polyhedra

2.2.1 Lattice and reflexive polytopes

This module provides tools for work with lattice and reflexive polytopes. A *convex polytope* is the convex hull of finitely many points in $\mathbf{R}^n$. The dimension n of a polytope is the smallest n such that the polytope can be embedded in $\mathbf{R}^n$.

A *lattice polytope* is a polytope whose vertices all have integer coordinates.

If L is a lattice polytope, the dual polytope of L is

$$\{y \in \mathbf{Z}^n : x \cdot y \geq -1 \text{ all } x \in L\}$$

A *reflexive polytope* is a lattice polytope, such that its polar is also a lattice polytope, i.e. it is bounded and has vertices with integer coordinates.

This Sage module uses Package for Analyzing Lattice Polytopes (PALP), which is a program written in C by Maximilian Kreuzer and Harald Skarke, which is freely available under the GNU license terms at http://hep.itp.tuwien.ac.at/~kreuzer/CY/. Moreover, PALP is included standard with Sage.

PALP is described in the paper arXiv math.SC/0204356. Its distribution also contains the application `nef.x`, which was created by Erwin Riegler and computes nef-partitions and Hodge data for toric complete intersections.

ACKNOWLEDGMENT: polytope.py module written by William Stein was used as an example of organizing an interface between an external program and Sage. William Stein also helped Andrey Novoseltsev with debugging and tuning of this module.

Robert Bradshaw helped Andrey Novoseltsev to realize plot3d function.

Note

IMPORTANT: PALP requires some parameters to be determined during compilation time, i.e., the maximum dimension of polytopes, the maximum number of points, etc. These limitations may lead to errors during calls to different functions of these module. Currently, a `ValueError` exception will be raised if the output of `poly.x` or `nef.x` is empty or contains the exclamation mark. The error message will contain the exact command that caused an error, the description and vertices of the polytope, and the obtained output.

Data obtained from PALP and some other data is cached and most returned values are immutable. In particular, you cannot change the vertices of the polytope or their order after creation of the polytope.

If you are going to work with large sets of data, take a look at `all_*` functions in this module. They precompute different data for sequences of polynomials with a few runs of external programs. This can significantly affect the time of future computations. You can also use dump/load, but not all data will be stored (currently only faces and the number of their internal and boundary points are stored, in addition to polytope vertices and its polar).

AUTHORS:

- Andrey Novoseltsev (2007-01-11): initial version
- Andrey Novoseltsev (2007-01-15): `all_*` functions
- Andrey Novoseltsev (2008-04-01): second version, including:
  - dual nef-partitions and necessary convex_hull and minkowski_sum
  - built-in sequences of 2- and 3-dimensional reflexive polytopes
  - plot3d, skeleton_show
- Andrey Novoseltsev (2009-08-26): dropped maximal dimension requirement
- Andrey Novoseltsev (2010-12-15): new version of nef-partitions
- Andrey Novoseltsev (2013-09-30): switch to PointCollection.
- Maximilian Kreuzer and Harald Skarke: authors of PALP (which was also used to obtain the list of 3-dimensional reflexive polytopes)
- Erwin Riegler: the author of nef.x

`sage.geometry.lattice_polytope.`**`LatticePolytope`**(*data*, *compute_vertices=True*, *n=0*, *lattice=None*)

Construct a lattice polytope.

INPUT:

- `data` – points spanning the lattice polytope, specified as one of:
  - a *point collection* (this is the preferred input and it is the quickest and the most memory efficient one);
  - an iterable of iterables (for example, a list of vectors) defining the point coordinates;
  - a file with matrix data, opened for reading, or
  - a filename of such a file, see `read_palp_point_collection()` for the file format;
- `compute_vertices` – boolean (default: `True`); if `True`, the convex hull of the given points will be computed for determining vertices. Otherwise, the given points must be vertices.
- `n` – integer (default: 0); if `data` is a name of a file, that contains data blocks for several polytopes, the `n`-th block will be used
- `lattice` – the ambient lattice of the polytope. If not given, a suitable lattice will be determined automatically, most likely the *toric lattice* M of the appropriate dimension.

OUTPUT: a *lattice polytope*

EXAMPLES:

```
sage: points = [(1,0,0), (0,1,0), (0,0,1), (-1,0,0), (0,-1,0), (0,0,-1)]
sage: p = LatticePolytope(points)
sage: p
3-d reflexive polytope in 3-d lattice M
```

(continues on next page)

(continued from previous page)

```
sage: p.vertices()
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1)
in 3-d lattice M
```

```
>>> from sage.all import *
>>> points = [(Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0)), (Integer(0),Integer(0),Integer(1)), (-Integer(1),Integer(0),
↪Integer(0)), (Integer(0),-Integer(1),Integer(0)), (Integer(0),Integer(0),-
↪Integer(1))]
>>> p = LatticePolytope(points)
>>> p
3-d reflexive polytope in 3-d lattice M
>>> p.vertices()
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1)
in 3-d lattice M
```

We draw a pretty picture of the polytope in 3-dimensional space:

```
sage: p.plot3d().show()                                                    #␣
↪needs palp sage.plot
```

```
>>> from sage.all import *
>>> p.plot3d().show()                                                    #␣
↪needs palp sage.plot
```

Now we add an extra point, which is in the interior of the polytope…

```
sage: points.append((0,0,0))
sage: p = LatticePolytope(points)
sage: p.nvertices()
6
```

```
>>> from sage.all import *
>>> points.append((Integer(0),Integer(0),Integer(0)))
>>> p = LatticePolytope(points)
>>> p.nvertices()
6
```

You can suppress vertex computation for speed but this can lead to mistakes:

```
sage: p = LatticePolytope(points, compute_vertices=False)
...
```

(continues on next page)

(continued from previous page)

```
sage: p.nvertices()
7
```

```
>>> from sage.all import *
>>> p = LatticePolytope(points, compute_vertices=False)
...
>>> p.nvertices()
7
```

Given points must be in the lattice:

```
sage: LatticePolytope([[1/2], [3/2]])
Traceback (most recent call last):
...
ValueError: points
[[1/2], [3/2]]
are not in 1-d lattice M!
```

```
>>> from sage.all import *
>>> LatticePolytope([[Integer(1)/Integer(2)], [Integer(3)/Integer(2)]])
Traceback (most recent call last):
...
ValueError: points
[[1/2], [3/2]]
are not in 1-d lattice M!
```

But it is OK to create polytopes of non-maximal dimension:

```
sage: p = LatticePolytope([(1,0,0), (0,1,0), (0,0,0),
....:       (-1,0,0), (0,-1,0), (0,0,0), (0,0,0)])
sage: p
2-d lattice polytope in 3-d lattice M
sage: p.vertices()
M(-1,  0, 0),
M( 0, -1, 0),
M( 1,  0, 0),
M( 0,  1, 0)
in 3-d lattice M
```

```
>>> from sage.all import *
>>> p = LatticePolytope([(Integer(1),Integer(0),Integer(0)), (Integer(0),
↪Integer(1),Integer(0)), (Integer(0),Integer(0),Integer(0)),
...       (-Integer(1),Integer(0),Integer(0)), (Integer(0),-Integer(1),
↪Integer(0)), (Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(0),
↪Integer(0))])
>>> p
2-d lattice polytope in 3-d lattice M
>>> p.vertices()
M(-1,  0, 0),
M( 0, -1, 0),
M( 1,  0, 0),
```

(continues on next page)

(continued from previous page)

```
M( 0,  1, 0)
in 3-d lattice M
```

An empty lattice polytope can be considered as well:

```
sage: p = LatticePolytope([], lattice=ToricLattice(3).dual()); p
-1-d lattice polytope in 3-d lattice M
sage: p.lattice_dim()
3
sage: p.npoints()
0
sage: p.nfacets()
0
sage: p.points()
Empty collection
in 3-d lattice M
sage: p.faces()                                                         #␣
↪needs sage.graphs
((-1-d lattice polytope in 3-d lattice M,),)
```

```
>>> from sage.all import *
>>> p = LatticePolytope([], lattice=ToricLattice(Integer(3)).dual()); p
-1-d lattice polytope in 3-d lattice M
>>> p.lattice_dim()
3
>>> p.npoints()
0
>>> p.nfacets()
0
>>> p.points()
Empty collection
in 3-d lattice M
>>> p.faces()                                                         #␣
↪needs sage.graphs
((-1-d lattice polytope in 3-d lattice M,),)
```

class sage.geometry.lattice_polytope.**LatticePolytopeClass**(*points=None*, *compute_vertices=None*, *ambient=None*, *ambient_vertex_indices=None*, *ambient_facet_indices=None*)

Bases: `ConvexSet_compact`, `Hashable`, `LatticePolytope`

Create a lattice polytope.

> **Warning**
>
> This class does not perform any checks of correctness of input nor does it convert input into the standard representation. Use `LatticePolytope()` to construct lattice polytopes.

Lattice polytopes are immutable, but they cache most of the returned values.

INPUT:

The input can be either:

- `points` – *PointCollection*
- `compute_vertices` – boolean

or (these parameters must be given as keywords):

- `ambient` – ambient structure, this polytope *must be a face of* `ambient`
- `ambient_vertex_indices` – increasing list or tuple of integers, indices of vertices of `ambient` generating this polytope
- `ambient_facet_indices` – increasing list or tuple of integers, indices of facets of `ambient` generating this polytope

OUTPUT: lattice polytope

> **Note**
>
> Every polytope has an ambient structure. If it was not specified, it is this polytope itself.

adjacent()

Return faces adjacent to `self` in the ambient face lattice.

Two *distinct* faces F_1 and F_2 of the same face lattice are **adjacent** if all of the following conditions hold:

- F_1 and F_2 have the same dimension d;
- F_1 and F_2 share a facet of dimension $d - 1$;
- F_1 and F_2 are facets of some face of dimension $d + 1$, unless d is the dimension of the ambient structure.

OUTPUT: `tuple` of *lattice polytopes*

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.adjacent()                                                          #␣
↪needs sage.graphs
()
sage: face = o.faces(1)[0]                                                  #␣
↪needs sage.graphs
sage: face.adjacent()                                                       #␣
↪needs sage.graphs
(1-d face of 3-d reflexive polytope in 3-d lattice M,
 1-d face of 3-d reflexive polytope in 3-d lattice M,
 1-d face of 3-d reflexive polytope in 3-d lattice M,
 1-d face of 3-d reflexive polytope in 3-d lattice M)
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> o.adjacent()                                                        #␣
↪needs sage.graphs
()
>>> face = o.faces(Integer(1))[Integer(0)]                                ␣
↪               # needs sage.graphs
>>> face.adjacent()                                                     #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.graphs
(1-d face of 3-d reflexive polytope in 3-d lattice M,
 1-d face of 3-d reflexive polytope in 3-d lattice M,
 1-d face of 3-d reflexive polytope in 3-d lattice M,
 1-d face of 3-d reflexive polytope in 3-d lattice M)
```

affine_transform(*a=1*, *b=0*)

Return a*P+b, where P is this lattice polytope.

Note

1. While `a` and `b` may be rational, the final result must be a lattice polytope, i.e. all vertices must be integral.
2. If the transform (restricted to this polytope) is bijective, facial structure will be preserved, e.g. the first facet of the image will be spanned by the images of vertices which span the first facet of the original polytope.

INPUT:

- `a` – (default: 1) rational scalar or matrix
- `b` – (default: 0) rational scalar or vector, scalars are interpreted as vectors with the same components

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(2)
sage: o.vertices()
M( 1,  0),
M( 0,  1),
M(-1,  0),
M( 0, -1)
in 2-d lattice M
sage: o.affine_transform(2).vertices()
M( 2,  0),
M( 0,  2),
M(-2,  0),
M( 0, -2)
in 2-d lattice M
sage: o.affine_transform(1,1).vertices()
M(2, 1),
M(1, 2),
M(0, 1),
M(1, 0)
in 2-d lattice M
sage: o.affine_transform(b=1).vertices()
M(2, 1),
M(1, 2),
M(0, 1),
M(1, 0)
in 2-d lattice M
sage: o.affine_transform(b=(1, 0)).vertices()
```

(continues on next page)

(continued from previous page)

```
M(2,  0),
M(1,  1),
M(0,  0),
M(1, -1)
in 2-d lattice M
sage: a = matrix(QQ, 2, [1/2, 0, 0, 3/2])
sage: o.polar().vertices()
N( 1,  1),
N( 1, -1),
N(-1, -1),
N(-1,  1)
in 2-d lattice N
sage: o.polar().affine_transform(a, (1/2, -1/2)).vertices()
M(1,  1),
M(1, -2),
M(0, -2),
M(0,  1)
in 2-d lattice M
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(2))
>>> o.vertices()
M( 1,  0),
M( 0,  1),
M(-1,  0),
M( 0, -1)
in 2-d lattice M
>>> o.affine_transform(Integer(2)).vertices()
M( 2,  0),
M( 0,  2),
M(-2,  0),
M( 0, -2)
in 2-d lattice M
>>> o.affine_transform(Integer(1),Integer(1)).vertices()
M(2, 1),
M(1, 2),
M(0, 1),
M(1, 0)
in 2-d lattice M
>>> o.affine_transform(b=Integer(1)).vertices()
M(2, 1),
M(1, 2),
M(0, 1),
M(1, 0)
in 2-d lattice M
>>> o.affine_transform(b=(Integer(1), Integer(0))).vertices()
M(2,  0),
M(1,  1),
M(0,  0),
M(1, -1)
in 2-d lattice M
>>> a = matrix(QQ, Integer(2), [Integer(1)/Integer(2), Integer(0), Integer(0),
```

(continues on next page)

(continued from previous page)

```
↪ Integer(3)/Integer(2)])
>>> o.polar().vertices()
N( 1,  1),
N( 1, -1),
N(-1, -1),
N(-1,  1)
in 2-d lattice N
>>> o.polar().affine_transform(a, (Integer(1)/Integer(2), -Integer(1)/
↪Integer(2))).vertices()
M(1,  1),
M(1, -2),
M(0, -2),
M(0,  1)
in 2-d lattice M
```

While you can use rational transformation, the result must be integer:

```
sage: o.affine_transform(a)
Traceback (most recent call last):
...
ValueError: points
[(1/2, 0), (0, 3/2), (-1/2, 0), (0, -3/2)]
are not in 2-d lattice M!
```

```
>>> from sage.all import *
>>> o.affine_transform(a)
Traceback (most recent call last):
...
ValueError: points
[(1/2, 0), (0, 3/2), (-1/2, 0), (0, -3/2)]
are not in 2-d lattice M!
```

ambient()

Return the ambient structure of `self`.

OUTPUT: lattice polytope containing `self` as a face

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.ambient()
3-d reflexive polytope in 3-d lattice M
sage: o.ambient() is o
True

sage: # needs sage.graphs
sage: face = o.faces(1)[0]
sage: face
1-d face of 3-d reflexive polytope in 3-d lattice M
sage: face.ambient()
3-d reflexive polytope in 3-d lattice M
sage: face.ambient() is o
True
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> o.ambient()
3-d reflexive polytope in 3-d lattice M
>>> o.ambient() is o
True

>>> # needs sage.graphs
>>> face = o.faces(Integer(1))[Integer(0)]
>>> face
1-d face of 3-d reflexive polytope in 3-d lattice M
>>> face.ambient()
3-d reflexive polytope in 3-d lattice M
>>> face.ambient() is o
True
```

ambient_dim()

Return the dimension of the ambient lattice of `self`.

An alias is *ambient_dim()*.

OUTPUT: integer

EXAMPLES:

```
sage: p = LatticePolytope([(1,0)])
sage: p.lattice_dim()
2
sage: p.dim()
0
```

```
>>> from sage.all import *
>>> p = LatticePolytope([(Integer(1),Integer(0))])
>>> p.lattice_dim()
2
>>> p.dim()
0
```

ambient_facet_indices()

Return indices of facets of the ambient polytope containing `self`.

OUTPUT: increasing `tuple` of integers

EXAMPLES:

The polytope itself is not contained in any of its facets:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.ambient_facet_indices()
()
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> o.ambient_facet_indices()
()
```

But each of its other faces is contained in one or more facets:

```
sage: # needs sage.graphs
sage: face = o.faces(1)[0]
sage: face.ambient_facet_indices()
(4, 5)
sage: face.vertices()
M(1, 0, 0),
M(0, 1, 0)
in 3-d lattice M
sage: o.facets()[face.ambient_facet_indices()[0]].vertices()
M(1, 0,  0),
M(0, 1,  0),
M(0, 0, -1)
in 3-d lattice M
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> face = o.faces(Integer(1))[Integer(0)]
>>> face.ambient_facet_indices()
(4, 5)
>>> face.vertices()
M(1, 0, 0),
M(0, 1, 0)
in 3-d lattice M
>>> o.facets()[face.ambient_facet_indices()[Integer(0)]].vertices()
M(1, 0,  0),
M(0, 1,  0),
M(0, 0, -1)
in 3-d lattice M
```

ambient_ordered_point_indices()

Return indices of points of the ambient polytope contained in this one.

OUTPUT:

- `tuple` of integers such that ambient points in this order are geometrically ordered, e.g. for an edge points will appear from one end point to the other.

EXAMPLES:

```
sage: cube = lattice_polytope.cross_polytope(3).polar()
sage: face = cube.facets()[0]                                          #␣
↪needs sage.graphs
sage: face.ambient_ordered_point_indices()                            #␣
↪needs palp sage.graphs
(5, 8, 4, 9, 10, 11, 6, 12, 7)
sage: cube.points(face.ambient_ordered_point_indices())               #␣
↪needs palp sage.graphs
N(-1, -1, -1),
N(-1, -1,  0),
N(-1, -1,  1),
N(-1,  0, -1),
N(-1,  0,  0),
N(-1,  0,  1),
```

(continues on next page)

(continued from previous page)

```
N(-1,  1, -1),
N(-1,  1,  0),
N(-1,  1,  1)
in 3-d lattice N
```

```
>>> from sage.all import *
>>> cube = lattice_polytope.cross_polytope(Integer(3)).polar()
>>> face = cube.facets()[Integer(0)]                                          ␣
↪     # needs sage.graphs
>>> face.ambient_ordered_point_indices()                                    #␣
↪needs palp sage.graphs
(5, 8, 4, 9, 10, 11, 6, 12, 7)
>>> cube.points(face.ambient_ordered_point_indices())                       #␣
↪needs palp sage.graphs
N(-1, -1, -1),
N(-1, -1,  0),
N(-1, -1,  1),
N(-1,  0, -1),
N(-1,  0,  0),
N(-1,  0,  1),
N(-1,  1, -1),
N(-1,  1,  0),
N(-1,  1,  1)
in 3-d lattice N
```

ambient_point_indices()

Return indices of points of the ambient polytope contained in this one.

OUTPUT:

- `tuple` of integers, the order corresponds to the order of points of this polytope.

EXAMPLES:

```
sage: cube = lattice_polytope.cross_polytope(3).polar()
sage: face = cube.facets()[0]                                                 #␣
↪needs sage.graphs
sage: face.ambient_point_indices()                                            #␣
↪needs palp sage.graphs
(4, 5, 6, 7, 8, 9, 10, 11, 12)
sage: cube.points(face.ambient_point_indices()) == face.points()              #␣
↪needs palp sage.graphs
True
```

```
>>> from sage.all import *
>>> cube = lattice_polytope.cross_polytope(Integer(3)).polar()
>>> face = cube.facets()[Integer(0)]                                          ␣
↪     # needs sage.graphs
>>> face.ambient_point_indices()                                            #␣
↪needs palp sage.graphs
(4, 5, 6, 7, 8, 9, 10, 11, 12)
>>> cube.points(face.ambient_point_indices()) == face.points()              #␣
↪needs palp sage.graphs
```

(continues on next page)

(continued from previous page)

```
True
```

ambient_vector_space(*base_field=None*)

Return the ambient vector space.

It is the ambient lattice (*lattice()*) tensored with a field.

INPUT:

- base_field – (default: the rationals) a field

EXAMPLES:

```
sage: p = LatticePolytope([(1,0)])
sage: p.ambient_vector_space()
Vector space of dimension 2 over Rational Field
sage: p.ambient_vector_space(AA)                                          #␣
↪needs sage.rings.number_field
Vector space of dimension 2 over Algebraic Real Field
```

```
>>> from sage.all import *
>>> p = LatticePolytope([(Integer(1),Integer(0))])
>>> p.ambient_vector_space()
Vector space of dimension 2 over Rational Field
>>> p.ambient_vector_space(AA)                                          #␣
↪needs sage.rings.number_field
Vector space of dimension 2 over Algebraic Real Field
```

ambient_vertex_indices()

Return indices of vertices of the ambient structure generating self.

OUTPUT: increasing tuple of integers

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.ambient_vertex_indices()
(0, 1, 2, 3, 4, 5)
sage: face = o.faces(1)[0]                                                #␣
↪needs sage.graphs
sage: face.ambient_vertex_indices()                                       #␣
↪needs sage.graphs
(0, 1)
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> o.ambient_vertex_indices()
(0, 1, 2, 3, 4, 5)
>>> face = o.faces(Integer(1))[Integer(0)]                                ␣
↪               # needs sage.graphs
>>> face.ambient_vertex_indices()                                       #␣
↪needs sage.graphs
(0, 1)
```

boundary_point_indices()

Return indices of (relative) boundary lattice points of this polytope.

OUTPUT: increasing `tuple` of integers

EXAMPLES:

All points but the origin are on the boundary of this square:

```
sage: square = lattice_polytope.cross_polytope(2).polar()
sage: square.points()                                                   #␣
↪needs palp
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
sage: square.boundary_point_indices()                                   #␣
↪needs palp
(0, 1, 2, 3, 4, 5, 7, 8)
```

```
>>> from sage.all import *
>>> square = lattice_polytope.cross_polytope(Integer(2)).polar()
>>> square.points()                                                     #␣
↪needs palp
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
>>> square.boundary_point_indices()                                     #␣
↪needs palp
(0, 1, 2, 3, 4, 5, 7, 8)
```

For an edge the boundary is formed by the end points:

```
sage: face = square.edges()[0]                                          #␣
↪needs sage.graphs
sage: face.points()                                                     #␣
↪needs sage.graphs
N(-1, -1),
N(-1,  1),
N(-1,  0)
in 2-d lattice N
sage: face.boundary_point_indices()                                     #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.graphs
(0, 1)
```

```
>>> from sage.all import *
>>> face = square.edges()[Integer(0)]
↪     # needs sage.graphs
>>> face.points()                                                        #␣
↪needs sage.graphs
N(-1, -1),
N(-1,  1),
N(-1,  0)
in 2-d lattice N
>>> face.boundary_point_indices()                                        #␣
↪needs sage.graphs
(0, 1)
```

boundary_points()

Return (relative) boundary lattice points of this polytope.

OUTPUT: a `point collection`

EXAMPLES:

All points but the origin are on the boundary of this square:

```
sage: square = lattice_polytope.cross_polytope(2).polar()
sage: square.boundary_points()                                           #␣
↪needs palp
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
```

```
>>> from sage.all import *
>>> square = lattice_polytope.cross_polytope(Integer(2)).polar()
>>> square.boundary_points()                                             #␣
↪needs palp
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
```

For an edge the boundary is formed by the end points:

```
sage: face = square.edges()[0]                                          #␣
↪needs sage.graphs
sage: face.boundary_points()                                            #␣
↪needs sage.graphs
N(-1, -1),
N(-1,  1)
in 2-d lattice N
```

```
>>> from sage.all import *
>>> face = square.edges()[Integer(0)]                                   ␣
↪     # needs sage.graphs
>>> face.boundary_points()                                            #␣
↪needs sage.graphs
N(-1, -1),
N(-1,  1)
in 2-d lattice N
```

contains(**args*)

Check if a given point is contained in `self`.

INPUT:

- an attempt will be made to convert all arguments into a single element of the ambient space of `self`; if it fails, `False` will be returned

OUTPUT:

- `True` if the given point is contained in `self`, `False` otherwise

EXAMPLES:

```
sage: p = lattice_polytope.cross_polytope(2)
sage: p.contains(p.lattice()(1,0))
True
sage: p.contains((1,0))
True
sage: p.contains(1,0)
True
sage: p.contains((2,0))
False
```

```
>>> from sage.all import *
>>> p = lattice_polytope.cross_polytope(Integer(2))
>>> p.contains(p.lattice()(Integer(1),Integer(0)))
True
>>> p.contains((Integer(1),Integer(0)))
True
>>> p.contains(Integer(1),Integer(0))
True
>>> p.contains((Integer(2),Integer(0)))
False
```

dim()

Return the dimension of this polytope.

EXAMPLES:

We create a 3-dimensional octahedron and check its dimension:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.dim()
3
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> o.dim()
3
```

Now we create a 2-dimensional diamond in a 3-dimensional space:

```
sage: p = LatticePolytope([(1,0,0), (0,1,0), (-1,0,0), (0,-1,0)])
sage: p.dim()
2
sage: p.lattice_dim()
3
```

```
>>> from sage.all import *
>>> p = LatticePolytope([(Integer(1),Integer(0),Integer(0)), (Integer(0),
↪Integer(1),Integer(0)), (-Integer(1),Integer(0),Integer(0)), (Integer(0),-
↪Integer(1),Integer(0))])
>>> p.dim()
2
>>> p.lattice_dim()
3
```

distances(*point=None*)

Return the matrix of distances for this polytope or distances for the given point.

The matrix of distances m gives distances m[i,j] between the i-th facet (which is also the i-th vertex of the polar polytope in the reflexive case) and j-th point of this polytope.

If point is specified, integral distances from the point to all facets of this polytope will be computed.

EXAMPLES: The matrix of distances for a 3-dimensional octahedron:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.distances()                                                     #␣
↪needs palp
[2 0 0 0 2 2 1]
[2 2 0 0 0 2 1]
[2 2 2 0 0 0 1]
[2 0 2 0 2 0 1]
[0 0 2 2 2 0 1]
[0 0 0 2 2 2 1]
[0 2 0 2 0 2 1]
[0 2 2 2 0 0 1]
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> o.distances()                                                      #␣
↪needs palp
```

(continues on next page)

(continued from previous page)

```
[2 0 0 0 2 2 1]
[2 2 0 0 0 2 1]
[2 2 2 0 0 0 1]
[2 0 2 0 2 0 1]
[0 0 2 2 2 0 1]
[0 0 0 2 2 2 1]
[0 2 0 2 0 2 1]
[0 2 2 2 0 0 1]
```

Distances from facets to the point (1,2,3):

```
sage: o.distances([1,2,3])
(-3, 1, 7, 3, 1, -5, -1, 5)
```

```
>>> from sage.all import *
>>> o.distances([Integer(1),Integer(2),Integer(3)])
(-3, 1, 7, 3, 1, -5, -1, 5)
```

It is OK to use RATIONAL coordinates:

```
sage: o.distances([1,2,3/2])
(-3/2, 5/2, 11/2, 3/2, -1/2, -7/2, 1/2, 7/2)
sage: o.distances([1,2,sqrt(2)])                                        #␣
↪needs sage.symbolic
Traceback (most recent call last):
...
TypeError: unable to convert sqrt(2) to an element of Rational Field
```

```
>>> from sage.all import *
>>> o.distances([Integer(1),Integer(2),Integer(3)/Integer(2)])
(-3/2, 5/2, 11/2, 3/2, -1/2, -7/2, 1/2, 7/2)
>>> o.distances([Integer(1),Integer(2),sqrt(Integer(2))])               ␣
↪                       # needs sage.symbolic
Traceback (most recent call last):
...
TypeError: unable to convert sqrt(2) to an element of Rational Field
```

Now we create a non-spanning polytope:

```
sage: p = LatticePolytope([(1,0,0), (0,1,0), (-1,0,0), (0,-1,0)])
sage: p.distances()                                                     #␣
↪needs palp
[2 2 0 0 1]
[2 0 0 2 1]
[0 0 2 2 1]
[0 2 2 0 1]
sage: p.distances((1/2, 3, 0))                                          #␣
↪needs palp
(9/2, -3/2, -5/2, 7/2)
```

```
>>> from sage.all import *
>>> p = LatticePolytope([(Integer(1),Integer(0),Integer(0)), (Integer(0),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1),Integer(0)), (-Integer(1),Integer(0),Integer(0)), (Integer(0),-
↪Integer(1),Integer(0))])
>>> p.distances()                                                        #␣
↪needs palp
[2 2 0 0 1]
[2 0 0 2 1]
[0 0 2 2 1]
[0 2 2 0 1]
>>> p.distances((Integer(1)/Integer(2), Integer(3), Integer(0)))             ␣
↪                                # needs palp
(9/2, -3/2, -5/2, 7/2)
```

This point is not even in the affine subspace of the polytope:

```
sage: p.distances((1, 1, 1))                                              #␣
↪needs palp
(3, 1, -1, 1)
```

```
>>> from sage.all import *
>>> p.distances((Integer(1), Integer(1), Integer(1)))                        ␣
↪                         # needs palp
(3, 1, -1, 1)
```

dual()

Return the dual face under face duality of polar reflexive polytopes.

This duality extends the correspondence between vertices and facets.

OUTPUT: a *lattice polytope*

EXAMPLES:

```
sage: # needs sage.graphs
sage: o = lattice_polytope.cross_polytope(4)
sage: e = o.edges()[0]; e
1-d face of 4-d reflexive polytope in 4-d lattice M
sage: ed = e.dual(); ed
2-d face of 4-d reflexive polytope in 4-d lattice N
sage: ed.ambient() is e.ambient().polar()
True
sage: e.ambient_vertex_indices() == ed.ambient_facet_indices()
True
sage: e.ambient_facet_indices() == ed.ambient_vertex_indices()
True
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> o = lattice_polytope.cross_polytope(Integer(4))
>>> e = o.edges()[Integer(0)]; e
1-d face of 4-d reflexive polytope in 4-d lattice M
>>> ed = e.dual(); ed
2-d face of 4-d reflexive polytope in 4-d lattice N
>>> ed.ambient() is e.ambient().polar()
```

(continues on next page)

(continued from previous page)

```
True
>>> e.ambient_vertex_indices() == ed.ambient_facet_indices()
True
>>> e.ambient_facet_indices() == ed.ambient_vertex_indices()
True
```

dual_lattice()

Return the dual of the ambient lattice of `self`.

OUTPUT:

- a lattice. If possible (that is, if *lattice()* has a `dual()` method), the dual lattice is returned. Otherwise, $\mathbf{Z}^n$ is returned, where n is the dimension of `self`.

EXAMPLES:

```
sage: LatticePolytope([(1,0)]).dual_lattice()
2-d lattice N
sage: LatticePolytope([], lattice=ZZ^3).dual_lattice()
Ambient free module of rank 3
over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> LatticePolytope([(Integer(1),Integer(0))]).dual_lattice()
2-d lattice N
>>> LatticePolytope([], lattice=ZZ**Integer(3)).dual_lattice()
Ambient free module of rank 3
over the principal ideal domain Integer Ring
```

edges()

Return edges (faces of dimension 1) of `self`.

OUTPUT: `tuple` of *lattice polytopes*

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.edges()                                                              #␣
↪needs sage.graphs
(1-d face of 3-d reflexive polytope in 3-d lattice M,
...
 1-d face of 3-d reflexive polytope in 3-d lattice M)
sage: len(o.edges())                                                         #␣
↪needs sage.graphs
12
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> o.edges()                                                              #␣
↪needs sage.graphs
(1-d face of 3-d reflexive polytope in 3-d lattice M,
...
 1-d face of 3-d reflexive polytope in 3-d lattice M)
>>> len(o.edges())                                                         #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.graphs
12
```

face_lattice()

Return the face lattice of `self`.

This lattice will have the empty polytope as the bottom and this polytope itself as the top.

OUTPUT:

- `finite poset` of *lattice polytopes*.

EXAMPLES:

Let's take a look at the face lattice of a square:

```
sage: square = LatticePolytope([(0,0), (1,0), (1,1), (0,1)])
sage: L = square.face_lattice(); L                                   #␣
↪needs sage.graphs
Finite lattice containing 10 elements with distinguished linear extension
```

```
>>> from sage.all import *
>>> square = LatticePolytope([(Integer(0),Integer(0)), (Integer(1),
↪Integer(0)), (Integer(1),Integer(1)), (Integer(0),Integer(1))])
>>> L = square.face_lattice(); L                                     #␣
↪needs sage.graphs
Finite lattice containing 10 elements with distinguished linear extension
```

To see all faces arranged by dimension, you can do this:

```
sage: for level in L.level_sets(): print(level)                      #␣
↪needs sage.graphs
[-1-d face of 2-d lattice polytope in 2-d lattice M]
[0-d face of 2-d lattice polytope in 2-d lattice M,
 0-d face of 2-d lattice polytope in 2-d lattice M,
 0-d face of 2-d lattice polytope in 2-d lattice M,
 0-d face of 2-d lattice polytope in 2-d lattice M]
[1-d face of 2-d lattice polytope in 2-d lattice M,
 1-d face of 2-d lattice polytope in 2-d lattice M,
 1-d face of 2-d lattice polytope in 2-d lattice M,
 1-d face of 2-d lattice polytope in 2-d lattice M]
[2-d lattice polytope in 2-d lattice M]
```

```
>>> from sage.all import *
>>> for level in L.level_sets(): print(level)                        #␣
↪needs sage.graphs
[-1-d face of 2-d lattice polytope in 2-d lattice M]
[0-d face of 2-d lattice polytope in 2-d lattice M,
 0-d face of 2-d lattice polytope in 2-d lattice M,
 0-d face of 2-d lattice polytope in 2-d lattice M,
 0-d face of 2-d lattice polytope in 2-d lattice M]
[1-d face of 2-d lattice polytope in 2-d lattice M,
 1-d face of 2-d lattice polytope in 2-d lattice M,
 1-d face of 2-d lattice polytope in 2-d lattice M,
```

(continues on next page)

(continued from previous page)

```
 1-d face of 2-d lattice polytope in 2-d lattice M]
[2-d lattice polytope in 2-d lattice M]
```

For a particular face you can look at its actual vertices…

```
sage: face = L.level_sets()[1][0]                                          #␣
↪needs sage.graphs
sage: face.vertices()                                                      #␣
↪needs sage.graphs
M(0, 0)
in 2-d lattice M
```

```
>>> from sage.all import *
>>> face = L.level_sets()[Integer(1)][Integer(0)]                          ␣
↪              # needs sage.graphs
>>> face.vertices()                                                      #␣
↪needs sage.graphs
M(0, 0)
in 2-d lattice M
```

… or you can see the index of the vertex of the original polytope that corresponds to the above one:

```
sage: face.ambient_vertex_indices()                                        #␣
↪needs sage.graphs
(0,)
sage: square.vertex(0)
M(0, 0)
```

```
>>> from sage.all import *
>>> face.ambient_vertex_indices()                                        #␣
↪needs sage.graphs
(0,)
>>> square.vertex(Integer(0))
M(0, 0)
```

An alternative to extracting faces from the face lattice is to use `faces()` method:

```
sage: face is square.faces(dim=0)[0]                                       #␣
↪needs sage.graphs
True
```

```
>>> from sage.all import *
>>> face is square.faces(dim=Integer(0))[Integer(0)]                       ␣
↪              # needs sage.graphs
True
```

The advantage of working with the face lattice directly is that you can (relatively easily) get faces that are related to the given one:

```
sage: face = L.level_sets()[1][0]                                          #␣
↪needs sage.graphs
sage: D = L.hasse_diagram()                                                #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.graphs
sage: sorted(D.neighbors(face))                                              #␣
↪needs sage.graphs
[-1-d face of 2-d lattice polytope in 2-d lattice M,
 1-d face of 2-d lattice polytope in 2-d lattice M,
 1-d face of 2-d lattice polytope in 2-d lattice M]
```

```
>>> from sage.all import *
>>> face = L.level_sets()[Integer(1)][Integer(0)]                              ␣
↪              # needs sage.graphs
>>> D = L.hasse_diagram()                                                    #␣
↪needs sage.graphs
>>> sorted(D.neighbors(face))                                                #␣
↪needs sage.graphs
[-1-d face of 2-d lattice polytope in 2-d lattice M,
 1-d face of 2-d lattice polytope in 2-d lattice M,
 1-d face of 2-d lattice polytope in 2-d lattice M]
```

However, you can achieve some of this functionality using *facets()*, *facet_of()*, and *adjacent()* methods:

```
sage: # needs sage.graphs
sage: face = square.faces(0)[0]
sage: face
0-d face of 2-d lattice polytope in 2-d lattice M
sage: face.vertices()
M(0, 0)
in 2-d lattice M
sage: face.facets()
(-1-d face of 2-d lattice polytope in 2-d lattice M,)
sage: face.facet_of()
(1-d face of 2-d lattice polytope in 2-d lattice M,
 1-d face of 2-d lattice polytope in 2-d lattice M)
sage: face.adjacent()
(0-d face of 2-d lattice polytope in 2-d lattice M,
 0-d face of 2-d lattice polytope in 2-d lattice M)
sage: face.adjacent()[0].vertices()
M(1, 0)
in 2-d lattice M
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> face = square.faces(Integer(0))[Integer(0)]
>>> face
0-d face of 2-d lattice polytope in 2-d lattice M
>>> face.vertices()
M(0, 0)
in 2-d lattice M
>>> face.facets()
(-1-d face of 2-d lattice polytope in 2-d lattice M,)
>>> face.facet_of()
(1-d face of 2-d lattice polytope in 2-d lattice M,
```

(continues on next page)

(continued from previous page)

```
 1-d face of 2-d lattice polytope in 2-d lattice M)
>>> face.adjacent()
(0-d face of 2-d lattice polytope in 2-d lattice M,
 0-d face of 2-d lattice polytope in 2-d lattice M)
>>> face.adjacent()[Integer(0)].vertices()
M(1, 0)
in 2-d lattice M
```

Note that if `p` is a face of `superp`, then the face lattice of `p` consists of (appropriate) faces of `superp`:

```
sage: # needs sage.graphs
sage: superp = LatticePolytope([(1,2,3,4), (5,6,7,8),
....:                           (1,2,4,8), (1,3,9,7)])
sage: superp.face_lattice()
Finite lattice containing 16 elements with distinguished linear extension
sage: superp.face_lattice().top()
3-d lattice polytope in 4-d lattice M
sage: p = superp.facets()[0]
sage: p
2-d face of 3-d lattice polytope in 4-d lattice M
sage: p.face_lattice()
Finite poset containing 8 elements with distinguished linear extension
sage: p.face_lattice().bottom()
-1-d face of 3-d lattice polytope in 4-d lattice M
sage: p.face_lattice().top()
2-d face of 3-d lattice polytope in 4-d lattice M
sage: p.face_lattice().top() is p
True
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> superp = LatticePolytope([(Integer(1),Integer(2),Integer(3),Integer(4)),␣
↪(Integer(5),Integer(6),Integer(7),Integer(8)),
...                           (Integer(1),Integer(2),Integer(4),Integer(8)),␣
↪(Integer(1),Integer(3),Integer(9),Integer(7))])
>>> superp.face_lattice()
Finite lattice containing 16 elements with distinguished linear extension
>>> superp.face_lattice().top()
3-d lattice polytope in 4-d lattice M
>>> p = superp.facets()[Integer(0)]
>>> p
2-d face of 3-d lattice polytope in 4-d lattice M
>>> p.face_lattice()
Finite poset containing 8 elements with distinguished linear extension
>>> p.face_lattice().bottom()
-1-d face of 3-d lattice polytope in 4-d lattice M
>>> p.face_lattice().top()
2-d face of 3-d lattice polytope in 4-d lattice M
>>> p.face_lattice().top() is p
True
```

faces(*dim=None*, *codim=None*)

Return faces of `self` of specified (co)dimension.

INPUT:

- `dim` – integer; dimension of the requested faces
- `codim` – integer; codimension of the requested faces

Note

You can specify at most one parameter. If you don't give any, then all faces will be returned.

OUTPUT:

- if either `dim` or `codim` is given, the output will be a `tuple` of *lattice polytopes*;
- if neither `dim` nor `codim` is given, the output will be the `tuple` of tuples as above, giving faces of all existing dimensions. If you care about inclusion relations between faces, consider using `face_lattice()` or `adjacent()`, `facet_of()`, and `facets()`.

EXAMPLES:

Let's take a look at the faces of a square:

```
sage: square = LatticePolytope([(0,0), (1,0), (1,1), (0,1)])
sage: square.faces()                                                        #␣
↪needs sage.graphs
((-1-d face of 2-d lattice polytope in 2-d lattice M,),
 (0-d face of 2-d lattice polytope in 2-d lattice M,
  0-d face of 2-d lattice polytope in 2-d lattice M,
  0-d face of 2-d lattice polytope in 2-d lattice M,
  0-d face of 2-d lattice polytope in 2-d lattice M),
 (1-d face of 2-d lattice polytope in 2-d lattice M,
  1-d face of 2-d lattice polytope in 2-d lattice M,
  1-d face of 2-d lattice polytope in 2-d lattice M,
  1-d face of 2-d lattice polytope in 2-d lattice M),
 (2-d lattice polytope in 2-d lattice M,))
```

```
>>> from sage.all import *
>>> square = LatticePolytope([(Integer(0),Integer(0)), (Integer(1),
↪Integer(0)), (Integer(1),Integer(1)), (Integer(0),Integer(1))])
>>> square.faces()                                                          #␣
↪needs sage.graphs
((-1-d face of 2-d lattice polytope in 2-d lattice M,),
 (0-d face of 2-d lattice polytope in 2-d lattice M,
  0-d face of 2-d lattice polytope in 2-d lattice M,
  0-d face of 2-d lattice polytope in 2-d lattice M,
  0-d face of 2-d lattice polytope in 2-d lattice M),
 (1-d face of 2-d lattice polytope in 2-d lattice M,
  1-d face of 2-d lattice polytope in 2-d lattice M,
  1-d face of 2-d lattice polytope in 2-d lattice M,
  1-d face of 2-d lattice polytope in 2-d lattice M),
 (2-d lattice polytope in 2-d lattice M,))
```

Its faces of dimension one (i.e., edges):

```
sage: square.faces(dim=1)                                               #␣
↪needs sage.graphs
(1-d face of 2-d lattice polytope in 2-d lattice M,
 1-d face of 2-d lattice polytope in 2-d lattice M,
 1-d face of 2-d lattice polytope in 2-d lattice M,
 1-d face of 2-d lattice polytope in 2-d lattice M)
```

```
>>> from sage.all import *
>>> square.faces(dim=Integer(1))                                         ␣
↪     # needs sage.graphs
(1-d face of 2-d lattice polytope in 2-d lattice M,
 1-d face of 2-d lattice polytope in 2-d lattice M,
 1-d face of 2-d lattice polytope in 2-d lattice M,
 1-d face of 2-d lattice polytope in 2-d lattice M)
```

Its faces of codimension one are the same (also edges):

```
sage: square.faces(codim=1) is square.faces(dim=1)                      #␣
↪needs sage.graphs
True
```

```
>>> from sage.all import *
>>> square.faces(codim=Integer(1)) is square.faces(dim=Integer(1))       ␣
↪              # needs sage.graphs
True
```

Let's pick a particular face:

```
sage: face = square.faces(dim=1)[0]                                     #␣
↪needs sage.graphs
```

```
>>> from sage.all import *
>>> face = square.faces(dim=Integer(1))[Integer(0)]                     ␣
↪              # needs sage.graphs
```

Now you can look at the actual vertices of this face…

```
sage: face.vertices()                                                   #␣
↪needs sage.graphs
M(0, 0),
M(0, 1)
in 2-d lattice M
```

```
>>> from sage.all import *
>>> face.vertices()                                                   #␣
↪needs sage.graphs
M(0, 0),
M(0, 1)
in 2-d lattice M
```

… or you can see indices of the vertices of the original polytope that correspond to the above ones:

```
sage: face.ambient_vertex_indices()                                         #␣
↪needs sage.graphs
(0, 3)
sage: square.vertices(face.ambient_vertex_indices())                        #␣
↪needs sage.graphs
M(0, 0),
M(0, 1)
in 2-d lattice M
```

```
>>> from sage.all import *
>>> face.ambient_vertex_indices()                                           #␣
↪needs sage.graphs
(0, 3)
>>> square.vertices(face.ambient_vertex_indices())                          #␣
↪needs sage.graphs
M(0, 0),
M(0, 1)
in 2-d lattice M
```

facet_constant(*i*)

Return the constant in the i-th facet inequality of this polytope.

This is equivalent to `facet_constants()[i]`.

INPUT:

- `i` – integer; the index of the facet

OUTPUT: integer; the constant in the i-th facet inequality

See also

facet_constants(), *facet_normal()*, *facet_normals()*, *facets()*.

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.facet_constant(0)
1
sage: o.facet_constant(0) == o.facet_constants()[0]
True
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> o.facet_constant(Integer(0))
1
>>> o.facet_constant(Integer(0)) == o.facet_constants()[Integer(0)]
True
```

facet_constants()

Return facet constants of `self`.

Facet inequalities have form $n \cdot x + c \geq 0$ where n is the inner normal and c is a constant.

OUTPUT: integer vector

See also

facet_constant(), *facet_normal()*, *facet_normals()*, *facets()*.

EXAMPLES:

For reflexive polytopes all constants are 1:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.vertices()
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1)
in 3-d lattice M
sage: o.facet_constants()
(1, 1, 1, 1, 1, 1, 1, 1)
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> o.vertices()
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1)
in 3-d lattice M
>>> o.facet_constants()
(1, 1, 1, 1, 1, 1, 1, 1)
```

Here is an example of a 3-dimensional polytope in a 4-dimensional space with 3 facets containing the origin:

```
sage: p = LatticePolytope([(0,0,0,0), (1,1,1,3),
....:                       (1,-1,1,3), (-1,-1,1,3)])
sage: p.vertices()
M( 0,  0, 0, 0),
M( 1,  1, 1, 3),
M( 1, -1, 1, 3),
M(-1, -1, 1, 3)
in 4-d lattice M
sage: p.facet_constants()
(0, 0, 3, 0)
```

```
>>> from sage.all import *
>>> p = LatticePolytope([(Integer(0),Integer(0),Integer(0),Integer(0)),␣
↪(Integer(1),Integer(1),Integer(1),Integer(3)),
...                       (Integer(1),-Integer(1),Integer(1),Integer(3)), (-
↪Integer(1),-Integer(1),Integer(1),Integer(3))])
>>> p.vertices()
M( 0,  0, 0, 0),
```

(continues on next page)

(continued from previous page)

```
M( 1,  1, 1, 3),
M( 1, -1, 1, 3),
M(-1, -1, 1, 3)
in 4-d lattice M
>>> p.facet_constants()
(0, 0, 3, 0)
```

facet_normal(*i*)

Return the inner normal to the `i`-th facet of this polytope.

This is equivalent to `facet_normals()[i]`.

INPUT:

- `i` – integer; the index of the facet

OUTPUT: a vector

See also

facet_constant(), *facet_constants()*, *facet_normals()*, *facets()*.

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.facet_normal(0)
N(1, -1, -1)
sage: o.facet_normal(0) is o.facet_normals()[0]
True
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> o.facet_normal(Integer(0))
N(1, -1, -1)
>>> o.facet_normal(Integer(0)) is o.facet_normals()[Integer(0)]
True
```

facet_normals()

Return inner normals to the facets of `self`.

If this polytope is not full-dimensional, facet normals will define this polytope in the affine subspace spanned by it.

OUTPUT:

- a *point collection* in the *dual_lattice()* of `self`.

See also

facet_constant(), *facet_constants()*, *facet_normal()*, *facets()*.

EXAMPLES:

Normals to facets of an octahedron are vertices of a cube:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.vertices()
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1)
in 3-d lattice M
sage: o.facet_normals()
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
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> o.vertices()
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1)
in 3-d lattice M
>>> o.facet_normals()
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
```

Here is an example of a 3-dimensional polytope in a 4-dimensional space:

```
sage: p = LatticePolytope([(0,0,0,0), (1,1,1,3),
....:                       (1,-1,1,3), (-1,-1,1,3)])
sage: p.vertices()
M( 0,  0, 0, 0),
M( 1,  1, 1, 3),
M( 1, -1, 1, 3),
M(-1, -1, 1, 3)
in 4-d lattice M
sage: p.facet_normals()
N( 0,  3, 0,  1),
```

(continues on next page)

(continued from previous page)

```
N( 1, -1, 0,  0),
N( 0,  0, 0, -1),
N(-3,  0, 0,  1)
in 4-d lattice N
sage: p.facet_constants()
(0, 0, 3, 0)
```

```
>>> from sage.all import *
>>> p = LatticePolytope([(Integer(0),Integer(0),Integer(0),Integer(0)),␣
↪(Integer(1),Integer(1),Integer(1),Integer(3)),
...                          (Integer(1),-Integer(1),Integer(1),Integer(3)),  (-
↪Integer(1),-Integer(1),Integer(1),Integer(3))])
>>> p.vertices()
M( 0,  0, 0, 0),
M( 1,  1, 1, 3),
M( 1, -1, 1, 3),
M(-1, -1, 1, 3)
in 4-d lattice M
>>> p.facet_normals()
N( 0,  3, 0,  1),
N( 1, -1, 0,  0),
N( 0,  0, 0, -1),
N(-3,  0, 0,  1)
in 4-d lattice N
>>> p.facet_constants()
(0, 0, 3, 0)
```

Now we manually compute the distance matrix of this polytope. Since it is a simplex, each line (corresponding to a facet) should consist of zeros (indicating generating vertices of the corresponding facet) and a single positive number (since our normals are inner):

```
sage: matrix([[n * v + c for v in p.vertices()]
....:     for n, c in zip(p.facet_normals(), p.facet_constants())])
[0 6 0 0]
[0 0 2 0]
[3 0 0 0]
[0 0 0 6]
```

```
>>> from sage.all import *
>>> matrix([[n * v + c for v in p.vertices()]
...     for n, c in zip(p.facet_normals(), p.facet_constants())])
[0 6 0 0]
[0 0 2 0]
[3 0 0 0]
[0 0 0 6]
```

facet_of()

Return elements of the ambient face lattice having `self` as a facet.

OUTPUT: `tuple` of *lattice polytopes*

EXAMPLES:

```
sage: # needs sage.graphs
sage: square = LatticePolytope([(0,0), (1,0), (1,1), (0,1)])
sage: square.facet_of()
()
sage: face = square.faces(0)[0]
sage: len(face.facet_of())
2
sage: face.facet_of()[1]
1-d face of 2-d lattice polytope in 2-d lattice M
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> square = LatticePolytope([(Integer(0),Integer(0)), (Integer(1),
↪Integer(0)), (Integer(1),Integer(1)), (Integer(0),Integer(1))])
>>> square.facet_of()
()
>>> face = square.faces(Integer(0))[Integer(0)]
>>> len(face.facet_of())
2
>>> face.facet_of()[Integer(1)]
1-d face of 2-d lattice polytope in 2-d lattice M
```

facets()

Return facets (faces of codimension 1) of `self`.

OUTPUT: `tuple` of *lattice polytopes*

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.facets()                                                          #␣
↪needs sage.graphs
(2-d face of 3-d reflexive polytope in 3-d lattice M,
...
 2-d face of 3-d reflexive polytope in 3-d lattice M)
sage: len(o.facets())                                                     #␣
↪needs sage.graphs
8
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> o.facets()                                                            #␣
↪needs sage.graphs
(2-d face of 3-d reflexive polytope in 3-d lattice M,
...
 2-d face of 3-d reflexive polytope in 3-d lattice M)
>>> len(o.facets())                                                       #␣
↪needs sage.graphs
8
```

incidence_matrix()

Return the incidence matrix.

Note

The columns correspond to facets/facet normals in the order of *facet_normals()*, the rows correspond to the vertices in the order of *vertices()*.

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(2)
sage: o.incidence_matrix()
[0 0 1 1]
[0 1 1 0]
[1 1 0 0]
[1 0 0 1]
sage: o.faces(1)[0].incidence_matrix()                                  #␣
↪needs sage.graphs
[1 0]
[0 1]

sage: o = lattice_polytope.cross_polytope(4)
sage: o.incidence_matrix().column(3).nonzero_positions()
[3, 4, 5, 6]
sage: o.facets()[3].ambient_vertex_indices()                            #␣
↪needs sage.graphs
(3, 4, 5, 6)
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(2))
>>> o.incidence_matrix()
[0 0 1 1]
[0 1 1 0]
[1 1 0 0]
[1 0 0 1]
>>> o.faces(Integer(1))[Integer(0)].incidence_matrix()                  ␣
↪                # needs sage.graphs
[1 0]
[0 1]

>>> o = lattice_polytope.cross_polytope(Integer(4))
>>> o.incidence_matrix().column(Integer(3)).nonzero_positions()
[3, 4, 5, 6]
>>> o.facets()[Integer(3)].ambient_vertex_indices()                     ␣
↪       # needs sage.graphs
(3, 4, 5, 6)
```

index()

Return the index of this polytope in the internal database of 2- or 3-dimensional reflexive polytopes. Databases are stored in the directory of the package.

Note

The first call to this function for each dimension can take a few seconds while the dictionary of all polytopes is constructed, but after that it is cached and fast.

Return type
integer

EXAMPLES: We check what is the index of the "diamond" in the database:

```
sage: d = lattice_polytope.cross_polytope(2)
sage: d.index()                                                      #␣
↪needs palp
3
```

```
>>> from sage.all import *
>>> d = lattice_polytope.cross_polytope(Integer(2))
>>> d.index()                                                      #␣
↪needs palp
3
```

Note that polytopes with the same index are not necessarily the same:

```
sage: d.vertices()
M( 1,  0),
M( 0,  1),
M(-1,  0),
M( 0, -1)
in 2-d lattice M
sage: lattice_polytope.ReflexivePolytope(2,3).vertices()
M( 1,  0),
M( 0,  1),
M( 0, -1),
M(-1,  0)
in 2-d lattice M
```

```
>>> from sage.all import *
>>> d.vertices()
M( 1,  0),
M( 0,  1),
M(-1,  0),
M( 0, -1)
in 2-d lattice M
>>> lattice_polytope.ReflexivePolytope(Integer(2),Integer(3)).vertices()
M( 1,  0),
M( 0,  1),
M( 0, -1),
M(-1,  0)
in 2-d lattice M
```

But they are in the same $GL(\mathbf{Z}^n)$ orbit and have the same normal form:

```
sage: d.normal_form()                                                #␣
↪needs sage.groups
M( 1,  0),
M( 0,  1),
M( 0, -1),
M(-1,  0)
```

(continues on next page)

(continued from previous page)

```
in 2-d lattice M
sage: lattice_polytope.ReflexivePolytope(2,3).normal_form()             #␣
↪needs sage.groups
M( 1,  0),
M( 0,  1),
M( 0, -1),
M(-1,  0)
in 2-d lattice M
```

```
>>> from sage.all import *
>>> d.normal_form()                                                    #␣
↪needs sage.groups
M( 1,  0),
M( 0,  1),
M( 0, -1),
M(-1,  0)
in 2-d lattice M
>>> lattice_polytope.ReflexivePolytope(Integer(2),Integer(3)).normal_form()  ␣
↪              # needs sage.groups
M( 1,  0),
M( 0,  1),
M( 0, -1),
M(-1,  0)
in 2-d lattice M
```

interior_point_indices()

Return indices of (relative) interior lattice points of this polytope.

OUTPUT: increasing `tuple` of integers

EXAMPLES:

The origin is the only interior point of this square:

```
sage: square = lattice_polytope.cross_polytope(2).polar()
sage: square.points()                                                   #␣
↪needs palp
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
sage: square.interior_point_indices()                                   #␣
↪needs palp
(6,)
```

```
>>> from sage.all import *
>>> square = lattice_polytope.cross_polytope(Integer(2)).polar()
```

(continues on next page)

(continued from previous page)

```
>>> square.points()                                                          #␣
↪needs palp
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
>>> square.interior_point_indices()                                          #␣
↪needs palp
(6,)
```

Its edges also have a single interior point each:

```
sage: face = square.edges()[0]                                                #␣
↪needs sage.graphs
sage: face.points()                                                           #␣
↪needs sage.graphs
N(-1, -1),
N(-1,  1),
N(-1,  0)
in 2-d lattice N
sage: face.interior_point_indices()                                           #␣
↪needs sage.graphs
(2,)
```

```
>>> from sage.all import *
>>> face = square.edges()[Integer(0)]                                          ␣
↪     # needs sage.graphs
>>> face.points()                                                            #␣
↪needs sage.graphs
N(-1, -1),
N(-1,  1),
N(-1,  0)
in 2-d lattice N
>>> face.interior_point_indices()                                            #␣
↪needs sage.graphs
(2,)
```

interior_points()

Return (relative) boundary lattice points of this polytope.

OUTPUT: a `point collection`

EXAMPLES:

The origin is the only interior point of this square:

```
sage: square = lattice_polytope.cross_polytope(2).polar()
sage: square.interior_points()                                          #␣
↪needs palp
N(0, 0)
in 2-d lattice N
```

```
>>> from sage.all import *
>>> square = lattice_polytope.cross_polytope(Integer(2)).polar()
>>> square.interior_points()                                          #␣
↪needs palp
N(0, 0)
in 2-d lattice N
```

Its edges also have a single interior point each:

```
sage: face = square.edges()[0]                                          #␣
↪needs sage.graphs
sage: face.interior_points()                                          #␣
↪needs sage.graphs
N(-1, 0)
in 2-d lattice N
```

```
>>> from sage.all import *
>>> face = square.edges()[Integer(0)]                                    ␣
↪      # needs sage.graphs
>>> face.interior_points()                                          #␣
↪needs sage.graphs
N(-1, 0)
in 2-d lattice N
```

is_reflexive()

Return `True` if this polytope is reflexive.

EXAMPLES: The 3-dimensional octahedron is reflexive (and 4319 other 3-polytopes):

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.is_reflexive()
True
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> o.is_reflexive()
True
```

But not all polytopes are reflexive:

```
sage: p = LatticePolytope([(1,0,0), (0,1,17), (-1,0,0), (0,-1,0)])
sage: p.is_reflexive()
False
```

```
>>> from sage.all import *
>>> p = LatticePolytope([(Integer(1),Integer(0),Integer(0)), (Integer(0),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1),Integer(17)), (-Integer(1),Integer(0),Integer(0)), (Integer(0),-
↪Integer(1),Integer(0))])
>>> p.is_reflexive()
False
```

Only full-dimensional polytopes can be reflexive (otherwise the polar set is not a polytope at all, since it is unbounded):

```
sage: p = LatticePolytope([(1,0,0), (0,1,0), (-1,0,0), (0,-1,0)])
sage: p.is_reflexive()
False
```

```
>>> from sage.all import *
>>> p = LatticePolytope([(Integer(1),Integer(0),Integer(0)), (Integer(0),
↪Integer(1),Integer(0)), (-Integer(1),Integer(0),Integer(0)), (Integer(0),-
↪Integer(1),Integer(0))])
>>> p.is_reflexive()
False
```

lattice()

Return the ambient lattice of `self`.

OUTPUT: a lattice

EXAMPLES:

```
sage: lattice_polytope.cross_polytope(3).lattice()
3-d lattice M
```

```
>>> from sage.all import *
>>> lattice_polytope.cross_polytope(Integer(3)).lattice()
3-d lattice M
```

lattice_dim()

Return the dimension of the ambient lattice of `self`.

An alias is *ambient_dim()*.

OUTPUT: integer

EXAMPLES:

```
sage: p = LatticePolytope([(1,0)])
sage: p.lattice_dim()
2
sage: p.dim()
0
```

```
>>> from sage.all import *
>>> p = LatticePolytope([(Integer(1),Integer(0))])
>>> p.lattice_dim()
2
>>> p.dim()
0
```

linearly_independent_vertices()

Return a maximal set of linearly independent vertices.

OUTPUT: a tuple of vertex indices

EXAMPLES:

```
sage: L = LatticePolytope([[0, 0], [-1, 1], [-1, -1]])
sage: L.linearly_independent_vertices()
(1, 2)
sage: L = LatticePolytope([[0, 0, 0]])
sage: L.linearly_independent_vertices()
()
sage: L = LatticePolytope([[0, 1, 0]])
sage: L.linearly_independent_vertices()
(0,)
```

```
>>> from sage.all import *
>>> L = LatticePolytope([[Integer(0), Integer(0)], [-Integer(1), Integer(1)],␣
↪[-Integer(1), -Integer(1)]])
>>> L.linearly_independent_vertices()
(1, 2)
>>> L = LatticePolytope([[Integer(0), Integer(0), Integer(0)]])
>>> L.linearly_independent_vertices()
()
>>> L = LatticePolytope([[Integer(0), Integer(1), Integer(0)]])
>>> L.linearly_independent_vertices()
(0,)
```

nef_partitions(*keep_symmetric=False*, *keep_products=True*, *keep_projections=True*, *hodge_numbers=False*)

Return 2-part nef-partitions of `self`.

INPUT:

- `keep_symmetric` – boolean (default: `False`); if `True`, "-s" option will be passed to `nef.x` in order to keep symmetric partitions, i.e. partitions related by lattice automorphisms preserving `self`
- `keep_products` – boolean (default: `True`); if `True`, "-D" option will be passed to `nef.x` in order to keep product partitions, with corresponding complete intersections being direct products
- `keep_projections` – boolean (default: `True`); if `True`, "-P" option will be passed to `nef.x` in order to keep projection partitions, i.e. partitions with one of the parts consisting of a single vertex
- `hodge_numbers` – boolean (default: `False`); if `False`, "-p" option will be passed to `nef.x` in order to skip Hodge numbers computation, which takes a lot of time

OUTPUT: a sequence of *nef-partitions*

Type `NefPartition?` for definitions and notation.

EXAMPLES:

Nef-partitions of the 4-dimensional cross-polytope:

```
sage: p = lattice_polytope.cross_polytope(4)
sage: p.nef_partitions()                                                    #␣
↪needs palp
```

(continues on next page)

(continued from previous page)

```
[Nef-partition {0, 1, 4, 5} ⊔ {2, 3, 6, 7} (direct product),
 Nef-partition {0, 1, 2, 4} ⊔ {3, 5, 6, 7},
 Nef-partition {0, 1, 2, 4, 5} ⊔ {3, 6, 7},
 Nef-partition {0, 1, 2, 4, 5, 6} ⊔ {3, 7} (direct product),
 Nef-partition {0, 1, 2, 3} ⊔ {4, 5, 6, 7},
 Nef-partition {0, 1, 2, 3, 4} ⊔ {5, 6, 7},
 Nef-partition {0, 1, 2, 3, 4, 5} ⊔ {6, 7},
 Nef-partition {0, 1, 2, 3, 4, 5, 6} ⊔ {7} (projection)]
```

```
>>> from sage.all import *
>>> p = lattice_polytope.cross_polytope(Integer(4))
>>> p.nef_partitions()                                                    #␣
↪needs palp
[Nef-partition {0, 1, 4, 5} ⊔ {2, 3, 6, 7} (direct product),
 Nef-partition {0, 1, 2, 4} ⊔ {3, 5, 6, 7},
 Nef-partition {0, 1, 2, 4, 5} ⊔ {3, 6, 7},
 Nef-partition {0, 1, 2, 4, 5, 6} ⊔ {3, 7} (direct product),
 Nef-partition {0, 1, 2, 3} ⊔ {4, 5, 6, 7},
 Nef-partition {0, 1, 2, 3, 4} ⊔ {5, 6, 7},
 Nef-partition {0, 1, 2, 3, 4, 5} ⊔ {6, 7},
 Nef-partition {0, 1, 2, 3, 4, 5, 6} ⊔ {7} (projection)]
```

Now we omit projections:

```
sage: p.nef_partitions(keep_projections=False)                              #␣
↪needs palp
[Nef-partition {0, 1, 4, 5} ⊔ {2, 3, 6, 7} (direct product),
 Nef-partition {0, 1, 2, 4} ⊔ {3, 5, 6, 7},
 Nef-partition {0, 1, 2, 4, 5} ⊔ {3, 6, 7},
 Nef-partition {0, 1, 2, 4, 5, 6} ⊔ {3, 7} (direct product),
 Nef-partition {0, 1, 2, 3} ⊔ {4, 5, 6, 7},
 Nef-partition {0, 1, 2, 3, 4} ⊔ {5, 6, 7},
 Nef-partition {0, 1, 2, 3, 4, 5} ⊔ {6, 7}]
```

```
>>> from sage.all import *
>>> p.nef_partitions(keep_projections=False)                              #␣
↪needs palp
[Nef-partition {0, 1, 4, 5} ⊔ {2, 3, 6, 7} (direct product),
 Nef-partition {0, 1, 2, 4} ⊔ {3, 5, 6, 7},
 Nef-partition {0, 1, 2, 4, 5} ⊔ {3, 6, 7},
 Nef-partition {0, 1, 2, 4, 5, 6} ⊔ {3, 7} (direct product),
 Nef-partition {0, 1, 2, 3} ⊔ {4, 5, 6, 7},
 Nef-partition {0, 1, 2, 3, 4} ⊔ {5, 6, 7},
 Nef-partition {0, 1, 2, 3, 4, 5} ⊔ {6, 7}]
```

Currently Hodge numbers cannot be computed for a given nef-partition:

```
sage: p.nef_partitions()[1].hodge_numbers()                                 #␣
↪needs palp
Traceback (most recent call last):
...
NotImplementedError: use nef_partitions(hodge_numbers=True)!
```

```
>>> from sage.all import *
>>> p.nef_partitions()[Integer(1)].hodge_numbers()                     ␣
↪     # needs palp
Traceback (most recent call last):
...
NotImplementedError: use nef_partitions(hodge_numbers=True)!
```

But they can be obtained from `nef.x` for all nef-partitions at once. Partitions will be exactly the same:

```
sage: p.nef_partitions(hodge_numbers=True)  # long time (2s on sage.math,␣
↪2011), needs palp
[Nef-partition {0, 1, 4, 5} ⊔ {2, 3, 6, 7} (direct product),
 Nef-partition {0, 1, 2, 4} ⊔ {3, 5, 6, 7},
 Nef-partition {0, 1, 2, 4, 5} ⊔ {3, 6, 7},
 Nef-partition {0, 1, 2, 4, 5, 6} ⊔ {3, 7} (direct product),
 Nef-partition {0, 1, 2, 3} ⊔ {4, 5, 6, 7},
 Nef-partition {0, 1, 2, 3, 4} ⊔ {5, 6, 7},
 Nef-partition {0, 1, 2, 3, 4, 5} ⊔ {6, 7},
 Nef-partition {0, 1, 2, 3, 4, 5, 6} ⊔ {7} (projection)]
```

```
>>> from sage.all import *
>>> p.nef_partitions(hodge_numbers=True)  # long time (2s on sage.math, 2011),
↪ needs palp
[Nef-partition {0, 1, 4, 5} ⊔ {2, 3, 6, 7} (direct product),
 Nef-partition {0, 1, 2, 4} ⊔ {3, 5, 6, 7},
 Nef-partition {0, 1, 2, 4, 5} ⊔ {3, 6, 7},
 Nef-partition {0, 1, 2, 4, 5, 6} ⊔ {3, 7} (direct product),
 Nef-partition {0, 1, 2, 3} ⊔ {4, 5, 6, 7},
 Nef-partition {0, 1, 2, 3, 4} ⊔ {5, 6, 7},
 Nef-partition {0, 1, 2, 3, 4, 5} ⊔ {6, 7},
 Nef-partition {0, 1, 2, 3, 4, 5, 6} ⊔ {7} (projection)]
```

Now it is possible to get Hodge numbers:

```
sage: p.nef_partitions(hodge_numbers=True)[1].hodge_numbers()              #␣
↪needs palp
(20,)
```

```
>>> from sage.all import *
>>> p.nef_partitions(hodge_numbers=True)[Integer(1)].hodge_numbers()       ␣
↪     # needs palp
(20,)
```

Since nef-partitions are cached, their Hodge numbers are accessible after the first request, even if you do not specify `hodge_numbers=True` anymore:

```
sage: p.nef_partitions()[1].hodge_numbers()                                #␣
↪needs palp
(20,)
```

```
>>> from sage.all import *
>>> p.nef_partitions()[Integer(1)].hodge_numbers()                     ␣
```

(continues on next page)

(continued from previous page)

```
↪     # needs palp
(20,)
```

We illustrate removal of symmetric partitions on a diamond:

```
sage: p = lattice_polytope.cross_polytope(2)
sage: p.nef_partitions()                                                    #␣
↪needs palp
[Nef-partition {0, 2} ⊔ {1, 3} (direct product),
 Nef-partition {0, 1} ⊔ {2, 3},
 Nef-partition {0, 1, 2} ⊔ {3} (projection)]
sage: p.nef_partitions(keep_symmetric=True)                                 #␣
↪needs palp
[Nef-partition {0, 1, 3} ⊔ {2} (projection),
 Nef-partition {0, 2, 3} ⊔ {1} (projection),
 Nef-partition {0, 3} ⊔ {1, 2},
 Nef-partition {1, 2, 3} ⊔ {0} (projection),
 Nef-partition {1, 3} ⊔ {0, 2} (direct product),
 Nef-partition {2, 3} ⊔ {0, 1},
 Nef-partition {0, 1, 2} ⊔ {3} (projection)]
```

```
>>> from sage.all import *
>>> p = lattice_polytope.cross_polytope(Integer(2))
>>> p.nef_partitions()                                                    #␣
↪needs palp
[Nef-partition {0, 2} ⊔ {1, 3} (direct product),
 Nef-partition {0, 1} ⊔ {2, 3},
 Nef-partition {0, 1, 2} ⊔ {3} (projection)]
>>> p.nef_partitions(keep_symmetric=True)                                 #␣
↪needs palp
[Nef-partition {0, 1, 3} ⊔ {2} (projection),
 Nef-partition {0, 2, 3} ⊔ {1} (projection),
 Nef-partition {0, 3} ⊔ {1, 2},
 Nef-partition {1, 2, 3} ⊔ {0} (projection),
 Nef-partition {1, 3} ⊔ {0, 2} (direct product),
 Nef-partition {2, 3} ⊔ {0, 1},
 Nef-partition {0, 1, 2} ⊔ {3} (projection)]
```

Nef-partitions can be computed only for reflexive polytopes:

```
sage: p = LatticePolytope([(1,0,0), (0,1,0), (0,0,2),
....:                      (-1,0,0), (0,-1,0), (0,0,-1)])
sage: p.nef_partitions()                                                    #␣
↪needs palp
Traceback (most recent call last):
...
ValueError: The given polytope is not reflexive!
Polytope: 3-d lattice polytope in 3-d lattice M
```

```
>>> from sage.all import *
>>> p = LatticePolytope([(Integer(1),Integer(0),Integer(0)), (Integer(0),
↪Integer(1),Integer(0)), (Integer(0),Integer(0),Integer(2)),
```

(continues on next page)

(continued from previous page)

```
...                          (-Integer(1),Integer(0),Integer(0)), (Integer(0),-
↪Integer(1),Integer(0)), (Integer(0),Integer(0),-Integer(1))])
>>> p.nef_partitions()                                                    #␣
↪needs palp
Traceback (most recent call last):
...
ValueError: The given polytope is not reflexive!
Polytope: 3-d lattice polytope in 3-d lattice M
```

nef_x(*keys*)

Run `nef.x` with given `keys` on vertices of this polytope.

INPUT:

- `keys` – string of options passed to `nef.x`; the key "-f" is added automatically

OUTPUT: the output of `nef.x` as a string

EXAMPLES: This call is used internally for computing nef-partitions:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: s = o.nef_x("-N -V -p")                                              #␣
↪needs palp
sage: s                         # output contains random time              #␣
↪needs palp
M:27 8 N:7 6  codim=2 #part=5
3 6  Vertices of P:
    1    0    0   -1    0    0
    0    1    0    0   -1    0
    0    0    1    0    0   -1
 P:0 V:2 4 5       0sec  0cpu
 P:2 V:3 4 5       0sec  0cpu
 P:3 V:4 5       0sec  0cpu
np=3 d:1 p:1    0sec     0cpu
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> s = o.nef_x("-N -V -p")                                               #␣
↪needs palp
>>> s                         # output contains random time               #␣
↪needs palp
M:27 8 N:7 6  codim=2 #part=5
3 6  Vertices of P:
    1    0    0   -1    0    0
    0    1    0    0   -1    0
    0    0    1    0    0   -1
 P:0 V:2 4 5       0sec  0cpu
 P:2 V:3 4 5       0sec  0cpu
 P:3 V:4 5       0sec  0cpu
np=3 d:1 p:1    0sec     0cpu
```

nfacets()

Return the number of facets of this polytope.

EXAMPLES: The number of facets of the 3-dimensional octahedron:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.nfacets()
8
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> o.nfacets()
8
```

The number of facets of an interval is 2:

```
sage: LatticePolytope(([1],[2])).nfacets()
2
```

```
>>> from sage.all import *
>>> LatticePolytope(([Integer(1)],[Integer(2)])).nfacets()
2
```

Now consider a 2-dimensional diamond in a 3-dimensional space:

```
sage: p = LatticePolytope([(1,0,0), (0,1,0), (-1,0,0), (0,-1,0)])
sage: p.nfacets()
4
```

```
>>> from sage.all import *
>>> p = LatticePolytope([(Integer(1),Integer(0),Integer(0)), (Integer(0),
↪Integer(1),Integer(0)), (-Integer(1),Integer(0),Integer(0)), (Integer(0),-
↪Integer(1),Integer(0))])
>>> p.nfacets()
4
```

normal_form(*algorithm='palp_native'*, *permutation=False*)

Return the normal form of vertices of `self`.

Two full-dimensional lattice polytopes are in the same $GL(\mathbf{Z}^n)$-orbit if and only if their normal forms are the same. Normal form is not defined and thus cannot be used for polytopes whose dimension is smaller than the dimension of the ambient space.

The original algorithm was presented in [KS1998] and implemented in PALP. A modified version of the PALP algorithm is discussed in [GK2013] and available here as `'palp_modified'`.

INPUT:

- `algorithm` – (default: `'palp_native'`) the algorithm which is used to compute the normal form. Options are:
  - `'palp'` – run external PALP code, usually the fastest option when it works; but reproducible crashes have been observed in dimension 5 and higher.
  - `'palp_native'` – the original PALP algorithm implemented in sage. Currently competitive with PALP in many cases.
  - `'palp_modified'` – a modified version of the PALP algorithm which determines the maximal vertex-facet pairing matrix first and then computes its automorphisms, while the PALP algorithm does both things concurrently.

- `permutation` – boolean (default: `False`); if `True`, the permutation applied to vertices to obtain the normal form is returned as well. Note that the different algorithms may return different results that nevertheless lead to the same normal form.

OUTPUT:

- a *point collection* in the *lattice()* of `self` or a tuple of it and a permutation.

EXAMPLES:

We compute the normal form of the "diamond":

```
sage: d = LatticePolytope([(1,0), (0,1), (-1,0), (0,-1)])
sage: d.vertices()
M( 1,  0),
M( 0,  1),
M(-1,  0),
M( 0, -1)
in 2-d lattice M
sage: d.normal_form()                                                    #␣
↪needs sage.groups
M( 1,  0),
M( 0,  1),
M( 0, -1),
M(-1,  0)
in 2-d lattice M
```

```
>>> from sage.all import *
>>> d = LatticePolytope([(Integer(1),Integer(0)), (Integer(0),Integer(1)), (-
↪Integer(1),Integer(0)), (Integer(0),-Integer(1))])
>>> d.vertices()
M( 1,  0),
M( 0,  1),
M(-1,  0),
M( 0, -1)
in 2-d lattice M
>>> d.normal_form()                                                     #␣
↪needs sage.groups
M( 1,  0),
M( 0,  1),
M( 0, -1),
M(-1,  0)
in 2-d lattice M
```

The diamond is the 3rd polytope in the internal database:

```
sage: d.index()                                                          #␣
↪needs palp
3
sage: d                                                                  #␣
↪needs palp
2-d reflexive polytope #3 in 2-d lattice M
```

```
>>> from sage.all import *
>>> d.index()                                                           #␣
```

(continues on next page)

(continued from previous page)

```
↪needs palp
3
>>> d                                                                    #␣
↪needs palp
2-d reflexive polytope #3 in 2-d lattice M
```

You can get it in its normal form (in the default lattice) as

```
sage: lattice_polytope.ReflexivePolytope(2, 3).vertices()
M( 1,  0),
M( 0,  1),
M( 0, -1),
M(-1,  0)
in 2-d lattice M
```

```
>>> from sage.all import *
>>> lattice_polytope.ReflexivePolytope(Integer(2), Integer(3)).vertices()
M( 1,  0),
M( 0,  1),
M( 0, -1),
M(-1,  0)
in 2-d lattice M
```

It is not possible to compute normal forms for polytopes which do not span the space:

```
sage: p = LatticePolytope([(1,0,0), (0,1,0), (-1,0,0), (0,-1,0)])
sage: p.normal_form()
Traceback (most recent call last):
...
ValueError: normal form is not defined for
2-d lattice polytope in 3-d lattice M
```

```
>>> from sage.all import *
>>> p = LatticePolytope([(Integer(1),Integer(0),Integer(0)), (Integer(0),
↪Integer(1),Integer(0)), (-Integer(1),Integer(0),Integer(0)), (Integer(0),-
↪Integer(1),Integer(0))])
>>> p.normal_form()
Traceback (most recent call last):
...
ValueError: normal form is not defined for
2-d lattice polytope in 3-d lattice M
```

We can perform the same examples using other algorithms:

```
sage: o = lattice_polytope.cross_polytope(2)
sage: o.normal_form(algorithm='palp_native')                              #␣
↪needs sage.groups
M( 1,  0),
M( 0,  1),
M( 0, -1),
M(-1,  0)
in 2-d lattice M
```

(continues on next page)

(continued from previous page)

```
sage: o = lattice_polytope.cross_polytope(2)
sage: o.normal_form(algorithm='palp_modified')                              #␣
↪needs sage.groups
M( 1,  0),
M( 0,  1),
M( 0, -1),
M(-1,  0)
in 2-d lattice M
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(2))
>>> o.normal_form(algorithm='palp_native')                                  #␣
↪needs sage.groups
M( 1,  0),
M( 0,  1),
M( 0, -1),
M(-1,  0)
in 2-d lattice M

>>> o = lattice_polytope.cross_polytope(Integer(2))
>>> o.normal_form(algorithm='palp_modified')                                #␣
↪needs sage.groups
M( 1,  0),
M( 0,  1),
M( 0, -1),
M(-1,  0)
in 2-d lattice M
```

The following examples demonstrate the speed of the available algorithms. In low dimensions, the default algorithm, `'palp_native'`, is the fastest. As the dimension increases, `'palp'` is relatively faster than `'palp_native'`. `'palp_native'` is usually much faster than `'palp_modified'`. In some cases when the polytope has high symmetry, however, `'palp_native'` is slower:

```
sage: # not tested
sage: o = lattice_polytope.cross_polytope(2)
sage: %timeit o.normal_form.clear_cache(); o.normal_form("palp")
625 loops, best of 3: 3.07 ms per loop
sage: %timeit o.normal_form.clear_cache(); o.normal_form("palp_native")
625 loops, best of 3: 0.445 ms per loop
sage: %timeit o.normal_form.clear_cache(); o.normal_form("palp_modified")
625 loops, best of 3: 5.01 ms per loop
sage: o = lattice_polytope.cross_polytope(3)
sage: %timeit o.normal_form.clear_cache(); o.normal_form("palp")
625 loops, best of 3: 3.22 ms per loop
sage: %timeit o.normal_form.clear_cache(); o.normal_form("palp_native")
625 loops, best of 3: 2.73 ms per loop
sage: %timeit o.normal_form.clear_cache(); o.normal_form("palp_modified")
625 loops, best of 3: 20.7 ms per loop
sage: o = lattice_polytope.cross_polytope(4)
sage: %timeit o.normal_form.clear_cache(); o.normal_form("palp")
625 loops, best of 3: 4.84 ms per loop
```

(continues on next page)

(continued from previous page)

```
sage: %timeit o.normal_form.clear_cache(); o.normal_form("palp_native")
625 loops, best of 3: 55.6 ms per loop
sage: %timeit o.normal_form.clear_cache(); o.normal_form("palp_modified")
625 loops, best of 3: 129 ms per loop
sage: o = lattice_polytope.cross_polytope(5)
sage: %timeit o.normal_form.clear_cache(); o.normal_form("palp")
10 loops, best of 3: 0.0364 s per loop
sage: %timeit o.normal_form.clear_cache(); o.normal_form("palp_native")
10 loops, best of 3: 1.68 s per loop
sage: %timeit o.normal_form.clear_cache(); o.normal_form("palp_modified")
10 loops, best of 3: 0.858 s per loop
```

```
>>> from sage.all import *
>>> # not tested
>>> o = lattice_polytope.cross_polytope(Integer(2))
>>> %timeit o.normal_form.clear_cache(); o.normal_form("palp")
625 loops, best of 3: 3.07 ms per loop
>>> %timeit o.normal_form.clear_cache(); o.normal_form("palp_native")
625 loops, best of 3: 0.445 ms per loop
>>> %timeit o.normal_form.clear_cache(); o.normal_form("palp_modified")
625 loops, best of 3: 5.01 ms per loop
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> %timeit o.normal_form.clear_cache(); o.normal_form("palp")
625 loops, best of 3: 3.22 ms per loop
>>> %timeit o.normal_form.clear_cache(); o.normal_form("palp_native")
625 loops, best of 3: 2.73 ms per loop
>>> %timeit o.normal_form.clear_cache(); o.normal_form("palp_modified")
625 loops, best of 3: 20.7 ms per loop
>>> o = lattice_polytope.cross_polytope(Integer(4))
>>> %timeit o.normal_form.clear_cache(); o.normal_form("palp")
625 loops, best of 3: 4.84 ms per loop
>>> %timeit o.normal_form.clear_cache(); o.normal_form("palp_native")
625 loops, best of 3: 55.6 ms per loop
>>> %timeit o.normal_form.clear_cache(); o.normal_form("palp_modified")
625 loops, best of 3: 129 ms per loop
>>> o = lattice_polytope.cross_polytope(Integer(5))
>>> %timeit o.normal_form.clear_cache(); o.normal_form("palp")
10 loops, best of 3: 0.0364 s per loop
>>> %timeit o.normal_form.clear_cache(); o.normal_form("palp_native")
10 loops, best of 3: 1.68 s per loop
>>> %timeit o.normal_form.clear_cache(); o.normal_form("palp_modified")
10 loops, best of 3: 0.858 s per loop
```

Note that the algorithm `'palp'` may crash for higher dimensions because of the overflow errors as mentioned in Issue #13525#comment:9. Then use `'palp_native'` instead, which is usually faster than `'palp_modified'`. Below is an example where `'palp'` fails and `'palp_native'` is much faster than `'palp_modified'`:

```
sage: P = LatticePolytope([[-3, -3, -6, -6, -1], [3, 3, 6, 6, 1], [-3, -3, -6,
↪ -6, 1],
....:                      [-3, -3, -3, -6, 0], [-3, -3, -3, 0, 0], [-3, -3,␣
↪0, 0, 0],
```

(continues on next page)

(continued from previous page)

```
....:                   [-3, 0, -6, -6, 0], [-3, 0, -3, -6, 0], [-3, 0, -3,
↪ 0, 0],
....:                   [-3, 0, 0, 0, -1], [3, 3, 6, 6, -1], [-3, 0, 0, 0,␣
↪1],
....:                   [0, -3, -6, -6, 0], [0, -3, -3, -6, 0], [0, -3, -3,
↪ 0, 0],
....:                   [0, -3, 0, 0, -1], [3, 3, 3, 6, 0], [0, -3, 0, 0,␣
↪1],
....:                   [0, 0, -6, -6, 0], [0, 0, -3, -6, -1], [3, 3, 3, 0,
↪ 0],
....:                   [0, 0, -3, -6, 1], [0, 0, -3, 0, -1], [3, 3, 0, 0,␣
↪0],
....:                   [0, 0, -3, 0, 1], [0, 0, 3, 0, -1], [3, 0, 6, 6,␣
↪0],
....:                   [0, 0, 3, 0, 1], [0, 0, 3, 6, -1], [3, 0, 3, 6, 0],
....:                   [0, 0, 3, 6, 1], [0, 0, 6, 6, 0], [0, 3, 0, 0, -1],
....:                   [3, 0, 3, 0, 0], [0, 3, 0, 0, 1], [0, 3, 3, 0, 0],
....:                   [0, 3, 3, 6, 0], [0, 3, 6, 6, 0], [3, 0,0, 0, -1],␣
↪[3, 0, 0, 0, 1]])
sage: P.normal_form(algorithm='palp')  # not tested
Traceback (most recent call last):
...
RuntimeError: Error executing ... for a polytope sequence!
Output:
b'*** stack smashing detected ***: terminated\nAborted\n'
sage: P.normal_form(algorithm='palp_native')                              #␣
↪needs sage.groups
M(  6,  0,  0,  0,  0),
M( -6,  0,  0,  0,  0),
M(  0,  1,  0,  0,  0),
M(  0,  0,  3,  0,  0),
M(  0,  1,  0,  3,  0),
M(  0,  0,  0,  0,  3),
M( -6,  1,  6,  3, -6),
M( -6,  0,  6,  0, -3),
M(-12,  1,  6,  3, -3),
M( -6,  1,  0,  3,  0),
M( -6,  0,  3,  3,  0),
M(  6,  0, -6, -3,  6),
M(-12,  1,  6,  3, -6),
M(-12,  0,  9,  3, -6),
M(  0,  0,  0, -3,  0),
M(-12,  1,  6,  6, -6),
M(-12,  0,  6,  3, -3),
M(  0,  1, -3,  0,  0),
M(  0,  0, -3, -3,  3),
M(  0,  1,  0,  3, -3),
M(  0, -1,  0, -3,  3),
M(  0,  0,  3,  3, -3),
M(  0, -1,  3,  0,  0),
M( 12,  0, -6, -3,  3),
M( 12, -1, -6, -6,  6),
```

(continues on next page)

(continued from previous page)

```
M(  0,  0,  0,  3,  0),
M( 12,  0, -9, -3,  6),
M( 12, -1, -6, -3,  6),
M( -6,  0,  6,  3, -6),
M(  6,  0, -3, -3,  0),
M(  6, -1,  0, -3,  0),
M(-12,  1,  9,  6, -6),
M(  6,  0, -6,  0,  3),
M(  6, -1, -6, -3,  6),
M(  0,  0,  0,  0, -3),
M(  0, -1,  0, -3,  0),
M(  0,  0, -3,  0,  0),
M(  0, -1,  0,  0,  0),
M( 12, -1, -9, -6,  6),
M( 12, -1, -6, -3,  3)
in 5-d lattice M
sage: P.normal_form(algorithm='palp_modified')      # not tested (22s;␣
↪MemoryError on 32 bit), needs sage.groups
M(  6,  0,  0,  0,  0),
M( -6,  0,  0,  0,  0),
M(  0,  1,  0,  0,  0),
M(  0,  0,  3,  0,  0),
M(  0,  1,  0,  3,  0),
M(  0,  0,  0,  0,  3),
M( -6,  1,  6,  3, -6),
M( -6,  0,  6,  0, -3),
M(-12,  1,  6,  3, -3),
M( -6,  1,  0,  3,  0),
M( -6,  0,  3,  3,  0),
M(  6,  0, -6, -3,  6),
M(-12,  1,  6,  3, -6),
M(-12,  0,  9,  3, -6),
M(  0,  0,  0, -3,  0),
M(-12,  1,  6,  6, -6),
M(-12,  0,  6,  3, -3),
M(  0,  1, -3,  0,  0),
M(  0,  0, -3, -3,  3),
M(  0,  1,  0,  3, -3),
M(  0, -1,  0, -3,  3),
M(  0,  0,  3,  3, -3),
M(  0, -1,  3,  0,  0),
M( 12,  0, -6, -3,  3),
M( 12, -1, -6, -6,  6),
M(  0,  0,  0,  3,  0),
M( 12,  0, -9, -3,  6),
M( 12, -1, -6, -3,  6),
M( -6,  0,  6,  3, -6),
M(  6,  0, -3, -3,  0),
M(  6, -1,  0, -3,  0),
M(-12,  1,  9,  6, -6),
M(  6,  0, -6,  0,  3),
M(  6, -1, -6, -3,  6),
```

(continues on next page)

(continued from previous page)

```
M(  0,  0,  0,  0, -3),
M(  0, -1,  0, -3,  0),
M(  0,  0, -3,  0,  0),
M(  0, -1,  0,  0,  0),
M( 12, -1, -9, -6,  6),
M( 12, -1, -6, -3,  3)
in 5-d lattice M
sage: %timeit P.normal_form.clear_cache(); P.normal_form("palp_native")    #␣
↪not tested
10 loops, best of 3: 0.137 s per loop
sage: %timeit P.normal_form.clear_cache(); P.normal_form("palp_modified")  #␣
↪not tested
10 loops, best of 3:  22.2 s per loop
```

```
>>> from sage.all import *
>>> P = LatticePolytope([[-Integer(3), -Integer(3), -Integer(6), -Integer(6),␣
↪-Integer(1)], [Integer(3), Integer(3), Integer(6), Integer(6), Integer(1)],␣
↪[-Integer(3), -Integer(3), -Integer(6), -Integer(6), Integer(1)],
...                      [-Integer(3), -Integer(3), -Integer(3), -Integer(6),␣
↪Integer(0)], [-Integer(3), -Integer(3), -Integer(3), Integer(0),␣
↪Integer(0)], [-Integer(3), -Integer(3), Integer(0), Integer(0), Integer(0)],
...                      [-Integer(3), Integer(0), -Integer(6), -Integer(6),␣
↪Integer(0)], [-Integer(3), Integer(0), -Integer(3), -Integer(6),␣
↪Integer(0)], [-Integer(3), Integer(0), -Integer(3), Integer(0), Integer(0)],
...                      [-Integer(3), Integer(0), Integer(0), Integer(0), -
↪Integer(1)], [Integer(3), Integer(3), Integer(6), Integer(6), -Integer(1)],␣
↪[-Integer(3), Integer(0), Integer(0), Integer(0), Integer(1)],
...                      [Integer(0), -Integer(3), -Integer(6), -Integer(6),␣
↪Integer(0)], [Integer(0), -Integer(3), -Integer(3), -Integer(6),␣
↪Integer(0)], [Integer(0), -Integer(3), -Integer(3), Integer(0), Integer(0)],
...                      [Integer(0), -Integer(3), Integer(0), Integer(0), -
↪Integer(1)], [Integer(3), Integer(3), Integer(3), Integer(6), Integer(0)],␣
↪[Integer(0), -Integer(3), Integer(0), Integer(0), Integer(1)],
...                      [Integer(0), Integer(0), -Integer(6), -Integer(6),␣
↪Integer(0)], [Integer(0), Integer(0), -Integer(3), -Integer(6), -
↪Integer(1)], [Integer(3), Integer(3), Integer(3), Integer(0), Integer(0)],
...                      [Integer(0), Integer(0), -Integer(3), -Integer(6),␣
↪Integer(1)], [Integer(0), Integer(0), -Integer(3), Integer(0), -Integer(1)],
↪ [Integer(3), Integer(3), Integer(0), Integer(0), Integer(0)],
...                      [Integer(0), Integer(0), -Integer(3), Integer(0),␣
↪Integer(1)], [Integer(0), Integer(0), Integer(3), Integer(0), -Integer(1)],␣
↪[Integer(3), Integer(0), Integer(6), Integer(6), Integer(0)],
...                      [Integer(0), Integer(0), Integer(3), Integer(0),␣
↪Integer(1)], [Integer(0), Integer(0), Integer(3), Integer(6), -Integer(1)],␣
↪[Integer(3), Integer(0), Integer(3), Integer(6), Integer(0)],
...                      [Integer(0), Integer(0), Integer(3), Integer(6),␣
↪Integer(1)], [Integer(0), Integer(0), Integer(6), Integer(6), Integer(0)],␣
↪[Integer(0), Integer(3), Integer(0), Integer(0), -Integer(1)],
...                      [Integer(3), Integer(0), Integer(3), Integer(0),␣
↪Integer(0)], [Integer(0), Integer(3), Integer(0), Integer(0), Integer(1)],␣
↪[Integer(0), Integer(3), Integer(3), Integer(0), Integer(0)],
...                      [Integer(0), Integer(3), Integer(3), Integer(6),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(0)], [Integer(0), Integer(3), Integer(6), Integer(6), Integer(0)],␣
↪[Integer(3), Integer(0),Integer(0), Integer(0), -Integer(1)], [Integer(3),␣
↪Integer(0), Integer(0), Integer(0), Integer(1)]])
>>> P.normal_form(algorithm='palp')  # not tested
Traceback (most recent call last):
...
RuntimeError: Error executing ... for a polytope sequence!
Output:
b'*** stack smashing detected ***: terminated\nAborted\n'
>>> P.normal_form(algorithm='palp_native')                          #␣
↪needs sage.groups
M(  6,  0,  0,  0,  0),
M( -6,  0,  0,  0,  0),
M(  0,  1,  0,  0,  0),
M(  0,  0,  3,  0,  0),
M(  0,  1,  0,  3,  0),
M(  0,  0,  0,  0,  3),
M( -6,  1,  6,  3, -6),
M( -6,  0,  6,  0, -3),
M(-12,  1,  6,  3, -3),
M( -6,  1,  0,  3,  0),
M( -6,  0,  3,  3,  0),
M(  6,  0, -6, -3,  6),
M(-12,  1,  6,  3, -6),
M(-12,  0,  9,  3, -6),
M(  0,  0,  0, -3,  0),
M(-12,  1,  6,  6, -6),
M(-12,  0,  6,  3, -3),
M(  0,  1, -3,  0,  0),
M(  0,  0, -3, -3,  3),
M(  0,  1,  0,  3, -3),
M(  0, -1,  0, -3,  3),
M(  0,  0,  3,  3, -3),
M(  0, -1,  3,  0,  0),
M( 12,  0, -6, -3,  3),
M( 12, -1, -6, -6,  6),
M(  0,  0,  0,  3,  0),
M( 12,  0, -9, -3,  6),
M( 12, -1, -6, -3,  6),
M( -6,  0,  6,  3, -6),
M(  6,  0, -3, -3,  0),
M(  6, -1,  0, -3,  0),
M(-12,  1,  9,  6, -6),
M(  6,  0, -6,  0,  3),
M(  6, -1, -6, -3,  6),
M(  0,  0,  0,  0, -3),
M(  0, -1,  0, -3,  0),
M(  0,  0, -3,  0,  0),
M(  0, -1,  0,  0,  0),
M( 12, -1, -9, -6,  6),
M( 12, -1, -6, -3,  3)
in 5-d lattice M
```

(continues on next page)

(continued from previous page)

```
>>> P.normal_form(algorithm='palp_modified')      # not tested (22s;␣
↪MemoryError on 32 bit), needs sage.groups
M(  6,  0,  0,  0,  0),
M( -6,  0,  0,  0,  0),
M(  0,  1,  0,  0,  0),
M(  0,  0,  3,  0,  0),
M(  0,  1,  0,  3,  0),
M(  0,  0,  0,  0,  3),
M( -6,  1,  6,  3, -6),
M( -6,  0,  6,  0, -3),
M(-12,  1,  6,  3, -3),
M( -6,  1,  0,  3,  0),
M( -6,  0,  3,  3,  0),
M(  6,  0, -6, -3,  6),
M(-12,  1,  6,  3, -6),
M(-12,  0,  9,  3, -6),
M(  0,  0,  0, -3,  0),
M(-12,  1,  6,  6, -6),
M(-12,  0,  6,  3, -3),
M(  0,  1, -3,  0,  0),
M(  0,  0, -3, -3,  3),
M(  0,  1,  0,  3, -3),
M(  0, -1,  0, -3,  3),
M(  0,  0,  3,  3, -3),
M(  0, -1,  3,  0,  0),
M( 12,  0, -6, -3,  3),
M( 12, -1, -6, -6,  6),
M(  0,  0,  0,  3,  0),
M( 12,  0, -9, -3,  6),
M( 12, -1, -6, -3,  6),
M( -6,  0,  6,  3, -6),
M(  6,  0, -3, -3,  0),
M(  6, -1,  0, -3,  0),
M(-12,  1,  9,  6, -6),
M(  6,  0, -6,  0,  3),
M(  6, -1, -6, -3,  6),
M(  0,  0,  0,  0, -3),
M(  0, -1,  0, -3,  0),
M(  0,  0, -3,  0,  0),
M(  0, -1,  0,  0,  0),
M( 12, -1, -9, -6,  6),
M( 12, -1, -6, -3,  3)
in 5-d lattice M
>>> %timeit P.normal_form.clear_cache(); P.normal_form("palp_native")    #␣
↪not tested
10 loops, best of 3: 0.137 s per loop
>>> %timeit P.normal_form.clear_cache(); P.normal_form("palp_modified")  #␣
↪not tested
10 loops, best of 3:  22.2 s per loop
```

npoints()

Return the number of lattice points of this polytope.

EXAMPLES: The number of lattice points of the 3-dimensional octahedron and its polar cube:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.npoints()                                                    #␣
↪needs palp
7
sage: cube = o.polar()
sage: cube.npoints()                                                 #␣
↪needs palp
27
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> o.npoints()                                                    #␣
↪needs palp
7
>>> cube = o.polar()
>>> cube.npoints()                                                 #␣
↪needs palp
27
```

nvertices()

Return the number of vertices of this polytope.

EXAMPLES: The number of vertices of the 3-dimensional octahedron and its polar cube:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.nvertices()
6
sage: cube = o.polar()
sage: cube.nvertices()
8
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> o.nvertices()
6
>>> cube = o.polar()
>>> cube.nvertices()
8
```

origin()

Return the index of the origin in the list of points of `self`.

OUTPUT: integer if the origin belongs to this polytope, `None` otherwise

EXAMPLES:

```
sage: p = lattice_polytope.cross_polytope(2)
sage: p.origin()                                                     #␣
↪needs palp
4
sage: p.point(p.origin())                                            #␣
↪needs palp
```

(continues on next page)

(continued from previous page)

```
M(0, 0)

sage: p = LatticePolytope(([1],[2]))
sage: p.points()
M(1),
M(2)
in 1-d lattice M
sage: print(p.origin())
None
```

```
>>> from sage.all import *
>>> p = lattice_polytope.cross_polytope(Integer(2))
>>> p.origin()                                                              #␣
↪needs palp
4
>>> p.point(p.origin())                                                     #␣
↪needs palp
M(0, 0)

>>> p = LatticePolytope(([Integer(1)],[Integer(2)]))
>>> p.points()
M(1),
M(2)
in 1-d lattice M
>>> print(p.origin())
None
```

Now we make sure that the origin of non-full-dimensional polytopes can be identified correctly (Issue #10661):

```
sage: LatticePolytope([(1,0,0), (-1,0,0)]).origin()
2
```

```
>>> from sage.all import *
>>> LatticePolytope([(Integer(1),Integer(0),Integer(0)), (-Integer(1),
↪Integer(0),Integer(0))]).origin()
2
```

parent()

Return the set of all lattice polytopes.

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.parent()
Set of all Lattice Polytopes
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> o.parent()
Set of all Lattice Polytopes
```

plot3d(*show_facets=True*, *facet_opacity=0.5*, *facet_color=(0, 1, 0)*, *facet_colors=None*, *show_edges=True*, *edge_thickness=3*, *edge_color=(0.5, 0.5, 0.5)*, *show_vertices=True*, *vertex_size=10*, *vertex_color=(1, 0, 0)*, *show_points=True*, *point_size=10*, *point_color=(0, 0, 1)*, *show_vindices=None*, *vindex_color=(0, 0, 0)*, *vlabels=None*, *show_pindices=None*, *pindex_color=(0, 0, 0)*, *index_shift=1.1*)

Return a 3d-plot of this polytope.

Polytopes with ambient dimension 1 and 2 will be plotted along x-axis or in xy-plane respectively. Polytopes of dimension 3 and less with ambient dimension 4 and greater will be plotted in some basis of the spanned space.

By default, everything is shown with more or less pretty combination of size and color parameters.

INPUT:

Most of the parameters are self-explanatory:

- `show_facets` – (default: `True`)
- `facet_opacity` – (default:0.5)
- `facet_color` – (default:(0,1,0))
- `facet_colors` – (default:None) if specified, must be a list of colors for each facet separately, used instead of `facet_color`
- `show_edges` – boolean (default: `True`); whether to draw edges as lines
- `edge_thickness` – (default:3)
- `edge_color` – (default:(0.5,0.5,0.5))
- `show_vertices` – boolean (default: `True`); whether to draw vertices as balls
- `vertex_size` – (default:10)
- `vertex_color` – (default:(1,0,0))
- `show_points` – boolean (default: `True`); whether to draw other points as balls
- `point_size` – (default:10)
- `point_color` – (default:(0,0,1))
- `show_vindices` – (default: same as `show_vertices`) whether to show indices of vertices
- `vindex_color` – (default:(0,0,0)) color for vertex labels
- `vlabels` – (default:None) if specified, must be a list of labels for each vertex, default labels are vertex indices
- `show_pindices` – (default: same as `show_points`) whether to show indices of other points
- `pindex_color` – (default:(0,0,0)) color for point labels
- `index_shift` – (default:1.1)) if 1, labels are placed exactly at the corresponding points. Otherwise the label position is computed as a multiple of the point position vector.

EXAMPLES: The default plot of a cube:

```
sage: c = lattice_polytope.cross_polytope(3).polar()
sage: c.plot3d()                                                          #␣
↪needs palp sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> c = lattice_polytope.cross_polytope(Integer(3)).polar()
>>> c.plot3d()                                                               #␣
↪needs palp sage.plot
Graphics3d Object
```

Plot without facets and points, shown without the frame:

```
sage: c.plot3d(show_facets=false,                                              #␣
↪needs palp sage.plot
....:          show_points=false).show(frame=False)
```

```
>>> from sage.all import *
>>> c.plot3d(show_facets=false,                                              #␣
↪needs palp sage.plot
...          show_points=false).show(frame=False)
```

Plot with facets of different colors:

```
sage: c.plot3d(facet_colors=rainbow(c.nfacets(), 'rgbtuple'))                  #␣
↪needs palp sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> c.plot3d(facet_colors=rainbow(c.nfacets(), 'rgbtuple'))                  #␣
↪needs palp sage.plot
Graphics3d Object
```

It is also possible to plot lower dimensional polytops in 3D (let's also change labels of vertices):

```
sage: c2 = lattice_polytope.cross_polytope(2)
sage: c2.plot3d(vlabels=["A", "B", "C", "D"])                                  #␣
↪needs palp sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> c2 = lattice_polytope.cross_polytope(Integer(2))
>>> c2.plot3d(vlabels=["A", "B", "C", "D"])                                  #␣
↪needs palp sage.plot
Graphics3d Object
```

point(*i*)

Return the i-th point of this polytope, i.e. the i-th column of the matrix returned by `points()`.

EXAMPLES: First few points are actually vertices:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.vertices()
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
```

(continues on next page)

(continued from previous page)

```
M( 0,  0, -1)
in 3-d lattice M
sage: o.point(1)                                                         #␣
↪needs palp
M(0, 1, 0)
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> o.vertices()
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1)
in 3-d lattice M
>>> o.point(Integer(1))                                                   ␣
↪      # needs palp
M(0, 1, 0)
```

The only other point in the octahedron is the origin:

```
sage: o.point(6)                                                         #␣
↪needs palp
M(0, 0, 0)
sage: o.points()                                                         #␣
↪needs palp
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1),
M( 0,  0,  0)
in 3-d lattice M
```

```
>>> from sage.all import *
>>> o.point(Integer(6))                                                   ␣
↪      # needs palp
M(0, 0, 0)
>>> o.points()                                                          #␣
↪needs palp
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1),
M( 0,  0,  0)
in 3-d lattice M
```

points(**args*, ***kwds*)

Return all lattice points of `self`.

INPUT:

- any arguments given will be passed on to the returned object.

OUTPUT: a *point collection*

EXAMPLES:

Lattice points of the octahedron and its polar cube:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.points()                                                        #␣
↪needs palp
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1),
M( 0,  0,  0)
in 3-d lattice M
sage: cube = o.polar()
sage: cube.points()                                                     #␣
↪needs palp
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),
N(-1, -1,  0),
N(-1,  0, -1),
N(-1,  0,  0),
N(-1,  0,  1),
N(-1,  1,  0),
N( 0, -1, -1),
N( 0, -1,  0),
N( 0, -1,  1),
N( 0,  0, -1),
N( 0,  0,  0),
N( 0,  0,  1),
N( 0,  1, -1),
N( 0,  1,  0),
N( 0,  1,  1),
N( 1, -1,  0),
N( 1,  0, -1),
N( 1,  0,  0),
N( 1,  0,  1),
N( 1,  1,  0)
in 3-d lattice N
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> o.points()                                                         #␣
↪needs palp
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1),
M( 0,  0,  0)
in 3-d lattice M
>>> cube = o.polar()
>>> cube.points()                                                      #␣
↪needs palp
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),
N(-1, -1,  0),
N(-1,  0, -1),
N(-1,  0,  0),
N(-1,  0,  1),
N(-1,  1,  0),
N( 0, -1, -1),
N( 0, -1,  0),
N( 0, -1,  1),
N( 0,  0, -1),
N( 0,  0,  0),
N( 0,  0,  1),
N( 0,  1, -1),
N( 0,  1,  0),
N( 0,  1,  1),
N( 1, -1,  0),
N( 1,  0, -1),
N( 1,  0,  0),
N( 1,  0,  1),
N( 1,  1,  0)
in 3-d lattice N
```

Lattice points of a 2-dimensional diamond in a 3-dimensional space:

```
sage: p = LatticePolytope([(1,0,0), (0,1,0), (-1,0,0), (0,-1,0)])
sage: p.points()                                                         #␣
↪needs palp
M( 1,  0, 0),
M( 0,  1, 0),
M(-1,  0, 0),
M( 0, -1, 0),
```

(continues on next page)

(continued from previous page)

```
M( 0,  0, 0)
in 3-d lattice M
```

```
>>> from sage.all import *
>>> p = LatticePolytope([(Integer(1),Integer(0),Integer(0)), (Integer(0),
↪Integer(1),Integer(0)), (-Integer(1),Integer(0),Integer(0)), (Integer(0),-
↪Integer(1),Integer(0))])
>>> p.points()                                                          #␣
↪needs palp
M( 1,  0, 0),
M( 0,  1, 0),
M(-1,  0, 0),
M( 0, -1, 0),
M( 0,  0, 0)
in 3-d lattice M
```

Only two of the above points:

```
sage: p.points(1, 3)                                                    #␣
↪needs palp
M(0,  1, 0),
M(0, -1, 0)
in 3-d lattice M
```

```
>>> from sage.all import *
>>> p.points(Integer(1), Integer(3))                                     ␣
↪                 # needs palp
M(0,  1, 0),
M(0, -1, 0)
in 3-d lattice M
```

We check that points of a zero-dimensional polytope can be computed:

```
sage: p = LatticePolytope([[1]])
sage: p.points()
M(1)
in 1-d lattice M
```

```
>>> from sage.all import *
>>> p = LatticePolytope([[Integer(1)]])
>>> p.points()
M(1)
in 1-d lattice M
```

polar()

Return the polar polytope, if this polytope is reflexive.

EXAMPLES: The polar polytope to the 3-dimensional octahedron:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: cube = o.polar()
```

(continues on next page)

(continued from previous page)

```
sage: cube
3-d reflexive polytope in 3-d lattice N
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> cube = o.polar()
>>> cube
3-d reflexive polytope in 3-d lattice N
```

The polar polytope "remembers" the original one:

```
sage: cube.polar()
3-d reflexive polytope in 3-d lattice M
sage: cube.polar().polar() is cube
True
```

```
>>> from sage.all import *
>>> cube.polar()
3-d reflexive polytope in 3-d lattice M
>>> cube.polar().polar() is cube
True
```

Only reflexive polytopes have polars:

```
sage: p = LatticePolytope([(1,0,0), (0,1,0), (0,0,2),
....:                      (-1,0,0), (0,-1,0), (0,0,-1)])
sage: p.polar()
Traceback (most recent call last):
...
ValueError: The given polytope is not reflexive!
Polytope: 3-d lattice polytope in 3-d lattice M
```

```
>>> from sage.all import *
>>> p = LatticePolytope([(Integer(1),Integer(0),Integer(0)), (Integer(0),
↪Integer(1),Integer(0)), (Integer(0),Integer(0),Integer(2)),
...                      (-Integer(1),Integer(0),Integer(0)), (Integer(0),-
↪Integer(1),Integer(0)), (Integer(0),Integer(0),-Integer(1))])
>>> p.polar()
Traceback (most recent call last):
...
ValueError: The given polytope is not reflexive!
Polytope: 3-d lattice polytope in 3-d lattice M
```

poly_x(*keys*, *reduce_dimension=False*)

Run `poly.x` with given `keys` on vertices of this polytope.

INPUT:

- `keys` – string of options passed to `poly.x`. The key "f" is added automatically
- `reduce_dimension` – boolean (default: `False`); if `True` and this polytope is not full-dimensional, `poly.x` will be called for the vertices of this polytope in some basis of the spanned affine space

OUTPUT: the output of `poly.x` as a string

EXAMPLES: This call is used for determining if a polytope is reflexive or not:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: print(o.poly_x("e"))                                                   #␣
↪needs palp
8 3  Vertices of P-dual <-> Equations of P
  -1  -1   1
   1  -1   1
  -1   1   1
   1   1   1
  -1  -1  -1
   1  -1  -1
  -1   1  -1
   1   1  -1
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> print(o.poly_x("e"))                                                    #␣
↪needs palp
8 3  Vertices of P-dual <-> Equations of P
  -1  -1   1
   1  -1   1
  -1   1   1
   1   1   1
  -1  -1  -1
   1  -1  -1
  -1   1  -1
   1   1  -1
```

Since PALP has limits on different parameters determined during compilation, the following code is likely to fail, unless you change default settings of PALP:

```
sage: BIG = lattice_polytope.cross_polytope(7)
sage: BIG
7-d reflexive polytope in 7-d lattice M
sage: BIG.poly_x("e")                                                        #␣
↪needs palp
Traceback (most recent call last):
...
ValueError: Error executing 'poly.x -fe' for the given polytope!
Output:
Please increase POLY_Dmax to at least 7
```

```
>>> from sage.all import *
>>> BIG = lattice_polytope.cross_polytope(Integer(7))
>>> BIG
7-d reflexive polytope in 7-d lattice M
>>> BIG.poly_x("e")                                                         #␣
↪needs palp
Traceback (most recent call last):
...
ValueError: Error executing 'poly.x -fe' for the given polytope!
Output:
```

(continues on next page)

(continued from previous page)

```
Please increase POLY_Dmax to at least 7
```

You cannot call `poly.x` for polytopes that don't span the space (if you could, it would crush anyway):

```
sage: p = LatticePolytope([(1,0,0), (0,1,0), (-1,0,0), (0,-1,0)])
sage: p.poly_x("e")                                                    #␣
↪needs palp
Traceback (most recent call last):
...
ValueError: Cannot run PALP for a 2-dimensional polytope in a 3-dimensional␣
↪space!
```

```
>>> from sage.all import *
>>> p = LatticePolytope([(Integer(1),Integer(0),Integer(0)), (Integer(0),
↪Integer(1),Integer(0)), (-Integer(1),Integer(0),Integer(0)), (Integer(0),-
↪Integer(1),Integer(0))])
>>> p.poly_x("e")                                                    #␣
↪needs palp
Traceback (most recent call last):
...
ValueError: Cannot run PALP for a 2-dimensional polytope in a 3-dimensional␣
↪space!
```

But if you know what you are doing, you can call it for the polytope in some basis of the spanned space:

```
sage: print(p.poly_x("e", reduce_dimension=True))                      #␣
↪needs palp
4 2  Equations of P
  -1   1     0
   1   1     2
  -1  -1     0
   1  -1     2
```

```
>>> from sage.all import *
>>> print(p.poly_x("e", reduce_dimension=True))                      #␣
↪needs palp
4 2  Equations of P
  -1   1     0
   1   1     2
  -1  -1     0
   1  -1     2
```

polyhedron(***kwds*)

Return the Polyhedron object determined by this polytope's vertices.

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(2)
sage: o.polyhedron()
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(2))
>>> o.polyhedron()
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices
```

show3d()

Show a 3d picture of the polytope with default settings and without axes or frame.

See self.plot3d? for more details.

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.show3d()                                                          #␣
↪needs palp sage.plot
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> o.show3d()                                                          #␣
↪needs palp sage.plot
```

skeleton()

Return the graph of the one-skeleton of this polytope.

EXAMPLES:

```
sage: d = lattice_polytope.cross_polytope(2)
sage: g = d.skeleton(); g                                                 #␣
↪needs palp sage.graphs
Graph on 4 vertices
sage: g.edges(sort=True)                                                  #␣
↪needs palp sage.graphs
[(0, 1, None), (0, 3, None), (1, 2, None), (2, 3, None)]
```

```
>>> from sage.all import *
>>> d = lattice_polytope.cross_polytope(Integer(2))
>>> g = d.skeleton(); g                                                 #␣
↪needs palp sage.graphs
Graph on 4 vertices
>>> g.edges(sort=True)                                                  #␣
↪needs palp sage.graphs
[(0, 1, None), (0, 3, None), (1, 2, None), (2, 3, None)]
```

skeleton_points(*k=1*)

Return the increasing list of indices of lattice points in k-skeleton of the polytope (k is 1 by default).

EXAMPLES: We compute all skeleton points for the cube:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: c = o.polar()
sage: c.skeleton_points()                                                 #␣
↪needs palp sage.graphs
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 15, 19, 21, 22, 23, 25, 26]
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> c = o.polar()
>>> c.skeleton_points()                                                    #␣
↪needs palp sage.graphs
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 15, 19, 21, 22, 23, 25, 26]
```

The default was 1-skeleton:

```
sage: c.skeleton_points(k=1)                                                 #␣
↪needs palp sage.graphs
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 15, 19, 21, 22, 23, 25, 26]
```

```
>>> from sage.all import *
>>> c.skeleton_points(k=Integer(1))                                          ␣
↪     # needs palp sage.graphs
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 12, 13, 15, 19, 21, 22, 23, 25, 26]
```

0-skeleton just lists all vertices:

```
sage: c.skeleton_points(k=0)                                                 #␣
↪needs palp sage.graphs
[0, 1, 2, 3, 4, 5, 6, 7]
```

```
>>> from sage.all import *
>>> c.skeleton_points(k=Integer(0))                                          ␣
↪     # needs palp sage.graphs
[0, 1, 2, 3, 4, 5, 6, 7]
```

2-skeleton lists all points except for the origin (point #17):

```
sage: c.skeleton_points(k=2)                                                 #␣
↪needs palp sage.graphs
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
 18, 19, 20, 21, 22, 23, 24, 25, 26]
```

```
>>> from sage.all import *
>>> c.skeleton_points(k=Integer(2))                                          ␣
↪     # needs palp sage.graphs
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16,
 18, 19, 20, 21, 22, 23, 24, 25, 26]
```

3-skeleton includes all points:

```
sage: c.skeleton_points(k=3)                                                 #␣
↪needs palp
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
 18, 19, 20, 21, 22, 23, 24, 25, 26]
```

```
>>> from sage.all import *
>>> c.skeleton_points(k=Integer(3))                                          ␣
↪     # needs palp
```

(continues on next page)

(continued from previous page)

```
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
 18, 19, 20, 21, 22, 23, 24, 25, 26]
```

It is OK to compute higher dimensional skeletons - you will get the list of all points:

```
sage: c.skeleton_points(k=100)                                          #␣
↪needs palp
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
 18, 19, 20, 21, 22, 23, 24, 25, 26]
```

```
>>> from sage.all import *
>>> c.skeleton_points(k=Integer(100))                                    ␣
↪      # needs palp
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17,
 18, 19, 20, 21, 22, 23, 24, 25, 26]
```

skeleton_show(*normal=None*)

Show the graph of one-skeleton of this polytope. Works only for polytopes in a 3-dimensional space.

INPUT:

- `normal` – a 3-dimensional vector (can be given as a list), which should be perpendicular to the screen. If not given, will be selected randomly (new each time and it may be far from "nice").

EXAMPLES: Show a pretty picture of the octahedron:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.skeleton_show([1,2,4])                                          #␣
↪needs palp sage.plot
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> o.skeleton_show([Integer(1),Integer(2),Integer(4)])                 ␣
↪                           # needs palp sage.plot
```

Does not work for a diamond at the moment:

```
sage: d = lattice_polytope.cross_polytope(2)
sage: d.skeleton_show()
Traceback (most recent call last):
...
NotImplementedError: skeleton view is implemented only in 3-d space
```

```
>>> from sage.all import *
>>> d = lattice_polytope.cross_polytope(Integer(2))
>>> d.skeleton_show()
Traceback (most recent call last):
...
NotImplementedError: skeleton view is implemented only in 3-d space
```

traverse_boundary()

Return a list of indices of vertices of a 2-dimensional polytope in their boundary order.

Needed for plot3d function of polytopes.

EXAMPLES:

```
sage: p = lattice_polytope.cross_polytope(2).polar()
sage: p.traverse_boundary()                                                    #␣
↪needs sage.graphs
[3, 0, 1, 2]
```

```
>>> from sage.all import *
>>> p = lattice_polytope.cross_polytope(Integer(2)).polar()
>>> p.traverse_boundary()                                                   #␣
↪needs sage.graphs
[3, 0, 1, 2]
```

vertex(i)

Return the i-th vertex of this polytope, i.e. the i-th column of the matrix returned by `vertices()`.

EXAMPLES: Note that numeration starts with zero:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.vertices()
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1)
in 3-d lattice M
sage: o.vertex(3)
M(-1, 0, 0)
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> o.vertices()
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1)
in 3-d lattice M
>>> o.vertex(Integer(3))
M(-1, 0, 0)
```

vertex_facet_pairing_matrix()

Return the vertex facet pairing matrix PM.

Return a matrix whose the i, j^{th} entry is the height of the j^{th} vertex over the i^{th} facet. The ordering of the vertices and facets is as in `vertices()` and `facets()`.

EXAMPLES:

```
sage: L = lattice_polytope.cross_polytope(3)
sage: L.vertex_facet_pairing_matrix()
[2 0 0 0 2 2]
```

(continues on next page)

(continued from previous page)

```
[2 2 0 0 0 2]
[2 2 2 0 0 0]
[2 0 2 0 2 0]
[0 0 2 2 2 0]
[0 0 0 2 2 2]
[0 2 0 2 0 2]
[0 2 2 2 0 0]
```

```
>>> from sage.all import *
>>> L = lattice_polytope.cross_polytope(Integer(3))
>>> L.vertex_facet_pairing_matrix()
[2 0 0 0 2 2]
[2 2 0 0 0 2]
[2 2 2 0 0 0]
[2 0 2 0 2 0]
[0 0 2 2 2 0]
[0 0 0 2 2 2]
[0 2 0 2 0 2]
[0 2 2 2 0 0]
```

vertices(**args*, ***kwds*)

Return vertices of `self`.

INPUT:

- any arguments given will be passed on to the returned object.

OUTPUT: a *point collection*

EXAMPLES:

Vertices of the octahedron and its polar cube are in dual lattices:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.vertices()
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1)
in 3-d lattice M
sage: cube = o.polar()
sage: cube.vertices()
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
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> o.vertices()
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1)
in 3-d lattice M
>>> cube = o.polar()
>>> cube.vertices()
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
```

class sage.geometry.lattice_polytope.**NefPartition**(*data*, *Delta_polar*, *check=True*)

Bases: `SageObject`, `Hashable`

Create a nef-partition.

INPUT:

- `data` – list of integers, the i-th element of this list must be the part of the i-th vertex of `Delta_polar` in this nef-partition;
- `Delta_polar` – a *lattice polytope*;
- `check` – by default the input will be checked for correctness, i.e. that `data` indeed specify a nef-partition. If you are sure that the input is correct, you can speed up construction via `check=False` option.

OUTPUT: a nef-partition of `Delta_polar`

Let M and N be dual lattices. Let $\Delta \subset M_{\mathbf{R}}$ be a reflexive polytope with polar $\Delta^\circ \subset N_{\mathbf{R}}$. Let X_Δ be the toric variety associated to the normal fan of Δ. A **nef-partition** is a decomposition of the vertex set V of Δ° into a disjoint union $V = V_0 \sqcup V_1 \sqcup \dots \sqcup V_{k-1}$ such that divisors $E_i = \sum_{v \in V_i} D_v$ are Cartier (here D_v are prime torus-invariant Weil divisors corresponding to vertices of Δ°). Equivalently, let $\nabla_i \subset N_{\mathbf{R}}$ be the convex hull of vertices from V_i and the origin. These polytopes form a nef-partition if their Minkowski sum $\nabla \subset N_{\mathbf{R}}$ is a reflexive polytope.

The **dual nef-partition** is formed by polytopes $\Delta_i \subset M_{\mathbf{R}}$ of E_i, which give a decomposition of the vertex set of $\nabla^\circ \subset M_{\mathbf{R}}$ and their Minkowski sum is Δ, i.e. the polar duality of reflexive polytopes switches convex hull and Minkowski sum for dual nef-partitions:

$$\begin{aligned} \Delta^\circ &= \mathrm{Conv}\left(\nabla_0, \nabla_1, \dots, \nabla_{k-1}\right), \\ \nabla &= \nabla_0 + \nabla_1 + \dots + \nabla_{k-1}, \end{aligned}$$

$$\begin{aligned} \Delta &= \Delta_0 + \Delta_1 + \dots + \Delta_{k-1}, \\ \nabla^\circ &= \mathrm{Conv}\left(\Delta_0, \Delta_1, \dots, \Delta_{k-1}\right). \end{aligned}$$

One can also interpret the duality of nef-partitions as the duality of the associated cones. Below $\overline{M} = M \times \mathbf{Z}^k$ and $\overline{N} = N \times \mathbf{Z}^k$ are dual lattices.

The **Cayley polytope** $P \subset \overline{M}_\mathbf{R}$ of a nef-partition is given by $P = \mathrm{Conv}(\Delta_0 \times e_0, \Delta_1 \times e_1, \ldots, \Delta_{k-1} \times e_{k-1})$, where $\{e_i\}_{i=0}^{k-1}$ is the standard basis of $\mathbf{Z}^k$. The **dual Cayley polytope** $P^* \subset \overline{N}_\mathbf{R}$ is the Cayley polytope of the dual nef-partition.

The **Cayley cone** $C \subset \overline{M}_\mathbf{R}$ of a nef-partition is the cone spanned by its Cayley polytope. The **dual Cayley cone** $C^\vee \subset \overline{M}_\mathbf{R}$ is the usual dual cone of C. It turns out, that $C^\vee$ is spanned by P^*.

It is also possible to go back from the Cayley cone to the Cayley polytope, since C is a reflexive Gorenstein cone supported by P: primitive integral ray generators of C are contained in an affine hyperplane and coincide with vertices of P.

See Section 4.3.1 in [CK1999] and references therein for further details, or [BN2008] for a purely combinatorial approach.

EXAMPLES:

It is very easy to create a nef-partition for the octahedron, since for this polytope any decomposition of vertices is a nef-partition. We create a 3-part nef-partition with the 0th and 1st vertices belonging to the 0th part (recall that numeration in Sage starts with 0), the 2nd and 5th vertices belonging to the 1st part, and 3rd and 4th vertices belonging to the 2nd part:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: np = NefPartition([0,0,1,2,2,1], o)
sage: np
Nef-partition {0, 1} ⊔ {2, 5} ⊔ {3, 4}
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> np = NefPartition([Integer(0),Integer(0),Integer(1),Integer(2),Integer(2),
↪Integer(1)], o)
>>> np
Nef-partition {0, 1} ⊔ {2, 5} ⊔ {3, 4}
```

The octahedron plays the role of Δ° in the above description:

```
sage: np.Delta_polar() is o
True
```

```
>>> from sage.all import *
>>> np.Delta_polar() is o
True
```

The dual nef-partition (corresponding to the "mirror complete intersection") gives decomposition of the vertex set of ∇°:

```
sage: np.dual()
Nef-partition {0, 1, 2} ⊔ {3, 4} ⊔ {5, 6, 7}
sage: np.nabla_polar().vertices()
N(-1, -1,  0),
N(-1,  0,  0),
N( 0, -1,  0),
N( 0,  0, -1),
N( 0,  0,  1),
```

(continues on next page)

(continued from previous page)

```
N( 1,  0,  0),
N( 0,  1,  0),
N( 1,  1,  0)
in 3-d lattice N
```

```
>>> from sage.all import *
>>> np.dual()
Nef-partition {0, 1, 2} ⊔ {3, 4} ⊔ {5, 6, 7}
>>> np.nabla_polar().vertices()
N(-1, -1,  0),
N(-1,  0,  0),
N( 0, -1,  0),
N( 0,  0, -1),
N( 0,  0,  1),
N( 1,  0,  0),
N( 0,  1,  0),
N( 1,  1,  0)
in 3-d lattice N
```

Of course, ∇° is Δ° from the point of view of the dual nef-partition:

```
sage: np.dual().Delta_polar() is np.nabla_polar()
True
sage: np.Delta(1).vertices()
N(0, 0, -1),
N(0, 0,  1)
in 3-d lattice N
sage: np.dual().nabla(1).vertices()
N(0, 0, -1),
N(0, 0,  1)
in 3-d lattice N
```

```
>>> from sage.all import *
>>> np.dual().Delta_polar() is np.nabla_polar()
True
>>> np.Delta(Integer(1)).vertices()
N(0, 0, -1),
N(0, 0,  1)
in 3-d lattice N
>>> np.dual().nabla(Integer(1)).vertices()
N(0, 0, -1),
N(0, 0,  1)
in 3-d lattice N
```

Instead of constructing nef-partitions directly, you can request all 2-part nef-partitions of a given reflexive polytope (they will be computed using `nef.x` program from PALP):

```
sage: o.nef_partitions()                                                    #␣
↪needs palp
[Nef-partition {0, 1, 3} ⊔ {2, 4, 5},
 Nef-partition {0, 1, 3, 4} ⊔ {2, 5} (direct product),
 Nef-partition {0, 1, 2} ⊔ {3, 4, 5},
```

(continues on next page)

(continued from previous page)

```
 Nef-partition {0, 1, 2, 3} ⊔ {4, 5},
 Nef-partition {0, 1, 2, 3, 4} ⊔ {5} (projection)]
```

```
>>> from sage.all import *
>>> o.nef_partitions()                                                    #␣
↪needs palp
[Nef-partition {0, 1, 3} ⊔ {2, 4, 5},
 Nef-partition {0, 1, 3, 4} ⊔ {2, 5} (direct product),
 Nef-partition {0, 1, 2} ⊔ {3, 4, 5},
 Nef-partition {0, 1, 2, 3} ⊔ {4, 5},
 Nef-partition {0, 1, 2, 3, 4} ⊔ {5} (projection)]
```

Delta(*i=None*)

Return the polytope Δ or Δ_i corresponding to `self`.

INPUT:

- `i` – integer; if not given, Δ will be returned

OUTPUT: a *lattice polytope*

See *nef-partition* class documentation for definitions and notation.

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: np = NefPartition([0, 0, 1, 0, 1, 1], o); np
Nef-partition {0, 1, 3} ⊔ {2, 4, 5}
sage: np.Delta().polar() is o
True
sage: np.Delta().vertices()
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: np.Delta(0).vertices()
N(-1, -1, 0),
N(-1,  0, 0),
N( 1,  0, 0),
N( 1, -1, 0)
in 3-d lattice N
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> np = NefPartition([Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(1)], o); np
Nef-partition {0, 1, 3} ⊔ {2, 4, 5}
>>> np.Delta().polar() is o
True
```

(continues on next page)

(continued from previous page)

```
>>> np.Delta().vertices()
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
>>> np.Delta(Integer(0)).vertices()
N(-1, -1, 0),
N(-1,  0, 0),
N( 1,  0, 0),
N( 1, -1, 0)
in 3-d lattice N
```

Delta_polar()

Return the polytope Δ° corresponding to `self`.

OUTPUT: a *lattice polytope*

See *nef-partition* class documentation for definitions and notation.

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: np = NefPartition([0, 0, 1, 0, 1, 1], o); np
Nef-partition {0, 1, 3} ⊔ {2, 4, 5}
sage: np.Delta_polar() is o
True
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> np = NefPartition([Integer(0), Integer(0), Integer(1), Integer(0),
↪Integer(1), Integer(1)], o); np
Nef-partition {0, 1, 3} ⊔ {2, 4, 5}
>>> np.Delta_polar() is o
True
```

Deltas()

Return the polytopes Δ_i corresponding to `self`.

OUTPUT: a tuple of *lattice polytopes*

See *nef-partition* class documentation for definitions and notation.

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: np = NefPartition([0, 0, 1, 0, 1, 1], o); np
Nef-partition {0, 1, 3} ⊔ {2, 4, 5}
sage: np.Delta().vertices()
N( 1, -1, -1),
N( 1,  1, -1),
```

(continues on next page)

(continued from previous page)

```
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: [Delta_i.vertices() for Delta_i in np.Deltas()]
[N(-1, -1, 0),
 N(-1,  0, 0),
 N( 1,  0, 0),
 N( 1, -1, 0)
 in 3-d lattice N,
 N(0, 0, -1),
 N(0, 1,  1),
 N(0, 0,  1),
 N(0, 1, -1)
 in 3-d lattice N]
sage: np.nabla_polar().vertices()
N(-1, -1,  0),
N( 1, -1,  0),
N( 1,  0,  0),
N(-1,  0,  0),
N( 0,  1, -1),
N( 0,  1,  1),
N( 0,  0,  1),
N( 0,  0, -1)
in 3-d lattice N
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> np = NefPartition([Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(1)], o); np
Nef-partition {0, 1, 3} ⊔ {2, 4, 5}
>>> np.Delta().vertices()
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
>>> [Delta_i.vertices() for Delta_i in np.Deltas()]
[N(-1, -1, 0),
 N(-1,  0, 0),
 N( 1,  0, 0),
 N( 1, -1, 0)
 in 3-d lattice N,
 N(0, 0, -1),
 N(0, 1,  1),
```

(continues on next page)

(continued from previous page)

```
 N(0, 0,  1),
 N(0, 1, -1)
 in 3-d lattice N]
>>> np.nabla_polar().vertices()
N(-1, -1,  0),
N( 1, -1,  0),
N( 1,  0,  0),
N(-1,  0,  0),
N( 0,  1, -1),
N( 0,  1,  1),
N( 0,  0,  1),
N( 0,  0, -1)
in 3-d lattice N
```

dual()

Return the dual nef-partition.

OUTPUT: a *nef-partition*

See the class documentation for the definition.

ALGORITHM:

See Proposition 3.19 in [BN2008].

Note

Automatically constructed dual nef-partitions will be ordered, i.e. vertex partition of ∇ will look like $\{0,1,2\} \sqcup \{3,4,5,6\} \sqcup \{7,8\}$.

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: np = NefPartition([0, 0, 1, 0, 1, 1], o); np
Nef-partition {0, 1, 3} ⊔ {2, 4, 5}
sage: np.dual()
Nef-partition {0, 1, 2, 3} ⊔ {4, 5, 6, 7}
sage: np.dual().Delta() is np.nabla()
True
sage: np.dual().nabla(0) is np.Delta(0)
True
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> np = NefPartition([Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(1)], o); np
Nef-partition {0, 1, 3} ⊔ {2, 4, 5}
>>> np.dual()
Nef-partition {0, 1, 2, 3} ⊔ {4, 5, 6, 7}
>>> np.dual().Delta() is np.nabla()
True
>>> np.dual().nabla(Integer(0)) is np.Delta(Integer(0))
True
```

hodge_numbers()

Return Hodge numbers corresponding to `self`.

OUTPUT: a tuple of integers (produced by `nef.x` program from PALP)

EXAMPLES:

Currently, you need to request Hodge numbers when you compute nef-partitions:

```
sage: # long time, needs palp
sage: p = lattice_polytope.cross_polytope(5)
sage: np = p.nef_partitions()[0]                    # 4s on sage.math, 2011
sage: np.hodge_numbers()
Traceback (most recent call last):
...
NotImplementedError: use nef_partitions(hodge_numbers=True)!
sage: np = p.nef_partitions(hodge_numbers=True)[0]  # 13s on sage.math, 2011
sage: np.hodge_numbers()
(19, 19)
```

```
>>> from sage.all import *
>>> # long time, needs palp
>>> p = lattice_polytope.cross_polytope(Integer(5))
>>> np = p.nef_partitions()[Integer(0)]                    # 4s on sage.math,␣
↪2011
>>> np.hodge_numbers()
Traceback (most recent call last):
...
NotImplementedError: use nef_partitions(hodge_numbers=True)!
>>> np = p.nef_partitions(hodge_numbers=True)[Integer(0)]  # 13s on sage.math,
↪ 2011
>>> np.hodge_numbers()
(19, 19)
```

nabla(*i=None*)

Return the polytope ∇ or ∇_i corresponding to `self`.

INPUT:

- `i` – integer; if not given, ∇ will be returned

OUTPUT: a *lattice polytope*

See *nef-partition* class documentation for definitions and notation.

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: np = NefPartition([0, 0, 1, 0, 1, 1], o); np
Nef-partition {0, 1, 3} ⊔ {2, 4, 5}
sage: np.Delta_polar().vertices()
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1)
```

(continues on next page)

(continued from previous page)

```
in 3-d lattice M
sage: np.nabla(0).vertices()
M(-1, 0, 0),
M( 1, 0, 0),
M( 0, 1, 0)
in 3-d lattice M
sage: np.nabla().vertices()
M(-1,  0,  1),
M(-1,  0, -1),
M( 1,  0,  1),
M( 1,  0, -1),
M( 0,  1,  1),
M( 0,  1, -1),
M( 1, -1,  0),
M(-1, -1,  0)
in 3-d lattice M
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> np = NefPartition([Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(1)], o); np
Nef-partition {0, 1, 3} ⊔ {2, 4, 5}
>>> np.Delta_polar().vertices()
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1)
in 3-d lattice M
>>> np.nabla(Integer(0)).vertices()
M(-1, 0, 0),
M( 1, 0, 0),
M( 0, 1, 0)
in 3-d lattice M
>>> np.nabla().vertices()
M(-1,  0,  1),
M(-1,  0, -1),
M( 1,  0,  1),
M( 1,  0, -1),
M( 0,  1,  1),
M( 0,  1, -1),
M( 1, -1,  0),
M(-1, -1,  0)
in 3-d lattice M
```

nabla_polar()

Return the polytope ∇° corresponding to `self`.

OUTPUT: a *lattice polytope*

See *nef-partition* class documentation for definitions and notation.

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: np = NefPartition([0, 0, 1, 0, 1, 1], o); np
Nef-partition {0, 1, 3} ⊔ {2, 4, 5}
sage: np.nabla_polar().vertices()
N(-1, -1,  0),
N( 1, -1,  0),
N( 1,  0,  0),
N(-1,  0,  0),
N( 0,  1, -1),
N( 0,  1,  1),
N( 0,  0,  1),
N( 0,  0, -1)
in 3-d lattice N
sage: np.nabla_polar() is np.dual().Delta_polar()
True
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> np = NefPartition([Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(1)], o); np
Nef-partition {0, 1, 3} ⊔ {2, 4, 5}
>>> np.nabla_polar().vertices()
N(-1, -1,  0),
N( 1, -1,  0),
N( 1,  0,  0),
N(-1,  0,  0),
N( 0,  1, -1),
N( 0,  1,  1),
N( 0,  0,  1),
N( 0,  0, -1)
in 3-d lattice N
>>> np.nabla_polar() is np.dual().Delta_polar()
True
```

nablas()

Return the polytopes ∇_i corresponding to `self`.

OUTPUT: a tuple of *lattice polytopes*

See *nef-partition* class documentation for definitions and notation.

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: np = NefPartition([0, 0, 1, 0, 1, 1], o); np
Nef-partition {0, 1, 3} ⊔ {2, 4, 5}
sage: np.Delta_polar().vertices()
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1)
in 3-d lattice M
```

(continues on next page)

(continued from previous page)

```
sage: [nabla_i.vertices() for nabla_i in np.nablas()]
[M(-1, 0, 0),
 M( 1, 0, 0),
 M( 0, 1, 0)
 in 3-d lattice M,
 M(0, -1,  0),
 M(0,  0, -1),
 M(0,  0,  1)
 in 3-d lattice M]
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> np = NefPartition([Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(1)], o); np
Nef-partition {0, 1, 3} ⊔ {2, 4, 5}
>>> np.Delta_polar().vertices()
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1)
in 3-d lattice M
>>> [nabla_i.vertices() for nabla_i in np.nablas()]
[M(-1, 0, 0),
 M( 1, 0, 0),
 M( 0, 1, 0)
 in 3-d lattice M,
 M(0, -1,  0),
 M(0,  0, -1),
 M(0,  0,  1)
 in 3-d lattice M]
```

nparts()

Return the number of parts in `self`.

OUTPUT: integer

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: np = NefPartition([0, 0, 1, 0, 1, 1], o); np
Nef-partition {0, 1, 3} ⊔ {2, 4, 5}
sage: np.nparts()
2
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> np = NefPartition([Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(1)], o); np
Nef-partition {0, 1, 3} ⊔ {2, 4, 5}
>>> np.nparts()
2
```

part (*i*, *all_points=False*)

Return the `i`-th part of `self`.

INPUT:

- `i` – integer
- `all_points` – boolean (default: `False`); whether to list all lattice points or just vertices

OUTPUT:

- a tuple of integers, indices of vertices (or all lattice points) of Δ° belonging to V_i.

See *nef-partition* class documentation for definitions and notation.

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: np = NefPartition([0, 0, 1, 0, 1, 1], o); np
Nef-partition {0, 1, 3} ⊔ {2, 4, 5}
sage: np.part(0)
(0, 1, 3)
sage: np.part(0, all_points=True)                                    #␣
↪needs palp
(0, 1, 3)
sage: np.dual().part(0)
(0, 1, 2, 3)
sage: np.dual().part(0, all_points=True)                             #␣
↪needs palp
(0, 1, 2, 3, 8)
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> np = NefPartition([Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(1)], o); np
Nef-partition {0, 1, 3} ⊔ {2, 4, 5}
>>> np.part(Integer(0))
(0, 1, 3)
>>> np.part(Integer(0), all_points=True)                              ␣
↪      # needs palp
(0, 1, 3)
>>> np.dual().part(Integer(0))
(0, 1, 2, 3)
>>> np.dual().part(Integer(0), all_points=True)                       ␣
↪      # needs palp
(0, 1, 2, 3, 8)
```

part_of (*i*)

Return the index of the part containing the `i`-th vertex.

INPUT:

- `i` – integer

OUTPUT:

- an integer j such that the `i`-th vertex of Δ° belongs to V_j.

See *nef-partition* class documentation for definitions and notation.

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: np = NefPartition([0, 0, 1, 0, 1, 1], o); np
Nef-partition {0, 1, 3} ⊔ {2, 4, 5}
sage: np.part_of(3)
0
sage: np.part_of(2)
1
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> np = NefPartition([Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(1)], o); np
Nef-partition {0, 1, 3} ⊔ {2, 4, 5}
>>> np.part_of(Integer(3))
0
>>> np.part_of(Integer(2))
1
```

part_of_point(*i*)

Return the index of the part containing the `i`-th point.

INPUT:

- `i` – integer

OUTPUT:

- an integer j such that the `i`-th point of Δ° belongs to ∇_j.

Note

Since a nef-partition induces a partition on the set of boundary lattice points of Δ°, the value of j is well-defined for all i but the one that corresponds to the origin, in which case this method will raise a `ValueError` exception. (The origin always belongs to all ∇_j.)

See *nef-partition* class documentation for definitions and notation.

EXAMPLES:

We consider a relatively complicated reflexive polytope #2252 (easily accessible in Sage as `ReflexivePolytope(3, 2252)`, we create it here explicitly to avoid loading the whole database):

```
sage: p = LatticePolytope([(1,0,0), (0,1,0), (0,0,1), (0,1,-1),
....:           (0,-1,1), (-1,1,0), (0,-1,-1), (-1,-1,0), (-1,-1,2)])
sage: np = p.nef_partitions()[0]; np                                    #␣
↪needs palp
Nef-partition {1, 2, 5, 7, 8} ⊔ {0, 3, 4, 6}
sage: p.nvertices()
9
sage: p.npoints()                                                       #␣
↪needs palp
15
```

```
>>> from sage.all import *
>>> p = LatticePolytope([(Integer(1),Integer(0),Integer(0)), (Integer(0),
↪Integer(1),Integer(0)), (Integer(0),Integer(0),Integer(1)), (Integer(0),
↪Integer(1),-Integer(1)),
...           (Integer(0),-Integer(1),Integer(1)), (-Integer(1),Integer(1),
↪Integer(0)), (Integer(0),-Integer(1),-Integer(1)), (-Integer(1),-Integer(1),
↪Integer(0)), (-Integer(1),-Integer(1),Integer(2))])
>>> np = p.nef_partitions()[Integer(0)]; np                                  ␣
↪     # needs palp
Nef-partition {1, 2, 5, 7, 8} ⊔ {0, 3, 4, 6}
>>> p.nvertices()
9
>>> p.npoints()                                                            #␣
↪needs palp
15
```

We see that the polytope has 6 more points in addition to vertices. One of them is the origin:

```
sage: p.origin()                                                             #␣
↪needs palp
14
sage: np.part_of_point(14)                                                   #␣
↪needs palp
Traceback (most recent call last):
...
ValueError: the origin belongs to all parts!
```

```
>>> from sage.all import *
>>> p.origin()                                                             #␣
↪needs palp
14
>>> np.part_of_point(Integer(14))                                            ␣
↪     # needs palp
Traceback (most recent call last):
...
ValueError: the origin belongs to all parts!
```

But the remaining 5 are partitioned by np:

```
sage: [n for n in range(p.npoints())                                         #␣
↪needs palp
....:     if p.origin() != n and np.part_of_point(n) == 0]
[1, 2, 5, 7, 8, 9, 11, 13]
sage: [n for n in range(p.npoints())                                         #␣
↪needs palp
....:     if p.origin() != n and np.part_of_point(n) == 1]
[0, 3, 4, 6, 10, 12]
```

```
>>> from sage.all import *
>>> [n for n in range(p.npoints())                                         #␣
↪needs palp
...     if p.origin() != n and np.part_of_point(n) == Integer(0)]
[1, 2, 5, 7, 8, 9, 11, 13]
```

(continues on next page)

(continued from previous page)

```
>>> [n for n in range(p.npoints())                                          #␣
↪needs palp
...     if p.origin() != n and np.part_of_point(n) == Integer(1)]
[0, 3, 4, 6, 10, 12]
```

parts(*all_points=False*)

Return all parts of `self`.

INPUT:

- `all_points` – boolean (default: `False`); whether to list all lattice points or just vertices

OUTPUT:

- a tuple of tuples of integers. The i-th tuple contains indices of vertices (or all lattice points) of Δ° belonging to V_i

See *nef-partition* class documentation for definitions and notation.

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: np = NefPartition([0, 0, 1, 0, 1, 1], o); np
Nef-partition {0, 1, 3} ⊔ {2, 4, 5}
sage: np.parts()
((0, 1, 3), (2, 4, 5))
sage: np.parts(all_points=True)                                             #␣
↪needs palp
((0, 1, 3), (2, 4, 5))
sage: np.dual().parts()
((0, 1, 2, 3), (4, 5, 6, 7))
sage: np.dual().parts(all_points=True)                                      #␣
↪needs palp
((0, 1, 2, 3, 8), (4, 5, 6, 7, 10))
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> np = NefPartition([Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(1), Integer(1)], o); np
Nef-partition {0, 1, 3} ⊔ {2, 4, 5}
>>> np.parts()
((0, 1, 3), (2, 4, 5))
>>> np.parts(all_points=True)                                               #␣
↪needs palp
((0, 1, 3), (2, 4, 5))
>>> np.dual().parts()
((0, 1, 2, 3), (4, 5, 6, 7))
>>> np.dual().parts(all_points=True)                                        #␣
↪needs palp
((0, 1, 2, 3, 8), (4, 5, 6, 7, 10))
```

`sage.geometry.lattice_polytope.`**ReflexivePolytope**(*dim*, *n*)

Return the n-th 2- or 3-dimensional reflexive polytope.

Note

1. Numeration starts with zero: $0 \leq n \leq 15$ for $\dim = 2$ and $0 \leq n \leq 4318$ for $\dim = 3$.
2. During the first call, all reflexive polytopes of requested dimension are loaded and cached for future use, so the first call for 3-dimensional polytopes can take several seconds, but all consecutive calls are fast.
3. Equivalent to `ReflexivePolytopes(dim)[n]` but checks bounds first.

EXAMPLES:

The 3rd 2-dimensional polytope is "the diamond":

```
sage: ReflexivePolytope(2, 3)
2-d reflexive polytope #3 in 2-d lattice M
sage: lattice_polytope.ReflexivePolytope(2,3).vertices()
M( 1,  0),
M( 0,  1),
M( 0, -1),
M(-1,  0)
in 2-d lattice M
```

```
>>> from sage.all import *
>>> ReflexivePolytope(Integer(2), Integer(3))
2-d reflexive polytope #3 in 2-d lattice M
>>> lattice_polytope.ReflexivePolytope(Integer(2),Integer(3)).vertices()
M( 1,  0),
M( 0,  1),
M( 0, -1),
M(-1,  0)
in 2-d lattice M
```

There are 16 reflexive polygons and numeration starts with 0:

```
sage: ReflexivePolytope(2,16)
Traceback (most recent call last):
...
ValueError: there are only 16 reflexive polygons!
```

```
>>> from sage.all import *
>>> ReflexivePolytope(Integer(2),Integer(16))
Traceback (most recent call last):
...
ValueError: there are only 16 reflexive polygons!
```

It is not possible to load a 4-dimensional polytope in this way:

```
sage: ReflexivePolytope(4,16)
Traceback (most recent call last):
...
NotImplementedError: only 2- and 3-dimensional reflexive polytopes are available!
```

```
>>> from sage.all import *
>>> ReflexivePolytope(Integer(4),Integer(16))
Traceback (most recent call last):
...
NotImplementedError: only 2- and 3-dimensional reflexive polytopes are available!
```

sage.geometry.lattice_polytope.**ReflexivePolytopes**(*dim*)

Return the sequence of all 2- or 3-dimensional reflexive polytopes.

Note

During the first call the database is loaded and cached for future use, so repetitive calls will return the same object in memory.

INPUT:

- `dim` – integer (2 or 3); dimension of required reflexive polytopes

OUTPUT: list of lattice polytopes

EXAMPLES:

There are 16 reflexive polygons:

```
sage: len(ReflexivePolytopes(2))
16
```

```
>>> from sage.all import *
>>> len(ReflexivePolytopes(Integer(2)))
16
```

It is not possible to load 4-dimensional polytopes in this way:

```
sage: ReflexivePolytopes(4)
Traceback (most recent call last):
...
NotImplementedError: only 2- and 3-dimensional reflexive polytopes are available!
```

```
>>> from sage.all import *
>>> ReflexivePolytopes(Integer(4))
Traceback (most recent call last):
...
NotImplementedError: only 2- and 3-dimensional reflexive polytopes are available!
```

class sage.geometry.lattice_polytope.**SetOfAllLatticePolytopesClass**

Bases: `Set_generic`

sage.geometry.lattice_polytope.**all_cached_data**(*polytopes*)

Compute all cached data for all given `polytopes` and their polars.

This functions does it MUCH faster than member functions of `LatticePolytope` during the first run. So it is recommended to use this functions if you work with big sets of data. None of the polytopes in the given sequence should be constructed as the polar polytope to another one.

INPUT:

- polytopes – a sequence of lattice polytopes

EXAMPLES: This function has no output, it is just a fast way to work with long sequences of polytopes. Of course, you can use short sequences as well:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: lattice_polytope.all_cached_data([o])                          #␣
↪needs palp
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> lattice_polytope.all_cached_data([o])                          #␣
↪needs palp
```

sage.geometry.lattice_polytope.**all_facet_equations**(*polytopes*)

Compute polar polytopes for all reflexive and equations of facets for all non-reflexive polytopes.

all_facet_equations and all_polars are synonyms.

This functions does it MUCH faster than member functions of LatticePolytope during the first run. So it is recommended to use this functions if you work with big sets of data.

INPUT:

- polytopes – a sequence of lattice polytopes

EXAMPLES: This function has no output, it is just a fast way to work with long sequences of polytopes. Of course, you can use short sequences as well:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: lattice_polytope.all_polars([o])                          #␣
↪needs palp
sage: o.polar()                          #␣
↪needs palp
3-d reflexive polytope in 3-d lattice N
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> lattice_polytope.all_polars([o])                          #␣
↪needs palp
>>> o.polar()                          #␣
↪needs palp
3-d reflexive polytope in 3-d lattice N
```

sage.geometry.lattice_polytope.**all_nef_partitions**(*polytopes*, *keep_symmetric=False*)

Compute nef-partitions for all given polytopes.

This functions does it MUCH faster than member functions of LatticePolytope during the first run. So it is recommended to use this functions if you work with big sets of data.

Note: member function is_reflexive will be called separately for each polytope. It is strictly recommended to call all_polars on the sequence of polytopes before using this function.

INPUT:

- polytopes – a sequence of lattice polytopes

EXAMPLES: This function has no output, it is just a fast way to work with long sequences of polytopes. Of course, you can use short sequences as well:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: lattice_polytope.all_nef_partitions([o])                          #␣
↪needs palp
sage: o.nef_partitions()                                                 #␣
↪needs palp
[Nef-partition {0, 1, 3} ⊔ {2, 4, 5},
 Nef-partition {0, 1, 3, 4} ⊔ {2, 5} (direct product),
 Nef-partition {0, 1, 2} ⊔ {3, 4, 5},
 Nef-partition {0, 1, 2, 3} ⊔ {4, 5},
 Nef-partition {0, 1, 2, 3, 4} ⊔ {5} (projection)]
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> lattice_polytope.all_nef_partitions([o])                           #␣
↪needs palp
>>> o.nef_partitions()                                                  #␣
↪needs palp
[Nef-partition {0, 1, 3} ⊔ {2, 4, 5},
 Nef-partition {0, 1, 3, 4} ⊔ {2, 5} (direct product),
 Nef-partition {0, 1, 2} ⊔ {3, 4, 5},
 Nef-partition {0, 1, 2, 3} ⊔ {4, 5},
 Nef-partition {0, 1, 2, 3, 4} ⊔ {5} (projection)]
```

You cannot use this function for non-reflexive polytopes:

```
sage: p = LatticePolytope([(1,0,0), (0,1,0), (0,0,2),
....:                      (-1,0,0), (0,-1,0), (0,0,-1)])
sage: lattice_polytope.all_nef_partitions([o, p])                       #␣
↪needs palp
Traceback (most recent call last):
...
ValueError: nef-partitions can be computed for reflexive polytopes only
```

```
>>> from sage.all import *
>>> p = LatticePolytope([(Integer(1),Integer(0),Integer(0)), (Integer(0),
↪Integer(1),Integer(0)), (Integer(0),Integer(0),Integer(2)),
...                      (-Integer(1),Integer(0),Integer(0)), (Integer(0),-
↪Integer(1),Integer(0)), (Integer(0),Integer(0),-Integer(1))])
>>> lattice_polytope.all_nef_partitions([o, p])                        #␣
↪needs palp
Traceback (most recent call last):
...
ValueError: nef-partitions can be computed for reflexive polytopes only
```

sage.geometry.lattice_polytope.**all_points**(*polytopes*)

Compute lattice points for all given `polytopes`.

This functions does it MUCH faster than member functions of `LatticePolytope` during the first run. So it is recommended to use this functions if you work with big sets of data.

INPUT:

- `polytopes` – a sequence of lattice polytopes

EXAMPLES: This function has no output, it is just a fast way to work with long sequences of polytopes. Of course, you can use short sequences as well:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: lattice_polytope.all_points([o])                                          #␣
↪needs palp
sage: o.points()                                                                #␣
↪needs palp
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1),
M( 0,  0,  0)
in 3-d lattice M
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> lattice_polytope.all_points([o])                                          #␣
↪needs palp
>>> o.points()                                                                #␣
↪needs palp
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1),
M( 0,  0,  0)
in 3-d lattice M
```

sage.geometry.lattice_polytope.**all_polars**(*polytopes*)

Compute polar polytopes for all reflexive and equations of facets for all non-reflexive `polytopes`.

`all_facet_equations` and `all_polars` are synonyms.

This functions does it MUCH faster than member functions of `LatticePolytope` during the first run. So it is recommended to use this functions if you work with big sets of data.

INPUT:

- `polytopes` – a sequence of lattice polytopes

EXAMPLES: This function has no output, it is just a fast way to work with long sequences of polytopes. Of course, you can use short sequences as well:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: lattice_polytope.all_polars([o])                                          #␣
↪needs palp
sage: o.polar()                                                                 #␣
↪needs palp
3-d reflexive polytope in 3-d lattice N
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> lattice_polytope.all_polars([o])                                        #␣
↪needs palp
>>> o.polar()                                                               #␣
↪needs palp
3-d reflexive polytope in 3-d lattice N
```

`sage.geometry.lattice_polytope.`**`convex_hull`**`(`*points*`)`

Compute the convex hull of the given points.

> **Note**
>
> `points` might not span the space. Also, it fails for large numbers of vertices in dimensions 4 or greater

INPUT:

- `points` – list that can be converted into vectors of the same dimension over **Z**

OUTPUT: list of vertices of the convex hull of the given points (as vectors)

EXAMPLES: Let's compute the convex hull of several points on a line in the plane:

```
sage: lattice_polytope.convex_hull([[1,2],[3,4],[5,6],[7,8]])
[(1, 2), (7, 8)]
```

```
>>> from sage.all import *
>>> lattice_polytope.convex_hull([[Integer(1),Integer(2)],[Integer(3),Integer(4)],
↪[Integer(5),Integer(6)],[Integer(7),Integer(8)]])
[(1, 2), (7, 8)]
```

`sage.geometry.lattice_polytope.`**`cross_polytope`**`(`*dim*`)`

Return a cross-polytope of the given dimension.

INPUT:

- `dim` – integer

OUTPUT: a *`lattice polytope`*

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o
3-d reflexive polytope in 3-d lattice M
sage: o.vertices()
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1)
in 3-d lattice M
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> o
3-d reflexive polytope in 3-d lattice M
>>> o.vertices()
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1)
in 3-d lattice M
```

`sage.geometry.lattice_polytope.`**`is_LatticePolytope`**`(x)`

Check if `x` is a lattice polytope.

INPUT:

- `x` – anything

OUTPUT:

- `True` if `x` is a *lattice polytope*, `False` otherwise.

EXAMPLES:

```
sage: from sage.geometry.lattice_polytope import is_LatticePolytope
sage: is_LatticePolytope(1)
doctest:warning...
DeprecationWarning: is_LatticePolytope is deprecated, use isinstance instead
See https://github.com/sagemath/sage/issues/34307 for details.
False
sage: p = LatticePolytope([(1,0), (0,1), (-1,-1)])
sage: p                                                                  #␣
↪needs palp
2-d reflexive polytope #0 in 2-d lattice M
sage: is_LatticePolytope(p)
True
```

```
>>> from sage.all import *
>>> from sage.geometry.lattice_polytope import is_LatticePolytope
>>> is_LatticePolytope(Integer(1))
doctest:warning...
DeprecationWarning: is_LatticePolytope is deprecated, use isinstance instead
See https://github.com/sagemath/sage/issues/34307 for details.
False
>>> p = LatticePolytope([(Integer(1),Integer(0)), (Integer(0),Integer(1)), (-
↪Integer(1),-Integer(1))])
>>> p                                                                    #␣
↪needs palp
2-d reflexive polytope #0 in 2-d lattice M
>>> is_LatticePolytope(p)
True
```

`sage.geometry.lattice_polytope.`**`is_NefPartition`**`(x)`

Check if `x` is a nef-partition.

INPUT:

- `x` – anything

OUTPUT:

- `True` if `x` is a *nef-partition* and `False` otherwise.

EXAMPLES:

```
sage: from sage.geometry.lattice_polytope import NefPartition
sage: isinstance(1, NefPartition)
False
sage: o = lattice_polytope.cross_polytope(3)
sage: np = o.nef_partitions()[0]; np                                    #␣
↪needs palp
Nef-partition {0, 1, 3} ⊔ {2, 4, 5}
sage: isinstance(np, NefPartition)                                      #␣
↪needs palp
True
```

```
>>> from sage.all import *
>>> from sage.geometry.lattice_polytope import NefPartition
>>> isinstance(Integer(1), NefPartition)
False
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> np = o.nef_partitions()[Integer(0)]; np                             ␣
↪     # needs palp
Nef-partition {0, 1, 3} ⊔ {2, 4, 5}
>>> isinstance(np, NefPartition)                                      #␣
↪needs palp
True
```

`sage.geometry.lattice_polytope.`**`minkowski_sum`**`(`*points1*, *points2*`)`

Compute the Minkowski sum of two convex polytopes.

Note

Polytopes might not be of maximal dimension.

INPUT:

- `points1`, `points2` – lists of objects that can be converted into vectors of the same dimension, treated as vertices of two polytopes.

OUTPUT: list of vertices of the Minkowski sum, given as vectors

EXAMPLES: Let's compute the Minkowski sum of two line segments:

```
sage: lattice_polytope.minkowski_sum([[1,0],[-1,0]],[[0,1],[0,-1]])
[(1, 1), (1, -1), (-1, 1), (-1, -1)]
```

```
>>> from sage.all import *
>>> lattice_polytope.minkowski_sum([[Integer(1),Integer(0)],[-Integer(1),
↪Integer(0)]],[[Integer(0),Integer(1)],[Integer(0),-Integer(1)]])
[(1, 1), (1, -1), (-1, 1), (-1, -1)]
```

sage.geometry.lattice_polytope.**positive_integer_relations**(*points*)

Return relations between given points.

INPUT:

- `points` – lattice points given as columns of a matrix

OUTPUT: matrix of relations between given points with nonnegative integer coefficients

EXAMPLES: This is a 3-dimensional reflexive polytope:

```
sage: p = LatticePolytope([(1,0,0), (0,1,0),
....:                      (-1,-1,0), (0,0,1), (-1,0,-1)])
sage: p.points()                                                            #␣
↪needs palp
M( 1,  0,  0),
M( 0,  1,  0),
M(-1, -1,  0),
M( 0,  0,  1),
M(-1,  0, -1),
M( 0,  0,  0)
in 3-d lattice M
```

```
>>> from sage.all import *
>>> p = LatticePolytope([(Integer(1),Integer(0),Integer(0)), (Integer(0),
↪Integer(1),Integer(0)),
...                      (-Integer(1),-Integer(1),Integer(0)), (Integer(0),
↪Integer(0),Integer(1)), (-Integer(1),Integer(0),-Integer(1))])
>>> p.points()                                                              #␣
↪needs palp
M( 1,  0,  0),
M( 0,  1,  0),
M(-1, -1,  0),
M( 0,  0,  1),
M(-1,  0, -1),
M( 0,  0,  0)
in 3-d lattice M
```

We can compute linear relations between its points in the following way:

```
sage: p.points().matrix().kernel().echelonized_basis_matrix()               #␣
↪needs palp
[ 1  0  0  1  1  0]
[ 0  1  1 -1 -1  0]
[ 0  0  0  0  0  1]
```

```
>>> from sage.all import *
>>> p.points().matrix().kernel().echelonized_basis_matrix()                 #␣
↪needs palp
[ 1  0  0  1  1  0]
[ 0  1  1 -1 -1  0]
[ 0  0  0  0  0  1]
```

However, the above relations may contain negative and rational numbers. This function transforms them in such a way, that all coefficients are nonnegative integers:

```
sage: points = p.points().column_matrix()
sage: lattice_polytope.positive_integer_relations(points)                    #␣
↪needs palp
[1 0 0 1 1 0]
[1 1 1 0 0 0]
[0 0 0 0 0 1]

sage: cm = ReflexivePolytope(2,1).vertices().column_matrix()
sage: lattice_polytope.positive_integer_relations(cm)
[2 1 1]
```

```
>>> from sage.all import *
>>> points = p.points().column_matrix()
>>> lattice_polytope.positive_integer_relations(points)                    #␣
↪needs palp
[1 0 0 1 1 0]
[1 1 1 0 0 0]
[0 0 0 0 0 1]

>>> cm = ReflexivePolytope(Integer(2),Integer(1)).vertices().column_matrix()
>>> lattice_polytope.positive_integer_relations(cm)
[2 1 1]
```

sage.geometry.lattice_polytope.**read_all_polytopes**(*file_name*)

Read all polytopes from the given file.

INPUT:

- `file_name` – string with the name of a file with VERTICES of polytopes

OUTPUT: a sequence of polytopes

EXAMPLES:

We use `poly.x` to compute two polar polytopes and read them:

```
sage: # needs palp
sage: d = lattice_polytope.cross_polytope(2)
sage: o = lattice_polytope.cross_polytope(3)
sage: result_name = lattice_polytope._palp("poly.x -fe", [d, o])
sage: with open(result_name) as f:
....:     print(f.read())
4 2  Vertices of P-dual <-> Equations of P
  -1   1
   1   1
  -1  -1
   1  -1
8 3  Vertices of P-dual <-> Equations of P
  -1  -1   1
   1  -1   1
  -1   1   1
   1   1   1
  -1  -1  -1
   1  -1  -1
  -1   1  -1
```

(continues on next page)

(continued from previous page)

```
   1   1  -1
sage: lattice_polytope.read_all_polytopes(result_name)
[2-d reflexive polytope #14 in 2-d lattice M,
 3-d reflexive polytope in 3-d lattice M]
sage: os.remove(result_name)
```

```
>>> from sage.all import *
>>> # needs palp
>>> d = lattice_polytope.cross_polytope(Integer(2))
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> result_name = lattice_polytope._palp("poly.x -fe", [d, o])
>>> with open(result_name) as f:
...     print(f.read())
4 2  Vertices of P-dual <-> Equations of P
  -1   1
   1   1
  -1  -1
   1  -1
8 3  Vertices of P-dual <-> Equations of P
  -1  -1   1
   1  -1   1
  -1   1   1
   1   1   1
  -1  -1  -1
   1  -1  -1
  -1   1  -1
   1   1  -1
>>> lattice_polytope.read_all_polytopes(result_name)
[2-d reflexive polytope #14 in 2-d lattice M,
 3-d reflexive polytope in 3-d lattice M]
>>> os.remove(result_name)
```

sage.geometry.lattice_polytope.**read_palp_matrix**(*data*, *permutation=False*)

Read and return an integer matrix from a string or an opened file.

First input line must start with two integers m and n, the number of rows and columns of the matrix. The rest of the first line is ignored. The next m lines must contain n numbers each.

If m>n, returns the transposed matrix. If the string is empty or EOF is reached, returns the empty matrix, constructed by `matrix()`.

INPUT:

- `data` – either a string containing the filename or the file itself containing the output by PALP
- `permutation` – boolean (default: `False`); if `True`, try to retrieve the permutation output by PALP. This parameter makes sense only when PALP computed the normal form of a lattice polytope.

OUTPUT: a matrix or a tuple of a matrix and a permutation

EXAMPLES:

```
sage: lattice_polytope.read_palp_matrix("2 3 comment \n 1 2 3 \n 4 5 6")
[1 2 3]
[4 5 6]
```

(continues on next page)

(continued from previous page)

```
sage: lattice_polytope.read_palp_matrix("3 2 Will be transposed \n 1 2 \n 3 4 \n␣
↪5 6")
[1 3 5]
[2 4 6]
```

```
>>> from sage.all import *
>>> lattice_polytope.read_palp_matrix("2 3 comment \n 1 2 3 \n 4 5 6")
[1 2 3]
[4 5 6]
>>> lattice_polytope.read_palp_matrix("3 2 Will be transposed \n 1 2 \n 3 4 \n 5 6
↪")
[1 3 5]
[2 4 6]
```

sage.geometry.lattice_polytope.**set_palp_dimension**(*d*)

Set the dimension for PALP calls to d.

INPUT:

- d – integer from the list [4,5,6,11] or None

OUTPUT: none

PALP has many hard-coded limits, which must be specified before compilation, one of them is dimension. Sage includes several versions with different dimension settings (which may also affect other limits and enable certain features of PALP). You can change the version which will be used by calling this function. Such a change is not done automatically for each polytope based on its dimension, since depending on what you are doing it may be necessary to use dimensions higher than that of the input polytope.

EXAMPLES:

Let's try to work with a 7-dimensional polytope:

```
sage: p = lattice_polytope.cross_polytope(7)
sage: p._palp("poly.x -fv")                                                  #␣
↪needs palp
Traceback (most recent call last):
...
ValueError: Error executing 'poly.x -fv' for the given polytope!
Output:
Please increase POLY_Dmax to at least 7
```

```
>>> from sage.all import *
>>> p = lattice_polytope.cross_polytope(Integer(7))
>>> p._palp("poly.x -fv")                                                   #␣
↪needs palp
Traceback (most recent call last):
...
ValueError: Error executing 'poly.x -fv' for the given polytope!
Output:
Please increase POLY_Dmax to at least 7
```

However, we can work with this polytope by changing PALP dimension to 11:

```
sage: lattice_polytope.set_palp_dimension(11)
sage: p._palp("poly.x -fv")                                              #␣
↪needs palp
'7 14  Vertices of P...'
```

```
>>> from sage.all import *
>>> lattice_polytope.set_palp_dimension(Integer(11))
>>> p._palp("poly.x -fv")                                               #␣
↪needs palp
'7 14  Vertices of P...'
```

Let's go back to default settings:

```
sage: lattice_polytope.set_palp_dimension(None)
```

```
>>> from sage.all import *
>>> lattice_polytope.set_palp_dimension(None)
```

`sage.geometry.lattice_polytope.`**`skip_palp_matrix`**`(`*data*, *n=1*`)`

Skip matrix data in a file.

INPUT:

- `data` – opened file with blocks of matrix data in the following format: A block consisting of m+1 lines has the number m as the first element of its first line.
- `n` – (default: 1) integer, specifies how many blocks should be skipped

If EOF is reached during the process, raises `ValueError` exception.

EXAMPLES: We create a file with vertices of the square and the cube, but read only the second set:

```
sage: # needs palp
sage: d = lattice_polytope.cross_polytope(2)
sage: o = lattice_polytope.cross_polytope(3)
sage: result_name = lattice_polytope._palp("poly.x -fe", [d, o])
sage: with open(result_name) as f:
....:     print(f.read())
4 2  Vertices of P-dual <-> Equations of P
  -1   1
   1   1
  -1  -1
   1  -1
8 3  Vertices of P-dual <-> Equations of P
  -1  -1   1
   1  -1   1
  -1   1   1
   1   1   1
  -1  -1  -1
   1  -1  -1
  -1   1  -1
   1   1  -1
sage: f = open(result_name)
sage: lattice_polytope.skip_palp_matrix(f)
sage: lattice_polytope.read_palp_matrix(f)
```

(continues on next page)

(continued from previous page)

```
[-1  1 -1  1 -1  1 -1  1]
[-1 -1  1  1 -1 -1  1  1]
[ 1  1  1  1 -1 -1 -1 -1]
sage: f.close()
sage: os.remove(result_name)
```

```
>>> from sage.all import *
>>> # needs palp
>>> d = lattice_polytope.cross_polytope(Integer(2))
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> result_name = lattice_polytope._palp("poly.x -fe", [d, o])
>>> with open(result_name) as f:
...     print(f.read())
4 2  Vertices of P-dual <-> Equations of P
  -1   1
   1   1
  -1  -1
   1  -1
8 3  Vertices of P-dual <-> Equations of P
  -1  -1   1
   1  -1   1
  -1   1   1
   1   1   1
  -1  -1  -1
   1  -1  -1
  -1   1  -1
   1   1  -1
>>> f = open(result_name)
>>> lattice_polytope.skip_palp_matrix(f)
>>> lattice_polytope.read_palp_matrix(f)
[-1  1 -1  1 -1  1 -1  1]
[-1 -1  1  1 -1 -1  1  1]
[ 1  1  1  1 -1 -1 -1 -1]
>>> f.close()
>>> os.remove(result_name)
```

sage.geometry.lattice_polytope.**write_palp_matrix**(*m*, *ofile=None*, *comment=''*, *format=None*)

Write `m` into `ofile` in PALP format.

INPUT:

- `m` – a matrix over integers or a *point collection*
- `ofile` – a file opened for writing (default: `stdout`)
- `comment` – string (default: empty); see output description
- `format` – a format string used to print matrix entries

OUTPUT: nothing is returned, output written to `ofile` has the format

- First line: number_of_rows number_of_columns comment
- Next number_of_rows lines: rows of the matrix.

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: lattice_polytope.write_palp_matrix(o.vertices(), comment="3D Octahedron")
3 6 3D Octahedron
 1  0  0 -1  0  0
 0  1  0  0 -1  0
 0  0  1  0  0 -1
sage: lattice_polytope.write_palp_matrix(o.vertices(), format='%4d')
3 6
   1    0    0   -1    0    0
   0    1    0    0   -1    0
   0    0    1    0    0   -1
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> lattice_polytope.write_palp_matrix(o.vertices(), comment="3D Octahedron")
3 6 3D Octahedron
 1  0  0 -1  0  0
 0  1  0  0 -1  0
 0  0  1  0  0 -1
>>> lattice_polytope.write_palp_matrix(o.vertices(), format='%4d')
3 6
   1    0    0   -1    0    0
   0    1    0    0   -1    0
   0    0    1    0    0   -1
```

2.2.2 Lattice Euclidean Group Elements

The classes here are used to return particular isomorphisms of `PPL lattice polytopes`.

class sage.geometry.polyhedron.lattice_euclidean_group_element.**LatticeEuclideanGroupElement**(A, b)

Bases: `SageObject`

An element of the lattice Euclidean group.

Note that this is just intended as a container for results from LatticePolytope_PPL. There is no group-theoretic functionality to speak of.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL, C_Polyhedron
sage: from sage.geometry.polyhedron.lattice_euclidean_group_element import␣
↪LatticeEuclideanGroupElement
sage: M = LatticeEuclideanGroupElement([[1,2],[2,3],[-1,2]], [1,2,3])
sage: M
The map A*x+b with A=
[ 1  2]
[ 2  3]
[-1  2]
b =
(1, 2, 3)
sage: M._A
[ 1  2]
```

(continues on next page)

(continued from previous page)

```
[ 2  3]
[-1  2]
sage: M._b
(1, 2, 3)
sage: M(vector([0,0]))
(1, 2, 3)
sage: M(LatticePolytope_PPL((0,0),(1,0),(0,1)))
A 2-dimensional lattice polytope in ZZ^3 with 3 vertices
sage: _.vertices()
((1, 2, 3), (2, 4, 2), (3, 5, 5))
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_PPL,
↪ C_Polyhedron
>>> from sage.geometry.polyhedron.lattice_euclidean_group_element import␣
↪LatticeEuclideanGroupElement
>>> M = LatticeEuclideanGroupElement([[Integer(1),Integer(2)],[Integer(2),
↪Integer(3)],[-Integer(1),Integer(2)]], [Integer(1),Integer(2),Integer(3)])
>>> M
The map A*x+b with A=
[ 1  2]
[ 2  3]
[-1  2]
b =
(1, 2, 3)
>>> M._A
[ 1  2]
[ 2  3]
[-1  2]
>>> M._b
(1, 2, 3)
>>> M(vector([Integer(0),Integer(0)]))
(1, 2, 3)
>>> M(LatticePolytope_PPL((Integer(0),Integer(0)),(Integer(1),Integer(0)),
↪(Integer(0),Integer(1))))
A 2-dimensional lattice polytope in ZZ^3 with 3 vertices
>>> _.vertices()
((1, 2, 3), (2, 4, 2), (3, 5, 5))
```

codomain_dim()

Return the dimension of the codomain lattice.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.lattice_euclidean_group_element import␣
↪LatticeEuclideanGroupElement
sage: M = LatticeEuclideanGroupElement([[1,2],[2,3],[-1,2]], [1,2,3])
sage: M
The map A*x+b with A=
[ 1  2]
[ 2  3]
[-1  2]
```

(continues on next page)

(continued from previous page)

```
b =
(1, 2, 3)
sage: M.codomain_dim()
3
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.lattice_euclidean_group_element import␣
↪LatticeEuclideanGroupElement
>>> M = LatticeEuclideanGroupElement([[Integer(1),Integer(2)],[Integer(2),
↪Integer(3)],[-Integer(1),Integer(2)]], [Integer(1),Integer(2),Integer(3)])
>>> M
The map A*x+b with A=
[ 1  2]
[ 2  3]
[-1  2]
b =
(1, 2, 3)
>>> M.codomain_dim()
3
```

Note that this is not the same as the rank. In fact, the codomain dimension depends only on the matrix shape, and not on the rank of the linear mapping:

```
sage: zero_map = LatticeEuclideanGroupElement([[0,0],[0,0],[0,0]], [0,0,0])
sage: zero_map.codomain_dim()
3
```

```
>>> from sage.all import *
>>> zero_map = LatticeEuclideanGroupElement([[Integer(0),Integer(0)],
↪[Integer(0),Integer(0)],[Integer(0),Integer(0)]], [Integer(0),Integer(0),
↪Integer(0)])
>>> zero_map.codomain_dim()
3
```

domain_dim()

Return the dimension of the domain lattice.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.lattice_euclidean_group_element import␣
↪LatticeEuclideanGroupElement
sage: M = LatticeEuclideanGroupElement([[1,2],[2,3],[-1,2]], [1,2,3])
sage: M
The map A*x+b with A=
[ 1  2]
[ 2  3]
[-1  2]
b =
(1, 2, 3)
sage: M.domain_dim()
2
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.lattice_euclidean_group_element import␣
↪LatticeEuclideanGroupElement
>>> M = LatticeEuclideanGroupElement([[Integer(1),Integer(2)],[Integer(2),
↪Integer(3)],[-Integer(1),Integer(2)]], [Integer(1),Integer(2),Integer(3)])
>>> M
The map A*x+b with A=
[ 1  2]
[ 2  3]
[-1  2]
b =
(1, 2, 3)
>>> M.domain_dim()
2
```

exception sage.geometry.polyhedron.lattice_euclidean_group_element.**LatticePolytopeError**

Bases: `Exception`

Base class for errors from lattice polytopes

exception sage.geometry.polyhedron.lattice_euclidean_group_element.
LatticePolytopeNoEmbeddingError

Bases: *LatticePolytopeError*

Raised when no embedding of the desired kind can be found.

exception sage.geometry.polyhedron.lattice_euclidean_group_element.
LatticePolytopesNotIsomorphicError

Bases: *LatticePolytopeError*

Raised when two lattice polytopes are not isomorphic.

2.2.3 Access the PALP database(s) of reflexive lattice polytopes

EXAMPLES:

```
sage: from sage.geometry.polyhedron.palp_database import PALPreader
sage: for lp in PALPreader(2):                                          #␣
↪needs sage.graphs
....:     cone = Cone([(1,r[0],r[1]) for r in lp.vertices()])
....:     fan = Fan([cone])
....:     X = ToricVariety(fan)
....:     ideal = X.affine_algebraic_patch(cone).defining_ideal()
....:     print("{} {}".format(lp.n_vertices(), ideal.hilbert_series()))
3 (t^2 + 7*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
3 (t^2 + t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
3 (t^2 + 6*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
3 (t^2 + 2*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
3 (t^2 + 4*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
4 (t^2 + 5*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
4 (t^2 + 3*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
4 (t^2 + 2*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
4 (t^2 + 6*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
4 (t^2 + 6*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
4 (t^2 + 2*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
```

(continues on next page)

(continued from previous page)

```
4 (t^2 + 4*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
5 (t^2 + 3*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
5 (t^2 + 5*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
5 (t^2 + 4*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
6 (t^2 + 4*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.palp_database import PALPreader
>>> for lp in PALPreader(Integer(2)):                                            ␣
↪    # needs sage.graphs
...     cone = Cone([(Integer(1),r[Integer(0)],r[Integer(1)]) for r in lp.vertices()])
...     fan = Fan([cone])
...     X = ToricVariety(fan)
...     ideal = X.affine_algebraic_patch(cone).defining_ideal()
...     print("{} {}".format(lp.n_vertices(), ideal.hilbert_series()))
3 (t^2 + 7*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
3 (t^2 + t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
3 (t^2 + 6*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
3 (t^2 + 2*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
3 (t^2 + 4*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
4 (t^2 + 5*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
4 (t^2 + 3*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
4 (t^2 + 2*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
4 (t^2 + 6*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
4 (t^2 + 6*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
4 (t^2 + 2*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
4 (t^2 + 4*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
5 (t^2 + 3*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
5 (t^2 + 5*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
5 (t^2 + 4*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
6 (t^2 + 4*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
```

class sage.geometry.polyhedron.palp_database.**PALPreader**(*dim*, *data_basename=None*, *output='Polyhedron'*)

Bases: SageObject

Read PALP database of polytopes.

INPUT:

- dim – integer; the dimension of the polyhedra
- data_basename – string or None (default). The directory and database base filename (PALP usually uses 'zzdb') name containing the PALP database to read. Defaults to the built-in database location.
- output – string. How to return the reflexive polyhedron data. Allowed values = 'list', 'Polyhedron' (default), 'pointcollection', and 'PPL'. Case is ignored.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.palp_database import PALPreader
sage: polygons = PALPreader(2)
sage: [ (p.n_Vrepresentation(), len(p.integral_points())) for p in polygons ]
[(3, 4), (3, 10), (3, 5), (3, 9), (3, 7), (4, 6), (4, 8), (4, 9),
 (4, 5), (4, 5), (4, 9), (4, 7), (5, 8), (5, 6), (5, 7), (6, 7)]
```

(continues on next page)

(continued from previous page)

```
sage: next(iter(PALPreader(2, output='list')))
[[1, 0], [0, 1], [-1, -1]]
sage: type(_)
<... 'list'>

sage: next(iter(PALPreader(2, output='Polyhedron')))
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 3 vertices
sage: type(_)
<class 'sage.geometry.polyhedron.parent.Polyhedra_ZZ_ppl_with_category.element_
↪class'>

sage: next(iter(PALPreader(2, output='PPL')))
A 2-dimensional lattice polytope in ZZ^2 with 3 vertices
sage: type(_)
<class 'sage.geometry.polyhedron.ppl_lattice_polygon.LatticePolygon_PPL_class'>

sage: next(iter(PALPreader(2, output='PointCollection')))
[ 1,  0],
[ 0,  1],
[-1, -1]
in Ambient free module of rank 2 over the principal ideal domain Integer Ring
sage: type(_)
<class 'sage.geometry.point_collection.PointCollection'>
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.palp_database import PALPreader
>>> polygons = PALPreader(Integer(2))
>>> [ (p.n_Vrepresentation(), len(p.integral_points())) for p in polygons ]
[(3, 4), (3, 10), (3, 5), (3, 9), (3, 7), (4, 6), (4, 8), (4, 9),
 (4, 5), (4, 5), (4, 9), (4, 7), (5, 8), (5, 6), (5, 7), (6, 7)]

>>> next(iter(PALPreader(Integer(2), output='list')))
[[1, 0], [0, 1], [-1, -1]]
>>> type(_)
<... 'list'>

>>> next(iter(PALPreader(Integer(2), output='Polyhedron')))
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 3 vertices
>>> type(_)
<class 'sage.geometry.polyhedron.parent.Polyhedra_ZZ_ppl_with_category.element_
↪class'>

>>> next(iter(PALPreader(Integer(2), output='PPL')))
A 2-dimensional lattice polytope in ZZ^2 with 3 vertices
>>> type(_)
<class 'sage.geometry.polyhedron.ppl_lattice_polygon.LatticePolygon_PPL_class'>

>>> next(iter(PALPreader(Integer(2), output='PointCollection')))
[ 1,  0],
[ 0,  1],
[-1, -1]
```

(continues on next page)

(continued from previous page)

```
in Ambient free module of rank 2 over the principal ideal domain Integer Ring
>>> type(_)
<class 'sage.geometry.point_collection.PointCollection'>
```

class sage.geometry.polyhedron.palp_database.**Reflexive4dHodge**(*h11*, *h21*, *data_basename=None*, ***kwds*)

Bases: *PALPreader*

Read the PALP database for Hodge numbers of 4d polytopes.

The database is very large and not installed by default. You can install it with the shell command `sage -i polytopes_db_4d`.

INPUT:

- `h11`, `h21` – integer; the Hodge numbers of the reflexive polytopes to list

Any additional keyword arguments are passed to *PALPreader*.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.palp_database import Reflexive4dHodge
sage: ref = Reflexive4dHodge(1,101)             # optional - polytopes_db_4d
sage: next(iter(ref)).Vrepresentation()          # optional - polytopes_db_4d
(A vertex at (-1, -1, -1, -1), A vertex at (0, 0, 0, 1),
 A vertex at (0, 0, 1, 0), A vertex at (0, 1, 0, 0), A vertex at (1, 0, 0, 0))
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.palp_database import Reflexive4dHodge
>>> ref = Reflexive4dHodge(Integer(1),Integer(101))            # optional -
↪polytopes_db_4d
>>> next(iter(ref)).Vrepresentation()          # optional - polytopes_db_4d
(A vertex at (-1, -1, -1, -1), A vertex at (0, 0, 0, 1),
 A vertex at (0, 0, 1, 0), A vertex at (0, 1, 0, 0), A vertex at (1, 0, 0, 0))
```

2.2.4 Fast Lattice Polygons using PPL

See `ppl_lattice_polytope` for the implementation of arbitrary-dimensional lattice polytopes. This module is about the specialization to 2 dimensions. To be more precise, the *LatticePolygon_PPL_class* is used if the ambient space is of dimension 2 or less. These all allow you to cyclically order (see *LatticePolygon_PPL_class.ordered_vertices()*) the vertices, which is in general not possible in higher dimensions.

class sage.geometry.polyhedron.ppl_lattice_polygon.**LatticePolygon_PPL_class**

Bases: *LatticePolytope_PPL_class*

A lattice polygon.

This includes 2-dimensional polytopes as well as degenerate (0 and 1-dimensional) lattice polygons. Any polytope in 2d is a polygon.

find_isomorphism(*polytope*)

Return a lattice isomorphism with `polytope`.

INPUT:

- `polytope` – a polytope, potentially higher-dimensional

OUTPUT:

A *LatticeEuclideanGroupElement*. It is not necessarily invertible if the affine dimension of `self` or `polytope` is not two. A *LatticePolytopesNotIsomorphicError* is raised if no such isomorphism exists.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import
↪LatticePolytope_PPL
sage: L1 = LatticePolytope_PPL((1,0),(0,1),(0,0))
sage: L2 = LatticePolytope_PPL((1,0,3),(0,1,0),(0,0,1))
sage: iso = L1.find_isomorphism(L2)
sage: iso(L1) == L2
True

sage: L1 = LatticePolytope_PPL((0, 1), (3, 0), (0, 3), (1, 0))
sage: L2 = LatticePolytope_PPL((0,0,2,1),(0,1,2,0),(2,0,0,3),(2,3,0,0))
sage: iso = L1.find_isomorphism(L2)
sage: iso(L1) == L2
True
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> L1 = LatticePolytope_PPL((Integer(1),Integer(0)),(Integer(0),Integer(1)),
↪(Integer(0),Integer(0)))
>>> L2 = LatticePolytope_PPL((Integer(1),Integer(0),Integer(3)),(Integer(0),
↪Integer(1),Integer(0)),(Integer(0),Integer(0),Integer(1)))
>>> iso = L1.find_isomorphism(L2)
>>> iso(L1) == L2
True

>>> L1 = LatticePolytope_PPL((Integer(0), Integer(1)), (Integer(3),
↪Integer(0)), (Integer(0), Integer(3)), (Integer(1), Integer(0)))
>>> L2 = LatticePolytope_PPL((Integer(0),Integer(0),Integer(2),Integer(1)),
↪(Integer(0),Integer(1),Integer(2),Integer(0)),(Integer(2),Integer(0),
↪Integer(0),Integer(3)),(Integer(2),Integer(3),Integer(0),Integer(0)))
>>> iso = L1.find_isomorphism(L2)
>>> iso(L1) == L2
True
```

The following polygons are isomorphic over **Q**, but not as lattice polytopes:

```
sage: L1 = LatticePolytope_PPL((1,0),(0,1),(-1,-1))
sage: L2 = LatticePolytope_PPL((0, 0), (0, 1), (1, 0))
sage: L1.find_isomorphism(L2)
Traceback (most recent call last):
...
LatticePolytopesNotIsomorphicError: different number of integral points
sage: L2.find_isomorphism(L1)
Traceback (most recent call last):
...
LatticePolytopesNotIsomorphicError: different number of integral points
```

```
>>> from sage.all import *
>>> L1 = LatticePolytope_PPL((Integer(1),Integer(0)),(Integer(0),Integer(1)),
↪(-Integer(1),-Integer(1)))
>>> L2 = LatticePolytope_PPL((Integer(0), Integer(0)), (Integer(0),␣
↪Integer(1)), (Integer(1), Integer(0)))
>>> L1.find_isomorphism(L2)
Traceback (most recent call last):
...
LatticePolytopesNotIsomorphicError: different number of integral points
>>> L2.find_isomorphism(L1)
Traceback (most recent call last):
...
LatticePolytopesNotIsomorphicError: different number of integral points
```

is_isomorphic(*polytope*)

Test if `self` and `polytope` are isomorphic.

INPUT:

- `polytope` – a lattice polytope

OUTPUT: boolean

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import␣
↪LatticePolytope_PPL
sage: L1 = LatticePolytope_PPL((1,0),(0,1),(0,0))
sage: L2 = LatticePolytope_PPL((1,0,3),(0,1,0),(0,0,1))
sage: L1.is_isomorphic(L2)
True
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> L1 = LatticePolytope_PPL((Integer(1),Integer(0)),(Integer(0),Integer(1)),
↪(Integer(0),Integer(0)))
>>> L2 = LatticePolytope_PPL((Integer(1),Integer(0),Integer(3)),(Integer(0),
↪Integer(1),Integer(0)),(Integer(0),Integer(0),Integer(1)))
>>> L1.is_isomorphic(L2)
True
```

ordered_vertices()

Return the vertices of a lattice polygon in cyclic order.

OUTPUT:

A tuple of vertices ordered along the perimeter of the polygon. The first point is arbitrary.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import␣
↪LatticePolytope_PPL
sage: square = LatticePolytope_PPL((0,0), (1,1), (0,1), (1,0))
sage: square.vertices()
((0, 0), (0, 1), (1, 0), (1, 1))
```

(continues on next page)

(continued from previous page)

```
sage: square.ordered_vertices()
((0, 0), (1, 0), (1, 1), (0, 1))
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> square = LatticePolytope_PPL((Integer(0),Integer(0)), (Integer(1),
↪Integer(1)), (Integer(0),Integer(1)), (Integer(1),Integer(0)))
>>> square.vertices()
((0, 0), (0, 1), (1, 0), (1, 1))
>>> square.ordered_vertices()
((0, 0), (1, 0), (1, 1), (0, 1))
```

plot()

Plot the lattice polygon.

OUTPUT: a graphics object

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import␣
↪LatticePolytope_PPL
sage: P = LatticePolytope_PPL((1,0), (0,1), (0,0), (2,2))
sage: P.plot()                                                    #␣
↪needs sage.plot
Graphics object consisting of 6 graphics primitives
sage: LatticePolytope_PPL([0], [1]).plot()                        #␣
↪needs sage.plot
Graphics object consisting of 3 graphics primitives
sage: LatticePolytope_PPL([0]).plot()                             #␣
↪needs sage.plot
Graphics object consisting of 2 graphics primitives
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> P = LatticePolytope_PPL((Integer(1),Integer(0)), (Integer(0),Integer(1)),␣
↪(Integer(0),Integer(0)), (Integer(2),Integer(2)))
>>> P.plot()                                                    #␣
↪needs sage.plot
Graphics object consisting of 6 graphics primitives
>>> LatticePolytope_PPL([Integer(0)], [Integer(1)]).plot()            ␣
↪               # needs sage.plot
Graphics object consisting of 3 graphics primitives
>>> LatticePolytope_PPL([Integer(0)]).plot()                          ␣
↪      # needs sage.plot
Graphics object consisting of 2 graphics primitives
```

sub_polytopes()

Return a list of all lattice sub-polygons up to isomorphism.

OUTPUT:

All non-empty sub-lattice polytopes up to isomorphism. This includes `self` as improper sub-polytope, but excludes the empty polytope. Isomorphic sub-polytopes that can be embedded in different places are only returned once.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import␣
↪LatticePolytope_PPL
sage: P1xP1 = LatticePolytope_PPL((1,0), (0,1), (-1,0), (0,-1))
sage: P1xP1.sub_polytopes()
(A 2-dimensional lattice polytope in ZZ^2 with 4 vertices,
 A 2-dimensional lattice polytope in ZZ^2 with 3 vertices,
 A 2-dimensional lattice polytope in ZZ^2 with 3 vertices,
 A 1-dimensional lattice polytope in ZZ^2 with 2 vertices,
 A 1-dimensional lattice polytope in ZZ^2 with 2 vertices,
 A 0-dimensional lattice polytope in ZZ^2 with 1 vertex)
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> P1xP1 = LatticePolytope_PPL((Integer(1),Integer(0)), (Integer(0),
↪Integer(1)), (-Integer(1),Integer(0)), (Integer(0),-Integer(1)))
>>> P1xP1.sub_polytopes()
(A 2-dimensional lattice polytope in ZZ^2 with 4 vertices,
 A 2-dimensional lattice polytope in ZZ^2 with 3 vertices,
 A 2-dimensional lattice polytope in ZZ^2 with 3 vertices,
 A 1-dimensional lattice polytope in ZZ^2 with 2 vertices,
 A 1-dimensional lattice polytope in ZZ^2 with 2 vertices,
 A 0-dimensional lattice polytope in ZZ^2 with 1 vertex)
```

sage.geometry.polyhedron.ppl_lattice_polygon.**polar_P1xP1_polytope**()

The polar of the $P^1 \times P^1$ polytope.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polygon import polar_P1xP1_
↪polytope
sage: polar_P1xP1_polytope()
A 2-dimensional lattice polytope in ZZ^2 with 4 vertices
sage: _.vertices()
((0, 0), (0, 2), (2, 0), (2, 2))
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polygon import polar_P1xP1_polytope
>>> polar_P1xP1_polytope()
A 2-dimensional lattice polytope in ZZ^2 with 4 vertices
>>> _.vertices()
((0, 0), (0, 2), (2, 0), (2, 2))
```

sage.geometry.polyhedron.ppl_lattice_polygon.**polar_P2_112_polytope**()

The polar of the $P^2[1, 1, 2]$ polytope.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polygon import polar_P2_112_
↪polytope
sage: polar_P2_112_polytope()
A 2-dimensional lattice polytope in ZZ^2 with 3 vertices
sage: _.vertices()
((0, 0), (0, 2), (4, 0))
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polygon import polar_P2_112_polytope
>>> polar_P2_112_polytope()
A 2-dimensional lattice polytope in ZZ^2 with 3 vertices
>>> _.vertices()
((0, 0), (0, 2), (4, 0))
```

sage.geometry.polyhedron.ppl_lattice_polygon.**polar_P2_polytope**()

The polar of the P^2 polytope.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polygon import polar_P2_polytope
sage: polar_P2_polytope()
A 2-dimensional lattice polytope in ZZ^2 with 3 vertices
sage: _.vertices()
((0, 0), (0, 3), (3, 0))
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polygon import polar_P2_polytope
>>> polar_P2_polytope()
A 2-dimensional lattice polytope in ZZ^2 with 3 vertices
>>> _.vertices()
((0, 0), (0, 3), (3, 0))
```

sage.geometry.polyhedron.ppl_lattice_polygon.**sub_reflexive_polygons**()

Return all lattice sub-polygons of reflexive polygons.

OUTPUT:

A tuple of all lattice sub-polygons. Each sub-polygon is returned as a pair sub-polygon, containing reflexive polygon.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polygon import sub_reflexive_
↪polygons
sage: l = sub_reflexive_polygons(); l[5]
(A 2-dimensional lattice polytope in ZZ^2 with 6 vertices,
 A 2-dimensional lattice polytope in ZZ^2 with 3 vertices)
sage: len(l)
33
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polygon import sub_reflexive_
↪polygons
>>> l = sub_reflexive_polygons(); l[Integer(5)]
(A 2-dimensional lattice polytope in ZZ^2 with 6 vertices,
```

(continues on next page)

(continued from previous page)

```
 A 2-dimensional lattice polytope in ZZ^2 with 3 vertices)
>>> len(l)
33
```

sage.geometry.polyhedron.ppl_lattice_polygon.**subpolygons_of_polar_P1xP1**()

The lattice sub-polygons of the polar $P^1 \times P^1$ polytope.

OUTPUT: a tuple of lattice polytopes

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polygon import subpolygons_of_
↪polar_P1xP1
sage: len(subpolygons_of_polar_P1xP1())
20
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polygon import subpolygons_of_polar_
↪P1xP1
>>> len(subpolygons_of_polar_P1xP1())
20
```

sage.geometry.polyhedron.ppl_lattice_polygon.**subpolygons_of_polar_P2**()

The lattice sub-polygons of the polar P^2 polytope.

OUTPUT: a tuple of lattice polytopes

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polygon import subpolygons_of_
↪polar_P2
sage: len(subpolygons_of_polar_P2())
27
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polygon import subpolygons_of_polar_
↪P2
>>> len(subpolygons_of_polar_P2())
27
```

sage.geometry.polyhedron.ppl_lattice_polygon.**subpolygons_of_polar_P2_112**()

The lattice sub-polygons of the polar $P^2[1,1,2]$ polytope.

OUTPUT: a tuple of lattice polytopes

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polygon import subpolygons_of_
↪polar_P2_112
sage: len(subpolygons_of_polar_P2_112())
28
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polygon import subpolygons_of_polar_
```

(continues on next page)

(continued from previous page)

```
↪P2_112
>>> len(subpolygons_of_polar_P2_112())
28
```

2.2.5 Fast Lattice Polytopes using PPL.

The `LatticePolytope_PPL()` class is a thin wrapper around PPL polyhedra. Its main purpose is to be fast to construct, at the cost of being much less full-featured than the usual polyhedra. This makes it possible to iterate with it over the list of all 473800776 reflexive polytopes in 4 dimensions.

> **Note**
>
> For general lattice polyhedra you should use `Polyhedron()` with `base_ring=ZZ`.

The class derives from the PPL `ppl.polyhedron.C_Polyhedron` class, so you can work with the underlying generator and constraint objects. However, integral points are generally represented by $\mathbf{Z}$-vectors. In the following, we always use *generator* to refer the PPL generator objects and *vertex* (or integral point) for the corresponding $\mathbf{Z}$-vector.

EXAMPLES:

```
sage: vertices = [(1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1), (-9, -6, -1,
↪ -1)]
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_PPL
sage: P = LatticePolytope_PPL(vertices);  P
A 4-dimensional lattice polytope in ZZ^4 with 5 vertices
sage: P.integral_points()
((-9, -6, -1, -1), (-3, -2, 0, 0), (-2, -1, 0, 0), (-1, -1, 0, 0),
 (-1, 0, 0, 0), (0, 0, 0, 0), (1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 0, 1), (0, 0, 1, 0))
sage: P.integral_points_not_interior_to_facets()
((-9, -6, -1, -1), (-3, -2, 0, 0), (0, 0, 0, 0), (1, 0, 0, 0),
 (0, 1, 0, 0), (0, 0, 0, 1), (0, 0, 1, 0))
```

```
>>> from sage.all import *
>>> vertices = [(Integer(1), Integer(0), Integer(0), Integer(0)), (Integer(0),␣
↪Integer(1), Integer(0), Integer(0)), (Integer(0), Integer(0), Integer(1),␣
↪Integer(0)), (Integer(0), Integer(0), Integer(0), Integer(1)), (-Integer(9), -
↪Integer(6), -Integer(1), -Integer(1))]
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_PPL
>>> P = LatticePolytope_PPL(vertices);  P
A 4-dimensional lattice polytope in ZZ^4 with 5 vertices
>>> P.integral_points()
((-9, -6, -1, -1), (-3, -2, 0, 0), (-2, -1, 0, 0), (-1, -1, 0, 0),
 (-1, 0, 0, 0), (0, 0, 0, 0), (1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 0, 1), (0, 0, 1, 0))
>>> P.integral_points_not_interior_to_facets()
((-9, -6, -1, -1), (-3, -2, 0, 0), (0, 0, 0, 0), (1, 0, 0, 0),
 (0, 1, 0, 0), (0, 0, 0, 1), (0, 0, 1, 0))
```

Fibrations of the lattice polytopes are defined as lattice sub-polytopes and give rise to fibrations of toric varieties for suitable fan refinements. We can compute them using `fibration_generator()`

```
sage: F = next(P.fibration_generator(2))
sage: F.vertices()
((1, 0, 0, 0), (0, 1, 0, 0), (-3, -2, 0, 0))
```

```
>>> from sage.all import *
>>> F = next(P.fibration_generator(Integer(2)))
>>> F.vertices()
((1, 0, 0, 0), (0, 1, 0, 0), (-3, -2, 0, 0))
```

Finally, we can compute automorphisms and identify fibrations that only differ by a lattice automorphism:

```
sage: square = LatticePolytope_PPL((-1,-1), (-1,1), (1,-1), (1,1))
sage: fibers = [ f.vertices() for f in square.fibration_generator(1) ];  fibers
[((1, 0), (-1, 0)), ((0, 1), (0, -1)), ((-1, -1), (1, 1)), ((-1, 1), (1, -1))]
sage: square.pointsets_mod_automorphism(fibers)                              #␣
↪needs sage.groups
(frozenset({(-1, -1), (1, 1)}), frozenset({(-1, 0), (1, 0)}))
```

```
>>> from sage.all import *
>>> square = LatticePolytope_PPL((-Integer(1),-Integer(1)), (-Integer(1),Integer(1)),␣
↪(Integer(1),-Integer(1)), (Integer(1),Integer(1)))
>>> fibers = [ f.vertices() for f in square.fibration_generator(Integer(1)) ];  fibers
[((1, 0), (-1, 0)), ((0, 1), (0, -1)), ((-1, -1), (1, 1)), ((-1, 1), (1, -1))]
>>> square.pointsets_mod_automorphism(fibers)                              #␣
↪needs sage.groups
(frozenset({(-1, -1), (1, 1)}), frozenset({(-1, 0), (1, 0)}))
```

AUTHORS:

- Volker Braun: initial version, 2012

`sage.geometry.polyhedron.ppl_lattice_polytope.`**`LatticePolytope_PPL`**`(*args)`

Construct a new instance of the PPL-based lattice polytope class.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
sage: LatticePolytope_PPL((0,0), (1,0), (0,1))
A 2-dimensional lattice polytope in ZZ^2 with 3 vertices

sage: from ppl import point, Generator_System, C_Polyhedron, Linear_Expression  #␣
↪needs pplpy
sage: p = point(Linear_Expression([2,3],0));  p                               #␣
↪needs pplpy
point(2/1, 3/1)
sage: LatticePolytope_PPL(p)                                                  #␣
↪needs pplpy
A 0-dimensional lattice polytope in ZZ^2 with 1 vertex

sage: P = C_Polyhedron(Generator_System(p));  P                               #␣
↪needs pplpy
A 0-dimensional polyhedron in QQ^2 defined as the convex hull of 1 point
sage: LatticePolytope_PPL(P)                                                  #␣
```

(continues on next page)

(continued from previous page)

```
↪needs pplpy
A 0-dimensional lattice polytope in ZZ^2 with 1 vertex
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_PPL
>>> LatticePolytope_PPL((Integer(0),Integer(0)), (Integer(1),Integer(0)),
↪(Integer(0),Integer(1)))
A 2-dimensional lattice polytope in ZZ^2 with 3 vertices

>>> from ppl import point, Generator_System, C_Polyhedron, Linear_Expression  #
↪needs pplpy
>>> p = point(Linear_Expression([Integer(2),Integer(3)],Integer(0))); p
↪                        # needs pplpy
point(2/1, 3/1)
>>> LatticePolytope_PPL(p)                                                     #
↪needs pplpy
A 0-dimensional lattice polytope in ZZ^2 with 1 vertex

>>> P = C_Polyhedron(Generator_System(p)); P                                   #
↪needs pplpy
A 0-dimensional polyhedron in QQ^2 defined as the convex hull of 1 point
>>> LatticePolytope_PPL(P)                                                     #
↪needs pplpy
A 0-dimensional lattice polytope in ZZ^2 with 1 vertex
```

A `TypeError` is raised if the arguments do not specify a lattice polytope:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
sage: LatticePolytope_PPL((0,0), (1/2,1))                                      #
↪needs pplpy
Traceback (most recent call last):
...
TypeError: unable to convert rational 1/2 to an integer

sage: from ppl import point, Generator_System, C_Polyhedron, Linear_Expression  #
↪needs pplpy
sage: p = point(Linear_Expression([2,3],0), 5); p                              #
↪needs pplpy
point(2/5, 3/5)
sage: LatticePolytope_PPL(p)                                                   #
↪needs pplpy
Traceback (most recent call last):
 ...
TypeError: generator is not a lattice polytope generator

sage: P = C_Polyhedron(Generator_System(p)); P                                 #
↪needs pplpy
A 0-dimensional polyhedron in QQ^2 defined as the convex hull of 1 point
sage: LatticePolytope_PPL(P)                                                   #
↪needs pplpy
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
TypeError: polyhedron has non-integral generators
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_PPL
>>> LatticePolytope_PPL((Integer(0),Integer(0)), (Integer(1)/Integer(2),
↪Integer(1)))                                         # needs pplpy
Traceback (most recent call last):
...
TypeError: unable to convert rational 1/2 to an integer

>>> from ppl import point, Generator_System, C_Polyhedron, Linear_Expression  #␣
↪needs pplpy
>>> p = point(Linear_Expression([Integer(2),Integer(3)],Integer(0)), Integer(5));␣
↪ p                                  # needs pplpy
point(2/5, 3/5)
>>> LatticePolytope_PPL(p)                                                    #␣
↪needs pplpy
Traceback (most recent call last):
 ...
TypeError: generator is not a lattice polytope generator

>>> P = C_Polyhedron(Generator_System(p));  P                                 #␣
↪needs pplpy
A 0-dimensional polyhedron in QQ^2 defined as the convex hull of 1 point
>>> LatticePolytope_PPL(P)                                                    #␣
↪needs pplpy
Traceback (most recent call last):
...
TypeError: polyhedron has non-integral generators
```

class sage.geometry.polyhedron.ppl_lattice_polytope.**LatticePolytope_PPL_class**

Bases: `C_Polyhedron`

The lattice polytope class.

You should use *LatticePolytope_PPL()* to construct instances.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
sage: LatticePolytope_PPL((0,0), (1,0), (0,1))
A 2-dimensional lattice polytope in ZZ^2 with 3 vertices
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_PPL
>>> LatticePolytope_PPL((Integer(0),Integer(0)), (Integer(1),Integer(0)),␣
↪(Integer(0),Integer(1)))
A 2-dimensional lattice polytope in ZZ^2 with 3 vertices
```

affine_lattice_polytope()

Return the lattice polytope restricted to *affine_space()*.

OUTPUT:

A new, full-dimensional lattice polytope.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import␣
↪LatticePolytope_PPL
sage: poly_4d = LatticePolytope_PPL((-9,-6,0,0), (0,1,0,0), (1,0,0,0));  poly_
↪4d
A 2-dimensional lattice polytope in ZZ^4 with 3 vertices
sage: poly_4d.space_dimension()
4
sage: poly_2d = poly_4d.affine_lattice_polytope();  poly_2d
A 2-dimensional lattice polytope in ZZ^2 with 3 vertices
sage: poly_2d.space_dimension()
2
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> poly_4d = LatticePolytope_PPL((-Integer(9),-Integer(6),Integer(0),
↪Integer(0)), (Integer(0),Integer(1),Integer(0),Integer(0)), (Integer(1),
↪Integer(0),Integer(0),Integer(0)));  poly_4d
A 2-dimensional lattice polytope in ZZ^4 with 3 vertices
>>> poly_4d.space_dimension()
4
>>> poly_2d = poly_4d.affine_lattice_polytope();  poly_2d
A 2-dimensional lattice polytope in ZZ^2 with 3 vertices
>>> poly_2d.space_dimension()
2
```

affine_space()

Return the affine space spanned by the polytope.

OUTPUT:

The free module $\mathbf{Z}^n$, where n is the dimension of the affine space spanned by the points of the polytope.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import␣
↪LatticePolytope_PPL
sage: point = LatticePolytope_PPL((1,2,3))
sage: point.affine_space()
Free module of degree 3 and rank 0 over Integer Ring
Echelon basis matrix:
[]
sage: line = LatticePolytope_PPL((1,1,1), (1,2,3))
sage: line.affine_space()
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[0 1 2]
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> point = LatticePolytope_PPL((Integer(1),Integer(2),Integer(3)))
>>> point.affine_space()
Free module of degree 3 and rank 0 over Integer Ring
Echelon basis matrix:
[]
>>> line = LatticePolytope_PPL((Integer(1),Integer(1),Integer(1)),
↪(Integer(1),Integer(2),Integer(3)))
>>> line.affine_space()
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[0 1 2]
```

ambient_space()

Return the ambient space.

OUTPUT:

The free module $\mathbf{Z}^d$, where d is the ambient space dimension.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import
↪LatticePolytope_PPL
sage: point = LatticePolytope_PPL((1,2,3))
sage: point.ambient_space()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> point = LatticePolytope_PPL((Integer(1),Integer(2),Integer(3)))
>>> point.ambient_space()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

base_projection(*fiber*)

The projection that maps the sub-polytope `fiber` to a single point.

OUTPUT:

The quotient module of the ambient space modulo the *affine_space()* spanned by the fiber.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import
↪LatticePolytope_PPL
sage: poly = LatticePolytope_PPL((-9,-6,-1,-1),
....:                            (0,0,0,1), (0,0,1,0), (0,1,0,0), (1,0,0,0))
sage: fiber = next(poly.fibration_generator(2))
sage: poly.base_projection(fiber)
Finitely generated module V/W over Integer Ring with invariants (0, 0)
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> poly = LatticePolytope_PPL((-Integer(9),-Integer(6),-Integer(1),-
↪Integer(1)),
...                            (Integer(0),Integer(0),Integer(0),Integer(1)),␣
↪(Integer(0),Integer(0),Integer(1),Integer(0)), (Integer(0),Integer(1),
↪Integer(0),Integer(0)), (Integer(1),Integer(0),Integer(0),Integer(0)))
>>> fiber = next(poly.fibration_generator(Integer(2)))
>>> poly.base_projection(fiber)
Finitely generated module V/W over Integer Ring with invariants (0, 0)
```

base_projection_matrix(*fiber*)

The projection that maps the sub-polytope `fiber` to a single point.

OUTPUT:

An integer matrix that represents the projection to the base.

See also

The *base_projection()* yields equivalent information, and is easier to use. However, just returning the matrix has lower overhead.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import␣
↪LatticePolytope_PPL
sage: poly = LatticePolytope_PPL((-9,-6,-1,-1),
....:                            (0,0,0,1), (0,0,1,0), (0,1,0,0), (1,0,0,0))
sage: fiber = next(poly.fibration_generator(2))
sage: poly.base_projection_matrix(fiber)
[ 0  0 -1  0]
[ 0  0  0 -1]
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> poly = LatticePolytope_PPL((-Integer(9),-Integer(6),-Integer(1),-
↪Integer(1)),
...                            (Integer(0),Integer(0),Integer(0),Integer(1)),␣
↪(Integer(0),Integer(0),Integer(1),Integer(0)), (Integer(0),Integer(1),
↪Integer(0),Integer(0)), (Integer(1),Integer(0),Integer(0),Integer(0)))
>>> fiber = next(poly.fibration_generator(Integer(2)))
>>> poly.base_projection_matrix(fiber)
[ 0  0 -1  0]
[ 0  0  0 -1]
```

Note that the basis choice in *base_projection()* for the quotient is usually different:

```
sage: proj = poly.base_projection(fiber)
sage: proj_matrix = poly.base_projection_matrix(fiber)
sage: [proj(p) for p in poly.integral_points()]
```

(continues on next page)

(continued from previous page)

```
[(-1, -1), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 1), (1,
↪ 0)]
sage: [proj_matrix*p for p in poly.integral_points()]
[(1, 1), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, -1), (-1,
↪ 0)]
```

```
>>> from sage.all import *
>>> proj = poly.base_projection(fiber)
>>> proj_matrix = poly.base_projection_matrix(fiber)
>>> [proj(p) for p in poly.integral_points()]
[(-1, -1), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 1), (1,
↪ 0)]
>>> [proj_matrix*p for p in poly.integral_points()]
[(1, 1), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, -1), (-1,
↪ 0)]
```

base_rays(*fiber*, *points*)

Return the primitive lattice vectors that generate the direction given by the base projection of points.

INPUT:

- `fiber` – a sub-lattice polytope defining the *base_projection()*
- `points` – the points to project to the base

OUTPUT: a tuple of primitive $\mathbf{Z}$-vectors

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import␣
↪LatticePolytope_PPL
sage: poly = LatticePolytope_PPL((-9,-6,-1,-1),
....:                            (0,0,0,1), (0,0,1,0), (0,1,0,0), (1,0,0,0))
sage: fiber = next(poly.fibration_generator(2))
sage: poly.base_rays(fiber, poly.integral_points_not_interior_to_facets())
((-1, -1), (0, 1), (1, 0))

sage: p = LatticePolytope_PPL((1,0), (1,2), (-1,0))
sage: f = LatticePolytope_PPL((1,0), (-1,0))
sage: p.base_rays(f, p.integral_points())
((1),)
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> poly = LatticePolytope_PPL((-Integer(9),-Integer(6),-Integer(1),-
↪Integer(1)),
...                            (Integer(0),Integer(0),Integer(0),Integer(1)),␣
↪(Integer(0),Integer(0),Integer(1),Integer(0)), (Integer(0),Integer(1),
↪Integer(0),Integer(0)), (Integer(1),Integer(0),Integer(0),Integer(0)))
>>> fiber = next(poly.fibration_generator(Integer(2)))
>>> poly.base_rays(fiber, poly.integral_points_not_interior_to_facets())
((-1, -1), (0, 1), (1, 0))
```

(continues on next page)

(continued from previous page)

```
>>> p = LatticePolytope_PPL((Integer(1),Integer(0)), (Integer(1),Integer(2)),␣
↪(-Integer(1),Integer(0)))
>>> f = LatticePolytope_PPL((Integer(1),Integer(0)), (-Integer(1),Integer(0)))
>>> p.base_rays(f, p.integral_points())
((1),)
```

bounding_box()

Return the coordinates of a rectangular box containing the non-empty polytope.

OUTPUT:

A pair of tuples `(box_min, box_max)` where `box_min` are the coordinates of a point bounding the coordinates of the polytope from below and `box_max` bounds the coordinates from above.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import␣
↪LatticePolytope_PPL
sage: LatticePolytope_PPL((0,0), (1,0), (0,1)).bounding_box()
((0, 0), (1, 1))
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> LatticePolytope_PPL((Integer(0),Integer(0)), (Integer(1),Integer(0)),␣
↪(Integer(0),Integer(1))).bounding_box()
((0, 0), (1, 1))
```

contains(*point_coordinates*)

Test whether point is contained in the polytope.

INPUT:

- `point_coordinates` – list/tuple/iterable of rational numbers; the coordinates of the point

OUTPUT: boolean

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import␣
↪LatticePolytope_PPL
sage: line = LatticePolytope_PPL((1,2,3), (-1,-2,-3))
sage: line.contains([0,0,0])
True
sage: line.contains([1,0,0])
False
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> line = LatticePolytope_PPL((Integer(1),Integer(2),Integer(3)), (-
↪Integer(1),-Integer(2),-Integer(3)))
>>> line.contains([Integer(0),Integer(0),Integer(0)])
True
```

(continues on next page)

(continued from previous page)

```
>>> line.contains([Integer(1),Integer(0),Integer(0)])
False
```

contains_origin()

Test whether the polytope contains the origin.

OUTPUT: boolean

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import␣
↪LatticePolytope_PPL
sage: LatticePolytope_PPL((1,2,3), (-1,-2,-3)).contains_origin()
True
sage: LatticePolytope_PPL((1,2,5), (-1,-2,-3)).contains_origin()
False
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> LatticePolytope_PPL((Integer(1),Integer(2),Integer(3)), (-Integer(1),-
↪Integer(2),-Integer(3))).contains_origin()
True
>>> LatticePolytope_PPL((Integer(1),Integer(2),Integer(5)), (-Integer(1),-
↪Integer(2),-Integer(3))).contains_origin()
False
```

embed_in_reflexive_polytope(*output='hom'*)

Find an embedding as a sub-polytope of a maximal reflexive polytope.

INPUT:

- `hom` – string. One of `'hom'` (default), `'polytope'`, or `points`. How the embedding is returned. See the output section for details.

OUTPUT:

An embedding into a reflexive polytope. Depending on the `output` option slightly different data is returned.

- If `output='hom'`, a map from a reflexive polytope onto `self` is returned.
- If `output='polytope'`, a reflexive polytope that contains `self` (up to a lattice linear transformation) is returned. That is, the domain of the `output='hom'` map is returned. If the affine span of `self` is less or equal 2-dimensional, the output is one of the following three possibilities:

  `polar_P2_polytope()`, `polar_P1xP1_polytope()`, or `polar_P2_112_polytope()`.
- If `output='points'`, a dictionary containing the integral points of `self` as keys and the corresponding integral point of the reflexive polytope as value.

If there is no such embedding, a `LatticePolytopeNoEmbeddingError` is raised. Even if it exists, the ambient reflexive polytope is usually not uniquely determined and a random but fixed choice will be returned.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import␣
↪LatticePolytope_PPL
sage: polygon = LatticePolytope_PPL((0,0,2,1), (0,1,2,0), (2,3,0,0), (2,0,0,
```

(continues on next page)

(continued from previous page)

```
↪3))
sage: polygon.embed_in_reflexive_polytope()
The map A*x+b with
  A=
    [ 1  1]
    [ 0  1]
    [-1 -1]
    [ 1  0]
  b = (-1, 0, 3, 0)
sage: polygon.embed_in_reflexive_polytope('polytope')
A 2-dimensional lattice polytope in ZZ^2 with 3 vertices
sage: polygon.embed_in_reflexive_polytope('points')
{(0, 0, 2, 1): (1, 0),
 (0, 1, 2, 0): (0, 1),
 (1, 0, 1, 2): (2, 0),
 (1, 1, 1, 1): (1, 1),
 (1, 2, 1, 0): (0, 2),
 (2, 0, 0, 3): (3, 0),
 (2, 1, 0, 2): (2, 1),
 (2, 2, 0, 1): (1, 2),
 (2, 3, 0, 0): (0, 3)}

sage: LatticePolytope_PPL((0,0), (4,0), (0,4)).embed_in_reflexive_polytope()
Traceback (most recent call last):
...
LatticePolytopeNoEmbeddingError: not a sub-polytope of a reflexive polygon
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> polygon = LatticePolytope_PPL((Integer(0),Integer(0),Integer(2),
↪Integer(1)), (Integer(0),Integer(1),Integer(2),Integer(0)), (Integer(2),
↪Integer(3),Integer(0),Integer(0)), (Integer(2),Integer(0),Integer(0),
↪Integer(3)))
>>> polygon.embed_in_reflexive_polytope()
The map A*x+b with
  A=
    [ 1  1]
    [ 0  1]
    [-1 -1]
    [ 1  0]
  b = (-1, 0, 3, 0)
>>> polygon.embed_in_reflexive_polytope('polytope')
A 2-dimensional lattice polytope in ZZ^2 with 3 vertices
>>> polygon.embed_in_reflexive_polytope('points')
{(0, 0, 2, 1): (1, 0),
 (0, 1, 2, 0): (0, 1),
 (1, 0, 1, 2): (2, 0),
 (1, 1, 1, 1): (1, 1),
 (1, 2, 1, 0): (0, 2),
 (2, 0, 0, 3): (3, 0),
 (2, 1, 0, 2): (2, 1),
```

(continues on next page)

(continued from previous page)

```
 (2, 2, 0, 1): (1, 2),
 (2, 3, 0, 0): (0, 3)}

>>> LatticePolytope_PPL((Integer(0),Integer(0)), (Integer(4),Integer(0)),
→(Integer(0),Integer(4))).embed_in_reflexive_polytope()
Traceback (most recent call last):
...
LatticePolytopeNoEmbeddingError: not a sub-polytope of a reflexive polygon
```

fibration_generator(*dim*)

Generate the lattice polytope fibrations.

For the purposes of this function, a lattice polytope fiber is a sub-lattice polytope. Projecting the plane spanned by the subpolytope to a point yields another lattice polytope, the base of the fibration.

INPUT:

- `dim` – integer; the dimension of the lattice polytope fiber

OUTPUT:

A generator yielding the distinct lattice polytope fibers of given dimension.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import
→LatticePolytope_PPL
sage: p = LatticePolytope_PPL((-9,-6,-1,-1),
....:                         (0,0,0,1), (0,0,1,0), (0,1,0,0), (1,0,0,0))
sage: list(p.fibration_generator(2))
[A 2-dimensional lattice polytope in ZZ^4 with 3 vertices]
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
→PPL
>>> p = LatticePolytope_PPL((-Integer(9),-Integer(6),-Integer(1),-Integer(1)),
...                         (Integer(0),Integer(0),Integer(0),Integer(1)),
→(Integer(0),Integer(0),Integer(1),Integer(0)), (Integer(0),Integer(1),
→Integer(0),Integer(0)), (Integer(1),Integer(0),Integer(0),Integer(0)))
>>> list(p.fibration_generator(Integer(2)))
[A 2-dimensional lattice polytope in ZZ^4 with 3 vertices]
```

has_IP_property()

Whether the lattice polytope has the IP property.

That is, the polytope is full-dimensional and the origin is a interior point not on the boundary.

OUTPUT: boolean

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import
→LatticePolytope_PPL
sage: LatticePolytope_PPL((-1,-1), (0,1), (1,0)).has_IP_property()
True
sage: LatticePolytope_PPL((-1,-1), (1,1)).has_IP_property()
False
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> LatticePolytope_PPL((-Integer(1),-Integer(1)), (Integer(0),Integer(1)),␣
↪(Integer(1),Integer(0))).has_IP_property()
True
>>> LatticePolytope_PPL((-Integer(1),-Integer(1)), (Integer(1),Integer(1))).
↪has_IP_property()
False
```

integral_points()

Return the integral points in the polyhedron.

Uses the naive algorithm (iterate over a rectangular bounding box).

OUTPUT:

The list of integral points in the polyhedron. If the polyhedron is not compact, a `ValueError` is raised.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import␣
↪LatticePolytope_PPL
sage: LatticePolytope_PPL((-1,-1), (1,0), (1,1), (0,1)).integral_points()
((-1, -1), (0, 0), (0, 1), (1, 0), (1, 1))

sage: simplex = LatticePolytope_PPL((1,2,3), (2,3,7), (-2,-3,-11))
sage: simplex.integral_points()
((-2, -3, -11), (0, 0, -2), (1, 2, 3), (2, 3, 7))
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> LatticePolytope_PPL((-Integer(1),-Integer(1)), (Integer(1),Integer(0)),␣
↪(Integer(1),Integer(1)), (Integer(0),Integer(1))).integral_points()
((-1, -1), (0, 0), (0, 1), (1, 0), (1, 1))

>>> simplex = LatticePolytope_PPL((Integer(1),Integer(2),Integer(3)),␣
↪(Integer(2),Integer(3),Integer(7)), (-Integer(2),-Integer(3),-Integer(11)))
>>> simplex.integral_points()
((-2, -3, -11), (0, 0, -2), (1, 2, 3), (2, 3, 7))
```

The polyhedron need not be full-dimensional:

```
sage: simplex = LatticePolytope_PPL((1,2,3,5), (2,3,7,5), (-2,-3,-11,5))
sage: simplex.integral_points()
((-2, -3, -11, 5), (0, 0, -2, 5), (1, 2, 3, 5), (2, 3, 7, 5))

sage: point = LatticePolytope_PPL((2,3,7))
sage: point.integral_points()
((2, 3, 7),)

sage: empty = LatticePolytope_PPL()
sage: empty.integral_points()
()
```

```
>>> from sage.all import *
>>> simplex = LatticePolytope_PPL((Integer(1),Integer(2),Integer(3),
↪Integer(5)), (Integer(2),Integer(3),Integer(7),Integer(5)), (-Integer(2),-
↪Integer(3),-Integer(11),Integer(5)))
>>> simplex.integral_points()
((-2, -3, -11, 5), (0, 0, -2, 5), (1, 2, 3, 5), (2, 3, 7, 5))

>>> point = LatticePolytope_PPL((Integer(2),Integer(3),Integer(7)))
>>> point.integral_points()
((2, 3, 7),)

>>> empty = LatticePolytope_PPL()
>>> empty.integral_points()
()
```

Here is a simplex where the naive algorithm of running over all points in a rectangular bounding box no longer works fast enough:

```
sage: v = [(1,0,7,-1), (-2,-2,4,-3), (-1,-1,-1,4), (2,9,0,-5), (-2,-1,5,1)]
sage: simplex = LatticePolytope_PPL(v); simplex
A 4-dimensional lattice polytope in ZZ^4 with 5 vertices
sage: len(simplex.integral_points())
49
```

```
>>> from sage.all import *
>>> v = [(Integer(1),Integer(0),Integer(7),-Integer(1)), (-Integer(2),-
↪Integer(2),Integer(4),-Integer(3)), (-Integer(1),-Integer(1),-Integer(1),
↪Integer(4)), (Integer(2),Integer(9),Integer(0),-Integer(5)), (-Integer(2),-
↪Integer(1),Integer(5),Integer(1))]
>>> simplex = LatticePolytope_PPL(v); simplex
A 4-dimensional lattice polytope in ZZ^4 with 5 vertices
>>> len(simplex.integral_points())
49
```

Finally, the 3-d reflexive polytope number 4078:

```
sage: v = [(1,0,0), (0,1,0), (0,0,1), (0,0,-1), (0,-2,1),
....:      (-1,2,-1), (-1,2,-2), (-1,1,-2), (-1,-1,2), (-1,-3,2)]
sage: P = LatticePolytope_PPL(*v)
sage: pts1 = P.integral_points()            # Sage's own code
sage: pts2 = LatticePolytope(v).points()                                  #␣
↪needs palp
sage: for p in pts1: p.set_immutable()
sage: set(pts1) == set(pts2)                                              #␣
↪needs palp
True

sage: len(Polyhedron(v).integral_points())  # takes about 1 ms
23
sage: len(LatticePolytope(v).points())      # takes about 13 ms           #␣
↪needs palp
23
sage: len(LatticePolytope_PPL(*v).integral_points())  # takes about 0.5 ms
```

(continues on next page)

(continued from previous page)

```
23
```

```
>>> from sage.all import *
>>> v = [(Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0)), (Integer(0),Integer(0),Integer(1)), (Integer(0),Integer(0),-
↪Integer(1)), (Integer(0),-Integer(2),Integer(1)),
...      (-Integer(1),Integer(2),-Integer(1)), (-Integer(1),Integer(2),-
↪Integer(2)), (-Integer(1),Integer(1),-Integer(2)), (-Integer(1),-Integer(1),
↪Integer(2)), (-Integer(1),-Integer(3),Integer(2))]
>>> P = LatticePolytope_PPL(*v)
>>> pts1 = P.integral_points()          # Sage's own code
>>> pts2 = LatticePolytope(v).points()                              #␣
↪needs palp
>>> for p in pts1: p.set_immutable()
>>> set(pts1) == set(pts2)                                          #␣
↪needs palp
True

>>> len(Polyhedron(v).integral_points())  # takes about 1 ms
23
>>> len(LatticePolytope(v).points())      # takes about 13 ms           #␣
↪needs palp
23
>>> len(LatticePolytope_PPL(*v).integral_points())  # takes about 0.5 ms
23
```

integral_points_not_interior_to_facets()

Return the integral points not interior to facets.

OUTPUT:

A tuple whose entries are the coordinate vectors of integral points not interior to facets (codimension one faces) of the lattice polytope.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import␣
↪LatticePolytope_PPL
sage: square = LatticePolytope_PPL((-1,-1), (-1,1), (1,-1), (1,1))
sage: square.n_integral_points()
9
sage: square.integral_points_not_interior_to_facets()
((-1, -1), (-1, 1), (0, 0), (1, -1), (1, 1))
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> square = LatticePolytope_PPL((-Integer(1),-Integer(1)), (-Integer(1),
↪Integer(1)), (Integer(1),-Integer(1)), (Integer(1),Integer(1)))
>>> square.n_integral_points()
9
>>> square.integral_points_not_interior_to_facets()
((-1, -1), (-1, 1), (0, 0), (1, -1), (1, 1))
```

is_bounded()

Return whether the lattice polytope is compact.

OUTPUT: always `True`, since polytopes are by definition compact

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import␣
↪LatticePolytope_PPL
sage: LatticePolytope_PPL((0,0), (1,0), (0,1)).is_bounded()
True
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> LatticePolytope_PPL((Integer(0),Integer(0)), (Integer(1),Integer(0)),␣
↪(Integer(0),Integer(1))).is_bounded()
True
```

is_full_dimensional()

Return whether the lattice polytope is full dimensional.

OUTPUT: boolean; whether the `affine_dimension()` equals the ambient space dimension

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import␣
↪LatticePolytope_PPL
sage: p = LatticePolytope_PPL((0,0), (0,1))
sage: p.is_full_dimensional()
False
sage: q = LatticePolytope_PPL((0,0), (0,1), (1,0))
sage: q.is_full_dimensional()
True
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> p = LatticePolytope_PPL((Integer(0),Integer(0)), (Integer(0),Integer(1)))
>>> p.is_full_dimensional()
False
>>> q = LatticePolytope_PPL((Integer(0),Integer(0)), (Integer(0),Integer(1)),␣
↪(Integer(1),Integer(0)))
>>> q.is_full_dimensional()
True
```

is_simplex()

Return whether the polyhedron is a simplex.

OUTPUT:

Boolean, whether the polyhedron is a simplex (possibly of strictly smaller dimension than the ambient space).

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import␣
↪LatticePolytope_PPL
sage: LatticePolytope_PPL((0,0,0), (1,0,0), (0,1,0)).is_simplex()
True
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> LatticePolytope_PPL((Integer(0),Integer(0),Integer(0)), (Integer(1),
↪Integer(0),Integer(0)), (Integer(0),Integer(1),Integer(0))).is_simplex()
True
```

lattice_automorphism_group(*points=None*, *point_labels=None*)

The integral subgroup of the restricted automorphism group.

INPUT:

- `points` – tuple of coordinate vectors or `None` (default). If specified, the points must form complete orbits under the lattice automorphism group. If `None` all vertices are used.
- `point_labels` – tuple of labels for the `points` or `None` (default). These will be used as labels for the do permutation group. If `None`, the `points` will be used themselves.

OUTPUT:

The integral subgroup of the restricted automorphism group acting on the given `points`, or all vertices if not specified.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import␣
↪LatticePolytope_PPL
sage: Z3square = LatticePolytope_PPL((0,0), (1,2), (2,1), (3,3))
sage: Z3square.lattice_automorphism_group()                              #␣
↪needs sage.graphs sage.groups
Permutation Group with generators [(), ((1,2),(2,1)),
((0,0),(3,3)), ((0,0),(3,3))((1,2),(2,1))]

sage: G1 = Z3square.lattice_automorphism_group(point_labels=(1,2,3,4))    #␣
↪needs sage.graphs sage.groups
sage: G1                                                                  #␣
↪needs sage.graphs sage.groups
Permutation Group with generators [(), (2,3), (1,4), (1,4)(2,3)]
sage: G1.cardinality()                                                    #␣
↪needs sage.graphs sage.groups
4

sage: G2 = Z3square.restricted_automorphism_group(vertex_labels=(1,2,3,4))  #␣
↪needs sage.graphs sage.groups
sage: G2 == PermutationGroup([[(2,3)], [(1,2),(3,4)], [(1,4)]])           #␣
↪needs sage.graphs sage.groups
True
sage: G2.cardinality()                                                    #␣
↪needs sage.graphs sage.groups
8
```

(continues on next page)

(continued from previous page)

```
sage: points = Z3square.integral_points(); points
((0, 0), (1, 1), (1, 2), (2, 1), (2, 2), (3, 3))
sage: Z3square.lattice_automorphism_group(points,                          #␣
↪needs sage.graphs sage.groups
....:                                    point_labels=(1,2,3,4,5,6))
Permutation Group with generators [(), (3,4), (1,6)(2,5), (1,6)(2,5)(3,4)]
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> Z3square = LatticePolytope_PPL((Integer(0),Integer(0)), (Integer(1),
↪Integer(2)), (Integer(2),Integer(1)), (Integer(3),Integer(3)))
>>> Z3square.lattice_automorphism_group()                                 #␣
↪needs sage.graphs sage.groups
Permutation Group with generators [(), ((1,2),(2,1)),
((0,0),(3,3)), ((0,0),(3,3))((1,2),(2,1))]

>>> G1 = Z3square.lattice_automorphism_group(point_labels=(Integer(1),
↪Integer(2),Integer(3),Integer(4)))      # needs sage.graphs sage.groups
>>> G1                                                                    #␣
↪needs sage.graphs sage.groups
Permutation Group with generators [(), (2,3), (1,4), (1,4)(2,3)]
>>> G1.cardinality()                                                      #␣
↪needs sage.graphs sage.groups
4

>>> G2 = Z3square.restricted_automorphism_group(vertex_labels=(Integer(1),
↪Integer(2),Integer(3),Integer(4)))   # needs sage.graphs sage.groups
>>> G2 == PermutationGroup([[(Integer(2),Integer(3))], [(Integer(1),
↪Integer(2)),(Integer(3),Integer(4))], [(Integer(1),Integer(4))]])          ␣
↪   # needs sage.graphs sage.groups
True
>>> G2.cardinality()                                                      #␣
↪needs sage.graphs sage.groups
8

>>> points = Z3square.integral_points(); points
((0, 0), (1, 1), (1, 2), (2, 1), (2, 2), (3, 3))
>>> Z3square.lattice_automorphism_group(points,                           #␣
↪needs sage.graphs sage.groups
...                                    point_labels=(Integer(1),Integer(2),
↪Integer(3),Integer(4),Integer(5),Integer(6)))
Permutation Group with generators [(), (3,4), (1,6)(2,5), (1,6)(2,5)(3,4)]
```

Point labels also work for lattice polytopes that are not full-dimensional, see Issue #16669:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import␣
↪LatticePolytope_PPL
sage: lp = LatticePolytope_PPL((1,0,0), (0,1,0), (-1,-1,0))
sage: lp.lattice_automorphism_group(point_labels=(0,1,2))                  #␣
↪needs sage.graphs sage.groups
```

(continues on next page)

(continued from previous page)

```
Permutation Group with generators [(), (1,2), (0,1), (0,1,2), (0,2,1), (0,2)]
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> lp = LatticePolytope_PPL((Integer(1),Integer(0),Integer(0)), (Integer(0),
↪Integer(1),Integer(0)), (-Integer(1),-Integer(1),Integer(0)))
>>> lp.lattice_automorphism_group(point_labels=(Integer(0),Integer(1),
↪Integer(2)))                    # needs sage.graphs sage.groups
Permutation Group with generators [(), (1,2), (0,1), (0,1,2), (0,2,1), (0,2)]
```

n_integral_points()

Return the number of integral points.

OUTPUT:

Integer. The number of integral points contained in the lattice polytope.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import␣
↪LatticePolytope_PPL
sage: LatticePolytope_PPL((0,0), (1,0), (0,1)).n_integral_points()
3
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> LatticePolytope_PPL((Integer(0),Integer(0)), (Integer(1),Integer(0)),␣
↪(Integer(0),Integer(1))).n_integral_points()
3
```

n_vertices()

Return the number of vertices.

OUTPUT: integer; the number of vertices

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import␣
↪LatticePolytope_PPL
sage: LatticePolytope_PPL((0,0,0), (1,0,0), (0,1,0)).n_vertices()
3
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> LatticePolytope_PPL((Integer(0),Integer(0),Integer(0)), (Integer(1),
↪Integer(0),Integer(0)), (Integer(0),Integer(1),Integer(0))).n_vertices()
3
```

pointsets_mod_automorphism(*pointsets*)

Return `pointsets` modulo the automorphisms of `self`.

INPUT:

- `polytopes` – tuple/list/iterable of subsets of the integral points of `self`

OUTPUT:

Representatives of the point sets modulo the `lattice_automorphism_group()` of `self`.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import␣
↪LatticePolytope_PPL
sage: square = LatticePolytope_PPL((-1,-1), (-1,1), (1,-1), (1,1))
sage: fibers = [f.vertices() for f in square.fibration_generator(1)]
sage: square.pointsets_mod_automorphism(fibers)                              #␣
↪needs sage.graphs sage.groups
(frozenset({(-1, -1), (1, 1)}), frozenset({(-1, 0), (1, 0)}))

sage: cell24 = LatticePolytope_PPL(
....:     (1,0,0,0), (0,1,0,0), (0,0,1,0), (0,0,0,1), (1,-1,-1,1), (0,0,-1,1),
....:     (0,-1,0,1), (-1,0,0,1), (1,0,0,-1), (0,1,0,-1), (0,0,1,-1), (-1,1,1,
↪-1),
....:     (1,-1,-1,0), (0,0,-1,0), (0,-1,0,0), (-1,0,0,0), (1,-1,0,0), (1,0,-
↪1,0),
....:     (0,1,1,-1), (-1,1,1,0), (-1,1,0,0), (-1,0,1,0), (0,-1,-1,1), (0,0,0,
↪-1))
sage: fibers = [f.vertices() for f in cell24.fibration_generator(2)]
sage: cell24.pointsets_mod_automorphism(fibers)     # long time              #␣
↪needs sage.graphs sage.groups
(frozenset({(-1, 0, 0, 0),
            (-1, 0, 0, 1),
            (0, 0, 0, -1),
            (0, 0, 0, 1),
            (1, 0, 0, -1),
            (1, 0, 0, 0)}),
 frozenset({(-1, 0, 0, 0), (-1, 1, 1, 0), (1, -1, -1, 0), (1, 0, 0, 0)}))
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> square = LatticePolytope_PPL((-Integer(1),-Integer(1)), (-Integer(1),
↪Integer(1)), (Integer(1),-Integer(1)), (Integer(1),Integer(1)))
>>> fibers = [f.vertices() for f in square.fibration_generator(Integer(1))]
>>> square.pointsets_mod_automorphism(fibers)                              #␣
↪needs sage.graphs sage.groups
(frozenset({(-1, -1), (1, 1)}), frozenset({(-1, 0), (1, 0)}))

>>> cell24 = LatticePolytope_PPL(
...     (Integer(1),Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0),Integer(0)), (Integer(0),Integer(0),Integer(1),Integer(0)),␣
↪(Integer(0),Integer(0),Integer(0),Integer(1)), (Integer(1),-Integer(1),-
↪Integer(1),Integer(1)), (Integer(0),Integer(0),-Integer(1),Integer(1)),
...     (Integer(0),-Integer(1),Integer(0),Integer(1)), (-Integer(1),
↪Integer(0),Integer(0),Integer(1)), (Integer(1),Integer(0),Integer(0),-
↪Integer(1)), (Integer(0),Integer(1),Integer(0),-Integer(1)), (Integer(0),
↪Integer(0),Integer(1),-Integer(1)), (-Integer(1),Integer(1),Integer(1),-
↪Integer(1)),
```

(continues on next page)

(continued from previous page)

```
...     (Integer(1),-Integer(1),-Integer(1),Integer(0)), (Integer(0),
↪Integer(0),-Integer(1),Integer(0)), (Integer(0),-Integer(1),Integer(0),
↪Integer(0)), (-Integer(1),Integer(0),Integer(0),Integer(0)), (Integer(1),-
↪Integer(1),Integer(0),Integer(0)), (Integer(1),Integer(0),-Integer(1),
↪Integer(0)),
...     (Integer(0),Integer(1),Integer(1),-Integer(1)), (-Integer(1),
↪Integer(1),Integer(1),Integer(0)), (-Integer(1),Integer(1),Integer(0),
↪Integer(0)), (-Integer(1),Integer(0),Integer(1),Integer(0)), (Integer(0),-
↪Integer(1),-Integer(1),Integer(1)), (Integer(0),Integer(0),Integer(0),-
↪Integer(1)))
>>> fibers = [f.vertices() for f in cell24.fibration_generator(Integer(2))]
>>> cell24.pointsets_mod_automorphism(fibers)     # long time             #␣
↪needs sage.graphs sage.groups
(frozenset({(-1, 0, 0, 0),
            (-1, 0, 0, 1),
            (0, 0, 0, -1),
            (0, 0, 0, 1),
            (1, 0, 0, -1),
            (1, 0, 0, 0)}),
 frozenset({(-1, 0, 0, 0), (-1, 1, 1, 0), (1, -1, -1, 0), (1, 0, 0, 0)}))
```

restricted_automorphism_group(*vertex_labels=None*)

Return the restricted automorphism group.

First, let the linear automorphism group be the subgroup of the Euclidean group $E(d) = GL(d, \mathbf{R}) \ltimes \mathbf{R}^d$ preserving the d-dimensional polyhedron. The Euclidean group acts in the usual way $\vec{x} \mapsto A\vec{x} + b$ on the ambient space. The restricted automorphism group is the subgroup of the linear automorphism group generated by permutations of vertices. If the polytope is full-dimensional, it is equal to the full (unrestricted) automorphism group.

INPUT:

- `vertex_labels` – tuple or `None` (default). The labels of the vertices that will be used in the output permutation group. By default, the vertices are used themselves.

OUTPUT:

A `PermutationGroup` acting on the vertices (or the `vertex_labels`, if specified).

REFERENCES:

[BSS2009]

EXAMPLES:

```
sage: # needs sage.graphs sage.groups
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import␣
↪LatticePolytope_PPL
sage: Z3square = LatticePolytope_PPL((0,0), (1,2), (2,1), (3,3))
sage: G1234 = Z3square.restricted_automorphism_group(
....:     vertex_labels=(1,2,3,4))
sage: G1234 == PermutationGroup([[(2,3)], [(1,2),(3,4)]])
True
sage: G = Z3square.restricted_automorphism_group()
sage: G == PermutationGroup([[((1,2),(2,1))], [((0,0),(1,2)),
....:                         ((2,1),(3,3))], [((0,0),(3,3))]])
```

(continues on next page)

(continued from previous page)

```
True
sage: set(G.domain()) == set(Z3square.vertices())
True
sage: (set(tuple(x) for x in G.orbit(Z3square.vertices()[0]))
....:    == set([(0, 0), (1, 2), (3, 3), (2, 1)]))
True
sage: cell24 = LatticePolytope_PPL(
....:     (1,0,0,0), (0,1,0,0), (0,0,1,0), (0,0,0,1), (1,-1,-1,1), (0,0,-1,1),
....:     (0,-1,0,1), (-1,0,0,1), (1,0,0,-1), (0,1,0,-1), (0,0,1,-1), (-1,1,1,
↪-1),
....:     (1,-1,-1,0), (0,0,-1,0), (0,-1,0,0), (-1,0,0,0), (1,-1,0,0), (1,0,-
↪1,0),
....:     (0,1,1,-1), (-1,1,1,0), (-1,1,0,0), (-1,0,1,0), (0,-1,-1,1), (0,0,0,
↪-1))
sage: cell24.restricted_automorphism_group().cardinality()
1152
```

```
>>> from sage.all import *
>>> # needs sage.graphs sage.groups
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> Z3square = LatticePolytope_PPL((Integer(0),Integer(0)), (Integer(1),
↪Integer(2)), (Integer(2),Integer(1)), (Integer(3),Integer(3)))
>>> G1234 = Z3square.restricted_automorphism_group(
...     vertex_labels=(Integer(1),Integer(2),Integer(3),Integer(4)))
>>> G1234 == PermutationGroup([[(Integer(2),Integer(3))], [(Integer(1),
↪Integer(2)),(Integer(3),Integer(4))]])
True
>>> G = Z3square.restricted_automorphism_group()
>>> G == PermutationGroup([[((Integer(1),Integer(2)),(Integer(2),
↪Integer(1)))], [((Integer(0),Integer(0)),(Integer(1),Integer(2))),
...                     ((Integer(2),Integer(1)),(Integer(3),
↪Integer(3)))], [((Integer(0),Integer(0)),(Integer(3),Integer(3)))]])
True
>>> set(G.domain()) == set(Z3square.vertices())
True
>>> (set(tuple(x) for x in G.orbit(Z3square.vertices()[Integer(0)]))
...    == set([(Integer(0), Integer(0)), (Integer(1), Integer(2)), (Integer(3),
↪ Integer(3)), (Integer(2), Integer(1))]))
True
>>> cell24 = LatticePolytope_PPL(
...     (Integer(1),Integer(0),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0),Integer(0)), (Integer(0),Integer(0),Integer(1),Integer(0)),␣
↪(Integer(0),Integer(0),Integer(0),Integer(1)), (Integer(1),-Integer(1),-
↪Integer(1),Integer(1)), (Integer(0),Integer(0),-Integer(1),Integer(1)),
...     (Integer(0),-Integer(1),Integer(0),Integer(1)), (-Integer(1),
↪Integer(0),Integer(0),Integer(1)), (Integer(1),Integer(0),Integer(0),-
↪Integer(1)), (Integer(0),Integer(1),Integer(0),-Integer(1)), (Integer(0),
↪Integer(0),Integer(1),-Integer(1)), (-Integer(1),Integer(1),Integer(1),-
↪Integer(1)),
...     (Integer(1),-Integer(1),-Integer(1),Integer(0)), (Integer(0),
↪Integer(0),-Integer(1),Integer(0)), (Integer(0),-Integer(1),Integer(0),
```

(continues on next page)

(continued from previous page)

```
↪Integer(0)), (-Integer(1),Integer(0),Integer(0),Integer(0)), (Integer(1),-
↪Integer(1),Integer(0),Integer(0)), (Integer(1),Integer(0),-Integer(1),
↪Integer(0)),
...      (Integer(0),Integer(1),Integer(1),-Integer(1)), (-Integer(1),
↪Integer(1),Integer(1),Integer(0)), (-Integer(1),Integer(1),Integer(0),
↪Integer(0)), (-Integer(1),Integer(0),Integer(1),Integer(0)), (Integer(0),-
↪Integer(1),-Integer(1),Integer(1)), (Integer(0),Integer(0),Integer(0),-
↪Integer(1)))
>>> cell24.restricted_automorphism_group().cardinality()
1152
```

sub_polytope_generator()

Generate the maximal lattice sub-polytopes.

OUTPUT:

A generator yielding the maximal (with respect to inclusion) lattice sub polytopes. That is, each can be gotten as the convex hull of the integral points of `self` with one vertex removed.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import␣
↪LatticePolytope_PPL
sage: P = LatticePolytope_PPL((1,0,0), (0,1,0), (0,0,1), (-1,-1,-1))
sage: for p in P.sub_polytope_generator():
....:     print(p.vertices())
((0, 0, 0), (0, 0, 1), (0, 1, 0), (1, 0, 0))
((-1, -1, -1), (0, 0, 0), (0, 1, 0), (1, 0, 0))
((-1, -1, -1), (0, 0, 0), (0, 0, 1), (1, 0, 0))
((-1, -1, -1), (0, 0, 0), (0, 0, 1), (0, 1, 0))
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> P = LatticePolytope_PPL((Integer(1),Integer(0),Integer(0)), (Integer(0),
↪Integer(1),Integer(0)), (Integer(0),Integer(0),Integer(1)), (-Integer(1),-
↪Integer(1),-Integer(1)))
>>> for p in P.sub_polytope_generator():
...     print(p.vertices())
((0, 0, 0), (0, 0, 1), (0, 1, 0), (1, 0, 0))
((-1, -1, -1), (0, 0, 0), (0, 1, 0), (1, 0, 0))
((-1, -1, -1), (0, 0, 0), (0, 0, 1), (1, 0, 0))
((-1, -1, -1), (0, 0, 0), (0, 0, 1), (0, 1, 0))
```

vertices()

Return the vertices as a tuple of $\mathbf{Z}$-vectors.

OUTPUT:

A tuple of $\mathbf{Z}$-vectors. Each entry is the coordinate vector of an integral points of the lattice polytope.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import␣
↪LatticePolytope_PPL
```

(continues on next page)

(continued from previous page)

```
sage: p = LatticePolytope_PPL((-9,-6,-1,-1),
....:                         (0,0,0,1), (0,0,1,0), (0,1,0,0), (1,0,0,0))
sage: p.vertices()
((-9, -6, -1, -1), (0, 0, 0, 1), (0, 0, 1, 0), (0, 1, 0, 0), (1, 0, 0, 0))
sage: p.minimized_generators()
Generator_System {point(-9/1, -6/1, -1/1, -1/1), point(0/1, 0/1, 0/1, 1/1),
point(0/1, 0/1, 1/1, 0/1), point(0/1, 1/1, 0/1, 0/1), point(1/1, 0/1, 0/1, 0/
↪1)}
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> p = LatticePolytope_PPL((-Integer(9),-Integer(6),-Integer(1),-Integer(1)),
...                         (Integer(0),Integer(0),Integer(0),Integer(1)),␣
↪(Integer(0),Integer(0),Integer(1),Integer(0)), (Integer(0),Integer(1),
↪Integer(0),Integer(0)), (Integer(1),Integer(0),Integer(0),Integer(0)))
>>> p.vertices()
((-9, -6, -1, -1), (0, 0, 0, 1), (0, 0, 1, 0), (0, 1, 0, 0), (1, 0, 0, 0))
>>> p.minimized_generators()
Generator_System {point(-9/1, -6/1, -1/1, -1/1), point(0/1, 0/1, 0/1, 1/1),
point(0/1, 0/1, 1/1, 0/1), point(0/1, 1/1, 0/1, 0/1), point(1/1, 0/1, 0/1, 0/
↪1)}
```

vertices_saturating(*constraint*)

Return the vertices saturating the constraint.

INPUT:

- `constraint` – a constraint (inequality or equation) of the polytope

OUTPUT:

The tuple of vertices saturating the constraint. The vertices are returned as $\mathbf{Z}$-vectors, as in *vertices()*.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import␣
↪LatticePolytope_PPL
sage: p = LatticePolytope_PPL((0,0), (0,1), (1,0))
sage: ieq = next(iter(p.constraints()));  ieq
x0>=0
sage: p.vertices_saturating(ieq)
((0, 0), (0, 1))
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
↪PPL
>>> p = LatticePolytope_PPL((Integer(0),Integer(0)), (Integer(0),Integer(1)),␣
↪(Integer(1),Integer(0)))
>>> ieq = next(iter(p.constraints()));  ieq
x0>=0
>>> p.vertices_saturating(ieq)
((0, 0), (0, 1))
```

2.2.6 Generating Function of Polyhedron's Integral Points

This module provides *generating_function_of_integral_points()* which computes the generating function of the integral points of a polyhedron.

The main function is accessible via `sage.geometry.polyhedron.base.Polyhedron_base.generating_function_of_integral_points()` as well.

Various

AUTHORS:

- Daniel Krenn (2016, 2021)

ACKNOWLEDGEMENT:

- Daniel Krenn is supported by the Austrian Science Fund (FWF): P 24644-N26 and by the Austrian Science Fund (FWF): P 28466-N35.

Functions

`sage.geometry.polyhedron.generating_function.`**`generating_function_of_integral_points`**(*polyhedron*, *split=False*, *result_as_tuple=None*, *name=None*, *names=None*, ***kwds*)

Return the multivariate generating function of the integral points of the `polyhedron`.

To be precise, this returns

$$\sum_{(r_0,\ldots,r_{d-1})\in\mathit{polyhedron}\cap\mathbf{Z}^d} y_0^{r_0}\cdots y_{d-1}^{r_{d-1}}.$$

INPUT:

- `polyhedron` – an instance of *Polyhedron_base* (see also *sage.geometry.polyhedron.constructor*)
- `split` – (default: `False`) a boolean or list
  - `split=False` computes the generating function directly, without any splitting.
  - When `split` is a list of disjoint polyhedra, then for each of these polyhedra, `polyhedron` is intersected with it, its generating function computed and all these generating functions are summed up.
  - `split=True` splits into $d!$ disjoint polyhedra.
- `result_as_tuple` – (default: `None`) a boolean or `None`

  This specifies whether the output is a (partial) factorization (`result_as_tuple=False`) or a sum of such (partial) factorizations (`result_as_tuple=True`). By default (`result_as_tuple=None`), this is automatically determined. If the output is a sum, it is represented as a tuple whose entries are the summands.
- `indices` – (default: `None`) a list or tuple

  If this is `None`, this is automatically determined.

- `name` – (default: `'y'`) a string

  The variable names of the Laurent polynomial ring of the output are this string followed by an integer.

- `names` – list or tuple of names (strings), or a comma separated string

  `name` is extracted from `names`, therefore `names` has to contain exactly one variable name, and `name` and``names`` cannot be specified both at the same time.

- `Factorization_sort` (default: `False`) and `Factorization_simplify` (default: `True`) – booleans

  These are passed on to `sage.structure.factorization.Factorization` when creating the result.

- `sort_factors` – (default: `False`) a boolean

  If set, then the factors of the output are sorted such that the numerator is first and only then all factors of the denominator. It is ensured that the sorting is always the same; use this for doctesting.

OUTPUT:

The generating function as a (partial) `Factorization` of the result whose factors are Laurent polynomials

The result might be a tuple of such factorizations (depending on the parameter `result_as_tuple`) as well.

> **Note**
>
> At the moment, only polyhedra with nonnegative coordinates (i.e. a polyhedron in the nonnegative orthant) are handled.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.generating_function import generating_
→function_of_integral_points
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.generating_function import generating_function_
→of_integral_points
```

```
sage: P2 = (
....:   Polyhedron(ieqs=[(0, 0, 0, 1), (0, 0, 1, 0), (0, 1, 0, -1)]),
....:   Polyhedron(ieqs=[(0, -1, 0, 1), (0, 1, 0, 0), (0, 0, 1, 0)]))
sage: generating_function_of_integral_points(P2[0], sort_factors=True)
1 * (-y0 + 1)^-1 * (-y1 + 1)^-1 * (-y0*y2 + 1)^-1
sage: generating_function_of_integral_points(P2[1], sort_factors=True)
1 * (-y1 + 1)^-1 * (-y2 + 1)^-1 * (-y0*y2 + 1)^-1
sage: (P2[0] & P2[1]).Hrepresentation()
(An equation (1, 0, -1) x + 0 == 0,
 An inequality (1, 0, 0) x + 0 >= 0,
 An inequality (0, 1, 0) x + 0 >= 0)
sage: generating_function_of_integral_points(P2[0] & P2[1], sort_factors=True)
1 * (-y1 + 1)^-1 * (-y0*y2 + 1)^-1
```

```
>>> from sage.all import *
>>> P2 = (
...   Polyhedron(ieqs=[(Integer(0), Integer(0), Integer(0), Integer(1)),␣
→(Integer(0), Integer(0), Integer(1), Integer(0)), (Integer(0), Integer(1),␣
→Integer(0), -Integer(1))]),
```

(continues on next page)

(continued from previous page)

```
...   Polyhedron(ieqs=[(Integer(0), -Integer(1), Integer(0), Integer(1)),␣
↪(Integer(0), Integer(1), Integer(0), Integer(0)), (Integer(0), Integer(0),␣
↪Integer(1), Integer(0))]))
>>> generating_function_of_integral_points(P2[Integer(0)], sort_factors=True)
1 * (-y0 + 1)^-1 * (-y1 + 1)^-1 * (-y0*y2 + 1)^-1
>>> generating_function_of_integral_points(P2[Integer(1)], sort_factors=True)
1 * (-y1 + 1)^-1 * (-y2 + 1)^-1 * (-y0*y2 + 1)^-1
>>> (P2[Integer(0)] & P2[Integer(1)]).Hrepresentation()
(An equation (1, 0, -1) x + 0 == 0,
 An inequality (1, 0, 0) x + 0 >= 0,
 An inequality (0, 1, 0) x + 0 >= 0)
>>> generating_function_of_integral_points(P2[Integer(0)] & P2[Integer(1)], sort_
↪factors=True)
1 * (-y1 + 1)^-1 * (-y0*y2 + 1)^-1
```

```
sage: P3 = (
....:   Polyhedron(
....:     ieqs=[(0, 0, 0, 0, 1), (0, 0, 0, 1, 0),
....:           (0, 0, 1, 0, -1), (-1, 1, 0, -1, -1)]),
....:   Polyhedron(
....:     ieqs=[(0, 0, -1, 0, 1), (0, 1, 0, 0, -1),
....:           (0, 0, 0, 1, 0), (0, 0, 1, 0, 0), (-1, 1, -1, -1, 0)]),
....:   Polyhedron(
....:     ieqs=[(1, -1, 0, 1, 1), (1, -1, 1, 1, 0),
....:           (0, 0, 0, 0, 1), (0, 0, 0, 1, 0), (0, 0, 1, 0, 0),
....:           (1, 0, 1, 1, -1), (0, 1, 0, 0, 0), (1, 1, 1, 0, -1)]),
....:   Polyhedron(
....:     ieqs=[(0, 1, 0, -1, 0), (0, -1, 0, 0, 1),
....:           (-1, 0, -1, -1, 1), (0, 0, 1, 0, 0), (0, 0, 0, 1, 0)]),
....:   Polyhedron(
....:     ieqs=[(0, 1, 0, 0, 0), (0, 0, 1, 0, 0),
....:           (-1, -1, -1, 0, 1), (0, -1, 0, 1, 0)]))
sage: def intersect(I):
....:     I = iter(I)
....:     result = next(I)
....:     for i in I:
....:         result &= i
....:     return result
sage: for J in subsets(range(len(P3))):
....:     if not J:
....:         continue
....:     P = intersect([P3[j] for j in J])
....:     print('{}: {}'.format(J, P.Hrepresentation()))
....:     print(generating_function_of_integral_points(P, sort_factors=True))
[0]: (An inequality (0, 0, 0, 1) x + 0 >= 0,
      An inequality (0, 0, 1, 0) x + 0 >= 0,
      An inequality (0, 1, 0, -1) x + 0 >= 0,
      An inequality (1, 0, -1, -1) x - 1 >= 0)
y0 * (-y0 + 1)^-1 * (-y1 + 1)^-1 * (-y0*y2 + 1)^-1 * (-y0*y1*y3 + 1)^-1
[1]: (An inequality (0, -1, 0, 1) x + 0 >= 0,
      An inequality (0, 0, 1, 0) x + 0 >= 0,
      An inequality (0, 1, 0, 0) x + 0 >= 0,
```

(continues on next page)

(continued from previous page)

```
      An inequality (1, -1, -1, 0) x - 1 >= 0,
      An inequality (1, 0, 0, -1) x + 0 >= 0)
(-y0^2*y2*y3 - y0^2*y3 + y0*y3 + y0) *
(-y0 + 1)^-1 * (-y0*y2 + 1)^-1 * (-y0*y3 + 1)^-1 *
(-y0*y1*y3 + 1)^-1 * (-y0*y2*y3 + 1)^-1
[0, 1]: (An equation (0, 1, 0, -1) x + 0 == 0,
         An inequality (1, -1, -1, 0) x - 1 >= 0,
         An inequality (0, 1, 0, 0) x + 0 >= 0,
         An inequality (0, 0, 1, 0) x + 0 >= 0)
y0 * (-y0 + 1)^-1 * (-y0*y2 + 1)^-1 * (-y0*y1*y3 + 1)^-1
[2]: (An inequality (-1, 0, 1, 1) x + 1 >= 0,
     An inequality (-1, 1, 1, 0) x + 1 >= 0,
      An inequality (0, 0, 0, 1) x + 0 >= 0,
      An inequality (0, 0, 1, 0) x + 0 >= 0,
      An inequality (0, 1, 0, 0) x + 0 >= 0,
      An inequality (0, 1, 1, -1) x + 1 >= 0,
      An inequality (1, 0, 0, 0) x + 0 >= 0,
      An inequality (1, 1, 0, -1) x + 1 >= 0)
(y0^2*y1*y2*y3^2 + y0^2*y2^2*y3 + y0*y1^2*y3^2 - y0^2*y2*y3 +
 y0*y1*y2*y3 - y0*y1*y3^2 - 2*y0*y1*y3 - 2*y0*y2*y3 - y0*y2 +
 y0*y3 - y1*y3 + y0 + y3 + 1) *
(-y1 + 1)^-1 * (-y2 + 1)^-1 * (-y0*y2 + 1)^-1 *
(-y1*y3 + 1)^-1 * (-y0*y1*y3 + 1)^-1 * (-y0*y2*y3 + 1)^-1
[0, 2]: (An equation (1, 0, -1, -1) x - 1 == 0,
         An inequality (-1, 1, 1, 0) x + 1 >= 0,
         An inequality (1, 0, -1, 0) x - 1 >= 0,
         An inequality (0, 0, 1, 0) x + 0 >= 0)
y0 * (-y1 + 1)^-1 * (-y0*y2 + 1)^-1 * (-y0*y1*y3 + 1)^-1
[1, 2]: (An equation (1, -1, -1, 0) x - 1 == 0,
         An inequality (0, -1, 0, 1) x + 0 >= 0,
         An inequality (0, 1, 0, 0) x + 0 >= 0,
         An inequality (1, 0, 0, -1) x + 0 >= 0,
         An inequality (1, -1, 0, 0) x - 1 >= 0)
(-y0^2*y2*y3 + y0*y3 + y0) *
(-y0*y2 + 1)^-1 * (-y0*y1*y3 + 1)^-1 * (-y0*y2*y3 + 1)^-1
[0, 1, 2]: (An equation (0, 1, 0, -1) x + 0 == 0,
            An equation (1, -1, -1, 0) x - 1 == 0,
            An inequality (0, 1, 0, 0) x + 0 >= 0,
            An inequality (1, -1, 0, 0) x - 1 >= 0)
y0 * (-y0*y2 + 1)^-1 * (-y0*y1*y3 + 1)^-1
[3]: (An inequality (-1, 0, 0, 1) x + 0 >= 0,
      An inequality (0, -1, -1, 1) x - 1 >= 0,
      An inequality (0, 0, 1, 0) x + 0 >= 0,
      An inequality (0, 1, 0, 0) x + 0 >= 0,
      An inequality (1, 0, -1, 0) x + 0 >= 0)
(-y0*y1*y3^2 - y0*y3^2 + y0*y3 + y3) *
(-y3 + 1)^-1 * (-y0*y3 + 1)^-1 *
(-y1*y3 + 1)^-1 * (-y0*y1*y3 + 1)^-1 * (-y0*y2*y3 + 1)^-1
[0, 3]: (An equation -1 == 0,)
0
[1, 3]: (An equation (1, 0, 0, -1) x + 0 == 0,
         An inequality (1, -1, -1, 0) x - 1 >= 0,
```

(continues on next page)

(continued from previous page)

```
         An inequality (0, 1, 0, 0) x + 0 >= 0,
         An inequality (0, 0, 1, 0) x + 0 >= 0)
y0*y3 * (-y0*y3 + 1)^-1 * (-y0*y1*y3 + 1)^-1 * (-y0*y2*y3 + 1)^-1
[0, 1, 3]: (An equation -1 == 0,)
0
[2, 3]: (An equation (0, 1, 1, -1) x + 1 == 0,
         An inequality (1, 0, -1, 0) x + 0 >= 0,
         An inequality (-1, 1, 1, 0) x + 1 >= 0,
         An inequality (0, 0, 1, 0) x + 0 >= 0,
         An inequality (0, 1, 0, 0) x + 0 >= 0)
(-y0*y1*y3^2 + y0*y3 + y3) *
(-y1*y3 + 1)^-1 * (-y0*y1*y3 + 1)^-1 * (-y0*y2*y3 + 1)^-1
[0, 2, 3]: (An equation -1 == 0,)
0
[1, 2, 3]: (An equation (1, 0, 0, -1) x + 0 == 0,
            An equation (1, -1, -1, 0) x - 1 == 0,
            An inequality (0, 1, 0, 0) x + 0 >= 0,
            An inequality (1, -1, 0, 0) x - 1 >= 0)
y0*y3 * (-y0*y1*y3 + 1)^-1 * (-y0*y2*y3 + 1)^-1
[0, 1, 2, 3]: (An equation -1 == 0,)
0
[4]: (An inequality (-1, -1, 0, 1) x - 1 >= 0,
      An inequality (-1, 0, 1, 0) x + 0 >= 0,
      An inequality (0, 1, 0, 0) x + 0 >= 0,
      An inequality (1, 0, 0, 0) x + 0 >= 0)
y3 * (-y2 + 1)^-1 * (-y3 + 1)^-1 * (-y1*y3 + 1)^-1 * (-y0*y2*y3 + 1)^-1
[0, 4]: (An equation -1 == 0,)
0
[1, 4]: (An equation -1 == 0,)
0
[0, 1, 4]: (An equation -1 == 0,)
0
[2, 4]: (An equation (1, 1, 0, -1) x + 1 == 0,
         An inequality (-1, 0, 1, 0) x + 0 >= 0,
         An inequality (1, 0, 0, 0) x + 0 >= 0,
         An inequality (0, 1, 0, 0) x + 0 >= 0)
y3 * (-y2 + 1)^-1 * (-y1*y3 + 1)^-1 * (-y0*y2*y3 + 1)^-1
[0, 2, 4]: (An equation -1 == 0,)
0
[1, 2, 4]: (An equation -1 == 0,)
0
[0, 1, 2, 4]: (An equation -1 == 0,)
0
[3, 4]: (An equation (1, 0, -1, 0) x + 0 == 0,
         An inequality (0, 1, 0, 0) x + 0 >= 0,
         An inequality (-1, -1, 0, 1) x - 1 >= 0,
         An inequality (1, 0, 0, 0) x + 0 >= 0)
y3 * (-y3 + 1)^-1 * (-y1*y3 + 1)^-1 * (-y0*y2*y3 + 1)^-1
[0, 3, 4]: (An equation -1 == 0,)
0
[1, 3, 4]: (An equation -1 == 0,)
0
```

(continues on next page)

(continued from previous page)

```
[0, 1, 3, 4]: (An equation -1 == 0,)
0
[2, 3, 4]: (An equation (1, 1, 0, -1) x + 1 == 0,
            An equation (1, 0, -1, 0) x + 0 == 0,
            An inequality (0, 1, 0, 0) x + 0 >= 0,
            An inequality (1, 0, 0, 0) x + 0 >= 0)
y3 * (-y1*y3 + 1)^-1 * (-y0*y2*y3 + 1)^-1
[0, 2, 3, 4]: (An equation -1 == 0,)
0
[1, 2, 3, 4]: (An equation -1 == 0,)
0
[0, 1, 2, 3, 4]: (An equation -1 == 0,)
0
```

```
>>> from sage.all import *
>>> P3 = (
...   Polyhedron(
...     ieqs=[(Integer(0), Integer(0), Integer(0), Integer(0), Integer(1)),␣
↪(Integer(0), Integer(0), Integer(0), Integer(1), Integer(0)),
...           (Integer(0), Integer(0), Integer(1), Integer(0), -Integer(1)), (-
↪Integer(1), Integer(1), Integer(0), -Integer(1), -Integer(1))]),
...   Polyhedron(
...     ieqs=[(Integer(0), Integer(0), -Integer(1), Integer(0), Integer(1)),␣
↪(Integer(0), Integer(1), Integer(0), Integer(0), -Integer(1)),
...           (Integer(0), Integer(0), Integer(0), Integer(1), Integer(0)),␣
↪(Integer(0), Integer(0), Integer(1), Integer(0), Integer(0)), (-Integer(1),␣
↪Integer(1), -Integer(1), -Integer(1), Integer(0))]),
...   Polyhedron(
...     ieqs=[(Integer(1), -Integer(1), Integer(0), Integer(1), Integer(1)),␣
↪(Integer(1), -Integer(1), Integer(1), Integer(1), Integer(0)),
...           (Integer(0), Integer(0), Integer(0), Integer(0), Integer(1)),␣
↪(Integer(0), Integer(0), Integer(0), Integer(1), Integer(0)), (Integer(0),␣
↪Integer(0), Integer(1), Integer(0), Integer(0)),
...           (Integer(1), Integer(0), Integer(1), Integer(1), -Integer(1)),␣
↪(Integer(0), Integer(1), Integer(0), Integer(0), Integer(0)), (Integer(1),␣
↪Integer(1), Integer(1), Integer(0), -Integer(1))]),
...   Polyhedron(
...     ieqs=[(Integer(0), Integer(1), Integer(0), -Integer(1), Integer(0)),␣
↪(Integer(0), -Integer(1), Integer(0), Integer(0), Integer(1)),
...           (-Integer(1), Integer(0), -Integer(1), -Integer(1), Integer(1)),␣
↪(Integer(0), Integer(0), Integer(1), Integer(0), Integer(0)), (Integer(0),␣
↪Integer(0), Integer(0), Integer(1), Integer(0))]),
...   Polyhedron(
...     ieqs=[(Integer(0), Integer(1), Integer(0), Integer(0), Integer(0)),␣
↪(Integer(0), Integer(0), Integer(1), Integer(0), Integer(0)),
...           (-Integer(1), -Integer(1), -Integer(1), Integer(0), Integer(1)),␣
↪(Integer(0), -Integer(1), Integer(0), Integer(1), Integer(0))]))
>>> def intersect(I):
...     I = iter(I)
...     result = next(I)
...     for i in I:
...         result &= i
```

(continues on next page)

(continued from previous page)

```
...     return result
>>> for J in subsets(range(len(P3))):
...     if not J:
...         continue
...     P = intersect([P3[j] for j in J])
...     print('{}: {}'.format(J, P.Hrepresentation()))
...     print(generating_function_of_integral_points(P, sort_factors=True))
[0]: (An inequality (0, 0, 0, 1) x + 0 >= 0,
      An inequality (0, 0, 1, 0) x + 0 >= 0,
      An inequality (0, 1, 0, -1) x + 0 >= 0,
      An inequality (1, 0, -1, -1) x - 1 >= 0)
y0 * (-y0 + 1)^-1 * (-y1 + 1)^-1 * (-y0*y2 + 1)^-1 * (-y0*y1*y3 + 1)^-1
[1]: (An inequality (0, -1, 0, 1) x + 0 >= 0,
      An inequality (0, 0, 1, 0) x + 0 >= 0,
      An inequality (0, 1, 0, 0) x + 0 >= 0,
      An inequality (1, -1, -1, 0) x - 1 >= 0,
      An inequality (1, 0, 0, -1) x + 0 >= 0)
(-y0^2*y2*y3 - y0^2*y3 + y0*y3 + y0) *
(-y0 + 1)^-1 * (-y0*y2 + 1)^-1 * (-y0*y3 + 1)^-1 *
(-y0*y1*y3 + 1)^-1 * (-y0*y2*y3 + 1)^-1
[0, 1]: (An equation (0, 1, 0, -1) x + 0 == 0,
         An inequality (1, -1, -1, 0) x - 1 >= 0,
         An inequality (0, 1, 0, 0) x + 0 >= 0,
         An inequality (0, 0, 1, 0) x + 0 >= 0)
y0 * (-y0 + 1)^-1 * (-y0*y2 + 1)^-1 * (-y0*y1*y3 + 1)^-1
[2]: (An inequality (-1, 0, 1, 1) x + 1 >= 0,
     An inequality (-1, 1, 1, 0) x + 1 >= 0,
      An inequality (0, 0, 0, 1) x + 0 >= 0,
      An inequality (0, 0, 1, 0) x + 0 >= 0,
      An inequality (0, 1, 0, 0) x + 0 >= 0,
      An inequality (0, 1, 1, -1) x + 1 >= 0,
      An inequality (1, 0, 0, 0) x + 0 >= 0,
      An inequality (1, 1, 0, -1) x + 1 >= 0)
(y0^2*y1*y2*y3^2 + y0^2*y2^2*y3 + y0*y1^2*y3^2 - y0^2*y2*y3 +
 y0*y1*y2*y3 - y0*y1*y3^2 - 2*y0*y1*y3 - 2*y0*y2*y3 - y0*y2 +
 y0*y3 - y1*y3 + y0 + y3 + 1) *
(-y1 + 1)^-1 * (-y2 + 1)^-1 * (-y0*y2 + 1)^-1 *
(-y1*y3 + 1)^-1 * (-y0*y1*y3 + 1)^-1 * (-y0*y2*y3 + 1)^-1
[0, 2]: (An equation (1, 0, -1, -1) x - 1 == 0,
         An inequality (-1, 1, 1, 0) x + 1 >= 0,
         An inequality (1, 0, -1, 0) x - 1 >= 0,
         An inequality (0, 0, 1, 0) x + 0 >= 0)
y0 * (-y1 + 1)^-1 * (-y0*y2 + 1)^-1 * (-y0*y1*y3 + 1)^-1
[1, 2]: (An equation (1, -1, -1, 0) x - 1 == 0,
         An inequality (0, -1, 0, 1) x + 0 >= 0,
         An inequality (0, 1, 0, 0) x + 0 >= 0,
         An inequality (1, 0, 0, -1) x + 0 >= 0,
         An inequality (1, -1, 0, 0) x - 1 >= 0)
(-y0^2*y2*y3 + y0*y3 + y0) *
(-y0*y2 + 1)^-1 * (-y0*y1*y3 + 1)^-1 * (-y0*y2*y3 + 1)^-1
[0, 1, 2]: (An equation (0, 1, 0, -1) x + 0 == 0,
            An equation (1, -1, -1, 0) x - 1 == 0,
```

(continues on next page)

(continued from previous page)

```
            An inequality (0, 1, 0, 0) x + 0 >= 0,
            An inequality (1, -1, 0, 0) x - 1 >= 0)
y0 * (-y0*y2 + 1)^-1 * (-y0*y1*y3 + 1)^-1
[3]: (An inequality (-1, 0, 0, 1) x + 0 >= 0,
      An inequality (0, -1, -1, 1) x - 1 >= 0,
      An inequality (0, 0, 1, 0) x + 0 >= 0,
      An inequality (0, 1, 0, 0) x + 0 >= 0,
      An inequality (1, 0, -1, 0) x + 0 >= 0)
(-y0*y1*y3^2 - y0*y3^2 + y0*y3 + y3) *
(-y3 + 1)^-1 * (-y0*y3 + 1)^-1 *
(-y1*y3 + 1)^-1 * (-y0*y1*y3 + 1)^-1 * (-y0*y2*y3 + 1)^-1
[0, 3]: (An equation -1 == 0,)
0
[1, 3]: (An equation (1, 0, 0, -1) x + 0 == 0,
         An inequality (1, -1, -1, 0) x - 1 >= 0,
         An inequality (0, 1, 0, 0) x + 0 >= 0,
         An inequality (0, 0, 1, 0) x + 0 >= 0)
y0*y3 * (-y0*y3 + 1)^-1 * (-y0*y1*y3 + 1)^-1 * (-y0*y2*y3 + 1)^-1
[0, 1, 3]: (An equation -1 == 0,)
0
[2, 3]: (An equation (0, 1, 1, -1) x + 1 == 0,
         An inequality (1, 0, -1, 0) x + 0 >= 0,
         An inequality (-1, 1, 1, 0) x + 1 >= 0,
         An inequality (0, 0, 1, 0) x + 0 >= 0,
         An inequality (0, 1, 0, 0) x + 0 >= 0)
(-y0*y1*y3^2 + y0*y3 + y3) *
(-y1*y3 + 1)^-1 * (-y0*y1*y3 + 1)^-1 * (-y0*y2*y3 + 1)^-1
[0, 2, 3]: (An equation -1 == 0,)
0
[1, 2, 3]: (An equation (1, 0, 0, -1) x + 0 == 0,
            An equation (1, -1, -1, 0) x - 1 == 0,
            An inequality (0, 1, 0, 0) x + 0 >= 0,
            An inequality (1, -1, 0, 0) x - 1 >= 0)
y0*y3 * (-y0*y1*y3 + 1)^-1 * (-y0*y2*y3 + 1)^-1
[0, 1, 2, 3]: (An equation -1 == 0,)
0
[4]: (An inequality (-1, -1, 0, 1) x - 1 >= 0,
      An inequality (-1, 0, 1, 0) x + 0 >= 0,
      An inequality (0, 1, 0, 0) x + 0 >= 0,
      An inequality (1, 0, 0, 0) x + 0 >= 0)
y3 * (-y2 + 1)^-1 * (-y3 + 1)^-1 * (-y1*y3 + 1)^-1 * (-y0*y2*y3 + 1)^-1
[0, 4]: (An equation -1 == 0,)
0
[1, 4]: (An equation -1 == 0,)
0
[0, 1, 4]: (An equation -1 == 0,)
0
[2, 4]: (An equation (1, 1, 0, -1) x + 1 == 0,
         An inequality (-1, 0, 1, 0) x + 0 >= 0,
         An inequality (1, 0, 0, 0) x + 0 >= 0,
         An inequality (0, 1, 0, 0) x + 0 >= 0)
y3 * (-y2 + 1)^-1 * (-y1*y3 + 1)^-1 * (-y0*y2*y3 + 1)^-1
```

(continues on next page)

(continued from previous page)

```
[0, 2, 4]: (An equation -1 == 0,)
0
[1, 2, 4]: (An equation -1 == 0,)
0
[0, 1, 2, 4]: (An equation -1 == 0,)
0
[3, 4]: (An equation (1, 0, -1, 0) x + 0 == 0,
         An inequality (0, 1, 0, 0) x + 0 >= 0,
         An inequality (-1, -1, 0, 1) x - 1 >= 0,
         An inequality (1, 0, 0, 0) x + 0 >= 0)
y3 * (-y3 + 1)^-1 * (-y1*y3 + 1)^-1 * (-y0*y2*y3 + 1)^-1
[0, 3, 4]: (An equation -1 == 0,)
0
[1, 3, 4]: (An equation -1 == 0,)
0
[0, 1, 3, 4]: (An equation -1 == 0,)
0
[2, 3, 4]: (An equation (1, 1, 0, -1) x + 1 == 0,
            An equation (1, 0, -1, 0) x + 0 == 0,
            An inequality (0, 1, 0, 0) x + 0 >= 0,
            An inequality (1, 0, 0, 0) x + 0 >= 0)
y3 * (-y1*y3 + 1)^-1 * (-y0*y2*y3 + 1)^-1
[0, 2, 3, 4]: (An equation -1 == 0,)
0
[1, 2, 3, 4]: (An equation -1 == 0,)
0
[0, 1, 2, 3, 4]: (An equation -1 == 0,)
0
```

```
sage: P = Polyhedron(vertices=[[1], [5]])
sage: P.generating_function_of_integral_points()
y0^5 + y0^4 + y0^3 + y0^2 + y0
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[[Integer(1)], [Integer(5)]])
>>> P.generating_function_of_integral_points()
y0^5 + y0^4 + y0^3 + y0^2 + y0
```

See also

This function is accessible via `sage.geometry.polyhedron.base.Polyhedron_base.generating_function_of_integral_points()` as well. More examples can be found there.

2.3 Combinatorial Polyhedra

2.3.1 Combinatorial polyhedron

This module gathers algorithms for polyhedra that only depend on the vertex-facet incidences and that are called combinatorial polyhedron. The main class is `CombinatorialPolyhedron`. Most importantly, this class allows to iterate quickly through the faces (possibly of given dimension) via the `FaceIterator` object. The `CombinatorialPolyhe-`

dron uses this iterator to quickly generate the f-vector, the edges, the ridges and the face lattice.

Terminology used in this module:

- Vrep – `[vertices, rays, lines]` of the polyhedron
- Hrep – inequalities and equations of the polyhedron
- Facets – facets of the polyhedron
- Vrepresentation – represents a face by the list of Vrep it contains
- Hrepresentation – represents a face by a list of Hrep it is contained in
- bit representation – represents incidences as bitset, where each bit represents one incidence. There might be trailing zeros, to fit alignment requirements. In most instances, faces are represented by the bit representation, where each bit corresponds to a Vrep or facet. Thus a bit representation can either be a Vrep or facet representation depending on context.

EXAMPLES:

Construction:

```
sage: P = polytopes.hypercube(4)
sage: C = CombinatorialPolyhedron(P); C
A 4-dimensional combinatorial polyhedron with 8 facets
```

```
>>> from sage.all import *
>>> P = polytopes.hypercube(Integer(4))
>>> C = CombinatorialPolyhedron(P); C
A 4-dimensional combinatorial polyhedron with 8 facets
```

Obtaining edges and ridges:

```
sage: C.edges()[:2]
((A vertex at (1, -1, -1, -1), A vertex at (-1, -1, -1, -1)),
 (A vertex at (-1, -1, -1, 1), A vertex at (-1, -1, -1, -1)))
sage: C.edges(names=False)[:2]
((6, 15), (14, 15))

sage: C.ridges()[:2]
((An inequality (0, 0, 1, 0) x + 1 >= 0,
  An inequality (0, 1, 0, 0) x + 1 >= 0),
 (An inequality (0, 0, 0, 1) x + 1 >= 0,
  An inequality (0, 1, 0, 0) x + 1 >= 0))
sage: C.ridges(names=False)[:2]
((6, 7), (5, 7))
```

```
>>> from sage.all import *
>>> C.edges()[:Integer(2)]
((A vertex at (1, -1, -1, -1), A vertex at (-1, -1, -1, -1)),
 (A vertex at (-1, -1, -1, 1), A vertex at (-1, -1, -1, -1)))
>>> C.edges(names=False)[:Integer(2)]
((6, 15), (14, 15))

>>> C.ridges()[:Integer(2)]
((An inequality (0, 0, 1, 0) x + 1 >= 0,
```

(continues on next page)

(continued from previous page)

```
 An inequality (0, 1, 0, 0) x + 1 >= 0),
(An inequality (0, 0, 0, 1) x + 1 >= 0,
 An inequality (0, 1, 0, 0) x + 1 >= 0))
>>> C.ridges(names=False)[:Integer(2)]
((6, 7), (5, 7))
```

Vertex-graph and facet-graph:

```
sage: C.vertex_graph()                                                    #␣
↪needs sage.graphs
Graph on 16 vertices
sage: C.facet_graph()                                                     #␣
↪needs sage.graphs
Graph on 8 vertices
```

```
>>> from sage.all import *
>>> C.vertex_graph()                                                    #␣
↪needs sage.graphs
Graph on 16 vertices
>>> C.facet_graph()                                                     #␣
↪needs sage.graphs
Graph on 8 vertices
```

Face lattice:

```
sage: C.face_lattice()                                                    #␣
↪needs sage.combinat
Finite lattice containing 82 elements
```

```
>>> from sage.all import *
>>> C.face_lattice()                                                    #␣
↪needs sage.combinat
Finite lattice containing 82 elements
```

Face iterator:

```
sage: C.face_generator()
Iterator over the proper faces of a 4-dimensional combinatorial polyhedron

sage: C.face_generator(2)
Iterator over the 2-faces of a 4-dimensional combinatorial polyhedron
```

```
>>> from sage.all import *
>>> C.face_generator()
Iterator over the proper faces of a 4-dimensional combinatorial polyhedron

>>> C.face_generator(Integer(2))
Iterator over the 2-faces of a 4-dimensional combinatorial polyhedron
```

AUTHOR:

- Jonathan Kliem (2019-04)

class sage.geometry.polyhedron.combinatorial_polyhedron.base.**CombinatorialPolyhedron**

Bases: `SageObject`

The class of the Combinatorial Type of a Polyhedron, a Polytope.

INPUT:

- **data – an instance of**
  - `Polyhedron_base`
  - or a *LatticePolytopeClass*
  - or a *ConvexRationalPolyhedralCone*
  - or an `incidence_matrix` as in `incidence_matrix()` In this case you should also specify the `Vrep` and `facets` arguments
  - or list of facets, each facet given as a list of `[vertices, rays, lines]` if the polyhedron is unbounded, then rays and lines and the extra argument `nr_lines` are required if the polyhedron contains no lines, the rays can be thought of as the vertices of the facets deleted from a bounded polyhedron see `Polyhedron_base` on how to use rays and lines
  - or an integer, representing the dimension of a polyhedron equal to its affine hull
  - or a tuple consisting of facets and vertices as two *ListOfFaces*.
- `Vrep` – (optional) when `data` is an incidence matrix, it should be the list of `[vertices, rays, lines]`, if the rows in the incidence_matrix should correspond to names
- `facets` – (optional) when `data` is an incidence matrix or a list of facets, it should be a list of facets that would be used instead of indices (of the columns of the incidence matrix).
- `unbounded` – value will be overwritten if `data` is a polyhedron; if `unbounded` and `data` is incidence matrix or a list of facets, need to specify `far_face`
- `far_face` – (semi-optional); if the polyhedron is unbounded this needs to be set to the list of indices of the rays and line unless `data` is an instance of `Polyhedron_base`.

EXAMPLES:

We illustrate all possible input: a polyhedron:

> sage: P = polytopes.cube() sage: CombinatorialPolyhedron(P) A 3-dimensional combinatorial polyhedron with 6 facets

a lattice polytope:

```
sage: points = [(1,0,0), (0,1,0), (0,0,1),
....: (-1,0,0), (0,-1,0), (0,0,-1)]
sage: L = LatticePolytope(points)
sage: CombinatorialPolyhedron(L)
A 3-dimensional combinatorial polyhedron with 8 facets
```

```
>>> from sage.all import *
>>> points = [(Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0)), (Integer(0),Integer(0),Integer(1)),
... (-Integer(1),Integer(0),Integer(0)), (Integer(0),-Integer(1),Integer(0)),␣
↪(Integer(0),Integer(0),-Integer(1))]
>>> L = LatticePolytope(points)
>>> CombinatorialPolyhedron(L)
A 3-dimensional combinatorial polyhedron with 8 facets
```

a cone:

```
sage: M = Cone([(1,0), (0,1)])
sage: CombinatorialPolyhedron(M)
A 2-dimensional combinatorial polyhedron with 2 facets
```

```
>>> from sage.all import *
>>> M = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> CombinatorialPolyhedron(M)
A 2-dimensional combinatorial polyhedron with 2 facets
```

an incidence matrix:

```
sage: P = Polyhedron(rays=[[0,1]])
sage: data = P.incidence_matrix()
sage: far_face = [i for i in range(2) if not P.Vrepresentation()[i].is_vertex()]
sage: CombinatorialPolyhedron(data, unbounded=True, far_face=far_face)
A 1-dimensional combinatorial polyhedron with 1 facet
sage: C = CombinatorialPolyhedron(data, Vrep=['myvertex'],
....: facets=['myfacet'], unbounded=True, far_face=far_face)
sage: C.Vrepresentation()
('myvertex',)
sage: C.Hrepresentation()
('myfacet',)
```

```
>>> from sage.all import *
>>> P = Polyhedron(rays=[[Integer(0),Integer(1)]])
>>> data = P.incidence_matrix()
>>> far_face = [i for i in range(Integer(2)) if not P.Vrepresentation()[i].is_
↪vertex()]
>>> CombinatorialPolyhedron(data, unbounded=True, far_face=far_face)
A 1-dimensional combinatorial polyhedron with 1 facet
>>> C = CombinatorialPolyhedron(data, Vrep=['myvertex'],
... facets=['myfacet'], unbounded=True, far_face=far_face)
>>> C.Vrepresentation()
('myvertex',)
>>> C.Hrepresentation()
('myfacet',)
```

a list of facets:

```
sage: CombinatorialPolyhedron(((1,2,3),(1,2,4),(1,3,4),(2,3,4)))
A 3-dimensional combinatorial polyhedron with 4 facets
sage: facetnames = ['facet0', 'facet1', 'facet2', 'myfacet3']
sage: facetinc = ((1,2,3),(1,2,4),(1,3,4),(2,3,4))
sage: C = CombinatorialPolyhedron(facetinc, facets=facetnames)
sage: C.Vrepresentation()
(1, 2, 3, 4)
sage: C.Hrepresentation()
('facet0', 'facet1', 'facet2', 'myfacet3')
```

```
>>> from sage.all import *
>>> CombinatorialPolyhedron(((Integer(1),Integer(2),Integer(3)),(Integer(1),
```

(continues on next page)

(continued from previous page)

```
↪Integer(2),Integer(4)),(Integer(1),Integer(3),Integer(4)),(Integer(2),
↪Integer(3),Integer(4))))
A 3-dimensional combinatorial polyhedron with 4 facets
>>> facetnames = ['facet0', 'facet1', 'facet2', 'myfacet3']
>>> facetinc = ((Integer(1),Integer(2),Integer(3)),(Integer(1),Integer(2),
↪Integer(4)),(Integer(1),Integer(3),Integer(4)),(Integer(2),Integer(3),
↪Integer(4)))
>>> C = CombinatorialPolyhedron(facetinc, facets=facetnames)
>>> C.Vrepresentation()
(1, 2, 3, 4)
>>> C.Hrepresentation()
('facet0', 'facet1', 'facet2', 'myfacet3')
```

an integer:

```
sage: CombinatorialPolyhedron(-1).f_vector()
(1)
sage: CombinatorialPolyhedron(0).f_vector()
(1, 1)
sage: CombinatorialPolyhedron(5).f_vector()
(1, 0, 0, 0, 0, 0, 1)
```

```
>>> from sage.all import *
>>> CombinatorialPolyhedron(-Integer(1)).f_vector()
(1)
>>> CombinatorialPolyhedron(Integer(0)).f_vector()
(1, 1)
>>> CombinatorialPolyhedron(Integer(5)).f_vector()
(1, 0, 0, 0, 0, 0, 1)
```

tuple of `ListOfFaces`:

```
sage: from sage.geometry.polyhedron.combinatorial_polyhedron.conversions \
....:     import facets_tuple_to_bit_rep_of_facets, \
....:            facets_tuple_to_bit_rep_of_Vrep
sage: bi_pyr = ((0,1,4), (1,2,4), (2,3,4), (3,0,4),
....:           (0,1,5), (1,2,5), (2,3,5), (3,0,5))
sage: facets = facets_tuple_to_bit_rep_of_facets(bi_pyr, 6)
sage: Vrep = facets_tuple_to_bit_rep_of_Vrep(bi_pyr, 6)
sage: C = CombinatorialPolyhedron((facets, Vrep)); C
A 3-dimensional combinatorial polyhedron with 8 facets
sage: C.f_vector()
(1, 6, 12, 8, 1)
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.combinatorial_polyhedron.conversions     import␣
↪facets_tuple_to_bit_rep_of_facets,            facets_tuple_to_bit_rep_of_Vrep
>>> bi_pyr = ((Integer(0),Integer(1),Integer(4)), (Integer(1),Integer(2),
↪Integer(4)), (Integer(2),Integer(3),Integer(4)), (Integer(3),Integer(0),
↪Integer(4)),
...           (Integer(0),Integer(1),Integer(5)), (Integer(1),Integer(2),
↪Integer(5)), (Integer(2),Integer(3),Integer(5)), (Integer(3),Integer(0),
```

(continues on next page)

(continued from previous page)

```
↪Integer(5)))
>>> facets = facets_tuple_to_bit_rep_of_facets(bi_pyr, Integer(6))
>>> Vrep = facets_tuple_to_bit_rep_of_Vrep(bi_pyr, Integer(6))
>>> C = CombinatorialPolyhedron((facets, Vrep)); C
A 3-dimensional combinatorial polyhedron with 8 facets
>>> C.f_vector()
(1, 6, 12, 8, 1)
```

Specifying that a polyhedron is unbounded is important. The following with a polyhedron works fine:

```
sage: P = Polyhedron(ieqs=[[1,-1,0],[1,1,0]])
sage: C = CombinatorialPolyhedron(P)  # this works fine
sage: C
A 2-dimensional combinatorial polyhedron with 2 facets
```

```
>>> from sage.all import *
>>> P = Polyhedron(ieqs=[[Integer(1),-Integer(1),Integer(0)],[Integer(1),
↪Integer(1),Integer(0)]])
>>> C = CombinatorialPolyhedron(P)  # this works fine
>>> C
A 2-dimensional combinatorial polyhedron with 2 facets
```

The following is incorrect, as `unbounded` is implicitly set to `False`:

```
sage: data = P.incidence_matrix()
sage: vert = P.Vrepresentation()
sage: C = CombinatorialPolyhedron(data, Vrep=vert)
sage: C
A 2-dimensional combinatorial polyhedron with 2 facets
sage: C.f_vector()
Traceback (most recent call last):
...
ValueError: not all vertices are intersections of facets
sage: C.vertices()
(A line in the direction (0, 1), A vertex at (1, 0), A vertex at (-1, 0))
```

```
>>> from sage.all import *
>>> data = P.incidence_matrix()
>>> vert = P.Vrepresentation()
>>> C = CombinatorialPolyhedron(data, Vrep=vert)
>>> C
A 2-dimensional combinatorial polyhedron with 2 facets
>>> C.f_vector()
Traceback (most recent call last):
...
ValueError: not all vertices are intersections of facets
>>> C.vertices()
(A line in the direction (0, 1), A vertex at (1, 0), A vertex at (-1, 0))
```

The correct usage is:

```
sage: far_face = [i for i in range(3) if not P.Vrepresentation()[i].is_vertex()]
sage: C = CombinatorialPolyhedron(data, Vrep=vert, unbounded=True, far_face=far_
↪face)
sage: C
A 2-dimensional combinatorial polyhedron with 2 facets
sage: C.f_vector()
(1, 0, 2, 1)
sage: C.vertices()
()
```

```
>>> from sage.all import *
>>> far_face = [i for i in range(Integer(3)) if not P.Vrepresentation()[i].is_
↪vertex()]
>>> C = CombinatorialPolyhedron(data, Vrep=vert, unbounded=True, far_face=far_
↪face)
>>> C
A 2-dimensional combinatorial polyhedron with 2 facets
>>> C.f_vector()
(1, 0, 2, 1)
>>> C.vertices()
()
```

Hrepresentation()

Return a list of names of facets and possibly some equations.

EXAMPLES:

```
sage: P = polytopes.permutahedron(3)
sage: C = CombinatorialPolyhedron(P)
sage: C.Hrepresentation()
(An inequality (1, 1, 0) x - 3 >= 0,
 An inequality (-1, -1, 0) x + 5 >= 0,
 An inequality (0, 1, 0) x - 1 >= 0,
 An inequality (-1, 0, 0) x + 3 >= 0,
 An inequality (1, 0, 0) x - 1 >= 0,
 An inequality (0, -1, 0) x + 3 >= 0,
 An equation (1, 1, 1) x - 6 == 0)

sage: points = [(1,0,0), (0,1,0), (0,0,1),
....: (-1,0,0), (0,-1,0), (0,0,-1)]
sage: L = LatticePolytope(points)
sage: C = CombinatorialPolyhedron(L)
sage: C.Hrepresentation()
(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))

sage: M = Cone([(1,0), (0,1)])
```

(continues on next page)

(continued from previous page)

```
sage: CombinatorialPolyhedron(M).Hrepresentation()
(M(0, 1), M(1, 0))
```

```
>>> from sage.all import *
>>> P = polytopes.permutahedron(Integer(3))
>>> C = CombinatorialPolyhedron(P)
>>> C.Hrepresentation()
(An inequality (1, 1, 0) x - 3 >= 0,
 An inequality (-1, -1, 0) x + 5 >= 0,
 An inequality (0, 1, 0) x - 1 >= 0,
 An inequality (-1, 0, 0) x + 3 >= 0,
 An inequality (1, 0, 0) x - 1 >= 0,
 An inequality (0, -1, 0) x + 3 >= 0,
 An equation (1, 1, 1) x - 6 == 0)

>>> points = [(Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0)), (Integer(0),Integer(0),Integer(1)),
... (-Integer(1),Integer(0),Integer(0)), (Integer(0),-Integer(1),Integer(0)),␣
↪(Integer(0),Integer(0),-Integer(1))]
>>> L = LatticePolytope(points)
>>> C = CombinatorialPolyhedron(L)
>>> C.Hrepresentation()
(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))

>>> M = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> CombinatorialPolyhedron(M).Hrepresentation()
(M(0, 1), M(1, 0))
```

Vrepresentation()

Return a list of names of `[vertices, rays, lines]`.

EXAMPLES:

```
sage: P = Polyhedron(rays=[[1,0,0], [0,1,0], \
....:                      [0,0,1],[0,0,-1]])
sage: C = CombinatorialPolyhedron(P)
sage: C.Vrepresentation()
(A line in the direction (0, 0, 1),
 A ray in the direction (1, 0, 0),
 A vertex at (0, 0, 0),
 A ray in the direction (0, 1, 0))

sage: points = [(1,0,0), (0,1,0), (0,0,1),
....: (-1,0,0), (0,-1,0), (0,0,-1)]
sage: L = LatticePolytope(points)
```

(continues on next page)

(continued from previous page)

```
sage: C = CombinatorialPolyhedron(L)
sage: C.Vrepresentation()
(M(1, 0, 0), M(0, 1, 0), M(0, 0, 1), M(-1, 0, 0), M(0, -1, 0), M(0, 0, -1))

sage: M = Cone([(1,0), (0,1)])
sage: CombinatorialPolyhedron(M).Vrepresentation()
(N(1, 0), N(0, 1), N(0, 0))
```

```
>>> from sage.all import *
>>> P = Polyhedron(rays=[[Integer(1),Integer(0),Integer(0)], [Integer(0),
↪Integer(1),Integer(0)],                              [Integer(0),Integer(0),
↪Integer(1)],[Integer(0),Integer(0),-Integer(1)]])
>>> C = CombinatorialPolyhedron(P)
>>> C.Vrepresentation()
(A line in the direction (0, 0, 1),
 A ray in the direction (1, 0, 0),
 A vertex at (0, 0, 0),
 A ray in the direction (0, 1, 0))

>>> points = [(Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0)), (Integer(0),Integer(0),Integer(1)),
... (-Integer(1),Integer(0),Integer(0)), (Integer(0),-Integer(1),Integer(0)),
↪(Integer(0),Integer(0),-Integer(1))]
>>> L = LatticePolytope(points)
>>> C = CombinatorialPolyhedron(L)
>>> C.Vrepresentation()
(M(1, 0, 0), M(0, 1, 0), M(0, 0, 1), M(-1, 0, 0), M(0, -1, 0), M(0, 0, -1))

>>> M = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> CombinatorialPolyhedron(M).Vrepresentation()
(N(1, 0), N(0, 1), N(0, 0))
```

a_maximal_chain(*Vindex=None*, *Hindex=None*)

Return a maximal chain of the face lattice in increasing order without empty face and whole polyhedron/maximal face.

INPUT:

- `Vindex` – integer (default: `None`); prescribe the index of the vertex in the chain
- `Hindex` – integer (default: `None`); prescribe the index of the facet in the chain

Each face is given as `CombinatorialFace`.

EXAMPLES:

```
sage: P = polytopes.cross_polytope(4)
sage: C = P.combinatorial_polyhedron()
sage: chain = C.a_maximal_chain(); chain
[A 0-dimensional face of a 4-dimensional combinatorial polyhedron,
 A 1-dimensional face of a 4-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 4-dimensional combinatorial polyhedron,
 A 3-dimensional face of a 4-dimensional combinatorial polyhedron]
sage: [face.ambient_V_indices() for face in chain]
```

(continues on next page)

(continued from previous page)

```
[(7,), (6, 7), (5, 6, 7), (4, 5, 6, 7)]

sage: P = polytopes.hypercube(4)
sage: C = P.combinatorial_polyhedron()
sage: chain = C.a_maximal_chain(); chain
[A 0-dimensional face of a 4-dimensional combinatorial polyhedron,
 A 1-dimensional face of a 4-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 4-dimensional combinatorial polyhedron,
 A 3-dimensional face of a 4-dimensional combinatorial polyhedron]
sage: [face.ambient_V_indices() for face in chain]
[(15,), (6, 15), (5, 6, 14, 15), (0, 5, 6, 7, 8, 9, 14, 15)]

sage: # needs sage.combinat
sage: P = polytopes.permutahedron(4)
sage: C = P.combinatorial_polyhedron()
sage: chain = C.a_maximal_chain(); chain
[A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 1-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron]
sage: [face.ambient_V_indices() for face in chain]
[(16,), (15, 16), (8, 9, 14, 15, 16, 17)]

sage: P = Polyhedron(rays=[[1,0]], lines=[[0,1]])
sage: C = P.combinatorial_polyhedron()
sage: chain = C.a_maximal_chain()
sage: [face.ambient_V_indices() for face in chain]
[(0, 1)]

sage: P = Polyhedron(rays=[[1,0,0],[0,0,1]], lines=[[0,1,0]])
sage: C = P.combinatorial_polyhedron()
sage: chain = C.a_maximal_chain()
sage: [face.ambient_V_indices() for face in chain]
[(0, 1), (0, 1, 3)]

sage: P = Polyhedron(rays=[[1,0,0]], lines=[[0,1,0],[0,0,1]])
sage: C = P.combinatorial_polyhedron()
sage: chain = C.a_maximal_chain()
sage: [face.ambient_V_indices() for face in chain]
[(0, 1, 2)]
```

```
>>> from sage.all import *
>>> P = polytopes.cross_polytope(Integer(4))
>>> C = P.combinatorial_polyhedron()
>>> chain = C.a_maximal_chain(); chain
[A 0-dimensional face of a 4-dimensional combinatorial polyhedron,
 A 1-dimensional face of a 4-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 4-dimensional combinatorial polyhedron,
 A 3-dimensional face of a 4-dimensional combinatorial polyhedron]
>>> [face.ambient_V_indices() for face in chain]
[(7,), (6, 7), (5, 6, 7), (4, 5, 6, 7)]

>>> P = polytopes.hypercube(Integer(4))
```

(continues on next page)

(continued from previous page)

```
>>> C = P.combinatorial_polyhedron()
>>> chain = C.a_maximal_chain(); chain
[A 0-dimensional face of a 4-dimensional combinatorial polyhedron,
 A 1-dimensional face of a 4-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 4-dimensional combinatorial polyhedron,
 A 3-dimensional face of a 4-dimensional combinatorial polyhedron]
>>> [face.ambient_V_indices() for face in chain]
[(15,), (6, 15), (5, 6, 14, 15), (0, 5, 6, 7, 8, 9, 14, 15)]

>>> # needs sage.combinat
>>> P = polytopes.permutahedron(Integer(4))
>>> C = P.combinatorial_polyhedron()
>>> chain = C.a_maximal_chain(); chain
[A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 1-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron]
>>> [face.ambient_V_indices() for face in chain]
[(16,), (15, 16), (8, 9, 14, 15, 16, 17)]

>>> P = Polyhedron(rays=[[Integer(1),Integer(0)]], lines=[[Integer(0),
↪Integer(1)]])
>>> C = P.combinatorial_polyhedron()
>>> chain = C.a_maximal_chain()
>>> [face.ambient_V_indices() for face in chain]
[(0, 1)]

>>> P = Polyhedron(rays=[[Integer(1),Integer(0),Integer(0)],[Integer(0),
↪Integer(0),Integer(1)]], lines=[[Integer(0),Integer(1),Integer(0)]])
>>> C = P.combinatorial_polyhedron()
>>> chain = C.a_maximal_chain()
>>> [face.ambient_V_indices() for face in chain]
[(0, 1), (0, 1, 3)]

>>> P = Polyhedron(rays=[[Integer(1),Integer(0),Integer(0)]],
↪lines=[[Integer(0),Integer(1),Integer(0)],[Integer(0),Integer(0),
↪Integer(1)]])
>>> C = P.combinatorial_polyhedron()
>>> chain = C.a_maximal_chain()
>>> [face.ambient_V_indices() for face in chain]
[(0, 1, 2)]
```

Specify an index for the vertex of the chain:

```
sage: P = polytopes.cube()
sage: C = P.combinatorial_polyhedron()
sage: [face.ambient_V_indices() for face in C.a_maximal_chain()]
[(5,), (0, 5), (0, 3, 4, 5)]
sage: [face.ambient_V_indices() for face in C.a_maximal_chain(Vindex=2)]
[(2,), (2, 7), (2, 3, 4, 7)]
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
```

(continues on next page)

(continued from previous page)

```
>>> C = P.combinatorial_polyhedron()
>>> [face.ambient_V_indices() for face in C.a_maximal_chain()]
[(5,), (0, 5), (0, 3, 4, 5)]
>>> [face.ambient_V_indices() for face in C.a_maximal_
↪chain(Vindex=Integer(2))]
[(2,), (2, 7), (2, 3, 4, 7)]
```

Specify an index for the facet of the chain:

```
sage: [face.ambient_H_indices() for face in C.a_maximal_chain()]
[(3, 4, 5), (4, 5), (5,)]
sage: [face.ambient_H_indices() for face in C.a_maximal_chain(Hindex=3)]
[(3, 4, 5), (3, 4), (3,)]
sage: [face.ambient_H_indices() for face in C.a_maximal_chain(Hindex=2)]
[(2, 3, 5), (2, 3), (2,)]
```

```
>>> from sage.all import *
>>> [face.ambient_H_indices() for face in C.a_maximal_chain()]
[(3, 4, 5), (4, 5), (5,)]
>>> [face.ambient_H_indices() for face in C.a_maximal_
↪chain(Hindex=Integer(3))]
[(3, 4, 5), (3, 4), (3,)]
>>> [face.ambient_H_indices() for face in C.a_maximal_
↪chain(Hindex=Integer(2))]
[(2, 3, 5), (2, 3), (2,)]
```

If the specified vertex is not contained in the specified facet an error is raised:

```
sage: C.a_maximal_chain(Vindex=0, Hindex=3)
Traceback (most recent call last):
...
ValueError: the given Vindex is not compatible with the given Hindex
```

```
>>> from sage.all import *
>>> C.a_maximal_chain(Vindex=Integer(0), Hindex=Integer(3))
Traceback (most recent call last):
...
ValueError: the given Vindex is not compatible with the given Hindex
```

An error is raised, if the specified index does not correspond to a facet:

```
sage: C.a_maximal_chain(Hindex=40)
Traceback (most recent call last):
...
ValueError: the given Hindex does not correspond to a facet
```

```
>>> from sage.all import *
>>> C.a_maximal_chain(Hindex=Integer(40))
Traceback (most recent call last):
...
ValueError: the given Hindex does not correspond to a facet
```

An error is raised, if the specified index does not correspond to a vertex:

```
sage: C.a_maximal_chain(Vindex=40)
Traceback (most recent call last):
...
ValueError: the given Vindex does not correspond to a vertex
```

```
>>> from sage.all import *
>>> C.a_maximal_chain(Vindex=Integer(40))
Traceback (most recent call last):
...
ValueError: the given Vindex does not correspond to a vertex
```

```
sage: P = Polyhedron(rays=[[1,0,0],[0,0,1]], lines=[[0,1,0]])
sage: C = P.combinatorial_polyhedron()
sage: C.a_maximal_chain(Vindex=0)
Traceback (most recent call last):
...
ValueError: the given Vindex does not correspond to a vertex
```

```
>>> from sage.all import *
>>> P = Polyhedron(rays=[[Integer(1),Integer(0),Integer(0)],[Integer(0),
↪Integer(0),Integer(1)]], lines=[[Integer(0),Integer(1),Integer(0)]])
>>> C = P.combinatorial_polyhedron()
>>> C.a_maximal_chain(Vindex=Integer(0))
Traceback (most recent call last):
...
ValueError: the given Vindex does not correspond to a vertex
```

```
sage: P = Polyhedron(rays=[[1,0,0],[0,0,1]])
sage: C = P.combinatorial_polyhedron()
sage: C.a_maximal_chain(Vindex=0)
[A 0-dimensional face of a 2-dimensional combinatorial polyhedron,
A 1-dimensional face of a 2-dimensional combinatorial polyhedron]
sage: C.a_maximal_chain(Vindex=1)
Traceback (most recent call last):
...
ValueError: the given Vindex does not correspond to a vertex
```

```
>>> from sage.all import *
>>> P = Polyhedron(rays=[[Integer(1),Integer(0),Integer(0)],[Integer(0),
↪Integer(0),Integer(1)]])
>>> C = P.combinatorial_polyhedron()
>>> C.a_maximal_chain(Vindex=Integer(0))
[A 0-dimensional face of a 2-dimensional combinatorial polyhedron,
A 1-dimensional face of a 2-dimensional combinatorial polyhedron]
>>> C.a_maximal_chain(Vindex=Integer(1))
Traceback (most recent call last):
...
ValueError: the given Vindex does not correspond to a vertex
```

choose_algorithm_to_compute_edges_or_ridges(*edges_or_ridges*)

Use some heuristics to pick primal or dual algorithm for computation of edges resp. ridges.

We estimate how long it takes to compute a face using the primal and the dual algorithm. This may differ significantly, so that e.g. visiting all faces with the primal algorithm is faster than using the dual algorithm to just visit vertices and edges.

We guess the number of edges and ridges and do a wild estimate on the total number of faces.

INPUT:

- `edges_or_ridges` – string; one of: * `'edges'` * `'ridges'`

OUTPUT: either `'primal'` or `'dual'`

EXAMPLES:

```
sage: C = polytopes.permutahedron(5).combinatorial_polyhedron()
sage: C.choose_algorithm_to_compute_edges_or_ridges("edges")
'primal'
sage: C.choose_algorithm_to_compute_edges_or_ridges("ridges")
'primal'
```

```
>>> from sage.all import *
>>> C = polytopes.permutahedron(Integer(5)).combinatorial_polyhedron()
>>> C.choose_algorithm_to_compute_edges_or_ridges("edges")
'primal'
>>> C.choose_algorithm_to_compute_edges_or_ridges("ridges")
'primal'
```

```
sage: C = polytopes.cross_polytope(5).combinatorial_polyhedron()
sage: C.choose_algorithm_to_compute_edges_or_ridges("edges")
'dual'
sage: C.choose_algorithm_to_compute_edges_or_ridges("ridges")
'dual'
```

```
>>> from sage.all import *
>>> C = polytopes.cross_polytope(Integer(5)).combinatorial_polyhedron()
>>> C.choose_algorithm_to_compute_edges_or_ridges("edges")
'dual'
>>> C.choose_algorithm_to_compute_edges_or_ridges("ridges")
'dual'
```

```
sage: C = polytopes.Birkhoff_polytope(5).combinatorial_polyhedron()
sage: C.choose_algorithm_to_compute_edges_or_ridges("edges")
'dual'
sage: C.choose_algorithm_to_compute_edges_or_ridges("ridges")
'primal'
sage: C.choose_algorithm_to_compute_edges_or_ridges("something_else")
Traceback (most recent call last):
...
ValueError: unknown computation goal something_else
```

```
>>> from sage.all import *
>>> C = polytopes.Birkhoff_polytope(Integer(5)).combinatorial_polyhedron()
>>> C.choose_algorithm_to_compute_edges_or_ridges("edges")
'dual'
```

(continues on next page)

(continued from previous page)

```
>>> C.choose_algorithm_to_compute_edges_or_ridges("ridges")
'primal'
>>> C.choose_algorithm_to_compute_edges_or_ridges("something_else")
Traceback (most recent call last):
...
ValueError: unknown computation goal something_else
```

dim()

Return the dimension of the polyhedron.

EXAMPLES:

```
sage: C = CombinatorialPolyhedron([(1,2,3), (1,2,4),
....:                               (1,3,4), (2,3,4)])
sage: C.dimension()
3

sage: P = Polyhedron(rays=[[1,0,0],[0,1,0],[0,0,1],[0,0,-1]])
sage: CombinatorialPolyhedron(P).dimension()
3
```

```
>>> from sage.all import *
>>> C = CombinatorialPolyhedron([(Integer(1),Integer(2),Integer(3)),␣
↪(Integer(1),Integer(2),Integer(4)),
...                             (Integer(1),Integer(3),Integer(4)),␣
↪(Integer(2),Integer(3),Integer(4))])
>>> C.dimension()
3

>>> P = Polyhedron(rays=[[Integer(1),Integer(0),Integer(0)],[Integer(0),
↪Integer(1),Integer(0)],[Integer(0),Integer(0),Integer(1)],[Integer(0),
↪Integer(0),-Integer(1)]])
>>> CombinatorialPolyhedron(P).dimension()
3
```

`dim` is an alias:

```
sage: CombinatorialPolyhedron(P).dim()
3
```

```
>>> from sage.all import *
>>> CombinatorialPolyhedron(P).dim()
3
```

dimension()

Return the dimension of the polyhedron.

EXAMPLES:

```
sage: C = CombinatorialPolyhedron([(1,2,3), (1,2,4),
....:                               (1,3,4), (2,3,4)])
sage: C.dimension()
3
```

(continues on next page)

(continued from previous page)

```
sage: P = Polyhedron(rays=[[1,0,0],[0,1,0],[0,0,1],[0,0,-1]])
sage: CombinatorialPolyhedron(P).dimension()
3
```

```
>>> from sage.all import *
>>> C = CombinatorialPolyhedron([(Integer(1),Integer(2),Integer(3)),␣
↪(Integer(1),Integer(2),Integer(4)),
...                                 (Integer(1),Integer(3),Integer(4)),␣
↪(Integer(2),Integer(3),Integer(4))])
>>> C.dimension()
3

>>> P = Polyhedron(rays=[[Integer(1),Integer(0),Integer(0)],[Integer(0),
↪Integer(1),Integer(0)],[Integer(0),Integer(0),Integer(1)],[Integer(0),
↪Integer(0),-Integer(1)]])
>>> CombinatorialPolyhedron(P).dimension()
3
```

`dim` is an alias:

```
sage: CombinatorialPolyhedron(P).dim()
3
```

```
>>> from sage.all import *
>>> CombinatorialPolyhedron(P).dim()
3
```

dual()

Return the dual/polar of `self`.

Only defined for bounded polyhedra.

See also

`polar()`.

EXAMPLES:

```
sage: P = polytopes.cube()
sage: C = P.combinatorial_polyhedron()
sage: D = C.dual()
sage: D.f_vector()
(1, 6, 12, 8, 1)
sage: D1 = P.polar().combinatorial_polyhedron()
sage: D1.face_lattice().is_isomorphic(D.face_lattice())                    #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
```

(continues on next page)

(continued from previous page)

```
>>> C = P.combinatorial_polyhedron()
>>> D = C.dual()
>>> D.f_vector()
(1, 6, 12, 8, 1)
>>> D1 = P.polar().combinatorial_polyhedron()
>>> D1.face_lattice().is_isomorphic(D.face_lattice())                    #␣
↪needs sage.combinat
True
```

Polar is an alias to be consistent with `Polyhedron_base`:

```
sage: C.polar().f_vector()
(1, 6, 12, 8, 1)
```

```
>>> from sage.all import *
>>> C.polar().f_vector()
(1, 6, 12, 8, 1)
```

For unbounded polyhedra, an error is raised:

```
sage: C = CombinatorialPolyhedron([[0,1], [0,2]], far_face=[1,2],␣
↪unbounded=True)
sage: C.dual()
Traceback (most recent call last):
...
ValueError: self must be bounded
```

```
>>> from sage.all import *
>>> C = CombinatorialPolyhedron([[Integer(0),Integer(1)], [Integer(0),
↪Integer(2)]], far_face=[Integer(1),Integer(2)], unbounded=True)
>>> C.dual()
Traceback (most recent call last):
...
ValueError: self must be bounded
```

edges(*names=True*, *algorithm=None*)

Return the edges of the polyhedron, i.e. the rank 1 faces.

INPUT:

- `names` – boolean (default: `True`); if `False`, then the Vrepresentatives in the edges are given by their indices in the Vrepresentation
- `algorithm` – string (optional); specify whether the face generator starts with facets or vertices: * `'primal'` – start with the facets * `'dual'` – start with the vertices * `None` – choose automatically

Note

To compute edges and f_vector, first compute the edges. This might be faster.

EXAMPLES:

```
sage: P = polytopes.cyclic_polytope(3,5)
sage: C = CombinatorialPolyhedron(P)
sage: C.edges()
((A vertex at (3, 9, 27), A vertex at (4, 16, 64)),
 (A vertex at (2, 4, 8), A vertex at (4, 16, 64)),
 (A vertex at (1, 1, 1), A vertex at (4, 16, 64)),
 (A vertex at (0, 0, 0), A vertex at (4, 16, 64)),
 (A vertex at (2, 4, 8), A vertex at (3, 9, 27)),
 (A vertex at (0, 0, 0), A vertex at (3, 9, 27)),
 (A vertex at (1, 1, 1), A vertex at (2, 4, 8)),
 (A vertex at (0, 0, 0), A vertex at (2, 4, 8)),
 (A vertex at (0, 0, 0), A vertex at (1, 1, 1)))

sage: C.edges(names=False)
((3, 4), (2, 4), (1, 4), (0, 4), (2, 3), (0, 3), (1, 2), (0, 2), (0, 1))

sage: P = Polyhedron(rays=[[-1,0],[1,0]])
sage: C = CombinatorialPolyhedron(P)
sage: C.edges()
((A line in the direction (1, 0), A vertex at (0, 0)),)

sage: P = Polyhedron(vertices=[[0,0],[1,0]])
sage: C = CombinatorialPolyhedron(P)
sage: C.edges()
((A vertex at (0, 0), A vertex at (1, 0)),)

sage: from itertools import combinations
sage: N = combinations(['a','b','c','d','e'], 4)
sage: C = CombinatorialPolyhedron(N)
sage: C.edges()
(('d', 'e'),
 ('c', 'e'),
 ('b', 'e'),
 ('a', 'e'),
 ('c', 'd'),
 ('b', 'd'),
 ('a', 'd'),
 ('b', 'c'),
 ('a', 'c'),
 ('a', 'b'))
```

```
>>> from sage.all import *
>>> P = polytopes.cyclic_polytope(Integer(3),Integer(5))
>>> C = CombinatorialPolyhedron(P)
>>> C.edges()
((A vertex at (3, 9, 27), A vertex at (4, 16, 64)),
 (A vertex at (2, 4, 8), A vertex at (4, 16, 64)),
 (A vertex at (1, 1, 1), A vertex at (4, 16, 64)),
 (A vertex at (0, 0, 0), A vertex at (4, 16, 64)),
 (A vertex at (2, 4, 8), A vertex at (3, 9, 27)),
 (A vertex at (0, 0, 0), A vertex at (3, 9, 27)),
 (A vertex at (1, 1, 1), A vertex at (2, 4, 8)),
```

(continues on next page)

(continued from previous page)

```
 (A vertex at (0, 0, 0), A vertex at (2, 4, 8)),
 (A vertex at (0, 0, 0), A vertex at (1, 1, 1)))

>>> C.edges(names=False)
((3, 4), (2, 4), (1, 4), (0, 4), (2, 3), (0, 3), (1, 2), (0, 2), (0, 1))

>>> P = Polyhedron(rays=[[-Integer(1),Integer(0)],[Integer(1),Integer(0)]])
>>> C = CombinatorialPolyhedron(P)
>>> C.edges()
((A line in the direction (1, 0), A vertex at (0, 0)),)

>>> P = Polyhedron(vertices=[[Integer(0),Integer(0)],[Integer(1),Integer(0)]])
>>> C = CombinatorialPolyhedron(P)
>>> C.edges()
((A vertex at (0, 0), A vertex at (1, 0)),)

>>> from itertools import combinations
>>> N = combinations(['a','b','c','d','e'], Integer(4))
>>> C = CombinatorialPolyhedron(N)
>>> C.edges()
(('d', 'e'),
 ('c', 'e'),
 ('b', 'e'),
 ('a', 'e'),
 ('c', 'd'),
 ('b', 'd'),
 ('a', 'd'),
 ('b', 'c'),
 ('a', 'c'),
 ('a', 'b'))
```

f_vector(*num_threads=None*, *parallelization_depth=None*, *algorithm=None*)

Compute the `f_vector` of the polyhedron.

The `f_vector` contains the number of faces of dimension k for each k in `range(-1, self.dimension() + 1)`.

INPUT:

- `num_threads` – integer (optional); specify the number of threads
- `parallelization_depth` – integer (optional); specify how deep in the lattice the parallelization is done
- `algorithm` – string (optional); specify whether the face generator starts with facets or vertices:
  - `'primal'` – start with the facets
  - `'dual'` – start with the vertices
  - `None` – choose automatically

Note

To obtain edges and/or ridges as well, first do so. This might already compute the `f_vector`.

EXAMPLES:

```
sage: P = polytopes.permutahedron(5)
sage: C = CombinatorialPolyhedron(P)
sage: C.f_vector()
(1, 120, 240, 150, 30, 1)

sage: P = polytopes.cyclic_polytope(6,10)
sage: C = CombinatorialPolyhedron(P)
sage: C.f_vector()
(1, 10, 45, 120, 185, 150, 50, 1)
```

```
>>> from sage.all import *
>>> P = polytopes.permutahedron(Integer(5))
>>> C = CombinatorialPolyhedron(P)
>>> C.f_vector()
(1, 120, 240, 150, 30, 1)

>>> P = polytopes.cyclic_polytope(Integer(6),Integer(10))
>>> C = CombinatorialPolyhedron(P)
>>> C.f_vector()
(1, 10, 45, 120, 185, 150, 50, 1)
```

Using two threads:

```
sage: P = polytopes.permutahedron(5)
sage: C = CombinatorialPolyhedron(P)
sage: C.f_vector(num_threads=2)
(1, 120, 240, 150, 30, 1)
```

```
>>> from sage.all import *
>>> P = polytopes.permutahedron(Integer(5))
>>> C = CombinatorialPolyhedron(P)
>>> C.f_vector(num_threads=Integer(2))
(1, 120, 240, 150, 30, 1)
```

face_by_face_lattice_index(*index*)

Return the element of `CombinatorialPolyhedron.face_lattice()` with corresponding index.

The element will be returned as `CombinatorialFace`.

EXAMPLES:

```
sage: # needs sage.combinat
sage: P = polytopes.cube()
sage: C = CombinatorialPolyhedron(P)
sage: F = C.face_lattice()
sage: F
Finite lattice containing 28 elements
sage: G = F.relabel(C.face_by_face_lattice_index)
sage: G.level_sets()[0]
[A -1-dimensional face of a 3-dimensional combinatorial polyhedron]
sage: G.level_sets()[3]
[A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
```

(continues on next page)

(continued from previous page)

```
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron]

sage: P = Polyhedron(rays=[[0,1], [1,0]])
sage: C = CombinatorialPolyhedron(P)
sage: F = C.face_lattice()                                              #␣
↪needs sage.combinat
sage: G = F.relabel(C.face_by_face_lattice_index)                       #␣
↪needs sage.combinat
sage: G._elements                                                       #␣
↪needs sage.combinat
(A -1-dimensional face of a 2-dimensional combinatorial polyhedron,
  A 0-dimensional face of a 2-dimensional combinatorial polyhedron,
  A 1-dimensional face of a 2-dimensional combinatorial polyhedron,
  A 1-dimensional face of a 2-dimensional combinatorial polyhedron,
  A 2-dimensional face of a 2-dimensional combinatorial polyhedron)

sage: def f(i): return C.face_by_face_lattice_index(i).ambient_V_indices()
sage: G = F.relabel(f)                                                  #␣
↪needs sage.combinat
sage: G._elements                                                       #␣
↪needs sage.combinat
((), (0,), (0, 1), (0, 2), (0, 1, 2))
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> P = polytopes.cube()
>>> C = CombinatorialPolyhedron(P)
>>> F = C.face_lattice()
>>> F
Finite lattice containing 28 elements
>>> G = F.relabel(C.face_by_face_lattice_index)
>>> G.level_sets()[Integer(0)]
[A -1-dimensional face of a 3-dimensional combinatorial polyhedron]
>>> G.level_sets()[Integer(3)]
[A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron]

>>> P = Polyhedron(rays=[[Integer(0),Integer(1)], [Integer(1),Integer(0)]])
>>> C = CombinatorialPolyhedron(P)
>>> F = C.face_lattice()                                              #␣
↪needs sage.combinat
>>> G = F.relabel(C.face_by_face_lattice_index)                       #␣
↪needs sage.combinat
>>> G._elements                                                       #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.combinat
(A -1-dimensional face of a 2-dimensional combinatorial polyhedron,
  A 0-dimensional face of a 2-dimensional combinatorial polyhedron,
  A 1-dimensional face of a 2-dimensional combinatorial polyhedron,
  A 1-dimensional face of a 2-dimensional combinatorial polyhedron,
  A 2-dimensional face of a 2-dimensional combinatorial polyhedron)

>>> def f(i): return C.face_by_face_lattice_index(i).ambient_V_indices()
>>> G = F.relabel(f)                                                    #␣
↪needs sage.combinat
>>> G._elements                                                         #␣
↪needs sage.combinat
((), (0,), (0, 1), (0, 2), (0, 1, 2))
```

face_generator(*dimension=None*, *algorithm=None*)

Iterator over all proper faces of specified dimension.

INPUT:

- `dimension` – if specified, then iterate over only this dimension
- `algorithm` – string (optional); specify whether the face generator starts with facets or vertices:
  - `'primal'` – start with the facets
  - `'dual'` – start with the vertices
  - `None` – choose automatically

OUTPUT: *FaceIterator*

> **Note**
>
> *FaceIterator* can ignore subfaces or supfaces of the current face.

EXAMPLES:

```
sage: # needs sage.combinat
sage: P = polytopes.permutahedron(5)
sage: C = CombinatorialPolyhedron(P)
sage: it = C.face_generator(dimension=2)
sage: face = next(it); face
A 2-dimensional face of a 4-dimensional combinatorial polyhedron
sage: face.ambient_Vrepresentation()
(A vertex at (1, 3, 2, 5, 4),
 A vertex at (2, 3, 1, 5, 4),
 A vertex at (3, 1, 2, 5, 4),
 A vertex at (3, 2, 1, 5, 4),
 A vertex at (2, 1, 3, 5, 4),
 A vertex at (1, 2, 3, 5, 4))
sage: face = next(it); face
A 2-dimensional face of a 4-dimensional combinatorial polyhedron
sage: face.ambient_Vrepresentation()
(A vertex at (2, 1, 4, 5, 3),
```

(continues on next page)

(continued from previous page)

```
 A vertex at (3, 2, 4, 5, 1),
 A vertex at (3, 1, 4, 5, 2),
 A vertex at (1, 3, 4, 5, 2),
 A vertex at (1, 2, 4, 5, 3),
 A vertex at (2, 3, 4, 5, 1))
sage: face.ambient_Hrepresentation()
(An inequality (0, 0, -1, -1, 0) x + 9 >= 0,
 An inequality (0, 0, 0, -1, 0) x + 5 >= 0,
 An equation (1, 1, 1, 1, 1) x - 15 == 0)
sage: face.ambient_H_indices()
(25, 29, 30)
sage: face = next(it); face
A 2-dimensional face of a 4-dimensional combinatorial polyhedron
sage: face.ambient_H_indices()
(24, 29, 30)
sage: face.ambient_V_indices()
(32, 89, 90, 94)

sage: C = CombinatorialPolyhedron([[0,1,2],[0,1,3],[0,2,3],[1,2,3]])
sage: it = C.face_generator()
sage: for face in it: face.ambient_Vrepresentation()
(1, 2, 3)
(0, 2, 3)
(0, 1, 3)
(0, 1, 2)
(2, 3)
(1, 3)
(1, 2)
(3,)
(2,)
(1,)
(0, 3)
(0, 2)
(0,)
(0, 1)

sage: P = Polyhedron(rays=[[1,0],[0,1]], vertices=[[1,0],[0,1]])
sage: C = CombinatorialPolyhedron(P)
sage: it = C.face_generator(1)
sage: for face in it: face.ambient_Vrepresentation()
(A vertex at (0, 1), A vertex at (1, 0))
(A ray in the direction (1, 0), A vertex at (1, 0))
(A ray in the direction (0, 1), A vertex at (0, 1))
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> P = polytopes.permutahedron(Integer(5))
>>> C = CombinatorialPolyhedron(P)
>>> it = C.face_generator(dimension=Integer(2))
>>> face = next(it); face
A 2-dimensional face of a 4-dimensional combinatorial polyhedron
>>> face.ambient_Vrepresentation()
```

(continues on next page)

(continued from previous page)

```
(A vertex at (1, 3, 2, 5, 4),
 A vertex at (2, 3, 1, 5, 4),
 A vertex at (3, 1, 2, 5, 4),
 A vertex at (3, 2, 1, 5, 4),
 A vertex at (2, 1, 3, 5, 4),
 A vertex at (1, 2, 3, 5, 4))
>>> face = next(it); face
A 2-dimensional face of a 4-dimensional combinatorial polyhedron
>>> face.ambient_Vrepresentation()
(A vertex at (2, 1, 4, 5, 3),
 A vertex at (3, 2, 4, 5, 1),
 A vertex at (3, 1, 4, 5, 2),
 A vertex at (1, 3, 4, 5, 2),
 A vertex at (1, 2, 4, 5, 3),
 A vertex at (2, 3, 4, 5, 1))
>>> face.ambient_Hrepresentation()
(An inequality (0, 0, -1, -1, 0) x + 9 >= 0,
 An inequality (0, 0, 0, -1, 0) x + 5 >= 0,
 An equation (1, 1, 1, 1, 1) x - 15 == 0)
>>> face.ambient_H_indices()
(25, 29, 30)
>>> face = next(it); face
A 2-dimensional face of a 4-dimensional combinatorial polyhedron
>>> face.ambient_H_indices()
(24, 29, 30)
>>> face.ambient_V_indices()
(32, 89, 90, 94)

>>> C = CombinatorialPolyhedron([[Integer(0),Integer(1),Integer(2)],
↪[Integer(0),Integer(1),Integer(3)],[Integer(0),Integer(2),Integer(3)],
↪[Integer(1),Integer(2),Integer(3)]])
>>> it = C.face_generator()
>>> for face in it: face.ambient_Vrepresentation()
(1, 2, 3)
(0, 2, 3)
(0, 1, 3)
(0, 1, 2)
(2, 3)
(1, 3)
(1, 2)
(3,)
(2,)
(1,)
(0, 3)
(0, 2)
(0,)
(0, 1)

>>> P = Polyhedron(rays=[[Integer(1),Integer(0)],[Integer(0),Integer(1)]],
↪vertices=[[Integer(1),Integer(0)],[Integer(0),Integer(1)]])
>>> C = CombinatorialPolyhedron(P)
>>> it = C.face_generator(Integer(1))
```

(continues on next page)

(continued from previous page)

```
>>> for face in it: face.ambient_Vrepresentation()
(A vertex at (0, 1), A vertex at (1, 0))
(A ray in the direction (1, 0), A vertex at (1, 0))
(A ray in the direction (0, 1), A vertex at (0, 1))
```

See also

FaceIterator, *CombinatorialFace*.

face_iter(*dimension=None*, *algorithm=None*)

Iterator over all proper faces of specified dimension.

INPUT:

- `dimension` – if specified, then iterate over only this dimension
- `algorithm` – string (optional); specify whether the face generator starts with facets or vertices:
  - `'primal'` – start with the facets
  - `'dual'` – start with the vertices
  - `None` – choose automatically

OUTPUT: *FaceIterator*

Note

FaceIterator can ignore subfaces or supfaces of the current face.

EXAMPLES:

```
sage: # needs sage.combinat
sage: P = polytopes.permutahedron(5)
sage: C = CombinatorialPolyhedron(P)
sage: it = C.face_generator(dimension=2)
sage: face = next(it); face
A 2-dimensional face of a 4-dimensional combinatorial polyhedron
sage: face.ambient_Vrepresentation()
(A vertex at (1, 3, 2, 5, 4),
 A vertex at (2, 3, 1, 5, 4),
 A vertex at (3, 1, 2, 5, 4),
 A vertex at (3, 2, 1, 5, 4),
 A vertex at (2, 1, 3, 5, 4),
 A vertex at (1, 2, 3, 5, 4))
sage: face = next(it); face
A 2-dimensional face of a 4-dimensional combinatorial polyhedron
sage: face.ambient_Vrepresentation()
(A vertex at (2, 1, 4, 5, 3),
 A vertex at (3, 2, 4, 5, 1),
 A vertex at (3, 1, 4, 5, 2),
 A vertex at (1, 3, 4, 5, 2),
 A vertex at (1, 2, 4, 5, 3),
```

(continues on next page)

(continued from previous page)

```
 A vertex at (2, 3, 4, 5, 1))
sage: face.ambient_Hrepresentation()
(An inequality (0, 0, -1, -1, 0) x + 9 >= 0,
 An inequality (0, 0, 0, -1, 0) x + 5 >= 0,
 An equation (1, 1, 1, 1, 1) x - 15 == 0)
sage: face.ambient_H_indices()
(25, 29, 30)
sage: face = next(it); face
A 2-dimensional face of a 4-dimensional combinatorial polyhedron
sage: face.ambient_H_indices()
(24, 29, 30)
sage: face.ambient_V_indices()
(32, 89, 90, 94)

sage: C = CombinatorialPolyhedron([[0,1,2],[0,1,3],[0,2,3],[1,2,3]])
sage: it = C.face_generator()
sage: for face in it: face.ambient_Vrepresentation()
(1, 2, 3)
(0, 2, 3)
(0, 1, 3)
(0, 1, 2)
(2, 3)
(1, 3)
(1, 2)
(3,)
(2,)
(1,)
(0, 3)
(0, 2)
(0,)
(0, 1)

sage: P = Polyhedron(rays=[[1,0],[0,1]], vertices=[[1,0],[0,1]])
sage: C = CombinatorialPolyhedron(P)
sage: it = C.face_generator(1)
sage: for face in it: face.ambient_Vrepresentation()
(A vertex at (0, 1), A vertex at (1, 0))
(A ray in the direction (1, 0), A vertex at (1, 0))
(A ray in the direction (0, 1), A vertex at (0, 1))
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> P = polytopes.permutahedron(Integer(5))
>>> C = CombinatorialPolyhedron(P)
>>> it = C.face_generator(dimension=Integer(2))
>>> face = next(it); face
A 2-dimensional face of a 4-dimensional combinatorial polyhedron
>>> face.ambient_Vrepresentation()
(A vertex at (1, 3, 2, 5, 4),
 A vertex at (2, 3, 1, 5, 4),
 A vertex at (3, 1, 2, 5, 4),
 A vertex at (3, 2, 1, 5, 4),
```

(continues on next page)

(continued from previous page)

```
 A vertex at (2, 1, 3, 5, 4),
 A vertex at (1, 2, 3, 5, 4))
>>> face = next(it); face
A 2-dimensional face of a 4-dimensional combinatorial polyhedron
>>> face.ambient_Vrepresentation()
(A vertex at (2, 1, 4, 5, 3),
 A vertex at (3, 2, 4, 5, 1),
 A vertex at (3, 1, 4, 5, 2),
 A vertex at (1, 3, 4, 5, 2),
 A vertex at (1, 2, 4, 5, 3),
 A vertex at (2, 3, 4, 5, 1))
>>> face.ambient_Hrepresentation()
(An inequality (0, 0, -1, -1, 0) x + 9 >= 0,
 An inequality (0, 0, 0, -1, 0) x + 5 >= 0,
 An equation (1, 1, 1, 1, 1) x - 15 == 0)
>>> face.ambient_H_indices()
(25, 29, 30)
>>> face = next(it); face
A 2-dimensional face of a 4-dimensional combinatorial polyhedron
>>> face.ambient_H_indices()
(24, 29, 30)
>>> face.ambient_V_indices()
(32, 89, 90, 94)

>>> C = CombinatorialPolyhedron([[Integer(0),Integer(1),Integer(2)],
↪[Integer(0),Integer(1),Integer(3)],[Integer(0),Integer(2),Integer(3)],
↪[Integer(1),Integer(2),Integer(3)]])
>>> it = C.face_generator()
>>> for face in it: face.ambient_Vrepresentation()
(1, 2, 3)
(0, 2, 3)
(0, 1, 3)
(0, 1, 2)
(2, 3)
(1, 3)
(1, 2)
(3,)
(2,)
(1,)
(0, 3)
(0, 2)
(0,)
(0, 1)

>>> P = Polyhedron(rays=[[Integer(1),Integer(0)],[Integer(0),Integer(1)]],␣
↪vertices=[[Integer(1),Integer(0)],[Integer(0),Integer(1)]])
>>> C = CombinatorialPolyhedron(P)
>>> it = C.face_generator(Integer(1))
>>> for face in it: face.ambient_Vrepresentation()
(A vertex at (0, 1), A vertex at (1, 0))
(A ray in the direction (1, 0), A vertex at (1, 0))
(A ray in the direction (0, 1), A vertex at (0, 1))
```

See also

FaceIterator, *CombinatorialFace*.

face_lattice()

Generate the face-lattice.

OUTPUT: `FiniteLatticePoset`

Note

Use *CombinatorialPolyhedron.face_by_face_lattice_index()* to get the face for each index.

Warning

The labeling of the face lattice might depend on architecture and implementation. Relabeling the face lattice with *CombinatorialPolyhedron.face_by_face_lattice_index()* or the properties obtained from this face will be platform independent.

EXAMPLES:

```
sage: P = Polyhedron(rays=[[1,0],[0,1]])
sage: C = CombinatorialPolyhedron(P)
sage: C.face_lattice()                                                    #␣
↪needs sage.combinat
Finite lattice containing 5 elements

sage: P = Polyhedron(rays=[[1,0,0], [-1,0,0], [0,-1,0], [0,1,0]])
sage: C = CombinatorialPolyhedron(P)
sage: P1 = Polyhedron(rays=[[1,0], [-1,0]])
sage: C1 = CombinatorialPolyhedron(P1)
sage: C.face_lattice().is_isomorphic(C1.face_lattice())                  #␣
↪needs sage.combinat
True

sage: P = polytopes.permutahedron(5)
sage: C = CombinatorialPolyhedron(P)
sage: C.face_lattice()                                                    #␣
↪needs sage.combinat
Finite lattice containing 542 elements
```

```
>>> from sage.all import *
>>> P = Polyhedron(rays=[[Integer(1),Integer(0)],[Integer(0),Integer(1)]])
>>> C = CombinatorialPolyhedron(P)
>>> C.face_lattice()                                                    #␣
↪needs sage.combinat
Finite lattice containing 5 elements

>>> P = Polyhedron(rays=[[Integer(1),Integer(0),Integer(0)], [-Integer(1),
```

(continues on next page)

(continued from previous page)

```
↪Integer(0),Integer(0)], [Integer(0),-Integer(1),Integer(0)], [Integer(0),
↪Integer(1),Integer(0)]])
>>> C = CombinatorialPolyhedron(P)
>>> P1 = Polyhedron(rays=[[Integer(1),Integer(0)], [-Integer(1),Integer(0)]])
>>> C1 = CombinatorialPolyhedron(P1)
>>> C.face_lattice().is_isomorphic(C1.face_lattice())                    #␣
↪needs sage.combinat
True

>>> P = polytopes.permutahedron(Integer(5))
>>> C = CombinatorialPolyhedron(P)
>>> C.face_lattice()                                                      #␣
↪needs sage.combinat
Finite lattice containing 542 elements
```

facet_adjacency_matrix(*algorithm=None*)

Return the binary matrix of facet adjacencies.

INPUT:

- `algorithm` – string (optional); specify whether the face generator starts with facets or vertices: * `'primal'` – start with the facets * `'dual'` – start with the vertices * `None` – choose automatically

See also

`vertex_adjacency_matrix()`.

EXAMPLES:

```
sage: P = polytopes.cube()
sage: C = P.combinatorial_polyhedron()
sage: C.facet_adjacency_matrix()
[0 1 1 0 1 1]
[1 0 1 1 1 0]
[1 1 0 1 0 1]
[0 1 1 0 1 1]
[1 1 0 1 0 1]
[1 0 1 1 1 0]
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> C = P.combinatorial_polyhedron()
>>> C.facet_adjacency_matrix()
[0 1 1 0 1 1]
[1 0 1 1 1 0]
[1 1 0 1 0 1]
[0 1 1 0 1 1]
[1 1 0 1 0 1]
[1 0 1 1 1 0]
```

facet_graph(*names=True, algorithm=None*)

Return the facet graph.

The facet graph of a polyhedron consists of ridges as edges and facets as vertices.

INPUT:

- `algorithm` – string (optional); specify whether the face generator starts with facets or vertices:
  - `'primal'` – start with the facets
  - `'dual'` – start with the vertices
  - `None` – choose automatically

If `names` is `False`, the `vertices` of the graph will be the indices of the facets in the Hrepresentation.

EXAMPLES:

```
sage: P = polytopes.cyclic_polytope(4,6)
sage: C = CombinatorialPolyhedron(P)
sage: C.facet_graph()                                                        #␣
↪needs sage.graphs
Graph on 9 vertices
```

```
>>> from sage.all import *
>>> P = polytopes.cyclic_polytope(Integer(4),Integer(6))
>>> C = CombinatorialPolyhedron(P)
>>> C.facet_graph()                                                        #␣
↪needs sage.graphs
Graph on 9 vertices
```

facets(*names=True*)

Return the facets as lists of `[vertices, rays, lines]`.

If `names` is `False`, then the Vrepresentatives in the facets are given by their indices in the Vrepresentation.

The facets are the maximal nontrivial faces.

EXAMPLES:

```
sage: P = polytopes.cube()
sage: C = CombinatorialPolyhedron(P)
sage: C.facets()
((A vertex at (1, -1, -1),
  A vertex at (1, 1, -1),
  A vertex at (1, 1, 1),
  A vertex at (1, -1, 1)),
 (A vertex at (1, 1, -1),
  A vertex at (1, 1, 1),
  A vertex at (-1, 1, -1),
  A vertex at (-1, 1, 1)),
 (A vertex at (1, 1, 1),
  A vertex at (1, -1, 1),
  A vertex at (-1, -1, 1),
  A vertex at (-1, 1, 1)),
 (A vertex at (-1, -1, 1),
  A vertex at (-1, -1, -1),
  A vertex at (-1, 1, -1),
  A vertex at (-1, 1, 1)),
 (A vertex at (1, -1, -1),
```

(continues on next page)

(continued from previous page)

```
  A vertex at (1, 1, -1),
  A vertex at (-1, -1, -1),
  A vertex at (-1, 1, -1)),
 (A vertex at (1, -1, -1),
  A vertex at (1, -1, 1),
  A vertex at (-1, -1, 1),
  A vertex at (-1, -1, -1)))
sage: C.facets(names=False)
((0, 1, 2, 3),
 (1, 2, 6, 7),
 (2, 3, 4, 7),
 (4, 5, 6, 7),
 (0, 1, 5, 6),
 (0, 3, 4, 5))
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> C = CombinatorialPolyhedron(P)
>>> C.facets()
((A vertex at (1, -1, -1),
  A vertex at (1, 1, -1),
  A vertex at (1, 1, 1),
  A vertex at (1, -1, 1)),
 (A vertex at (1, 1, -1),
  A vertex at (1, 1, 1),
  A vertex at (-1, 1, -1),
  A vertex at (-1, 1, 1)),
 (A vertex at (1, 1, 1),
  A vertex at (1, -1, 1),
  A vertex at (-1, -1, 1),
  A vertex at (-1, 1, 1)),
 (A vertex at (-1, -1, 1),
  A vertex at (-1, -1, -1),
  A vertex at (-1, 1, -1),
  A vertex at (-1, 1, 1)),
 (A vertex at (1, -1, -1),
  A vertex at (1, 1, -1),
  A vertex at (-1, -1, -1),
  A vertex at (-1, 1, -1)),
 (A vertex at (1, -1, -1),
  A vertex at (1, -1, 1),
  A vertex at (-1, -1, 1),
  A vertex at (-1, -1, -1)))
>>> C.facets(names=False)
((0, 1, 2, 3),
 (1, 2, 6, 7),
 (2, 3, 4, 7),
 (4, 5, 6, 7),
 (0, 1, 5, 6),
 (0, 3, 4, 5))
```

The empty face is trivial and hence the `0`-dimensional polyhedron does not have facets:

```
sage: C = CombinatorialPolyhedron(0)
sage: C.facets()
()
```

```
>>> from sage.all import *
>>> C = CombinatorialPolyhedron(Integer(0))
>>> C.facets()
()
```

flag_f_vector(**args*)

Return the flag f-vector.

For each $-1 < i_0 < \cdots < i_n < d$ the flag f-vector counts the number of flags $F_0 \subset \cdots \subset F_n$ with F_j of dimension i_j for each $0 \leq j \leq n$, where d is the dimension of the polyhedron.

INPUT:

- `args` – integer (optional); specify an entry of the flag-f-vector (must be an increasing sequence of integers)

OUTPUT:

- a dictionary, if no arguments were given
- an integer, if arguments were given

EXAMPLES:

Obtain the entire flag-f-vector:

```
sage: C = polytopes.hypercube(4).combinatorial_polyhedron()
sage: C.flag_f_vector()                                                 #␣
↪needs sage.combinat
    {(-1,): 1,
     (0,): 16,
     (0, 1): 64,
     (0, 1, 2): 192,
     (0, 1, 2, 3): 384,
     (0, 1, 3): 192,
     (0, 2): 96,
     (0, 2, 3): 192,
     (0, 3): 64,
     (1,): 32,
     (1, 2): 96,
     (1, 2, 3): 192,
     (1, 3): 96,
     (2,): 24,
     (2, 3): 48,
     (3,): 8,
     (4,): 1}
```

```
>>> from sage.all import *
>>> C = polytopes.hypercube(Integer(4)).combinatorial_polyhedron()
>>> C.flag_f_vector()                                                 #␣
↪needs sage.combinat
    {(-1,): 1,
```

(continues on next page)

(continued from previous page)

```
     (0,): 16,
     (0, 1): 64,
     (0, 1, 2): 192,
     (0, 1, 2, 3): 384,
     (0, 1, 3): 192,
     (0, 2): 96,
     (0, 2, 3): 192,
     (0, 3): 64,
     (1,): 32,
     (1, 2): 96,
     (1, 2, 3): 192,
     (1, 3): 96,
     (2,): 24,
     (2, 3): 48,
     (3,): 8,
     (4,): 1}
```

Specify an entry:

```
sage: C.flag_f_vector(0,3)                                                    #␣
↪needs sage.combinat
64
sage: C.flag_f_vector(2)                                                      #␣
↪needs sage.combinat
24
```

```
>>> from sage.all import *
>>> C.flag_f_vector(Integer(0),Integer(3))                                     ␣
↪               # needs sage.combinat
64
>>> C.flag_f_vector(Integer(2))                                                ␣
↪      # needs sage.combinat
24
```

Leading `-1` and trailing entry of dimension are allowed:

```
sage: C.flag_f_vector(-1,0,3)                                                 #␣
↪needs sage.combinat
64
sage: C.flag_f_vector(-1,0,3,4)                                               #␣
↪needs sage.combinat
64
```

```
>>> from sage.all import *
>>> C.flag_f_vector(-Integer(1),Integer(0),Integer(3))                         ␣
↪                        # needs sage.combinat
64
>>> C.flag_f_vector(-Integer(1),Integer(0),Integer(3),Integer(4))              ␣
↪                                 # needs sage.combinat
64
```

One can get the number of trivial faces:

```
sage: C.flag_f_vector(-1)                                                    #␣
↪needs sage.combinat
1
sage: C.flag_f_vector(4)                                                     #␣
↪needs sage.combinat
1
```

```
>>> from sage.all import *
>>> C.flag_f_vector(-Integer(1))                                             ␣
↪     # needs sage.combinat
1
>>> C.flag_f_vector(Integer(4))                                              ␣
↪     # needs sage.combinat
1
```

Polyhedra with lines, have `0` entries accordingly:

```
sage: C = (Polyhedron(lines=[[1]]) * polytopes.hypercube(2)).combinatorial_
↪polyhedron()
sage: C.flag_f_vector()                                                      #␣
↪needs sage.combinat
{(-1,): 1, (0, 1): 0, (0, 2): 0, (0,): 0, (1, 2): 8, (1,): 4, (2,): 4, 3: 1}
```

```
>>> from sage.all import *
>>> C = (Polyhedron(lines=[[Integer(1)]]) * polytopes.hypercube(Integer(2))).
↪combinatorial_polyhedron()
>>> C.flag_f_vector()                                                      #␣
↪needs sage.combinat
{(-1,): 1, (0, 1): 0, (0, 2): 0, (0,): 0, (1, 2): 8, (1,): 4, (2,): 4, 3: 1}
```

If the arguments are not stricly increasing or out of range, a key error is raised:

```
sage: C.flag_f_vector(-1,0,3,5)                                              #␣
↪needs sage.combinat
Traceback (most recent call last):
...
KeyError: (0, 3, 5)
sage: C.flag_f_vector(-1,3,0)                                                #␣
↪needs sage.combinat
Traceback (most recent call last):
...
KeyError: (3, 0)
```

```
>>> from sage.all import *
>>> C.flag_f_vector(-Integer(1),Integer(0),Integer(3),Integer(5))            ␣
↪                          # needs sage.combinat
Traceback (most recent call last):
...
KeyError: (0, 3, 5)
>>> C.flag_f_vector(-Integer(1),Integer(3),Integer(0))                       ␣
↪                 # needs sage.combinat
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
KeyError: (3, 0)
```

graph(*names=True*, *algorithm=None*)

Return a graph in which the vertices correspond to vertices of the polyhedron, and edges to bounded rank 1 faces.

INPUT:

- `names` – boolean (default: `True`); if `False`, then the nodes of the graph are labeld by the indices of the Vrepresentation
- `algorithm` – string (optional); specify whether the face generator starts with facets or vertices: * `'primal'` – start with the facets * `'dual'` – start with the vertices * `None` – choose automatically

EXAMPLES:

```
sage: P = polytopes.cyclic_polytope(3,5)
sage: C = CombinatorialPolyhedron(P)
sage: G = C.vertex_graph(); G                                          #␣
↪needs sage.graphs
Graph on 5 vertices
sage: sorted(G.degree())                                               #␣
↪needs sage.graphs
[3, 3, 4, 4, 4]

sage: P = Polyhedron(rays=[[1]])
sage: C = CombinatorialPolyhedron(P)
sage: C.graph()                                                        #␣
↪needs sage.graphs
Graph on 1 vertex
```

```
>>> from sage.all import *
>>> P = polytopes.cyclic_polytope(Integer(3),Integer(5))
>>> C = CombinatorialPolyhedron(P)
>>> G = C.vertex_graph(); G                                          #␣
↪needs sage.graphs
Graph on 5 vertices
>>> sorted(G.degree())                                               #␣
↪needs sage.graphs
[3, 3, 4, 4, 4]

>>> P = Polyhedron(rays=[[Integer(1)]])
>>> C = CombinatorialPolyhedron(P)
>>> C.graph()                                                        #␣
↪needs sage.graphs
Graph on 1 vertex
```

hasse_diagram()

Return the Hasse diagram of `self`.

This is the Hasse diagram of the poset of the faces of `self`: A directed graph consisting of a vertex for each face and an edge for each minimal inclusion of faces.

Note

The vertices of the Hasse diagram are given by indices. Use `CombinatorialPolyhedron.face_by_face_lattice_index()` to relabel.

Warning

The indices of the Hasse diagram might depend on architecture and implementation. Relabeling the face lattice with `CombinatorialPolyhedron.face_by_face_lattice_index()` or the properties obtained from this face will be platform independent

EXAMPLES:

```
sage: # needs sage.graphs sage.rings.number_field
sage: P = polytopes.regular_polygon(4).pyramid()
sage: C = CombinatorialPolyhedron(P)
sage: D = C.hasse_diagram(); D
Digraph on 20 vertices
sage: D.average_degree()
21/5
sage: D.relabel(C.face_by_face_lattice_index)
sage: dim_0_vert = D.vertices(sort=True)[1:6]; dim_0_vert
[A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 0-dimensional face of a 3-dimensional combinatorial polyhedron]
sage: sorted(D.out_degree(vertices=dim_0_vert))
[3, 3, 3, 3, 4]
```

```
>>> from sage.all import *
>>> # needs sage.graphs sage.rings.number_field
>>> P = polytopes.regular_polygon(Integer(4)).pyramid()
>>> C = CombinatorialPolyhedron(P)
>>> D = C.hasse_diagram(); D
Digraph on 20 vertices
>>> D.average_degree()
21/5
>>> D.relabel(C.face_by_face_lattice_index)
>>> dim_0_vert = D.vertices(sort=True)[Integer(1):Integer(6)]; dim_0_vert
[A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 0-dimensional face of a 3-dimensional combinatorial polyhedron]
>>> sorted(D.out_degree(vertices=dim_0_vert))
[3, 3, 3, 3, 4]
```

incidence_matrix()

Return the incidence matrix.

Note

The columns correspond to inequalities/equations in the order *Hrepresentation()*, the rows correspond to vertices/rays/lines in the order *Vrepresentation()*.

See also

`incidence_matrix()`.

EXAMPLES:

```
sage: P = polytopes.cube()
sage: C = P.combinatorial_polyhedron()
sage: C.incidence_matrix()
[1 0 0 0 1 1]
[1 1 0 0 1 0]
[1 1 1 0 0 0]
[1 0 1 0 0 1]
[0 0 1 1 0 1]
[0 0 0 1 1 1]
[0 1 0 1 1 0]
[0 1 1 1 0 0]
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> C = P.combinatorial_polyhedron()
>>> C.incidence_matrix()
[1 0 0 0 1 1]
[1 1 0 0 1 0]
[1 1 1 0 0 0]
[1 0 1 0 0 1]
[0 0 1 1 0 1]
[0 0 0 1 1 1]
[0 1 0 1 1 0]
[0 1 1 1 0 0]
```

In this case the incidence matrix is only computed once:

```
sage: P.incidence_matrix() is C.incidence_matrix()
True
sage: C.incidence_matrix.clear_cache()
sage: C.incidence_matrix() is P.incidence_matrix()
False
sage: C.incidence_matrix() == P.incidence_matrix()
True
```

```
>>> from sage.all import *
>>> P.incidence_matrix() is C.incidence_matrix()
True
>>> C.incidence_matrix.clear_cache()
>>> C.incidence_matrix() is P.incidence_matrix()
```

(continues on next page)

(continued from previous page)

```
False
>>> C.incidence_matrix() == P.incidence_matrix()
True
```

```
sage: # needs sage.combinat
sage: P = polytopes.permutahedron(5, backend='field')
sage: C = P.combinatorial_polyhedron()
sage: C.incidence_matrix.clear_cache()
sage: C.incidence_matrix() == P.incidence_matrix()
True
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> P = polytopes.permutahedron(Integer(5), backend='field')
>>> C = P.combinatorial_polyhedron()
>>> C.incidence_matrix.clear_cache()
>>> C.incidence_matrix() == P.incidence_matrix()
True
```

The incidence matrix is consistent with `incidence_matrix()`:

```
sage: P = Polyhedron([[0,0]])
sage: P.incidence_matrix()
[1 1]
sage: C = P.combinatorial_polyhedron()
sage: C.incidence_matrix.clear_cache()
sage: P.combinatorial_polyhedron().incidence_matrix()
[1 1]
```

```
>>> from sage.all import *
>>> P = Polyhedron([[Integer(0),Integer(0)]])
>>> P.incidence_matrix()
[1 1]
>>> C = P.combinatorial_polyhedron()
>>> C.incidence_matrix.clear_cache()
>>> P.combinatorial_polyhedron().incidence_matrix()
[1 1]
```

is_bipyramid(*certificate=False*)

Test whether the polytope is a bipyramid over some other polytope.

INPUT:

- `certificate` – boolean (default: `False`); specifies whether to return a vertex of the polytope which is the apex of a pyramid, if found

INPUT:

- `certificate` – boolean (default: `False`); specifies whether to return two vertices of the polytope which are the apices of a bipyramid, if found

OUTPUT:

If `certificate` is `True`, returns a tuple containing:

1. Boolean.
2. **`None` or a tuple containing:**
   a. The first apex.
   b. The second apex.

If `certificate` is `False` returns a boolean.

EXAMPLES:

```
sage: C = polytopes.hypercube(4).combinatorial_polyhedron()
sage: C.is_bipyramid()
False
sage: C.is_bipyramid(certificate=True)
(False, None)
sage: C = polytopes.cross_polytope(4).combinatorial_polyhedron()
sage: C.is_bipyramid()
True
sage: C.is_bipyramid(certificate=True)
(True, [A vertex at (1, 0, 0, 0), A vertex at (-1, 0, 0, 0)])
```

```
>>> from sage.all import *
>>> C = polytopes.hypercube(Integer(4)).combinatorial_polyhedron()
>>> C.is_bipyramid()
False
>>> C.is_bipyramid(certificate=True)
(False, None)
>>> C = polytopes.cross_polytope(Integer(4)).combinatorial_polyhedron()
>>> C.is_bipyramid()
True
>>> C.is_bipyramid(certificate=True)
(True, [A vertex at (1, 0, 0, 0), A vertex at (-1, 0, 0, 0)])
```

For unbounded polyhedra, an error is raised:

```
sage: C = CombinatorialPolyhedron([[0,1], [0,2]], far_face=[1,2],
↪unbounded=True)
sage: C.is_pyramid()
Traceback (most recent call last):
...
ValueError: polyhedron has to be compact
```

```
>>> from sage.all import *
>>> C = CombinatorialPolyhedron([[Integer(0),Integer(1)], [Integer(0),
↪Integer(2)]], far_face=[Integer(1),Integer(2)], unbounded=True)
>>> C.is_pyramid()
Traceback (most recent call last):
...
ValueError: polyhedron has to be compact
```

ALGORITHM:

Assume all faces of a polyhedron to be given as lists of vertices.

A polytope is a bipyramid with apexes v, w if and only if for each proper face $v \in F$ there exists a face G with $G \setminus \{w\} = F \setminus \{v\}$ and vice versa (for each proper face $w \in F$ there exists $\dots$).

To check this property it suffices to check for all facets of the polyhedron.

is_compact()

Return whether the polyhedron is compact.

EXAMPLES:

```
sage: C = CombinatorialPolyhedron([[0,1], [0,2]], far_face=[1,2],␣
↪unbounded=True)
sage: C.is_compact()
False
sage: C = CombinatorialPolyhedron([[0,1], [0,2], [1,2]])
sage: C.is_compact()
True
sage: P = polytopes.simplex()
sage: P.combinatorial_polyhedron().is_compact()
True
sage: P = Polyhedron(rays=P.vertices())
sage: P.combinatorial_polyhedron().is_compact()
False
```

```
>>> from sage.all import *
>>> C = CombinatorialPolyhedron([[Integer(0),Integer(1)], [Integer(0),
↪Integer(2)]], far_face=[Integer(1),Integer(2)], unbounded=True)
>>> C.is_compact()
False
>>> C = CombinatorialPolyhedron([[Integer(0),Integer(1)], [Integer(0),
↪Integer(2)], [Integer(1),Integer(2)]])
>>> C.is_compact()
True
>>> P = polytopes.simplex()
>>> P.combinatorial_polyhedron().is_compact()
True
>>> P = Polyhedron(rays=P.vertices())
>>> P.combinatorial_polyhedron().is_compact()
False
```

is_lawrence_polytope()

Return `True` if `self` is a Lawrence polytope.

A polytope is called a Lawrence polytope if it has a centrally symmetric (normalized) Gale diagram.

Equivalently, there exists a partition $P_1, \dots, P_k$ of the vertices V such that each part P_i has size 2 or 1 and for each part there exists a facet with vertices exactly $V \setminus P_i$.

EXAMPLES:

```
sage: C = polytopes.simplex(5).combinatorial_polyhedron()
sage: C.is_lawrence_polytope()
True
sage: P = polytopes.hypercube(4).lawrence_polytope()
sage: C = P.combinatorial_polyhedron()
sage: C.is_lawrence_polytope()
```

(continues on next page)

(continued from previous page)

```
True
sage: P = polytopes.hypercube(4)
sage: C = P.combinatorial_polyhedron()
sage: C.is_lawrence_polytope()
False
```

```
>>> from sage.all import *
>>> C = polytopes.simplex(Integer(5)).combinatorial_polyhedron()
>>> C.is_lawrence_polytope()
True
>>> P = polytopes.hypercube(Integer(4)).lawrence_polytope()
>>> C = P.combinatorial_polyhedron()
>>> C.is_lawrence_polytope()
True
>>> P = polytopes.hypercube(Integer(4))
>>> C = P.combinatorial_polyhedron()
>>> C.is_lawrence_polytope()
False
```

For unbounded polyhedra, an error is raised:

```
sage: C = CombinatorialPolyhedron([[0,1], [0,2]], far_face=[1,2],␣
↪unbounded=True)
sage: C.is_lawrence_polytope()
Traceback (most recent call last):
...
NotImplementedError: this function is implemented for polytopes only
```

```
>>> from sage.all import *
>>> C = CombinatorialPolyhedron([[Integer(0),Integer(1)], [Integer(0),
↪Integer(2)]], far_face=[Integer(1),Integer(2)], unbounded=True)
>>> C.is_lawrence_polytope()
Traceback (most recent call last):
...
NotImplementedError: this function is implemented for polytopes only
```

AUTHORS:

- Laith Rastanawi
- Jonathan Kliem

REFERENCES:

For more information, see [BaSt1990].

is_neighborly(*k=None*)

Return whether the polyhedron is neighborly.

If the input k is provided, then return whether the polyhedron is k-neighborly.

A polyhedron is neighborly if every set of n vertices forms a face for n up to floor of half the dimension of the polyhedron. It is k-neighborly if this is true for n up to k.

INPUT:

- `k` – the dimension up to which to check if every set of `k` vertices forms a face. If no `k` is provided, check up to floor of half the dimension of the polyhedron.

OUTPUT:

- `True` if the every set of up to `k` vertices forms a face,
- `False` otherwise

See also:

neighborliness()

EXAMPLES:

```
sage: P = polytopes.cyclic_polytope(8,12)
sage: C = P.combinatorial_polyhedron()
sage: C.is_neighborly()
True
sage: P = polytopes.simplex(6)
sage: C = P.combinatorial_polyhedron()
sage: C.is_neighborly()
True
sage: P = polytopes.cyclic_polytope(4,10)
sage: P = P.join(P)
sage: C = P.combinatorial_polyhedron()
sage: C.is_neighborly()
False
sage: C.is_neighborly(k=2)
True
```

```
>>> from sage.all import *
>>> P = polytopes.cyclic_polytope(Integer(8),Integer(12))
>>> C = P.combinatorial_polyhedron()
>>> C.is_neighborly()
True
>>> P = polytopes.simplex(Integer(6))
>>> C = P.combinatorial_polyhedron()
>>> C.is_neighborly()
True
>>> P = polytopes.cyclic_polytope(Integer(4),Integer(10))
>>> P = P.join(P)
>>> C = P.combinatorial_polyhedron()
>>> C.is_neighborly()
False
>>> C.is_neighborly(k=Integer(2))
True
```

is_prism(*certificate=False*)

Test whether the polytope is a prism of some polytope.

INPUT:

- `certificate` – boolean (default: `False`); specifies whether to return two facets of the polytope which are the bases of a prism, if found

OUTPUT:

If `certificate` is `True`, returns a tuple containing:

1. Boolean.
2. **`None` or a tuple containing:**
   a. List of the vertices of the first base facet.
   b. List of the vertices of the second base facet.

If `certificate` is `False` returns a boolean.

is_pyramid(*certificate=False*)

Test whether the polytope is a pyramid over one of its facets.

INPUT:

- `certificate` – boolean (default: `False`); specifies whether to return a vertex of the polytope which is the apex of a pyramid, if found

OUTPUT:

If `certificate` is `True`, returns a tuple containing:

1. Boolean.
2. The apex of the pyramid or `None`.

If `certificate` is `False` returns a boolean.

AUTHORS:

- Laith Rastanawi
- Jonathan Kliem

EXAMPLES:

```
sage: C = polytopes.cross_polytope(4).combinatorial_polyhedron()
sage: C.is_pyramid()
False
sage: C.is_pyramid(certificate=True)
(False, None)
sage: C = polytopes.cross_polytope(4).pyramid().combinatorial_polyhedron()
sage: C.is_pyramid()
True
sage: C.is_pyramid(certificate=True)
(True, A vertex at (1, 0, 0, 0, 0))
sage: C = polytopes.simplex(5).combinatorial_polyhedron()
sage: C.is_pyramid(certificate=True)
(True, A vertex at (1, 0, 0, 0, 0, 0))
```

```
>>> from sage.all import *
>>> C = polytopes.cross_polytope(Integer(4)).combinatorial_polyhedron()
>>> C.is_pyramid()
False
>>> C.is_pyramid(certificate=True)
(False, None)
>>> C = polytopes.cross_polytope(Integer(4)).pyramid().combinatorial_
↪polyhedron()
```

(continues on next page)

(continued from previous page)

```
>>> C.is_pyramid()
True
>>> C.is_pyramid(certificate=True)
(True, A vertex at (1, 0, 0, 0, 0))
>>> C = polytopes.simplex(Integer(5)).combinatorial_polyhedron()
>>> C.is_pyramid(certificate=True)
(True, A vertex at (1, 0, 0, 0, 0, 0))
```

For unbounded polyhedra, an error is raised:

```
sage: C = CombinatorialPolyhedron([[0,1], [0,2]], far_face=[1,2],␣
↪unbounded=True)
sage: C.is_pyramid()
Traceback (most recent call last):
...
ValueError: polyhedron has to be compact
```

```
>>> from sage.all import *
>>> C = CombinatorialPolyhedron([[Integer(0),Integer(1)], [Integer(0),
↪Integer(2)]], far_face=[Integer(1),Integer(2)], unbounded=True)
>>> C.is_pyramid()
Traceback (most recent call last):
...
ValueError: polyhedron has to be compact
```

is_simple()

Test whether the polytope is simple.

If the polyhedron is unbounded, return `False`.

A polytope is simple, if each vertex is contained in exactly d facets, where d is the dimension of the polytope.

EXAMPLES:

```
sage: P = polytopes.cyclic_polytope(4,10)
sage: C = P.combinatorial_polyhedron()
sage: C.is_simple()
False
sage: P = polytopes.hypercube(4)
sage: C = P.combinatorial_polyhedron()
sage: C.is_simple()
True
```

```
>>> from sage.all import *
>>> P = polytopes.cyclic_polytope(Integer(4),Integer(10))
>>> C = P.combinatorial_polyhedron()
>>> C.is_simple()
False
>>> P = polytopes.hypercube(Integer(4))
>>> C = P.combinatorial_polyhedron()
>>> C.is_simple()
True
```

Return `False` for unbounded polyhedra:

```
sage: C = CombinatorialPolyhedron([[0,1], [0,2]], far_face=[1,2],␣
↪unbounded=True)
sage: C.is_simple()
False
```

```
>>> from sage.all import *
>>> C = CombinatorialPolyhedron([[Integer(0),Integer(1)], [Integer(0),
↪Integer(2)]], far_face=[Integer(1),Integer(2)], unbounded=True)
>>> C.is_simple()
False
```

is_simplex()

Return whether the polyhedron is a simplex.

A simplex is a bounded polyhedron with $d + 1$ vertices, where d is the dimension.

EXAMPLES:

```
sage: CombinatorialPolyhedron(2).is_simplex()
False
sage: CombinatorialPolyhedron([[0,1],[0,2],[1,2]]).is_simplex()
True
```

```
>>> from sage.all import *
>>> CombinatorialPolyhedron(Integer(2)).is_simplex()
False
>>> CombinatorialPolyhedron([[Integer(0),Integer(1)],[Integer(0),Integer(2)],
↪[Integer(1),Integer(2)]]).is_simplex()
True
```

is_simplicial()

Test whether the polytope is simplicial.

This method is not implemented for unbounded polyhedra.

A polytope is simplicial, if each facet contains exactly d vertices, where d is the dimension of the polytope.

EXAMPLES:

```
sage: P = polytopes.cyclic_polytope(4,10)
sage: C = P.combinatorial_polyhedron()
sage: C.is_simplicial()
True
sage: P = polytopes.hypercube(4)
sage: C = P.combinatorial_polyhedron()
sage: C.is_simplicial()
False
```

```
>>> from sage.all import *
>>> P = polytopes.cyclic_polytope(Integer(4),Integer(10))
>>> C = P.combinatorial_polyhedron()
>>> C.is_simplicial()
True
>>> P = polytopes.hypercube(Integer(4))
```

(continues on next page)

(continued from previous page)

```
>>> C = P.combinatorial_polyhedron()
>>> C.is_simplicial()
False
```

For unbounded polyhedra, an error is raised:

```
sage: C = CombinatorialPolyhedron([[0,1], [0,2]], far_face=[1,2],␣
↪unbounded=True)
sage: C.is_simplicial()
Traceback (most recent call last):
...
NotImplementedError: this function is implemented for polytopes only
```

```
>>> from sage.all import *
>>> C = CombinatorialPolyhedron([[Integer(0),Integer(1)], [Integer(0),
↪Integer(2)]], far_face=[Integer(1),Integer(2)], unbounded=True)
>>> C.is_simplicial()
Traceback (most recent call last):
...
NotImplementedError: this function is implemented for polytopes only
```

join_of_Vrep(**indices*)

Return the smallest face containing all Vrepresentatives indicated by the indices.

See also

join_of_Vrep().

EXAMPLES:

```
sage: # needs sage.combinat
sage: P = polytopes.permutahedron(4)
sage: C = CombinatorialPolyhedron(P)
sage: C.join_of_Vrep(0,1)
A 1-dimensional face of a 3-dimensional combinatorial polyhedron
sage: C.join_of_Vrep(0,11).ambient_V_indices()
(0, 1, 10, 11, 12, 13)
sage: C.join_of_Vrep(8).ambient_V_indices()
(8,)
sage: C.join_of_Vrep().ambient_V_indices()
()
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> P = polytopes.permutahedron(Integer(4))
>>> C = CombinatorialPolyhedron(P)
>>> C.join_of_Vrep(Integer(0),Integer(1))
A 1-dimensional face of a 3-dimensional combinatorial polyhedron
>>> C.join_of_Vrep(Integer(0),Integer(11)).ambient_V_indices()
(0, 1, 10, 11, 12, 13)
>>> C.join_of_Vrep(Integer(8)).ambient_V_indices()
```

(continues on next page)

(continued from previous page)

```
(8,)
>>> C.join_of_Vrep().ambient_V_indices()
()
```

meet_of_Hrep(**indices*)

Return the largest face contained in all facets indicated by the indices.

> **See also**
>
> `meet_of_Hrep()`.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: P = polytopes.dodecahedron()
sage: C = CombinatorialPolyhedron(P)
sage: C.meet_of_Hrep(0)
A 2-dimensional face of a 3-dimensional combinatorial polyhedron
sage: C.meet_of_Hrep(0).ambient_H_indices()
(0,)
sage: C.meet_of_Hrep(0,1).ambient_H_indices()
(0, 1)
sage: C.meet_of_Hrep(0,2).ambient_H_indices()
(0, 2)
sage: C.meet_of_Hrep(0,2,3).ambient_H_indices()
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)
sage: C.meet_of_Hrep().ambient_H_indices()
()
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> P = polytopes.dodecahedron()
>>> C = CombinatorialPolyhedron(P)
>>> C.meet_of_Hrep(Integer(0))
A 2-dimensional face of a 3-dimensional combinatorial polyhedron
>>> C.meet_of_Hrep(Integer(0)).ambient_H_indices()
(0,)
>>> C.meet_of_Hrep(Integer(0),Integer(1)).ambient_H_indices()
(0, 1)
>>> C.meet_of_Hrep(Integer(0),Integer(2)).ambient_H_indices()
(0, 2)
>>> C.meet_of_Hrep(Integer(0),Integer(2),Integer(3)).ambient_H_indices()
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)
>>> C.meet_of_Hrep().ambient_H_indices()
()
```

n_facets()

Return the number of facets.

Is equivalent to `len(self.facets())`.

EXAMPLES:

```
sage: P = polytopes.cube()
sage: C = CombinatorialPolyhedron(P)
sage: C.n_facets()
6

sage: P = polytopes.cyclic_polytope(4,20)
sage: C = CombinatorialPolyhedron(P)
sage: C.n_facets()
170

sage: P = Polyhedron(lines=[[0,1]], vertices=[[1,0], [-1,0]])
sage: C = CombinatorialPolyhedron(P)
sage: C.n_facets()
2

sage: P = Polyhedron(rays=[[1,0], [-1,0], [0,1]])
sage: C = CombinatorialPolyhedron(P)
sage: C.n_facets()
1

sage: C = CombinatorialPolyhedron(-1)
sage: C.f_vector()
(1)
sage: C.n_facets()
0
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> C = CombinatorialPolyhedron(P)
>>> C.n_facets()
6

>>> P = polytopes.cyclic_polytope(Integer(4),Integer(20))
>>> C = CombinatorialPolyhedron(P)
>>> C.n_facets()
170

>>> P = Polyhedron(lines=[[Integer(0),Integer(1)]], vertices=[[Integer(1),
↪Integer(0)], [-Integer(1),Integer(0)]])
>>> C = CombinatorialPolyhedron(P)
>>> C.n_facets()
2

>>> P = Polyhedron(rays=[[Integer(1),Integer(0)], [-Integer(1),Integer(0)],␣
↪[Integer(0),Integer(1)]])
>>> C = CombinatorialPolyhedron(P)
>>> C.n_facets()
1

>>> C = CombinatorialPolyhedron(-Integer(1))
>>> C.f_vector()
(1)
```

(continues on next page)

(continued from previous page)

```
>>> C.n_facets()
0
```

Facets are defined to be the maximal nontrivial faces. The `0`-dimensional polyhedron does not have nontrivial faces:

```
sage: C = CombinatorialPolyhedron(0)
sage: C.f_vector()
(1, 1)
sage: C.n_facets()
0
```

```
>>> from sage.all import *
>>> C = CombinatorialPolyhedron(Integer(0))
>>> C.f_vector()
(1, 1)
>>> C.n_facets()
0
```

n_vertices()

Return the number of vertices.

Is equivalent to `len(self.vertices())`.

EXAMPLES:

```
sage: P = polytopes.cube()
sage: C = CombinatorialPolyhedron(P)
sage: C.n_vertices()
8

sage: P = polytopes.cyclic_polytope(4,20)
sage: C = CombinatorialPolyhedron(P)
sage: C.n_vertices()
20

sage: P = Polyhedron(lines=[[0,1]], vertices=[[1,0], [-1,0]])
sage: C = CombinatorialPolyhedron(P)
sage: C.n_vertices()
0

sage: P = Polyhedron(rays=[[1,0,0], [0,1,0]], lines=[[0,0,1]])
sage: C = CombinatorialPolyhedron(P)
sage: C.n_vertices()
0

sage: C = CombinatorialPolyhedron(4)
sage: C.f_vector()
(1, 0, 0, 0, 0, 1)
sage: C.n_vertices()
0

sage: C = CombinatorialPolyhedron(0)
```

(continues on next page)

(continued from previous page)

```
sage: C.f_vector()
(1, 1)
sage: C.n_vertices()
1
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> C = CombinatorialPolyhedron(P)
>>> C.n_vertices()
8

>>> P = polytopes.cyclic_polytope(Integer(4),Integer(20))
>>> C = CombinatorialPolyhedron(P)
>>> C.n_vertices()
20

>>> P = Polyhedron(lines=[[Integer(0),Integer(1)]], vertices=[[Integer(1),
↪Integer(0)], [-Integer(1),Integer(0)]])
>>> C = CombinatorialPolyhedron(P)
>>> C.n_vertices()
0

>>> P = Polyhedron(rays=[[Integer(1),Integer(0),Integer(0)], [Integer(0),
↪Integer(1),Integer(0)]], lines=[[Integer(0),Integer(0),Integer(1)]])
>>> C = CombinatorialPolyhedron(P)
>>> C.n_vertices()
0

>>> C = CombinatorialPolyhedron(Integer(4))
>>> C.f_vector()
(1, 0, 0, 0, 0, 1)
>>> C.n_vertices()
0

>>> C = CombinatorialPolyhedron(Integer(0))
>>> C.f_vector()
(1, 1)
>>> C.n_vertices()
1
```

neighborliness()

Return the largest `k`, such that the polyhedron is `k`-neighborly.

A polyhedron is k-neighborly if every set of n vertices forms a face for n up to k.

In case of the d-dimensional simplex, it returns $d + 1$.

See also

is_neighborly()

EXAMPLES:

```
sage: P = polytopes.cyclic_polytope(8,12)
sage: C = P.combinatorial_polyhedron()
sage: C.neighborliness()
4
sage: P = polytopes.simplex(6)
sage: C = P.combinatorial_polyhedron()
sage: C.neighborliness()
7
sage: P = polytopes.cyclic_polytope(4,10)
sage: P = P.join(P)
sage: C = P.combinatorial_polyhedron()
sage: C.neighborliness()
2
```

```
>>> from sage.all import *
>>> P = polytopes.cyclic_polytope(Integer(8),Integer(12))
>>> C = P.combinatorial_polyhedron()
>>> C.neighborliness()
4
>>> P = polytopes.simplex(Integer(6))
>>> C = P.combinatorial_polyhedron()
>>> C.neighborliness()
7
>>> P = polytopes.cyclic_polytope(Integer(4),Integer(10))
>>> P = P.join(P)
>>> C = P.combinatorial_polyhedron()
>>> C.neighborliness()
2
```

polar()

Return the dual/polar of `self`.

Only defined for bounded polyhedra.

See also

`polar()`.

EXAMPLES:

```
sage: P = polytopes.cube()
sage: C = P.combinatorial_polyhedron()
sage: D = C.dual()
sage: D.f_vector()
(1, 6, 12, 8, 1)
sage: D1 = P.polar().combinatorial_polyhedron()
sage: D1.face_lattice().is_isomorphic(D.face_lattice())                  #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
```

(continues on next page)

(continued from previous page)

```
>>> C = P.combinatorial_polyhedron()
>>> D = C.dual()
>>> D.f_vector()
(1, 6, 12, 8, 1)
>>> D1 = P.polar().combinatorial_polyhedron()
>>> D1.face_lattice().is_isomorphic(D.face_lattice())                    #␣
↪needs sage.combinat
True
```

Polar is an alias to be consistent with *Polyhedron_base*:

```
sage: C.polar().f_vector()
(1, 6, 12, 8, 1)
```

```
>>> from sage.all import *
>>> C.polar().f_vector()
(1, 6, 12, 8, 1)
```

For unbounded polyhedra, an error is raised:

```
sage: C = CombinatorialPolyhedron([[0,1], [0,2]], far_face=[1,2],␣
↪unbounded=True)
sage: C.dual()
Traceback (most recent call last):
...
ValueError: self must be bounded
```

```
>>> from sage.all import *
>>> C = CombinatorialPolyhedron([[Integer(0),Integer(1)], [Integer(0),
↪Integer(2)]], far_face=[Integer(1),Integer(2)], unbounded=True)
>>> C.dual()
Traceback (most recent call last):
...
ValueError: self must be bounded
```

pyramid(*new_vertex=None*, *new_facet=None*)

Return the pyramid of `self`.

INPUT:

- `new_vertex` – (optional); specify a new vertex name to set up the pyramid with vertex names
- `new_facet` – (optional); specify a new facet name to set up the pyramid with facet names

EXAMPLES:

```
sage: C = CombinatorialPolyhedron(((1,2,3),(1,2,4),(1,3,4),(2,3,4)))
sage: C1 = C.pyramid()
sage: C1.facets()
((0, 1, 2, 4), (0, 1, 3, 4), (0, 2, 3, 4), (1, 2, 3, 4), (0, 1, 2, 3))
```

```
>>> from sage.all import *
>>> C = CombinatorialPolyhedron(((Integer(1),Integer(2),Integer(3)),
```

(continues on next page)

(continued from previous page)

```
↪(Integer(1),Integer(2),Integer(4)),(Integer(1),Integer(3),Integer(4)),
↪(Integer(2),Integer(3),Integer(4))))
>>> C1 = C.pyramid()
>>> C1.facets()
((0, 1, 2, 4), (0, 1, 3, 4), (0, 2, 3, 4), (1, 2, 3, 4), (0, 1, 2, 3))
```

```
sage: P = polytopes.cube()
sage: C = CombinatorialPolyhedron(P)
sage: C1 = C.pyramid()
sage: P1 = P.pyramid()
sage: C2 = P1.combinatorial_polyhedron()
sage: C2.vertex_facet_graph().is_isomorphic(C1.vertex_facet_graph())         #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> C = CombinatorialPolyhedron(P)
>>> C1 = C.pyramid()
>>> P1 = P.pyramid()
>>> C2 = P1.combinatorial_polyhedron()
>>> C2.vertex_facet_graph().is_isomorphic(C1.vertex_facet_graph())         #␣
↪needs sage.combinat
True
```

One can specify a name for the new vertex:

```
sage: P = polytopes.cyclic_polytope(4,10)
sage: C = P.combinatorial_polyhedron()
sage: C1 = C.pyramid(new_vertex='apex')
sage: C1.is_pyramid(certificate=True)
(True, 'apex')
sage: C1.facets()[0]
(A vertex at (0, 0, 0, 0),
 A vertex at (1, 1, 1, 1),
 A vertex at (2, 4, 8, 16),
 A vertex at (3, 9, 27, 81),
 'apex')
```

```
>>> from sage.all import *
>>> P = polytopes.cyclic_polytope(Integer(4),Integer(10))
>>> C = P.combinatorial_polyhedron()
>>> C1 = C.pyramid(new_vertex='apex')
>>> C1.is_pyramid(certificate=True)
(True, 'apex')
>>> C1.facets()[Integer(0)]
(A vertex at (0, 0, 0, 0),
 A vertex at (1, 1, 1, 1),
 A vertex at (2, 4, 8, 16),
 A vertex at (3, 9, 27, 81),
 'apex')
```

One can specify a name for the new facets:

```
sage: # needs sage.rings.number_field
sage: P = polytopes.regular_polygon(4)
sage: C = P.combinatorial_polyhedron()
sage: C1 = C.pyramid(new_facet='base')
sage: C1.Hrepresentation()
(An inequality (-1/2, 1/2) x + 1/2 >= 0,
 An inequality (-1/2, -1/2) x + 1/2 >= 0,
 An inequality (1/2, 0.50000000000000000?) x + 1/2 >= 0,
 An inequality (1/2, -1/2) x + 1/2 >= 0,
 'base')
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> P = polytopes.regular_polygon(Integer(4))
>>> C = P.combinatorial_polyhedron()
>>> C1 = C.pyramid(new_facet='base')
>>> C1.Hrepresentation()
(An inequality (-1/2, 1/2) x + 1/2 >= 0,
 An inequality (-1/2, -1/2) x + 1/2 >= 0,
 An inequality (1/2, 0.50000000000000000?) x + 1/2 >= 0,
 An inequality (1/2, -1/2) x + 1/2 >= 0,
 'base')
```

For unbounded polyhedra, an error is raised:

```
sage: C = CombinatorialPolyhedron([[0,1], [0,2]], far_face=[1,2],␣
↪unbounded=True)
sage: C.pyramid()
Traceback (most recent call last):
...
ValueError: self must be bounded
```

```
>>> from sage.all import *
>>> C = CombinatorialPolyhedron([[Integer(0),Integer(1)], [Integer(0),
↪Integer(2)]], far_face=[Integer(1),Integer(2)], unbounded=True)
>>> C.pyramid()
Traceback (most recent call last):
...
ValueError: self must be bounded
```

ridges(*add_equations=False*, *names=True*, *algorithm=None*)

Return the ridges.

The ridges of a polyhedron are the faces contained in exactly two facets.

To obtain all faces of codimension 1 use *CombinatorialPolyhedron.face_generator()* instead.

The ridges will be given by the facets, they are contained in.

INPUT:

- `add_equations` – if `True`, then equations of the polyhedron will be added (only applicable when `names` is `True`)
- `names` – boolean (default: `True`); if `False`, then the facets are given by their indices

- algorithm – string (optional); specify whether the face generator starts with facets or vertices: * 'primal' – start with the facets * 'dual' – start with the vertices * None – choose automatically

Note

To compute ridges and f_vector, compute the ridges first. This might be faster.

EXAMPLES:

```
sage: # needs sage.combinat
sage: P = polytopes.permutahedron(2)
sage: C = CombinatorialPolyhedron(P)
sage: C.ridges()
((An inequality (1, 0) x - 1 >= 0, An inequality (-1, 0) x + 2 >= 0),)
sage: C.ridges(add_equations=True)
(((An inequality (1, 0) x - 1 >= 0, An equation (1, 1) x - 3 == 0),
  (An inequality (-1, 0) x + 2 >= 0, An equation (1, 1) x - 3 == 0)),)

sage: P = polytopes.cyclic_polytope(4,5)
sage: C = CombinatorialPolyhedron(P)
sage: C.ridges()
((An inequality (24, -26, 9, -1) x + 0 >= 0,
  An inequality (-50, 35, -10, 1) x + 24 >= 0),
 (An inequality (-12, 19, -8, 1) x + 0 >= 0,
  An inequality (-50, 35, -10, 1) x + 24 >= 0),
 (An inequality (8, -14, 7, -1) x + 0 >= 0,
  An inequality (-50, 35, -10, 1) x + 24 >= 0),
 (An inequality (-6, 11, -6, 1) x + 0 >= 0,
  An inequality (-50, 35, -10, 1) x + 24 >= 0),
 (An inequality (-12, 19, -8, 1) x + 0 >= 0,
  An inequality (24, -26, 9, -1) x + 0 >= 0),
 (An inequality (8, -14, 7, -1) x + 0 >= 0,
  An inequality (24, -26, 9, -1) x + 0 >= 0),
 (An inequality (-6, 11, -6, 1) x + 0 >= 0,
  An inequality (24, -26, 9, -1) x + 0 >= 0),
 (An inequality (8, -14, 7, -1) x + 0 >= 0,
  An inequality (-12, 19, -8, 1) x + 0 >= 0),
 (An inequality (-6, 11, -6, 1) x + 0 >= 0,
  An inequality (-12, 19, -8, 1) x + 0 >= 0),
 (An inequality (-6, 11, -6, 1) x + 0 >= 0,
  An inequality (8, -14, 7, -1) x + 0 >= 0))
sage: C.ridges(names=False)
((3, 4),
 (2, 4),
 (1, 4),
 (0, 4),
 (2, 3),
 (1, 3),
 (0, 3),
 (1, 2),
 (0, 2),
 (0, 1))
```

(continues on next page)

(continued from previous page)

```
sage: P = Polyhedron(rays=[[1,0]])
sage: C = CombinatorialPolyhedron(P)
sage: C
A 1-dimensional combinatorial polyhedron with 1 facet
sage: C.ridges()
()
sage: it = C.face_generator(0)
sage: for face in it: face.ambient_Hrepresentation()
(An inequality (1, 0) x + 0 >= 0, An equation (0, 1) x + 0 == 0)
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> P = polytopes.permutahedron(Integer(2))
>>> C = CombinatorialPolyhedron(P)
>>> C.ridges()
((An inequality (1, 0) x - 1 >= 0, An inequality (-1, 0) x + 2 >= 0),)
>>> C.ridges(add_equations=True)
(((An inequality (1, 0) x - 1 >= 0, An equation (1, 1) x - 3 == 0),
  (An inequality (-1, 0) x + 2 >= 0, An equation (1, 1) x - 3 == 0)),)

>>> P = polytopes.cyclic_polytope(Integer(4),Integer(5))
>>> C = CombinatorialPolyhedron(P)
>>> C.ridges()
((An inequality (24, -26, 9, -1) x + 0 >= 0,
  An inequality (-50, 35, -10, 1) x + 24 >= 0),
 (An inequality (-12, 19, -8, 1) x + 0 >= 0,
  An inequality (-50, 35, -10, 1) x + 24 >= 0),
 (An inequality (8, -14, 7, -1) x + 0 >= 0,
  An inequality (-50, 35, -10, 1) x + 24 >= 0),
 (An inequality (-6, 11, -6, 1) x + 0 >= 0,
  An inequality (-50, 35, -10, 1) x + 24 >= 0),
 (An inequality (-12, 19, -8, 1) x + 0 >= 0,
  An inequality (24, -26, 9, -1) x + 0 >= 0),
 (An inequality (8, -14, 7, -1) x + 0 >= 0,
  An inequality (24, -26, 9, -1) x + 0 >= 0),
 (An inequality (-6, 11, -6, 1) x + 0 >= 0,
  An inequality (24, -26, 9, -1) x + 0 >= 0),
 (An inequality (8, -14, 7, -1) x + 0 >= 0,
  An inequality (-12, 19, -8, 1) x + 0 >= 0),
 (An inequality (-6, 11, -6, 1) x + 0 >= 0,
  An inequality (-12, 19, -8, 1) x + 0 >= 0),
 (An inequality (-6, 11, -6, 1) x + 0 >= 0,
  An inequality (8, -14, 7, -1) x + 0 >= 0))
>>> C.ridges(names=False)
((3, 4),
 (2, 4),
 (1, 4),
 (0, 4),
 (2, 3),
 (1, 3),
 (0, 3),
```

(continues on next page)

(continued from previous page)

```
 (1, 2),
 (0, 2),
 (0, 1))

>>> P = Polyhedron(rays=[[Integer(1),Integer(0)]])
>>> C = CombinatorialPolyhedron(P)
>>> C
A 1-dimensional combinatorial polyhedron with 1 facet
>>> C.ridges()
()
>>> it = C.face_generator(Integer(0))
>>> for face in it: face.ambient_Hrepresentation()
(An inequality (1, 0) x + 0 >= 0, An equation (0, 1) x + 0 == 0)
```

simpliciality()

Return the largest k such that the polytope is k-simplicial.

Return the dimension in case of a simplex.

A polytope is k-simplicial, if every k-face is a simplex.

EXAMPLES:

```
sage: cyclic = polytopes.cyclic_polytope(10,4)
sage: CombinatorialPolyhedron(cyclic).simpliciality()
3

sage: hypersimplex = polytopes.hypersimplex(5,2)
sage: CombinatorialPolyhedron(hypersimplex).simpliciality()
2

sage: cross = polytopes.cross_polytope(4)
sage: P = cross.join(cross)
sage: CombinatorialPolyhedron(P).simpliciality()
3

sage: P = polytopes.simplex(3)
sage: CombinatorialPolyhedron(P).simpliciality()
3

sage: P = polytopes.simplex(1)
sage: CombinatorialPolyhedron(P).simpliciality()
1
```

```
>>> from sage.all import *
>>> cyclic = polytopes.cyclic_polytope(Integer(10),Integer(4))
>>> CombinatorialPolyhedron(cyclic).simpliciality()
3

>>> hypersimplex = polytopes.hypersimplex(Integer(5),Integer(2))
>>> CombinatorialPolyhedron(hypersimplex).simpliciality()
2
```

(continues on next page)

(continued from previous page)

```
>>> cross = polytopes.cross_polytope(Integer(4))
>>> P = cross.join(cross)
>>> CombinatorialPolyhedron(P).simpliciality()
3

>>> P = polytopes.simplex(Integer(3))
>>> CombinatorialPolyhedron(P).simpliciality()
3

>>> P = polytopes.simplex(Integer(1))
>>> CombinatorialPolyhedron(P).simpliciality()
1
```

simplicity()

Return the largest k such that the polytope is k-simple.

Return the dimension in case of a simplex.

A polytope P is k-simple, if every $(d-1-k)$-face is contained in exactly $k+1$ facets of P for $1 \leq k \leq d-1$.

Equivalently it is k-simple if the polar/dual polytope is k-simplicial.

EXAMPLES:

```
sage: hyper4 = polytopes.hypersimplex(4,2)
sage: CombinatorialPolyhedron(hyper4).simplicity()
1

sage: hyper5 = polytopes.hypersimplex(5,2)
sage: CombinatorialPolyhedron(hyper5).simplicity()
2

sage: hyper6 = polytopes.hypersimplex(6,2)
sage: CombinatorialPolyhedron(hyper6).simplicity()
3

sage: P = polytopes.simplex(3)
sage: CombinatorialPolyhedron(P).simplicity()
3

sage: P = polytopes.simplex(1)
sage: CombinatorialPolyhedron(P).simplicity()
1
```

```
>>> from sage.all import *
>>> hyper4 = polytopes.hypersimplex(Integer(4),Integer(2))
>>> CombinatorialPolyhedron(hyper4).simplicity()
1

>>> hyper5 = polytopes.hypersimplex(Integer(5),Integer(2))
>>> CombinatorialPolyhedron(hyper5).simplicity()
2

>>> hyper6 = polytopes.hypersimplex(Integer(6),Integer(2))
```

(continues on next page)

(continued from previous page)

```
>>> CombinatorialPolyhedron(hyper6).simplicity()
3

>>> P = polytopes.simplex(Integer(3))
>>> CombinatorialPolyhedron(P).simplicity()
3

>>> P = polytopes.simplex(Integer(1))
>>> CombinatorialPolyhedron(P).simplicity()
1
```

vertex_adjacency_matrix(*algorithm=None*)

Return the binary matrix of vertex adjacencies.

INPUT:

- `algorithm` – string (optional); specify whether the face generator starts with facets or vertices: * `'primal'` – start with the facets * `'dual'` – start with the vertices * `None` – choose automatically

See also

`vertex_adjacency_matrix()`.

EXAMPLES:

```
sage: P = polytopes.cube()
sage: C = P.combinatorial_polyhedron()
sage: C.vertex_adjacency_matrix()
[0 1 0 1 0 1 0 0]
[1 0 1 0 0 0 1 0]
[0 1 0 1 0 0 0 1]
[1 0 1 0 1 0 0 0]
[0 0 0 1 0 1 0 1]
[1 0 0 0 1 0 1 0]
[0 1 0 0 0 1 0 1]
[0 0 1 0 1 0 1 0]
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> C = P.combinatorial_polyhedron()
>>> C.vertex_adjacency_matrix()
[0 1 0 1 0 1 0 0]
[1 0 1 0 0 0 1 0]
[0 1 0 1 0 0 0 1]
[1 0 1 0 1 0 0 0]
[0 0 0 1 0 1 0 1]
[1 0 0 0 1 0 1 0]
[0 1 0 0 0 1 0 1]
[0 0 1 0 1 0 1 0]
```

vertex_facet_graph(*names=True*)

Return the vertex-facet graph.

This method constructs a directed bipartite graph. The nodes of the graph correspond to elements of the Vrepresentation and facets. There is a directed edge from Vrepresentation to facets for each incidence.

If `names` is set to `False`, then the vertices (of the graph) are given by integers.

INPUT:

- `names` – boolean (default: `True`); if `True` label the vertices of the graph by the corresponding names of the Vrepresentation resp. Hrepresentation; if `False` label the vertices of the graph by integers

EXAMPLES:

```
sage: P = polytopes.hypercube(2).pyramid()
sage: C = CombinatorialPolyhedron(P)
sage: G = C.vertex_facet_graph(); G                                     #␣
↪needs sage.graphs
Digraph on 10 vertices
sage: C.Vrepresentation()
(A vertex at (0, -1, -1),
 A vertex at (0, -1, 1),
 A vertex at (0, 1, -1),
 A vertex at (0, 1, 1),
 A vertex at (1, 0, 0))
sage: sorted(G.neighbors_out(C.Vrepresentation()[4]))                   #␣
↪needs sage.graphs
[An inequality (-1, -1, 0) x + 1 >= 0,
 An inequality (-1, 0, -1) x + 1 >= 0,
 An inequality (-1, 0, 1) x + 1 >= 0,
 An inequality (-1, 1, 0) x + 1 >= 0]
```

```
>>> from sage.all import *
>>> P = polytopes.hypercube(Integer(2)).pyramid()
>>> C = CombinatorialPolyhedron(P)
>>> G = C.vertex_facet_graph(); G                                     #␣
↪needs sage.graphs
Digraph on 10 vertices
>>> C.Vrepresentation()
(A vertex at (0, -1, -1),
 A vertex at (0, -1, 1),
 A vertex at (0, 1, -1),
 A vertex at (0, 1, 1),
 A vertex at (1, 0, 0))
>>> sorted(G.neighbors_out(C.Vrepresentation()[Integer(4)]))           ␣
↪     # needs sage.graphs
[An inequality (-1, -1, 0) x + 1 >= 0,
 An inequality (-1, 0, -1) x + 1 >= 0,
 An inequality (-1, 0, 1) x + 1 >= 0,
 An inequality (-1, 1, 0) x + 1 >= 0]
```

If `names` is `True` (the default) but the combinatorial polyhedron has been initialized without specifying names to `Vrepresentation` and `Hrepresentation`, then indices of the Vrepresentation and the facets will be used along with a string 'H' or 'V':

```
sage: C = CombinatorialPolyhedron(P.incidence_matrix())
sage: C.vertex_facet_graph().vertices(sort=True)                        #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.graphs
[('H', 0),
 ('H', 1),
 ('H', 2),
 ('H', 3),
 ('H', 4),
 ('V', 0),
 ('V', 1),
 ('V', 2),
 ('V', 3),
 ('V', 4)]
```

```
>>> from sage.all import *
>>> C = CombinatorialPolyhedron(P.incidence_matrix())
>>> C.vertex_facet_graph().vertices(sort=True)                          #␣
↪needs sage.graphs
[('H', 0),
 ('H', 1),
 ('H', 2),
 ('H', 3),
 ('H', 4),
 ('V', 0),
 ('V', 1),
 ('V', 2),
 ('V', 3),
 ('V', 4)]
```

If `names` is `False` then the vertices of the graph are given by integers:

```
sage: C.vertex_facet_graph(names=False).vertices(sort=True)             #␣
↪needs sage.graphs
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
>>> from sage.all import *
>>> C.vertex_facet_graph(names=False).vertices(sort=True)               #␣
↪needs sage.graphs
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

vertex_graph(*names=True*, *algorithm=None*)

Return a graph in which the vertices correspond to vertices of the polyhedron, and edges to bounded rank 1 faces.

INPUT:

- `names` – boolean (default: `True`); if `False`, then the nodes of the graph are labeld by the indices of the Vrepresentation
- `algorithm` – string (optional); specify whether the face generator starts with facets or vertices: * `'primal'` – start with the facets * `'dual'` – start with the vertices * `None` – choose automatically

EXAMPLES:

```
sage: P = polytopes.cyclic_polytope(3,5)
sage: C = CombinatorialPolyhedron(P)
```

(continues on next page)

(continued from previous page)

```
sage: G = C.vertex_graph(); G                                          #␣
↪needs sage.graphs
Graph on 5 vertices
sage: sorted(G.degree())                                               #␣
↪needs sage.graphs
[3, 3, 4, 4, 4]

sage: P = Polyhedron(rays=[[1]])
sage: C = CombinatorialPolyhedron(P)
sage: C.graph()                                                        #␣
↪needs sage.graphs
Graph on 1 vertex
```

```
>>> from sage.all import *
>>> P = polytopes.cyclic_polytope(Integer(3),Integer(5))
>>> C = CombinatorialPolyhedron(P)
>>> G = C.vertex_graph(); G                                          #␣
↪needs sage.graphs
Graph on 5 vertices
>>> sorted(G.degree())                                               #␣
↪needs sage.graphs
[3, 3, 4, 4, 4]

>>> P = Polyhedron(rays=[[Integer(1)]])
>>> C = CombinatorialPolyhedron(P)
>>> C.graph()                                                        #␣
↪needs sage.graphs
Graph on 1 vertex
```

vertices(*names=True*)

Return the elements in the Vrepresentation that are vertices.

In case of an unbounded polyhedron, there might be lines and rays in the Vrepresentation.

If `names` is set to `False`, then the vertices are given by their indices in the Vrepresentation.

EXAMPLES:

```
sage: P = Polyhedron(rays=[[1,0,0],[0,1,0],[0,0,1]])
sage: C = CombinatorialPolyhedron(P)
sage: C.vertices()
(A vertex at (0, 0, 0),)
sage: C.Vrepresentation()
(A vertex at (0, 0, 0),
 A ray in the direction (0, 0, 1),
 A ray in the direction (0, 1, 0),
 A ray in the direction (1, 0, 0))
sage: P = polytopes.cross_polytope(3)
sage: C = CombinatorialPolyhedron(P)
sage: C.vertices()
(A vertex at (-1, 0, 0),
 A vertex at (0, -1, 0),
 A vertex at (0, 0, -1),
```

(continues on next page)

(continued from previous page)

```
 A vertex at (0, 0, 1),
 A vertex at (0, 1, 0),
 A vertex at (1, 0, 0))
sage: C.vertices(names=False)
(0, 1, 2, 3, 4, 5)

sage: points = [(1,0,0), (0,1,0), (0,0,1),
....:           (-1,0,0), (0,-1,0), (0,0,-1)]
sage: L = LatticePolytope(points)
sage: C = CombinatorialPolyhedron(L)
sage: C.vertices()
(M(1, 0, 0), M(0, 1, 0), M(0, 0, 1), M(-1, 0, 0), M(0, -1, 0), M(0, 0, -1))
sage: C.vertices(names=False)
(0, 1, 2, 3, 4, 5)

sage: P = Polyhedron(vertices=[[0,0]])
sage: C = CombinatorialPolyhedron(P)
sage: C.vertices()
(A vertex at (0, 0),)
```

```
>>> from sage.all import *
>>> P = Polyhedron(rays=[[Integer(1),Integer(0),Integer(0)],[Integer(0),
↪Integer(1),Integer(0)],[Integer(0),Integer(0),Integer(1)]])
>>> C = CombinatorialPolyhedron(P)
>>> C.vertices()
(A vertex at (0, 0, 0),)
>>> C.Vrepresentation()
(A vertex at (0, 0, 0),
 A ray in the direction (0, 0, 1),
 A ray in the direction (0, 1, 0),
 A ray in the direction (1, 0, 0))
>>> P = polytopes.cross_polytope(Integer(3))
>>> C = CombinatorialPolyhedron(P)
>>> C.vertices()
(A vertex at (-1, 0, 0),
 A vertex at (0, -1, 0),
 A vertex at (0, 0, -1),
 A vertex at (0, 0, 1),
 A vertex at (0, 1, 0),
 A vertex at (1, 0, 0))
>>> C.vertices(names=False)
(0, 1, 2, 3, 4, 5)

>>> points = [(Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0)), (Integer(0),Integer(0),Integer(1)),
...           (-Integer(1),Integer(0),Integer(0)), (Integer(0),-Integer(1),
↪Integer(0)), (Integer(0),Integer(0),-Integer(1))]
>>> L = LatticePolytope(points)
>>> C = CombinatorialPolyhedron(L)
>>> C.vertices()
(M(1, 0, 0), M(0, 1, 0), M(0, 0, 1), M(-1, 0, 0), M(0, -1, 0), M(0, 0, -1))
>>> C.vertices(names=False)
```

(continues on next page)

(continued from previous page)

```
(0, 1, 2, 3, 4, 5)

>>> P = Polyhedron(vertices=[[Integer(0),Integer(0)]])
>>> C = CombinatorialPolyhedron(P)
>>> C.vertices()
(A vertex at (0, 0),)
```

2.3.2 Combinatorial face of a polyhedron

This module provides the combinatorial type of a polyhedral face.

See also

sage.geometry.polyhedron.combinatorial_polyhedron.base, *sage.geometry.polyhedron.combinatorial_polyhedron.face_iterator*.

EXAMPLES:

Obtain a face from a face iterator:

```
sage: P = polytopes.cube()
sage: C = CombinatorialPolyhedron(P)
sage: it = C.face_generator()
sage: face = next(it); face
A 2-dimensional face of a 3-dimensional combinatorial polyhedron
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> C = CombinatorialPolyhedron(P)
>>> it = C.face_generator()
>>> face = next(it); face
A 2-dimensional face of a 3-dimensional combinatorial polyhedron
```

Obtain a face from a face lattice index:

```
sage: P = polytopes.simplex(2)
sage: C = CombinatorialPolyhedron(P)
sage: sorted(C.face_lattice()._elements)                                    #␣
↪needs sage.combinat
[0, 1, 2, 3, 4, 5, 6, 7]
sage: face = C.face_by_face_lattice_index(0); face
A -1-dimensional face of a 2-dimensional combinatorial polyhedron
```

```
>>> from sage.all import *
>>> P = polytopes.simplex(Integer(2))
>>> C = CombinatorialPolyhedron(P)
>>> sorted(C.face_lattice()._elements)                                    #␣
↪needs sage.combinat
[0, 1, 2, 3, 4, 5, 6, 7]
>>> face = C.face_by_face_lattice_index(Integer(0)); face
A -1-dimensional face of a 2-dimensional combinatorial polyhedron
```

Obtain further information regarding a face:

```
sage: P = polytopes.octahedron()
sage: C = CombinatorialPolyhedron(P)
sage: it = C.face_generator(2)
sage: face = next(it); face
A 2-dimensional face of a 3-dimensional combinatorial polyhedron
sage: face.ambient_Vrepresentation()
(A vertex at (0, 0, 1), A vertex at (0, 1, 0), A vertex at (1, 0, 0))
sage: face.n_ambient_Vrepresentation()
3
sage: face.ambient_H_indices()
(5,)
sage: face.dimension()
2
sage: face.ambient_dimension()
3
```

```
>>> from sage.all import *
>>> P = polytopes.octahedron()
>>> C = CombinatorialPolyhedron(P)
>>> it = C.face_generator(Integer(2))
>>> face = next(it); face
A 2-dimensional face of a 3-dimensional combinatorial polyhedron
>>> face.ambient_Vrepresentation()
(A vertex at (0, 0, 1), A vertex at (0, 1, 0), A vertex at (1, 0, 0))
>>> face.n_ambient_Vrepresentation()
3
>>> face.ambient_H_indices()
(5,)
>>> face.dimension()
2
>>> face.ambient_dimension()
3
```

See also

sage.geometry.polyhedron.combinatorial_polyhedron.base.CombinatorialPolyhedron.

AUTHOR:

- Jonathan Kliem (2019-05)

class sage.geometry.polyhedron.combinatorial_polyhedron.combinatorial_face.**CombinatorialFace**

Bases: SageObject

A class of the combinatorial type of a polyhedral face.

EXAMPLES:

Obtain a combinatorial face from a face iterator:

```
sage: P = polytopes.cyclic_polytope(5,8)
sage: C = CombinatorialPolyhedron(P)
sage: it = C.face_generator()
sage: next(it)
A 0-dimensional face of a 5-dimensional combinatorial polyhedron
```

```
>>> from sage.all import *
>>> P = polytopes.cyclic_polytope(Integer(5),Integer(8))
>>> C = CombinatorialPolyhedron(P)
>>> it = C.face_generator()
>>> next(it)
A 0-dimensional face of a 5-dimensional combinatorial polyhedron
```

Obtain a combinatorial face from an index of the face lattice:

```
sage: F = C.face_lattice()                                                  #␣
↪needs sage.combinat
sage: F._elements[3]                                                        #␣
↪needs sage.combinat
34
sage: C.face_by_face_lattice_index(29)
A 1-dimensional face of a 5-dimensional combinatorial polyhedron
```

```
>>> from sage.all import *
>>> F = C.face_lattice()                                                  #␣
↪needs sage.combinat
>>> F._elements[Integer(3)]                                                 ␣
↪      # needs sage.combinat
34
>>> C.face_by_face_lattice_index(Integer(29))
A 1-dimensional face of a 5-dimensional combinatorial polyhedron
```

Obtain the dimension of a combinatorial face:

```
sage: face = next(it)
sage: face.dimension()
0
```

```
>>> from sage.all import *
>>> face = next(it)
>>> face.dimension()
0
```

The dimension of the polyhedron:

```
sage: face.ambient_dimension()
5
```

```
>>> from sage.all import *
>>> face.ambient_dimension()
5
```

The Vrepresentation:

```
sage: face.ambient_Vrepresentation()
(A vertex at (6, 36, 216, 1296, 7776),)
sage: face.ambient_V_indices()
(6,)
sage: face.n_ambient_Vrepresentation()
1
```

```
>>> from sage.all import *
>>> face.ambient_Vrepresentation()
(A vertex at (6, 36, 216, 1296, 7776),)
>>> face.ambient_V_indices()
(6,)
>>> face.n_ambient_Vrepresentation()
1
```

The Hrepresentation:

```
sage: face.ambient_Hrepresentation()
(An inequality (60, -112, 65, -14, 1) x + 0 >= 0,
 An inequality (180, -216, 91, -16, 1) x + 0 >= 0,
 An inequality (360, -342, 119, -18, 1) x + 0 >= 0,
 An inequality (840, -638, 179, -22, 1) x + 0 >= 0,
 An inequality (-2754, 1175, -245, 25, -1) x + 2520 >= 0,
 An inequality (504, -450, 145, -20, 1) x + 0 >= 0,
 An inequality (-1692, 853, -203, 23, -1) x + 1260 >= 0,
 An inequality (252, -288, 113, -18, 1) x + 0 >= 0,
 An inequality (-844, 567, -163, 21, -1) x + 420 >= 0,
 An inequality (84, -152, 83, -16, 1) x + 0 >= 0,
 An inequality (-210, 317, -125, 19, -1) x + 0 >= 0)
sage: face.ambient_H_indices()
(3, 4, 5, 6, 7, 8, 9, 10, 11, 18, 19)
sage: face.n_ambient_Hrepresentation()
11
```

```
>>> from sage.all import *
>>> face.ambient_Hrepresentation()
(An inequality (60, -112, 65, -14, 1) x + 0 >= 0,
 An inequality (180, -216, 91, -16, 1) x + 0 >= 0,
 An inequality (360, -342, 119, -18, 1) x + 0 >= 0,
 An inequality (840, -638, 179, -22, 1) x + 0 >= 0,
 An inequality (-2754, 1175, -245, 25, -1) x + 2520 >= 0,
 An inequality (504, -450, 145, -20, 1) x + 0 >= 0,
 An inequality (-1692, 853, -203, 23, -1) x + 1260 >= 0,
 An inequality (252, -288, 113, -18, 1) x + 0 >= 0,
 An inequality (-844, 567, -163, 21, -1) x + 420 >= 0,
 An inequality (84, -152, 83, -16, 1) x + 0 >= 0,
 An inequality (-210, 317, -125, 19, -1) x + 0 >= 0)
>>> face.ambient_H_indices()
(3, 4, 5, 6, 7, 8, 9, 10, 11, 18, 19)
>>> face.n_ambient_Hrepresentation()
11
```

ambient_H_indices(*add_equations=True*)

Return the indices of the Hrepresentation objects of the ambient polyhedron defining the face.

INPUT:

- `add_equations` – boolean (default: `True`); whether or not to include the equations

EXAMPLES:

```
sage: # needs sage.combinat
sage: P = polytopes.permutahedron(5)
sage: C = CombinatorialPolyhedron(P)
sage: it = C.face_generator(2)
sage: face = next(it)
sage: face.ambient_H_indices(add_equations=False)
(28, 29)
sage: face2 = next(it)
sage: face2.ambient_H_indices(add_equations=False)
(25, 29)
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> P = polytopes.permutahedron(Integer(5))
>>> C = CombinatorialPolyhedron(P)
>>> it = C.face_generator(Integer(2))
>>> face = next(it)
>>> face.ambient_H_indices(add_equations=False)
(28, 29)
>>> face2 = next(it)
>>> face2.ambient_H_indices(add_equations=False)
(25, 29)
```

Add the indices of the equation:

```
sage: face.ambient_H_indices(add_equations=True)                              #␣
↪needs sage.combinat
(28, 29, 30)
sage: face2.ambient_H_indices(add_equations=True)                             #␣
↪needs sage.combinat
(25, 29, 30)
```

```
>>> from sage.all import *
>>> face.ambient_H_indices(add_equations=True)                              #␣
↪needs sage.combinat
(28, 29, 30)
>>> face2.ambient_H_indices(add_equations=True)                             #␣
↪needs sage.combinat
(25, 29, 30)
```

Another example:

```
sage: P = polytopes.cyclic_polytope(4,6)
sage: C = CombinatorialPolyhedron(P)
sage: it = C.face_generator()
sage: _ = next(it); _ = next(it)
sage: next(it).ambient_H_indices()
```

(continues on next page)

(continued from previous page)

```
(0, 1, 2, 4, 5, 7)
sage: next(it).ambient_H_indices()
(0, 1, 5, 6, 7, 8)
sage: next(it).ambient_H_indices()
(0, 1, 2, 3, 6, 8)
sage: [next(it).dimension() for _ in range(2)]
[0, 1]
sage: face = next(it)
sage: face.ambient_H_indices()
(4, 5, 7)
```

```
>>> from sage.all import *
>>> P = polytopes.cyclic_polytope(Integer(4),Integer(6))
>>> C = CombinatorialPolyhedron(P)
>>> it = C.face_generator()
>>> _ = next(it); _ = next(it)
>>> next(it).ambient_H_indices()
(0, 1, 2, 4, 5, 7)
>>> next(it).ambient_H_indices()
(0, 1, 5, 6, 7, 8)
>>> next(it).ambient_H_indices()
(0, 1, 2, 3, 6, 8)
>>> [next(it).dimension() for _ in range(Integer(2))]
[0, 1]
>>> face = next(it)
>>> face.ambient_H_indices()
(4, 5, 7)
```

See also

ambient_Hrepresentation().

ambient_Hrepresentation()

Return the Hrepresentation objects of the ambient polyhedron defining the face.

It consists of the facets/inequalities that contain the face and the equations defining the ambient polyhedron.

EXAMPLES:

```
sage: # needs sage.combinat
sage: P = polytopes.permutahedron(5)
sage: C = CombinatorialPolyhedron(P)
sage: it = C.face_generator(2)
sage: next(it).ambient_Hrepresentation()
(An inequality (1, 1, 1, 0, 0) x - 6 >= 0,
 An inequality (0, 0, 0, -1, 0) x + 5 >= 0,
 An equation (1, 1, 1, 1, 1) x - 15 == 0)
sage: next(it).ambient_Hrepresentation()
(An inequality (0, 0, -1, -1, 0) x + 9 >= 0,
 An inequality (0, 0, 0, -1, 0) x + 5 >= 0,
 An equation (1, 1, 1, 1, 1) x - 15 == 0)
```

(continues on next page)

(continued from previous page)

```
sage: P = polytopes.cyclic_polytope(4,6)
sage: C = CombinatorialPolyhedron(P)
sage: it = C.face_generator()
sage: next(it).ambient_Hrepresentation()
(An inequality (-20, 29, -10, 1) x + 0 >= 0,
 An inequality (60, -47, 12, -1) x + 0 >= 0,
 An inequality (30, -31, 10, -1) x + 0 >= 0,
 An inequality (10, -17, 8, -1) x + 0 >= 0,
 An inequality (-154, 71, -14, 1) x + 120 >= 0,
 An inequality (-78, 49, -12, 1) x + 40 >= 0)
sage: next(it).ambient_Hrepresentation()
(An inequality (-50, 35, -10, 1) x + 24 >= 0,
 An inequality (-12, 19, -8, 1) x + 0 >= 0,
 An inequality (-20, 29, -10, 1) x + 0 >= 0,
 An inequality (60, -47, 12, -1) x + 0 >= 0,
 An inequality (-154, 71, -14, 1) x + 120 >= 0,
 An inequality (-78, 49, -12, 1) x + 40 >= 0)
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> P = polytopes.permutahedron(Integer(5))
>>> C = CombinatorialPolyhedron(P)
>>> it = C.face_generator(Integer(2))
>>> next(it).ambient_Hrepresentation()
(An inequality (1, 1, 1, 0, 0) x - 6 >= 0,
 An inequality (0, 0, 0, -1, 0) x + 5 >= 0,
 An equation (1, 1, 1, 1, 1) x - 15 == 0)
>>> next(it).ambient_Hrepresentation()
(An inequality (0, 0, -1, -1, 0) x + 9 >= 0,
 An inequality (0, 0, 0, -1, 0) x + 5 >= 0,
 An equation (1, 1, 1, 1, 1) x - 15 == 0)

>>> P = polytopes.cyclic_polytope(Integer(4),Integer(6))
>>> C = CombinatorialPolyhedron(P)
>>> it = C.face_generator()
>>> next(it).ambient_Hrepresentation()
(An inequality (-20, 29, -10, 1) x + 0 >= 0,
 An inequality (60, -47, 12, -1) x + 0 >= 0,
 An inequality (30, -31, 10, -1) x + 0 >= 0,
 An inequality (10, -17, 8, -1) x + 0 >= 0,
 An inequality (-154, 71, -14, 1) x + 120 >= 0,
 An inequality (-78, 49, -12, 1) x + 40 >= 0)
>>> next(it).ambient_Hrepresentation()
(An inequality (-50, 35, -10, 1) x + 24 >= 0,
 An inequality (-12, 19, -8, 1) x + 0 >= 0,
 An inequality (-20, 29, -10, 1) x + 0 >= 0,
 An inequality (60, -47, 12, -1) x + 0 >= 0,
 An inequality (-154, 71, -14, 1) x + 120 >= 0,
 An inequality (-78, 49, -12, 1) x + 40 >= 0)
```

See also

ambient_H_indices().

ambient_V_indices()

Return the indices of the Vrepresentation objects of the ambient polyhedron defining the face.

EXAMPLES:

```
sage: # needs sage.combinat
sage: P = polytopes.permutahedron(5)
sage: C = CombinatorialPolyhedron(P)
sage: it = C.face_generator(dimension=2)
sage: face = next(it)
sage: next(it).ambient_V_indices()
(32, 91, 92, 93, 94, 95)
sage: next(it).ambient_V_indices()
(32, 89, 90, 94)

sage: C = CombinatorialPolyhedron([[0,1,2],[0,1,3],[0,2,3],[1,2,3]])
sage: it = C.face_generator()
sage: for face in it: (face.dimension(), face.ambient_V_indices())
(2, (1, 2, 3))
(2, (0, 2, 3))
(2, (0, 1, 3))
(2, (0, 1, 2))
(1, (2, 3))
(1, (1, 3))
(1, (1, 2))
(0, (3,))
(0, (2,))
(0, (1,))
(1, (0, 3))
(1, (0, 2))
(0, (0,))
(1, (0, 1))
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> P = polytopes.permutahedron(Integer(5))
>>> C = CombinatorialPolyhedron(P)
>>> it = C.face_generator(dimension=Integer(2))
>>> face = next(it)
>>> next(it).ambient_V_indices()
(32, 91, 92, 93, 94, 95)
>>> next(it).ambient_V_indices()
(32, 89, 90, 94)

>>> C = CombinatorialPolyhedron([[Integer(0),Integer(1),Integer(2)],
↪[Integer(0),Integer(1),Integer(3)],[Integer(0),Integer(2),Integer(3)],
↪[Integer(1),Integer(2),Integer(3)]])
>>> it = C.face_generator()
>>> for face in it: (face.dimension(), face.ambient_V_indices())
```

(continues on next page)

(continued from previous page)

```
(2, (1, 2, 3))
(2, (0, 2, 3))
(2, (0, 1, 3))
(2, (0, 1, 2))
(1, (2, 3))
(1, (1, 3))
(1, (1, 2))
(0, (3,))
(0, (2,))
(0, (1,))
(1, (0, 3))
(1, (0, 2))
(0, (0,))
(1, (0, 1))
```

See also:

ambient_Vrepresentation().

ambient_Vrepresentation()

Return the Vrepresentation objects of the ambient polyhedron defining the face.

It consists of the vertices/rays/lines that face contains.

EXAMPLES:

```
sage: # needs sage.combinat
sage: P = polytopes.permutahedron(5)
sage: C = CombinatorialPolyhedron(P)
sage: it = C.face_generator(dimension=2)
sage: face = next(it)
sage: face.ambient_Vrepresentation()
(A vertex at (1, 3, 2, 5, 4),
 A vertex at (2, 3, 1, 5, 4),
 A vertex at (3, 1, 2, 5, 4),
 A vertex at (3, 2, 1, 5, 4),
 A vertex at (2, 1, 3, 5, 4),
 A vertex at (1, 2, 3, 5, 4))
sage: face = next(it)
sage: face.ambient_Vrepresentation()
(A vertex at (2, 1, 4, 5, 3),
 A vertex at (3, 2, 4, 5, 1),
 A vertex at (3, 1, 4, 5, 2),
 A vertex at (1, 3, 4, 5, 2),
 A vertex at (1, 2, 4, 5, 3),
 A vertex at (2, 3, 4, 5, 1))

sage: C = CombinatorialPolyhedron([[0,1,2],[0,1,3],[0,2,3],[1,2,3]])
sage: it = C.face_generator()
sage: for face in it: (face.dimension(), face.ambient_Vrepresentation())
(2, (1, 2, 3))
(2, (0, 2, 3))
```

(continues on next page)

(continued from previous page)

```
(2, (0, 1, 3))
(2, (0, 1, 2))
(1, (2, 3))
(1, (1, 3))
(1, (1, 2))
(0, (3,))
(0, (2,))
(0, (1,))
(1, (0, 3))
(1, (0, 2))
(0, (0,))
(1, (0, 1))
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> P = polytopes.permutahedron(Integer(5))
>>> C = CombinatorialPolyhedron(P)
>>> it = C.face_generator(dimension=Integer(2))
>>> face = next(it)
>>> face.ambient_Vrepresentation()
(A vertex at (1, 3, 2, 5, 4),
 A vertex at (2, 3, 1, 5, 4),
 A vertex at (3, 1, 2, 5, 4),
 A vertex at (3, 2, 1, 5, 4),
 A vertex at (2, 1, 3, 5, 4),
 A vertex at (1, 2, 3, 5, 4))
>>> face = next(it)
>>> face.ambient_Vrepresentation()
(A vertex at (2, 1, 4, 5, 3),
 A vertex at (3, 2, 4, 5, 1),
 A vertex at (3, 1, 4, 5, 2),
 A vertex at (1, 3, 4, 5, 2),
 A vertex at (1, 2, 4, 5, 3),
 A vertex at (2, 3, 4, 5, 1))

>>> C = CombinatorialPolyhedron([[Integer(0),Integer(1),Integer(2)],
↪[Integer(0),Integer(1),Integer(3)],[Integer(0),Integer(2),Integer(3)],
↪[Integer(1),Integer(2),Integer(3)]])
>>> it = C.face_generator()
>>> for face in it: (face.dimension(), face.ambient_Vrepresentation())
(2, (1, 2, 3))
(2, (0, 2, 3))
(2, (0, 1, 3))
(2, (0, 1, 2))
(1, (2, 3))
(1, (1, 3))
(1, (1, 2))
(0, (3,))
(0, (2,))
(0, (1,))
(1, (0, 3))
(1, (0, 2))
```

(continues on next page)

(continued from previous page)

```
(0, (0,))
(1, (0, 1))
```

See also

ambient_V_indices().

ambient_dimension()

Return the dimension of the polyhedron.

EXAMPLES:

```
sage: P = polytopes.cube()
sage: C = CombinatorialPolyhedron(P)
sage: it = C.face_generator()
sage: face = next(it)
sage: face.ambient_dimension()
3
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> C = CombinatorialPolyhedron(P)
>>> it = C.face_generator()
>>> face = next(it)
>>> face.ambient_dimension()
3
```

as_combinatorial_polyhedron(*quotient=False*)

Return `self` as combinatorial polyhedron.

If `quotient` is `True`, return the quotient of the polyhedron by `self`. Let `G` be the face corresponding to `self` in the dual/polar polytope. The `quotient` is the dual/polar of `G`.

Let $[\hat{0}, \hat{1}]$ be the face lattice of the ambient polyhedron and F be `self` as element of the face lattice. The face lattice of `self` as polyhedron corresponds to $[\hat{0}, F]$ and the face lattice of the quotient by `self` corresponds to $[F, \hat{1}]$.

EXAMPLES:

```
sage: P = polytopes.cyclic_polytope(7,11)
sage: C = CombinatorialPolyhedron(P)
sage: it = C.face_generator(4)
sage: f = next(it); f
A 4-dimensional face of a 7-dimensional combinatorial polyhedron
sage: F = f.as_combinatorial_polyhedron(); F
A 4-dimensional combinatorial polyhedron with 5 facets
sage: F.f_vector()
(1, 5, 10, 10, 5, 1)
sage: F_alt = polytopes.cyclic_polytope(4,5).combinatorial_polyhedron()
sage: F_alt.vertex_facet_graph().is_isomorphic(F.vertex_facet_graph())       #␣
↪needs sage.graphs
True
```

```
>>> from sage.all import *
>>> P = polytopes.cyclic_polytope(Integer(7),Integer(11))
>>> C = CombinatorialPolyhedron(P)
>>> it = C.face_generator(Integer(4))
>>> f = next(it); f
A 4-dimensional face of a 7-dimensional combinatorial polyhedron
>>> F = f.as_combinatorial_polyhedron(); F
A 4-dimensional combinatorial polyhedron with 5 facets
>>> F.f_vector()
(1, 5, 10, 10, 5, 1)
>>> F_alt = polytopes.cyclic_polytope(Integer(4),Integer(5)).combinatorial_
↪polyhedron()
>>> F_alt.vertex_facet_graph().is_isomorphic(F.vertex_facet_graph())       #␣
↪needs sage.graphs
True
```

Obtaining the quotient:

```
sage: Q = f.as_combinatorial_polyhedron(quotient=True); Q
A 2-dimensional combinatorial polyhedron with 6 facets
sage: Q
A 2-dimensional combinatorial polyhedron with 6 facets
sage: Q.f_vector()
(1, 6, 6, 1)
```

```
>>> from sage.all import *
>>> Q = f.as_combinatorial_polyhedron(quotient=True); Q
A 2-dimensional combinatorial polyhedron with 6 facets
>>> Q
A 2-dimensional combinatorial polyhedron with 6 facets
>>> Q.f_vector()
(1, 6, 6, 1)
```

The Vrepresentation of the face as polyhedron is given by the ambient Vrepresentation of the face in that order:

```
sage: P = polytopes.cube()
sage: C = CombinatorialPolyhedron(P)
sage: it = C.face_generator(2)
sage: f = next(it)
sage: F = f.as_combinatorial_polyhedron()
sage: C.Vrepresentation()
(A vertex at (1, -1, -1),
A vertex at (1, 1, -1),
A vertex at (1, 1, 1),
A vertex at (1, -1, 1),
A vertex at (-1, -1, 1),
A vertex at (-1, -1, -1),
A vertex at (-1, 1, -1),
A vertex at (-1, 1, 1))
sage: f.ambient_Vrepresentation()
(A vertex at (1, -1, -1),
A vertex at (1, -1, 1),
```

(continues on next page)

(continued from previous page)

```
A vertex at (-1, -1, 1),
A vertex at (-1, -1, -1))
sage: F.Vrepresentation()
(0, 1, 2, 3)
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> C = CombinatorialPolyhedron(P)
>>> it = C.face_generator(Integer(2))
>>> f = next(it)
>>> F = f.as_combinatorial_polyhedron()
>>> C.Vrepresentation()
(A vertex at (1, -1, -1),
A vertex at (1, 1, -1),
A vertex at (1, 1, 1),
A vertex at (1, -1, 1),
A vertex at (-1, -1, 1),
A vertex at (-1, -1, -1),
A vertex at (-1, 1, -1),
A vertex at (-1, 1, 1))
>>> f.ambient_Vrepresentation()
(A vertex at (1, -1, -1),
A vertex at (1, -1, 1),
A vertex at (-1, -1, 1),
A vertex at (-1, -1, -1))
>>> F.Vrepresentation()
(0, 1, 2, 3)
```

To obtain the facets of the face as polyhedron, we compute the meet of each facet with the face. The first representative of each element strictly contained in the face is kept:

```
sage: C.facets(names=False)
((0, 1, 2, 3),
 (1, 2, 6, 7),
 (2, 3, 4, 7),
 (4, 5, 6, 7),
 (0, 1, 5, 6),
 (0, 3, 4, 5))
sage: F.facets(names=False)
((0, 1), (1, 2), (2, 3), (0, 3))
```

```
>>> from sage.all import *
>>> C.facets(names=False)
((0, 1, 2, 3),
 (1, 2, 6, 7),
 (2, 3, 4, 7),
 (4, 5, 6, 7),
 (0, 1, 5, 6),
 (0, 3, 4, 5))
>>> F.facets(names=False)
((0, 1), (1, 2), (2, 3), (0, 3))
```

The Hrepresentation of the quotient by the face is given by the ambient Hrepresentation of the face in that order:

```
sage: it = C.face_generator(1)
sage: f = next(it)
sage: Q = f.as_combinatorial_polyhedron(quotient=True)
sage: C.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, 0) x + 1 >= 0,
An inequality (0, 0, 1) x + 1 >= 0,
An inequality (0, 1, 0) x + 1 >= 0)
sage: f.ambient_Hrepresentation()
(An inequality (0, 0, 1) x + 1 >= 0, An inequality (0, 1, 0) x + 1 >= 0)
sage: Q.Hrepresentation()
(0, 1)
```

```
>>> from sage.all import *
>>> it = C.face_generator(Integer(1))
>>> f = next(it)
>>> Q = f.as_combinatorial_polyhedron(quotient=True)
>>> C.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, 0) x + 1 >= 0,
An inequality (0, 0, 1) x + 1 >= 0,
An inequality (0, 1, 0) x + 1 >= 0)
>>> f.ambient_Hrepresentation()
(An inequality (0, 0, 1) x + 1 >= 0, An inequality (0, 1, 0) x + 1 >= 0)
>>> Q.Hrepresentation()
(0, 1)
```

To obtain the vertices of the face as polyhedron, we compute the join of each vertex with the face. The first representative of each element strictly containing the face is kept:

```
sage: [g.ambient_H_indices() for g in C.face_generator(0)]
[(3, 4, 5),
(0, 4, 5),
(2, 3, 5),
(0, 2, 5),
(1, 3, 4),
(0, 1, 4),
(1, 2, 3),
(0, 1, 2)]
sage: [g.ambient_H_indices() for g in Q.face_generator(0)]
[(1,), (0,)]
```

```
>>> from sage.all import *
>>> [g.ambient_H_indices() for g in C.face_generator(Integer(0))]
[(3, 4, 5),
(0, 4, 5),
```

(continues on next page)

(continued from previous page)

```
(2, 3, 5),
(0, 2, 5),
(1, 3, 4),
(0, 1, 4),
(1, 2, 3),
(0, 1, 2)]
>>> [g.ambient_H_indices() for g in Q.face_generator(Integer(0))]
[(1,), (0,)]
```

The method is not implemented for unbounded polyhedra:

```
sage: P = Polyhedron(rays=[[0,1]])*polytopes.cube()
sage: C = CombinatorialPolyhedron(P)
sage: it = C.face_generator(2)
sage: f = next(it)
sage: f.as_combinatorial_polyhedron()
Traceback (most recent call last):
...
NotImplementedError: only implemented for bounded polyhedra
```

```
>>> from sage.all import *
>>> P = Polyhedron(rays=[[Integer(0),Integer(1)]])*polytopes.cube()
>>> C = CombinatorialPolyhedron(P)
>>> it = C.face_generator(Integer(2))
>>> f = next(it)
>>> f.as_combinatorial_polyhedron()
Traceback (most recent call last):
...
NotImplementedError: only implemented for bounded polyhedra
```

REFERENCES:

For more information, see Exercise 2.9 of [Zie2007].

Note

This method is tested in `_test_combinatorial_face_as_combinatorial_polyhedron()`.

dim()

Return the dimension of the face.

EXAMPLES:

```
sage: # needs sage.combinat
sage: P = polytopes.associahedron(['A', 3])
sage: C = CombinatorialPolyhedron(P)
sage: it = C.face_generator()
sage: face = next(it)
sage: face.dimension()
2
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> P = polytopes.associahedron(['A', Integer(3)])
>>> C = CombinatorialPolyhedron(P)
>>> it = C.face_generator()
>>> face = next(it)
>>> face.dimension()
2
```

`dim` is an alias:

```
sage: face.dim()                                                              #␣
↪needs sage.combinat
2
```

```
>>> from sage.all import *
>>> face.dim()                                                              #␣
↪needs sage.combinat
2
```

dimension()

Return the dimension of the face.

EXAMPLES:

```
sage: # needs sage.combinat
sage: P = polytopes.associahedron(['A', 3])
sage: C = CombinatorialPolyhedron(P)
sage: it = C.face_generator()
sage: face = next(it)
sage: face.dimension()
2
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> P = polytopes.associahedron(['A', Integer(3)])
>>> C = CombinatorialPolyhedron(P)
>>> it = C.face_generator()
>>> face = next(it)
>>> face.dimension()
2
```

`dim` is an alias:

```
sage: face.dim()                                                              #␣
↪needs sage.combinat
2
```

```
>>> from sage.all import *
>>> face.dim()                                                              #␣
↪needs sage.combinat
2
```

is_subface(*other*)

Return whether `self` is contained in `other`.

EXAMPLES:

```
sage: P = polytopes.cube()
sage: C = P.combinatorial_polyhedron()
sage: it = C.face_generator()
sage: face = next(it)
sage: face.ambient_V_indices()
(0, 3, 4, 5)
sage: face2 = next(it)
sage: face2.ambient_V_indices()
(0, 1, 5, 6)
sage: face.is_subface(face2)
False
sage: face2.is_subface(face)
False
sage: it.only_subfaces()
sage: face3 = next(it)
sage: face3.ambient_V_indices()
(0, 5)
sage: face3.is_subface(face2)
True
sage: face3.is_subface(face)
True
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> C = P.combinatorial_polyhedron()
>>> it = C.face_generator()
>>> face = next(it)
>>> face.ambient_V_indices()
(0, 3, 4, 5)
>>> face2 = next(it)
>>> face2.ambient_V_indices()
(0, 1, 5, 6)
>>> face.is_subface(face2)
False
>>> face2.is_subface(face)
False
>>> it.only_subfaces()
>>> face3 = next(it)
>>> face3.ambient_V_indices()
(0, 5)
>>> face3.is_subface(face2)
True
>>> face3.is_subface(face)
True
```

Works for faces of the same combinatorial polyhedron; also from different iterators:

```
sage: it = C.face_generator(algorithm='dual')
sage: v7 = next(it); v7.ambient_V_indices()
```

(continues on next page)

(continued from previous page)

```
(7,)
sage: v6 = next(it); v6.ambient_V_indices()
(6,)
sage: v5 = next(it); v5.ambient_V_indices()
(5,)
sage: face.ambient_V_indices()
(0, 3, 4, 5)
sage: face.is_subface(v7)
False
sage: v7.is_subface(face)
False
sage: v6.is_subface(face)
False
sage: v5.is_subface(face)
True
sage: face2.ambient_V_indices()
(0, 1, 5, 6)
sage: face2.is_subface(v7)
False
sage: v7.is_subface(face2)
False
sage: v6.is_subface(face2)
True
sage: v5.is_subface(face2)
True
```

```
>>> from sage.all import *
>>> it = C.face_generator(algorithm='dual')
>>> v7 = next(it); v7.ambient_V_indices()
(7,)
>>> v6 = next(it); v6.ambient_V_indices()
(6,)
>>> v5 = next(it); v5.ambient_V_indices()
(5,)
>>> face.ambient_V_indices()
(0, 3, 4, 5)
>>> face.is_subface(v7)
False
>>> v7.is_subface(face)
False
>>> v6.is_subface(face)
False
>>> v5.is_subface(face)
True
>>> face2.ambient_V_indices()
(0, 1, 5, 6)
>>> face2.is_subface(v7)
False
>>> v7.is_subface(face2)
False
>>> v6.is_subface(face2)
True
```

(continues on next page)

(continued from previous page)

```
>>> v5.is_subface(face2)
True
```

Only implemented for faces of the same combinatorial polyhedron:

```
sage: P1 = polytopes.cube()
sage: C1 = P1.combinatorial_polyhedron()
sage: it = C1.face_generator()
sage: other_face = next(it)
sage: other_face.ambient_V_indices()
(0, 3, 4, 5)
sage: face.ambient_V_indices()
(0, 3, 4, 5)
sage: C is C1
False
sage: face.is_subface(other_face)
Traceback (most recent call last):
...
NotImplementedError: is_subface only implemented for faces of the same␣
↪polyhedron
```

```
>>> from sage.all import *
>>> P1 = polytopes.cube()
>>> C1 = P1.combinatorial_polyhedron()
>>> it = C1.face_generator()
>>> other_face = next(it)
>>> other_face.ambient_V_indices()
(0, 3, 4, 5)
>>> face.ambient_V_indices()
(0, 3, 4, 5)
>>> C is C1
False
>>> face.is_subface(other_face)
Traceback (most recent call last):
...
NotImplementedError: is_subface only implemented for faces of the same␣
↪polyhedron
```

n_ambient_Hrepresentation(*add_equations=True*)

Return the length of the `CombinatorialFace.ambient_H_indices()`.

Might be faster than then using `len`.

INPUT:

- `add_equations` – boolean (default: `True`); whether or not to count the equations

EXAMPLES:

```
sage: P = polytopes.cube()
sage: C = CombinatorialPolyhedron(P)
sage: it = C.face_generator()
sage: all(face.n_ambient_Hrepresentation() == len(face.ambient_
```

(continues on next page)

(continued from previous page)

```
↪Hrepresentation()) for face in it)
True
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> C = CombinatorialPolyhedron(P)
>>> it = C.face_generator()
>>> all(face.n_ambient_Hrepresentation() == len(face.ambient_
↪Hrepresentation()) for face in it)
True
```

Specifying whether to count the equations or not:

```
sage: # needs sage.combinat
sage: P = polytopes.permutahedron(5)
sage: C = CombinatorialPolyhedron(P)
sage: it = C.face_generator(2)
sage: f = next(it)
sage: f.n_ambient_Hrepresentation(add_equations=True)
3
sage: f.n_ambient_Hrepresentation(add_equations=False)
2
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> P = polytopes.permutahedron(Integer(5))
>>> C = CombinatorialPolyhedron(P)
>>> it = C.face_generator(Integer(2))
>>> f = next(it)
>>> f.n_ambient_Hrepresentation(add_equations=True)
3
>>> f.n_ambient_Hrepresentation(add_equations=False)
2
```

n_ambient_Vrepresentation()

Return the length of the `CombinatorialFace.ambient_V_indices()`.

Might be faster than using `len`.

EXAMPLES:

```
sage: P = polytopes.cube()
sage: C = CombinatorialPolyhedron(P)
sage: it = C.face_generator()
sage: all(face.n_ambient_Vrepresentation() == len(face.ambient_
↪Vrepresentation()) for face in it)
True
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> C = CombinatorialPolyhedron(P)
>>> it = C.face_generator()
```

(continues on next page)

(continued from previous page)

```
>>> all(face.n_ambient_Vrepresentation() == len(face.ambient_
↪Vrepresentation()) for face in it)
True
```

2.3.3 PolyhedronFaceLattice

This module provides a class that stores and sorts all faces of the polyhedron.

CombinatorialPolyhedron implicitly uses this class to generate the face lattice of a polyhedron.

Terminology in this module:

- Vrep – `[vertices, rays, lines]` of the polyhedron
- Hrep – inequalities and equations of the polyhedron
- Facets – facets of the polyhedron
- Coatoms – the faces from which all others are constructed in the face iterator. This will be facets or Vrep. In non-dual mode, faces are constructed as intersections of the facets. In dual mode, the are constructed theoretically as joins of vertices. The coatoms are repsented as incidences with the atoms they contain.
- Atoms – facets or Vrep depending on application of algorithm. Atoms are repsented as incidences of coatoms they are contained in.
- Vrepresentation – represents a face by a list of Vrep it contains
- Hrepresentation – represents a face by a list of Hrep it is contained in
- bit representation – represents incidences as `uint64_t`-array, where each bit represents one incidence. There might be trailing zeros, to fit alignment requirements. In most instances, faces are represented by the bit representation, where each bit corresponds to an atom.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.combinatorial_polyhedron.polyhedron_face_lattice \
....: import PolyhedronFaceLattice
sage: P = polytopes.octahedron()
sage: C = CombinatorialPolyhedron(P)
sage: all_faces = PolyhedronFaceLattice(C)
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.combinatorial_polyhedron.polyhedron_face_lattice␣
↪import PolyhedronFaceLattice
>>> P = polytopes.octahedron()
>>> C = CombinatorialPolyhedron(P)
>>> all_faces = PolyhedronFaceLattice(C)
```

See also

base, *PolyhedronFaceLattice*.

AUTHOR:

- Jonathan Kliem (2019-04)

class sage.geometry.polyhedron.combinatorial_polyhedron.polyhedron_face_lattice.**PolyhedronFaceLattice**

Bases: object

A class to generate incidences of *CombinatorialPolyhedron*.

On initialization all faces of the given *CombinatorialPolyhedron* are added and sorted (except coatoms). The incidences can be used to generate the face_lattice.

Might generate the faces of the dual polyhedron for speed.

INPUT:

- baseCombinatorialPolyhedron

See also

_record_all_faces(), _record_all_faces_helper(), *face_lattice()*, _compute_face_lattice_incidences().

EXAMPLES:

```
sage: P = polytopes.Birkhoff_polytope(3)
sage: C = CombinatorialPolyhedron(P)
sage: C._record_all_faces()   # indirect doctests
sage: C.face_lattice()                                                      #␣
↪needs sage.combinat
Finite lattice containing 50 elements
```

```
>>> from sage.all import *
>>> P = polytopes.Birkhoff_polytope(Integer(3))
>>> C = CombinatorialPolyhedron(P)
>>> C._record_all_faces()   # indirect doctests
>>> C.face_lattice()                                                      #␣
↪needs sage.combinat
Finite lattice containing 50 elements
```

ALGORITHM:

The faces are recorded with *FaceIterator* in Bit-representation. Once created, all level-sets but the coatoms are sorted with merge sort. Non-trivial incidences of elements whose rank differs by 1 are determined by intersecting with all coatoms. Then each intersection is looked up in the sorted level sets.

dual

get_face(*dimension*, *index*)

Return the face of dimension dimension and index index.

INPUT:

- dimension – dimension of the face
- index – index of the face
- names – if True returns the names of the [vertices, rays, lines] as given on initialization of *CombinatorialPolyhedron*

EXAMPLES:

```
sage: from sage.geometry.polyhedron.combinatorial_polyhedron.polyhedron_face_
↪lattice \
....:         import PolyhedronFaceLattice
sage: P = polytopes.permutahedron(4)
sage: C = CombinatorialPolyhedron(P)
sage: F = PolyhedronFaceLattice(C)
sage: it = C.face_generator(dimension=1)
sage: face = next(it)
sage: index = F._find_face_from_combinatorial_face(face)
sage: F.get_face(face.dimension(), index).ambient_Vrepresentation()
(A vertex at (2, 1, 4, 3), A vertex at (1, 2, 4, 3))
sage: face.ambient_Vrepresentation()
(A vertex at (2, 1, 4, 3), A vertex at (1, 2, 4, 3))
sage: all(F.get_face(face.dimension(),
....:                F._find_face_from_combinatorial_face(face)).ambient_
↪Vrepresentation() ==
....:     face.ambient_Vrepresentation() for face in it)
True

sage: P = polytopes.twenty_four_cell()
sage: C = CombinatorialPolyhedron(P)
sage: F = PolyhedronFaceLattice(C)
sage: it = C.face_generator()
sage: face = next(it)
sage: while (face.dimension() == 3): face = next(it)
sage: index = F._find_face_from_combinatorial_face(face)
sage: F.get_face(face.dimension(), index).ambient_Vrepresentation()
(A vertex at (-1/2, 1/2, -1/2, -1/2),
 A vertex at (-1/2, 1/2, 1/2, -1/2),
 A vertex at (0, 0, 0, -1))
sage: all(F.get_face(face.dimension(),
....:                F._find_face_from_combinatorial_face(face)).ambient_V_
↪indices() ==
....:     face.ambient_V_indices() for face in it)
True
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.combinatorial_polyhedron.polyhedron_face_
↪lattice         import PolyhedronFaceLattice
>>> P = polytopes.permutahedron(Integer(4))
>>> C = CombinatorialPolyhedron(P)
>>> F = PolyhedronFaceLattice(C)
>>> it = C.face_generator(dimension=Integer(1))
>>> face = next(it)
>>> index = F._find_face_from_combinatorial_face(face)
>>> F.get_face(face.dimension(), index).ambient_Vrepresentation()
(A vertex at (2, 1, 4, 3), A vertex at (1, 2, 4, 3))
>>> face.ambient_Vrepresentation()
(A vertex at (2, 1, 4, 3), A vertex at (1, 2, 4, 3))
>>> all(F.get_face(face.dimension(),
...                F._find_face_from_combinatorial_face(face)).ambient_
↪Vrepresentation() ==
```

(continues on next page)

(continued from previous page)

```
...     face.ambient_Vrepresentation() for face in it)
True

>>> P = polytopes.twenty_four_cell()
>>> C = CombinatorialPolyhedron(P)
>>> F = PolyhedronFaceLattice(C)
>>> it = C.face_generator()
>>> face = next(it)
>>> while (face.dimension() == Integer(3)): face = next(it)
>>> index = F._find_face_from_combinatorial_face(face)
>>> F.get_face(face.dimension(), index).ambient_Vrepresentation()
(A vertex at (-1/2, 1/2, -1/2, -1/2),
 A vertex at (-1/2, 1/2, 1/2, -1/2),
 A vertex at (0, 0, 0, -1))
>>> all(F.get_face(face.dimension(),
...                 F._find_face_from_combinatorial_face(face)).ambient_V_
↪indices() ==
...     face.ambient_V_indices() for face in it)
True
```

2.3.4 Face iterator for polyhedra

This iterator in principle works on every graded lattice, where every interval of length two has exactly 4 elements (diamond property).

It also works on unbounded polyhedra, as those satisfy the diamond property, except for intervals including the empty face. A (slightly generalized) description of the algorithm can be found in [KS2019].

Terminology in this module:

- Coatoms – the faces from which all others are constructed in the face iterator. This will be facets or Vrep. In non-dual mode, faces are constructed as intersections of the facets. In dual mode, they are constructed theoretically as joins of vertices. The coatoms are repsented as incidences with the atoms they contain.
- Atoms – facets or Vrep depending on application of algorithm. Atoms are represented as incidences of coatoms they are contained in.

See also

sage.geometry.polyhedron.combinatorial_polyhedron.base.

EXAMPLES:

Construct a face iterator:

```
sage: from sage.geometry.polyhedron.combinatorial_polyhedron.face_iterator \
....:         import FaceIterator
sage: P = polytopes.octahedron()
sage: C = CombinatorialPolyhedron(P)

sage: FaceIterator(C, False)
Iterator over the proper faces of a 3-dimensional combinatorial polyhedron
sage: FaceIterator(C, False, output_dimension=2)
Iterator over the 2-faces of a 3-dimensional combinatorial polyhedron
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.combinatorial_polyhedron.face_iterator␣
↪import FaceIterator
>>> P = polytopes.octahedron()
>>> C = CombinatorialPolyhedron(P)

>>> FaceIterator(C, False)
Iterator over the proper faces of a 3-dimensional combinatorial polyhedron
>>> FaceIterator(C, False, output_dimension=Integer(2))
Iterator over the 2-faces of a 3-dimensional combinatorial polyhedron
```

Iterator in the non-dual mode starts with facets:

```
sage: it = FaceIterator(C, False)
sage: [next(it) for _ in range(9)]
[A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 1-dimensional face of a 3-dimensional combinatorial polyhedron]
```

```
>>> from sage.all import *
>>> it = FaceIterator(C, False)
>>> [next(it) for _ in range(Integer(9))]
[A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 1-dimensional face of a 3-dimensional combinatorial polyhedron]
```

Iterator in the dual-mode starts with vertices:

```
sage: it = FaceIterator(C, True)
sage: [next(it) for _ in range(7)]
[A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 1-dimensional face of a 3-dimensional combinatorial polyhedron]
```

```
>>> from sage.all import *
>>> it = FaceIterator(C, True)
>>> [next(it) for _ in range(Integer(7))]
```

(continues on next page)

(continued from previous page)

```
[A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 1-dimensional face of a 3-dimensional combinatorial polyhedron]
```

Obtain the Vrepresentation:

```
sage: it = FaceIterator(C, False)
sage: face = next(it)
sage: face.ambient_Vrepresentation()
(A vertex at (0, -1, 0), A vertex at (0, 0, -1), A vertex at (1, 0, 0))
sage: face.n_ambient_Vrepresentation()
3
```

```
>>> from sage.all import *
>>> it = FaceIterator(C, False)
>>> face = next(it)
>>> face.ambient_Vrepresentation()
(A vertex at (0, -1, 0), A vertex at (0, 0, -1), A vertex at (1, 0, 0))
>>> face.n_ambient_Vrepresentation()
3
```

Obtain the facet-representation:

```
sage: it = FaceIterator(C, True)
sage: face = next(it)
sage: face.ambient_Hrepresentation()
(An inequality (-1, -1, 1) x + 1 >= 0,
 An inequality (-1, -1, -1) x + 1 >= 0,
  An inequality (-1, 1, -1) x + 1 >= 0,
   An inequality (-1, 1, 1) x + 1 >= 0)
sage: face.ambient_H_indices()
(4, 5, 6, 7)
sage: face.n_ambient_Hrepresentation()
4
```

```
>>> from sage.all import *
>>> it = FaceIterator(C, True)
>>> face = next(it)
>>> face.ambient_Hrepresentation()
(An inequality (-1, -1, 1) x + 1 >= 0,
 An inequality (-1, -1, -1) x + 1 >= 0,
  An inequality (-1, 1, -1) x + 1 >= 0,
   An inequality (-1, 1, 1) x + 1 >= 0)
>>> face.ambient_H_indices()
(4, 5, 6, 7)
>>> face.n_ambient_Hrepresentation()
4
```

In non-dual mode one can ignore all faces contained in the current face:

```
sage: it = FaceIterator(C, False)
sage: face = next(it)
sage: face.ambient_H_indices()
(7,)
sage: it.ignore_subfaces()
sage: [face.ambient_H_indices() for face in it]
[(6,),
(5,),
(4,),
(3,),
(2,),
(1,),
(0,),
(5, 6),
(1, 6),
(0, 1, 5, 6),
(4, 5),
(0, 5),
(0, 3, 4, 5),
(3, 4),
(2, 3),
(0, 3),
(0, 1, 2, 3),
(1, 2),
(0, 1)]
```

```
>>> from sage.all import *
>>> it = FaceIterator(C, False)
>>> face = next(it)
>>> face.ambient_H_indices()
(7,)
>>> it.ignore_subfaces()
>>> [face.ambient_H_indices() for face in it]
[(6,),
(5,),
(4,),
(3,),
(2,),
(1,),
(0,),
(5, 6),
(1, 6),
(0, 1, 5, 6),
(4, 5),
(0, 5),
(0, 3, 4, 5),
(3, 4),
(2, 3),
(0, 3),
(0, 1, 2, 3),
(1, 2),
(0, 1)]
```

In dual mode one can ignore all faces that contain the current face:

```
sage: it = FaceIterator(C, True)
sage: face = next(it)
sage: face.ambient_V_indices()
(5,)
sage: it.ignore_supfaces()
sage: [face.ambient_V_indices() for face in it]
[(4,),
(3,),
(2,),
(1,),
(0,),
(3, 4),
(2, 4),
(0, 4),
(0, 3, 4),
(0, 2, 4),
(1, 3),
(0, 3),
(0, 1, 3),
(1, 2),
(0, 2),
(0, 1, 2),
(0, 1)]
```

```
>>> from sage.all import *
>>> it = FaceIterator(C, True)
>>> face = next(it)
>>> face.ambient_V_indices()
(5,)
>>> it.ignore_supfaces()
>>> [face.ambient_V_indices() for face in it]
[(4,),
(3,),
(2,),
(1,),
(0,),
(3, 4),
(2, 4),
(0, 4),
(0, 3, 4),
(0, 2, 4),
(1, 3),
(0, 3),
(0, 1, 3),
(1, 2),
(0, 2),
(0, 1, 2),
(0, 1)]
```

There is a special face iterator class for geometric polyhedra. It yields (geometric) polyhedral faces and it also yields trivial faces. Otherwise, it works exactly the same:

```
sage: from sage.geometry.polyhedron.combinatorial_polyhedron.face_iterator \
....:         import FaceIterator_geom
sage: P = polytopes.cube()
sage: it = FaceIterator_geom(P)
sage: [next(it) for _ in range(5)]
[A 3-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 8␣
↪vertices,
 A -1-dimensional face of a Polyhedron in ZZ^3,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices]
sage: it
Iterator over the faces of a 3-dimensional polyhedron in ZZ^3
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.combinatorial_polyhedron.face_iterator        ␣
↪import FaceIterator_geom
>>> P = polytopes.cube()
>>> it = FaceIterator_geom(P)
>>> [next(it) for _ in range(Integer(5))]
[A 3-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 8␣
↪vertices,
 A -1-dimensional face of a Polyhedron in ZZ^3,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices]
>>> it
Iterator over the faces of a 3-dimensional polyhedron in ZZ^3
```

AUTHOR:

- Jonathan Kliem (2019-04)

class sage.geometry.polyhedron.combinatorial_polyhedron.face_iterator.**FaceIterator**

Bases: *FaceIterator_base*

A class to iterate over all combinatorial faces of a polyhedron.

Construct all proper faces from the facets. In dual mode, construct all proper faces from the vertices. Dual will be faster for less vertices than facets.

INPUT:

- C – a *CombinatorialPolyhedron*
- dual – if True, then dual polyhedron is used for iteration (only possible for bounded Polyhedra)
- output_dimension – if not None, then the face iterator will only yield faces of this dimension

See also

FaceIterator, *FaceIterator_geom*, *CombinatorialPolyhedron*.

EXAMPLES:

Construct a face iterator:

```
sage: P = polytopes.cuboctahedron()
sage: C = CombinatorialPolyhedron(P)
sage: it = C.face_generator()
sage: next(it)
A 0-dimensional face of a 3-dimensional combinatorial polyhedron
```

```
>>> from sage.all import *
>>> P = polytopes.cuboctahedron()
>>> C = CombinatorialPolyhedron(P)
>>> it = C.face_generator()
>>> next(it)
A 0-dimensional face of a 3-dimensional combinatorial polyhedron
```

Construct faces by the dual or not:

```
sage: it = C.face_generator(algorithm='primal')
sage: next(it).dimension()
2

sage: it = C.face_generator(algorithm='dual')
sage: next(it).dimension()
0
```

```
>>> from sage.all import *
>>> it = C.face_generator(algorithm='primal')
>>> next(it).dimension()
2

>>> it = C.face_generator(algorithm='dual')
>>> next(it).dimension()
0
```

For unbounded polyhedra only non-dual iteration is possible:

```
sage: P = Polyhedron(rays=[[0,0,1], [0,1,0], [1,0,0]])
sage: C = CombinatorialPolyhedron(P)
sage: it = C.face_generator()
sage: [face.ambient_Vrepresentation() for face in it]
[(A vertex at (0, 0, 0),
  A ray in the direction (0, 1, 0),
  A ray in the direction (1, 0, 0)),
 (A vertex at (0, 0, 0),
  A ray in the direction (0, 0, 1),
  A ray in the direction (1, 0, 0)),
 (A vertex at (0, 0, 0),
```

(continues on next page)

(continued from previous page)

```
  A ray in the direction (0, 0, 1),
  A ray in the direction (0, 1, 0)),
 (A vertex at (0, 0, 0), A ray in the direction (1, 0, 0)),
 (A vertex at (0, 0, 0), A ray in the direction (0, 1, 0)),
 (A vertex at (0, 0, 0),),
 (A vertex at (0, 0, 0), A ray in the direction (0, 0, 1))]
sage: it = C.face_generator(algorithm='dual')
Traceback (most recent call last):
...
ValueError: dual algorithm only available for bounded polyhedra
```

```
>>> from sage.all import *
>>> P = Polyhedron(rays=[[Integer(0),Integer(0),Integer(1)], [Integer(0),
↪Integer(1),Integer(0)], [Integer(1),Integer(0),Integer(0)]])
>>> C = CombinatorialPolyhedron(P)
>>> it = C.face_generator()
>>> [face.ambient_Vrepresentation() for face in it]
[(A vertex at (0, 0, 0),
  A ray in the direction (0, 1, 0),
  A ray in the direction (1, 0, 0)),
 (A vertex at (0, 0, 0),
  A ray in the direction (0, 0, 1),
  A ray in the direction (1, 0, 0)),
 (A vertex at (0, 0, 0),
  A ray in the direction (0, 0, 1),
  A ray in the direction (0, 1, 0)),
 (A vertex at (0, 0, 0), A ray in the direction (1, 0, 0)),
 (A vertex at (0, 0, 0), A ray in the direction (0, 1, 0)),
 (A vertex at (0, 0, 0),),
 (A vertex at (0, 0, 0), A ray in the direction (0, 0, 1))]
>>> it = C.face_generator(algorithm='dual')
Traceback (most recent call last):
...
ValueError: dual algorithm only available for bounded polyhedra
```

Construct a face iterator only yielding dimension 2 faces:

```
sage: P = polytopes.permutahedron(5)
sage: C = CombinatorialPolyhedron(P)
sage: it = C.face_generator(dimension=2)
sage: counter = 0
sage: for _ in it: counter += 1
sage: print ('permutahedron(5) has', counter,
....:        'faces of dimension 2')
permutahedron(5) has 150 faces of dimension 2
sage: C.f_vector()
(1, 120, 240, 150, 30, 1)
```

```
>>> from sage.all import *
>>> P = polytopes.permutahedron(Integer(5))
>>> C = CombinatorialPolyhedron(P)
>>> it = C.face_generator(dimension=Integer(2))
```

(continues on next page)

(continued from previous page)

```
>>> counter = Integer(0)
>>> for _ in it: counter += Integer(1)
>>> print ('permutahedron(5) has', counter,
...        'faces of dimension 2')
permutahedron(5) has 150 faces of dimension 2
>>> C.f_vector()
(1, 120, 240, 150, 30, 1)
```

In non-dual mode one can ignore all faces contained in the current face:

```
sage: P = polytopes.cube()
sage: C = CombinatorialPolyhedron(P)
sage: it = C.face_generator(algorithm='primal')
sage: face = next(it)
sage: face.ambient_H_indices()
(5,)
sage: it.ignore_subfaces()
sage: [face.ambient_H_indices() for face in it]
[(4,),
 (3,),
 (2,),
 (1,),
 (0,),
 (3, 4),
 (1, 4),
 (0, 4),
 (1, 3, 4),
 (0, 1, 4),
 (2, 3),
 (1, 3),
 (1, 2, 3),
 (1, 2),
 (0, 2),
 (0, 1, 2),
 (0, 1)]

sage: it = C.face_generator(algorithm='dual')
sage: next(it)
A 0-dimensional face of a 3-dimensional combinatorial polyhedron
sage: it.ignore_subfaces()
Traceback (most recent call last):
...
ValueError: only possible when not in dual mode
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> C = CombinatorialPolyhedron(P)
>>> it = C.face_generator(algorithm='primal')
>>> face = next(it)
>>> face.ambient_H_indices()
(5,)
>>> it.ignore_subfaces()
```

(continues on next page)

(continued from previous page)

```
>>> [face.ambient_H_indices() for face in it]
[(4,),
 (3,),
 (2,),
 (1,),
 (0,),
 (3, 4),
 (1, 4),
 (0, 4),
 (1, 3, 4),
 (0, 1, 4),
 (2, 3),
 (1, 3),
 (1, 2, 3),
 (1, 2),
 (0, 2),
 (0, 1, 2),
 (0, 1)]

>>> it = C.face_generator(algorithm='dual')
>>> next(it)
A 0-dimensional face of a 3-dimensional combinatorial polyhedron
>>> it.ignore_subfaces()
Traceback (most recent call last):
...
ValueError: only possible when not in dual mode
```

In dual mode one can ignore all faces that contain the current face:

```
sage: it = C.face_generator(algorithm='dual')
sage: next(it)
A 0-dimensional face of a 3-dimensional combinatorial polyhedron
sage: face = next(it)
sage: face.ambient_V_indices()
(6,)
sage: [face.ambient_V_indices() for face in it]
[(5,),
 (4,),
 (3,),
 (2,),
 (1,),
 (0,),
 (6, 7),
 (4, 7),
 (2, 7),
 (4, 5, 6, 7),
 (1, 2, 6, 7),
 (2, 3, 4, 7),
 (5, 6),
 (1, 6),
 (0, 1, 5, 6),
 (4, 5),
```

(continues on next page)

(continued from previous page)

```
 (0, 5),
 (0, 3, 4, 5),
 (3, 4),
 (2, 3),
 (0, 3),
 (0, 1, 2, 3),
 (1, 2),
 (0, 1)]

sage: it = C.face_generator(algorithm='primal')
sage: next(it)
A 2-dimensional face of a 3-dimensional combinatorial polyhedron
sage: it.ignore_supfaces()
Traceback (most recent call last):
...
ValueError: only possible when in dual mode
```

```
>>> from sage.all import *
>>> it = C.face_generator(algorithm='dual')
>>> next(it)
A 0-dimensional face of a 3-dimensional combinatorial polyhedron
>>> face = next(it)
>>> face.ambient_V_indices()
(6,)
>>> [face.ambient_V_indices() for face in it]
[(5,),
 (4,),
 (3,),
 (2,),
 (1,),
 (0,),
 (6, 7),
 (4, 7),
 (2, 7),
 (4, 5, 6, 7),
 (1, 2, 6, 7),
 (2, 3, 4, 7),
 (5, 6),
 (1, 6),
 (0, 1, 5, 6),
 (4, 5),
 (0, 5),
 (0, 3, 4, 5),
 (3, 4),
 (2, 3),
 (0, 3),
 (0, 1, 2, 3),
 (1, 2),
 (0, 1)]

>>> it = C.face_generator(algorithm='primal')
>>> next(it)
```

(continues on next page)

(continued from previous page)

```
A 2-dimensional face of a 3-dimensional combinatorial polyhedron
>>> it.ignore_supfaces()
Traceback (most recent call last):
...
ValueError: only possible when in dual mode
```

ALGORITHM:

The algorithm to visit all proper faces exactly once is roughly equivalent to the following. A (slightly generalized) description of the algorithm can be found in [KS2019].

Initialization:

```
faces = [set(facet) for facet in P.facets()]
face_iterator(faces, [])
```

The function `face_iterator` is defined recursively. It visits all faces of the polyhedron P, except those contained in any of `visited_all`. It assumes `faces` to be exactly those facets of P that are not contained in any of the `visited_all`. It assumes `visited_all` to be some list of faces of a polyhedron P_2, which contains P as one of its faces:

```
def face_iterator(faces, visited_all):
    while facets:
        one_face = faces.pop()
        maybe_new_faces = [one_face.intersection(face) for face in faces]
...
```

At this point we claim that `maybe_new_faces` contains all facets of `one_face`, which we have not visited before.

Proof: Let F be a facet of `one_face`. We have a chain: $P \supset$ `one_face` $\supset F$. By the diamond property, there exists a `second_face` with $P \supset$ `second_face` $\supset F$.

Now either `second_face` is not an element of `faces`: Hence `second_face` is contained in one of `visited_all`. In particular, F is contained in `visited_all`.

Or `second_face` is an element of `faces`: Then, intersecting `one_face` with `second_face` gives `F`.

This concludes the proof.

Moreover, if an element in `maybe_new_faces` is inclusion-maximal and not contained in any of the `visited_all`, it is a facet of `one_face`. Any facet in `maybe_new_faces` of `one_face` is inclusion-maximal.

Hence, in the following loop, an element `face1` in `maybe_new_faces` is a facet of `one_face` if and only if it is not contained in another facet:

```
...
        maybe_new_faces2 = []
        for i, face1 in enumerate(maybe_new_faces):
            if (all(not face1 < face2 for face2 in maybe_new_faces[:i])
                    and all(not face1 <= face2 for face2 in maybe_new_
↪faces[i+1:])):
                maybe_new_faces2.append(face1)
...
```

Now `maybe_new_faces2` contains only facets of `one_face` and some faces contained in any of `visited_all`. It also contains all the facets not contained in any of `visited_all`.

We construct `new_faces` as the list of all facets of `one_face` not contained in any of `visited_all`:

```
...
        new_faces = []
        for face1 in maybe_new_faces2:
            if all(not face1 < face2 for face2 in visited_all):
                new_faces.append(face1)
...
```

By induction we can apply the algorithm, to visit all faces of `one_face` not contained in `visited_all`:

```
...
        face_iterator(new_faces, visited_all)
...
```

Finally we visit `one_face` and add it to `visited_all`:

```
...
        visit(one_face)
        visited_all.append(one_face)
```

Note: At this point, we have visited exactly those faces, contained in any of the `visited_all`. The function ends here.

ALGORITHM for the special case that all intervals of the lattice not containing zero are boolean (e.g. when the polyhedron is simple):

We do not assume any other properties of our lattice in this case. Note that intervals of length 2 not containing zero, have exactly 2 elements now. But the atom-representation of faces might not be unique.

We do the following modifications:

- To check whether an intersection of faces is zero, we check whether the atom-representation is zero. Although not unique, it works to distinct from zero.
- The intersection of two (relative) facets has always codimension 1 unless empty.
- To intersect we now additionally unite the coatom representation. This gives the correct representation of the new face unless the intersection is zero.
- To mark a face as visited, we save its coatom representation.
- To check whether we have seen a face already, we check containment of the coatom representation.

class sage.geometry.polyhedron.combinatorial_polyhedron.face_iterator.**FaceIterator_base**

Bases: `SageObject`

A base class to iterate over all faces of a polyhedron.

Construct all proper faces from the facets. In dual mode, construct all proper faces from the vertices. Dual will be faster for less vertices than facets.

See *FaceIterator*.

current()

Retrieve the last value of *next()*.

EXAMPLES:

```
sage: P = polytopes.octahedron()
sage: it = P.combinatorial_polyhedron().face_generator()
sage: next(it)
```

(continues on next page)

(continued from previous page)

```
A 0-dimensional face of a 3-dimensional combinatorial polyhedron
sage: it.current()
A 0-dimensional face of a 3-dimensional combinatorial polyhedron
sage: next(it).ambient_V_indices() == it.current().ambient_V_indices()
True
```

```
>>> from sage.all import *
>>> P = polytopes.octahedron()
>>> it = P.combinatorial_polyhedron().face_generator()
>>> next(it)
A 0-dimensional face of a 3-dimensional combinatorial polyhedron
>>> it.current()
A 0-dimensional face of a 3-dimensional combinatorial polyhedron
>>> next(it).ambient_V_indices() == it.current().ambient_V_indices()
True
```

dual

ignore_subfaces()

The iterator will not visit any faces of the current face.

Only possible when not in dual mode.

EXAMPLES:

```
sage: P = polytopes.Gosset_3_21()
sage: C = CombinatorialPolyhedron(P)
sage: it = C.face_generator(algorithm='primal')
sage: n_non_simplex_faces = 1
sage: for face in it:
....:     if face.n_ambient_Vrepresentation() > face.dimension() + 1:
....:         n_non_simplex_faces += 1
....:     else:
....:         it.ignore_subfaces()
....:
sage: n_non_simplex_faces
127
```

```
>>> from sage.all import *
>>> P = polytopes.Gosset_3_21()
>>> C = CombinatorialPolyhedron(P)
>>> it = C.face_generator(algorithm='primal')
>>> n_non_simplex_faces = Integer(1)
>>> for face in it:
...     if face.n_ambient_Vrepresentation() > face.dimension() + Integer(1):
...         n_non_simplex_faces += Integer(1)
...     else:
...         it.ignore_subfaces()
....:
>>> n_non_simplex_faces
127
```

Face iterator must not be in dual mode:

```
sage: it = C.face_generator(algorithm='dual')
sage: _ = next(it)
sage: it.ignore_subfaces()
Traceback (most recent call last):
...
ValueError: only possible when not in dual mode
```

```
>>> from sage.all import *
>>> it = C.face_generator(algorithm='dual')
>>> _ = next(it)
>>> it.ignore_subfaces()
Traceback (most recent call last):
...
ValueError: only possible when not in dual mode
```

Ignoring the same face as was requested to visit only consumes the iterator:

```
sage: it = C.face_generator(algorithm='primal')
sage: _ = next(it)
sage: it.only_subfaces()
sage: it.ignore_subfaces()
sage: list(it)
[]
```

```
>>> from sage.all import *
>>> it = C.face_generator(algorithm='primal')
>>> _ = next(it)
>>> it.only_subfaces()
>>> it.ignore_subfaces()
>>> list(it)
[]
```

Face iterator must be set to a face first:

```
sage: it = C.face_generator(algorithm='primal')
sage: it.ignore_subfaces()
Traceback (most recent call last):
...
ValueError: iterator not set to a face yet
```

```
>>> from sage.all import *
>>> it = C.face_generator(algorithm='primal')
>>> it.ignore_subfaces()
Traceback (most recent call last):
...
ValueError: iterator not set to a face yet
```

ignore_supfaces()

The iterator will not visit any faces containing the current face.

Only possible when in dual mode.

EXAMPLES:

```
sage: P = polytopes.Gosset_3_21()
sage: C = CombinatorialPolyhedron(P)
sage: it = C.face_generator(algorithm='dual')
sage: n_faces_with_non_simplex_quotient = 1
sage: for face in it:
....:     n_facets = face.n_ambient_Hrepresentation(add_equations=False)
....:     if n_facets > C.dimension() - face.dimension() + 1:
....:         n_faces_with_non_simplex_quotient += 1
....:     else:
....:         it.ignore_supfaces()
....:
sage: n_faces_with_non_simplex_quotient
4845
```

```
>>> from sage.all import *
>>> P = polytopes.Gosset_3_21()
>>> C = CombinatorialPolyhedron(P)
>>> it = C.face_generator(algorithm='dual')
>>> n_faces_with_non_simplex_quotient = Integer(1)
>>> for face in it:
...     n_facets = face.n_ambient_Hrepresentation(add_equations=False)
...     if n_facets > C.dimension() - face.dimension() + Integer(1):
...         n_faces_with_non_simplex_quotient += Integer(1)
...     else:
...         it.ignore_supfaces()
....:
>>> n_faces_with_non_simplex_quotient
4845
```

Face iterator must be in dual mode:

```
sage: it = C.face_generator(algorithm='primal')
sage: _ = next(it)
sage: it.ignore_supfaces()
Traceback (most recent call last):
...
ValueError: only possible when in dual mode
```

```
>>> from sage.all import *
>>> it = C.face_generator(algorithm='primal')
>>> _ = next(it)
>>> it.ignore_supfaces()
Traceback (most recent call last):
...
ValueError: only possible when in dual mode
```

join_of_Vrep(**indices*)

Construct the join of the Vrepresentatives indicated by the indices.

This is the smallest face containing all Vrepresentatives with the given indices.

The iterator must be reset if not newly initialized.

Note

In the case of unbounded polyhedra, the smallest face containing given Vrepresentatives may not be well defined.

EXAMPLES:

```
sage: P = polytopes.cube()
sage: it = P.face_generator()
sage: it.join_of_Vrep(1)
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex
sage: it.join_of_Vrep(1,2).ambient_V_indices()
(1, 2)
sage: it.join_of_Vrep(1,3).ambient_V_indices()
(0, 1, 2, 3)
sage: it.join_of_Vrep(1,5).ambient_V_indices()
(0, 1, 5, 6)

sage: P = polytopes.cross_polytope(4)
sage: it = P.face_generator()
sage: it.join_of_Vrep().ambient_V_indices()
()
sage: it.join_of_Vrep(1,3).ambient_V_indices()
(1, 3)
sage: it.join_of_Vrep(1,2).ambient_V_indices()
(1, 2)
sage: it.join_of_Vrep(1,6).ambient_V_indices()
(0, 1, 2, 3, 4, 5, 6, 7)
sage: it.join_of_Vrep(8)
Traceback (most recent call last):
...
IndexError: coatoms out of range
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> it = P.face_generator()
>>> it.join_of_Vrep(Integer(1))
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex
>>> it.join_of_Vrep(Integer(1),Integer(2)).ambient_V_indices()
(1, 2)
>>> it.join_of_Vrep(Integer(1),Integer(3)).ambient_V_indices()
(0, 1, 2, 3)
>>> it.join_of_Vrep(Integer(1),Integer(5)).ambient_V_indices()
(0, 1, 5, 6)

>>> P = polytopes.cross_polytope(Integer(4))
>>> it = P.face_generator()
>>> it.join_of_Vrep().ambient_V_indices()
()
>>> it.join_of_Vrep(Integer(1),Integer(3)).ambient_V_indices()
```

(continues on next page)

(continued from previous page)

```
(1, 3)
>>> it.join_of_Vrep(Integer(1),Integer(2)).ambient_V_indices()
(1, 2)
>>> it.join_of_Vrep(Integer(1),Integer(6)).ambient_V_indices()
(0, 1, 2, 3, 4, 5, 6, 7)
>>> it.join_of_Vrep(Integer(8))
Traceback (most recent call last):
...
IndexError: coatoms out of range
```

If the iterator has already been used, it must be reset before:

```
sage: # needs sage.groups sage.rings.number_field
sage: P = polytopes.dodecahedron()
sage: it = P.face_generator()
sage: _ = next(it), next(it)
sage: next(it).ambient_V_indices()
(15, 16, 17, 18, 19)
sage: it.join_of_Vrep(1,10)
Traceback (most recent call last):
...
ValueError: please reset the face iterator
sage: it.reset()
sage: it.join_of_Vrep(1,10).ambient_V_indices()
(1, 10)
```

```
>>> from sage.all import *
>>> # needs sage.groups sage.rings.number_field
>>> P = polytopes.dodecahedron()
>>> it = P.face_generator()
>>> _ = next(it), next(it)
>>> next(it).ambient_V_indices()
(15, 16, 17, 18, 19)
>>> it.join_of_Vrep(Integer(1),Integer(10))
Traceback (most recent call last):
...
ValueError: please reset the face iterator
>>> it.reset()
>>> it.join_of_Vrep(Integer(1),Integer(10)).ambient_V_indices()
(1, 10)
```

In the case of an unbounded polyhedron, we try to make sense of the input:

```
sage: P = polytopes.cube()*Polyhedron(lines=[[1]])
sage: it = P.face_generator()
sage: it.join_of_Vrep(1)
A 1-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 1␣
↪vertex and 1 line
sage: it.join_of_Vrep(0, 1)
A 1-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 1␣
↪vertex and 1 line
sage: it.join_of_Vrep(0)
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: the join is not well-defined

sage: P = Polyhedron(vertices=[[1,0], [0,1]], rays=[[1,1]])
sage: it = P.face_generator()
sage: it.join_of_Vrep(0)
A 0-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 1␣
↪vertex
sage: it.join_of_Vrep(1)
A 0-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 1␣
↪vertex
sage: it.join_of_Vrep(2)
Traceback (most recent call last):
...
ValueError: the join is not well-defined
sage: it.join_of_Vrep(0,2)
A 1-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 1␣
↪vertex and 1 ray

sage: P = Polyhedron(rays=[[1,0], [0,1]])
sage: it = P.face_generator()
sage: it.join_of_Vrep(0)
A 0-dimensional face of a Polyhedron in ZZ^2 defined as the convex hull of 1␣
↪vertex
sage: it.join_of_Vrep(1,2)
A 2-dimensional face of a Polyhedron in ZZ^2 defined as the convex hull of 1␣
↪vertex and 2 rays
```

```
>>> from sage.all import *
>>> P = polytopes.cube()*Polyhedron(lines=[[Integer(1)]])
>>> it = P.face_generator()
>>> it.join_of_Vrep(Integer(1))
A 1-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 1␣
↪vertex and 1 line
>>> it.join_of_Vrep(Integer(0), Integer(1))
A 1-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 1␣
↪vertex and 1 line
>>> it.join_of_Vrep(Integer(0))
Traceback (most recent call last):
...
ValueError: the join is not well-defined

>>> P = Polyhedron(vertices=[[Integer(1),Integer(0)], [Integer(0),
↪Integer(1)]], rays=[[Integer(1),Integer(1)]])
>>> it = P.face_generator()
>>> it.join_of_Vrep(Integer(0))
A 0-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 1␣
↪vertex
>>> it.join_of_Vrep(Integer(1))
A 0-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 1␣
↪vertex
```

(continues on next page)

(continued from previous page)

```
>>> it.join_of_Vrep(Integer(2))
Traceback (most recent call last):
...
ValueError: the join is not well-defined
>>> it.join_of_Vrep(Integer(0),Integer(2))
A 1-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 1␣
↪vertex and 1 ray

>>> P = Polyhedron(rays=[[Integer(1),Integer(0)], [Integer(0),Integer(1)]])
>>> it = P.face_generator()
>>> it.join_of_Vrep(Integer(0))
A 0-dimensional face of a Polyhedron in ZZ^2 defined as the convex hull of 1␣
↪vertex
>>> it.join_of_Vrep(Integer(1),Integer(2))
A 2-dimensional face of a Polyhedron in ZZ^2 defined as the convex hull of 1␣
↪vertex and 2 rays
```

meet_of_Hrep(**indices*)

Construct the meet of the facets indicated by the indices.

This is the largest face contained in all facets with the given indices.

The iterator must be reset if not newly initialized.

EXAMPLES:

```
sage: P = polytopes.cube()
sage: it = P.face_generator()
sage: it.meet_of_Hrep(1,2)
A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices
sage: it.meet_of_Hrep(1,2).ambient_H_indices()
(1, 2)
sage: it.meet_of_Hrep(1,3).ambient_H_indices()
(1, 3)
sage: it.meet_of_Hrep(1,5).ambient_H_indices()
(0, 1, 2, 3, 4, 5)

sage: P = polytopes.cross_polytope(4)
sage: it = P.face_generator()
sage: it.meet_of_Hrep().ambient_H_indices()
()
sage: it.meet_of_Hrep(1,3).ambient_H_indices()
(1, 2, 3, 4)
sage: it.meet_of_Hrep(1,2).ambient_H_indices()
(1, 2)
sage: it.meet_of_Hrep(1,6).ambient_H_indices()
(1, 6)
sage: it.meet_of_Hrep(1,2,6).ambient_H_indices()
(1, 2, 6, 7)
sage: it.meet_of_Hrep(1,2,5,6).ambient_H_indices()
(0, 1, 2, 3, 4, 5, 6, 7)
```

(continues on next page)

(continued from previous page)

```
sage: s = cones.schur(4)
sage: C = CombinatorialPolyhedron(s)
sage: it = C.face_generator()
sage: it.meet_of_Hrep(1,2).ambient_H_indices()
(1, 2)
sage: it.meet_of_Hrep(1,2,3).ambient_H_indices()
Traceback (most recent call last):
...
IndexError: coatoms out of range
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> it = P.face_generator()
>>> it.meet_of_Hrep(Integer(1),Integer(2))
A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices
>>> it.meet_of_Hrep(Integer(1),Integer(2)).ambient_H_indices()
(1, 2)
>>> it.meet_of_Hrep(Integer(1),Integer(3)).ambient_H_indices()
(1, 3)
>>> it.meet_of_Hrep(Integer(1),Integer(5)).ambient_H_indices()
(0, 1, 2, 3, 4, 5)

>>> P = polytopes.cross_polytope(Integer(4))
>>> it = P.face_generator()
>>> it.meet_of_Hrep().ambient_H_indices()
()
>>> it.meet_of_Hrep(Integer(1),Integer(3)).ambient_H_indices()
(1, 2, 3, 4)
>>> it.meet_of_Hrep(Integer(1),Integer(2)).ambient_H_indices()
(1, 2)
>>> it.meet_of_Hrep(Integer(1),Integer(6)).ambient_H_indices()
(1, 6)
>>> it.meet_of_Hrep(Integer(1),Integer(2),Integer(6)).ambient_H_indices()
(1, 2, 6, 7)
>>> it.meet_of_Hrep(Integer(1),Integer(2),Integer(5),Integer(6)).ambient_H_
↪indices()
(0, 1, 2, 3, 4, 5, 6, 7)

>>> s = cones.schur(Integer(4))
>>> C = CombinatorialPolyhedron(s)
>>> it = C.face_generator()
>>> it.meet_of_Hrep(Integer(1),Integer(2)).ambient_H_indices()
(1, 2)
>>> it.meet_of_Hrep(Integer(1),Integer(2),Integer(3)).ambient_H_indices()
Traceback (most recent call last):
...
IndexError: coatoms out of range
```

If the iterator has already been used, it must be reset before:

```
sage: # needs sage.groups sage.rings.number_field
```

(continues on next page)

(continued from previous page)

```
sage: P = polytopes.dodecahedron()
sage: it = P.face_generator()
sage: _ = next(it), next(it)
sage: next(it).ambient_V_indices()
(15, 16, 17, 18, 19)
sage: it.meet_of_Hrep(9,11)
Traceback (most recent call last):
...
ValueError: please reset the face iterator
sage: it.reset()
sage: it.meet_of_Hrep(9,11).ambient_H_indices()
(9, 11)
```

```
>>> from sage.all import *
>>> # needs sage.groups sage.rings.number_field
>>> P = polytopes.dodecahedron()
>>> it = P.face_generator()
>>> _ = next(it), next(it)
>>> next(it).ambient_V_indices()
(15, 16, 17, 18, 19)
>>> it.meet_of_Hrep(Integer(9),Integer(11))
Traceback (most recent call last):
...
ValueError: please reset the face iterator
>>> it.reset()
>>> it.meet_of_Hrep(Integer(9),Integer(11)).ambient_H_indices()
(9, 11)
```

next()

Must be implemented by a derived class.

only_subfaces()

The iterator will visit all (remaining) subfaces of the current face and then terminate.

EXAMPLES:

```
sage: P = polytopes.cube()
sage: it = P.face_generator()
sage: next(it).ambient_H_indices()
()
sage: next(it).ambient_H_indices()
(0, 1, 2, 3, 4, 5)
sage: next(it).ambient_H_indices()
(5,)
sage: next(it).ambient_H_indices()
(4,)
sage: it.only_subfaces()
sage: list(f.ambient_H_indices() for f in it)
[(4, 5), (3, 4), (1, 4), (0, 4), (3, 4, 5), (0, 4, 5), (1, 3, 4), (0, 1, 4)]
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
```

(continues on next page)

(continued from previous page)

```
>>> it = P.face_generator()
>>> next(it).ambient_H_indices()
()
>>> next(it).ambient_H_indices()
(0, 1, 2, 3, 4, 5)
>>> next(it).ambient_H_indices()
(5,)
>>> next(it).ambient_H_indices()
(4,)
>>> it.only_subfaces()
>>> list(f.ambient_H_indices() for f in it)
[(4, 5), (3, 4), (1, 4), (0, 4), (3, 4, 5), (0, 4, 5), (1, 3, 4), (0, 1, 4)]
```

```
sage: P = polytopes.Birkhoff_polytope(4)
sage: C = P.combinatorial_polyhedron()
sage: it = C.face_generator()
sage: next(it).ambient_H_indices(add_equations=False)
(15,)
sage: next(it).ambient_H_indices(add_equations=False)
(14,)
sage: it.only_subfaces()
sage: all(14 in f.ambient_H_indices() for f in it)
True
```

```
>>> from sage.all import *
>>> P = polytopes.Birkhoff_polytope(Integer(4))
>>> C = P.combinatorial_polyhedron()
>>> it = C.face_generator()
>>> next(it).ambient_H_indices(add_equations=False)
(15,)
>>> next(it).ambient_H_indices(add_equations=False)
(14,)
>>> it.only_subfaces()
>>> all(Integer(14) in f.ambient_H_indices() for f in it)
True
```

Face iterator needs to be set to a face first:

```
sage: it = C.face_generator()
sage: it.only_subfaces()
Traceback (most recent call last):
...
ValueError: iterator not set to a face yet
```

```
>>> from sage.all import *
>>> it = C.face_generator()
>>> it.only_subfaces()
Traceback (most recent call last):
...
ValueError: iterator not set to a face yet
```

Face iterator must not be in dual mode:

```
sage: it = C.face_generator(algorithm='dual')
sage: _ = next(it)
sage: it.only_subfaces()
Traceback (most recent call last):
...
ValueError: only possible when not in dual mode
```

```
>>> from sage.all import *
>>> it = C.face_generator(algorithm='dual')
>>> _ = next(it)
>>> it.only_subfaces()
Traceback (most recent call last):
...
ValueError: only possible when not in dual mode
```

Cannot run `only_subfaces` after `ignore_subfaces`:

```
sage: it = C.face_generator()
sage: _ = next(it)
sage: it.ignore_subfaces()
sage: it.only_subfaces()
Traceback (most recent call last):
...
ValueError: cannot only visit subsets after ignoring a face
```

```
>>> from sage.all import *
>>> it = C.face_generator()
>>> _ = next(it)
>>> it.ignore_subfaces()
>>> it.only_subfaces()
Traceback (most recent call last):
...
ValueError: cannot only visit subsets after ignoring a face
```

only_supfaces()

The iterator will visit all (remaining) faces containing the current face and then terminate.

EXAMPLES:

```
sage: P = polytopes.cross_polytope(3)
sage: it = P.face_generator()
sage: next(it).ambient_V_indices()
(0, 1, 2, 3, 4, 5)
sage: next(it).ambient_V_indices()
()
sage: next(it).ambient_V_indices()
(5,)
sage: next(it).ambient_V_indices()
(4,)
sage: it.only_supfaces()
sage: list(f.ambient_V_indices() for f in it)
[(4, 5), (3, 4), (2, 4), (0, 4), (3, 4, 5), (2, 4, 5), (0, 3, 4), (0, 2, 4)]
```

```
>>> from sage.all import *
>>> P = polytopes.cross_polytope(Integer(3))
>>> it = P.face_generator()
>>> next(it).ambient_V_indices()
(0, 1, 2, 3, 4, 5)
>>> next(it).ambient_V_indices()
()
>>> next(it).ambient_V_indices()
(5,)
>>> next(it).ambient_V_indices()
(4,)
>>> it.only_supfaces()
>>> list(f.ambient_V_indices() for f in it)
[(4, 5), (3, 4), (2, 4), (0, 4), (3, 4, 5), (2, 4, 5), (0, 3, 4), (0, 2, 4)]
```

```
sage: P = polytopes.Birkhoff_polytope(4)
sage: C = P.combinatorial_polyhedron()
sage: it = C.face_generator(algorithm='dual')
sage: next(it).ambient_V_indices()
(23,)
sage: next(it).ambient_V_indices()
(22,)
sage: it.only_supfaces()
sage: all(22 in f.ambient_V_indices() for f in it)
True
```

```
>>> from sage.all import *
>>> P = polytopes.Birkhoff_polytope(Integer(4))
>>> C = P.combinatorial_polyhedron()
>>> it = C.face_generator(algorithm='dual')
>>> next(it).ambient_V_indices()
(23,)
>>> next(it).ambient_V_indices()
(22,)
>>> it.only_supfaces()
>>> all(Integer(22) in f.ambient_V_indices() for f in it)
True
```

reset()

Reset the iterator.

The iterator will start with the first face again.

EXAMPLES:

```
sage: P = polytopes.cube()
sage: C = P.combinatorial_polyhedron()
sage: it = C.face_generator()
sage: next(it).ambient_V_indices()
(0, 3, 4, 5)
sage: it.reset()
sage: next(it).ambient_V_indices()
(0, 3, 4, 5)
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> C = P.combinatorial_polyhedron()
>>> it = C.face_generator()
>>> next(it).ambient_V_indices()
(0, 3, 4, 5)
>>> it.reset()
>>> next(it).ambient_V_indices()
(0, 3, 4, 5)
```

class sage.geometry.polyhedron.combinatorial_polyhedron.face_iterator.**FaceIterator_geom**

Bases: *FaceIterator_base*

A class to iterate over all geometric faces of a polyhedron.

Construct all faces from the facets. In dual mode, construct all faces from the vertices. Dual will be faster for less vertices than facets.

INPUT:

- `P` – an instance of *Polyhedron_base*
- `dual` – if `True`, then dual polyhedron is used for iteration (only possible for bounded Polyhedra)
- `output_dimension` – if not `None`, then the FaceIterator will only yield faces of this dimension

EXAMPLES:

Construct a geometric face iterator:

```
sage: P = polytopes.cuboctahedron()
sage: it = P.face_generator()
sage: next(it)
A 3-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 12␣
↪vertices
```

```
>>> from sage.all import *
>>> P = polytopes.cuboctahedron()
>>> it = P.face_generator()
>>> next(it)
A 3-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 12␣
↪vertices
```

Construct faces by the dual or not:

```
sage: it = P.face_generator(algorithm='primal')
sage: _ = next(it), next(it)
sage: next(it).dim()
2

sage: it = P.face_generator(algorithm='dual')
sage: _ = next(it), next(it)
sage: next(it).dim()
0
```

```
>>> from sage.all import *
>>> it = P.face_generator(algorithm='primal')
>>> _ = next(it), next(it)
>>> next(it).dim()
2

>>> it = P.face_generator(algorithm='dual')
>>> _ = next(it), next(it)
>>> next(it).dim()
0
```

For unbounded polyhedra only non-dual iteration is possible:

```
sage: P = Polyhedron(rays=[[0,0,1], [0,1,0], [1,0,0]])
sage: it = P.face_generator()
sage: [face.ambient_Vrepresentation() for face in it]
[(A vertex at (0, 0, 0),
  A ray in the direction (0, 0, 1),
  A ray in the direction (0, 1, 0),
  A ray in the direction (1, 0, 0)),
 (),
 (A vertex at (0, 0, 0),
  A ray in the direction (0, 1, 0),
  A ray in the direction (1, 0, 0)),
 (A vertex at (0, 0, 0),
  A ray in the direction (0, 0, 1),
  A ray in the direction (1, 0, 0)),
 (A vertex at (0, 0, 0),
  A ray in the direction (0, 0, 1),
  A ray in the direction (0, 1, 0)),
 (A vertex at (0, 0, 0), A ray in the direction (1, 0, 0)),
 (A vertex at (0, 0, 0), A ray in the direction (0, 1, 0)),
 (A vertex at (0, 0, 0),),
 (A vertex at (0, 0, 0), A ray in the direction (0, 0, 1))]
sage: it = P.face_generator(algorithm='dual')
Traceback (most recent call last):
...
ValueError: cannot iterate over dual of unbounded Polyedron
```

```
>>> from sage.all import *
>>> P = Polyhedron(rays=[[Integer(0),Integer(0),Integer(1)], [Integer(0),
↪Integer(1),Integer(0)], [Integer(1),Integer(0),Integer(0)]])
>>> it = P.face_generator()
>>> [face.ambient_Vrepresentation() for face in it]
[(A vertex at (0, 0, 0),
  A ray in the direction (0, 0, 1),
  A ray in the direction (0, 1, 0),
  A ray in the direction (1, 0, 0)),
 (),
 (A vertex at (0, 0, 0),
  A ray in the direction (0, 1, 0),
  A ray in the direction (1, 0, 0)),
 (A vertex at (0, 0, 0),
```

(continues on next page)

(continued from previous page)

```
  A ray in the direction (0, 0, 1),
  A ray in the direction (1, 0, 0)),
 (A vertex at (0, 0, 0),
  A ray in the direction (0, 0, 1),
  A ray in the direction (0, 1, 0)),
 (A vertex at (0, 0, 0), A ray in the direction (1, 0, 0)),
 (A vertex at (0, 0, 0), A ray in the direction (0, 1, 0)),
 (A vertex at (0, 0, 0),),
 (A vertex at (0, 0, 0), A ray in the direction (0, 0, 1))]
>>> it = P.face_generator(algorithm='dual')
Traceback (most recent call last):
...
ValueError: cannot iterate over dual of unbounded Polyedron
```

Construct a FaceIterator only yielding dimension 2 faces:

```
sage: P = polytopes.permutahedron(5)
sage: it = P.face_generator(face_dimension=2)
sage: counter = 0
sage: for _ in it: counter += 1
sage: print ('permutahedron(5) has', counter,
....:        'faces of dimension 2')
permutahedron(5) has 150 faces of dimension 2
sage: P.f_vector()
(1, 120, 240, 150, 30, 1)
```

```
>>> from sage.all import *
>>> P = polytopes.permutahedron(Integer(5))
>>> it = P.face_generator(face_dimension=Integer(2))
>>> counter = Integer(0)
>>> for _ in it: counter += Integer(1)
>>> print ('permutahedron(5) has', counter,
...        'faces of dimension 2')
permutahedron(5) has 150 faces of dimension 2
>>> P.f_vector()
(1, 120, 240, 150, 30, 1)
```

In non-dual mode one can ignore all faces contained in the current face:

```
sage: P = polytopes.cube()
sage: it = P.face_generator(algorithm='primal')
sage: _ = next(it), next(it)
sage: face = next(it)
sage: face.ambient_H_indices()
(5,)
sage: it.ignore_subfaces()
sage: [face.ambient_H_indices() for face in it]
[(4,),
 (3,),
 (2,),
 (1,),
 (0,),
```

(continues on next page)

(continued from previous page)

```
 (3, 4),
 (1, 4),
 (0, 4),
 (1, 3, 4),
 (0, 1, 4),
 (2, 3),
 (1, 3),
 (1, 2, 3),
 (1, 2),
 (0, 2),
 (0, 1, 2),
 (0, 1)]

sage: it = P.face_generator(algorithm='dual')
sage: _ = next(it), next(it)
sage: next(it)
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex
sage: it.ignore_subfaces()
Traceback (most recent call last):
...
ValueError: only possible when not in dual mode
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> it = P.face_generator(algorithm='primal')
>>> _ = next(it), next(it)
>>> face = next(it)
>>> face.ambient_H_indices()
(5,)
>>> it.ignore_subfaces()
>>> [face.ambient_H_indices() for face in it]
[(4,),
 (3,),
 (2,),
 (1,),
 (0,),
 (3, 4),
 (1, 4),
 (0, 4),
 (1, 3, 4),
 (0, 1, 4),
 (2, 3),
 (1, 3),
 (1, 2, 3),
 (1, 2),
 (0, 2),
 (0, 1, 2),
 (0, 1)]

>>> it = P.face_generator(algorithm='dual')
>>> _ = next(it), next(it)
```

(continues on next page)

(continued from previous page)

```
>>> next(it)
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex
>>> it.ignore_subfaces()
Traceback (most recent call last):
...
ValueError: only possible when not in dual mode
```

In dual mode one can ignore all faces that contain the current face:

```
sage: P = polytopes.cube()
sage: it = P.face_generator(algorithm='dual')
sage: _ = next(it), next(it)
sage: next(it)
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex
sage: face = next(it)
sage: face.ambient_V_indices()
(6,)
sage: [face.ambient_V_indices() for face in it]
[(5,),
 (4,),
 (3,),
 (2,),
 (1,),
 (0,),
 (6, 7),
 (4, 7),
 (2, 7),
 (4, 5, 6, 7),
 (1, 2, 6, 7),
 (2, 3, 4, 7),
 (5, 6),
 (1, 6),
 (0, 1, 5, 6),
 (4, 5),
 (0, 5),
 (0, 3, 4, 5),
 (3, 4),
 (2, 3),
 (0, 3),
 (0, 1, 2, 3),
 (1, 2),
 (0, 1)]

sage: it = P.face_generator(algorithm='primal')
sage: _ = next(it), next(it)
sage: next(it)
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices
sage: it.ignore_supfaces()
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: only possible when in dual mode
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> it = P.face_generator(algorithm='dual')
>>> _ = next(it), next(it)
>>> next(it)
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex
>>> face = next(it)
>>> face.ambient_V_indices()
(6,)
>>> [face.ambient_V_indices() for face in it]
[(5,),
 (4,),
 (3,),
 (2,),
 (1,),
 (0,),
 (6, 7),
 (4, 7),
 (2, 7),
 (4, 5, 6, 7),
 (1, 2, 6, 7),
 (2, 3, 4, 7),
 (5, 6),
 (1, 6),
 (0, 1, 5, 6),
 (4, 5),
 (0, 5),
 (0, 3, 4, 5),
 (3, 4),
 (2, 3),
 (0, 3),
 (0, 1, 2, 3),
 (1, 2),
 (0, 1)]

>>> it = P.face_generator(algorithm='primal')
>>> _ = next(it), next(it)
>>> next(it)
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices
>>> it.ignore_supfaces()
Traceback (most recent call last):
...
ValueError: only possible when in dual mode
```

See also

FaceIterator_base.

P

current()

Retrieve the last value of `__next__()`.

EXAMPLES:

```
sage: P = polytopes.octahedron()
sage: it = P.face_generator()
sage: _ = next(it), next(it)
sage: next(it)
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex
sage: it.current()
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex
sage: next(it).ambient_V_indices() == it.current().ambient_V_indices()
True
```

```
>>> from sage.all import *
>>> P = polytopes.octahedron()
>>> it = P.face_generator()
>>> _ = next(it), next(it)
>>> next(it)
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex
>>> it.current()
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex
>>> next(it).ambient_V_indices() == it.current().ambient_V_indices()
True
```

reset()

Reset the iterator.

The iterator will start with the first face again.

EXAMPLES:

```
sage: P = polytopes.cube()
sage: it = P.face_generator()
sage: next(it).ambient_V_indices()
(0, 1, 2, 3, 4, 5, 6, 7)
sage: next(it).ambient_V_indices()
()
sage: next(it).ambient_V_indices()
(0, 3, 4, 5)
sage: it.reset()
sage: next(it).ambient_V_indices()
(0, 1, 2, 3, 4, 5, 6, 7)
sage: next(it).ambient_V_indices()
()
```

(continues on next page)

(continued from previous page)

```
sage: next(it).ambient_V_indices()
(0, 3, 4, 5)
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> it = P.face_generator()
>>> next(it).ambient_V_indices()
(0, 1, 2, 3, 4, 5, 6, 7)
>>> next(it).ambient_V_indices()
()
>>> next(it).ambient_V_indices()
(0, 3, 4, 5)
>>> it.reset()
>>> next(it).ambient_V_indices()
(0, 1, 2, 3, 4, 5, 6, 7)
>>> next(it).ambient_V_indices()
()
>>> next(it).ambient_V_indices()
(0, 3, 4, 5)
```

2.3.5 List of faces

This module provides a class to store faces of a polyhedron in Bit-representation.

This class allocates memory to store the faces in. A face will be stored as vertex-incidences, where each Bit represents an incidence. In *conversions* there a methods to actually convert facets of a polyhedron to bit-representations of vertices stored in *ListOfFaces*.

Moreover, *ListOfFaces* calculates the dimension of a polyhedron, assuming the faces are the facets of this polyhedron.

Each face is stored over-aligned according to the `chunktype`.

See also

sage.geometry.polyhedron.combinatorial_polyhedron.base.

EXAMPLES:

Provide enough space to store 20 faces as incidences to 60 vertices:

```
sage: from sage.geometry.polyhedron.combinatorial_polyhedron.list_of_faces \
....: import ListOfFaces
sage: face_list = ListOfFaces(20, 60, 20)
sage: face_list.matrix().is_zero()
True
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.combinatorial_polyhedron.list_of_faces import␣
↪ListOfFaces
>>> face_list = ListOfFaces(Integer(20), Integer(60), Integer(20))
>>> face_list.matrix().is_zero()
True
```

Obtain the facets of a polyhedron:

```
sage: from sage.geometry.polyhedron.combinatorial_polyhedron.conversions \
....:         import incidence_matrix_to_bit_rep_of_facets
sage: P = polytopes.cube()
sage: face_list = incidence_matrix_to_bit_rep_of_facets(P.incidence_matrix())
sage: face_list = incidence_matrix_to_bit_rep_of_facets(P.incidence_matrix())
sage: face_list.compute_dimension()
3
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.combinatorial_polyhedron.conversions         import␣
↪incidence_matrix_to_bit_rep_of_facets
>>> P = polytopes.cube()
>>> face_list = incidence_matrix_to_bit_rep_of_facets(P.incidence_matrix())
>>> face_list = incidence_matrix_to_bit_rep_of_facets(P.incidence_matrix())
>>> face_list.compute_dimension()
3
```

Obtain the Vrepresentation of a polyhedron as facet-incidences:

```
sage: # needs sage.combinat
sage: from sage.geometry.polyhedron.combinatorial_polyhedron.conversions \
....:         import incidence_matrix_to_bit_rep_of_Vrep
sage: P = polytopes.associahedron(['A',3])
sage: face_list = incidence_matrix_to_bit_rep_of_Vrep(P.incidence_matrix())
sage: face_list.compute_dimension()
3
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> from sage.geometry.polyhedron.combinatorial_polyhedron.conversions         import␣
↪incidence_matrix_to_bit_rep_of_Vrep
>>> P = polytopes.associahedron(['A',Integer(3)])
>>> face_list = incidence_matrix_to_bit_rep_of_Vrep(P.incidence_matrix())
>>> face_list.compute_dimension()
3
```

Obtain the facets of a polyhedron as `ListOfFaces` from a facet list:

```
sage: from sage.geometry.polyhedron.combinatorial_polyhedron.conversions \
....:         import facets_tuple_to_bit_rep_of_facets
sage: facets = ((0,1,2), (0,1,3), (0,2,3), (1,2,3))
sage: face_list = facets_tuple_to_bit_rep_of_facets(facets, 4)
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.combinatorial_polyhedron.conversions         import␣
↪facets_tuple_to_bit_rep_of_facets
>>> facets = ((Integer(0),Integer(1),Integer(2)), (Integer(0),Integer(1),Integer(3)),␣
↪(Integer(0),Integer(2),Integer(3)), (Integer(1),Integer(2),Integer(3)))
>>> face_list = facets_tuple_to_bit_rep_of_facets(facets, Integer(4))
```

Likewise for the Vrepresentatives as facet-incidences:

```
sage: from sage.geometry.polyhedron.combinatorial_polyhedron.conversions \
....:         import facets_tuple_to_bit_rep_of_Vrep
sage: facets = ((0,1,2), (0,1,3), (0,2,3), (1,2,3))
sage: face_list = facets_tuple_to_bit_rep_of_Vrep(facets, 4)
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.combinatorial_polyhedron.conversions         import␣
↪facets_tuple_to_bit_rep_of_Vrep
>>> facets = ((Integer(0),Integer(1),Integer(2)), (Integer(0),Integer(1),Integer(3)),␣
↪(Integer(0),Integer(2),Integer(3)), (Integer(1),Integer(2),Integer(3)))
>>> face_list = facets_tuple_to_bit_rep_of_Vrep(facets, Integer(4))
```

Obtain the matrix of a list of faces:

```
sage: face_list.matrix()
[1 1 1 0]
[1 1 0 1]
[1 0 1 1]
[0 1 1 1]
```

```
>>> from sage.all import *
>>> face_list.matrix()
[1 1 1 0]
[1 1 0 1]
[1 0 1 1]
[0 1 1 1]
```

See also

base, *face_iterator*, *conversions*, `polyhedron_faces_lattice`.

AUTHOR:

- Jonathan Kliem (2019-04)

class `sage.geometry.polyhedron.combinatorial_polyhedron.list_of_faces.`**ListOfFaces**

Bases: `object`

A class to store the Bit-representation of faces in.

This class will allocate the memory for the faces.

INPUT:

- `n_faces` – the number of faces to be stored
- `n_atoms` – the total number of atoms the faces contain
- `n_coatoms` – the total number of coatoms of the polyhedron

See also

`incidence_matrix_to_bit_rep_of_facets()`, `incidence_matrix_to_bit_rep_of_Vrep()`, `facets_tuple_to_bit_rep_of_facets()`, `facets_tuple_to_bit_rep_of_Vrep()`, *FaceIterator*, *CombinatorialPolyhedron*.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.combinatorial_polyhedron.list_of_faces \
....:     import ListOfFaces
sage: facets = ListOfFaces(5, 13, 5)
sage: facets.matrix().dimensions()
(5, 13)
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.combinatorial_polyhedron.list_of_faces␣
↪import ListOfFaces
>>> facets = ListOfFaces(Integer(5), Integer(13), Integer(5))
>>> facets.matrix().dimensions()
(5, 13)
```

compute_dimension()

Compute the dimension of a polyhedron by its facets.

This assumes that `self` is the list of facets of a polyhedron.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.combinatorial_polyhedron.conversions \
....:     import facets_tuple_to_bit_rep_of_facets, \
....:            facets_tuple_to_bit_rep_of_Vrep
sage: bi_pyr = ((0,1,4), (1,2,4), (2,3,4), (3,0,4),
....:           (0,1,5), (1,2,5), (2,3,5), (3,0,5))
sage: facets = facets_tuple_to_bit_rep_of_facets(bi_pyr, 6)
sage: Vrep = facets_tuple_to_bit_rep_of_Vrep(bi_pyr, 6)
sage: facets.compute_dimension()
3
sage: Vrep.compute_dimension()
3
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.combinatorial_polyhedron.conversions   ␣
↪import facets_tuple_to_bit_rep_of_facets,            facets_tuple_to_bit_
↪rep_of_Vrep
>>> bi_pyr = ((Integer(0),Integer(1),Integer(4)), (Integer(1),Integer(2),
↪Integer(4)), (Integer(2),Integer(3),Integer(4)), (Integer(3),Integer(0),
↪Integer(4)),
...           (Integer(0),Integer(1),Integer(5)), (Integer(1),Integer(2),
↪Integer(5)), (Integer(2),Integer(3),Integer(5)), (Integer(3),Integer(0),
↪Integer(5)))
>>> facets = facets_tuple_to_bit_rep_of_facets(bi_pyr, Integer(6))
>>> Vrep = facets_tuple_to_bit_rep_of_Vrep(bi_pyr, Integer(6))
>>> facets.compute_dimension()
3
>>> Vrep.compute_dimension()
3
```

ALGORITHM:

This is done by iteration:

Computes the facets of one of the facets (i.e. the ridges contained in one of the facets). Then computes the dimension of the facet, by considering its facets.

Repeats until a face has only one facet. Usually this is a vertex.

However, in the unbounded case, this might be different. The face with only one facet might be a ray or a line. So the correct dimension of a polyhedron with one facet is the number of `[lines, rays, vertices]` that the facet contains.

Hence, we know the dimension of a face, which has only one facet and iteratively we know the dimension of entire polyhedron we started from.

matrix()

Obtain the matrix of `self`.

Each row represents a face and each column an atom.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.combinatorial_polyhedron.conversions \
....:     import facets_tuple_to_bit_rep_of_facets, \
....:     facets_tuple_to_bit_rep_of_Vrep
sage: bi_pyr = ((0,1,4), (1,2,4), (2,3,4), (3,0,4), (0,1,5), (1,2,5), (2,3,5),
↪ (3,0,5))
sage: facets = facets_tuple_to_bit_rep_of_facets(bi_pyr, 6)
sage: Vrep = facets_tuple_to_bit_rep_of_Vrep(bi_pyr, 6)
sage: facets.matrix()
[1 1 0 0 1 0]
[0 1 1 0 1 0]
[0 0 1 1 1 0]
[1 0 0 1 1 0]
[1 1 0 0 0 1]
[0 1 1 0 0 1]
[0 0 1 1 0 1]
[1 0 0 1 0 1]
sage: facets.matrix().transpose() == Vrep.matrix()
True
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.combinatorial_polyhedron.conversions     ␣
↪import facets_tuple_to_bit_rep_of_facets,     facets_tuple_to_bit_rep_of_
↪Vrep
>>> bi_pyr = ((Integer(0),Integer(1),Integer(4)), (Integer(1),Integer(2),
↪Integer(4)), (Integer(2),Integer(3),Integer(4)), (Integer(3),Integer(0),
↪Integer(4)), (Integer(0),Integer(1),Integer(5)), (Integer(1),Integer(2),
↪Integer(5)), (Integer(2),Integer(3),Integer(5)), (Integer(3),Integer(0),
↪Integer(5)))
>>> facets = facets_tuple_to_bit_rep_of_facets(bi_pyr, Integer(6))
>>> Vrep = facets_tuple_to_bit_rep_of_Vrep(bi_pyr, Integer(6))
>>> facets.matrix()
[1 1 0 0 1 0]
[0 1 1 0 1 0]
[0 0 1 1 1 0]
[1 0 0 1 1 0]
[1 1 0 0 0 1]
[0 1 1 0 0 1]
```

(continues on next page)

(continued from previous page)

```
[0 0 1 1 0 1]
[1 0 0 1 0 1]
>>> facets.matrix().transpose() == Vrep.matrix()
True
```

pyramid()

Return the list of faces of the pyramid.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.combinatorial_polyhedron.conversions \
....:         import facets_tuple_to_bit_rep_of_facets
sage: facets = ((0,1,2), (0,1,3), (0,2,3), (1,2,3))
sage: face_list = facets_tuple_to_bit_rep_of_facets(facets, 4)
sage: face_list.matrix()
[1 1 1 0]
[1 1 0 1]
[1 0 1 1]
[0 1 1 1]
sage: face_list.pyramid().matrix()
[1 1 1 0 1]
[1 1 0 1 1]
[1 0 1 1 1]
[0 1 1 1 1]
[1 1 1 1 0]
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.combinatorial_polyhedron.conversions ␣
↪import facets_tuple_to_bit_rep_of_facets
>>> facets = ((Integer(0),Integer(1),Integer(2)), (Integer(0),Integer(1),
↪Integer(3)), (Integer(0),Integer(2),Integer(3)), (Integer(1),Integer(2),
↪Integer(3)))
>>> face_list = facets_tuple_to_bit_rep_of_facets(facets, Integer(4))
>>> face_list.matrix()
[1 1 1 0]
[1 1 0 1]
[1 0 1 1]
[0 1 1 1]
>>> face_list.pyramid().matrix()
[1 1 1 0 1]
[1 1 0 1 1]
[1 0 1 1 1]
[0 1 1 1 1]
[1 1 1 1 0]
```

Incorrect facets that illustrate how this method works:

```
sage: facets = ((0,1,2,3), (0,1,2,3), (0,1,2,3), (0,1,2,3))
sage: face_list = facets_tuple_to_bit_rep_of_facets(facets, 4)
sage: face_list.matrix()
[1 1 1 1]
[1 1 1 1]
```

(continues on next page)

(continued from previous page)

```
[1 1 1 1]
[1 1 1 1]
sage: face_list.pyramid().matrix()
[1 1 1 1 1]
[1 1 1 1 1]
[1 1 1 1 1]
[1 1 1 1 1]
[1 1 1 1 0]
```

```
>>> from sage.all import *
>>> facets = ((Integer(0),Integer(1),Integer(2),Integer(3)), (Integer(0),
↪Integer(1),Integer(2),Integer(3)), (Integer(0),Integer(1),Integer(2),
↪Integer(3)), (Integer(0),Integer(1),Integer(2),Integer(3)))
>>> face_list = facets_tuple_to_bit_rep_of_facets(facets, Integer(4))
>>> face_list.matrix()
[1 1 1 1]
[1 1 1 1]
[1 1 1 1]
[1 1 1 1]
>>> face_list.pyramid().matrix()
[1 1 1 1 1]
[1 1 1 1 1]
[1 1 1 1 1]
[1 1 1 1 1]
[1 1 1 1 0]
```

```
sage: facets = ((), (), (), ())
sage: face_list = facets_tuple_to_bit_rep_of_facets(facets, 4)
sage: face_list.matrix()
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
sage: face_list.pyramid().matrix()
[0 0 0 0 1]
[0 0 0 0 1]
[0 0 0 0 1]
[0 0 0 0 1]
[1 1 1 1 0]
```

```
>>> from sage.all import *
>>> facets = ((), (), (), ())
>>> face_list = facets_tuple_to_bit_rep_of_facets(facets, Integer(4))
>>> face_list.matrix()
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
>>> face_list.pyramid().matrix()
[0 0 0 0 1]
[0 0 0 0 1]
```

(continues on next page)

(continued from previous page)

```
[0 0 0 0 1]
[0 0 0 0 1]
[1 1 1 1 0]
```

2.3.6 Conversions

This module provides conversions to *ListOfFaces* from - an incidence matrix of a polyhedron or - a tuple of facets (as tuple of vertices each).

Also this module provides a conversion from the data of *ListOfFaces*, which is a Bit-vector representing incidences of a face, to a list of entries which are incident.

See also:

list_of_faces, *face_iterator*, *base*.

EXAMPLES:

Obtain the facets of a polyhedron as *ListOfFaces*:

```
sage: from sage.geometry.polyhedron.combinatorial_polyhedron.conversions \
....:         import incidence_matrix_to_bit_rep_of_facets
sage: P = polytopes.simplex(4)
sage: inc = P.incidence_matrix()
sage: mod_inc = inc.delete_columns([i for i,V in enumerate(P.Hrepresentation()) if V.
↪is_equation()])
sage: face_list = incidence_matrix_to_bit_rep_of_facets(mod_inc)
sage: face_list.compute_dimension()
4
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.combinatorial_polyhedron.conversions         import␣
↪incidence_matrix_to_bit_rep_of_facets
>>> P = polytopes.simplex(Integer(4))
>>> inc = P.incidence_matrix()
>>> mod_inc = inc.delete_columns([i for i,V in enumerate(P.Hrepresentation()) if V.is_
↪equation()])
>>> face_list = incidence_matrix_to_bit_rep_of_facets(mod_inc)
>>> face_list.compute_dimension()
4
```

Obtain the Vrepresentation of a polyhedron as facet-incidences stored in *ListOfFaces*:

```
sage: # needs sage.combinat
sage: from sage.geometry.polyhedron.combinatorial_polyhedron.conversions \
....:         import incidence_matrix_to_bit_rep_of_Vrep
sage: P = polytopes.associahedron(['A',4])
sage: face_list = incidence_matrix_to_bit_rep_of_Vrep(P.incidence_matrix())
sage: face_list.compute_dimension()
4
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> from sage.geometry.polyhedron.combinatorial_polyhedron.conversions          import␣
↪incidence_matrix_to_bit_rep_of_Vrep
>>> P = polytopes.associahedron(['A',Integer(4)])
>>> face_list = incidence_matrix_to_bit_rep_of_Vrep(P.incidence_matrix())
>>> face_list.compute_dimension()
4
```

Obtain the facets of a polyhedron as `ListOfFaces` from a facet list:

```
sage: from sage.geometry.polyhedron.combinatorial_polyhedron.conversions \
....:         import facets_tuple_to_bit_rep_of_facets
sage: facets = ((0,1,2), (0,1,3), (0,2,3), (1,2,3))
sage: face_list = facets_tuple_to_bit_rep_of_facets(facets, 4)
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.combinatorial_polyhedron.conversions          import␣
↪facets_tuple_to_bit_rep_of_facets
>>> facets = ((Integer(0),Integer(1),Integer(2)), (Integer(0),Integer(1),Integer(3)),␣
↪(Integer(0),Integer(2),Integer(3)), (Integer(1),Integer(2),Integer(3)))
>>> face_list = facets_tuple_to_bit_rep_of_facets(facets, Integer(4))
```

Likewise for the Vrep as facet-incidences:

```
sage: from sage.geometry.polyhedron.combinatorial_polyhedron.conversions \
....:         import facets_tuple_to_bit_rep_of_Vrep
sage: facets = ((0,1,2), (0,1,3), (0,2,3), (1,2,3))
sage: face_list = facets_tuple_to_bit_rep_of_Vrep(facets, 4)
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.combinatorial_polyhedron.conversions          import␣
↪facets_tuple_to_bit_rep_of_Vrep
>>> facets = ((Integer(0),Integer(1),Integer(2)), (Integer(0),Integer(1),Integer(3)),␣
↪(Integer(0),Integer(2),Integer(3)), (Integer(1),Integer(2),Integer(3)))
>>> face_list = facets_tuple_to_bit_rep_of_Vrep(facets, Integer(4))
```

AUTHOR:

- Jonathan Kliem (2019-04)

`sage.geometry.polyhedron.combinatorial_polyhedron.conversions.`**`facets_tuple_to_bit_rep_of_Vrep`**(*facets_input*, *n_Vrep*)

Initialize Vrepresentatives in Bit-representation as `ListOfFaces`.

Each Vrepresentative is represented as the facets it is contained in. Those are the facets of the polar polyhedron, if it exists.

INPUT:

- `facets_input` – tuple of facets, each facet a tuple of Vrep, Vrep must be exactly `range(n_Vrep)`
- `n_Vrep`

OUTPUT: `ListOfFaces`

EXAMPLES:

```
sage: from sage.geometry.polyhedron.combinatorial_polyhedron.conversions \
....:     import facets_tuple_to_bit_rep_of_Vrep, \
....:            _bit_rep_to_Vrep_list_wrapper
sage: bi_pyr = ((0,1,4), (1,2,4), (2,3,4), (3,0,4),
....:           (0,1,5), (1,2,5), (2,3,5), (3,0,5))
sage: vertices = facets_tuple_to_bit_rep_of_Vrep(bi_pyr, 6)
sage: for i in range(6):
....:     print(_bit_rep_to_Vrep_list_wrapper(vertices, i))
(0, 3, 4, 7)
(0, 1, 4, 5)
(1, 2, 5, 6)
(2, 3, 6, 7)
(0, 1, 2, 3)
(4, 5, 6, 7)
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.combinatorial_polyhedron.conversions     import␣
↪facets_tuple_to_bit_rep_of_Vrep,            _bit_rep_to_Vrep_list_wrapper
>>> bi_pyr = ((Integer(0),Integer(1),Integer(4)), (Integer(1),Integer(2),
↪Integer(4)), (Integer(2),Integer(3),Integer(4)), (Integer(3),Integer(0),
↪Integer(4)),
...           (Integer(0),Integer(1),Integer(5)), (Integer(1),Integer(2),
↪Integer(5)), (Integer(2),Integer(3),Integer(5)), (Integer(3),Integer(0),
↪Integer(5)))
>>> vertices = facets_tuple_to_bit_rep_of_Vrep(bi_pyr, Integer(6))
>>> for i in range(Integer(6)):
...     print(_bit_rep_to_Vrep_list_wrapper(vertices, i))
(0, 3, 4, 7)
(0, 1, 4, 5)
(1, 2, 5, 6)
(2, 3, 6, 7)
(0, 1, 2, 3)
(4, 5, 6, 7)
```

`sage.geometry.polyhedron.combinatorial_polyhedron.conversions.`**`facets_tuple_to_bit_rep_of_facets`**(*facets_input*, *n_Vrep*)

Initialize facets in Bit-representation as `ListOfFaces`.

INPUT:

- `facets_input` – tuple of facets, each facet a tuple of Vrep, Vrep must be exactly `range(n_Vrep)`
- `n_Vrep`

OUTPUT: `ListOfFaces`

EXAMPLES:

```
sage: from sage.geometry.polyhedron.combinatorial_polyhedron.conversions \
....:     import facets_tuple_to_bit_rep_of_facets, \
....:            _bit_rep_to_Vrep_list_wrapper
sage: bi_pyr = ((0,1,4), (1,2,4), (2,3,4), (3,0,4),
```

(continues on next page)

(continued from previous page)

```
....:            (0,1,5), (1,2,5), (2,3,5), (3,0,5))
sage: facets = facets_tuple_to_bit_rep_of_facets(bi_pyr, 6)
sage: for i in range(8):
....:     print(_bit_rep_to_Vrep_list_wrapper(facets, i))
(0, 1, 4)
(1, 2, 4)
(2, 3, 4)
(0, 3, 4)
(0, 1, 5)
(1, 2, 5)
(2, 3, 5)
(0, 3, 5)
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.combinatorial_polyhedron.conversions     import␣
↪facets_tuple_to_bit_rep_of_facets,            _bit_rep_to_Vrep_list_wrapper
>>> bi_pyr = ((Integer(0),Integer(1),Integer(4)), (Integer(1),Integer(2),
↪Integer(4)), (Integer(2),Integer(3),Integer(4)), (Integer(3),Integer(0),
↪Integer(4)),
...            (Integer(0),Integer(1),Integer(5)), (Integer(1),Integer(2),
↪Integer(5)), (Integer(2),Integer(3),Integer(5)), (Integer(3),Integer(0),
↪Integer(5)))
>>> facets = facets_tuple_to_bit_rep_of_facets(bi_pyr, Integer(6))
>>> for i in range(Integer(8)):
...     print(_bit_rep_to_Vrep_list_wrapper(facets, i))
(0, 1, 4)
(1, 2, 4)
(2, 3, 4)
(0, 3, 4)
(0, 1, 5)
(1, 2, 5)
(2, 3, 5)
(0, 3, 5)
```

sage.geometry.polyhedron.combinatorial_polyhedron.conversions.**incidence_matrix_to_bit_rep_of_Vrep**(*matrix*)

Initialize Vrepresentatives in Bit-representation as `ListOfFaces`.

Each Vrepresentative is represented as the facets it is contained in. Those are the facets of the polar polyhedron, if it exists.

INPUT:

- `matrix` – an incidence matrix as in `sage.geometry.polyhedron.base.Polyhedron_base.incidence_matrix()` with columns corresponding to equations deleted of type `sage.matrix.matrix_dense.Matrix_dense`

OUTPUT: `ListOfFaces`

EXAMPLES:

```
sage: from sage.geometry.polyhedron.combinatorial_polyhedron.conversions \
....:     import incidence_matrix_to_bit_rep_of_Vrep, \
....:            _bit_rep_to_Vrep_list_wrapper
```

(continues on next page)

(continued from previous page)

```
sage: P = polytopes.permutahedron(4)
sage: inc = P.incidence_matrix()
sage: mod_inc = inc.delete_columns([i for i,V in enumerate(P.Hrepresentation())␣
↪if V.is_equation()])
sage: vertices = incidence_matrix_to_bit_rep_of_Vrep(mod_inc)
sage: vertices.matrix().dimensions()
(24, 14)
sage: for row in vertices.matrix():
....:     row.nonzero_positions()
[8, 9, 11]
[8, 10, 11]
[2, 3, 7]
[1, 5, 7]
[4, 5, 7]
[1, 3, 7]
[4, 6, 7]
[2, 6, 7]
[1, 5, 13]
[8, 9, 13]
[1, 9, 11]
[2, 10, 11]
[1, 3, 11]
[2, 3, 11]
[4, 5, 13]
[4, 12, 13]
[8, 12, 13]
[1, 9, 13]
[0, 8, 12]
[0, 4, 12]
[0, 2, 10]
[0, 2, 6]
[0, 8, 10]
[0, 4, 6]
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.combinatorial_polyhedron.conversions     import␣
↪incidence_matrix_to_bit_rep_of_Vrep,            _bit_rep_to_Vrep_list_wrapper
>>> P = polytopes.permutahedron(Integer(4))
>>> inc = P.incidence_matrix()
>>> mod_inc = inc.delete_columns([i for i,V in enumerate(P.Hrepresentation()) if␣
↪V.is_equation()])
>>> vertices = incidence_matrix_to_bit_rep_of_Vrep(mod_inc)
>>> vertices.matrix().dimensions()
(24, 14)
>>> for row in vertices.matrix():
...     row.nonzero_positions()
[8, 9, 11]
[8, 10, 11]
[2, 3, 7]
[1, 5, 7]
[4, 5, 7]
[1, 3, 7]
```

(continues on next page)

(continued from previous page)

```
[4, 6, 7]
[2, 6, 7]
[1, 5, 13]
[8, 9, 13]
[1, 9, 11]
[2, 10, 11]
[1, 3, 11]
[2, 3, 11]
[4, 5, 13]
[4, 12, 13]
[8, 12, 13]
[1, 9, 13]
[0, 8, 12]
[0, 4, 12]
[0, 2, 10]
[0, 2, 6]
[0, 8, 10]
[0, 4, 6]
```

sage.geometry.polyhedron.combinatorial_polyhedron.conversions.**incidence_matrix_to_bit_rep_of_facets**(

Initialize facets in Bit-representation as *ListOfFaces*.

INPUT:

- matrix – an incidence matrix as in sage.geometry.polyhedron.base.Polyhedron_base.incidence_matrix() with columns corresponding to equations deleted of type sage.matrix.matrix_dense.Matrix_dense

OUTPUT: *ListOfFaces*

EXAMPLES:

```
sage: from sage.geometry.polyhedron.combinatorial_polyhedron.conversions \
....:     import incidence_matrix_to_bit_rep_of_facets, \
....:            _bit_rep_to_Vrep_list_wrapper
sage: P = polytopes.permutahedron(4)
sage: inc = P.incidence_matrix()
sage: mod_inc = inc.delete_columns([i for i,V in enumerate(P.Hrepresentation())␣
↪if V.is_equation()])
sage: facets = incidence_matrix_to_bit_rep_of_facets(mod_inc)
sage: facets.matrix().dimensions()
(14, 24)
sage: for row in facets.matrix():
....:     row.nonzero_positions()
[18, 19, 20, 21, 22, 23]
[3, 5, 8, 10, 12, 17]
[2, 7, 11, 13, 20, 21]
[2, 5, 12, 13]
[4, 6, 14, 15, 19, 23]
[3, 4, 8, 14]
[6, 7, 21, 23]
[2, 3, 4, 5, 6, 7]
[0, 1, 9, 16, 18, 22]
```

(continues on next page)

(continued from previous page)

```
[0, 9, 10, 17]
[1, 11, 20, 22]
[0, 1, 10, 11, 12, 13]
[15, 16, 18, 19]
[8, 9, 14, 15, 16, 17]
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.combinatorial_polyhedron.conversions     import␣
↪incidence_matrix_to_bit_rep_of_facets,            _bit_rep_to_Vrep_list_wrapper
>>> P = polytopes.permutahedron(Integer(4))
>>> inc = P.incidence_matrix()
>>> mod_inc = inc.delete_columns([i for i,V in enumerate(P.Hrepresentation()) if␣
↪V.is_equation()])
>>> facets = incidence_matrix_to_bit_rep_of_facets(mod_inc)
>>> facets.matrix().dimensions()
(14, 24)
>>> for row in facets.matrix():
...     row.nonzero_positions()
[18, 19, 20, 21, 22, 23]
[3, 5, 8, 10, 12, 17]
[2, 7, 11, 13, 20, 21]
[2, 5, 12, 13]
[4, 6, 14, 15, 19, 23]
[3, 4, 8, 14]
[6, 7, 21, 23]
[2, 3, 4, 5, 6, 7]
[0, 1, 9, 16, 18, 22]
[0, 9, 10, 17]
[1, 11, 20, 22]
[0, 1, 10, 11, 12, 13]
[15, 16, 18, 19]
[8, 9, 14, 15, 16, 17]
```

2.4 Polyhedral complexes

2.4.1 Finite polyhedral complexes

This module implements the basic structure of finite polyhedral complexes. For more information, see `PolyhedralComplex`.

AUTHORS:

- Yuan Zhou (2021-05): initial implementation

List of PolyhedralComplex methods

Maximal cells and cells

maximal_cells()	Return the dictionary of the maximal cells in this polyhedral complex.
maximal_cell_iterator()	Return an iterator over maximal cells in this polyhedral complex.
maximal_cells_sorted()	Return the sorted list of all maximal cells in this polyhedral complex.
n_maximal_cells()	List the maximal cells of dimension n in this polyhedral complex.
_n_maximal_cells_sorted()	Return the sorted list of maximal cells of dim n in this complex.
is_maximal_cell()	Return True if the given cell is a maximal cell in this complex.
cells()	Return the dictionary of the cells in this polyhedral complex.
cell_iterator()	Return an iterator over cells in this polyhedral complex.
cells_sorted()	Return the sorted list of all cells in this polyhedral complex.
n_cells()	List the cells of dimension n in this polyhedral complex.
_n_cells_sorted()	Return the sorted list of n-cells in this polyhedral complex.
is_cell()	Return True if the given cell is in this polyhedral complex.
face_poset()	Return the poset of nonempty cells in the polyhedral complex.
relative_boundary_cells()	List the maximal cells on the boundary of the polyhedral complex.

Properties of the polyhedral complex

dimension()	Return the dimension of the polyhedral complex.
ambient_dimension()	Return the ambient dimension of the polyhedral complex.
is_pure()	Return True if the polyhedral complex is pure.
is_full_dimensional()	Return True if the polyhedral complex is full dimensional.
is_compact()	Return True if the polyhedral complex is bounded.
is_connected()	Return True if the polyhedral complex is connected.
is_subcomplex()	Return True if this complex is a subcomplex of the other.
is_convex()	Return True if the polyhedral complex is convex.
is_mutable()	Return True if the polyhedral complex is mutable.
is_immutable()	Return True if the polyhedral complex is not mutable.
is_simplicial_complex()	Return True if the polyhedral complex is a simplicial complex.
is_polyhedral_fan()	Return True if the polyhedral complex is a fan.
is_simplicial_fan()	Return True if the polyhedral complex is a simplicial fan.

New polyhedral complexes from old ones

connected_component()	Return the connected component containing a cell as a subcomplex.
connected_components()	Return the connected components of this polyhedral complex.
n_skeleton()	Return the n-skeleton of this polyhedral complex.
stratify()	Return the (pure) subcomplex formed by the maximal cells of dim n in this complex.
boundary_subcomplex()	Return the boundary subcomplex of this polyhedral complex.
product()	Return the (Cartesian) product of this polyhedral complex with another one.
disjoint_union()	Return the disjoint union of this polyhedral complex with another one.
union()	Return the union of this polyhedral complex with another one.
join()	Return the join of this polyhedral complex with another one.
subdivide()	Return a new polyhedral complex (with option make_simplicial) subdividing this one.

Update polyhedral complex

`set_immutable()`	Make this polyhedral complex immutable.
`add_cell()`	Add a cell to this polyhedral complex.
`remove_cell()`	Remove a cell from this polyhedral complex.

Miscellaneous

`plot()`	Return a Graphic object showing the plot of polyhedral complex.
`graph()`	Return a directed graph corresponding to the 1-skeleton of this polyhedral complex, given that it is bounded.
`union_as_polyhedron()`	Return a `Polyhedron` which is the union of cells in this polyhedral complex, given that it is convex.

Classes and functions

class sage.geometry.polyhedral_complex.**PolyhedralComplex**(*maximal_cells=None*, *backend=None*, *maximality_check=True*, *face_to_face_check=False*, *is_mutable=True*, *is_immutable=False*, *ambient_dim=None*)

Bases: `GenericCellComplex`

A polyhedral complex.

A **polyhedral complex** PC is a collection of polyhedra in a certain ambient space $\mathbf{R}^n$ such that the following hold.

- If a polyhedron P is in PC, then all the faces of P are in PC.
- If polyhedra P and Q are in PC, then $P \cap Q$ is either empty or a face of both P and Q.

In this context, a "polyhedron" means the geometric realization of a polyhedron. This is in contrast to `simplicial complex`, whose cells are abstract simplices. The concept of a polyhedral complex generalizes that of a **geometric** simplicial complex.

> **Note**
>
> This class derives from `GenericCellComplex`, and so inherits its methods. Some of those methods are not listed here; see the `Generic Cell Complex` page instead.

INPUT:

- `maximal_cells` – list, tuple, or dictionary (indexed by dimension) of cells of the Complex. Each cell is of class `Polyhedron` of the same ambient dimension. To set up a :class:PolyhedralComplex, it is sufficient to provide the maximal faces. Use keyword argument `partial=True` to set up a partial polyhedral complex, which is a subset of the faces (viewed as relatively open) of a polyhedral complex that is not necessarily closed under taking intersection.
- `maximality_check` – boolean (default: `True`); if `True`, then the constructor checks that each given maximal cell is indeed maximal, and ignores those that are not
- `face_to_face_check` – boolean (default: `False`); if `True`, then the constructor checks whether the cells are face-to-face, and it raises a `ValueError` if they are not
- `is_mutable` and `is_immutable` – boolean (default: `True` and `False` respectively); set `is_mutable=False` or `is_immutable=True` to make this polyhedral complex immutable

- `backend` – string (optional); the name of the backend used for computations on Sage polyhedra; if it is not given, then each cell has its own backend; otherwise it must be one of the following:
  - `'ppl'` – the Parma Polyhedra Library
  - `'cdd'` – CDD
  - `'normaliz'` – normaliz
  - `'polymake'` – polymake
  - `'field'` – a generic Sage implementation
- `ambient_dim` – integer (optional); used to set up an empty complex in the intended ambient space

EXAMPLES:

```
sage: pc = PolyhedralComplex([
....:         Polyhedron(vertices=[(1/3, 1/3), (0, 0), (1/7, 2/7)]),
....:         Polyhedron(vertices=[(1/7, 2/7), (0, 0), (0, 1/4)])])
sage: [p.Vrepresentation() for p in pc.cells_sorted()]
[(A vertex at (0, 0), A vertex at (0, 1/4), A vertex at (1/7, 2/7)),
 (A vertex at (0, 0), A vertex at (1/3, 1/3), A vertex at (1/7, 2/7)),
 (A vertex at (0, 0), A vertex at (0, 1/4)),
 (A vertex at (0, 0), A vertex at (1/7, 2/7)),
 (A vertex at (0, 0), A vertex at (1/3, 1/3)),
 (A vertex at (0, 1/4), A vertex at (1/7, 2/7)),
 (A vertex at (1/3, 1/3), A vertex at (1/7, 2/7)),
 (A vertex at (0, 0),),
 (A vertex at (0, 1/4),),
 (A vertex at (1/7, 2/7),),
 (A vertex at (1/3, 1/3),)]
sage: pc.plot()                                                              #␣
↪needs sage.plot
Graphics object consisting of 10 graphics primitives
sage: pc.is_pure()
True
sage: pc.is_full_dimensional()
True
sage: pc.is_compact()
True
sage: pc.boundary_subcomplex()
Polyhedral complex with 4 maximal cells
sage: pc.is_convex()
True
sage: pc.union_as_polyhedron().Hrepresentation()
(An inequality (1, -4) x + 1 >= 0,
 An inequality (-1, 1) x + 0 >= 0,
 An inequality (1, 0) x + 0 >= 0)
sage: pc.face_poset()
Finite poset containing 11 elements
sage: pc.is_connected()
True
sage: pc.connected_component() == pc
True
```

```
>>> from sage.all import *
>>> pc = PolyhedralComplex([
...         Polyhedron(vertices=[(Integer(1)/Integer(3), Integer(1)/Integer(3)),␣
↪(Integer(0), Integer(0)), (Integer(1)/Integer(7), Integer(2)/Integer(7))]),
...         Polyhedron(vertices=[(Integer(1)/Integer(7), Integer(2)/Integer(7)),␣
↪(Integer(0), Integer(0)), (Integer(0), Integer(1)/Integer(4))])])
>>> [p.Vrepresentation() for p in pc.cells_sorted()]
[(A vertex at (0, 0), A vertex at (0, 1/4), A vertex at (1/7, 2/7)),
 (A vertex at (0, 0), A vertex at (1/3, 1/3), A vertex at (1/7, 2/7)),
 (A vertex at (0, 0), A vertex at (0, 1/4)),
 (A vertex at (0, 0), A vertex at (1/7, 2/7)),
 (A vertex at (0, 0), A vertex at (1/3, 1/3)),
 (A vertex at (0, 1/4), A vertex at (1/7, 2/7)),
 (A vertex at (1/3, 1/3), A vertex at (1/7, 2/7)),
 (A vertex at (0, 0),),
 (A vertex at (0, 1/4),),
 (A vertex at (1/7, 2/7),),
 (A vertex at (1/3, 1/3),)]
>>> pc.plot()                                                               #␣
↪needs sage.plot
Graphics object consisting of 10 graphics primitives
>>> pc.is_pure()
True
>>> pc.is_full_dimensional()
True
>>> pc.is_compact()
True
>>> pc.boundary_subcomplex()
Polyhedral complex with 4 maximal cells
>>> pc.is_convex()
True
>>> pc.union_as_polyhedron().Hrepresentation()
(An inequality (1, -4) x + 1 >= 0,
 An inequality (-1, 1) x + 0 >= 0,
 An inequality (1, 0) x + 0 >= 0)
>>> pc.face_poset()
Finite poset containing 11 elements
>>> pc.is_connected()
True
>>> pc.connected_component() == pc
True
```

add_cell(*cell*)

Add a cell to this polyhedral complex.

INPUT:

- `cell` – a polyhedron

This **changes** the polyhedral complex, by adding a new cell and all of its subfaces.

EXAMPLES:

Set up an empty complex in the intended ambient space, then add a cell:

```
sage: pc = PolyhedralComplex(ambient_dim=2)
sage: pc.add_cell(Polyhedron(vertices=[(1, 2), (0, 2)]))
sage: pc
Polyhedral complex with 1 maximal cell
```

```
>>> from sage.all import *
>>> pc = PolyhedralComplex(ambient_dim=Integer(2))
>>> pc.add_cell(Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
→Integer(2))]))
>>> pc
Polyhedral complex with 1 maximal cell
```

If you add a cell which is already present, there is no effect:

```
sage: pc.add_cell(Polyhedron(vertices=[(1, 2)]))
sage: pc
Polyhedral complex with 1 maximal cell
sage: pc.dimension()
1
```

```
>>> from sage.all import *
>>> pc.add_cell(Polyhedron(vertices=[(Integer(1), Integer(2))]))
>>> pc
Polyhedral complex with 1 maximal cell
>>> pc.dimension()
1
```

Add a cell and check that dimension is correctly updated:

```
sage: pc.add_cell(Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)]))
sage: pc.dimension()
2
sage: pc.maximal_cells()
{2: {A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 3␣
→vertices}}
sage: pc.is_convex()
True
```

```
>>> from sage.all import *
>>> pc.add_cell(Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
→Integer(0)), (Integer(0), Integer(2))]))
>>> pc.dimension()
2
>>> pc.maximal_cells()
{2: {A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 3␣
→vertices}}
>>> pc.is_convex()
True
```

Add another cell and check that the properties are correctly updated:

```
sage: pc.add_cell(Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)]))
sage: pc
```

(continues on next page)

(continued from previous page)

```
Polyhedral complex with 2 maximal cells
sage: len(pc._cells[1])
5
sage: pc._face_poset
Finite poset containing 11 elements
sage: pc._is_convex
True
sage: pc._polyhedron.vertices_list()
[[0, 0], [0, 2], [1, 1], [1, 2]]
```

```
>>> from sage.all import *
>>> pc.add_cell(Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0),␣
↪Integer(0)), (Integer(1), Integer(2))]))
>>> pc
Polyhedral complex with 2 maximal cells
>>> len(pc._cells[Integer(1)])
5
>>> pc._face_poset
Finite poset containing 11 elements
>>> pc._is_convex
True
>>> pc._polyhedron.vertices_list()
[[0, 0], [0, 2], [1, 1], [1, 2]]
```

Add a ray which makes the complex non convex:

```
sage: pc.add_cell(Polyhedron(rays=[(1, 0)]))
sage: pc
Polyhedral complex with 3 maximal cells
sage: len(pc._cells[1])
6
sage: (pc._is_convex is False) and (pc._polyhedron is None)
True
```

```
>>> from sage.all import *
>>> pc.add_cell(Polyhedron(rays=[(Integer(1), Integer(0))]))
>>> pc
Polyhedral complex with 3 maximal cells
>>> len(pc._cells[Integer(1)])
6
>>> (pc._is_convex is False) and (pc._polyhedron is None)
True
```

alexander_whitney(*cell*, *dim_left*)

The decomposition of `cell` in this complex into left and right factors, suitable for computing cup products.

Todo

Implement *alexander_whitney()* of a polyhedral complex.

EXAMPLES:

```
sage: pc = PolyhedralComplex([Polyhedron(vertices=[[0], [1]])])
sage: pc.alexander_whitney(None, 1)
Traceback (most recent call last):
...
NotImplementedError: alexander_whitney is not implemented for polyhedral␣
↪complex
```

```
>>> from sage.all import *
>>> pc = PolyhedralComplex([Polyhedron(vertices=[[Integer(0)],␣
↪[Integer(1)]])])
>>> pc.alexander_whitney(None, Integer(1))
Traceback (most recent call last):
...
NotImplementedError: alexander_whitney is not implemented for polyhedral␣
↪complex
```

ambient_dimension()

The ambient dimension of this cell complex: the ambient dimension of each of its cells.

EXAMPLES:

```
sage: pc = PolyhedralComplex([Polyhedron(vertices=[(1, 2, 3)])])
sage: pc.ambient_dimension()
3
sage: empty_pc = PolyhedralComplex([])
sage: empty_pc.ambient_dimension()
-1
sage: pc0 = PolyhedralComplex(ambient_dim=2)
sage: pc0.ambient_dimension()
2
```

```
>>> from sage.all import *
>>> pc = PolyhedralComplex([Polyhedron(vertices=[(Integer(1), Integer(2),␣
↪Integer(3))])])
>>> pc.ambient_dimension()
3
>>> empty_pc = PolyhedralComplex([])
>>> empty_pc.ambient_dimension()
-1
>>> pc0 = PolyhedralComplex(ambient_dim=Integer(2))
>>> pc0.ambient_dimension()
2
```

boundary_subcomplex()

Return the sub-polyhedral complex that is the boundary of `self`.

A point P is on the boundary of a set S if P is in the closure of S but not in the interior of S.

EXAMPLES:

```
sage: p1 = Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)])
sage: p2 = Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])
sage: p3 = Polyhedron(vertices=[(1, 2), (0, 2)])
sage: bd = PolyhedralComplex([p1, p2]).boundary_subcomplex()
```

(continues on next page)

(continued from previous page)

```
sage: len(bd.n_maximal_cells(2))
0
sage: len(bd.n_maximal_cells(1))
4
sage: pt = PolyhedralComplex([p3])
sage: pt.boundary_subcomplex() == pt
True
```

```
>>> from sage.all import *
>>> p1 = Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0),␣
→Integer(0)), (Integer(1), Integer(2))])
>>> p2 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
→Integer(0)), (Integer(0), Integer(2))])
>>> p3 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
→Integer(2))])
>>> bd = PolyhedralComplex([p1, p2]).boundary_subcomplex()
>>> len(bd.n_maximal_cells(Integer(2)))
0
>>> len(bd.n_maximal_cells(Integer(1)))
4
>>> pt = PolyhedralComplex([p3])
>>> pt.boundary_subcomplex() == pt
True
```

Test on polyhedral complex which is not pure:

```
sage: pc_non_pure = PolyhedralComplex([p1, p3])
sage: pc_non_pure.boundary_subcomplex() == pc_non_pure.n_skeleton(1)
True
```

```
>>> from sage.all import *
>>> pc_non_pure = PolyhedralComplex([p1, p3])
>>> pc_non_pure.boundary_subcomplex() == pc_non_pure.n_skeleton(Integer(1))
True
```

Test with `maximality_check == False`:

```
sage: pc_invalid = PolyhedralComplex([p2, p3],
....:                                 maximality_check=False)
sage: pc_invalid.boundary_subcomplex() == pc_invalid.n_skeleton(1)
True
```

```
>>> from sage.all import *
>>> pc_invalid = PolyhedralComplex([p2, p3],
...                                 maximality_check=False)
>>> pc_invalid.boundary_subcomplex() == pc_invalid.n_skeleton(Integer(1))
True
```

Test unbounded cases:

```
sage: pc1 = PolyhedralComplex([
....:         Polyhedron(vertices=[[1,0], [0,1]], rays=[[1,0], [0,1]])])
```

(continues on next page)

(continued from previous page)

```
sage: pc1.boundary_subcomplex() == pc1.n_skeleton(1)
True
sage: pc1b = PolyhedralComplex([Polyhedron(
....:         vertices=[[1,0,0], [0,1,0]], rays=[[1,0,0],[0,1,0]])])
sage: pc1b.boundary_subcomplex() == pc1b
True
sage: pc2 = PolyhedralComplex([
....:         Polyhedron(vertices=[[-1,0], [1,0]], lines=[[0,1]])])
sage: pc2.boundary_subcomplex() == pc2.n_skeleton(1)
True
sage: pc3 = PolyhedralComplex([
....:         Polyhedron(vertices=[[1,0], [0,1]], rays=[[1,0], [0,1]]),
....:         Polyhedron(vertices=[[1,0], [0,-1]], rays=[[1,0], [0,-1]])])
sage: pc3.boundary_subcomplex() == pc3.n_skeleton(1)
False
```

```
>>> from sage.all import *
>>> pc1 = PolyhedralComplex([
...         Polyhedron(vertices=[[Integer(1),Integer(0)], [Integer(0),
↪Integer(1)]], rays=[[Integer(1),Integer(0)], [Integer(0),Integer(1)]])])
>>> pc1.boundary_subcomplex() == pc1.n_skeleton(Integer(1))
True
>>> pc1b = PolyhedralComplex([Polyhedron(
...         vertices=[[Integer(1),Integer(0),Integer(0)], [Integer(0),
↪Integer(1),Integer(0)]], rays=[[Integer(1),Integer(0),Integer(0)],
↪[Integer(0),Integer(1),Integer(0)]])])
>>> pc1b.boundary_subcomplex() == pc1b
True
>>> pc2 = PolyhedralComplex([
...         Polyhedron(vertices=[[-Integer(1),Integer(0)], [Integer(1),
↪Integer(0)]], lines=[[Integer(0),Integer(1)]])])
>>> pc2.boundary_subcomplex() == pc2.n_skeleton(Integer(1))
True
>>> pc3 = PolyhedralComplex([
...         Polyhedron(vertices=[[Integer(1),Integer(0)], [Integer(0),
↪Integer(1)]], rays=[[Integer(1),Integer(0)], [Integer(0),Integer(1)]]),
...         Polyhedron(vertices=[[Integer(1),Integer(0)], [Integer(0),-
↪Integer(1)]], rays=[[Integer(1),Integer(0)], [Integer(0),-Integer(1)]])])
>>> pc3.boundary_subcomplex() == pc3.n_skeleton(Integer(1))
False
```

cell_iterator(*increasing=True*)

An iterator for the cells in this polyhedral complex.

INPUT:

- `increasing` – boolean (default: `True`); if `True`, return cells in increasing order of dimension, thus starting with the zero-dimensional cells; otherwise it returns cells in decreasing order of dimension

> **Note**
>
> Among the cells of a fixed dimension, there is no sorting.

EXAMPLES:

```
sage: pc = PolyhedralComplex([
....:         Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)]),
....:         Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])])
sage: len(list(pc.cell_iterator()))
11
```

```
>>> from sage.all import *
>>> pc = PolyhedralComplex([
...         Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0),␣
↪Integer(0)), (Integer(1), Integer(2))]),
...         Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
↪Integer(0)), (Integer(0), Integer(2))])])
>>> len(list(pc.cell_iterator()))
11
```

cells(*subcomplex=None*)

The cells of this polyhedral complex, in the form of a dictionary: the keys are integers, representing dimension, and the value associated to an integer d is the set of d-cells.

INPUT:

- `subcomplex` – (optional) if a subcomplex is given then return the cells which are **not** in this subcomplex

EXAMPLES:

```
sage: pc = PolyhedralComplex([
....:         Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)]),
....:         Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])])
sage: list(pc.cells().keys())
[2, 1, 0]
```

```
>>> from sage.all import *
>>> pc = PolyhedralComplex([
...         Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0),␣
↪Integer(0)), (Integer(1), Integer(2))]),
...         Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
↪Integer(0)), (Integer(0), Integer(2))])])
>>> list(pc.cells().keys())
[2, 1, 0]
```

cells_sorted(*subcomplex=None*)

The sorted list of the cells of this polyhedral complex in non-increasing dimensions.

INPUT:

- `subcomplex` – (optional) if a subcomplex is given then return the cells which are **not** in this subcomplex

EXAMPLES:

```
sage: pc = PolyhedralComplex([
....: Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)]),
....: Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])])
sage: len(pc.cells_sorted())
11
```

(continues on next page)

(continued from previous page)

```
sage: pc.cells_sorted()[0].Vrepresentation()
(A vertex at (0, 0), A vertex at (0, 2), A vertex at (1, 2))
```

```
>>> from sage.all import *
>>> pc = PolyhedralComplex([
... Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0), Integer(0)),␣
↪(Integer(1), Integer(2))]),
... Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0), Integer(0)),␣
↪(Integer(0), Integer(2))])])
>>> len(pc.cells_sorted())
11
>>> pc.cells_sorted()[Integer(0)].Vrepresentation()
(A vertex at (0, 0), A vertex at (0, 2), A vertex at (1, 2))
```

chain_complex(*subcomplex=None*, *augmented=False*, *verbose=False*, *check=True*, *dimensions=None*, *base_ring=Integer Ring*, *cochain=False*)

The chain complex associated to this polyhedral complex.

Todo

Implement chain complexes of a polyhedral complex.

EXAMPLES:

```
sage: pc = PolyhedralComplex([Polyhedron(vertices=[[0], [1]])])
sage: pc.chain_complex()
Traceback (most recent call last):
...
NotImplementedError: chain_complex is not implemented for polyhedral complex
```

```
>>> from sage.all import *
>>> pc = PolyhedralComplex([Polyhedron(vertices=[[Integer(0)],␣
↪[Integer(1)]])])
>>> pc.chain_complex()
Traceback (most recent call last):
...
NotImplementedError: chain_complex is not implemented for polyhedral complex
```

connected_component(*cell=None*)

Return the connected component of this polyhedral complex containing a given cell.

INPUT:

- `cell` – (default: `self.an_element()`) a cell of `self`

OUTPUT:

The connected component containing `cell`. If the polyhedral complex is empty or if it does not contain the given cell, raise an error.

EXAMPLES:

```
sage: t1 = Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)])
sage: t2 = Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])
sage: v1 = Polyhedron(vertices=[(1, 1)])
sage: v2 = Polyhedron(vertices=[(0, 2)])
sage: v3 = Polyhedron(vertices=[(-1, 0)])
sage: o =  Polyhedron(vertices=[(0, 0)])
sage: r = Polyhedron(rays=[(1, 0)])
sage: l = Polyhedron(vertices=[(-1, 0)], lines=[(1, -1)])
sage: pc1 = PolyhedralComplex([t1, t2])
sage: pc1.connected_component() == pc1
True
sage: pc1.connected_component(v1) == pc1
True
sage: pc2 = PolyhedralComplex([t1, v2])
sage: pc2.connected_component(t1) == PolyhedralComplex([t1])
True
sage: pc2.connected_component(o) == PolyhedralComplex([t1])
True
sage: pc2.connected_component(v3)
Traceback (most recent call last):
...
ValueError: the polyhedral complex does not contain the given cell
sage: pc2.connected_component(r)
Traceback (most recent call last):
...
ValueError: the polyhedral complex does not contain the given cell
sage: pc3 = PolyhedralComplex([t1, t2, r])
sage: pc3.connected_component(v2) == pc3
True
sage: pc4 = PolyhedralComplex([t1, t2, r, l])
sage: pc4.connected_component(o) == pc3
True
sage: pc4.connected_component(v3)
Traceback (most recent call last):
...
ValueError: the polyhedral complex does not contain the given cell
sage: pc5 = PolyhedralComplex([t1, t2, r, l, v3])
sage: pc5.connected_component(v3) == PolyhedralComplex([v3])
True
sage: PolyhedralComplex([]).connected_component()
Traceback (most recent call last):
...
ValueError: the empty polyhedral complex has no connected components
```

```
>>> from sage.all import *
>>> t1 = Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0),␣
↪Integer(0)), (Integer(1), Integer(2))])
>>> t2 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
↪Integer(0)), (Integer(0), Integer(2))])
>>> v1 = Polyhedron(vertices=[(Integer(1), Integer(1))])
>>> v2 = Polyhedron(vertices=[(Integer(0), Integer(2))])
>>> v3 = Polyhedron(vertices=[(-Integer(1), Integer(0))])
```

(continues on next page)

(continued from previous page)

```
>>> o =  Polyhedron(vertices=[(Integer(0), Integer(0))])
>>> r = Polyhedron(rays=[(Integer(1), Integer(0))])
>>> l = Polyhedron(vertices=[(-Integer(1), Integer(0))], lines=[(Integer(1), -
↪Integer(1))])
>>> pc1 = PolyhedralComplex([t1, t2])
>>> pc1.connected_component() == pc1
True
>>> pc1.connected_component(v1) == pc1
True
>>> pc2 = PolyhedralComplex([t1, v2])
>>> pc2.connected_component(t1) == PolyhedralComplex([t1])
True
>>> pc2.connected_component(o) == PolyhedralComplex([t1])
True
>>> pc2.connected_component(v3)
Traceback (most recent call last):
...
ValueError: the polyhedral complex does not contain the given cell
>>> pc2.connected_component(r)
Traceback (most recent call last):
...
ValueError: the polyhedral complex does not contain the given cell
>>> pc3 = PolyhedralComplex([t1, t2, r])
>>> pc3.connected_component(v2) == pc3
True
>>> pc4 = PolyhedralComplex([t1, t2, r, l])
>>> pc4.connected_component(o) == pc3
True
>>> pc4.connected_component(v3)
Traceback (most recent call last):
...
ValueError: the polyhedral complex does not contain the given cell
>>> pc5 = PolyhedralComplex([t1, t2, r, l, v3])
>>> pc5.connected_component(v3) == PolyhedralComplex([v3])
True
>>> PolyhedralComplex([]).connected_component()
Traceback (most recent call last):
...
ValueError: the empty polyhedral complex has no connected components
```

connected_components()

Return the connected components of this polyhedral complex, as list of (sub-)PolyhedralComplexes.

EXAMPLES:

```
sage: t1 = Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)])
sage: t2 = Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])
sage: v1 = Polyhedron(vertices=[(1, 1)])
sage: v2 = Polyhedron(vertices=[(0, 2)])
sage: v3 = Polyhedron(vertices=[(-1, 0)])
sage: o =  Polyhedron(vertices=[(0, 0)])
sage: r = Polyhedron(rays=[(1, 0)])
```

(continues on next page)

(continued from previous page)

```
sage: l = Polyhedron(vertices=[(-1, 0)], lines=[(1, -1)])
sage: pc1 = PolyhedralComplex([t1, t2])
sage: len(pc1.connected_components())
1
sage: pc2 = PolyhedralComplex([t1, v2])
sage: len(pc2.connected_components())
2
sage: pc3 = PolyhedralComplex([t1, t2, r])
sage: len(pc3.connected_components())
1
sage: pc4 = PolyhedralComplex([t1, t2, r, l])
sage: len(pc4.connected_components())
2
sage: pc5 = PolyhedralComplex([t1, t2, r, l, v3])
sage: len(pc5.connected_components())
3
sage: PolyhedralComplex([]).connected_components()
Traceback (most recent call last):
...
ValueError: the empty polyhedral complex has no connected components
```

```
>>> from sage.all import *
>>> t1 = Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0),␣
↪Integer(0)), (Integer(1), Integer(2))])
>>> t2 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
↪Integer(0)), (Integer(0), Integer(2))])
>>> v1 = Polyhedron(vertices=[(Integer(1), Integer(1))])
>>> v2 = Polyhedron(vertices=[(Integer(0), Integer(2))])
>>> v3 = Polyhedron(vertices=[(-Integer(1), Integer(0))])
>>> o =  Polyhedron(vertices=[(Integer(0), Integer(0))])
>>> r = Polyhedron(rays=[(Integer(1), Integer(0))])
>>> l = Polyhedron(vertices=[(-Integer(1), Integer(0))], lines=[(Integer(1), -
↪Integer(1))])
>>> pc1 = PolyhedralComplex([t1, t2])
>>> len(pc1.connected_components())
1
>>> pc2 = PolyhedralComplex([t1, v2])
>>> len(pc2.connected_components())
2
>>> pc3 = PolyhedralComplex([t1, t2, r])
>>> len(pc3.connected_components())
1
>>> pc4 = PolyhedralComplex([t1, t2, r, l])
>>> len(pc4.connected_components())
2
>>> pc5 = PolyhedralComplex([t1, t2, r, l, v3])
>>> len(pc5.connected_components())
3
>>> PolyhedralComplex([]).connected_components()
Traceback (most recent call last):
...
ValueError: the empty polyhedral complex has no connected components
```

dimension()

The dimension of this cell complex: the maximum dimension of its cells.

EXAMPLES:

```
sage: pc = PolyhedralComplex([
....:         Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)]),
....:         Polyhedron(vertices=[(1, 2), (0, 2)]) ])
sage: pc.dimension()
2
sage: empty_pc = PolyhedralComplex([])
sage: empty_pc.dimension()
-1
```

```
>>> from sage.all import *
>>> pc = PolyhedralComplex([
...         Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0),␣
↪Integer(0)), (Integer(1), Integer(2))]),
...         Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
↪Integer(2))]) ])
>>> pc.dimension()
2
>>> empty_pc = PolyhedralComplex([])
>>> empty_pc.dimension()
-1
```

disjoint_union(*right*)

The disjoint union of this polyhedral complex with another one.

INPUT:

- `right` – the other polyhedral complex (the right-hand factor)

EXAMPLES:

```
sage: p1 = Polyhedron(vertices=[(-1, 0), (0, 0), (0, 1)])
sage: p2 = Polyhedron(vertices=[(0, -1), (0, 0), (1, 0)])
sage: p3 = Polyhedron(vertices=[(0, -1), (1, -1), (1, 0)])
sage: pc = PolyhedralComplex([p1]).disjoint_union(PolyhedralComplex([p3]))
sage: set(pc.maximal_cell_iterator()) == set([p1, p3])
True
sage: pc.disjoint_union(PolyhedralComplex([p2]))
Traceback (most recent call last):
...
ValueError: the two complexes are not disjoint
```

```
>>> from sage.all import *
>>> p1 = Polyhedron(vertices=[(-Integer(1), Integer(0)), (Integer(0),␣
↪Integer(0)), (Integer(0), Integer(1))])
>>> p2 = Polyhedron(vertices=[(Integer(0), -Integer(1)), (Integer(0),␣
↪Integer(0)), (Integer(1), Integer(0))])
>>> p3 = Polyhedron(vertices=[(Integer(0), -Integer(1)), (Integer(1), -
↪Integer(1)), (Integer(1), Integer(0))])
>>> pc = PolyhedralComplex([p1]).disjoint_union(PolyhedralComplex([p3]))
>>> set(pc.maximal_cell_iterator()) == set([p1, p3])
```

(continues on next page)

(continued from previous page)

```
True
>>> pc.disjoint_union(PolyhedralComplex([p2]))
Traceback (most recent call last):
...
ValueError: the two complexes are not disjoint
```

face_poset()

The face poset of this polyhedral complex, the poset of nonempty cells, ordered by inclusion.

EXAMPLES:

```
sage: pc = PolyhedralComplex([
....:          Polyhedron(vertices=[(1/3, 1/3), (0, 0), (1, 2)]),
....:          Polyhedron(vertices=[(1, 2), (0, 0), (0, 1/2)])])
sage: poset = pc.face_poset()
sage: poset
Finite poset containing 11 elements
sage: d = {i: i.vertices_matrix() for i in poset}
sage: poset.plot(element_labels=d)                                          #␣
↪needs sage.plot
Graphics object consisting of 28 graphics primitives
```

```
>>> from sage.all import *
>>> pc = PolyhedralComplex([
...          Polyhedron(vertices=[(Integer(1)/Integer(3), Integer(1)/
↪Integer(3)), (Integer(0), Integer(0)), (Integer(1), Integer(2))]),
...          Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
↪Integer(0)), (Integer(0), Integer(1)/Integer(2))])])
>>> poset = pc.face_poset()
>>> poset
Finite poset containing 11 elements
>>> d = {i: i.vertices_matrix() for i in poset}
>>> poset.plot(element_labels=d)                                          #␣
↪needs sage.plot
Graphics object consisting of 28 graphics primitives
```

For a nonbounded polyhedral complex:

```
sage: pc = PolyhedralComplex([
....:          Polyhedron(vertices=[(1/3, 1/3), (0, 0), (1, 2)]),
....:          Polyhedron(vertices=[(1, 2), (0, 0), (0, 1/2)]),
....:          Polyhedron(vertices=[(-1/2, -1/2)], lines=[(1, -1)]),
....:          Polyhedron(rays=[(1, 0)])])
sage: poset = pc.face_poset()
sage: poset
Finite poset containing 13 elements
sage: d = {i:''.join([str(v)+'\n'
....:        for v in i.Vrepresentation()]) for i in poset}
sage: poset.show(element_labels=d, figsize=15)        # not tested
sage: pc = PolyhedralComplex([
....: Polyhedron(rays=[(1,0),(0,1)]),
....: Polyhedron(rays=[(-1,0),(0,1)]),
```

(continues on next page)

(continued from previous page)

```
....: Polyhedron(rays=[(-1,0),(0,-1)]),
....: Polyhedron(rays=[(1,0),(0,-1)])])
sage: pc.face_poset()
Finite poset containing 9 elements
```

```
>>> from sage.all import *
>>> pc = PolyhedralComplex([
...         Polyhedron(vertices=[(Integer(1)/Integer(3), Integer(1)/
↪Integer(3)), (Integer(0), Integer(0)), (Integer(1), Integer(2))]),
...         Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),
↪Integer(0)), (Integer(0), Integer(1)/Integer(2))]),
...         Polyhedron(vertices=[(-Integer(1)/Integer(2), -Integer(1)/
↪Integer(2))], lines=[(Integer(1), -Integer(1))]),
...         Polyhedron(rays=[(Integer(1), Integer(0))])])
>>> poset = pc.face_poset()
>>> poset
Finite poset containing 13 elements
>>> d = {i:''.join([str(v)+'\n'
...      for v in i.Vrepresentation()]) for i in poset}
>>> poset.show(element_labels=d, figsize=Integer(15))        # not tested
>>> pc = PolyhedralComplex([
... Polyhedron(rays=[(Integer(1),Integer(0)),(Integer(0),Integer(1))]),
... Polyhedron(rays=[(-Integer(1),Integer(0)),(Integer(0),Integer(1))]),
... Polyhedron(rays=[(-Integer(1),Integer(0)),(Integer(0),-Integer(1))]),
... Polyhedron(rays=[(Integer(1),Integer(0)),(Integer(0),-Integer(1))])])
>>> pc.face_poset()
Finite poset containing 9 elements
```

graph()

Return the 1-skeleton of this polyhedral complex, as a graph.

The vertices of the graph are of type `vector`. This raises a `NotImplementedError` if the polyhedral complex is unbounded.

> **Warning**
>
> This may give the wrong answer if the polyhedral complex was constructed with `maximality_check` set to `False`.

EXAMPLES:

```
sage: pc = PolyhedralComplex([
....:         Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)]),
....:         Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])])
sage: g = pc.graph(); g
Graph on 4 vertices
sage: g.vertices(sort=True)
[(0, 0), (0, 2), (1, 1), (1, 2)]
sage: g.edges(sort=True, labels=False)
[((0, 0), (0, 2)), ((0, 0), (1, 1)), ((0, 0), (1, 2)), ((0, 2), (1, 2)), ((1,
↪1), (1, 2))]
```

(continues on next page)

(continued from previous page)

```
sage: PolyhedralComplex([Polyhedron(rays=[(1,1)])]).graph()
Traceback (most recent call last):
...
NotImplementedError: the polyhedral complex is unbounded
```

```
>>> from sage.all import *
>>> pc = PolyhedralComplex([
...         Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0),␣
→Integer(0)), (Integer(1), Integer(2))]),
...         Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
→Integer(0)), (Integer(0), Integer(2))])])
>>> g = pc.graph(); g
Graph on 4 vertices
>>> g.vertices(sort=True)
[(0, 0), (0, 2), (1, 1), (1, 2)]
>>> g.edges(sort=True, labels=False)
[((0, 0), (0, 2)), ((0, 0), (1, 1)), ((0, 0), (1, 2)), ((0, 2), (1, 2)), ((1,␣
→1), (1, 2))]
>>> PolyhedralComplex([Polyhedron(rays=[(Integer(1),Integer(1))])]).graph()
Traceback (most recent call last):
...
NotImplementedError: the polyhedral complex is unbounded
```

Wrong answer due to `maximality_check=False`:

```
sage: p1 = Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)])
sage: p2 = Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])
sage: p3 = Polyhedron(vertices=[(1, 2), (0, 2)])
sage: PolyhedralComplex([p1, p2]).is_pure()
True
sage: PolyhedralComplex([p2, p3], maximality_check=True).is_pure()
True
sage: PolyhedralComplex([p2, p3], maximality_check=False).is_pure()
False
```

```
>>> from sage.all import *
>>> p1 = Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0),␣
→Integer(0)), (Integer(1), Integer(2))])
>>> p2 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
→Integer(0)), (Integer(0), Integer(2))])
>>> p3 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
→Integer(2))])
>>> PolyhedralComplex([p1, p2]).is_pure()
True
>>> PolyhedralComplex([p2, p3], maximality_check=True).is_pure()
True
>>> PolyhedralComplex([p2, p3], maximality_check=False).is_pure()
False
```

is_cell(*c*)

Return whether the given cell `c` is a cell of `self`.

EXAMPLES:

```
sage: p1 = Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)])
sage: p2 = Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])
sage: p3 = Polyhedron(vertices=[(1, 2), (0, 2)])
sage: pc = PolyhedralComplex([p1, p2])
sage: pc.is_cell(p3)
True
sage: pc.is_cell(Polyhedron(vertices=[(0, 0)]))
True
```

```
>>> from sage.all import *
>>> p1 = Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0),␣
↪Integer(0)), (Integer(1), Integer(2))])
>>> p2 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
↪Integer(0)), (Integer(0), Integer(2))])
>>> p3 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
↪Integer(2))])
>>> pc = PolyhedralComplex([p1, p2])
>>> pc.is_cell(p3)
True
>>> pc.is_cell(Polyhedron(vertices=[(Integer(0), Integer(0))]))
True
```

is_compact()

Test for boundedness of the polyhedral complex.

EXAMPLES:

```
sage: p1 = Polyhedron(vertices=[(1, 2), (0, 0), (0, 1/2)])
sage: p2 = Polyhedron(rays=[(1, 0)])
sage: PolyhedralComplex([p1]).is_compact()
True
sage: PolyhedralComplex([p1, p2]).is_compact()
False
```

```
>>> from sage.all import *
>>> p1 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
↪Integer(0)), (Integer(0), Integer(1)/Integer(2))])
>>> p2 = Polyhedron(rays=[(Integer(1), Integer(0))])
>>> PolyhedralComplex([p1]).is_compact()
True
>>> PolyhedralComplex([p1, p2]).is_compact()
False
```

is_connected()

Return whether `self` is connected.

EXAMPLES:

```
sage: pc1 = PolyhedralComplex([
....:         Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)]),
....:         Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])])
sage: pc1.is_connected()
True
```

(continues on next page)

(continued from previous page)

```
sage: pc2 = PolyhedralComplex([
....:         Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)]),
....:         Polyhedron(vertices=[(0, 2)])])
sage: pc2.is_connected()
False
sage: pc3 = PolyhedralComplex([
....:         Polyhedron(vertices=[(1/3, 1/3), (0, 0), (1, 2)]),
....:         Polyhedron(vertices=[(1, 2), (0, 0), (0, 1/2)]),
....:         Polyhedron(vertices=[(-1/2, -1/2)], lines=[(1, -1)]),
....:         Polyhedron(rays=[(1, 0)])])
sage: pc3.is_connected()
False
sage: pc4 = PolyhedralComplex([
....:         Polyhedron(vertices=[(1/3, 1/3), (0, 0), (1, 2)]),
....:         Polyhedron(rays=[(1, 0)])])
sage: pc4.is_connected()
True
```

```
>>> from sage.all import *
>>> pc1 = PolyhedralComplex([
...         Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0),␣
↪Integer(0)), (Integer(1), Integer(2))]),
...         Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
↪Integer(0)), (Integer(0), Integer(2))])])
>>> pc1.is_connected()
True
>>> pc2 = PolyhedralComplex([
...         Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0),␣
↪Integer(0)), (Integer(1), Integer(2))]),
...         Polyhedron(vertices=[(Integer(0), Integer(2))])])
>>> pc2.is_connected()
False
>>> pc3 = PolyhedralComplex([
...         Polyhedron(vertices=[(Integer(1)/Integer(3), Integer(1)/
↪Integer(3)), (Integer(0), Integer(0)), (Integer(1), Integer(2))]),
...         Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
↪Integer(0)), (Integer(0), Integer(1)/Integer(2))]),
...         Polyhedron(vertices=[(-Integer(1)/Integer(2), -Integer(1)/
↪Integer(2))], lines=[(Integer(1), -Integer(1))]),
...         Polyhedron(rays=[(Integer(1), Integer(0))])])
>>> pc3.is_connected()
False
>>> pc4 = PolyhedralComplex([
...         Polyhedron(vertices=[(Integer(1)/Integer(3), Integer(1)/
↪Integer(3)), (Integer(0), Integer(0)), (Integer(1), Integer(2))]),
...         Polyhedron(rays=[(Integer(1), Integer(0))])])
>>> pc4.is_connected()
True
```

is_convex()

Return whether the set of points in `self` is a convex set.

When `self` is convex, the union of its cells is a Polyhedron.

> **See also**
>
> *union_as_polyhedron()*

EXAMPLES:

```
sage: p1 = Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)])
sage: p2 = Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])
sage: p3 = Polyhedron(vertices=[(0, 0), (1, 1), (2, 0)])
sage: p4 = Polyhedron(vertices=[(2, 2)])
sage: PolyhedralComplex([p1, p2]).is_convex()
True
sage: PolyhedralComplex([p1, p3]).is_convex()
False
sage: PolyhedralComplex([p1, p4]).is_convex()
False
```

```
>>> from sage.all import *
>>> p1 = Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0),␣
↪Integer(0)), (Integer(1), Integer(2))])
>>> p2 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
↪Integer(0)), (Integer(0), Integer(2))])
>>> p3 = Polyhedron(vertices=[(Integer(0), Integer(0)), (Integer(1),␣
↪Integer(1)), (Integer(2), Integer(0))])
>>> p4 = Polyhedron(vertices=[(Integer(2), Integer(2))])
>>> PolyhedralComplex([p1, p2]).is_convex()
True
>>> PolyhedralComplex([p1, p3]).is_convex()
False
>>> PolyhedralComplex([p1, p4]).is_convex()
False
```

Test unbounded cases:

```
sage: pc1 = PolyhedralComplex([
....:         Polyhedron(vertices=[[1,0], [0,1]], rays=[[1,0], [0,1]])])
sage: pc1.is_convex()
True
sage: pc2 = PolyhedralComplex([
....:         Polyhedron(vertices=[[-1,0], [1,0]], lines=[[0,1]])])
sage: pc2.is_convex()
True
sage: pc3 = PolyhedralComplex([
....:         Polyhedron(vertices=[[1,0], [0,1]], rays=[[1,0], [0,1]]),
....:         Polyhedron(vertices=[[1,0], [0,-1]], rays=[[1,0], [0,-1]])])
sage: pc3.is_convex()
False
sage: pc4 = PolyhedralComplex([Polyhedron(rays=[[1,0], [-1,1]]),
....:                          Polyhedron(rays=[[1,0], [-1,-1]])])
sage: pc4.is_convex()
False
```

```
>>> from sage.all import *
>>> pc1 = PolyhedralComplex([
...         Polyhedron(vertices=[[Integer(1),Integer(0)], [Integer(0),
↪Integer(1)]], rays=[[Integer(1),Integer(0)], [Integer(0),Integer(1)]])])
>>> pc1.is_convex()
True
>>> pc2 = PolyhedralComplex([
...         Polyhedron(vertices=[[-Integer(1),Integer(0)], [Integer(1),
↪Integer(0)]], lines=[[Integer(0),Integer(1)]])])
>>> pc2.is_convex()
True
>>> pc3 = PolyhedralComplex([
...         Polyhedron(vertices=[[Integer(1),Integer(0)], [Integer(0),
↪Integer(1)]], rays=[[Integer(1),Integer(0)], [Integer(0),Integer(1)]]),
...         Polyhedron(vertices=[[Integer(1),Integer(0)], [Integer(0),-
↪Integer(1)]], rays=[[Integer(1),Integer(0)], [Integer(0),-Integer(1)]])])
>>> pc3.is_convex()
False
>>> pc4 = PolyhedralComplex([Polyhedron(rays=[[Integer(1),Integer(0)], [-
↪Integer(1),Integer(1)]]),
...                          Polyhedron(rays=[[Integer(1),Integer(0)], [-
↪Integer(1),-Integer(1)]])])
>>> pc4.is_convex()
False
```

The whole 3d space minus the first orthant is not convex:

```
sage: pc5 = PolyhedralComplex([
....:         Polyhedron(rays=[[1,0,0], [0,1,0], [0,0,-1]]),
....:         Polyhedron(rays=[[1,0,0], [0,-1,0], [0,0,-1]]),
....:         Polyhedron(rays=[[1,0,0], [0,-1,0], [0,0,1]]),
....:         Polyhedron(rays=[[-1,0,0], [0,-1,0], [0,0,-1]]),
....:         Polyhedron(rays=[[-1,0,0], [0,-1,0], [0,0,1]]),
....:         Polyhedron(rays=[[-1,0,0], [0,1,0], [0,0,-1]]),
....:         Polyhedron(rays=[[-1,0,0], [0,1,0], [0,0,1]])])
sage: pc5.is_convex()
False
```

```
>>> from sage.all import *
>>> pc5 = PolyhedralComplex([
...         Polyhedron(rays=[[Integer(1),Integer(0),Integer(0)], [Integer(0),
↪Integer(1),Integer(0)], [Integer(0),Integer(0),-Integer(1)]]),
...         Polyhedron(rays=[[Integer(1),Integer(0),Integer(0)], [Integer(0),-
↪Integer(1),Integer(0)], [Integer(0),Integer(0),-Integer(1)]]),
...         Polyhedron(rays=[[Integer(1),Integer(0),Integer(0)], [Integer(0),-
↪Integer(1),Integer(0)], [Integer(0),Integer(0),Integer(1)]]),
...         Polyhedron(rays=[[-Integer(1),Integer(0),Integer(0)], [Integer(0),
↪-Integer(1),Integer(0)], [Integer(0),Integer(0),-Integer(1)]]),
...         Polyhedron(rays=[[-Integer(1),Integer(0),Integer(0)], [Integer(0),
↪-Integer(1),Integer(0)], [Integer(0),Integer(0),Integer(1)]]),
...         Polyhedron(rays=[[-Integer(1),Integer(0),Integer(0)], [Integer(0),
↪Integer(1),Integer(0)], [Integer(0),Integer(0),-Integer(1)]]),
...         Polyhedron(rays=[[-Integer(1),Integer(0),Integer(0)], [Integer(0),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1),Integer(0)], [Integer(0),Integer(0),Integer(1)]])])
>>> pc5.is_convex()
False
```

Test some non-full-dimensional examples:

```
sage: l = PolyhedralComplex([Polyhedron(vertices=[(1, 2), (0, 2)])])
sage: l.is_convex()
True
sage: pc1b = PolyhedralComplex([Polyhedron(
....:         vertices=[[1,0,0], [0,1,0]], rays=[[1,0,0],[0,1,0]])])
sage: pc1b.is_convex()
True
sage: pc4b = PolyhedralComplex([
....:         Polyhedron(rays=[[1,0,0], [-1,1,0]]),
....:         Polyhedron(rays=[[1,0,0], [-1,-1,0]])])
sage: pc4b.is_convex()
False
```

```
>>> from sage.all import *
>>> l = PolyhedralComplex([Polyhedron(vertices=[(Integer(1), Integer(2)),␣
↪(Integer(0), Integer(2))])])
>>> l.is_convex()
True
>>> pc1b = PolyhedralComplex([Polyhedron(
...         vertices=[[Integer(1),Integer(0),Integer(0)], [Integer(0),
↪Integer(1),Integer(0)]], rays=[[Integer(1),Integer(0),Integer(0)],
↪[Integer(0),Integer(1),Integer(0)]])])
>>> pc1b.is_convex()
True
>>> pc4b = PolyhedralComplex([
...         Polyhedron(rays=[[Integer(1),Integer(0),Integer(0)], [-Integer(1),
↪Integer(1),Integer(0)]]),
...         Polyhedron(rays=[[Integer(1),Integer(0),Integer(0)], [-Integer(1),
↪-Integer(1),Integer(0)]])])
>>> pc4b.is_convex()
False
```

is_full_dimensional()

Return whether this polyhedral complex is full-dimensional.

This means that its dimension is equal to its ambient dimension.

EXAMPLES:

```
sage: p1 = Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)])
sage: p2 = Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])
sage: p3 = Polyhedron(vertices=[(1, 2), (0, 2)])
sage: pc = PolyhedralComplex([p1, p2, p3])
sage: pc.is_full_dimensional()
True
sage: PolyhedralComplex([p3]).is_full_dimensional()
False
```

```
>>> from sage.all import *
>>> p1 = Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0),␣
↪Integer(0)), (Integer(1), Integer(2))])
>>> p2 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
↪Integer(0)), (Integer(0), Integer(2))])
>>> p3 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
↪Integer(2))])
>>> pc = PolyhedralComplex([p1, p2, p3])
>>> pc.is_full_dimensional()
True
>>> PolyhedralComplex([p3]).is_full_dimensional()
False
```

is_immutable()

Return whether self is immutable.

EXAMPLES:

```
sage: pc1 = PolyhedralComplex([Polyhedron(vertices=[[0], [1]])])
sage: pc1.is_immutable()
False
sage: pc2 = PolyhedralComplex([Polyhedron(vertices=[[0], [1]])],
....:                          is_mutable=False)
sage: pc2.is_immutable()
True
sage: pc3 = PolyhedralComplex([Polyhedron(vertices=[[0], [1]])],
....:                          is_immutable=True)
sage: pc3.is_immutable()
True
```

```
>>> from sage.all import *
>>> pc1 = PolyhedralComplex([Polyhedron(vertices=[[Integer(0)],␣
↪[Integer(1)]])])
>>> pc1.is_immutable()
False
>>> pc2 = PolyhedralComplex([Polyhedron(vertices=[[Integer(0)],␣
↪[Integer(1)]])],
...                          is_mutable=False)
>>> pc2.is_immutable()
True
>>> pc3 = PolyhedralComplex([Polyhedron(vertices=[[Integer(0)],␣
↪[Integer(1)]])],
...                          is_immutable=True)
>>> pc3.is_immutable()
True
```

is_maximal_cell(*c*)

Return whether the given cell c is a maximal cell of self.

> **Warning**
>
> This may give the wrong answer if the polyhedral complex was constructed with maximality_check

set to False.

EXAMPLES:

```
sage: p1 = Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)])
sage: p2 = Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])
sage: p3 = Polyhedron(vertices=[(1, 2), (0, 2)])
sage: pc = PolyhedralComplex([p1, p2, p3])
sage: pc.is_maximal_cell(p1)
True
sage: pc.is_maximal_cell(p3)
False
```

```
>>> from sage.all import *
>>> p1 = Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0),␣
↪Integer(0)), (Integer(1), Integer(2))])
>>> p2 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
↪Integer(0)), (Integer(0), Integer(2))])
>>> p3 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
↪Integer(2))])
>>> pc = PolyhedralComplex([p1, p2, p3])
>>> pc.is_maximal_cell(p1)
True
>>> pc.is_maximal_cell(p3)
False
```

Wrong answer due to maximality_check=False:

```
sage: pc_invalid = PolyhedralComplex([p1, p2, p3],
....:                    maximality_check=False)
sage: pc_invalid.is_maximal_cell(p3)
True
```

```
>>> from sage.all import *
>>> pc_invalid = PolyhedralComplex([p1, p2, p3],
...                    maximality_check=False)
>>> pc_invalid.is_maximal_cell(p3)
True
```

is_mutable()

Return whether self is mutable.

EXAMPLES:

```
sage: pc1 = PolyhedralComplex([Polyhedron(vertices=[[0], [1]])])
sage: pc1.is_mutable()
True
sage: pc2 = PolyhedralComplex([Polyhedron(vertices=[[0], [1]])],
....:                          is_mutable=False)
sage: pc2.is_mutable()
False
sage: pc1 == pc2
True
```

(continues on next page)

(continued from previous page)

```
sage: pc3 = PolyhedralComplex([Polyhedron(vertices=[[0], [1]])],
....:                         is_immutable=True)
sage: pc3.is_mutable()
False
sage: pc2 == pc3
True
```

```
>>> from sage.all import *
>>> pc1 = PolyhedralComplex([Polyhedron(vertices=[[Integer(0)],
↪[Integer(1)]])])
>>> pc1.is_mutable()
True
>>> pc2 = PolyhedralComplex([Polyhedron(vertices=[[Integer(0)],
↪[Integer(1)]])],
...                         is_mutable=False)
>>> pc2.is_mutable()
False
>>> pc1 == pc2
True
>>> pc3 = PolyhedralComplex([Polyhedron(vertices=[[Integer(0)],
↪[Integer(1)]])],
...                         is_immutable=True)
>>> pc3.is_mutable()
False
>>> pc2 == pc3
True
```

is_polyhedral_fan()

Test if this polyhedral complex is a polyhedral fan.

A polyhedral complex is a **fan** if all of its (maximal) cells are cones.

EXAMPLES:

```
sage: p1 = Polyhedron(vertices=[(0, 0), (1, 1), (1, 2)])
sage: p2 = Polyhedron(rays=[(1, 0)])
sage: PolyhedralComplex([p1]).is_polyhedral_fan()
False
sage: PolyhedralComplex([p2]).is_polyhedral_fan()
True
sage: halfplane = Polyhedron(rays=[(1, 0), (-1, 0), (0, 1)])
sage: PolyhedralComplex([halfplane]).is_polyhedral_fan()
True
```

```
>>> from sage.all import *
>>> p1 = Polyhedron(vertices=[(Integer(0), Integer(0)), (Integer(1),
↪Integer(1)), (Integer(1), Integer(2))])
>>> p2 = Polyhedron(rays=[(Integer(1), Integer(0))])
>>> PolyhedralComplex([p1]).is_polyhedral_fan()
False
>>> PolyhedralComplex([p2]).is_polyhedral_fan()
True
```

(continues on next page)

(continued from previous page)

```
>>> halfplane = Polyhedron(rays=[(Integer(1), Integer(0)), (-Integer(1),
↪Integer(0)), (Integer(0), Integer(1))])
>>> PolyhedralComplex([halfplane]).is_polyhedral_fan()
True
```

is_pure()

Test if this polyhedral complex is pure.

A polyhedral complex is pure if and only if all of its maximal cells have the same dimension.

> **Warning**
>
> This may give the wrong answer if the polyhedral complex was constructed with `maximality_check` set to `False`.

EXAMPLES:

```
sage: p1 = Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)])
sage: p2 = Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])
sage: p3 = Polyhedron(vertices=[(1, 2), (0, 2)])
sage: pc = PolyhedralComplex([p1, p2, p3])
sage: pc.is_pure()
True
```

```
>>> from sage.all import *
>>> p1 = Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0),
↪Integer(0)), (Integer(1), Integer(2))])
>>> p2 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),
↪Integer(0)), (Integer(0), Integer(2))])
>>> p3 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),
↪Integer(2))])
>>> pc = PolyhedralComplex([p1, p2, p3])
>>> pc.is_pure()
True
```

Wrong answer due to `maximality_check=False`:

```
sage: pc_invalid = PolyhedralComplex([p1, p2, p3],
....:                 maximality_check=False)
sage: pc_invalid.is_pure()
False
```

```
>>> from sage.all import *
>>> pc_invalid = PolyhedralComplex([p1, p2, p3],
...                 maximality_check=False)
>>> pc_invalid.is_pure()
False
```

is_simplicial_complex()

Test if this polyhedral complex is a simplicial complex.

A polyhedral complex is **simplicial** if all of its (maximal) cells are simplices, i.e., every cell is a bounded polytope with $d + 1$ vertices, where d is the dimension of the polytope.

EXAMPLES:

```
sage: p1 = Polyhedron(vertices=[(0, 0), (1, 1), (1, 2)])
sage: p2 = Polyhedron(rays=[(1, 0)])
sage: PolyhedralComplex([p1]).is_simplicial_complex()
True
sage: PolyhedralComplex([p2]).is_simplicial_complex()
False
```

```
>>> from sage.all import *
>>> p1 = Polyhedron(vertices=[(Integer(0), Integer(0)), (Integer(1),␣
→Integer(1)), (Integer(1), Integer(2))])
>>> p2 = Polyhedron(rays=[(Integer(1), Integer(0))])
>>> PolyhedralComplex([p1]).is_simplicial_complex()
True
>>> PolyhedralComplex([p2]).is_simplicial_complex()
False
```

is_simplicial_fan()

Test if this polyhedral complex is a simplicial fan.

A polyhedral complex is a **simplicial fan** if all of its (maximal) cells are simplicial cones, i.e., every cell is a pointed cone (with vertex being the origin) generated by d linearly independent rays, where d is the dimension of the cone.

EXAMPLES:

```
sage: p1 = Polyhedron(vertices=[(0, 0), (1, 1), (1, 2)])
sage: p2 = Polyhedron(rays=[(1, 0)])
sage: PolyhedralComplex([p1]).is_simplicial_fan()
False
sage: PolyhedralComplex([p2]).is_simplicial_fan()
True
sage: halfplane = Polyhedron(rays=[(1, 0), (-1, 0), (0, 1)])
sage: PolyhedralComplex([halfplane]).is_simplicial_fan()
False
```

```
>>> from sage.all import *
>>> p1 = Polyhedron(vertices=[(Integer(0), Integer(0)), (Integer(1),␣
→Integer(1)), (Integer(1), Integer(2))])
>>> p2 = Polyhedron(rays=[(Integer(1), Integer(0))])
>>> PolyhedralComplex([p1]).is_simplicial_fan()
False
>>> PolyhedralComplex([p2]).is_simplicial_fan()
True
>>> halfplane = Polyhedron(rays=[(Integer(1), Integer(0)), (-Integer(1),␣
→Integer(0)), (Integer(0), Integer(1))])
>>> PolyhedralComplex([halfplane]).is_simplicial_fan()
False
```

is_subcomplex(*other*)

Return whether `self` is a subcomplex of `other`.

INPUT:

- `other` – a polyhedral complex

Each maximal cell of `self` must be a cell of `other` for this to be `True`.

EXAMPLES:

```
sage: p1 = Polyhedron(vertices=[(1/3, 1/3), (0, 0), (1, 2)])
sage: p2 = Polyhedron(vertices=[(1, 2), (0, 0), (0, 1/2)])
sage: p3 = Polyhedron(vertices=[(0, 0), (1, 0)])
sage: pc = PolyhedralComplex([p1, Polyhedron(vertices=[(1, 0)])])
sage: pc.is_subcomplex(PolyhedralComplex([p1, p2, p3]))
True
sage: pc.is_subcomplex(PolyhedralComplex([p1, p2]))
False
```

```
>>> from sage.all import *
>>> p1 = Polyhedron(vertices=[(Integer(1)/Integer(3), Integer(1)/Integer(3)),␣
↪(Integer(0), Integer(0)), (Integer(1), Integer(2))])
>>> p2 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
↪Integer(0)), (Integer(0), Integer(1)/Integer(2))])
>>> p3 = Polyhedron(vertices=[(Integer(0), Integer(0)), (Integer(1),␣
↪Integer(0))])
>>> pc = PolyhedralComplex([p1, Polyhedron(vertices=[(Integer(1),␣
↪Integer(0))])])
>>> pc.is_subcomplex(PolyhedralComplex([p1, p2, p3]))
True
>>> pc.is_subcomplex(PolyhedralComplex([p1, p2]))
False
```

join(*right*)

The join of this polyhedral complex with another one.

INPUT:

- `right` – the other polyhedral complex (the right-hand factor)

EXAMPLES:

```
sage: pc = PolyhedralComplex([Polyhedron(vertices=[[0], [1]])])
sage: pc_join = pc.join(pc)
sage: pc_join
Polyhedral complex with 1 maximal cell
sage: next(pc_join.maximal_cell_iterator()).vertices()
(A vertex at (0, 0, 0),
 A vertex at (0, 0, 1),
 A vertex at (0, 1, 1),
 A vertex at (1, 0, 0))
```

```
>>> from sage.all import *
>>> pc = PolyhedralComplex([Polyhedron(vertices=[[Integer(0)],␣
↪[Integer(1)]])])
>>> pc_join = pc.join(pc)
>>> pc_join
Polyhedral complex with 1 maximal cell
```

(continues on next page)

(continued from previous page)

```
>>> next(pc_join.maximal_cell_iterator()).vertices()
(A vertex at (0, 0, 0),
 A vertex at (0, 0, 1),
 A vertex at (0, 1, 1),
 A vertex at (1, 0, 0))
```

maximal_cell_iterator(*increasing=False*)

An iterator for the maximal cells in this polyhedral complex.

INPUT:

- `increasing` – boolean (default: `False`); if `True`, return maximal cells in increasing order of dimension. Otherwise it returns cells in decreasing order of dimension.

Note

Among the cells of a fixed dimension, there is no sorting.

Warning

This may give the wrong answer if the polyhedral complex was constructed with `maximality_check` set to `False`.

EXAMPLES:

```
sage: p1 = Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)])
sage: p2 = Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])
sage: p3 = Polyhedron(vertices=[(1, 2), (0, 2)])
sage: pc = PolyhedralComplex([p1, p2, p3])
sage: len(list(pc.maximal_cell_iterator()))
2
```

```
>>> from sage.all import *
>>> p1 = Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0),␣
↪Integer(0)), (Integer(1), Integer(2))])
>>> p2 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
↪Integer(0)), (Integer(0), Integer(2))])
>>> p3 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
↪Integer(2))])
>>> pc = PolyhedralComplex([p1, p2, p3])
>>> len(list(pc.maximal_cell_iterator()))
2
```

Wrong answer due to `maximality_check=False`:

```
sage: pc_invalid = PolyhedralComplex([p1, p2, p3],
....:                                maximality_check=False)
sage: len(list(pc_invalid.maximal_cell_iterator()))
3
```

```
>>> from sage.all import *
>>> pc_invalid = PolyhedralComplex([p1, p2, p3],
...                  maximality_check=False)
>>> len(list(pc_invalid.maximal_cell_iterator()))
3
```

maximal_cells()

The maximal cells of this polyhedral complex, in the form of a dictionary: the keys are integers, representing dimension, and the value associated to an integer d is the set of d-maximal cells.

> **Warning**
>
> This may give the wrong answer if the polyhedral complex was constructed with `maximality_check` set to `False`.

EXAMPLES:

```
sage: p1 = Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)])
sage: p2 = Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])
sage: p3 = Polyhedron(vertices=[(1, 2), (0, 2)])
sage: pc = PolyhedralComplex([p1, p2, p3])
sage: len(pc.maximal_cells()[2])
2
sage: 1 in pc.maximal_cells()
False
```

```
>>> from sage.all import *
>>> p1 = Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0),␣
↪Integer(0)), (Integer(1), Integer(2))])
>>> p2 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
↪Integer(0)), (Integer(0), Integer(2))])
>>> p3 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
↪Integer(2))])
>>> pc = PolyhedralComplex([p1, p2, p3])
>>> len(pc.maximal_cells()[Integer(2)])
2
>>> Integer(1) in pc.maximal_cells()
False
```

Wrong answer due to `maximality_check=False`:

```
sage: pc_invalid = PolyhedralComplex([p1, p2, p3],
....:                  maximality_check=False)
sage: len(pc_invalid.maximal_cells()[1])
1
```

```
>>> from sage.all import *
>>> pc_invalid = PolyhedralComplex([p1, p2, p3],
...                  maximality_check=False)
>>> len(pc_invalid.maximal_cells()[Integer(1)])
1
```

maximal_cells_sorted()

Return the sorted list of the maximal cells of this polyhedral complex by non-increasing dimensions.

EXAMPLES:

```
sage: pc = PolyhedralComplex([
....:         Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)]),
....:         Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])])
sage: [p.vertices_list() for p in pc.maximal_cells_sorted()]
[[[0, 0], [0, 2], [1, 2]], [[0, 0], [1, 1], [1, 2]]]
```

```
>>> from sage.all import *
>>> pc = PolyhedralComplex([
...         Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0),␣
→Integer(0)), (Integer(1), Integer(2))]),
...         Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
→Integer(0)), (Integer(0), Integer(2))])])
>>> [p.vertices_list() for p in pc.maximal_cells_sorted()]
[[[0, 0], [0, 2], [1, 2]], [[0, 0], [1, 1], [1, 2]]]
```

n_maximal_cells(*n*)

List of maximal cells of dimension `n` of this polyhedral complex.

INPUT:

- `n` – nonnegative integer; the dimension

Note

The resulting list need not be sorted. If you want a sorted list of n-cells, use `_n_maximal_cells_sorted()`.

Warning

This may give the wrong answer if the polyhedral complex was constructed with `maximality_check` set to `False`.

EXAMPLES:

```
sage: p1 = Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)])
sage: p2 = Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])
sage: p3 = Polyhedron(vertices=[(1, 2), (0, 2)])
sage: pc = PolyhedralComplex([p1, p2, p3])
sage: len(pc.n_maximal_cells(2))
2
sage: len(pc.n_maximal_cells(1))
0
```

```
>>> from sage.all import *
>>> p1 = Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0),␣
→Integer(0)), (Integer(1), Integer(2))])
>>> p2 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(0)), (Integer(0), Integer(2))])
>>> p3 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
↪Integer(2))])
>>> pc = PolyhedralComplex([p1, p2, p3])
>>> len(pc.n_maximal_cells(Integer(2)))
2
>>> len(pc.n_maximal_cells(Integer(1)))
0
```

Wrong answer due to `maximality_check=False`:

```
sage: pc_invalid = PolyhedralComplex([p1, p2, p3],
....:                    maximality_check=False)
sage: len(pc_invalid.n_maximal_cells(1))
1
```

```
>>> from sage.all import *
>>> pc_invalid = PolyhedralComplex([p1, p2, p3],
...                    maximality_check=False)
>>> len(pc_invalid.n_maximal_cells(Integer(1)))
1
```

n_skeleton(*n*)

The n-skeleton of this polyhedral complex.

The n-skeleton of a polyhedral complex is obtained by discarding all of the cells in dimensions larger than n.

INPUT:

- `n` – nonnegative integer; the dimension

See also

stratify()

EXAMPLES:

```
sage: pc = PolyhedralComplex([
....:         Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)]),
....:         Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])])
sage: pc.n_skeleton(2)
Polyhedral complex with 2 maximal cells
sage: pc.n_skeleton(1)
Polyhedral complex with 5 maximal cells
sage: pc.n_skeleton(0)
Polyhedral complex with 4 maximal cells
```

```
>>> from sage.all import *
>>> pc = PolyhedralComplex([
...         Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0),␣
↪Integer(0)), (Integer(1), Integer(2))]),
...         Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(0)), (Integer(0), Integer(2))])])
>>> pc.n_skeleton(Integer(2))
Polyhedral complex with 2 maximal cells
>>> pc.n_skeleton(Integer(1))
Polyhedral complex with 5 maximal cells
>>> pc.n_skeleton(Integer(0))
Polyhedral complex with 4 maximal cells
```

plot(***kwds*)

Return a plot of the polyhedral complex, if it is of dim at most 3.

INPUT:

- `explosion_factor` – (default: 0) if positive, separate the cells of the complex by extra space. In this case, the following keyword arguments can be passed to `exploded_plot()`:
  - `center` – (default: `None`, denoting the origin) the center of explosion
  - `sticky_vertices` – (default: `False`) boolean or dict; whether to draw line segments between shared vertices of the given polyhedra. A dict gives options for `sage.plot.line()`.
  - `sticky_center` – (default: `True`) boolean or dict. When `center` is a vertex of some of the polyhedra, whether to draw line segments connecting the `center` to the shifted copies of these vertices. A dict gives options for `sage.plot.line()`.
- `color` – (default: `None`) if `'rainbow'`, assign a different color to every maximal cell; otherwise, passed on to `plot()`.
- other keyword arguments are passed on to `plot()`.

EXAMPLES:

```
sage: p1 = Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)])
sage: p2 = Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])
sage: p3 = Polyhedron(vertices=[(0, 0), (0, 2), (-1, 1)])
sage: pc1 = PolyhedralComplex([p1, p2, p3, -p1, -p2, -p3])
sage: bb = dict(xmin=-2, xmax=2, ymin=-3, ymax=3, axes=False)
sage: g0 = pc1.plot(color='rainbow', **bb)                                    #␣
↪needs sage.plot
sage: g1 = pc1.plot(explosion_factor=0.5, **bb)                               #␣
↪needs sage.plot
sage: g2 = pc1.plot(explosion_factor=1, color='rainbow', alpha=0.5, **bb)    #␣
↪needs sage.plot
sage: graphics_array([g0, g1, g2]).show(axes=False)                          #␣
↪not tested

sage: pc2 = PolyhedralComplex([polytopes.hypercube(3)])
sage: pc3 = pc2.subdivide(new_vertices=[(0, 0, 0)])
sage: g3 = pc3.plot(explosion_factor=1, color='rainbow',                      #␣
↪needs sage.plot
....:               alpha=0.5, axes=False, online=True)
sage: pc4 = pc2.subdivide(make_simplicial=True)
sage: g4 = pc4.plot(explosion_factor=1, center=(1, -1, 1), fill='blue',       #␣
↪needs sage.plot
....:             wireframe='white', point={'color':'red', 'size':10},
....:             alpha=0.6, online=True)
```

(continues on next page)

(continued from previous page)

```
sage: pc5 = PolyhedralComplex([
....:         Polyhedron(rays=[[1,0,0], [0,1,0], [0,0,-1]]),
....:         Polyhedron(rays=[[1,0,0], [0,-1,0], [0,0,-1]]),
....:         Polyhedron(rays=[[1,0,0], [0,-1,0], [0,0,1]]),
....:         Polyhedron(rays=[[-1,0,0], [0,-1,0], [0,0,-1]]),
....:         Polyhedron(rays=[[-1,0,0], [0,-1,0], [0,0,1]]),
....:         Polyhedron(rays=[[-1,0,0], [0,1,0], [0,0,-1]]),
....:         Polyhedron(rays=[[-1,0,0], [0,1,0], [0,0,1]])])
sage: g5 = pc5.plot(explosion_factor=0.3, color='rainbow', alpha=0.8,       #␣
↪needs sage.plot
....:               point={'size': 20}, axes=False, online=True)
```

```
>>> from sage.all import *
>>> p1 = Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0),␣
↪Integer(0)), (Integer(1), Integer(2))])
>>> p2 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
↪Integer(0)), (Integer(0), Integer(2))])
>>> p3 = Polyhedron(vertices=[(Integer(0), Integer(0)), (Integer(0),␣
↪Integer(2)), (-Integer(1), Integer(1))])
>>> pc1 = PolyhedralComplex([p1, p2, p3, -p1, -p2, -p3])
>>> bb = dict(xmin=-Integer(2), xmax=Integer(2), ymin=-Integer(3),␣
↪ymax=Integer(3), axes=False)
>>> g0 = pc1.plot(color='rainbow', **bb)                                   #␣
↪needs sage.plot
>>> g1 = pc1.plot(explosion_factor=RealNumber('0.5'), **bb)                  ␣
↪        # needs sage.plot
>>> g2 = pc1.plot(explosion_factor=Integer(1), color='rainbow',␣
↪alpha=RealNumber('0.5'), **bb)    # needs sage.plot
>>> graphics_array([g0, g1, g2]).show(axes=False)                          #␣
↪not tested

>>> pc2 = PolyhedralComplex([polytopes.hypercube(Integer(3))])
>>> pc3 = pc2.subdivide(new_vertices=[(Integer(0), Integer(0), Integer(0))])
>>> g3 = pc3.plot(explosion_factor=Integer(1), color='rainbow',               ␣
↪     # needs sage.plot
...               alpha=RealNumber('0.5'), axes=False, online=True)
>>> pc4 = pc2.subdivide(make_simplicial=True)
>>> g4 = pc4.plot(explosion_factor=Integer(1), center=(Integer(1), -
↪Integer(1), Integer(1)), fill='blue',      # needs sage.plot
...               wireframe='white', point={'color':'red', 'size':Integer(10)},
...               alpha=RealNumber('0.6'), online=True)
>>> pc5 = PolyhedralComplex([
...         Polyhedron(rays=[[Integer(1),Integer(0),Integer(0)], [Integer(0),
↪Integer(1),Integer(0)], [Integer(0),Integer(0),-Integer(1)]]),
...         Polyhedron(rays=[[Integer(1),Integer(0),Integer(0)], [Integer(0),-
↪Integer(1),Integer(0)], [Integer(0),Integer(0),-Integer(1)]]),
...         Polyhedron(rays=[[Integer(1),Integer(0),Integer(0)], [Integer(0),-
↪Integer(1),Integer(0)], [Integer(0),Integer(0),Integer(1)]]),
...         Polyhedron(rays=[[-Integer(1),Integer(0),Integer(0)], [Integer(0),
↪-Integer(1),Integer(0)], [Integer(0),Integer(0),-Integer(1)]]),
...         Polyhedron(rays=[[-Integer(1),Integer(0),Integer(0)], [Integer(0),
↪-Integer(1),Integer(0)], [Integer(0),Integer(0),Integer(1)]]),
```

(continues on next page)

(continued from previous page)

```
...         Polyhedron(rays=[[-Integer(1),Integer(0),Integer(0)], [Integer(0),
↪Integer(1),Integer(0)], [Integer(0),Integer(0),-Integer(1)]]),
...         Polyhedron(rays=[[-Integer(1),Integer(0),Integer(0)], [Integer(0),
↪Integer(1),Integer(0)], [Integer(0),Integer(0),Integer(1)]])])
>>> g5 = pc5.plot(explosion_factor=RealNumber('0.3'), color='rainbow',␣
↪alpha=RealNumber('0.8'),        # needs sage.plot
...                point={'size': Integer(20)}, axes=False, online=True)
```

product (*right*)

The (Cartesian) product of this polyhedral complex with another one.

INPUT:

- `right` – the other polyhedral complex (the right-hand factor)

OUTPUT: the product `self x right`

EXAMPLES:

```
sage: pc = PolyhedralComplex([Polyhedron(vertices=[[0], [1]])])
sage: pc_square = pc.product(pc)
sage: pc_square
Polyhedral complex with 1 maximal cell
sage: next(pc_square.maximal_cell_iterator()).vertices()
(A vertex at (0, 0),
 A vertex at (0, 1),
 A vertex at (1, 0),
 A vertex at (1, 1))
```

```
>>> from sage.all import *
>>> pc = PolyhedralComplex([Polyhedron(vertices=[[Integer(0)],␣
↪[Integer(1)]])])
>>> pc_square = pc.product(pc)
>>> pc_square
Polyhedral complex with 1 maximal cell
>>> next(pc_square.maximal_cell_iterator()).vertices()
(A vertex at (0, 0),
 A vertex at (0, 1),
 A vertex at (1, 0),
 A vertex at (1, 1))
```

relative_boundary_cells ()

Return the maximal cells of the relative-boundary sub-complex.

A point P is in the relative boundary of a set S if P is in the closure of S but not in the relative interior of S.

> **Warning**
>
> This may give the wrong answer if the polyhedral complex was constructed with `maximality_check` set to `False`.

EXAMPLES:

```
sage: p1 = Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)])
sage: p2 = Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])
sage: p3 = Polyhedron(vertices=[(1, 2), (0, 2)])
sage: p4 = Polyhedron(vertices=[(2, 2)])
sage: pc = PolyhedralComplex([p1, p2])
sage: rbd_cells = pc.relative_boundary_cells()
sage: len(rbd_cells)
4
sage: all(p.dimension() == 1 for p in rbd_cells)
True
sage: pc_lower_dim = PolyhedralComplex([p3])
sage: sorted([p.vertices() for p in pc_lower_dim.relative_boundary_cells()])
[(A vertex at (0, 2),), (A vertex at (1, 2),)]
```

```
>>> from sage.all import *
>>> p1 = Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0),␣
↪Integer(0)), (Integer(1), Integer(2))])
>>> p2 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
↪Integer(0)), (Integer(0), Integer(2))])
>>> p3 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
↪Integer(2))])
>>> p4 = Polyhedron(vertices=[(Integer(2), Integer(2))])
>>> pc = PolyhedralComplex([p1, p2])
>>> rbd_cells = pc.relative_boundary_cells()
>>> len(rbd_cells)
4
>>> all(p.dimension() == Integer(1) for p in rbd_cells)
True
>>> pc_lower_dim = PolyhedralComplex([p3])
>>> sorted([p.vertices() for p in pc_lower_dim.relative_boundary_cells()])
[(A vertex at (0, 2),), (A vertex at (1, 2),)]
```

Test on polyhedral complex which is not pure:

```
sage: pc_non_pure = PolyhedralComplex([p1, p3, p4])
sage: (set(pc_non_pure.relative_boundary_cells())
....:  == set([f.as_polyhedron() for f in p1.faces(1)] + [p3, p4]))
True
```

```
>>> from sage.all import *
>>> pc_non_pure = PolyhedralComplex([p1, p3, p4])
>>> (set(pc_non_pure.relative_boundary_cells())
...  == set([f.as_polyhedron() for f in p1.faces(Integer(1))] + [p3, p4]))
True
```

Test with `maximality_check == False`:

```
sage: pc_invalid = PolyhedralComplex([p2, p3],
....:                                  maximality_check=False)
sage: (set(pc_invalid.relative_boundary_cells())
....:  == set([f.as_polyhedron() for f in p2.faces(1)]))
True
```

```
>>> from sage.all import *
>>> pc_invalid = PolyhedralComplex([p2, p3],
...                                maximality_check=False)
>>> (set(pc_invalid.relative_boundary_cells())
...  == set([f.as_polyhedron() for f in p2.faces(Integer(1))]))
True
```

Test unbounded case:

```
sage: pc3 = PolyhedralComplex([
....:         Polyhedron(vertices=[[1,0], [0,1]], rays=[[1,0], [0,1]]),
....:         Polyhedron(vertices=[[1,0], [0,-1]], rays=[[1,0], [0,-1]])])
sage: len(pc3.relative_boundary_cells())
4
```

```
>>> from sage.all import *
>>> pc3 = PolyhedralComplex([
...         Polyhedron(vertices=[[Integer(1),Integer(0)], [Integer(0),
↪Integer(1)]], rays=[[Integer(1),Integer(0)], [Integer(0),Integer(1)]]),
...         Polyhedron(vertices=[[Integer(1),Integer(0)], [Integer(0),-
↪Integer(1)]], rays=[[Integer(1),Integer(0)], [Integer(0),-Integer(1)]])])
>>> len(pc3.relative_boundary_cells())
4
```

remove_cell(*cell*, *check=False*)

Remove `cell` from `self` and all the cells that contain `cell` as a subface.

INPUT:

- `cell` – a cell of the polyhedral complex
- `check` – boolean (default: `False`); if `True`, raise an error if `cell` is not a cell of this complex

This does not return anything; instead, it **changes** the polyhedral complex.

EXAMPLES:

If you add a cell which is already present, there is no effect:

```
sage: p1 = Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)])
sage: p2 = Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])
sage: r = Polyhedron(rays=[(1, 0)])
sage: pc = PolyhedralComplex([p1, p2, r])
sage: pc.dimension()
2
sage: pc.remove_cell(Polyhedron(vertices=[(0, 0), (1, 2)]))
sage: pc.dimension()
1
sage: pc
Polyhedral complex with 5 maximal cells
sage: pc.remove_cell(Polyhedron(vertices=[(1, 2)]))
sage: pc.dimension()
1
sage: pc
Polyhedral complex with 3 maximal cells
```

(continues on next page)

(continued from previous page)

```
sage: pc.remove_cell(Polyhedron(vertices=[(0, 0)]))
sage: pc.dimension()
0
```

```
>>> from sage.all import *
>>> p1 = Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0),␣
↪Integer(0)), (Integer(1), Integer(2))])
>>> p2 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
↪Integer(0)), (Integer(0), Integer(2))])
>>> r = Polyhedron(rays=[(Integer(1), Integer(0))])
>>> pc = PolyhedralComplex([p1, p2, r])
>>> pc.dimension()
2
>>> pc.remove_cell(Polyhedron(vertices=[(Integer(0), Integer(0)), (Integer(1),
↪ Integer(2))]))
>>> pc.dimension()
1
>>> pc
Polyhedral complex with 5 maximal cells
>>> pc.remove_cell(Polyhedron(vertices=[(Integer(1), Integer(2))]))
>>> pc.dimension()
1
>>> pc
Polyhedral complex with 3 maximal cells
>>> pc.remove_cell(Polyhedron(vertices=[(Integer(0), Integer(0))]))
>>> pc.dimension()
0
```

set_immutable()

Make this polyhedral complex immutable.

EXAMPLES:

```
sage: pc = PolyhedralComplex([Polyhedron(vertices=[[0], [1]])])
sage: pc.is_mutable()
True
sage: pc.set_immutable()
sage: pc.is_mutable()
False
```

```
>>> from sage.all import *
>>> pc = PolyhedralComplex([Polyhedron(vertices=[[Integer(0)],␣
↪[Integer(1)]])])
>>> pc.is_mutable()
True
>>> pc.set_immutable()
>>> pc.is_mutable()
False
```

stratify(n)

Return the pure sub-polyhedral complex which is constructed from the n-dimensional maximal cells of this polyhedral complex.

See also:

n_skeleton()

Warning: This may give the wrong answer if the polyhedral complex was constructed with `maximality_check` set to `False`.

EXAMPLES:

```
sage: p1 = Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)])
sage: p2 = Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])
sage: p3 = Polyhedron(vertices=[(1, 2), (0, 2)])
sage: pc = PolyhedralComplex([p1, p2, p3])
sage: pc.stratify(2) == pc
True
sage: pc.stratify(1)
Polyhedral complex with 0 maximal cells
```

```
>>> from sage.all import *
>>> p1 = Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0),␣
↪Integer(0)), (Integer(1), Integer(2))])
>>> p2 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
↪Integer(0)), (Integer(0), Integer(2))])
>>> p3 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),␣
↪Integer(2))])
>>> pc = PolyhedralComplex([p1, p2, p3])
>>> pc.stratify(Integer(2)) == pc
True
>>> pc.stratify(Integer(1))
Polyhedral complex with 0 maximal cells
```

Wrong answer due to `maximality_check=False`:

```
sage: pc_invalid = PolyhedralComplex([p1, p2, p3],
....:                                  maximality_check=False)
sage: pc_invalid.stratify(1)
Polyhedral complex with 1 maximal cell
```

```
>>> from sage.all import *
>>> pc_invalid = PolyhedralComplex([p1, p2, p3],
...                                  maximality_check=False)
>>> pc_invalid.stratify(Integer(1))
Polyhedral complex with 1 maximal cell
```

subdivide(*make_simplicial=False*, *new_vertices=None*, *new_rays=None*)

Construct a new polyhedral complex by iterative stellar subdivision of `self` for each new vertex/ray given.

Currently, subdivision is only supported for bounded polyhedral complex or polyhedral fan.

INPUT:

- `make_simplicial` – boolean (default: `False`); if `True`, the returned polyhedral complex is simplicial
- `new_vertices`, `new_rays` – list (optional); new generators to be added during subdivision

EXAMPLES:

```
sage: square_vertices = [(1, 1, 1), (-1, 1, 1), (-1, -1, 1), (1, -1, 1)]
sage: pc = PolyhedralComplex([
....:         Polyhedron(vertices=[(0, 0, 0)] + square_vertices),
....:         Polyhedron(vertices=[(0, 0, 2)] + square_vertices)])
sage: pc.is_compact() and not pc.is_simplicial_complex()
True
sage: subdivided_pc = pc.subdivide(new_vertices=[(0, 0, 1)])
sage: subdivided_pc
Polyhedral complex with 8 maximal cells
sage: subdivided_pc.is_simplicial_complex()
True
sage: simplicial_pc = pc.subdivide(make_simplicial=True)
sage: simplicial_pc
Polyhedral complex with 4 maximal cells
sage: simplicial_pc.is_simplicial_complex()
True

sage: fan = PolyhedralComplex([Polyhedron(rays=square_vertices)])
sage: fan.is_polyhedral_fan() and not fan.is_simplicial_fan()
True
sage: fan.subdivide(new_vertices=[(0, 0, 1)])
Traceback (most recent call last):
...
ValueError: new vertices cannot be used for subdivision
sage: subdivided_fan = fan.subdivide(new_rays=[(0, 0, 1)])
sage: subdivided_fan
Polyhedral complex with 4 maximal cells
sage: subdivided_fan.is_simplicial_fan()
True
sage: simplicial_fan = fan.subdivide(make_simplicial=True)
sage: simplicial_fan
Polyhedral complex with 2 maximal cells
sage: simplicial_fan.is_simplicial_fan()
True

sage: halfspace = PolyhedralComplex([Polyhedron(rays=[(0, 0, 1)],
....:               lines=[(1, 0, 0), (0, 1, 0)])])
sage: halfspace.is_simplicial_fan()
False
sage: subdiv_halfspace = halfspace.subdivide(make_simplicial=True)
sage: subdiv_halfspace
Polyhedral complex with 4 maximal cells
sage: subdiv_halfspace.is_simplicial_fan()
True
```

```
>>> from sage.all import *
>>> square_vertices = [(Integer(1), Integer(1), Integer(1)), (-Integer(1),␣
↪Integer(1), Integer(1)), (-Integer(1), -Integer(1), Integer(1)),␣
```

(continues on next page)

(continued from previous page)

```
↪(Integer(1), -Integer(1), Integer(1))]
>>> pc = PolyhedralComplex([
...         Polyhedron(vertices=[(Integer(0), Integer(0), Integer(0))] +␣
↪square_vertices),
...         Polyhedron(vertices=[(Integer(0), Integer(0), Integer(2))] +␣
↪square_vertices)])
>>> pc.is_compact() and not pc.is_simplicial_complex()
True
>>> subdivided_pc = pc.subdivide(new_vertices=[(Integer(0), Integer(0),␣
↪Integer(1))])
>>> subdivided_pc
Polyhedral complex with 8 maximal cells
>>> subdivided_pc.is_simplicial_complex()
True
>>> simplicial_pc = pc.subdivide(make_simplicial=True)
>>> simplicial_pc
Polyhedral complex with 4 maximal cells
>>> simplicial_pc.is_simplicial_complex()
True

>>> fan = PolyhedralComplex([Polyhedron(rays=square_vertices)])
>>> fan.is_polyhedral_fan() and not fan.is_simplicial_fan()
True
>>> fan.subdivide(new_vertices=[(Integer(0), Integer(0), Integer(1))])
Traceback (most recent call last):
...
ValueError: new vertices cannot be used for subdivision
>>> subdivided_fan = fan.subdivide(new_rays=[(Integer(0), Integer(0),␣
↪Integer(1))])
>>> subdivided_fan
Polyhedral complex with 4 maximal cells
>>> subdivided_fan.is_simplicial_fan()
True
>>> simplicial_fan = fan.subdivide(make_simplicial=True)
>>> simplicial_fan
Polyhedral complex with 2 maximal cells
>>> simplicial_fan.is_simplicial_fan()
True

>>> halfspace = PolyhedralComplex([Polyhedron(rays=[(Integer(0), Integer(0),␣
↪Integer(1))],
...             lines=[(Integer(1), Integer(0), Integer(0)), (Integer(0),␣
↪Integer(1), Integer(0))])])
>>> halfspace.is_simplicial_fan()
False
>>> subdiv_halfspace = halfspace.subdivide(make_simplicial=True)
>>> subdiv_halfspace
Polyhedral complex with 4 maximal cells
>>> subdiv_halfspace.is_simplicial_fan()
True
```

union(*right*)

The union of this polyhedral complex with another one.

INPUT:

- `right` – the other polyhedral complex (the right-hand factor)

EXAMPLES:

```
sage: p1 = Polyhedron(vertices=[(-1, 0), (0, 0), (0, 1)])
sage: p2 = Polyhedron(vertices=[(0, -1), (0, 0), (1, 0)])
sage: p3 = Polyhedron(vertices=[(0, -1), (1, -1), (1, 0)])
sage: pc = PolyhedralComplex([p1]).union(PolyhedralComplex([p3]))
sage: set(pc.maximal_cell_iterator()) == set([p1, p3])
True
sage: pc.union(PolyhedralComplex([p2]))
Polyhedral complex with 3 maximal cells
sage: p4 = Polyhedron(vertices=[(0, -1), (0, 0), (1, 0), (1, -1)])
sage: pc.union(PolyhedralComplex([p4]))
Traceback (most recent call last):
...
ValueError: the given cells are not face-to-face
```

```
>>> from sage.all import *
>>> p1 = Polyhedron(vertices=[(-Integer(1), Integer(0)), (Integer(0),␣
→Integer(0)), (Integer(0), Integer(1))])
>>> p2 = Polyhedron(vertices=[(Integer(0), -Integer(1)), (Integer(0),␣
→Integer(0)), (Integer(1), Integer(0))])
>>> p3 = Polyhedron(vertices=[(Integer(0), -Integer(1)), (Integer(1), -
→Integer(1)), (Integer(1), Integer(0))])
>>> pc = PolyhedralComplex([p1]).union(PolyhedralComplex([p3]))
>>> set(pc.maximal_cell_iterator()) == set([p1, p3])
True
>>> pc.union(PolyhedralComplex([p2]))
Polyhedral complex with 3 maximal cells
>>> p4 = Polyhedron(vertices=[(Integer(0), -Integer(1)), (Integer(0),␣
→Integer(0)), (Integer(1), Integer(0)), (Integer(1), -Integer(1))])
>>> pc.union(PolyhedralComplex([p4]))
Traceback (most recent call last):
...
ValueError: the given cells are not face-to-face
```

union_as_polyhedron()

Return `self` as a *Polyhedron* if `self` is convex.

EXAMPLES:

```
sage: p1 = Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)])
sage: p2 = Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])
sage: p3 = Polyhedron(vertices=[(0, 0), (1, 1), (2, 0)])
sage: P = PolyhedralComplex([p1, p2]).union_as_polyhedron()
sage: P.vertices_list()
[[0, 0], [0, 2], [1, 1], [1, 2]]
sage: PolyhedralComplex([p1, p3]).union_as_polyhedron()
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
ValueError: the polyhedral complex is not convex
```

```
>>> from sage.all import *
>>> p1 = Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0),
↪Integer(0)), (Integer(1), Integer(2))])
>>> p2 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0),
↪Integer(0)), (Integer(0), Integer(2))])
>>> p3 = Polyhedron(vertices=[(Integer(0), Integer(0)), (Integer(1),
↪Integer(1)), (Integer(2), Integer(0))])
>>> P = PolyhedralComplex([p1, p2]).union_as_polyhedron()
>>> P.vertices_list()
[[0, 0], [0, 2], [1, 1], [1, 2]]
>>> PolyhedralComplex([p1, p3]).union_as_polyhedron()
Traceback (most recent call last):
...
ValueError: the polyhedral complex is not convex
```

wedge (*right*)

The wedge (one-point union) of `self` with `right`.

Todo

Implement the wedge product of two polyhedral complexes.

EXAMPLES:

```
sage: pc = PolyhedralComplex([Polyhedron(vertices=[[0], [1]])])
sage: pc.wedge(pc)
Traceback (most recent call last):
...
NotImplementedError: wedge is not implemented for polyhedral complex
```

```
>>> from sage.all import *
>>> pc = PolyhedralComplex([Polyhedron(vertices=[[Integer(0)],
↪[Integer(1)]])])
>>> pc.wedge(pc)
Traceback (most recent call last):
...
NotImplementedError: wedge is not implemented for polyhedral complex
```

sage.geometry.polyhedral_complex.**cells_list_to_cells_dict** (*cells_list*)

Helper function that returns the dictionary whose keys are the dimensions, and the value associated to an integer d is the set of d-dimensional polyhedra in the given list.

EXAMPLES:

```
sage: p1 = Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)])
sage: p2 = Polyhedron(vertices=[(1, 1), (0, 0)])
sage: p3 = Polyhedron(vertices=[(0, 0)])
sage: p4 = Polyhedron(vertices=[(1, 1)])
sage: sage.geometry.polyhedral_complex.cells_list_to_cells_dict([p1, p2, p3, p4])
```

(continues on next page)

(continued from previous page)

```
{0: {A 0-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex,
  A 0-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex},
 1: {A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices},
 2: {A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 3 vertices}}
```

```
>>> from sage.all import *
>>> p1 = Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0), Integer(0)),␣
↪(Integer(1), Integer(2))])
>>> p2 = Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0), Integer(0))])
>>> p3 = Polyhedron(vertices=[(Integer(0), Integer(0))])
>>> p4 = Polyhedron(vertices=[(Integer(1), Integer(1))])
>>> sage.geometry.polyhedral_complex.cells_list_to_cells_dict([p1, p2, p3, p4])
{0: {A 0-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex,
  A 0-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex},
 1: {A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices},
 2: {A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 3 vertices}}
```

`sage.geometry.polyhedral_complex.`**`exploded_plot`**(*polyhedra*, *center*, *explosion_factor*, *sticky_vertices*, *sticky_center*, *point*, ***kwds*)

Return a plot of several `polyhedra` in one figure with extra space between them.

INPUT:

- `polyhedra` – an iterable of *`Polyhedron_base`* objects
- `center` – (default: `None`, denoting the origin) the center of explosion
- `explosion_factor` – (default: 1) a nonnegative number; translate polyhedra by this factor of the distance from `center` to their center
- `sticky_vertices` – (default: `False`) boolean or dict; whether to draw line segments between shared vertices of the given polyhedra. A dict gives options for `sage.plot.line()`.
- `sticky_center` – (default: `True`) boolean or dict. When `center` is a vertex of some of the polyhedra, whether to draw line segments connecting the `center` to the shifted copies of these vertices. A dict gives options for `sage.plot.line()`.
- `color` – (default: `None`) if `'rainbow'`, assign a different color to every maximal cell and every vertex; otherwise, passed on to `plot()`
- other keyword arguments are passed on to `plot()`

EXAMPLES:

```
sage: from sage.geometry.polyhedral_complex import exploded_plot
sage: p1 = Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)])
sage: p2 = Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])
sage: p3 = Polyhedron(vertices=[(0, 0), (1, 1), (2, 0)])
sage: exploded_plot([p1, p2, p3])                                            #␣
↪needs sage.plot
Graphics object consisting of 20 graphics primitives
sage: exploded_plot([p1, p2, p3], center=(1, 1))                             #␣
↪needs sage.plot
Graphics object consisting of 19 graphics primitives
sage: exploded_plot([p1, p2, p3], center=(1, 1), sticky_vertices=True)       #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.plot
Graphics object consisting of 23 graphics primitives
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedral_complex import exploded_plot
>>> p1 = Polyhedron(vertices=[(Integer(1), Integer(1)), (Integer(0), Integer(0)),␣
↪(Integer(1), Integer(2))])
>>> p2 = Polyhedron(vertices=[(Integer(1), Integer(2)), (Integer(0), Integer(0)),␣
↪(Integer(0), Integer(2))])
>>> p3 = Polyhedron(vertices=[(Integer(0), Integer(0)), (Integer(1), Integer(1)),␣
↪(Integer(2), Integer(0))])
>>> exploded_plot([p1, p2, p3])                                           #␣
↪needs sage.plot
Graphics object consisting of 20 graphics primitives
>>> exploded_plot([p1, p2, p3], center=(Integer(1), Integer(1)))                ␣
↪               # needs sage.plot
Graphics object consisting of 19 graphics primitives
>>> exploded_plot([p1, p2, p3], center=(Integer(1), Integer(1)), sticky_
↪vertices=True)          # needs sage.plot
Graphics object consisting of 23 graphics primitives
```

2.5 Toric geometry

2.5.1 Toric lattices

This module was designed as a part of the framework for toric varieties (`variety`, `fano_variety`).

All toric lattices are isomorphic to $\mathbf{Z}^n$ for some n, but will prevent you from doing "wrong" operations with objects from different lattices.

AUTHORS:

- Andrey Novoseltsev (2010-05-27): initial version.
- Andrey Novoseltsev (2010-07-30): sublattices and quotients.

EXAMPLES:

The simplest way to create a toric lattice is to specify its dimension only:

```
sage: N = ToricLattice(3)
sage: N
3-d lattice N
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(3))
>>> N
3-d lattice N
```

While our lattice `N` is called exactly "N" it is a coincidence: all lattices are called "N" by default:

```
sage: another_name = ToricLattice(3)
sage: another_name
3-d lattice N
```

```
>>> from sage.all import *
>>> another_name = ToricLattice(Integer(3))
>>> another_name
3-d lattice N
```

If fact, the above lattice is exactly the same as before as an object in memory:

```
sage: N is another_name
True
```

```
>>> from sage.all import *
>>> N is another_name
True
```

There are actually four names associated to a toric lattice and they all must be the same for two lattices to coincide:

```
sage: N, N.dual(), latex(N), latex(N.dual())
(3-d lattice N, 3-d lattice M, N, M)
```

```
>>> from sage.all import *
>>> N, N.dual(), latex(N), latex(N.dual())
(3-d lattice N, 3-d lattice M, N, M)
```

Notice that the lattice dual to `N` is called "M" which is standard in toric geometry. This happens only if you allow completely automatic handling of names:

```
sage: another_N = ToricLattice(3, "N")
sage: another_N.dual()
3-d lattice N*
sage: N is another_N
False
```

```
>>> from sage.all import *
>>> another_N = ToricLattice(Integer(3), "N")
>>> another_N.dual()
3-d lattice N*
>>> N is another_N
False
```

What can you do with toric lattices? Well, their main purpose is to allow creation of elements of toric lattices:

```
sage: n = N([1,2,3])
sage: n
N(1, 2, 3)
sage: M = N.dual()
sage: m = M(1,2,3)
sage: m
M(1, 2, 3)
```

```
>>> from sage.all import *
>>> n = N([Integer(1),Integer(2),Integer(3)])
>>> n
```

(continues on next page)

(continued from previous page)

```
N(1, 2, 3)
>>> M = N.dual()
>>> m = M(Integer(1),Integer(2),Integer(3))
>>> m
M(1, 2, 3)
```

Dual lattices can act on each other:

```
sage: n * m
14
sage: m * n
14
```

```
>>> from sage.all import *
>>> n * m
14
>>> m * n
14
```

You can also add elements of the same lattice or scale them:

```
sage: 2 * n
N(2, 4, 6)
sage: n * 2
N(2, 4, 6)
sage: n + n
N(2, 4, 6)
```

```
>>> from sage.all import *
>>> Integer(2) * n
N(2, 4, 6)
>>> n * Integer(2)
N(2, 4, 6)
>>> n + n
N(2, 4, 6)
```

However, you cannot "mix wrong lattices" in your expressions:

```
sage: n + m
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +:
'3-d lattice N' and '3-d lattice M'
sage: n * n
Traceback (most recent call last):
...
TypeError: elements of the same toric lattice cannot be multiplied!
sage: n == m
False
```

```
>>> from sage.all import *
>>> n + m
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for +:
'3-d lattice N' and '3-d lattice M'
>>> n * n
Traceback (most recent call last):
...
TypeError: elements of the same toric lattice cannot be multiplied!
>>> n == m
False
```

Note that n and m are not equal to each other even though they are both "just (1,2,3)." Moreover, you cannot easily convert elements between toric lattices:

```
sage: M(n)
Traceback (most recent call last):
...
TypeError: N(1, 2, 3) cannot be converted to 3-d lattice M!
```

```
>>> from sage.all import *
>>> M(n)
Traceback (most recent call last):
...
TypeError: N(1, 2, 3) cannot be converted to 3-d lattice M!
```

If you really need to consider elements of one lattice as elements of another, you can either use intermediate conversion to "just a vector":

```
sage: ZZ3 = ZZ^3
sage: n_in_M = M(ZZ3(n))
sage: n_in_M
M(1, 2, 3)
sage: n == n_in_M
False
sage: n_in_M == m
True
```

```
>>> from sage.all import *
>>> ZZ3 = ZZ**Integer(3)
>>> n_in_M = M(ZZ3(n))
>>> n_in_M
M(1, 2, 3)
>>> n == n_in_M
False
>>> n_in_M == m
True
```

Or you can create a homomorphism from one lattice to any other:

```
sage: h = N.hom(identity_matrix(3), M)
sage: h(n)
M(1, 2, 3)
```

```
>>> from sage.all import *
>>> h = N.hom(identity_matrix(Integer(3)), M)
>>> h(n)
M(1, 2, 3)
```

Warning

While integer vectors (elements of $\mathbf{Z}^n$) are printed as `(1,2,3)`, in the code `(1,2,3)` is a `tuple`, which has nothing to do neither with vectors, nor with toric lattices, so the following is probably not what you want while working with toric geometry objects:

```
sage: (1,2,3) + (1,2,3)
(1, 2, 3, 1, 2, 3)
```

```
>>> from sage.all import *
>>> (Integer(1),Integer(2),Integer(3)) + (Integer(1),Integer(2),Integer(3))
(1, 2, 3, 1, 2, 3)
```

Instead, use syntax like

```
sage: N(1,2,3) + N(1,2,3)
N(2, 4, 6)
```

```
>>> from sage.all import *
>>> N(Integer(1),Integer(2),Integer(3)) + N(Integer(1),Integer(2),Integer(3))
N(2, 4, 6)
```

class sage.geometry.toric_lattice.**ToricLatticeFactory**

Bases: `UniqueFactory`

Create a lattice for toric geometry objects.

INPUT:

- `rank` – nonnegative integer; the only mandatory parameter
- `name` – string
- `dual_name` – string
- `latex_name` – string
- `latex_dual_name` – string

OUTPUT: lattice

A toric lattice is uniquely determined by its rank and associated names. There are four such “associated names” whose meaning should be clear from the names of the corresponding parameters, but the choice of default values is a little bit involved. So here is the full description of the “naming algorithm”:

1. If no names were given at all, then this lattice will be called “N” and the dual one “M”. These are the standard choices in toric geometry.
2. If `name` was given and `dual_name` was not, then `dual_name` will be `name` followed by “*”.
3. If LaTeX names were not given, they will coincide with the “usual” names, but if `dual_name` was constructed automatically, the trailing star will be typeset as a superscript.

EXAMPLES:

Let's start with no names at all and see how automatic names are given:

```
sage: L1 = ToricLattice(3)
sage: L1
3-d lattice N
sage: L1.dual()
3-d lattice M
```

```
>>> from sage.all import *
>>> L1 = ToricLattice(Integer(3))
>>> L1
3-d lattice N
>>> L1.dual()
3-d lattice M
```

If we give the name "N" explicitly, the dual lattice will be called "N*":

```
sage: L2 = ToricLattice(3, "N")
sage: L2
3-d lattice N
sage: L2.dual()
3-d lattice N*
```

```
>>> from sage.all import *
>>> L2 = ToricLattice(Integer(3), "N")
>>> L2
3-d lattice N
>>> L2.dual()
3-d lattice N*
```

However, we can give an explicit name for it too:

```
sage: L3 = ToricLattice(3, "N", "M")
sage: L3
3-d lattice N
sage: L3.dual()
3-d lattice M
```

```
>>> from sage.all import *
>>> L3 = ToricLattice(Integer(3), "N", "M")
>>> L3
3-d lattice N
>>> L3.dual()
3-d lattice M
```

If you want, you may also give explicit LaTeX names:

```
sage: L4 = ToricLattice(3, "N", "M", r"\mathbb{N}", r"\mathbb{M}")
sage: latex(L4)
\mathbb{N}
sage: latex(L4.dual())
\mathbb{M}
```

```
>>> from sage.all import *
>>> L4 = ToricLattice(Integer(3), "N", "M", r"\mathbb{N}", r"\mathbb{M}")
>>> latex(L4)
\mathbb{N}
>>> latex(L4.dual())
\mathbb{M}
```

While all four lattices above are called "N", only two of them are equal (and are actually the same):

```
sage: L1 == L2
False
sage: L1 == L3
True
sage: L1 is L3
True
sage: L1 == L4
False
```

```
>>> from sage.all import *
>>> L1 == L2
False
>>> L1 == L3
True
>>> L1 is L3
True
>>> L1 == L4
False
```

The reason for this is that `L2` and `L4` have different names either for dual lattices or for LaTeX typesetting.

create_key(*rank*, *name=None*, *dual_name=None*, *latex_name=None*, *latex_dual_name=None*)

Create a key that uniquely identifies this toric lattice.

See *ToricLattice* for documentation.

> **Warning**
>
> You probably should not use this function directly.

create_object(*version*, *key*)

Create the toric lattice described by `key`.

See *ToricLattice* for documentation.

> **Warning**
>
> You probably should not use this function directly.

class sage.geometry.toric_lattice.**ToricLattice_ambient**(*rank*, *name*, *dual_name*, *latex_name*, *latex_dual_name*)

Bases: *ToricLattice_generic*, `FreeModule_ambient_pid`

Create a toric lattice.

See *ToricLattice* for documentation.

> **Warning**
>
> There should be only one toric lattice with the given rank and associated names. Using this class directly to create toric lattices may lead to unexpected results. Please, use *ToricLattice* to create toric lattices.

Element

alias of `ToricLatticeElement`

ambient_module()

Return the ambient module of `self`.

OUTPUT: *toric lattice*

> **Note**
>
> For any ambient toric lattice its ambient module is the lattice itself.

EXAMPLES:

```
sage: N = ToricLattice(3)
sage: N.ambient_module()
3-d lattice N
sage: N.ambient_module() is N
True
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(3))
>>> N.ambient_module()
3-d lattice N
>>> N.ambient_module() is N
True
```

dual()

Return the lattice dual to `self`.

OUTPUT: *toric lattice*

EXAMPLES:

```
sage: N = ToricLattice(3)
sage: N
3-d lattice N
sage: M = N.dual()
sage: M
3-d lattice M
sage: M.dual() is N
True
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(3))
>>> N
3-d lattice N
>>> M = N.dual()
>>> M
3-d lattice M
>>> M.dual() is N
True
```

Elements of dual lattices can act on each other:

```
sage: n = N(1,2,3)
sage: m = M(4,5,6)
sage: n * m
32
sage: m * n
32
```

```
>>> from sage.all import *
>>> n = N(Integer(1),Integer(2),Integer(3))
>>> m = M(Integer(4),Integer(5),Integer(6))
>>> n * m
32
>>> m * n
32
```

plot(***options*)

Plot `self`.

INPUT:

- any options for toric plots (see *toric_plotter.options*), none are mandatory.

OUTPUT: a plot

EXAMPLES:

```
sage: N = ToricLattice(3)
sage: N.plot()                                                        #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(3))
>>> N.plot()                                                      #␣
↪needs sage.plot
Graphics3d Object
```

class sage.geometry.toric_lattice.**ToricLattice_generic**(*base_ring*, *rank*, *degree*, *sparse=False*, *coordinate_ring=None*, *category=None*)

Bases: `FreeModule_generic_pid`

Abstract base class for toric lattices.

Element

alias of `ToricLatticeElement`

construction()

Return the functorial construction of `self`.

OUTPUT:

`None`, we do not think of toric lattices as constructed from simpler objects since we do not want to perform arithmetic involving different lattices.

direct_sum(*other*)

Return the direct sum with `other`.

INPUT:

- `other` – a toric lattice or more general module

OUTPUT:

The direct sum of `self` and `other` as **Z**-modules. If `other` is a *ToricLattice*, another toric lattice will be returned.

EXAMPLES:

```
sage: K = ToricLattice(3, 'K')
sage: L = ToricLattice(3, 'L')
sage: N = K.direct_sum(L); N
6-d lattice K+L
sage: N, N.dual(), latex(N), latex(N.dual())
(6-d lattice K+L, 6-d lattice K*+L*, K \oplus L, K^* \oplus L^*)
```

```
>>> from sage.all import *
>>> K = ToricLattice(Integer(3), 'K')
>>> L = ToricLattice(Integer(3), 'L')
>>> N = K.direct_sum(L); N
6-d lattice K+L
>>> N, N.dual(), latex(N), latex(N.dual())
(6-d lattice K+L, 6-d lattice K*+L*, K \oplus L, K^* \oplus L^*)
```

With default names:

```
sage: N = ToricLattice(3).direct_sum(ToricLattice(2))
sage: N, N.dual(), latex(N), latex(N.dual())
(5-d lattice N+N, 5-d lattice M+M, N \oplus N, M \oplus M)
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(3)).direct_sum(ToricLattice(Integer(2)))
>>> N, N.dual(), latex(N), latex(N.dual())
(5-d lattice N+N, 5-d lattice M+M, N \oplus N, M \oplus M)
```

If `other` is not a *ToricLattice*, fall back to sum of modules:

```
sage: ToricLattice(3).direct_sum(ZZ^2)
Free module of degree 5 and rank 5 over Integer Ring
Echelon basis matrix:
[1 0 0 0 0]
```

(continues on next page)

(continued from previous page)

```
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
```

```
>>> from sage.all import *
>>> ToricLattice(Integer(3)).direct_sum(ZZ**Integer(2))
Free module of degree 5 and rank 5 over Integer Ring
Echelon basis matrix:
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
```

intersection(*other*)

Return the intersection of `self` and `other`.

INPUT:

- `other` – a toric (sub)lattice.dual

OUTPUT:

- a toric (sub)lattice.

EXAMPLES:

```
sage: N = ToricLattice(3)
sage: Ns1 = N.submodule([N(2,4,0), N(9,12,0)])
sage: Ns2 = N.submodule([N(1,4,9), N(9,2,0)])
sage: Ns1.intersection(Ns2)
Sublattice <N(54, 12, 0)>
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(3))
>>> Ns1 = N.submodule([N(Integer(2),Integer(4),Integer(0)), N(Integer(9),
↪Integer(12),Integer(0))])
>>> Ns2 = N.submodule([N(Integer(1),Integer(4),Integer(9)), N(Integer(9),
↪Integer(2),Integer(0))])
>>> Ns1.intersection(Ns2)
Sublattice <N(54, 12, 0)>
```

Note that if one of the intersecting sublattices is a sublattice of another, no new lattices will be constructed:

```
sage: N.intersection(N) is N
True
sage: Ns1.intersection(N) is Ns1
True
sage: N.intersection(Ns1) is Ns1
True
```

```
>>> from sage.all import *
>>> N.intersection(N) is N
```

(continues on next page)

(continued from previous page)

```
True
>>> Ns1.intersection(N) is Ns1
True
>>> N.intersection(Ns1) is Ns1
True
```

quotient(*sub*, *check=True*, *positive_point=None*, *positive_dual_point=None*, ***kwds*)

Return the quotient of `self` by the given sublattice `sub`.

INPUT:

- `sub` – sublattice of self
- `check` – boolean (default: `True`); whether or not to check that `sub` is a valid sublattice

If the quotient is one-dimensional and torsion free, the following two mutually exclusive keyword arguments are also allowed. They decide the sign choice for the (single) generator of the quotient lattice:

- `positive_point` – a lattice point of `self` not in the sublattice `sub` (that is, not zero in the quotient lattice). The quotient generator will be in the same direction as `positive_point`.
- `positive_dual_point` – a dual lattice point. The quotient generator will be chosen such that its lift has a positive product with `positive_dual_point`. Note: if `positive_dual_point` is not zero on the sublattice `sub`, then the notion of positivity will depend on the choice of lift!

Further named arguments are passed to the constructor of a toric lattice quotient.

EXAMPLES:

```
sage: N = ToricLattice(3)
sage: Ns = N.submodule([N(2,4,0), N(9,12,0)])
sage: Q = N/Ns
sage: Q
Quotient with torsion of 3-d lattice N
by Sublattice <N(1, 8, 0), N(0, 12, 0)>
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(3))
>>> Ns = N.submodule([N(Integer(2),Integer(4),Integer(0)), N(Integer(9),
↪Integer(12),Integer(0))])
>>> Q = N/Ns
>>> Q
Quotient with torsion of 3-d lattice N
by Sublattice <N(1, 8, 0), N(0, 12, 0)>
```

Attempting to quotient one lattice by a sublattice of another will result in a `ValueError`:

```
sage: N = ToricLattice(3)
sage: M = ToricLattice(3, name='M')
sage: Ms = M.submodule([M(2,4,0), M(9,12,0)])
sage: N.quotient(Ms)
Traceback (most recent call last):
...
ValueError: M(1, 8, 0) cannot generate a sublattice of
3-d lattice N
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(3))
>>> M = ToricLattice(Integer(3), name='M')
>>> Ms = M.submodule([M(Integer(2),Integer(4),Integer(0)), M(Integer(9),
↪Integer(12),Integer(0))])
>>> N.quotient(Ms)
Traceback (most recent call last):
...
ValueError: M(1, 8, 0) cannot generate a sublattice of
3-d lattice N
```

However, if we forget the sublattice structure, then it is possible to quotient by vector spaces or modules constructed from any sublattice:

```
sage: N = ToricLattice(3)
sage: M = ToricLattice(3, name='M')
sage: Ms = M.submodule([M(2,4,0), M(9,12,0)])
sage: N.quotient(Ms.vector_space())
Quotient with torsion of 3-d lattice N by Sublattice
<N(1, 8, 0), N(0, 12, 0)>
sage: N.quotient(Ms.sparse_module())
Quotient with torsion of 3-d lattice N by Sublattice
<N(1, 8, 0), N(0, 12, 0)>
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(3))
>>> M = ToricLattice(Integer(3), name='M')
>>> Ms = M.submodule([M(Integer(2),Integer(4),Integer(0)), M(Integer(9),
↪Integer(12),Integer(0))])
>>> N.quotient(Ms.vector_space())
Quotient with torsion of 3-d lattice N by Sublattice
<N(1, 8, 0), N(0, 12, 0)>
>>> N.quotient(Ms.sparse_module())
Quotient with torsion of 3-d lattice N by Sublattice
<N(1, 8, 0), N(0, 12, 0)>
```

See `ToricLattice_quotient` for more examples.

saturation()

Return the saturation of `self`.

OUTPUT: a `toric lattice`

EXAMPLES:

```
sage: N = ToricLattice(3)
sage: Ns = N.submodule([(1,2,3), (4,5,6)])
sage: Ns
Sublattice <N(1, 2, 3), N(0, 3, 6)>
sage: Ns_sat = Ns.saturation()
sage: Ns_sat
Sublattice <N(1, 0, -1), N(0, 1, 2)>
sage: Ns_sat is Ns_sat.saturation()
True
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(3))
>>> Ns = N.submodule([(Integer(1),Integer(2),Integer(3)), (Integer(4),
↪Integer(5),Integer(6))])
>>> Ns
Sublattice <N(1, 2, 3), N(0, 3, 6)>
>>> Ns_sat = Ns.saturation()
>>> Ns_sat
Sublattice <N(1, 0, -1), N(0, 1, 2)>
>>> Ns_sat is Ns_sat.saturation()
True
```

span(*gens*, *base_ring=Integer Ring*, **args*, ***kwds*)

Return the span of the given generators.

INPUT:

- `gens` – list of elements of the ambient vector space of `self`
- `base_ring` – (default: $\mathbf{Z}$) base ring for the generated module

OUTPUT: submodule spanned by `gens`

> **Note**
>
> The output need not be a submodule of `self`, nor even of the ambient space. It must, however, be contained in the ambient vector space.

See also *span_of_basis()*, `submodule()`, and `submodule_with_basis()`,

EXAMPLES:

```
sage: N = ToricLattice(3)
sage: Ns = N.submodule([N.gen(0)])
sage: Ns.span([N.gen(1)])
Sublattice <N(0, 1, 0)>
sage: Ns.submodule([N.gen(1)])
Traceback (most recent call last):
...
ArithmeticError: argument gens (= [N(0, 1, 0)]) does not generate a submodule␣
↪of self
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(3))
>>> Ns = N.submodule([N.gen(Integer(0))])
>>> Ns.span([N.gen(Integer(1))])
Sublattice <N(0, 1, 0)>
>>> Ns.submodule([N.gen(Integer(1))])
Traceback (most recent call last):
...
ArithmeticError: argument gens (= [N(0, 1, 0)]) does not generate a submodule␣
↪of self
```

span_of_basis(*basis*, *base_ring=Integer Ring*, **args*, ***kwds*)

Return the submodule with the given `basis`.

INPUT:

- `basis` – list of elements of the ambient vector space of `self`
- `base_ring` – (default: $\mathbf{Z}$) base ring for the generated module

OUTPUT: submodule spanned by `basis`

Note: The output need not be a submodule of `self`, nor even of the ambient space. It must, however, be contained in the ambient vector space.

See also `span()`, `submodule()`, and `submodule_with_basis()`,

EXAMPLES:

```
sage: N = ToricLattice(3)
sage: Ns = N.span_of_basis([(1,2,3)])
sage: Ns.span_of_basis([(2,4,0)])
Sublattice <N(2, 4, 0)>
sage: Ns.span_of_basis([(1/5,2/5,0), (1/7,1/7,0)])
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1/5 2/5   0]
[1/7 1/7   0]
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(3))
>>> Ns = N.span_of_basis([(Integer(1),Integer(2),Integer(3))])
>>> Ns.span_of_basis([(Integer(2),Integer(4),Integer(0))])
Sublattice <N(2, 4, 0)>
>>> Ns.span_of_basis([(Integer(1)/Integer(5),Integer(2)/Integer(5),
↪Integer(0)), (Integer(1)/Integer(7),Integer(1)/Integer(7),Integer(0))])
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1/5 2/5   0]
[1/7 1/7   0]
```

Of course the input basis vectors must be linearly independent:

```
sage: Ns.span_of_basis([(1,2,0), (2,4,0)])
Traceback (most recent call last):
...
ValueError: The given basis vectors must be linearly independent.
```

```
>>> from sage.all import *
>>> Ns.span_of_basis([(Integer(1),Integer(2),Integer(0)), (Integer(2),
↪Integer(4),Integer(0))])
Traceback (most recent call last):
...
ValueError: The given basis vectors must be linearly independent.
```

class sage.geometry.toric_lattice.**ToricLattice_quotient**(*V*, *W*, *check=True*, *positive_point=None*, *positive_dual_point=None*, ***kwds*)

Bases: `FGP_Module_class`

Construct the quotient of a toric lattice `V` by its sublattice `W`.

INPUT:

- `V` – ambient toric lattice
- `W` – sublattice of `V`
- `check` – boolean (default: `True`); whether to check correctness of input or not

If the quotient is one-dimensional and torsion free, the following two mutually exclusive keyword arguments are also allowed. They decide the sign choice for the (single) generator of the quotient lattice:

- `positive_point` – a lattice point of `self` not in the sublattice `sub` (that is, not zero in the quotient lattice). The quotient generator will be in the same direction as `positive_point`.
- `positive_dual_point` – a dual lattice point. The quotient generator will be chosen such that its lift has a positive product with `positive_dual_point`. Note: if `positive_dual_point` is not zero on the sublattice `sub`, then the notion of positivity will depend on the choice of lift!

Further given named arguments are passed to the constructor of an FGP module.

OUTPUT: quotient of `V` by `W`

EXAMPLES:

The intended way to get objects of this class is to use `quotient()` method of toric lattices:

```
sage: N = ToricLattice(3)
sage: sublattice = N.submodule([(1,1,0), (3,2,1)])
sage: Q = N/sublattice
sage: Q
1-d lattice, quotient of 3-d lattice N by Sublattice <N(1, 0, 1), N(0, 1, -1)>
sage: Q.gens()
(N[1, 0, 0],)
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(3))
>>> sublattice = N.submodule([(Integer(1),Integer(1),Integer(0)), (Integer(3),
↪Integer(2),Integer(1))])
>>> Q = N/sublattice
>>> Q
1-d lattice, quotient of 3-d lattice N by Sublattice <N(1, 0, 1), N(0, 1, -1)>
>>> Q.gens()
(N[1, 0, 0],)
```

Here, `sublattice` happens to be of codimension one in `N`. If you want to prescribe the sign of the quotient generator, you can do either:

```
sage: Q = N.quotient(sublattice, positive_point=N(0,0,-1)); Q
1-d lattice, quotient of 3-d lattice N by Sublattice <N(1, 0, 1), N(0, 1, -1)>
sage: Q.gens()
(N[1, 0, 0],)
```

```
>>> from sage.all import *
>>> Q = N.quotient(sublattice, positive_point=N(Integer(0),Integer(0),-
↪Integer(1))); Q
1-d lattice, quotient of 3-d lattice N by Sublattice <N(1, 0, 1), N(0, 1, -1)>
>>> Q.gens()
(N[1, 0, 0],)
```

or:

```
sage: M = N.dual()
sage: Q = N.quotient(sublattice, positive_dual_point=M(1,0,0)); Q
1-d lattice, quotient of 3-d lattice N by Sublattice <N(1, 0, 1), N(0, 1, -1)>
sage: Q.gens()
(N[1, 0, 0],)
```

```
>>> from sage.all import *
>>> M = N.dual()
>>> Q = N.quotient(sublattice, positive_dual_point=M(Integer(1),Integer(0),
↪Integer(0))); Q
1-d lattice, quotient of 3-d lattice N by Sublattice <N(1, 0, 1), N(0, 1, -1)>
>>> Q.gens()
(N[1, 0, 0],)
```

Element

alias of *ToricLattice_quotient_element*

base_extend(R)

Return the base change of `self` to the ring `R`.

INPUT:

- `R` – either $\mathbf{Z}$ or $\mathbf{Q}$

OUTPUT: `self` if $R = \mathbf{Z}$, quotient of the base extension of the ambient lattice by the base extension of the sublattice if $R = \mathbf{Q}$

EXAMPLES:

```
sage: N = ToricLattice(3)
sage: Ns = N.submodule([N(2,4,0), N(9,12,0)])
sage: Q = N/Ns
sage: Q.base_extend(ZZ) is Q
True
sage: Q.base_extend(QQ)
Vector space quotient V/W of dimension 1 over Rational Field where
V: Vector space of dimension 3 over Rational Field
W: Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[1 0 0]
[0 1 0]
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(3))
>>> Ns = N.submodule([N(Integer(2),Integer(4),Integer(0)), N(Integer(9),
```

(continues on next page)

(continued from previous page)

```
↪Integer(12),Integer(0))])
>>> Q = N/Ns
>>> Q.base_extend(ZZ) is Q
True
>>> Q.base_extend(QQ)
Vector space quotient V/W of dimension 1 over Rational Field where
V: Vector space of dimension 3 over Rational Field
W: Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[1 0 0]
[0 1 0]
```

coordinate_vector(*x*, *reduce=False*)

Return coordinates of `x` with respect to the optimized representation of `self`.

INPUT:

- `x` – element of `self` or convertible to `self`
- `reduce` – (default: `False`) if `True`, reduce coefficients modulo invariants

OUTPUT: the coordinates as a vector

EXAMPLES:

```
sage: N = ToricLattice(3)
sage: Q = N.quotient(N.span([N(1,2,3), N(0,2,1)]), positive_point=N(0,-1,0))
sage: q = Q.gen(0); q
N[0, -1, 0]
sage: q.vector()  # indirect test
(1)
sage: Q.coordinate_vector(q)
(1)
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(3))
>>> Q = N.quotient(N.span([N(Integer(1),Integer(2),Integer(3)), N(Integer(0),
↪Integer(2),Integer(1))]), positive_point=N(Integer(0),-Integer(1),
↪Integer(0)))
>>> q = Q.gen(Integer(0)); q
N[0, -1, 0]
>>> q.vector()  # indirect test
(1)
>>> Q.coordinate_vector(q)
(1)
```

dimension()

Return the rank of `self`.

OUTPUT: integer; the dimension of the free part of the quotient

EXAMPLES:

```
sage: N = ToricLattice(3)
sage: Ns = N.submodule([N(2,4,0), N(9,12,0)])
```

(continues on next page)

(continued from previous page)

```
sage: Q = N/Ns
sage: Q.ngens()
2
sage: Q.rank()
1
sage: Ns = N.submodule([N(1,4,0)])
sage: Q = N/Ns
sage: Q.ngens()
2
sage: Q.rank()
2
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(3))
>>> Ns = N.submodule([N(Integer(2),Integer(4),Integer(0)), N(Integer(9),
↪Integer(12),Integer(0))])
>>> Q = N/Ns
>>> Q.ngens()
2
>>> Q.rank()
1
>>> Ns = N.submodule([N(Integer(1),Integer(4),Integer(0))])
>>> Q = N/Ns
>>> Q.ngens()
2
>>> Q.rank()
2
```

dual()

Return the lattice dual to `self`.

OUTPUT: a *toric lattice quotient*

EXAMPLES:

```
sage: N = ToricLattice(3)
sage: Ns = N.submodule([(1, -1, -1)])
sage: Q = N / Ns
sage: Q.dual()
Sublattice <M(1, 0, 1), M(0, 1, -1)>
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(3))
>>> Ns = N.submodule([(Integer(1), -Integer(1), -Integer(1))])
>>> Q = N / Ns
>>> Q.dual()
Sublattice <M(1, 0, 1), M(0, 1, -1)>
```

gens()

Return the generators of the quotient.

OUTPUT:

A tuple of *ToricLattice_quotient_element* generating the quotient.

EXAMPLES:

```
sage: N = ToricLattice(3)
sage: Q = N.quotient(N.span([N(1,2,3), N(0,2,1)]), positive_point=N(0,-1,0))
sage: Q.gens()
(N[0, -1, 0],)
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(3))
>>> Q = N.quotient(N.span([N(Integer(1),Integer(2),Integer(3)), N(Integer(0),
↪Integer(2),Integer(1))]), positive_point=N(Integer(0),-Integer(1),
↪Integer(0)))
>>> Q.gens()
(N[0, -1, 0],)
```

is_torsion_free()

Check if `self` is torsion-free.

OUTPUT: `True` if `self` has no torsion and `False` otherwise

EXAMPLES:

```
sage: N = ToricLattice(3)
sage: Ns = N.submodule([N(2,4,0), N(9,12,0)])
sage: Q = N/Ns
sage: Q.is_torsion_free()
False
sage: Ns = N.submodule([N(1,4,0)])
sage: Q = N/Ns
sage: Q.is_torsion_free()
True
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(3))
>>> Ns = N.submodule([N(Integer(2),Integer(4),Integer(0)), N(Integer(9),
↪Integer(12),Integer(0))])
>>> Q = N/Ns
>>> Q.is_torsion_free()
False
>>> Ns = N.submodule([N(Integer(1),Integer(4),Integer(0))])
>>> Q = N/Ns
>>> Q.is_torsion_free()
True
```

rank()

Return the rank of `self`.

OUTPUT: integer; the dimension of the free part of the quotient

EXAMPLES:

```
sage: N = ToricLattice(3)
sage: Ns = N.submodule([N(2,4,0), N(9,12,0)])
sage: Q = N/Ns
sage: Q.ngens()
```

(continues on next page)

(continued from previous page)

```
2
sage: Q.rank()
1
sage: Ns = N.submodule([N(1,4,0)])
sage: Q = N/Ns
sage: Q.ngens()
2
sage: Q.rank()
2
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(3))
>>> Ns = N.submodule([N(Integer(2),Integer(4),Integer(0)), N(Integer(9),
↪Integer(12),Integer(0))])
>>> Q = N/Ns
>>> Q.ngens()
2
>>> Q.rank()
1
>>> Ns = N.submodule([N(Integer(1),Integer(4),Integer(0))])
>>> Q = N/Ns
>>> Q.ngens()
2
>>> Q.rank()
2
```

class sage.geometry.toric_lattice.**ToricLattice_quotient_element**(*parent*, *x*, *check=True*)

Bases: FGP_Element

Create an element of a toric lattice quotient.

> **Warning**
>
> You probably should not construct such elements explicitly.

INPUT:

- same as for FGP_Element.

OUTPUT: element of a toric lattice quotient

set_immutable()

Make self immutable.

OUTPUT: none

> **Note**
>
> Elements of toric lattice quotients are always immutable, so this method does nothing, it is introduced for compatibility purposes only.

EXAMPLES:

```
sage: N = ToricLattice(3)
sage: Ns = N.submodule([N(2,4,0), N(9,12,0)])
sage: Q = N/Ns
sage: Q.0.set_immutable()
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(3))
>>> Ns = N.submodule([N(Integer(2),Integer(4),Integer(0)), N(Integer(9),
↪Integer(12),Integer(0))])
>>> Q = N/Ns
>>> Q.gen(0).set_immutable()
```

class sage.geometry.toric_lattice.**ToricLattice_sublattice**(*ambient*, *gens*, *check=True*, *already_echelonized=False*, *category=None*)

Bases: *ToricLattice_sublattice_with_basis*, FreeModule_submodule_pid

Construct the sublattice of `ambient` toric lattice generated by `gens`.

INPUT (same as for FreeModule_submodule_pid):

- `ambient` – ambient *toric lattice* for this sublattice
- `gens` – list of elements of `ambient` generating the constructed sublattice
- see the base class for other available options

OUTPUT: sublattice of a toric lattice with an automatically chosen basis

See also *ToricLattice_sublattice_with_basis* if you want to specify an explicit basis.

EXAMPLES:

The intended way to get objects of this class is to use `submodule()` method of toric lattices:

```
sage: N = ToricLattice(3)
sage: sublattice = N.submodule([(1,1,0), (3,2,1)])
sage: sublattice.has_user_basis()
False
sage: sublattice.basis()
[N(1, 0, 1), N(0, 1, -1)]
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(3))
>>> sublattice = N.submodule([(Integer(1),Integer(1),Integer(0)), (Integer(3),
↪Integer(2),Integer(1))])
>>> sublattice.has_user_basis()
False
>>> sublattice.basis()
[N(1, 0, 1), N(0, 1, -1)]
```

For sublattices without user-specified basis, the basis obtained above is the same as the "standard" one:

```
sage: sublattice.echelonized_basis()
[N(1, 0, 1), N(0, 1, -1)]
```

```
>>> from sage.all import *
>>> sublattice.echelonized_basis()
[N(1, 0, 1), N(0, 1, -1)]
```

class sage.geometry.toric_lattice.**ToricLattice_sublattice_with_basis**(*ambient*, *basis*, *check=True*, *echelonize=False*, *echelonized_basis=None*, *already_echelonized=False*, *category=None*)

Bases: *ToricLattice_generic*, FreeModule_submodule_with_basis_pid

Construct the sublattice of ambient toric lattice with given basis.

INPUT (same as for FreeModule_submodule_with_basis_pid):

- ambient – ambient *toric lattice* for this sublattice
- basis – list of linearly independent elements of ambient, these elements will be used as the default basis of the constructed sublattice
- see the base class for other available options

OUTPUT: sublattice of a toric lattice with a user-specified basis

See also *ToricLattice_sublattice* if you do not want to specify an explicit basis.

EXAMPLES:

The intended way to get objects of this class is to use submodule_with_basis() method of toric lattices:

```
sage: N = ToricLattice(3)
sage: sublattice = N.submodule_with_basis([(1,1,0), (3,2,1)])
sage: sublattice.has_user_basis()
True
sage: sublattice.basis()
[N(1, 1, 0), N(3, 2, 1)]
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(3))
>>> sublattice = N.submodule_with_basis([(Integer(1),Integer(1),Integer(0)),␣
↪(Integer(3),Integer(2),Integer(1))])
>>> sublattice.has_user_basis()
True
>>> sublattice.basis()
[N(1, 1, 0), N(3, 2, 1)]
```

Even if you have provided your own basis, you still can access the "standard" one:

```
sage: sublattice.echelonized_basis()
[N(1, 0, 1), N(0, 1, -1)]
```

```
>>> from sage.all import *
>>> sublattice.echelonized_basis()
[N(1, 0, 1), N(0, 1, -1)]
```

dual()

Return the lattice dual to `self`.

OUTPUT: a *toric lattice quotient*

EXAMPLES:

```
sage: N = ToricLattice(3)
sage: Ns = N.submodule([(1,1,0), (3,2,1)])
sage: Ns.dual()
2-d lattice, quotient of 3-d lattice M by Sublattice <M(1, -1, -1)>
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(3))
>>> Ns = N.submodule([(Integer(1),Integer(1),Integer(0)), (Integer(3),
↪Integer(2),Integer(1))])
>>> Ns.dual()
2-d lattice, quotient of 3-d lattice M by Sublattice <M(1, -1, -1)>
```

plot(***options*)

Plot `self`.

INPUT:

- any options for toric plots (see *toric_plotter.options*), none are mandatory.

OUTPUT: a plot

EXAMPLES:

```
sage: N = ToricLattice(3)
sage: sublattice = N.submodule_with_basis([(1,1,0), (3,2,1)])
sage: sublattice.plot()                                                     #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(3))
>>> sublattice = N.submodule_with_basis([(Integer(1),Integer(1),Integer(0)),␣
↪(Integer(3),Integer(2),Integer(1))])
>>> sublattice.plot()                                                      #␣
↪needs sage.plot
Graphics3d Object
```

Now we plot both the ambient lattice and its sublattice:

```
sage: N.plot() + sublattice.plot(point_color='red')                         #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> N.plot() + sublattice.plot(point_color='red')                          #␣
↪needs sage.plot
Graphics3d Object
```

sage.geometry.toric_lattice.**is_ToricLattice**(*x*)

Check if x is a toric lattice.

INPUT:

- x – anything

OUTPUT: True if x is a toric lattice and False otherwise

EXAMPLES:

```
sage: from sage.geometry.toric_lattice import (
....:   is_ToricLattice)
sage: is_ToricLattice(1)
doctest:warning...
DeprecationWarning: The function is_ToricLattice is deprecated;
use 'isinstance(..., ToricLattice_generic)' instead.
See https://github.com/sagemath/sage/issues/38126 for details.
False
sage: N = ToricLattice(3)
sage: N
3-d lattice N
sage: is_ToricLattice(N)
True
```

```
>>> from sage.all import *
>>> from sage.geometry.toric_lattice import (
...   is_ToricLattice)
>>> is_ToricLattice(Integer(1))
doctest:warning...
DeprecationWarning: The function is_ToricLattice is deprecated;
use 'isinstance(..., ToricLattice_generic)' instead.
See https://github.com/sagemath/sage/issues/38126 for details.
False
>>> N = ToricLattice(Integer(3))
>>> N
3-d lattice N
>>> is_ToricLattice(N)
True
```

sage.geometry.toric_lattice.**is_ToricLatticeQuotient**(*x*)

Check if x is a toric lattice quotient.

INPUT:

- x – anything

OUTPUT: True if x is a toric lattice quotient and False otherwise

EXAMPLES:

```
sage: from sage.geometry.toric_lattice import (
....:   is_ToricLatticeQuotient)
sage: is_ToricLatticeQuotient(1)
doctest:warning...
DeprecationWarning: The function is_ToricLatticeQuotient is deprecated;
use 'isinstance(..., ToricLattice_quotient)' instead.
```

(continues on next page)

(continued from previous page)

```
See https://github.com/sagemath/sage/issues/38126 for details.
False
sage: N = ToricLattice(3)
sage: N
3-d lattice N
sage: is_ToricLatticeQuotient(N)
False
sage: Q = N / N.submodule([(1,2,3), (3,2,1)])
sage: Q
Quotient with torsion of 3-d lattice N
by Sublattice <N(1, 2, 3), N(0, 4, 8)>
sage: is_ToricLatticeQuotient(Q)
True
```

```
>>> from sage.all import *
>>> from sage.geometry.toric_lattice import (
...   is_ToricLatticeQuotient)
>>> is_ToricLatticeQuotient(Integer(1))
doctest:warning...
DeprecationWarning: The function is_ToricLatticeQuotient is deprecated;
use 'isinstance(..., ToricLattice_quotient)' instead.
See https://github.com/sagemath/sage/issues/38126 for details.
False
>>> N = ToricLattice(Integer(3))
>>> N
3-d lattice N
>>> is_ToricLatticeQuotient(N)
False
>>> Q = N / N.submodule([(Integer(1),Integer(2),Integer(3)), (Integer(3),
↪Integer(2),Integer(1))])
>>> Q
Quotient with torsion of 3-d lattice N
by Sublattice <N(1, 2, 3), N(0, 4, 8)>
>>> is_ToricLatticeQuotient(Q)
True
```

2.5.2 Convex rational polyhedral cones

This module was designed as a part of framework for toric varieties (`variety`, `fano_variety`). While the emphasis is on strictly convex cones, non-strictly convex cones are supported as well. Work with distinct lattices (in the sense of discrete subgroups spanning vector spaces) is supported. The default lattice is `ToricLattice` N of the appropriate dimension. The only case when you must specify lattice explicitly is creation of a 0-dimensional cone, where dimension of the ambient space cannot be guessed.

AUTHORS:

- Andrey Novoseltsev (2010-05-13): initial version.
- Andrey Novoseltsev (2010-06-17): substantial improvement during review by Volker Braun.
- Volker Braun (2010-06-21): various spanned/quotient/dual lattice computations added.
- Volker Braun (2010-12-28): Hilbert basis for cones.
- Andrey Novoseltsev (2012-02-23): switch to PointCollection container.

EXAMPLES:

Use `Cone()` to construct cones:

```
sage: octant = Cone([(1,0,0), (0,1,0), (0,0,1)])
sage: halfspace = Cone([(1,0,0), (0,1,0), (-1,-1,0), (0,0,1)])
sage: positive_xy = Cone([(1,0,0), (0,1,0)])
sage: four_rays = Cone([(1,1,1), (1,-1,1), (-1,-1,1), (-1,1,1)])
```

```
>>> from sage.all import *
>>> octant = Cone([(Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0)), (Integer(0),Integer(0),Integer(1))])
>>> halfspace = Cone([(Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0)), (-Integer(1),-Integer(1),Integer(0)), (Integer(0),Integer(0),
↪Integer(1))])
>>> positive_xy = Cone([(Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0))])
>>> four_rays = Cone([(Integer(1),Integer(1),Integer(1)), (Integer(1),-Integer(1),
↪Integer(1)), (-Integer(1),-Integer(1),Integer(1)), (-Integer(1),Integer(1),
↪Integer(1))])
```

For all of the cones above we have provided primitive generating rays, but in fact this is not necessary - a cone can be constructed from any collection of rays (from the same space, of course). If there are non-primitive (or even non-integral) rays, they will be replaced with primitive ones. If there are extra rays, they will be discarded. Of course, this means that `Cone()` has to do some work before actually constructing the cone and sometimes it is not desirable, if you know for sure that your input is already "good". In this case you can use options `check=False` to force `Cone()` to use exactly the directions that you have specified and `normalize=False` to force it to use exactly the rays that you have specified. However, it is better not to use these possibilities without necessity, since cones are assumed to be represented by a minimal set of primitive generating rays. See `Cone()` for further documentation on construction.

Once you have a cone, you can perform numerous operations on it. The most important ones are, probably, ray accessing methods:

```
sage: rays = halfspace.rays()
sage: rays
N( 0,  0, 1),
N( 0,  1, 0),
N( 0, -1, 0),
N( 1,  0, 0),
N(-1,  0, 0)
in 3-d lattice N
sage: rays.set()
frozenset({N(-1, 0, 0), N(0, -1, 0), N(0, 0, 1), N(0, 1, 0), N(1, 0, 0)})
sage: rays.matrix()
[ 0  0  1]
[ 0  1  0]
[ 0 -1  0]
[ 1  0  0]
[-1  0  0]
sage: rays.column_matrix()
[ 0  0  0  1 -1]
[ 0  1 -1  0  0]
[ 1  0  0  0  0]
sage: rays(3)
```

(continues on next page)

(continued from previous page)

```
N(1, 0, 0)
in 3-d lattice N
sage: rays[3]
N(1, 0, 0)
sage: halfspace.ray(3)
N(1, 0, 0)
```

```
>>> from sage.all import *
>>> rays = halfspace.rays()
>>> rays
N( 0,  0, 1),
N( 0,  1, 0),
N( 0, -1, 0),
N( 1,  0, 0),
N(-1,  0, 0)
in 3-d lattice N
>>> rays.set()
frozenset({N(-1, 0, 0), N(0, -1, 0), N(0, 0, 1), N(0, 1, 0), N(1, 0, 0)})
>>> rays.matrix()
[ 0  0  1]
[ 0  1  0]
[ 0 -1  0]
[ 1  0  0]
[-1  0  0]
>>> rays.column_matrix()
[ 0  0  0  1 -1]
[ 0  1 -1  0  0]
[ 1  0  0  0  0]
>>> rays(Integer(3))
N(1, 0, 0)
in 3-d lattice N
>>> rays[Integer(3)]
N(1, 0, 0)
>>> halfspace.ray(Integer(3))
N(1, 0, 0)
```

The method *rays()* returns a *PointCollection* with the i-th element being the primitive integral generator of the i-th ray. It is possible to convert this collection to a matrix with either rows or columns corresponding to these generators. You may also change the default *output_format()* of all point collections to be such a matrix.

If you want to do something with each ray of a cone, you can write

```
sage: for ray in positive_xy: print(ray)
N(1, 0, 0)
N(0, 1, 0)
```

```
>>> from sage.all import *
>>> for ray in positive_xy: print(ray)
N(1, 0, 0)
N(0, 1, 0)
```

There are two dimensions associated to each cone - the dimension of the subspace spanned by the cone and the dimension of the space where it lives:

```
sage: positive_xy.dim()
2
sage: positive_xy.lattice_dim()
3
```

```
>>> from sage.all import *
>>> positive_xy.dim()
2
>>> positive_xy.lattice_dim()
3
```

You also may be interested in this dimension:

```
sage: dim(positive_xy.linear_subspace())
0
sage: dim(halfspace.linear_subspace())
2
```

```
>>> from sage.all import *
>>> dim(positive_xy.linear_subspace())
0
>>> dim(halfspace.linear_subspace())
2
```

Or, perhaps, all you care about is whether it is zero or not:

```
sage: positive_xy.is_strictly_convex()
True
sage: halfspace.is_strictly_convex()
False
```

```
>>> from sage.all import *
>>> positive_xy.is_strictly_convex()
True
>>> halfspace.is_strictly_convex()
False
```

You can also perform these checks:

```
sage: positive_xy.is_simplicial()
True
sage: four_rays.is_simplicial()
False
sage: positive_xy.is_smooth()
True
```

```
>>> from sage.all import *
>>> positive_xy.is_simplicial()
True
>>> four_rays.is_simplicial()
False
>>> positive_xy.is_smooth()
True
```

You can work with subcones that form faces of other cones:

```
sage: # needs sage.graphs
sage: face = four_rays.faces(dim=2)[0]
sage: face
2-d face of 3-d cone in 3-d lattice N
sage: face.rays()
N(-1, -1, 1),
N(-1,  1, 1)
in 3-d lattice N
sage: face.ambient_ray_indices()
(2, 3)
sage: four_rays.rays(face.ambient_ray_indices())
N(-1, -1, 1),
N(-1,  1, 1)
in 3-d lattice N
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> face = four_rays.faces(dim=Integer(2))[Integer(0)]
>>> face
2-d face of 3-d cone in 3-d lattice N
>>> face.rays()
N(-1, -1, 1),
N(-1,  1, 1)
in 3-d lattice N
>>> face.ambient_ray_indices()
(2, 3)
>>> four_rays.rays(face.ambient_ray_indices())
N(-1, -1, 1),
N(-1,  1, 1)
in 3-d lattice N
```

If you need to know inclusion relations between faces, you can use

```
sage: # needs sage.graphs
sage: L = four_rays.face_lattice()
sage: [len(s) for s in L.level_sets()]
[1, 4, 4, 1]
sage: face = L.level_sets()[2][0]
sage: face.rays()
N(1,  1, 1),
N(1, -1, 1)
in 3-d lattice N
sage: L.hasse_diagram().neighbors_in(face)
[1-d face of 3-d cone in 3-d lattice N,
 1-d face of 3-d cone in 3-d lattice N]
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> L = four_rays.face_lattice()
>>> [len(s) for s in L.level_sets()]
[1, 4, 4, 1]
```

(continues on next page)

(continued from previous page)

```
>>> face = L.level_sets()[Integer(2)][Integer(0)]
>>> face.rays()
N(1,  1, 1),
N(1, -1, 1)
in 3-d lattice N
>>> L.hasse_diagram().neighbors_in(face)
[1-d face of 3-d cone in 3-d lattice N,
 1-d face of 3-d cone in 3-d lattice N]
```

Warning

The order of faces in level sets of the face lattice may differ from the order of faces returned by `faces()`. While the first order is random, the latter one ensures that one-dimensional faces are listed in the same order as generating rays.

When all the functionality provided by cones is not enough, you may want to check if you can do necessary things using polyhedra corresponding to cones:

```
sage: four_rays.polyhedron()
A 3-dimensional polyhedron in ZZ^3 defined as
the convex hull of 1 vertex and 4 rays
```

```
>>> from sage.all import *
>>> four_rays.polyhedron()
A 3-dimensional polyhedron in ZZ^3 defined as
the convex hull of 1 vertex and 4 rays
```

And of course you are always welcome to suggest new features that should be added to cones!

REFERENCES:

- [Ful1993]

`sage.geometry.cone.`**`Cone`**(*rays*, *lattice=None*, *check=True*, *normalize=True*)

Construct a (not necessarily strictly) convex rational polyhedral cone.

INPUT:

- `rays` – list of rays; each ray should be given as a list or a vector convertible to the rational extension of the given `lattice`. May also be specified by a `Polyhedron_base` object;
- `lattice` – `ToricLattice`, $\mathbf{Z}^n$, or any other object that behaves like these. If not specified, an attempt will be made to determine an appropriate toric lattice automatically;
- `check` – by default the input data will be checked for correctness (e.g. that all rays have the same number of components) and generating rays will be constructed from `rays`. If you know that the input is a minimal set of generators of a valid cone, you may significantly decrease construction time using `check=False` option;
- `normalize` – you can further speed up construction using `normalize=False` option. In this case `rays` must be a list of immutable primitive rays in `lattice`. In general, you should not use this option, it is designed for code optimization and does not give as drastic improvement in speed as the previous one.

OUTPUT: convex rational polyhedral cone determined by `rays`

EXAMPLES:

Let's define a cone corresponding to the first quadrant of the plane (note, you can even mix objects of different types to represent rays, as long as you let this function to perform all the checks and necessary conversions!):

```
sage: quadrant = Cone([(1,0), [0,1]])
sage: quadrant
2-d cone in 2-d lattice N
sage: quadrant.rays()
N(1, 0),
N(0, 1)
in 2-d lattice N
```

```
>>> from sage.all import *
>>> quadrant = Cone([(Integer(1),Integer(0)), [Integer(0),Integer(1)]])
>>> quadrant
2-d cone in 2-d lattice N
>>> quadrant.rays()
N(1, 0),
N(0, 1)
in 2-d lattice N
```

If you give more rays than necessary, the extra ones will be discarded:

```
sage: Cone([(1,0), (0,1), (1,1), (0,1)]).rays()
N(0, 1),
N(1, 0)
in 2-d lattice N
```

```
>>> from sage.all import *
>>> Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1)), (Integer(1),
↪Integer(1)), (Integer(0),Integer(1))]).rays()
N(0, 1),
N(1, 0)
in 2-d lattice N
```

However, this work is not done with `check=False` option, so use it carefully!

```
sage: Cone([(1,0), (0,1), (1,1), (0,1)], check=False).rays()
N(1, 0),
N(0, 1),
N(1, 1),
N(0, 1)
in 2-d lattice N
```

```
>>> from sage.all import *
>>> Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1)), (Integer(1),
↪Integer(1)), (Integer(0),Integer(1))], check=False).rays()
N(1, 0),
N(0, 1),
N(1, 1),
N(0, 1)
in 2-d lattice N
```

Even worse things can happen with `normalize=False` option:

```
sage: Cone([(1,0), (0,1)], check=False, normalize=False)
Traceback (most recent call last):
...
AttributeError: 'tuple' object has no attribute 'parent'...
```

```
>>> from sage.all import *
>>> Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))], check=False,␣
↪normalize=False)
Traceback (most recent call last):
...
AttributeError: 'tuple' object has no attribute 'parent'...
```

You can construct different "not" cones: not full-dimensional, not strictly convex, not containing any rays:

```
sage: one_dimensional_cone = Cone([(1,0)])
sage: one_dimensional_cone.dim()
1
sage: half_plane = Cone([(1,0), (0,1), (-1,0)])
sage: half_plane.rays()
N( 0, 1),
N( 1, 0),
N(-1, 0)
in 2-d lattice N
sage: half_plane.is_strictly_convex()
False
sage: origin = Cone([(0,0)])
sage: origin.rays()
Empty collection
in 2-d lattice N
sage: origin.dim()
0
sage: origin.lattice_dim()
2
```

```
>>> from sage.all import *
>>> one_dimensional_cone = Cone([(Integer(1),Integer(0))])
>>> one_dimensional_cone.dim()
1
>>> half_plane = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1)), (-
↪Integer(1),Integer(0))])
>>> half_plane.rays()
N( 0, 1),
N( 1, 0),
N(-1, 0)
in 2-d lattice N
>>> half_plane.is_strictly_convex()
False
>>> origin = Cone([(Integer(0),Integer(0))])
>>> origin.rays()
Empty collection
in 2-d lattice N
>>> origin.dim()
```

(continues on next page)

(continued from previous page)

```
0
>>> origin.lattice_dim()
2
```

You may construct the cone above without giving any rays, but in this case you must provide `lattice` explicitly:

```
sage: origin = Cone([])
Traceback (most recent call last):
...
ValueError: lattice must be given explicitly if there are no rays!
sage: origin = Cone([], lattice=ToricLattice(2))
sage: origin.dim()
0
sage: origin.lattice_dim()
2
sage: origin.lattice()
2-d lattice N
```

```
>>> from sage.all import *
>>> origin = Cone([])
Traceback (most recent call last):
...
ValueError: lattice must be given explicitly if there are no rays!
>>> origin = Cone([], lattice=ToricLattice(Integer(2)))
>>> origin.dim()
0
>>> origin.lattice_dim()
2
>>> origin.lattice()
2-d lattice N
```

However, the trivial cone in `n` dimensions has a predefined constructor for you to use:

```
sage: origin = cones.trivial(2)
sage: origin.rays()
Empty collection
in 2-d lattice N
```

```
>>> from sage.all import *
>>> origin = cones.trivial(Integer(2))
>>> origin.rays()
Empty collection
in 2-d lattice N
```

Of course, you can also provide `lattice` in other cases:

```
sage: L = ToricLattice(3, "L")
sage: c1 = Cone([(1,0,0),(1,1,1)], lattice=L)
sage: c1.rays()
L(1, 0, 0),
L(1, 1, 1)
in 3-d lattice L
```

```
>>> from sage.all import *
>>> L = ToricLattice(Integer(3), "L")
>>> c1 = Cone([(Integer(1),Integer(0),Integer(0)),(Integer(1),Integer(1),
↪Integer(1))], lattice=L)
>>> c1.rays()
L(1, 0, 0),
L(1, 1, 1)
in 3-d lattice L
```

Or you can construct cones from rays of a particular lattice:

```
sage: ray1 = L(1,0,0)
sage: ray2 = L(1,1,1)
sage: c2 = Cone([ray1, ray2])
sage: c2.rays()
L(1, 0, 0),
L(1, 1, 1)
in 3-d lattice L
sage: c1 == c2
True
```

```
>>> from sage.all import *
>>> ray1 = L(Integer(1),Integer(0),Integer(0))
>>> ray2 = L(Integer(1),Integer(1),Integer(1))
>>> c2 = Cone([ray1, ray2])
>>> c2.rays()
L(1, 0, 0),
L(1, 1, 1)
in 3-d lattice L
>>> c1 == c2
True
```

When the cone in question is not strictly convex, the standard form for the "generating rays" of the linear subspace is "basis vectors and their negatives", as in the following example:

```
sage: plane = Cone([(1,0), (0,1), (-1,-1)])
sage: plane.rays()
N( 0,  1),
N( 0, -1),
N( 1,  0),
N(-1,  0)
in 2-d lattice N
```

```
>>> from sage.all import *
>>> plane = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1)), (-Integer(1),-
↪Integer(1))])
>>> plane.rays()
N( 0,  1),
N( 0, -1),
N( 1,  0),
N(-1,  0)
in 2-d lattice N
```

The cone can also be specified by a *Polyhedron_base*:

```
sage: p = plane.polyhedron()
sage: Cone(p)
2-d cone in 2-d lattice N
sage: Cone(p) == plane
True
```

```
>>> from sage.all import *
>>> p = plane.polyhedron()
>>> Cone(p)
2-d cone in 2-d lattice N
>>> Cone(p) == plane
True
```

class sage.geometry.cone.**ConvexRationalPolyhedralCone**(*rays=None*, *lattice=None*, *ambient=None*, *ambient_ray_indices=None*, *PPL=None*)

Bases: *IntegralRayCollection*, Container, *ConvexSet_closed*, *ConvexRationalPolyhedralCone*

Create a convex rational polyhedral cone.

> **Warning**
>
> This class does not perform any checks of correctness of input nor does it convert input into the standard representation. Use *Cone()* to construct cones.

Cones are immutable, but they cache most of the returned values.

INPUT:

The input can be either:

- rays – list of immutable primitive vectors in lattice;
- lattice – *ToricLattice*, $\mathbf{Z}^n$, or any other object that behaves like these. If None, it will be determined as parent() of the first ray. Of course, this cannot be done if there are no rays, so in this case you must give an appropriate lattice directly.

or (these parameters must be given as keywords):

- ambient – ambient structure of this cone, a bigger *cone* or a *fan*, this cone *must be a face of* ambient;
- ambient_ray_indices – increasing list or tuple of integers, indices of rays of ambient generating this cone

In both cases, the following keyword parameter may be specified in addition:

- PPL – either None (default) or a C_Polyhedron representing the cone. This serves only to cache the polyhedral data if you know it already. The constructor does not make a copy so the PPL object should not be modified afterwards.

OUTPUT: convex rational polyhedral cone

> **Note**
>
> Every cone has its ambient structure. If it was not specified, it is this cone itself.

Hilbert_basis()

Return the Hilbert basis of the cone.

Given a strictly convex cone $C \subset \mathbf{R}^d$, the Hilbert basis of C is the set of all irreducible elements in the semigroup $C \cap \mathbf{Z}^d$. It is the unique minimal generating set over $\mathbf{Z}$ for the integral points $C \cap \mathbf{Z}^d$.

If the cone C is not strictly convex, this method finds the (unique) minimal set of lattice points that need to be added to the defining rays of the cone to generate the whole semigroup $C \cap \mathbf{Z}^d$. But because the rays of the cone are not unique nor necessarily minimal in this case, neither is the returned generating set (consisting of the rays plus additional generators).

See also *semigroup_generators()* if you are not interested in a minimal set of generators.

OUTPUT:

- a *PointCollection*. The rays of `self` are the first `self.nrays()` entries.

EXAMPLES:

The following command ensures that the output ordering in the examples below is independent of TOPCOM, you don't have to use it:

```
sage: PointConfiguration.set_engine('internal')
```

```
>>> from sage.all import *
>>> PointConfiguration.set_engine('internal')
```

We start with a simple case of a non-smooth 2-dimensional cone:

```
sage: Cone([ (1,0), (1,2) ]).Hilbert_basis()
N(1, 0),
N(1, 2),
N(1, 1)
in 2-d lattice N
```

```
>>> from sage.all import *
>>> Cone([ (Integer(1),Integer(0)), (Integer(1),Integer(2)) ]).Hilbert_basis()
N(1, 0),
N(1, 2),
N(1, 1)
in 2-d lattice N
```

Two more complicated example from GAP/toric:

```
sage: Cone([[1,0], [3,4]]).dual().Hilbert_basis()
M(0,  1),
M(4, -3),
M(1,  0),
M(2, -1),
M(3, -2)
in 2-d lattice M
sage: cone = Cone([[1,2,3,4], [0,1,0,7], [3,1,0,2], [0,0,1,0]]).dual()
sage: cone.Hilbert_basis()           # long time
M(10,  -7,  0,  1),
M(-5,  21,  0, -3),
M( 0,  -2,  0,  1),
```

(continues on next page)

(continued from previous page)

```
M(15, -63, 25,  9),
M( 2,  -3,  0,  1),
M( 1,  -4,  1,  1),
M( 4,  -4,  0,  1),
M(-1,   3,  0,  0),
M( 1,  -5,  2,  1),
M( 3,  -5,  1,  1),
M( 6,  -5,  0,  1),
M( 3, -13,  5,  2),
M( 2,  -6,  2,  1),
M( 5,  -6,  1,  1),
M( 8,  -6,  0,  1),
M( 0,   1,  0,  0),
M(-2,   8,  0, -1),
M(10, -42, 17,  6),
M( 7, -28, 11,  4),
M( 5, -21,  9,  3),
M( 6, -21,  8,  3),
M( 5, -14,  5,  2),
M( 2,  -7,  3,  1),
M( 4,  -7,  2,  1),
M( 7,  -7,  1,  1),
M( 0,   0,  1,  0),
M( 1,   0,  0,  0),
M(-1,   7,  0, -1),
M(-3,  14,  0, -2)
in 4-d lattice M
```

```
>>> from sage.all import *
>>> Cone([[Integer(1),Integer(0)], [Integer(3),Integer(4)]]).dual().Hilbert_
↪basis()
M(0,  1),
M(4, -3),
M(1,  0),
M(2, -1),
M(3, -2)
in 2-d lattice M
>>> cone = Cone([[Integer(1),Integer(2),Integer(3),Integer(4)], [Integer(0),
↪Integer(1),Integer(0),Integer(7)], [Integer(3),Integer(1),Integer(0),
↪Integer(2)], [Integer(0),Integer(0),Integer(1),Integer(0)]]).dual()
>>> cone.Hilbert_basis()           # long time
M(10,  -7,  0,  1),
M(-5,  21,  0, -3),
M( 0,  -2,  0,  1),
M(15, -63, 25,  9),
M( 2,  -3,  0,  1),
M( 1,  -4,  1,  1),
M( 4,  -4,  0,  1),
M(-1,   3,  0,  0),
M( 1,  -5,  2,  1),
M( 3,  -5,  1,  1),
M( 6,  -5,  0,  1),
```

(continues on next page)

(continued from previous page)

```
M( 3, -13,  5,  2),
M( 2,  -6,  2,  1),
M( 5,  -6,  1,  1),
M( 8,  -6,  0,  1),
M( 0,   1,  0,  0),
M(-2,   8,  0, -1),
M(10, -42, 17,  6),
M( 7, -28, 11,  4),
M( 5, -21,  9,  3),
M( 6, -21,  8,  3),
M( 5, -14,  5,  2),
M( 2,  -7,  3,  1),
M( 4,  -7,  2,  1),
M( 7,  -7,  1,  1),
M( 0,   0,  1,  0),
M( 1,   0,  0,  0),
M(-1,   7,  0, -1),
M(-3,  14,  0, -2)
in 4-d lattice M
```

Not a strictly convex cone:

```
sage: wedge = Cone([(1,0,0), (1,2,0), (0,0,1), (0,0,-1)])
sage: sorted(wedge.semigroup_generators())
[N(0, 0, -1), N(0, 0, 1), N(1, 0, 0), N(1, 1, 0), N(1, 2, 0)]
sage: wedge.Hilbert_basis()
N(1, 2,  0),
N(1, 0,  0),
N(0, 0,  1),
N(0, 0, -1),
N(1, 1,  0)
in 3-d lattice N
```

```
>>> from sage.all import *
>>> wedge = Cone([(Integer(1),Integer(0),Integer(0)), (Integer(1),Integer(2),
↪Integer(0)), (Integer(0),Integer(0),Integer(1)), (Integer(0),Integer(0),-
↪Integer(1))])
>>> sorted(wedge.semigroup_generators())
[N(0, 0, -1), N(0, 0, 1), N(1, 0, 0), N(1, 1, 0), N(1, 2, 0)]
>>> wedge.Hilbert_basis()
N(1, 2,  0),
N(1, 0,  0),
N(0, 0,  1),
N(0, 0, -1),
N(1, 1,  0)
in 3-d lattice N
```

Not full-dimensional cones are ok, too (see Issue #11312):

```
sage: Cone([(1,1,0), (-1,1,0)]).Hilbert_basis()
N( 1, 1, 0),
N(-1, 1, 0),
```

(continues on next page)

(continued from previous page)

```
N( 0, 1, 0)
in 3-d lattice N
```

```
>>> from sage.all import *
>>> Cone([(Integer(1),Integer(1),Integer(0)), (-Integer(1),Integer(1),
↪Integer(0))]).Hilbert_basis()
N( 1, 1, 0),
N(-1, 1, 0),
N( 0, 1, 0)
in 3-d lattice N
```

ALGORITHM:

The primal Normaliz algorithm, see [Normaliz].

Hilbert_coefficients(*point*, *solver*, *verbose=None*, *integrality_tolerance=0*)

Return the expansion coefficients of `point` with respect to `Hilbert_basis()`.

INPUT:

- `point` – a `lattice()` point in the cone, or something that can be converted to a point. For example, a list or tuple of integers.
- `solver` – (default: `None`) specify a Mixed Integer Linear Programming (MILP) solver to be used. If set to `None`, the default one is used. For more information on MILP solvers and which default solver is used, see the method `solve` of the class `MixedIntegerLinearProgram`.
- `verbose` – integer (default: 0); sets the level of verbosity of the LP solver. Set to 0 by default, which means quiet.
- `integrality_tolerance` – parameter for use with MILP solvers over an inexact base ring; see `MixedIntegerLinearProgram.get_values()`

OUTPUT:

A $\mathbf{Z}$-vector of length `len(self.Hilbert_basis())` with nonnegative components.

Note

Since the Hilbert basis elements are not necessarily linearly independent, the expansion coefficients are not unique. However, this method will always return the same expansion coefficients when invoked with the same argument.

EXAMPLES:

```
sage: cone = Cone([(1,0), (0,1)])
sage: cone.rays()
N(1, 0),
N(0, 1)
in 2-d lattice N
sage: cone.Hilbert_coefficients([3,2])
(3, 2)
```

```
>>> from sage.all import *
>>> cone = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
```

(continues on next page)

(continued from previous page)

```
>>> cone.rays()
N(1, 0),
N(0, 1)
in 2-d lattice N
>>> cone.Hilbert_coefficients([Integer(3),Integer(2)])
(3, 2)
```

A more complicated example:

```
sage: N = ToricLattice(2)
sage: cone = Cone([N(1,0), N(1,2)])
sage: cone.Hilbert_basis()
N(1, 0),
N(1, 2),
N(1, 1)
in 2-d lattice N
sage: cone.Hilbert_coefficients(N(1,1))
(0, 0, 1)
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(2))
>>> cone = Cone([N(Integer(1),Integer(0)), N(Integer(1),Integer(2))])
>>> cone.Hilbert_basis()
N(1, 0),
N(1, 2),
N(1, 1)
in 2-d lattice N
>>> cone.Hilbert_coefficients(N(Integer(1),Integer(1)))
(0, 0, 1)
```

The cone need not be strictly convex:

```
sage: N = ToricLattice(3)
sage: cone = Cone([N(1,0,0), N(1,2,0), N(0,0,1), N(0,0,-1)])
sage: cone.Hilbert_basis()
N(1, 2,  0),
N(1, 0,  0),
N(0, 0,  1),
N(0, 0, -1),
N(1, 1,  0)
in 3-d lattice N
sage: cone.Hilbert_coefficients(N(1,1,3))
(0, 0, 3, 0, 1)
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(3))
>>> cone = Cone([N(Integer(1),Integer(0),Integer(0)), N(Integer(1),Integer(2),
↪Integer(0)), N(Integer(0),Integer(0),Integer(1)), N(Integer(0),Integer(0),-
↪Integer(1))])
>>> cone.Hilbert_basis()
N(1, 2,  0),
N(1, 0,  0),
```

(continues on next page)

(continued from previous page)

```
N(0, 0,  1),
N(0, 0, -1),
N(1, 1,  0)
in 3-d lattice N
>>> cone.Hilbert_coefficients(N(Integer(1),Integer(1),Integer(3)))
(0, 0, 3, 0, 1)
```

Z_operators_gens()

Compute minimal generators of the Z-operators on this cone.

The Z-operators on a cone generalize the Z-matrices over the nonnegative orthant. They are simply negations of the *cross_positive_operators_gens()*.

OUTPUT:

A list of n-by-n matrices where n is the ambient dimension of this cone. Each matrix L in the list has the property that $s(L(x)) \leq 0$ whenever (x, s) is an element of this cone's *discrete_complementarity_set()*.

The returned matrices generate the cone of Z-operators on this cone; that is,

- Any nonnegative linear combination of the returned matrices is a Z-operator on this cone.
- Every Z-operator on this cone is some nonnegative linear combination of the returned matrices.

See also

cross_positive_operators_gens(), *lyapunov_like_basis()*, *positive_operators_gens()*

REFERENCES:

- [BP1994]
- [Or2018b]

adjacent()

Return faces adjacent to `self` in the ambient face lattice.

Two *distinct* faces F_1 and F_2 of the same face lattice are **adjacent** if all of the following conditions hold:

- F_1 and F_2 have the same dimension d;
- F_1 and F_2 share a facet of dimension $d - 1$;
- F_1 and F_2 are facets of some face of dimension $d + 1$, unless d is the dimension of the ambient structure.

OUTPUT: `tuple` of *cones*

EXAMPLES:

```
sage: # needs sage.graphs
sage: octant = Cone([(1,0,0), (0,1,0), (0,0,1)])
sage: octant.adjacent()
()
sage: one_face = octant.faces(1)[0]
sage: len(one_face.adjacent())
2
```

(continues on next page)

(continued from previous page)

```
sage: one_face.adjacent()[1]
1-d face of 3-d cone in 3-d lattice N
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> octant = Cone([(Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0)), (Integer(0),Integer(0),Integer(1))])
>>> octant.adjacent()
()
>>> one_face = octant.faces(Integer(1))[Integer(0)]
>>> len(one_face.adjacent())
2
>>> one_face.adjacent()[Integer(1)]
1-d face of 3-d cone in 3-d lattice N
```

Things are a little bit subtle with fans, as we illustrate below.

First, we create a fan from two cones in the plane:

```
sage: fan = Fan(cones=[(0,1), (1,2)],
....:           rays=[(1,0), (0,1), (-1,0)])
sage: cone = fan.generating_cone(0)
sage: len(cone.adjacent())                                              #␣
↪needs sage.graphs
1
```

```
>>> from sage.all import *
>>> fan = Fan(cones=[(Integer(0),Integer(1)), (Integer(1),Integer(2))],
...           rays=[(Integer(1),Integer(0)), (Integer(0),Integer(1)), (-
↪Integer(1),Integer(0))])
>>> cone = fan.generating_cone(Integer(0))
>>> len(cone.adjacent())                                                #␣
↪needs sage.graphs
1
```

The second generating cone is adjacent to this one. Now we create the same fan, but embedded into the 3-dimensional space:

```
sage: fan = Fan(cones=[(0,1), (1,2)],
....:           rays=[(1,0,0), (0,1,0), (-1,0,0)])
sage: cone = fan.generating_cone(0)
sage: len(cone.adjacent())                                              #␣
↪needs sage.graphs
1
```

```
>>> from sage.all import *
>>> fan = Fan(cones=[(Integer(0),Integer(1)), (Integer(1),Integer(2))],
...           rays=[(Integer(1),Integer(0),Integer(0)), (Integer(0),
↪Integer(1),Integer(0)), (-Integer(1),Integer(0),Integer(0))])
>>> cone = fan.generating_cone(Integer(0))
>>> len(cone.adjacent())                                                #␣
↪needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
1
```

The result is as before, since we still have:

```
sage: fan.dim()
2
```

```
>>> from sage.all import *
>>> fan.dim()
2
```

Now we add another cone to make the fan 3-dimensional:

```
sage: fan = Fan(cones=[(0,1), (1,2), (3,)],
....:           rays=[(1,0,0), (0,1,0), (-1,0,0), (0,0,1)])
sage: cone = fan.generating_cone(0)
sage: len(cone.adjacent())                                                  #␣
↪needs sage.graphs
0
```

```
>>> from sage.all import *
>>> fan = Fan(cones=[(Integer(0),Integer(1)), (Integer(1),Integer(2)),␣
↪(Integer(3),)],
...           rays=[(Integer(1),Integer(0),Integer(0)), (Integer(0),
↪Integer(1),Integer(0)), (-Integer(1),Integer(0),Integer(0)), (Integer(0),
↪Integer(0),Integer(1))])
>>> cone = fan.generating_cone(Integer(0))
>>> len(cone.adjacent())                                                    #␣
↪needs sage.graphs
0
```

Since now `cone` has smaller dimension than `fan`, it and its adjacent cones must be facets of a bigger one, but since `cone` in this example is generating, it is not contained in any other.

ambient()

Return the ambient structure of `self`.

OUTPUT: cone or fan containing `self` as a face

EXAMPLES:

```
sage: cone = Cone([(1,2,3), (4,6,5), (9,8,7)])
sage: cone.ambient()
3-d cone in 3-d lattice N
sage: cone.ambient() is cone
True

sage: # needs sage.graphs
sage: face = cone.faces(1)[0]
sage: face
1-d face of 3-d cone in 3-d lattice N
sage: face.ambient()
3-d cone in 3-d lattice N
```

(continues on next page)

(continued from previous page)

```
sage: face.ambient() is cone
True
```

```
>>> from sage.all import *
>>> cone = Cone([(Integer(1),Integer(2),Integer(3)), (Integer(4),Integer(6),
↪Integer(5)), (Integer(9),Integer(8),Integer(7))])
>>> cone.ambient()
3-d cone in 3-d lattice N
>>> cone.ambient() is cone
True

>>> # needs sage.graphs
>>> face = cone.faces(Integer(1))[Integer(0)]
>>> face
1-d face of 3-d cone in 3-d lattice N
>>> face.ambient()
3-d cone in 3-d lattice N
>>> face.ambient() is cone
True
```

ambient_ray_indices()

Return indices of rays of the ambient structure generating `self`.

OUTPUT: increasing `tuple` of integers

EXAMPLES:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: quadrant.ambient_ray_indices()
(0, 1)
sage: quadrant.facets()[1].ambient_ray_indices()                              #␣
↪needs sage.graphs
(1,)
```

```
>>> from sage.all import *
>>> quadrant = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> quadrant.ambient_ray_indices()
(0, 1)
>>> quadrant.facets()[Integer(1)].ambient_ray_indices()                       ␣
↪     # needs sage.graphs
(1,)
```

an_affine_basis()

Return points in `self` that form a basis for the affine hull.

EXAMPLES:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: quadrant.an_affine_basis()
[(0, 0), (1, 0), (0, 1)]
sage: ray = Cone([(1, 1)])
sage: ray.an_affine_basis()
[(0, 0), (1, 1)]
```

(continues on next page)

(continued from previous page)

```
sage: line = Cone([(1,0), (-1,0)])
sage: line.an_affine_basis()
[(1, 0), (0, 0)]
```

```
>>> from sage.all import *
>>> quadrant = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> quadrant.an_affine_basis()
[(0, 0), (1, 0), (0, 1)]
>>> ray = Cone([(Integer(1), Integer(1))])
>>> ray.an_affine_basis()
[(0, 0), (1, 1)]
>>> line = Cone([(Integer(1),Integer(0)), (-Integer(1),Integer(0))])
>>> line.an_affine_basis()
[(1, 0), (0, 0)]
```

cartesian_product(*other*, *lattice=None*)

Return the Cartesian product of `self` with `other`.

INPUT:

- `other` – a *cone*
- `lattice` – (optional) the ambient lattice for the Cartesian product cone; by default, the direct sum of the ambient lattices of `self` and `other` is constructed

OUTPUT: a *cone*

EXAMPLES:

```
sage: c = Cone([(1,)])
sage: c.cartesian_product(c)
2-d cone in 2-d lattice N+N
sage: _.rays()
N+N(1, 0),
N+N(0, 1)
in 2-d lattice N+N
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(1),)])
>>> c.cartesian_product(c)
2-d cone in 2-d lattice N+N
>>> _.rays()
N+N(1, 0),
N+N(0, 1)
in 2-d lattice N+N
```

contains(**args*)

Check if a given point is contained in `self`.

INPUT:

- anything. An attempt will be made to convert all arguments into a single element of the ambient space of `self`. If it fails, `False` will be returned.

OUTPUT:

- `True` if the given point is contained in `self`, `False` otherwise.

EXAMPLES:

```
sage: c = Cone([(1,0), (0,1)])
sage: c.contains(c.lattice()(1,0))
True
sage: c.contains((1,0))
True
sage: c.contains((1,1))
True
sage: c.contains(1,1)
True
sage: c.contains((-1,0))
False
sage: c.contains(c.dual_lattice()(1,0))  # random output (warning)
False
sage: c.contains(c.dual_lattice()(1,0))
False
sage: c.contains(1)
False
sage: c.contains(1/2, sqrt(3))                                              #␣
↪needs sage.symbolic
True
sage: c.contains(-1/2, sqrt(3))                                             #␣
↪needs sage.symbolic
False
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> c.contains(c.lattice()(Integer(1),Integer(0)))
True
>>> c.contains((Integer(1),Integer(0)))
True
>>> c.contains((Integer(1),Integer(1)))
True
>>> c.contains(Integer(1),Integer(1))
True
>>> c.contains((-Integer(1),Integer(0)))
False
>>> c.contains(c.dual_lattice()(Integer(1),Integer(0)))  # random output␣
↪(warning)
False
>>> c.contains(c.dual_lattice()(Integer(1),Integer(0)))
False
>>> c.contains(Integer(1))
False
>>> c.contains(Integer(1)/Integer(2), sqrt(Integer(3)))                    ␣
↪                        # needs sage.symbolic
True
>>> c.contains(-Integer(1)/Integer(2), sqrt(Integer(3)))                   ␣
↪                        # needs sage.symbolic
False
```

cross_positive_operators_gens()

Compute minimal generators of the cross-positive operators on this cone.

Any positive operator P on this cone will have $s(P(x)) \geq 0$ whenever x is an element of this cone and s is an element of its dual. By contrast, the cross-positive operators need only satisfy that property on the `discrete_complementarity_set()`; that is, when x and s are "cross" (orthogonal).

The cross-positive operators (on some fixed cone) themselves form a closed convex cone. This method computes and returns the generators of that cone as a list of matrices.

Cross-positive operators are also called exponentially-positive, since they become positive operators when exponentiated. Other equivalent names are resolvent-positive, essentially-positive, and quasimonotone.

OUTPUT:

A list of n-by-n matrices where n is the ambient dimension of this cone. Each matrix L in the list has the property that $s(L(x)) \geq 0$ whenever (x, s) is an element of this cone's `discrete_complementarity_set()`.

The returned matrices generate the cone of cross-positive operators on this cone; that is,

- Any nonnegative linear combination of the returned matrices is cross-positive on this cone.
- Every cross-positive operator on this cone is some nonnegative linear combination of the returned matrices.

See also

`lyapunov_like_basis()`, `positive_operators_gens()`, `Z_operators_gens()`

REFERENCES:

- [SV1970]
- [Or2018b]

EXAMPLES:

Cross-positive operators on the nonnegative orthant are negations of Z-matrices; that is, matrices whose off-diagonal elements are nonnegative:

```
sage: K = cones.nonnegative_orthant(2)
sage: K.cross_positive_operators_gens()
[
[0 1]  [0 0]  [1 0]  [-1  0]  [0 0]  [ 0  0]
[0 0], [1 0], [0 0], [ 0  0], [0 1], [ 0 -1]
]
sage: K = Cone([(1,0,0,0), (0,1,0,0), (0,0,1,0), (0,0,0,1)])
sage: all(c[i][j] >= 0 for c in K.cross_positive_operators_gens()
....:                  for i in range(c.nrows())
....:                  for j in range(c.ncols())
....:                  if i != j)
True
```

```
>>> from sage.all import *
>>> K = cones.nonnegative_orthant(Integer(2))
>>> K.cross_positive_operators_gens()
[
```

(continues on next page)

(continued from previous page)

```
[0 1]  [0 0]  [1 0]  [-1  0]  [0 0]  [ 0  0]
[0 0], [1 0], [0 0], [ 0  0], [0 1], [ 0 -1]
]
>>> K = Cone([(Integer(1),Integer(0),Integer(0),Integer(0)), (Integer(0),
↪Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(0),Integer(1),
↪Integer(0)), (Integer(0),Integer(0),Integer(0),Integer(1))])
>>> all(c[i][j] >= Integer(0) for c in K.cross_positive_operators_gens()
...                  for i in range(c.nrows())
...                  for j in range(c.ncols())
...                  if i != j)
True
```

The trivial cone in a trivial space has no cross-positive operators:

```
sage: K = cones.trivial(0)
sage: K.cross_positive_operators_gens()
[]
```

```
>>> from sage.all import *
>>> K = cones.trivial(Integer(0))
>>> K.cross_positive_operators_gens()
[]
```

Every operator is a cross-positive operator on the ambient vector space:

```
sage: K = Cone([(1,), (-1,)])
sage: K.is_full_space()
True
sage: K.cross_positive_operators_gens()
[[1], [-1]]

sage: K = Cone([(1,0), (-1,0), (0,1), (0,-1)])
sage: K.is_full_space()
True
sage: K.cross_positive_operators_gens()
[
[1 0]  [-1  0]  [0 1]  [ 0 -1]  [0 0]  [ 0  0]  [0 0]  [ 0  0]
[0 0], [ 0  0], [0 0], [ 0  0], [1 0], [-1  0], [0 1], [ 0 -1]
]
```

```
>>> from sage.all import *
>>> K = Cone([(Integer(1),), (-Integer(1),)])
>>> K.is_full_space()
True
>>> K.cross_positive_operators_gens()
[[1], [-1]]

>>> K = Cone([(Integer(1),Integer(0)), (-Integer(1),Integer(0)), (Integer(0),
↪Integer(1)), (Integer(0),-Integer(1))])
>>> K.is_full_space()
True
>>> K.cross_positive_operators_gens()
```

(continues on next page)

(continued from previous page)

```
[
[1 0]  [-1  0]  [0 1]  [ 0 -1]  [0 0]  [ 0  0]  [0 0]  [ 0  0]
[0 0], [ 0  0], [0 0], [ 0  0], [1 0], [-1  0], [0 1], [ 0 -1]
]
```

A non-obvious application is to find the cross-positive operators on the right half-plane [Or2018b]:

```
sage: K = Cone([(1,0), (0,1), (0,-1)])
sage: K.cross_positive_operators_gens()
[
[1 0]  [-1  0]  [0 0]  [ 0  0]  [0 0]  [ 0  0]
[0 0], [ 0  0], [1 0], [-1  0], [0 1], [ 0 -1]
]
```

```
>>> from sage.all import *
>>> K = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1)), (Integer(0),-
↪Integer(1))])
>>> K.cross_positive_operators_gens()
[
[1 0]  [-1  0]  [0 0]  [ 0  0]  [0 0]  [ 0  0]
[0 0], [ 0  0], [1 0], [-1  0], [0 1], [ 0 -1]
]
```

Cross-positive operators on a subspace are Lyapunov-like and vice-versa:

```
sage: K = Cone([(1,0), (-1,0), (0,1), (0,-1)])
sage: K.is_full_space()
True
sage: lls = span(vector(l.list())
....:            for l in K.lyapunov_like_basis())
sage: cs  = span(vector(c.list())
....:            for c in K.cross_positive_operators_gens())
sage: cs == lls
True
```

```
>>> from sage.all import *
>>> K = Cone([(Integer(1),Integer(0)), (-Integer(1),Integer(0)), (Integer(0),
↪Integer(1)), (Integer(0),-Integer(1))])
>>> K.is_full_space()
True
>>> lls = span(vector(l.list())
...            for l in K.lyapunov_like_basis())
>>> cs  = span(vector(c.list())
...            for c in K.cross_positive_operators_gens())
>>> cs == lls
True
```

discrete_complementarity_set()

Compute a discrete complementarity set of this cone.

A discrete complementarity set of a cone is the set of all orthogonal pairs (x, s) where x is in some fixed generating set of the cone, and s is in some fixed generating set of its dual. The generators chosen for this cone and its dual are simply their `rays()`.

OUTPUT:

A tuple of pairs (x, s) such that,

- x and s are nonzero.
- $s(x)$ is zero.
- x is one of this cone's `rays()`.
- s is one of the `rays()` of this cone's `dual()`.

REFERENCES:

- [Or2017]

EXAMPLES:

Pairs of standard basis elements form a discrete complementarity set for the nonnegative orthant:

```
sage: K = cones.nonnegative_orthant(2)
sage: K.discrete_complementarity_set()
((N(1, 0), M(0, 1)), (N(0, 1), M(1, 0)))
```

```
>>> from sage.all import *
>>> K = cones.nonnegative_orthant(Integer(2))
>>> K.discrete_complementarity_set()
((N(1, 0), M(0, 1)), (N(0, 1), M(1, 0)))
```

If a cone consists of a single ray, then the second components of a discrete complementarity set for that cone should generate the orthogonal complement of the ray:

```
sage: K = Cone([(1,0)])
sage: K.discrete_complementarity_set()
((N(1, 0), M(0, 1)), (N(1, 0), M(0, -1)))
sage: K = Cone([(1,0,0)])
sage: K.discrete_complementarity_set()
((N(1, 0, 0), M(0, 1, 0)),
 (N(1, 0, 0), M(0, -1, 0)),
 (N(1, 0, 0), M(0, 0, 1)),
 (N(1, 0, 0), M(0, 0, -1)))
```

```
>>> from sage.all import *
>>> K = Cone([(Integer(1),Integer(0))])
>>> K.discrete_complementarity_set()
((N(1, 0), M(0, 1)), (N(1, 0), M(0, -1)))
>>> K = Cone([(Integer(1),Integer(0),Integer(0))])
>>> K.discrete_complementarity_set()
((N(1, 0, 0), M(0, 1, 0)),
 (N(1, 0, 0), M(0, -1, 0)),
 (N(1, 0, 0), M(0, 0, 1)),
 (N(1, 0, 0), M(0, 0, -1)))
```

When a cone is the entire space, its dual is the trivial cone, so the only discrete complementarity set for it is empty:

```
sage: K = Cone([(1,0), (-1,0), (0,1), (0,-1)])
sage: K.is_full_space()
```

(continues on next page)

(continued from previous page)

```
True
sage: K.discrete_complementarity_set()
()
```

```
>>> from sage.all import *
>>> K = Cone([(Integer(1),Integer(0)), (-Integer(1),Integer(0)), (Integer(0),
↪Integer(1)), (Integer(0),-Integer(1))])
>>> K.is_full_space()
True
>>> K.discrete_complementarity_set()
()
```

Likewise for trivial cones, whose duals are the entire space:

```
sage: cones.trivial(0).discrete_complementarity_set()
()
```

```
>>> from sage.all import *
>>> cones.trivial(Integer(0)).discrete_complementarity_set()
()
```

dual()

Return the dual cone of `self`.

OUTPUT: *cone*

EXAMPLES:

```
sage: cone = Cone([(1,0), (-1,3)])
sage: cone.dual().rays()
M(0, 1),
M(3, 1)
in 2-d lattice M
```

```
>>> from sage.all import *
>>> cone = Cone([(Integer(1),Integer(0)), (-Integer(1),Integer(3))])
>>> cone.dual().rays()
M(0, 1),
M(3, 1)
in 2-d lattice M
```

Now let's look at a more complicated case:

```
sage: cone = Cone([(-2,-1,2), (4,1,0), (-4,-1,-5), (4,1,5)])
sage: cone.is_strictly_convex()
False
sage: cone.dim()
3
sage: cone.dual().rays()
M(7, -18, -2),
M(1,  -4,  0)
in 3-d lattice M
```

(continues on next page)

(continued from previous page)

```
sage: cone.dual().dual() is cone
True
```

```
>>> from sage.all import *
>>> cone = Cone([(-Integer(2),-Integer(1),Integer(2)), (Integer(4),Integer(1),
↪Integer(0)), (-Integer(4),-Integer(1),-Integer(5)), (Integer(4),Integer(1),
↪Integer(5))])
>>> cone.is_strictly_convex()
False
>>> cone.dim()
3
>>> cone.dual().rays()
M(7, -18, -2),
M(1,  -4,  0)
in 3-d lattice M
>>> cone.dual().dual() is cone
True
```

We correctly handle the degenerate cases:

```
sage: N = ToricLattice(2)
sage: Cone([], lattice=N).dual().rays()  # empty cone
M( 1,  0),
M(-1,  0),
M( 0,  1),
M( 0, -1)
in 2-d lattice M
sage: Cone([(1,0)], lattice=N).dual().rays()  # ray in 2d
M(1,  0),
M(0,  1),
M(0, -1)
in 2-d lattice M
sage: Cone([(1,0),(-1,0)], lattice=N).dual().rays()  # line in 2d
M(0,  1),
M(0, -1)
in 2-d lattice M
sage: Cone([(1,0),(0,1)], lattice=N).dual().rays()  # strictly convex cone
M(0, 1),
M(1, 0)
in 2-d lattice M
sage: Cone([(1,0),(-1,0),(0,1)], lattice=N).dual().rays()  # half space
M(0, 1)
in 2-d lattice M
sage: Cone([(1,0),(0,1),(-1,-1)], lattice=N).dual().rays()  # whole space
Empty collection
in 2-d lattice M
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(2))
>>> Cone([], lattice=N).dual().rays()  # empty cone
M( 1,  0),
M(-1,  0),
```

(continues on next page)

(continued from previous page)

```
M( 0,  1),
M( 0, -1)
in 2-d lattice M
>>> Cone([(Integer(1),Integer(0))], lattice=N).dual().rays()   # ray in 2d
M(1,  0),
M(0,  1),
M(0, -1)
in 2-d lattice M
>>> Cone([(Integer(1),Integer(0)),(-Integer(1),Integer(0))], lattice=N).
↪dual().rays()   # line in 2d
M(0,  1),
M(0, -1)
in 2-d lattice M
>>> Cone([(Integer(1),Integer(0)),(Integer(0),Integer(1))], lattice=N).dual().
↪rays()   # strictly convex cone
M(0, 1),
M(1, 0)
in 2-d lattice M
>>> Cone([(Integer(1),Integer(0)),(-Integer(1),Integer(0)),(Integer(0),
↪Integer(1))], lattice=N).dual().rays()   # half space
M(0, 1)
in 2-d lattice M
>>> Cone([(Integer(1),Integer(0)),(Integer(0),Integer(1)),(-Integer(1),-
↪Integer(1))], lattice=N).dual().rays()   # whole space
Empty collection
in 2-d lattice M
```

embed(*cone*)

Return the cone equivalent to the given one, but sitting in `self` as a face.

You may need to use this method before calling methods of `cone` that depend on the ambient structure, such as *ambient_ray_indices()* or *facet_of()*. The cone returned by this method will have `self` as ambient. If `cone` does not represent a valid cone of `self`, `ValueError` exception is raised.

> **Note**
>
> This method is very quick if `self` is already the ambient structure of `cone`, so you can use without extra checks and performance hit even if `cone` is likely to sit in `self` but in principle may not.

INPUT:

- `cone` – a *cone*

OUTPUT:

- a *cone*, equivalent to `cone` but sitting inside `self`.

EXAMPLES:

Let's take a 3-d cone on 4 rays:

```
sage: c = Cone([(1,0,1), (0,1,1), (-1,0,1), (0,-1,1)])
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(1),Integer(0),Integer(1)), (Integer(0),Integer(1),
↪Integer(1)), (-Integer(1),Integer(0),Integer(1)), (Integer(0),-Integer(1),
↪Integer(1))])
```

Then any ray generates a 1-d face of this cone, but if you construct such a face directly, it will not "sit" inside the cone:

```
sage: ray = Cone([(0,-1,1)])
sage: ray
1-d cone in 3-d lattice N
sage: ray.ambient_ray_indices()
(0,)
sage: ray.adjacent()                                                    #␣
↪needs sage.graphs
()
sage: ray.ambient()
1-d cone in 3-d lattice N
```

```
>>> from sage.all import *
>>> ray = Cone([(Integer(0),-Integer(1),Integer(1))])
>>> ray
1-d cone in 3-d lattice N
>>> ray.ambient_ray_indices()
(0,)
>>> ray.adjacent()                                                    #␣
↪needs sage.graphs
()
>>> ray.ambient()
1-d cone in 3-d lattice N
```

If we want to operate with this ray as a face of the cone, we need to embed it first:

```
sage: # needs sage.graphs
sage: e_ray = c.embed(ray)
sage: e_ray
1-d face of 3-d cone in 3-d lattice N
sage: e_ray.rays()
N(0, -1, 1)
in 3-d lattice N
sage: e_ray is ray
False
sage: e_ray.is_equivalent(ray)
True
sage: e_ray.ambient_ray_indices()
(3,)
sage: e_ray.adjacent()
(1-d face of 3-d cone in 3-d lattice N,
 1-d face of 3-d cone in 3-d lattice N)
sage: e_ray.ambient()
3-d cone in 3-d lattice N
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> e_ray = c.embed(ray)
>>> e_ray
1-d face of 3-d cone in 3-d lattice N
>>> e_ray.rays()
N(0, -1, 1)
in 3-d lattice N
>>> e_ray is ray
False
>>> e_ray.is_equivalent(ray)
True
>>> e_ray.ambient_ray_indices()
(3,)
>>> e_ray.adjacent()
(1-d face of 3-d cone in 3-d lattice N,
 1-d face of 3-d cone in 3-d lattice N)
>>> e_ray.ambient()
3-d cone in 3-d lattice N
```

Not every cone can be embedded into a fixed ambient cone:

```
sage: c.embed(Cone([(0,0,1)]))
Traceback (most recent call last):
...
ValueError: 1-d cone in 3-d lattice N is not a face
of 3-d cone in 3-d lattice N!
sage: c.embed(Cone([(1,0,1), (-1,0,1)]))                                    #␣
↪needs sage.graphs
Traceback (most recent call last):
...
ValueError: 2-d cone in 3-d lattice N is not a face
of 3-d cone in 3-d lattice N!
```

```
>>> from sage.all import *
>>> c.embed(Cone([(Integer(0),Integer(0),Integer(1))]))
Traceback (most recent call last):
...
ValueError: 1-d cone in 3-d lattice N is not a face
of 3-d cone in 3-d lattice N!
>>> c.embed(Cone([(Integer(1),Integer(0),Integer(1)), (-Integer(1),Integer(0),
↪Integer(1))]))                                # needs sage.graphs
Traceback (most recent call last):
...
ValueError: 2-d cone in 3-d lattice N is not a face
of 3-d cone in 3-d lattice N!
```

face_lattice()

Return the face lattice of `self`.

This lattice will have the origin as the bottom (we do not include the empty set as a face) and this cone itself as the top.

OUTPUT:

- finite poset of *cones*.

EXAMPLES:

Let's take a look at the face lattice of the first quadrant:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: L = quadrant.face_lattice()                                         #␣
↪needs sage.combinat sage.graphs
sage: L                                                                   #␣
↪needs sage.combinat sage.graphs
Finite lattice containing 4 elements with distinguished linear extension
```

```
>>> from sage.all import *
>>> quadrant = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> L = quadrant.face_lattice()                                         #␣
↪needs sage.combinat sage.graphs
>>> L                                                                   #␣
↪needs sage.combinat sage.graphs
Finite lattice containing 4 elements with distinguished linear extension
```

To see all faces arranged by dimension, you can do this:

```
sage: for level in L.level_sets(): print(level)                           #␣
↪needs sage.combinat sage.graphs
[0-d face of 2-d cone in 2-d lattice N]
[1-d face of 2-d cone in 2-d lattice N,
 1-d face of 2-d cone in 2-d lattice N]
[2-d cone in 2-d lattice N]
```

```
>>> from sage.all import *
>>> for level in L.level_sets(): print(level)                           #␣
↪needs sage.combinat sage.graphs
[0-d face of 2-d cone in 2-d lattice N]
[1-d face of 2-d cone in 2-d lattice N,
 1-d face of 2-d cone in 2-d lattice N]
[2-d cone in 2-d lattice N]
```

For a particular face you can look at its actual rays…

```
sage: face = L.level_sets()[1][0]                                         #␣
↪needs sage.combinat sage.graphs
sage: face.rays()                                                         #␣
↪needs sage.combinat sage.graphs
N(1, 0)
in 2-d lattice N
```

```
>>> from sage.all import *
>>> face = L.level_sets()[Integer(1)][Integer(0)]                         ␣
↪              # needs sage.combinat sage.graphs
>>> face.rays()                                                         #␣
↪needs sage.combinat sage.graphs
N(1, 0)
in 2-d lattice N
```

… or you can see the index of the ray of the original cone that corresponds to the above one:

```
sage: face.ambient_ray_indices()                                          #␣
↪needs sage.combinat sage.graphs
(0,)
sage: quadrant.ray(0)
N(1, 0)
```

```
>>> from sage.all import *
>>> face.ambient_ray_indices()                                          #␣
↪needs sage.combinat sage.graphs
(0,)
>>> quadrant.ray(Integer(0))
N(1, 0)
```

An alternative to extracting faces from the face lattice is to use *faces()* method:

```
sage: face is quadrant.faces(dim=1)[0]                                    #␣
↪needs sage.combinat sage.graphs
True
```

```
>>> from sage.all import *
>>> face is quadrant.faces(dim=Integer(1))[Integer(0)]                    ␣
↪               # needs sage.combinat sage.graphs
True
```

The advantage of working with the face lattice directly is that you can (relatively easily) get faces that are related to the given one:

```
sage: face = L.level_sets()[1][0]                                         #␣
↪needs sage.combinat sage.graphs
sage: D = L.hasse_diagram()                                               #␣
↪needs sage.combinat sage.graphs
sage: sorted(D.neighbors(face))                                           #␣
↪needs sage.combinat sage.graphs
[0-d face of 2-d cone in 2-d lattice N,
 2-d cone in 2-d lattice N]
```

```
>>> from sage.all import *
>>> face = L.level_sets()[Integer(1)][Integer(0)]                         ␣
↪               # needs sage.combinat sage.graphs
>>> D = L.hasse_diagram()                                               #␣
↪needs sage.combinat sage.graphs
>>> sorted(D.neighbors(face))                                           #␣
↪needs sage.combinat sage.graphs
[0-d face of 2-d cone in 2-d lattice N,
 2-d cone in 2-d lattice N]
```

However, you can achieve some of this functionality using *facets()*, *facet_of()*, and *adjacent()* methods:

```
sage: # needs sage.graphs
sage: face = quadrant.faces(1)[0]
```

(continues on next page)

(continued from previous page)

```
sage: face
1-d face of 2-d cone in 2-d lattice N
sage: face.rays()
N(1, 0)
in 2-d lattice N
sage: face.facets()
(0-d face of 2-d cone in 2-d lattice N,)
sage: face.facet_of()
(2-d cone in 2-d lattice N,)
sage: face.adjacent()
(1-d face of 2-d cone in 2-d lattice N,)
sage: face.adjacent()[0].rays()
N(0, 1)
in 2-d lattice N
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> face = quadrant.faces(Integer(1))[Integer(0)]
>>> face
1-d face of 2-d cone in 2-d lattice N
>>> face.rays()
N(1, 0)
in 2-d lattice N
>>> face.facets()
(0-d face of 2-d cone in 2-d lattice N,)
>>> face.facet_of()
(2-d cone in 2-d lattice N,)
>>> face.adjacent()
(1-d face of 2-d cone in 2-d lattice N,)
>>> face.adjacent()[Integer(0)].rays()
N(0, 1)
in 2-d lattice N
```

Note that if `cone` is a face of `supercone`, then the face lattice of `cone` consists of (appropriate) faces of `supercone`:

```
sage: # needs sage.combinat sage.graphs
sage: supercone = Cone([(1,2,3,4), (5,6,7,8),
....:                   (1,2,4,8), (1,3,9,7)])
sage: supercone.face_lattice()
Finite lattice containing 16 elements with distinguished linear extension
sage: supercone.face_lattice().top()
4-d cone in 4-d lattice N
sage: cone = supercone.facets()[0]
sage: cone
3-d face of 4-d cone in 4-d lattice N
sage: cone.face_lattice()
Finite poset containing 8 elements with distinguished linear extension
sage: cone.face_lattice().bottom()
0-d face of 4-d cone in 4-d lattice N
sage: cone.face_lattice().top()
3-d face of 4-d cone in 4-d lattice N
```

(continues on next page)

(continued from previous page)

```
sage: cone.face_lattice().top() == cone
True
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.graphs
>>> supercone = Cone([(Integer(1),Integer(2),Integer(3),Integer(4)),␣
↪(Integer(5),Integer(6),Integer(7),Integer(8)),
...                   (Integer(1),Integer(2),Integer(4),Integer(8)),␣
↪(Integer(1),Integer(3),Integer(9),Integer(7))])
>>> supercone.face_lattice()
Finite lattice containing 16 elements with distinguished linear extension
>>> supercone.face_lattice().top()
4-d cone in 4-d lattice N
>>> cone = supercone.facets()[Integer(0)]
>>> cone
3-d face of 4-d cone in 4-d lattice N
>>> cone.face_lattice()
Finite poset containing 8 elements with distinguished linear extension
>>> cone.face_lattice().bottom()
0-d face of 4-d cone in 4-d lattice N
>>> cone.face_lattice().top()
3-d face of 4-d cone in 4-d lattice N
>>> cone.face_lattice().top() == cone
True
```

faces(*dim=None*, *codim=None*)

Return faces of `self` of specified (co)dimension.

INPUT:

- `dim` – integer; dimension of the requested faces
- `codim` – integer; codimension of the requested faces

Note

You can specify at most one parameter. If you don't give any, then all faces will be returned.

OUTPUT:

- if either `dim` or `codim` is given, the output will be a `tuple` of *cones*;
- if neither `dim` nor `codim` is given, the output will be the `tuple` of tuples as above, giving faces of all existing dimensions. If you care about inclusion relations between faces, consider using *face_lattice()* or *adjacent()*, *facet_of()*, and *facets()*.

EXAMPLES:

Let's take a look at the faces of the first quadrant:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: quadrant.faces()                                                    #␣
↪needs sage.graphs
((0-d face of 2-d cone in 2-d lattice N,),
```

(continues on next page)

(continued from previous page)

```
 (1-d face of 2-d cone in 2-d lattice N,
  1-d face of 2-d cone in 2-d lattice N),
 (2-d cone in 2-d lattice N,))
sage: quadrant.faces(dim=1)                                                    #␣
↪needs sage.graphs
(1-d face of 2-d cone in 2-d lattice N,
 1-d face of 2-d cone in 2-d lattice N)
sage: face = quadrant.faces(dim=1)[0]                                          #␣
↪needs sage.graphs
```

```
>>> from sage.all import *
>>> quadrant = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> quadrant.faces()                                                          #␣
↪needs sage.graphs
((0-d face of 2-d cone in 2-d lattice N,),
 (1-d face of 2-d cone in 2-d lattice N,
  1-d face of 2-d cone in 2-d lattice N),
 (2-d cone in 2-d lattice N,))
>>> quadrant.faces(dim=Integer(1))                                             ␣
↪     # needs sage.graphs
(1-d face of 2-d cone in 2-d lattice N,
 1-d face of 2-d cone in 2-d lattice N)
>>> face = quadrant.faces(dim=Integer(1))[Integer(0)]                          ␣
↪              # needs sage.graphs
```

Now you can look at the actual rays of this face…

```
sage: face.rays()                                                              #␣
↪needs sage.graphs
N(1, 0)
in 2-d lattice N
```

```
>>> from sage.all import *
>>> face.rays()                                                               #␣
↪needs sage.graphs
N(1, 0)
in 2-d lattice N
```

… or you can see indices of the rays of the original cone that correspond to the above ray:

```
sage: face.ambient_ray_indices()                                               #␣
↪needs sage.graphs
(0,)
sage: quadrant.ray(0)
N(1, 0)
```

```
>>> from sage.all import *
>>> face.ambient_ray_indices()                                                #␣
↪needs sage.graphs
(0,)
>>> quadrant.ray(Integer(0))
N(1, 0)
```

Note that it is OK to ask for faces of too small or high dimension:

```
sage: quadrant.faces(-1)                                                    #␣
↪needs sage.graphs
()
sage: quadrant.faces(3)                                                     #␣
↪needs sage.graphs
()
```

```
>>> from sage.all import *
>>> quadrant.faces(-Integer(1))                                              ␣
↪     # needs sage.graphs
()
>>> quadrant.faces(Integer(3))                                               ␣
↪     # needs sage.graphs
()
```

In the case of non-strictly convex cones even faces of small nonnegative dimension may be missing:

```
sage: # needs sage.graphs
sage: halfplane = Cone([(1,0), (0,1), (-1,0)])
sage: halfplane.faces(0)
()
sage: halfplane.faces()
((1-d face of 2-d cone in 2-d lattice N,),
 (2-d cone in 2-d lattice N,))
sage: plane = Cone([(1,0), (0,1), (-1,-1)])
sage: plane.faces(1)
()
sage: plane.faces()
((2-d cone in 2-d lattice N,),)
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> halfplane = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1)), (-
↪Integer(1),Integer(0))])
>>> halfplane.faces(Integer(0))
()
>>> halfplane.faces()
((1-d face of 2-d cone in 2-d lattice N,),
 (2-d cone in 2-d lattice N,))
>>> plane = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1)), (-
↪Integer(1),-Integer(1))])
>>> plane.faces(Integer(1))
()
>>> plane.faces()
((2-d cone in 2-d lattice N,),)
```

facet_normals()

Return inward normals to facets of `self`.

Note

1. For a not full-dimensional cone facet normals will specify hyperplanes whose intersections with the space spanned by `self` give facets of `self`.
2. For a not strictly convex cone facet normals will be orthogonal to the linear subspace of `self`, i.e. they always will be elements of the dual cone of `self`.
3. The order of normals is random, but consistent with `facets()`.

OUTPUT: a `PointCollection`

If the ambient `lattice()` of `self` is a `toric lattice`, the facet normals will be elements of the dual lattice. If it is a general lattice (like `ZZ^n`) that does not have a `dual()` method, the facet normals will be returned as integral vectors.

EXAMPLES:

```
sage: cone = Cone([(1,0), (-1,3)])
sage: cone.facet_normals()
M(0, 1),
M(3, 1)
in 2-d lattice M
```

```
>>> from sage.all import *
>>> cone = Cone([(Integer(1),Integer(0)), (-Integer(1),Integer(3))])
>>> cone.facet_normals()
M(0, 1),
M(3, 1)
in 2-d lattice M
```

Now let's look at a more complicated case:

```
sage: cone = Cone([(-2,-1,2), (4,1,0), (-4,-1,-5), (4,1,5)])
sage: cone.is_strictly_convex()
False
sage: cone.dim()
3
sage: cone.linear_subspace().dimension()
1
sage: lsg = (QQ^3)(cone.linear_subspace().gen(0)); lsg
(1, 1/4, 5/4)
sage: cone.facet_normals()
M(7, -18, -2),
M(1,  -4,  0)
in 3-d lattice M
sage: [lsg*normal for normal in cone.facet_normals()]
[0, 0]
```

```
>>> from sage.all import *
>>> cone = Cone([(-Integer(2),-Integer(1),Integer(2)), (Integer(4),Integer(1),
↪Integer(0)), (-Integer(4),-Integer(1),-Integer(5)), (Integer(4),Integer(1),
↪Integer(5))])
>>> cone.is_strictly_convex()
False
>>> cone.dim()
```

(continues on next page)

(continued from previous page)

```
3
>>> cone.linear_subspace().dimension()
1
>>> lsg = (QQ**Integer(3))(cone.linear_subspace().gen(Integer(0))); lsg
(1, 1/4, 5/4)
>>> cone.facet_normals()
M(7, -18, -2),
M(1,  -4,  0)
in 3-d lattice M
>>> [lsg*normal for normal in cone.facet_normals()]
[0, 0]
```

A lattice that does not have a `dual()` method:

```
sage: Cone([(1,1),(0,1)], lattice=ZZ^2).facet_normals()
(-1, 1),
( 1, 0)
in Ambient free module of rank 2
over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> Cone([(Integer(1),Integer(1)),(Integer(0),Integer(1))],␣
↪lattice=ZZ**Integer(2)).facet_normals()
(-1, 1),
( 1, 0)
in Ambient free module of rank 2
over the principal ideal domain Integer Ring
```

We correctly handle the degenerate cases:

```
sage: N = ToricLattice(2)
sage: Cone([], lattice=N).facet_normals()  # empty cone
Empty collection
in 2-d lattice M
sage: Cone([(1,0)], lattice=N).facet_normals()  # ray in 2d
M(1, 0)
in 2-d lattice M
sage: Cone([(1,0),(-1,0)], lattice=N).facet_normals()  # line in 2d
Empty collection
in 2-d lattice M
sage: Cone([(1,0),(0,1)], lattice=N).facet_normals()  # strictly convex cone
M(0, 1),
M(1, 0)
in 2-d lattice M
sage: Cone([(1,0),(-1,0),(0,1)], lattice=N).facet_normals()  # half space
M(0, 1)
in 2-d lattice M
sage: Cone([(1,0),(0,1),(-1,-1)], lattice=N).facet_normals()  # whole space
Empty collection
in 2-d lattice M
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(2))
>>> Cone([], lattice=N).facet_normals()   # empty cone
Empty collection
in 2-d lattice M
>>> Cone([(Integer(1),Integer(0))], lattice=N).facet_normals()   # ray in 2d
M(1, 0)
in 2-d lattice M
>>> Cone([(Integer(1),Integer(0)),(-Integer(1),Integer(0))], lattice=N).facet_
↪normals()   # line in 2d
Empty collection
in 2-d lattice M
>>> Cone([(Integer(1),Integer(0)),(Integer(0),Integer(1))], lattice=N).facet_
↪normals()   # strictly convex cone
M(0, 1),
M(1, 0)
in 2-d lattice M
>>> Cone([(Integer(1),Integer(0)),(-Integer(1),Integer(0)),(Integer(0),
↪Integer(1))], lattice=N).facet_normals()   # half space
M(0, 1)
in 2-d lattice M
>>> Cone([(Integer(1),Integer(0)),(Integer(0),Integer(1)),(-Integer(1),-
↪Integer(1))], lattice=N).facet_normals()   # whole space
Empty collection
in 2-d lattice M
```

facet_of()

Return *cones* of the ambient face lattice having `self` as a facet.

OUTPUT: `tuple` of *cones*

EXAMPLES:

```
sage: # needs sage.graphs
sage: octant = Cone([(1,0,0), (0,1,0), (0,0,1)])
sage: octant.facet_of()
()
sage: one_face = octant.faces(1)[0]
sage: len(one_face.facet_of())
2
sage: one_face.facet_of()[1]
2-d face of 3-d cone in 3-d lattice N
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> octant = Cone([(Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0)), (Integer(0),Integer(0),Integer(1))])
>>> octant.facet_of()
()
>>> one_face = octant.faces(Integer(1))[Integer(0)]
>>> len(one_face.facet_of())
2
>>> one_face.facet_of()[Integer(1)]
2-d face of 3-d cone in 3-d lattice N
```

While fan is the top element of its own cone lattice, which is a variant of a face lattice, we do not refer to cones as its facets:

```
sage: fan = Fan([octant])                                                    #␣
↪needs sage.graphs
sage: fan.generating_cone(0).facet_of()                                      #␣
↪needs sage.graphs
()
```

```
>>> from sage.all import *
>>> fan = Fan([octant])                                                    #␣
↪needs sage.graphs
>>> fan.generating_cone(Integer(0)).facet_of()                               ␣
↪     # needs sage.graphs
()
```

Subcones of generating cones work as before:

```
sage: one_cone = fan(1)[0]                                                   #␣
↪needs sage.graphs
sage: len(one_cone.facet_of())                                               #␣
↪needs sage.graphs
2
```

```
>>> from sage.all import *
>>> one_cone = fan(Integer(1))[Integer(0)]                                   ␣
↪             # needs sage.graphs
>>> len(one_cone.facet_of())                                               #␣
↪needs sage.graphs
2
```

facets()

Return facets (faces of codimension 1) of `self`.

OUTPUT: `tuple` of *cones*

EXAMPLES:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: quadrant.facets()                                                      #␣
↪needs sage.graphs
(1-d face of 2-d cone in 2-d lattice N,
 1-d face of 2-d cone in 2-d lattice N)
```

```
>>> from sage.all import *
>>> quadrant = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> quadrant.facets()                                                      #␣
↪needs sage.graphs
(1-d face of 2-d cone in 2-d lattice N,
 1-d face of 2-d cone in 2-d lattice N)
```

incidence_matrix()

Return the incidence matrix.

Note: The columns correspond to facets/facet normals in the order of *facet_normals()*, the rows correspond to the rays in the order of `rays()`.

EXAMPLES:

```
sage: octant = Cone([(1,0,0), (0,1,0), (0,0,1)])
sage: octant.incidence_matrix()
[0 1 1]
[1 0 1]
[1 1 0]

sage: halfspace = Cone([(1,0,0), (0,1,0), (-1,-1,0), (0,0,1)])
sage: halfspace.incidence_matrix()
[0]
[1]
[1]
[1]
[1]
```

```
>>> from sage.all import *
>>> octant = Cone([(Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0)), (Integer(0),Integer(0),Integer(1))])
>>> octant.incidence_matrix()
[0 1 1]
[1 0 1]
[1 1 0]

>>> halfspace = Cone([(Integer(1),Integer(0),Integer(0)), (Integer(0),
↪Integer(1),Integer(0)), (-Integer(1),-Integer(1),Integer(0)), (Integer(0),
↪Integer(0),Integer(1))])
>>> halfspace.incidence_matrix()
[0]
[1]
[1]
[1]
[1]
```

interior()

Return the interior of `self`.

OUTPUT:

- either `self`, an empty polyhedron, or an instance of *RelativeInterior*.

EXAMPLES:

```
sage: c = Cone([(1,0,0), (0,1,0)]); c
2-d cone in 3-d lattice N
sage: c.interior()
The empty polyhedron in ZZ^3

sage: origin = cones.trivial(2); origin
```

(continues on next page)

(continued from previous page)

```
0-d cone in 2-d lattice N
sage: origin.interior()
The empty polyhedron in ZZ^2

sage: K = cones.nonnegative_orthant(2); K
2-d cone in 2-d lattice N
sage: K.interior()
Relative interior of 2-d cone in 2-d lattice N

sage: K2 = Cone([(1,0),(-1,0),(0,1),(0,-1)]); K2
2-d cone in 2-d lattice N
sage: K2.interior() is K2
True
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0))]); c
2-d cone in 3-d lattice N
>>> c.interior()
The empty polyhedron in ZZ^3

>>> origin = cones.trivial(Integer(2)); origin
0-d cone in 2-d lattice N
>>> origin.interior()
The empty polyhedron in ZZ^2

>>> K = cones.nonnegative_orthant(Integer(2)); K
2-d cone in 2-d lattice N
>>> K.interior()
Relative interior of 2-d cone in 2-d lattice N

>>> K2 = Cone([(Integer(1),Integer(0)),(-Integer(1),Integer(0)),(Integer(0),
↪Integer(1)),(Integer(0),-Integer(1))]); K2
2-d cone in 2-d lattice N
>>> K2.interior() is K2
True
```

interior_contains(**args*)

Check if a given point is contained in the interior of `self`.

For a cone of strictly lower-dimension than the ambient space, the interior is always empty. You probably want to use *relative_interior_contains()* in this case.

INPUT:

- anything. An attempt will be made to convert all arguments into a single element of the ambient space of `self`. If it fails, `False` will be returned.

OUTPUT:

- `True` if the given point is contained in the interior of `self`, `False` otherwise.

EXAMPLES:

```
sage: c = Cone([(1,0), (0,1)])
sage: c.contains((1,1))
True
sage: c.interior_contains((1,1))
True
sage: c.contains((1,0))
True
sage: c.interior_contains((1,0))
False
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> c.contains((Integer(1),Integer(1)))
True
>>> c.interior_contains((Integer(1),Integer(1)))
True
>>> c.contains((Integer(1),Integer(0)))
True
>>> c.interior_contains((Integer(1),Integer(0)))
False
```

intersection(*other*)

Compute the intersection of two cones.

INPUT:

- other – *cone*

OUTPUT: *cone*

This raises ValueError if the ambient space dimensions are not compatible.

EXAMPLES:

```
sage: cone1 = Cone([(1,0), (-1, 3)])
sage: cone2 = Cone([(-1,0), (2, 5)])
sage: cone1.intersection(cone2).rays()
N(-1, 3),
N( 2, 5)
in 2-d lattice N
```

```
>>> from sage.all import *
>>> cone1 = Cone([(Integer(1),Integer(0)), (-Integer(1), Integer(3))])
>>> cone2 = Cone([(-Integer(1),Integer(0)), (Integer(2), Integer(5))])
>>> cone1.intersection(cone2).rays()
N(-1, 3),
N( 2, 5)
in 2-d lattice N
```

The intersection can also be expressed using the operator &:

```
sage: (cone1 & cone2).rays()
N(-1, 3),
N( 2, 5)
in 2-d lattice N
```

```
>>> from sage.all import *
>>> (cone1 & cone2).rays()
N(-1, 3),
N( 2, 5)
in 2-d lattice N
```

It is OK to intersect cones living in sublattices of the same ambient lattice:

```
sage: N = cone1.lattice()
sage: Ns = N.submodule([(1,1)])
sage: cone3 = Cone([(1,1)], lattice=Ns)
sage: I = cone1.intersection(cone3)
sage: I.rays()
N(1, 1)
in Sublattice <N(1, 1)>
sage: I.lattice()
Sublattice <N(1, 1)>
```

```
>>> from sage.all import *
>>> N = cone1.lattice()
>>> Ns = N.submodule([(Integer(1),Integer(1))])
>>> cone3 = Cone([(Integer(1),Integer(1))], lattice=Ns)
>>> I = cone1.intersection(cone3)
>>> I.rays()
N(1, 1)
in Sublattice <N(1, 1)>
>>> I.lattice()
Sublattice <N(1, 1)>
```

But you cannot intersect cones from incompatible lattices without explicit conversion:

```
sage: cone1.intersection(cone1.dual())
Traceback (most recent call last):
...
ValueError: 2-d lattice N and 2-d lattice M
have different ambient lattices!
sage: cone1.intersection(Cone(cone1.dual().rays(), N)).rays()
N(3, 1),
N(0, 1)
in 2-d lattice N
```

```
>>> from sage.all import *
>>> cone1.intersection(cone1.dual())
Traceback (most recent call last):
...
ValueError: 2-d lattice N and 2-d lattice M
have different ambient lattices!
>>> cone1.intersection(Cone(cone1.dual().rays(), N)).rays()
N(3, 1),
N(0, 1)
in 2-d lattice N
```

An intersection with a polyhedron returns a polyhedron:

```
sage: cone = Cone([(1,0), (-1,0), (0,1)])
sage: p = polytopes.hypercube(2)
sage: cone & p
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices
sage: sorted(_.vertices_list())
[[-1, 0], [-1, 1], [1, 0], [1, 1]]
```

```
>>> from sage.all import *
>>> cone = Cone([(Integer(1),Integer(0)), (-Integer(1),Integer(0)),
↪(Integer(0),Integer(1))])
>>> p = polytopes.hypercube(Integer(2))
>>> cone & p
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices
>>> sorted(_.vertices_list())
[[-1, 0], [-1, 1], [1, 0], [1, 1]]
```

is_compact()

Check if the cone has no rays.

OUTPUT: `True` if the cone has no rays, `False` otherwise

EXAMPLES:

```
sage: c0 = cones.trivial(3)
sage: c0.is_trivial()
True
sage: c0.nrays()
0
```

```
>>> from sage.all import *
>>> c0 = cones.trivial(Integer(3))
>>> c0.is_trivial()
True
>>> c0.nrays()
0
```

is_empty()

Return whether `self` is the empty set.

Because a cone always contains the origin, this method returns `False`.

EXAMPLES:

```
sage: trivial_cone = cones.trivial(3)
sage: trivial_cone.is_empty()
False
```

```
>>> from sage.all import *
>>> trivial_cone = cones.trivial(Integer(3))
>>> trivial_cone.is_empty()
False
```

is_equivalent(*other*)

Check if `self` is "mathematically" the same as `other`.

INPUT:

- `other` – cone

OUTPUT:

- `True` if `self` and `other` define the same cones as sets of points in the same lattice, `False` otherwise.

There are three different equivalences between cones C_1 and C_2 in the same lattice:

1. They have the same generating rays in the same order. This is tested by `C1 == C2`.
2. They describe the same sets of points. This is tested by `C1.is_equivalent(C2)`.
3. They are in the same orbit of $GL(n, \mathbf{Z})$ (and, therefore, correspond to isomorphic affine toric varieties). This is tested by `C1.is_isomorphic(C2)`.

EXAMPLES:

```
sage: cone1 = Cone([(1,0), (-1, 3)])
sage: cone2 = Cone([(-1,3), (1, 0)])
sage: cone1.rays()
N( 1, 0),
N(-1, 3)
in 2-d lattice N
sage: cone2.rays()
N(-1, 3),
N( 1, 0)
in 2-d lattice N
sage: cone1 == cone2
False
sage: cone1.is_equivalent(cone2)
True
```

```
>>> from sage.all import *
>>> cone1 = Cone([(Integer(1),Integer(0)), (-Integer(1), Integer(3))])
>>> cone2 = Cone([(-Integer(1),Integer(3)), (Integer(1), Integer(0))])
>>> cone1.rays()
N( 1, 0),
N(-1, 3)
in 2-d lattice N
>>> cone2.rays()
N(-1, 3),
N( 1, 0)
in 2-d lattice N
>>> cone1 == cone2
False
>>> cone1.is_equivalent(cone2)
True
```

is_face_of(*cone*)

Check if `self` forms a face of another `cone`.

INPUT:

- `cone` – cone

OUTPUT: `True` if `self` is a face of `cone`, `False` otherwise

EXAMPLES:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: cone1 = Cone([(1,0)])
sage: cone2 = Cone([(1,2)])
sage: quadrant.is_face_of(quadrant)
True
sage: cone1.is_face_of(quadrant)
True
sage: cone2.is_face_of(quadrant)
False
```

```
>>> from sage.all import *
>>> quadrant = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> cone1 = Cone([(Integer(1),Integer(0))])
>>> cone2 = Cone([(Integer(1),Integer(2))])
>>> quadrant.is_face_of(quadrant)
True
>>> cone1.is_face_of(quadrant)
True
>>> cone2.is_face_of(quadrant)
False
```

Being a face means more than just saturating a facet inequality:

```
sage: octant = Cone([(1,0,0), (0,1,0), (0,0,1)])
sage: cone = Cone([(2,1,0),(1,2,0)])
sage: cone.is_face_of(octant)
False
```

```
>>> from sage.all import *
>>> octant = Cone([(Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0)), (Integer(0),Integer(0),Integer(1))])
>>> cone = Cone([(Integer(2),Integer(1),Integer(0)),(Integer(1),Integer(2),
↪Integer(0))])
>>> cone.is_face_of(octant)
False
```

is_full_dimensional()

Check if this cone is solid.

A cone is said to be solid if it has nonempty interior. That is, if its extreme rays span the entire ambient space.

An alias is *is_full_dimensional()*.

OUTPUT: `True` if this cone is solid, and `False` otherwise

See also

is_proper()

EXAMPLES:

The nonnegative orthant is always solid:

```
sage: quadrant = cones.nonnegative_orthant(2)
sage: quadrant.is_solid()
True
sage: octant = Cone([(1,0,0), (0,1,0), (0,0,1)])
sage: octant.is_solid()
True
```

```
>>> from sage.all import *
>>> quadrant = cones.nonnegative_orthant(Integer(2))
>>> quadrant.is_solid()
True
>>> octant = Cone([(Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0)), (Integer(0),Integer(0),Integer(1))])
>>> octant.is_solid()
True
```

However, if we embed the two-dimensional nonnegative quadrant into three-dimensional space, then the resulting cone no longer has interior, so it is not solid:

```
sage: quadrant = Cone([(1,0,0), (0,1,0)])
sage: quadrant.is_solid()
False
```

```
>>> from sage.all import *
>>> quadrant = Cone([(Integer(1),Integer(0),Integer(0)), (Integer(0),
↪Integer(1),Integer(0))])
>>> quadrant.is_solid()
False
```

is_full_space()

Check if this cone is equal to its ambient vector space.

An alias is *is_universe()*.

OUTPUT:

`True` if this cone equals its entire ambient vector space and `False` otherwise.

EXAMPLES:

A single ray in two dimensions is not equal to the entire space:

```
sage: K = Cone([(1,0)])
sage: K.is_full_space()
False
```

```
>>> from sage.all import *
>>> K = Cone([(Integer(1),Integer(0))])
>>> K.is_full_space()
False
```

Neither is the nonnegative orthant:

```
sage: K = cones.nonnegative_orthant(2)
sage: K.is_full_space()
False
```

```
>>> from sage.all import *
>>> K = cones.nonnegative_orthant(Integer(2))
>>> K.is_full_space()
False
```

The right half-space contains a vector subspace, but it is still not equal to the entire space:

```
sage: K = Cone([(1,0), (-1,0), (0,1)])
sage: K.is_full_space()
False
```

```
>>> from sage.all import *
>>> K = Cone([(Integer(1),Integer(0)), (-Integer(1),Integer(0)), (Integer(0),
→Integer(1))])
>>> K.is_full_space()
False
```

However, if we allow conic combinations of both axes, then the resulting cone is the entire two-dimensional space:

```
sage: K = Cone([(1,0), (-1,0), (0,1), (0,-1)])
sage: K.is_full_space()
True
```

```
>>> from sage.all import *
>>> K = Cone([(Integer(1),Integer(0)), (-Integer(1),Integer(0)), (Integer(0),
→Integer(1)), (Integer(0),-Integer(1))])
>>> K.is_full_space()
True
```

is_isomorphic(*other*)

Check if `self` is in the same $GL(n, \mathbf{Z})$-orbit as `other`.

INPUT:

- `other` – cone

OUTPUT: `True` if `self` and `other` are in the same $GL(n, \mathbf{Z})$-orbit, `False` otherwise

There are three different equivalences between cones C_1 and C_2 in the same lattice:

1. They have the same generating rays in the same order. This is tested by `C1 == C2`.
2. They describe the same sets of points. This is tested by `C1.is_equivalent(C2)`.
3. They are in the same orbit of $GL(n, \mathbf{Z})$ (and, therefore, correspond to isomorphic affine toric varieties). This is tested by `C1.is_isomorphic(C2)`.

EXAMPLES:

```
sage: cone1 = Cone([(1,0), (0, 3)])
sage: m = matrix(ZZ, [(1, -5), (-1, 4)]) # a GL(2,ZZ)-matrix
```

(continues on next page)

(continued from previous page)

```
sage: cone2 = Cone( m*r for r in cone1.rays() )
sage: cone1.is_isomorphic(cone2)
True

sage: cone1 = Cone([(1,0), (0, 3)])
sage: cone2 = Cone([(-1,3), (1, 0)])
sage: cone1.is_isomorphic(cone2)
False
```

```
>>> from sage.all import *
>>> cone1 = Cone([(Integer(1),Integer(0)), (Integer(0), Integer(3))])
>>> m = matrix(ZZ, [(Integer(1), -Integer(5)), (-Integer(1), Integer(4))]) #␣
↪a GL(2,ZZ)-matrix
>>> cone2 = Cone( m*r for r in cone1.rays() )
>>> cone1.is_isomorphic(cone2)
True

>>> cone1 = Cone([(Integer(1),Integer(0)), (Integer(0), Integer(3))])
>>> cone2 = Cone([(-Integer(1),Integer(3)), (Integer(1), Integer(0))])
>>> cone1.is_isomorphic(cone2)
False
```

is_proper()

Check if this cone is proper.

A cone is said to be proper if it is closed, convex, solid, and contains no lines. This cone is assumed to be closed and convex; therefore it is proper if it is solid and contains no lines.

OUTPUT: `True` if this cone is proper, and `False` otherwise

See also

is_strictly_convex(), *is_solid()*

EXAMPLES:

The nonnegative orthant is always proper:

```
sage: quadrant = cones.nonnegative_orthant(2)
sage: quadrant.is_proper()
True
sage: octant = Cone([(1,0,0), (0,1,0), (0,0,1)])
sage: octant.is_proper()
True
```

```
>>> from sage.all import *
>>> quadrant = cones.nonnegative_orthant(Integer(2))
>>> quadrant.is_proper()
True
>>> octant = Cone([(Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0)), (Integer(0),Integer(0),Integer(1))])
```

(continues on next page)

(continued from previous page)

```
>>> octant.is_proper()
True
```

However, if we embed the two-dimensional nonnegative quadrant into three-dimensional space, then the resulting cone no longer has interior, so it is not solid, and thus not proper:

```
sage: quadrant = Cone([(1,0,0), (0,1,0)])
sage: quadrant.is_proper()
False
```

```
>>> from sage.all import *
>>> quadrant = Cone([(Integer(1),Integer(0),Integer(0)), (Integer(0),
↪Integer(1),Integer(0))])
>>> quadrant.is_proper()
False
```

Likewise, a half-space contains at least one line, so it is not proper:

```
sage: halfspace = Cone([(1,0), (0,1), (-1,0)])
sage: halfspace.is_proper()
False
```

```
>>> from sage.all import *
>>> halfspace = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1)), (-
↪Integer(1),Integer(0))])
>>> halfspace.is_proper()
False
```

is_relatively_open()

Return whether `self` is relatively open.

OUTPUT: boolean

EXAMPLES:

```
sage: K = cones.nonnegative_orthant(3)
sage: K.is_relatively_open()
False

sage: K1 = Cone([(1,0), (-1,0)]); K1
1-d cone in 2-d lattice N
sage: K1.is_relatively_open()
True
```

```
>>> from sage.all import *
>>> K = cones.nonnegative_orthant(Integer(3))
>>> K.is_relatively_open()
False

>>> K1 = Cone([(Integer(1),Integer(0)), (-Integer(1),Integer(0))]); K1
1-d cone in 2-d lattice N
>>> K1.is_relatively_open()
True
```

is_simplicial()

Check if `self` is simplicial.

A cone is called **simplicial** if primitive vectors along its generating rays form a part of a *rational* basis of the ambient space.

OUTPUT: `True` if `self` is simplicial, `False` otherwise

EXAMPLES:

```
sage: cone1 = Cone([(1,0), (0, 3)])
sage: cone2 = Cone([(1,0), (0, 3), (-1,-1)])
sage: cone1.is_simplicial()
True
sage: cone2.is_simplicial()
False
```

```
>>> from sage.all import *
>>> cone1 = Cone([(Integer(1),Integer(0)), (Integer(0), Integer(3))])
>>> cone2 = Cone([(Integer(1),Integer(0)), (Integer(0), Integer(3)), (-
↪Integer(1),-Integer(1))])
>>> cone1.is_simplicial()
True
>>> cone2.is_simplicial()
False
```

is_smooth()

Check if `self` is smooth.

A cone is called **smooth** if primitive vectors along its generating rays form a part of an *integral* basis of the ambient space. Equivalently, they generate the whole lattice on the linear subspace spanned by the rays.

OUTPUT: `True` if `self` is smooth, `False` otherwise

EXAMPLES:

```
sage: cone1 = Cone([(1,0), (0, 1)])
sage: cone2 = Cone([(1,0), (-1, 3)])
sage: cone1.is_smooth()
True
sage: cone2.is_smooth()
False
```

```
>>> from sage.all import *
>>> cone1 = Cone([(Integer(1),Integer(0)), (Integer(0), Integer(1))])
>>> cone2 = Cone([(Integer(1),Integer(0)), (-Integer(1), Integer(3))])
>>> cone1.is_smooth()
True
>>> cone2.is_smooth()
False
```

The following cones are the same up to a $SL(2,\mathbf{Z})$ coordinate transformation:

```
sage: Cone([(1,0,0), (2,1,-1)]).is_smooth()
True
sage: Cone([(1,0,0), (2,1,1)]).is_smooth()
```

(continues on next page)

(continued from previous page)

```
True
sage: Cone([(1,0,0), (2,1,2)]).is_smooth()
True
```

```
>>> from sage.all import *
>>> Cone([(Integer(1),Integer(0),Integer(0)), (Integer(2),Integer(1),-
↪Integer(1))]).is_smooth()
True
>>> Cone([(Integer(1),Integer(0),Integer(0)), (Integer(2),Integer(1),
↪Integer(1))]).is_smooth()
True
>>> Cone([(Integer(1),Integer(0),Integer(0)), (Integer(2),Integer(1),
↪Integer(2))]).is_smooth()
True
```

is_solid()

Check if this cone is solid.

A cone is said to be solid if it has nonempty interior. That is, if its extreme rays span the entire ambient space.

An alias is *is_full_dimensional()*.

OUTPUT: `True` if this cone is solid, and `False` otherwise

See also

is_proper()

EXAMPLES:

The nonnegative orthant is always solid:

```
sage: quadrant = cones.nonnegative_orthant(2)
sage: quadrant.is_solid()
True
sage: octant = Cone([(1,0,0), (0,1,0), (0,0,1)])
sage: octant.is_solid()
True
```

```
>>> from sage.all import *
>>> quadrant = cones.nonnegative_orthant(Integer(2))
>>> quadrant.is_solid()
True
>>> octant = Cone([(Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0)), (Integer(0),Integer(0),Integer(1))])
>>> octant.is_solid()
True
```

However, if we embed the two-dimensional nonnegative quadrant into three-dimensional space, then the resulting cone no longer has interior, so it is not solid:

```
sage: quadrant = Cone([(1,0,0), (0,1,0)])
sage: quadrant.is_solid()
False
```

```
>>> from sage.all import *
>>> quadrant = Cone([(Integer(1),Integer(0),Integer(0)), (Integer(0),
↪Integer(1),Integer(0))])
>>> quadrant.is_solid()
False
```

is_strictly_convex()

Check if `self` is strictly convex.

A cone is called **strictly convex** if it does not contain any lines.

OUTPUT: `True` if `self` is strictly convex, `False` otherwise

EXAMPLES:

```
sage: cone1 = Cone([(1,0), (0, 1)])
sage: cone2 = Cone([(1,0), (-1, 0)])
sage: cone1.is_strictly_convex()
True
sage: cone2.is_strictly_convex()
False
```

```
>>> from sage.all import *
>>> cone1 = Cone([(Integer(1),Integer(0)), (Integer(0), Integer(1))])
>>> cone2 = Cone([(Integer(1),Integer(0)), (-Integer(1), Integer(0))])
>>> cone1.is_strictly_convex()
True
>>> cone2.is_strictly_convex()
False
```

is_trivial()

Check if the cone has no rays.

OUTPUT: `True` if the cone has no rays, `False` otherwise

EXAMPLES:

```
sage: c0 = cones.trivial(3)
sage: c0.is_trivial()
True
sage: c0.nrays()
0
```

```
>>> from sage.all import *
>>> c0 = cones.trivial(Integer(3))
>>> c0.is_trivial()
True
>>> c0.nrays()
0
```

is_universe()

Check if this cone is equal to its ambient vector space.

An alias is *is_universe()*.

OUTPUT:

`True` if this cone equals its entire ambient vector space and `False` otherwise.

EXAMPLES:

A single ray in two dimensions is not equal to the entire space:

```
sage: K = Cone([(1,0)])
sage: K.is_full_space()
False
```

```
>>> from sage.all import *
>>> K = Cone([(Integer(1),Integer(0))])
>>> K.is_full_space()
False
```

Neither is the nonnegative orthant:

```
sage: K = cones.nonnegative_orthant(2)
sage: K.is_full_space()
False
```

```
>>> from sage.all import *
>>> K = cones.nonnegative_orthant(Integer(2))
>>> K.is_full_space()
False
```

The right half-space contains a vector subspace, but it is still not equal to the entire space:

```
sage: K = Cone([(1,0), (-1,0), (0,1)])
sage: K.is_full_space()
False
```

```
>>> from sage.all import *
>>> K = Cone([(Integer(1),Integer(0)), (-Integer(1),Integer(0)), (Integer(0),
↪Integer(1))])
>>> K.is_full_space()
False
```

However, if we allow conic combinations of both axes, then the resulting cone is the entire two-dimensional space:

```
sage: K = Cone([(1,0), (-1,0), (0,1), (0,-1)])
sage: K.is_full_space()
True
```

```
>>> from sage.all import *
>>> K = Cone([(Integer(1),Integer(0)), (-Integer(1),Integer(0)), (Integer(0),
↪Integer(1)), (Integer(0),-Integer(1))])
```

(continues on next page)

(continued from previous page)

```
>>> K.is_full_space()
True
```

lineality()

Return the lineality of this cone.

The lineality of a cone is the dimension of the largest linear subspace contained in that cone.

OUTPUT:

A nonnegative integer; the dimension of the largest subspace contained within this cone.

REFERENCES:

- [Roc1970]

EXAMPLES:

The lineality of the nonnegative orthant is zero, since it clearly contains no lines:

```
sage: K = cones.nonnegative_orthant(3)
sage: K.lineality()
0
```

```
>>> from sage.all import *
>>> K = cones.nonnegative_orthant(Integer(3))
>>> K.lineality()
0
```

However, if we add another ray so that the entire x-axis belongs to the cone, then the resulting cone will have lineality one:

```
sage: K = Cone([(1,0,0), (-1,0,0), (0,1,0), (0,0,1)])
sage: K.lineality()
1
```

```
>>> from sage.all import *
>>> K = Cone([(Integer(1),Integer(0),Integer(0)), (-Integer(1),Integer(0),
↪Integer(0)), (Integer(0),Integer(1),Integer(0)), (Integer(0),Integer(0),
↪Integer(1))])
>>> K.lineality()
1
```

If our cone is all of $\mathbb{R}^2$, then its lineality is equal to the dimension of the ambient space (i.e. two):

```
sage: K = Cone([(1,0), (-1,0), (0,1), (0,-1)])
sage: K.is_full_space()
True
sage: K.lineality()
2
sage: K.lattice_dim()
2
```

```
>>> from sage.all import *
>>> K = Cone([(Integer(1),Integer(0)), (-Integer(1),Integer(0)), (Integer(0),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1)), (Integer(0),-Integer(1))])
>>> K.is_full_space()
True
>>> K.lineality()
2
>>> K.lattice_dim()
2
```

Per the definition, the lineality of the trivial cone in a trivial space is zero:

```
sage: K = cones.trivial(0)
sage: K.lineality()
0
```

```
>>> from sage.all import *
>>> K = cones.trivial(Integer(0))
>>> K.lineality()
0
```

linear_subspace()

Return the largest linear subspace contained inside of `self`.

OUTPUT: subspace of the ambient space of `self`

EXAMPLES:

```
sage: halfplane = Cone([(1,0), (0,1), (-1,0)])
sage: halfplane.linear_subspace()
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 0]
```

```
>>> from sage.all import *
>>> halfplane = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1)), (-
↪Integer(1),Integer(0))])
>>> halfplane.linear_subspace()
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 0]
```

lines()

Return lines generating the linear subspace of `self`.

OUTPUT:

- `tuple` of primitive vectors in the lattice of `self` giving directions of lines that span the linear subspace of `self`. These lines are arbitrary, but fixed. If you do not care about the order, see also `line_set()`.

EXAMPLES:

```
sage: halfplane = Cone([(1,0), (0,1), (-1,0)])
sage: halfplane.lines()
N(1, 0)
in 2-d lattice N
```

(continues on next page)

(continued from previous page)

```
sage: fullplane = Cone([(1,0), (0,1), (-1,-1)])
sage: fullplane.lines()
N(0, 1),
N(1, 0)
in 2-d lattice N
```

```
>>> from sage.all import *
>>> halfplane = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1)), (-
↪Integer(1),Integer(0))])
>>> halfplane.lines()
N(1, 0)
in 2-d lattice N
>>> fullplane = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1)), (-
↪Integer(1),-Integer(1))])
>>> fullplane.lines()
N(0, 1),
N(1, 0)
in 2-d lattice N
```

lyapunov_like_basis()

Compute a basis of Lyapunov-like transformations on this cone.

A linear transformation L is said to be Lyapunov-like on this cone if $L(x)$ and s are orthogonal for every pair (x, s) in its *discrete_complementarity_set()*. The set of all such transformations forms a vector space, namely the Lie algebra of the automorphism group of this cone.

OUTPUT:

A list of matrices forming a basis for the space of all Lyapunov-like transformations on this cone.

See also

cross_positive_operators_gens(), *positive_operators_gens()*, *Z_operators_gens()*

REFERENCES:

- [Or2017]
- [RNPA2011]

EXAMPLES:

Every transformation is Lyapunov-like on the trivial cone:

```
sage: K = cones.trivial(2)
sage: M = MatrixSpace(K.lattice().base_field(), K.lattice_dim())
sage: list(M.basis()) == K.lyapunov_like_basis()
True
```

```
>>> from sage.all import *
>>> K = cones.trivial(Integer(2))
>>> M = MatrixSpace(K.lattice().base_field(), K.lattice_dim())
```

(continues on next page)

(continued from previous page)

```
>>> list(M.basis()) == K.lyapunov_like_basis()
True
```

And by duality, every transformation is Lyapunov-like on the ambient space:

```
sage: K = Cone([(1,0), (-1,0), (0,1), (0,-1)])
sage: K.is_full_space()
True
sage: M = MatrixSpace(K.lattice().base_field(), K.lattice_dim())
sage: list(M.basis()) == K.lyapunov_like_basis()
True
```

```
>>> from sage.all import *
>>> K = Cone([(Integer(1),Integer(0)), (-Integer(1),Integer(0)), (Integer(0),
↪Integer(1)), (Integer(0),-Integer(1))])
>>> K.is_full_space()
True
>>> M = MatrixSpace(K.lattice().base_field(), K.lattice_dim())
>>> list(M.basis()) == K.lyapunov_like_basis()
True
```

However, in a trivial space, there are no non-trivial linear maps, so there can be no Lyapunov-like basis:

```
sage: K = cones.trivial(0)
sage: K.lyapunov_like_basis()
[]
```

```
>>> from sage.all import *
>>> K = cones.trivial(Integer(0))
>>> K.lyapunov_like_basis()
[]
```

The Lyapunov-like transformations on the nonnegative orthant are diagonal matrices:

```
sage: K = cones.nonnegative_orthant(1)
sage: K.lyapunov_like_basis()
[[1]]

sage: K = cones.nonnegative_orthant(2)
sage: K.lyapunov_like_basis()
[
[1 0]  [0 0]
[0 0], [0 1]
]

sage: K = cones.nonnegative_orthant(3)
sage: K.lyapunov_like_basis()
[
[1 0 0]  [0 0 0]  [0 0 0]
[0 0 0]  [0 1 0]  [0 0 0]
[0 0 0], [0 0 0], [0 0 1]
]
```

```
>>> from sage.all import *
>>> K = cones.nonnegative_orthant(Integer(1))
>>> K.lyapunov_like_basis()
[[1]]

>>> K = cones.nonnegative_orthant(Integer(2))
>>> K.lyapunov_like_basis()
[
[1 0]  [0 0]
[0 0], [0 1]
]

>>> K = cones.nonnegative_orthant(Integer(3))
>>> K.lyapunov_like_basis()
[
[1 0 0]  [0 0 0]  [0 0 0]
[0 0 0]  [0 1 0]  [0 0 0]
[0 0 0], [0 0 0], [0 0 1]
]
```

Only the identity matrix is Lyapunov-like on the pyramids defined by the one- and infinity-norms [RNPA2011]:

```
sage: l31 = Cone([(1,0,1), (0,-1,1), (-1,0,1), (0,1,1)])
sage: l31.lyapunov_like_basis()
[
[1 0 0]
[0 1 0]
[0 0 1]
]

sage: l3infty = Cone([(0,1,1), (1,0,1), (0,-1,1), (-1,0,1)])
sage: l3infty.lyapunov_like_basis()
[
[1 0 0]
[0 1 0]
[0 0 1]
]
```

```
>>> from sage.all import *
>>> l31 = Cone([(Integer(1),Integer(0),Integer(1)), (Integer(0),-Integer(1),
↪Integer(1)), (-Integer(1),Integer(0),Integer(1)), (Integer(0),Integer(1),
↪Integer(1))])
>>> l31.lyapunov_like_basis()
[
[1 0 0]
[0 1 0]
[0 0 1]
]

>>> l3infty = Cone([(Integer(0),Integer(1),Integer(1)), (Integer(1),
↪Integer(0),Integer(1)), (Integer(0),-Integer(1),Integer(1)), (-Integer(1),
↪Integer(0),Integer(1))])
```

(continues on next page)

(continued from previous page)

```
>>> l3infty.lyapunov_like_basis()
[
[1 0 0]
[0 1 0]
[0 0 1]
]
```

lyapunov_rank()

Compute the Lyapunov rank of this cone.

The Lyapunov rank of a cone is the dimension of the space of its Lyapunov-like transformations — that is, the length of a *lyapunov_like_basis()*. Equivalently, the Lyapunov rank is the dimension of the Lie algebra of the automorphism group of the cone.

OUTPUT: nonnegative integer representing the Lyapunov rank of this cone

If the ambient space is trivial, then the Lyapunov rank will be zero. On the other hand, if the dimension of the ambient vector space is $n > 0$, then the resulting Lyapunov rank will be between 1 and n^2 inclusive. If this cone *is_proper()*, then that upper bound reduces from n^2 to n. A Lyapunov rank of $n - 1$ is not possible (by Lemma 6 [Or2017]) in either case.

ALGORITHM:

Algorithm 3 [Or2017] is used. Every closed convex cone is isomorphic to a Cartesian product of a proper cone, a subspace, and a trivial cone. The Lyapunov ranks of the subspace and trivial cone are easy to compute. Essentially, we "peel off" those easy parts of the cone and compute their Lyapunov ranks separately. We then compute the rank of the proper cone by counting a *lyapunov_like_basis()* for it. Summing the individual ranks gives the Lyapunov rank of the original cone.

REFERENCES:

- [GT2014]
- [Or2017]
- [RNPA2011]

EXAMPLES:

The Lyapunov rank of the nonnegative orthant is the same as the dimension of the ambient space [RNPA2011]:

```
sage: positives = cones.nonnegative_orthant(1)
sage: positives.lyapunov_rank()
1
sage: quadrant = cones.nonnegative_orthant(2)
sage: quadrant.lyapunov_rank()
2
sage: octant = cones.nonnegative_orthant(3)
sage: octant.lyapunov_rank()
3
```

```
>>> from sage.all import *
>>> positives = cones.nonnegative_orthant(Integer(1))
>>> positives.lyapunov_rank()
1
>>> quadrant = cones.nonnegative_orthant(Integer(2))
```

(continues on next page)

(continued from previous page)

```
>>> quadrant.lyapunov_rank()
2
>>> octant = cones.nonnegative_orthant(Integer(3))
>>> octant.lyapunov_rank()
3
```

A vector space of dimension n has Lyapunov rank n^2 [Or2017]:

```
sage: Q5 = VectorSpace(QQ, 5)
sage: gs = Q5.basis() + [-r for r in Q5.basis()]
sage: K = Cone(gs)
sage: K.lyapunov_rank()
25
```

```
>>> from sage.all import *
>>> Q5 = VectorSpace(QQ, Integer(5))
>>> gs = Q5.basis() + [-r for r in Q5.basis()]
>>> K = Cone(gs)
>>> K.lyapunov_rank()
25
```

A pyramid in three dimensions has Lyapunov rank one [RNPA2011]:

```
sage: l31 = Cone([(1,0,1), (0,-1,1), (-1,0,1), (0,1,1)])
sage: l31.lyapunov_rank()
1
sage: l3infty = Cone([(0,1,1), (1,0,1), (0,-1,1), (-1,0,1)])
sage: l3infty.lyapunov_rank()
1
```

```
>>> from sage.all import *
>>> l31 = Cone([(Integer(1),Integer(0),Integer(1)), (Integer(0),-Integer(1),
↪Integer(1)), (-Integer(1),Integer(0),Integer(1)), (Integer(0),Integer(1),
↪Integer(1))])
>>> l31.lyapunov_rank()
1
>>> l3infty = Cone([(Integer(0),Integer(1),Integer(1)), (Integer(1),
↪Integer(0),Integer(1)), (Integer(0),-Integer(1),Integer(1)), (-Integer(1),
↪Integer(0),Integer(1))])
>>> l3infty.lyapunov_rank()
1
```

A ray in n dimensions has Lyapunov rank $n^2 - n + 1$ [Or2017]:

```
sage: K = Cone([(1,0,0,0,0)])
sage: K.lyapunov_rank()
21
sage: K.lattice_dim()**2 - K.lattice_dim() + 1
21
```

```
>>> from sage.all import *
>>> K = Cone([(Integer(1),Integer(0),Integer(0),Integer(0),Integer(0))])
```

(continues on next page)

(continued from previous page)

```
>>> K.lyapunov_rank()
21
>>> K.lattice_dim()**Integer(2) - K.lattice_dim() + Integer(1)
21
```

A subspace of dimension m in an n-dimensional ambient space has Lyapunov rank $n^2 - m(n-m)$ [Or2017]:

```
sage: e1 = vector(QQ, [1,0,0,0,0])
sage: e2 = vector(QQ, [0,1,0,0,0])
sage: z = (0,0,0,0,0)
sage: K = Cone([e1, -e1, e2, -e2, z, z, z])
sage: K.lyapunov_rank()
19
sage: K.lattice_dim()**2 - K.dim()*K.codim()
19
```

```
>>> from sage.all import *
>>> e1 = vector(QQ, [Integer(1),Integer(0),Integer(0),Integer(0),Integer(0)])
>>> e2 = vector(QQ, [Integer(0),Integer(1),Integer(0),Integer(0),Integer(0)])
>>> z = (Integer(0),Integer(0),Integer(0),Integer(0),Integer(0))
>>> K = Cone([e1, -e1, e2, -e2, z, z, z])
>>> K.lyapunov_rank()
19
>>> K.lattice_dim()**Integer(2) - K.dim()*K.codim()
19
```

Lyapunov rank is additive on a product of proper cones [RNPA2011]:

```
sage: l31 = Cone([(1,0,1), (0,-1,1), (-1,0,1), (0,1,1)])
sage: octant = Cone([(1,0,0), (0,1,0), (0,0,1)])
sage: K = l31.cartesian_product(octant)
sage: K.lyapunov_rank()
4
sage: l31.lyapunov_rank() + octant.lyapunov_rank()
4
```

```
>>> from sage.all import *
>>> l31 = Cone([(Integer(1),Integer(0),Integer(1)), (Integer(0),-Integer(1),
↪Integer(1)), (-Integer(1),Integer(0),Integer(1)), (Integer(0),Integer(1),
↪Integer(1))])
>>> octant = Cone([(Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0)), (Integer(0),Integer(0),Integer(1))])
>>> K = l31.cartesian_product(octant)
>>> K.lyapunov_rank()
4
>>> l31.lyapunov_rank() + octant.lyapunov_rank()
4
```

Two linearly-isomorphic cones have the same Lyapunov rank [RNPA2011]. A cone linearly-isomorphic to the nonnegative octant will have Lyapunov rank 3:

```
sage: K = Cone([(1,2,3), (-1,1,0), (1,0,6)])
sage: K.lyapunov_rank()
3
```

```
>>> from sage.all import *
>>> K = Cone([(Integer(1),Integer(2),Integer(3)), (-Integer(1),Integer(1),
↪Integer(0)), (Integer(1),Integer(0),Integer(6))])
>>> K.lyapunov_rank()
3
```

Lyapunov rank is invariant under `dual()` [RNPA2011]:

```
sage: K = Cone([(2,2,4), (-1,9,0), (2,0,6)])
sage: K.lyapunov_rank() == K.dual().lyapunov_rank()
True
```

```
>>> from sage.all import *
>>> K = Cone([(Integer(2),Integer(2),Integer(4)), (-Integer(1),Integer(9),
↪Integer(0)), (Integer(2),Integer(0),Integer(6))])
>>> K.lyapunov_rank() == K.dual().lyapunov_rank()
True
```

max_angle(*other=None*, *exact=True*, *epsilon=0*)

Return the maximal angle between `self` and `other`.

The maximal angle between two closed convex cones is the unique largest angle formed by any two unit-norm vectors in those cones. In pathological cases, this computation can fail.

If it fails when `exact` is `True` and if each of the cones `is_strictly_convex()`, then a second attempt will be made using inexact arithmetic. (This sometimes avoids the problem noted in [Or2024]). If the computation fails when the cones are not strictly convex or when `exact` is `False`, a `ValueError` is raised.

INPUT:

- `other` – (default: `None`) a rational, polyhedral convex cone
- `exact` – boolean (default: `True`); whether or not to use exact rational arithmetic instead of floating point computations; beware that `True` is not guaranteed to avoid floating point computations if the algorithm runs into trouble in rational arithmetic
- `epsilon` – (default: `0`) the tolerance to use when making comparisons

Warning

Using inexact arithmetic (`exact=False`) is faster, but this computation is only known to be stable when both of the cones are strictly convex (or when one of them is the entire space, but the maximal angle is obviously π in that case).

OUTPUT:

A triple $(\theta_{\text{max}}, u, v)$ containing:

- the maximal angle θ_{max} between `self` and `other`
- a vector u in `self` that achieves the maximal angle
- a vector v in `other` that achieves the maximal angle

If `other` is `None` (the default), then the maximal angle within this cone (between this cone and itself) is returned.

If an eigenspace of dimension greater than one is encountered and if the corresponding angle cannot be ruled out as a maximum, the behavior of this function depends on `exact`:

- If `exact` is `True` and if both `self` and `other` are strictly convex, then the algorithm will fall back to inexact arithmetic. In that case, the returned angle and vectors will be over `sage.rings.real_double.RDF`.
- If `exact` is `False` or if either cone is not strictly convex, then a `ValueError` is raised to indicate that we have failed; i.e. we cannot say with certainty what the maximal angle is.

REFERENCES:

- [IS2005]
- [SS2016]
- [Or2020]
- [Or2024]

ALGORITHM:

Algorithm 3 in [Or2020] is used. If a potentially-maximal angle corresponds to an eigenspace of dimension two or more, we sometimes fall back to inexact arithmetic which has the effect of perturbing the cones. That this will not affect the answer too much is one conclusion of [Or2024].

EXAMPLES:

The maximal angle in a single ray is zero:

```
sage: K = random_cone(min_rays=1, max_rays=1, max_ambient_dim=5)
sage: K.max_angle()[0]
0
```

```
>>> from sage.all import *
>>> K = random_cone(min_rays=Integer(1), max_rays=Integer(1), max_ambient_
↪dim=Integer(5))
>>> K.max_angle()[Integer(0)]
0
```

The maximal angle in the nonnegative octant is $\pi/2$:

```
sage: K = cones.nonnegative_orthant(3)
sage: K.max_angle()[0]
1/2*pi
```

```
>>> from sage.all import *
>>> K = cones.nonnegative_orthant(Integer(3))
>>> K.max_angle()[Integer(0)]
1/2*pi
```

The maximal angle between the nonnegative quintant and the Schur cone of dimension 5 is about 0.8524π. The same result can be obtained faster using inexact arithmetic, but only confidently so because we already know the answer:

```
sage: # long time
sage: P = cones.nonnegative_orthant(5)
sage: Q = cones.schur(5)
sage: actual = P.max_angle(Q)[0]
sage: expected = 0.8524*pi
sage: bool( (actual - expected).abs() < 0.0001 )
True
sage: actual = P.max_angle(Q,exact=False)[0]
sage: bool( (actual - expected).abs() < 0.0001 )
True
```

```
>>> from sage.all import *
>>> # long time
>>> P = cones.nonnegative_orthant(Integer(5))
>>> Q = cones.schur(Integer(5))
>>> actual = P.max_angle(Q)[Integer(0)]
>>> expected = RealNumber('0.8524')*pi
>>> bool( (actual - expected).abs() < RealNumber('0.0001') )
True
>>> actual = P.max_angle(Q,exact=False)[Integer(0)]
>>> bool( (actual - expected).abs() < RealNumber('0.0001') )
True
```

The maximal angle within the Schur cone is known explicitly via Gourion and Seeger's Proposition 2 [GS2010]:

```
sage: n = 3
sage: K = cones.schur(n)
sage: bool(K.max_angle()[0] == ((n-1)/n)*pi)
True
```

```
>>> from sage.all import *
>>> n = Integer(3)
>>> K = cones.schur(n)
>>> bool(K.max_angle()[Integer(0)] == ((n-Integer(1))/n)*pi)
True
```

Sage can't prove that the actual and expected results are equal in the next two cases without a little nudge in the right direction, and, moreover, it's crashy about it:

```
sage: n = 4
sage: K = cones.schur(n)
sage: actual = K.max_angle()[0].simplify()._sympy_()._sage_()
sage: expected = ((n-1)/n)*pi
sage: bool( actual == expected )
True
sage: n = 5
sage: K = cones.schur(n)
sage: actual = K.max_angle()[0].simplify()._sympy_()._sage_()
sage: expected = ((n-1)/n)*pi
sage: bool( actual == expected )
True
```

```
>>> from sage.all import *
>>> n = Integer(4)
>>> K = cones.schur(n)
>>> actual = K.max_angle()[Integer(0)].simplify()._sympy_()._sage_()
>>> expected = ((n-Integer(1))/n)*pi
>>> bool( actual == expected )
True
>>> n = Integer(5)
>>> K = cones.schur(n)
>>> actual = K.max_angle()[Integer(0)].simplify()._sympy_()._sage_()
>>> expected = ((n-Integer(1))/n)*pi
>>> bool( actual == expected )
True
```

When there's a unit norm vector in `self` whose negation is in `other`, they form a maximal angle of π:

```
sage: P = Cone([(5,1), (1,-1)])
sage: Q = Cone([(-1,0), (-1,0)])
sage: P.max_angle(Q)[0]
pi
```

```
>>> from sage.all import *
>>> P = Cone([(Integer(5),Integer(1)), (Integer(1),-Integer(1))])
>>> Q = Cone([(-Integer(1),Integer(0)), (-Integer(1),Integer(0))])
>>> P.max_angle(Q)[Integer(0)]
pi
```

orthogonal_sublattice(**args*, ***kwds*)

The sublattice (in the dual lattice) orthogonal to the sublattice spanned by the cone.

Let $M =$ `self.dual_lattice()` be the lattice dual to the ambient lattice of the given cone σ. Then, in the notation of [Ful1993], this method returns the sublattice

$$M(\sigma) \stackrel{\text{def}}{=} \sigma^{\perp} \cap M \subset M$$

INPUT:

- either nothing or something that can be turned into an element of this lattice.

OUTPUT:

- if no arguments were given, a `toric sublattice`, otherwise the corresponding element of it.

EXAMPLES:

```
sage: c = Cone([(1,1,1), (1,-1,1), (-1,-1,1), (-1,1,1)])
sage: c.orthogonal_sublattice()
Sublattice <>
sage: c12 = Cone([(1,1,1), (1,-1,1)])
sage: c12.sublattice()
Sublattice <N(1, 1, 1), N(0, -1, 0)>
sage: c12.orthogonal_sublattice()
Sublattice <M(1, 0, -1)>
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(1),Integer(1),Integer(1)), (Integer(1),-Integer(1),
↪Integer(1)), (-Integer(1),-Integer(1),Integer(1)), (-Integer(1),Integer(1),
↪Integer(1))])
>>> c.orthogonal_sublattice()
Sublattice <>
>>> c12 = Cone([(Integer(1),Integer(1),Integer(1)), (Integer(1),-Integer(1),
↪Integer(1))])
>>> c12.sublattice()
Sublattice <N(1, 1, 1), N(0, -1, 0)>
>>> c12.orthogonal_sublattice()
Sublattice <M(1, 0, -1)>
```

plot(***options*)

Plot `self`.

INPUT:

- any options for toric plots (see *toric_plotter.options*), none are mandatory

OUTPUT: a plot

EXAMPLES:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: quadrant.plot()                                                    #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 9 graphics primitives
```

```
>>> from sage.all import *
>>> quadrant = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> quadrant.plot()                                                      #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 9 graphics primitives
```

polyhedron(***kwds*)

Return the polyhedron associated to `self`.

Mathematically this polyhedron is the same as `self`.

OUTPUT: *Polyhedron_base*

EXAMPLES:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: quadrant.polyhedron()
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull
of 1 vertex and 2 rays
sage: line = Cone([(1,0), (-1,0)])
sage: line.polyhedron()
A 1-dimensional polyhedron in ZZ^2 defined as the convex hull
of 1 vertex and 1 line
```

```
>>> from sage.all import *
>>> quadrant = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
```

(continues on next page)

(continued from previous page)

```
>>> quadrant.polyhedron()
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull
of 1 vertex and 2 rays
>>> line = Cone([(Integer(1),Integer(0)), (-Integer(1),Integer(0))])
>>> line.polyhedron()
A 1-dimensional polyhedron in ZZ^2 defined as the convex hull
of 1 vertex and 1 line
```

Here is an example of a trivial cone (see Issue #10237):

```
sage: origin = Cone([], lattice=ZZ^2)
sage: origin.polyhedron()
A 0-dimensional polyhedron in ZZ^2 defined as the convex hull
of 1 vertex
```

```
>>> from sage.all import *
>>> origin = Cone([], lattice=ZZ**Integer(2))
>>> origin.polyhedron()
A 0-dimensional polyhedron in ZZ^2 defined as the convex hull
of 1 vertex
```

positive_operators_gens(*K2=None*)

Compute minimal generators of the positive operators on this cone.

A linear operator on a cone is positive if the image of the cone under the operator is a subset of the cone. This concept can be extended to two cones: the image of the first cone under a positive operator is a subset of the second cone, which may live in a different space.

The positive operators (on one or two fixed cones) themselves form a closed convex cone. This method computes and returns the generators of that cone as a list of matrices.

INPUT:

- `K2` – (default: `self`) the codomain cone; the image of this cone under the returned generators is a subset of `K2`

OUTPUT:

A list of m-by-n matrices where m is the ambient dimension of `K2` and n is the ambient dimension of this cone. Each matrix P in the list has the property that $P(x)$ is an element of `K2` whenever x is an element of this cone.

The returned matrices generate the cone of positive operators from this cone to `K2`; that is,

- Any nonnegative linear combination of the returned matrices sends elements of this cone to `K2`.
- Every positive operator on this cone (with respect to `K2`) is some nonnegative linear combination of the returned matrices.

ALGORITHM:

Computing positive operators directly is difficult, but computing their dual is straightforward using the generators of Berman and Gaiha. We construct the dual of the positive operators, and then return the dual of that, which is guaranteed to be the desired positive operators because everything is closed, convex, and polyhedral.

See also

cross_positive_operators_gens(), *lyapunov_like_basis()*, *Z_operators_gens()*

REFERENCES:

- [BG1972]
- [BP1994]
- [Or2018b]

EXAMPLES:

Positive operators on the nonnegative orthant are nonnegative matrices:

```
sage: K = Cone([(1,)])
sage: K.positive_operators_gens()
[[1]]

sage: K = Cone([(1,0), (0,1)])
sage: K.positive_operators_gens()
[
[1 0]  [0 1]  [0 0]  [0 0]
[0 0], [0 0], [1 0], [0 1]
]
```

```
>>> from sage.all import *
>>> K = Cone([(Integer(1),)])
>>> K.positive_operators_gens()
[[1]]

>>> K = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> K.positive_operators_gens()
[
[1 0]  [0 1]  [0 0]  [0 0]
[0 0], [0 0], [1 0], [0 1]
]
```

The trivial cone in a trivial space has no positive operators:

```
sage: K = cones.trivial(0)
sage: K.positive_operators_gens()
[]
```

```
>>> from sage.all import *
>>> K = cones.trivial(Integer(0))
>>> K.positive_operators_gens()
[]
```

Every operator is positive on the trivial cone:

```
sage: K = cones.trivial(1)
sage: K.positive_operators_gens()
```

(continues on next page)

(continued from previous page)

```
[[1], [-1]]

sage: K = cones.trivial(2)
sage: K.is_trivial()
True
sage: K.positive_operators_gens()
[
[1 0]   [-1  0]   [0 1]   [ 0 -1]   [0 0]   [ 0  0]   [0 0]   [ 0  0]
[0 0], [ 0  0], [0 0], [ 0  0], [1 0], [-1  0], [0 1], [ 0 -1]
]
```

```
>>> from sage.all import *
>>> K = cones.trivial(Integer(1))
>>> K.positive_operators_gens()
[[1], [-1]]

>>> K = cones.trivial(Integer(2))
>>> K.is_trivial()
True
>>> K.positive_operators_gens()
[
[1 0]   [-1  0]   [0 1]   [ 0 -1]   [0 0]   [ 0  0]   [0 0]   [ 0  0]
[0 0], [ 0  0], [0 0], [ 0  0], [1 0], [-1  0], [0 1], [ 0 -1]
]
```

Every operator is positive on the ambient vector space:

```
sage: K = Cone([(1,), (-1,)])
sage: K.is_full_space()
True
sage: K.positive_operators_gens()
[[1], [-1]]

sage: K = Cone([(1,0), (-1,0), (0,1), (0,-1)])
sage: K.is_full_space()
True
sage: K.positive_operators_gens()
[
[1 0]   [-1  0]   [0 1]   [ 0 -1]   [0 0]   [ 0  0]   [0 0]   [ 0  0]
[0 0], [ 0  0], [0 0], [ 0  0], [1 0], [-1  0], [0 1], [ 0 -1]
]
```

```
>>> from sage.all import *
>>> K = Cone([(Integer(1),), (-Integer(1),)])
>>> K.is_full_space()
True
>>> K.positive_operators_gens()
[[1], [-1]]

>>> K = Cone([(Integer(1),Integer(0)), (-Integer(1),Integer(0)), (Integer(0),
↪Integer(1)), (Integer(0),-Integer(1))])
>>> K.is_full_space()
```

(continues on next page)

(continued from previous page)

```
True
>>> K.positive_operators_gens()
[
[1 0]  [-1  0]  [0 1]  [ 0 -1]  [0 0]  [ 0  0]  [0 0]  [ 0  0]
[0 0], [ 0  0], [0 0], [ 0  0], [1 0], [-1  0], [0 1], [ 0 -1]
]
```

A non-obvious application is to find the positive operators on the right half-plane [Or2018b]:

```
sage: K = Cone([(1,0), (0,1), (0,-1)])
sage: K.positive_operators_gens()
[
[1 0]  [0 0]  [ 0  0]  [0 0]  [ 0  0]
[0 0], [1 0], [-1  0], [0 1], [ 0 -1]
]
```

```
>>> from sage.all import *
>>> K = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1)), (Integer(0),-
↪Integer(1))])
>>> K.positive_operators_gens()
[
[1 0]  [0 0]  [ 0  0]  [0 0]  [ 0  0]
[0 0], [1 0], [-1  0], [0 1], [ 0 -1]
]
```

random_element(*ring=Integer Ring*)

Return a random element of this cone.

All elements of a convex cone can be represented as a nonnegative linear combination of its generators. A random element is thus constructed by assigning random nonnegative weights to the generators of this cone. By default, these weights are integral and the resulting random element will live in the same lattice as the cone.

The random nonnegative weights are chosen from `ring` which defaults to `ZZ`. When `ring` is not `ZZ`, the random element returned will be a vector. Only the rings `ZZ` and `QQ` are currently supported.

INPUT:

- `ring` – (default: `ZZ`) the ring from which the random generator weights are chosen; either `ZZ` or `QQ`

OUTPUT:

Either a lattice element or vector contained in both this cone and its ambient vector space. If `ring` is `ZZ`, a lattice element is returned; otherwise a vector is returned. If `ring` is neither `ZZ` nor `QQ`, then a `NotImplementedError` is raised.

EXAMPLES:

The trivial element `()` is always returned in a trivial space:

```
sage: K = cones.trivial(0)
sage: K.random_element()
N()
sage: K.random_element(ring=QQ)
()
```

```
>>> from sage.all import *
>>> K = cones.trivial(Integer(0))
>>> K.random_element()
N()
>>> K.random_element(ring=QQ)
()
```

A random element of the trivial cone in a nontrivial space is zero:

```
sage: K = cones.trivial(3)
sage: K.random_element()
N(0, 0, 0)
sage: K.random_element(ring=QQ)
(0, 0, 0)
```

```
>>> from sage.all import *
>>> K = cones.trivial(Integer(3))
>>> K.random_element()
N(0, 0, 0)
>>> K.random_element(ring=QQ)
(0, 0, 0)
```

A random element of the nonnegative orthant should have all components nonnegative:

```
sage: K = cones.nonnegative_orthant(3)
sage: all(x >= 0 for x in K.random_element())
True
sage: all(x >= 0 for x in K.random_element(ring=QQ))
True
```

```
>>> from sage.all import *
>>> K = cones.nonnegative_orthant(Integer(3))
>>> all(x >= Integer(0) for x in K.random_element())
True
>>> all(x >= Integer(0) for x in K.random_element(ring=QQ))
True
```

If `ring` is not `ZZ` or `QQ`, an error is raised:

```
sage: K = Cone([(1,0), (0,1)])
sage: K.random_element(ring=RR)
Traceback (most recent call last):
...
NotImplementedError: ring must be either ZZ or QQ.
```

```
>>> from sage.all import *
>>> K = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> K.random_element(ring=RR)
Traceback (most recent call last):
...
NotImplementedError: ring must be either ZZ or QQ.
```

relative_interior()

Return the relative interior of `self`.

OUTPUT:

- either `self` or an instance of *RelativeInterior*.

EXAMPLES:

```
sage: c = Cone([(1,0,0), (0,1,0)]); c
2-d cone in 3-d lattice N
sage: c.relative_interior()
Relative interior of 2-d cone in 3-d lattice N

sage: origin = cones.trivial(2); origin
0-d cone in 2-d lattice N
sage: origin.relative_interior() is origin
True

sage: K1 = Cone([(1,0), (-1,0)]); K1
1-d cone in 2-d lattice N
sage: K1.relative_interior() is K1
True

sage: K2 = Cone([(1,0),(-1,0),(0,1),(0,-1)]); K2
2-d cone in 2-d lattice N
sage: K2.relative_interior() is K2
True
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0))]); c
2-d cone in 3-d lattice N
>>> c.relative_interior()
Relative interior of 2-d cone in 3-d lattice N

>>> origin = cones.trivial(Integer(2)); origin
0-d cone in 2-d lattice N
>>> origin.relative_interior() is origin
True

>>> K1 = Cone([(Integer(1),Integer(0)), (-Integer(1),Integer(0))]); K1
1-d cone in 2-d lattice N
>>> K1.relative_interior() is K1
True

>>> K2 = Cone([(Integer(1),Integer(0)),(-Integer(1),Integer(0)),(Integer(0),
↪Integer(1)),(Integer(0),-Integer(1))]); K2
2-d cone in 2-d lattice N
>>> K2.relative_interior() is K2
True
```

relative_interior_contains(**args*)

Check if a given point is contained in the relative interior of `self`.

For a full-dimensional cone the relative interior is simply the interior, so this method will do the same check as *interior_contains()*. For a strictly lower-dimensional cone, the relative interior is the cone without its facets.

INPUT:

- anything. An attempt will be made to convert all arguments into a single element of the ambient space of `self`. If it fails, `False` will be returned.

OUTPUT:

- `True` if the given point is contained in the relative interior of `self`, `False` otherwise.

EXAMPLES:

```
sage: c = Cone([(1,0,0), (0,1,0)])
sage: c.contains((1,1,0))
True
sage: c.relative_interior_contains((1,1,0))
True
sage: c.interior_contains((1,1,0))
False
sage: c.contains((1,0,0))
True
sage: c.relative_interior_contains((1,0,0))
False
sage: c.interior_contains((1,0,0))
False
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0))])
>>> c.contains((Integer(1),Integer(1),Integer(0)))
True
>>> c.relative_interior_contains((Integer(1),Integer(1),Integer(0)))
True
>>> c.interior_contains((Integer(1),Integer(1),Integer(0)))
False
>>> c.contains((Integer(1),Integer(0),Integer(0)))
True
>>> c.relative_interior_contains((Integer(1),Integer(0),Integer(0)))
False
>>> c.interior_contains((Integer(1),Integer(0),Integer(0)))
False
```

relative_orthogonal_quotient(*supercone*)

The quotient of the dual spanned lattice by the dual of the supercone's spanned lattice.

In the notation of [Ful1993], if `supercone` = $\rho > \sigma$ = `self` is a cone that contains σ as a face, then $M(\rho)$ = `supercone.orthogonal_sublattice()` is a saturated sublattice of $M(\sigma)$ = `self.orthogonal_sublattice()`. This method returns the quotient lattice. The lifts of the quotient generators are $\dim(\rho) - \dim(\sigma)$ linearly independent M-lattice lattice points that, together with $M(\rho)$, generate $M(\sigma)$.

OUTPUT:

- *toric lattice quotient*.

If we call the output `Mrho`, then

- `Mrho.cover() == self.orthogonal_sublattice()`, and

- `Mrho.relations() == supercone.orthogonal_sublattice()`.

Note

- $M(\sigma)/M(\rho)$ has no torsion since the sublattice $M(\rho)$ is saturated.
- In the codimension one case, (a lift of) the generator of $M(\sigma)/M(\rho)$ is chosen to be positive on σ.

EXAMPLES:

```
sage: # needs sage.graphs
sage: rho = Cone([(1,1,1,3), (1,-1,1,3), (-1,-1,1,3), (-1,1,1,3)])
sage: rho.orthogonal_sublattice()
Sublattice <M(0, 0, 3, -1)>
sage: sigma = rho.facets()[1]
sage: sigma.orthogonal_sublattice()
Sublattice <M(0, 1, 1, 0), M(0, 0, 3, -1)>
sage: sigma.is_face_of(rho)
True
sage: Q = sigma.relative_orthogonal_quotient(rho); Q
1-d lattice, quotient
of Sublattice <M(0, 1, 1, 0), M(0, 0, 3, -1)>
by Sublattice <M(0, 0, 3, -1)>
sage: Q.gens()
(M[0, 1, 1, 0],)
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> rho = Cone([(Integer(1),Integer(1),Integer(1),Integer(3)), (Integer(1),-
↪Integer(1),Integer(1),Integer(3)), (-Integer(1),-Integer(1),Integer(1),
↪Integer(3)), (-Integer(1),Integer(1),Integer(1),Integer(3))])
>>> rho.orthogonal_sublattice()
Sublattice <M(0, 0, 3, -1)>
>>> sigma = rho.facets()[Integer(1)]
>>> sigma.orthogonal_sublattice()
Sublattice <M(0, 1, 1, 0), M(0, 0, 3, -1)>
>>> sigma.is_face_of(rho)
True
>>> Q = sigma.relative_orthogonal_quotient(rho); Q
1-d lattice, quotient
of Sublattice <M(0, 1, 1, 0), M(0, 0, 3, -1)>
by Sublattice <M(0, 0, 3, -1)>
>>> Q.gens()
(M[0, 1, 1, 0],)
```

Different codimension:

```
sage: # needs sage.graphs
sage: rho = Cone([[1,-1,1,3],[-1,-1,1,3]])
sage: sigma = rho.facets()[0]
sage: sigma.orthogonal_sublattice()
```

(continues on next page)

(continued from previous page)

```
Sublattice <M(1, 0, 2, -1), M(0, 1, 1, 0), M(0, 0, 3, -1)>
sage: rho.orthogonal_sublattice()
Sublattice <M(0, 1, 1, 0), M(0, 0, 3, -1)>
sage: sigma.relative_orthogonal_quotient(rho).gens()
(M[-1, 0, -2, 1],)
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> rho = Cone([[Integer(1),-Integer(1),Integer(1),Integer(3)],[-Integer(1),-
↪Integer(1),Integer(1),Integer(3)]])
>>> sigma = rho.facets()[Integer(0)]
>>> sigma.orthogonal_sublattice()
Sublattice <M(1, 0, 2, -1), M(0, 1, 1, 0), M(0, 0, 3, -1)>
>>> rho.orthogonal_sublattice()
Sublattice <M(0, 1, 1, 0), M(0, 0, 3, -1)>
>>> sigma.relative_orthogonal_quotient(rho).gens()
(M[-1, 0, -2, 1],)
```

Sign choice in the codimension one case:

```
sage: sigma1 = Cone([(1, 2, 3), (1, -1, 1), (-1, 1, 1), (-1, -1, 1)])  # 3d
sage: sigma2 = Cone([(1, 1, -1), (1, 2, 3), (1, -1, 1), (1, -1, -1)])  # 3d
sage: rho = sigma1.intersection(sigma2)
sage: rho.relative_orthogonal_quotient(sigma1).gens()
(M[-5, -2, 3],)
sage: rho.relative_orthogonal_quotient(sigma2).gens()
(M[5, 2, -3],)
```

```
>>> from sage.all import *
>>> sigma1 = Cone([(Integer(1), Integer(2), Integer(3)), (Integer(1), -
↪Integer(1), Integer(1)), (-Integer(1), Integer(1), Integer(1)), (-
↪Integer(1), -Integer(1), Integer(1))])  # 3d
>>> sigma2 = Cone([(Integer(1), Integer(1), -Integer(1)), (Integer(1),␣
↪Integer(2), Integer(3)), (Integer(1), -Integer(1), Integer(1)), (Integer(1),
↪ -Integer(1), -Integer(1))])  # 3d
>>> rho = sigma1.intersection(sigma2)
>>> rho.relative_orthogonal_quotient(sigma1).gens()
(M[-5, -2, 3],)
>>> rho.relative_orthogonal_quotient(sigma2).gens()
(M[5, 2, -3],)
```

relative_quotient(*subcone*)

The quotient of the spanned lattice by the lattice spanned by a subcone.

In the notation of [Ful1993], let N be the ambient lattice and N_σ the sublattice spanned by the given cone σ. If $\rho < \sigma$ is a subcone, then N_ρ = `rho.sublattice()` is a saturated sublattice of N_σ = `self.sublattice()`. This method returns the quotient lattice. The lifts of the quotient generators are $\dim(\sigma) - \dim(\rho)$ linearly independent primitive lattice points that, together with N_ρ, generate N_σ.

OUTPUT:

- *toric lattice quotient*.

Note

- The quotient N_σ/N_ρ of spanned sublattices has no torsion since the sublattice N_ρ is saturated.
- In the codimension one case, the generator of N_σ/N_ρ is chosen to be in the same direction as the image σ/N_ρ

EXAMPLES:

```
sage: sigma = Cone([(1,1,1,3),(1,-1,1,3),(-1,-1,1,3),(-1,1,1,3)])
sage: rho   = Cone([(-1, -1, 1, 3), (-1, 1, 1, 3)])
sage: sigma.sublattice()
Sublattice <N(1, 1, 1, 3), N(0, -1, 0, 0), N(-1, -1, 0, 0)>
sage: rho.sublattice()
Sublattice <N(-1, -1, 1, 3), N(0, 1, 0, 0)>
sage: sigma.relative_quotient(rho)
1-d lattice, quotient
of Sublattice <N(1, 1, 1, 3), N(0, -1, 0, 0), N(-1, -1, 0, 0)>
by Sublattice <N(1, 0, -1, -3), N(0, 1, 0, 0)>
sage: sigma.relative_quotient(rho).gens()
(N[1, 0, 0, 0],)
```

```
>>> from sage.all import *
>>> sigma = Cone([(Integer(1),Integer(1),Integer(1),Integer(3)),(Integer(1),-
↪Integer(1),Integer(1),Integer(3)),(-Integer(1),-Integer(1),Integer(1),
↪Integer(3)),(-Integer(1),Integer(1),Integer(1),Integer(3))])
>>> rho   = Cone([(-Integer(1), -Integer(1), Integer(1), Integer(3)), (-
↪Integer(1), Integer(1), Integer(1), Integer(3))])
>>> sigma.sublattice()
Sublattice <N(1, 1, 1, 3), N(0, -1, 0, 0), N(-1, -1, 0, 0)>
>>> rho.sublattice()
Sublattice <N(-1, -1, 1, 3), N(0, 1, 0, 0)>
>>> sigma.relative_quotient(rho)
1-d lattice, quotient
of Sublattice <N(1, 1, 1, 3), N(0, -1, 0, 0), N(-1, -1, 0, 0)>
by Sublattice <N(1, 0, -1, -3), N(0, 1, 0, 0)>
>>> sigma.relative_quotient(rho).gens()
(N[1, 0, 0, 0],)
```

More complicated example:

```
sage: rho = Cone([(1, 2, 3), (1, -1, 1)])
sage: sigma = Cone([(1, 2, 3), (1, -1, 1), (-1, 1, 1), (-1, -1, 1)])
sage: N_sigma = sigma.sublattice()
sage: N_sigma
Sublattice <N(1, 2, 3), N(1, -1, 1), N(-1, -1, -2)>
sage: N_rho = rho.sublattice()
sage: N_rho
Sublattice <N(1, -1, 1), N(1, 2, 3)>
sage: sigma.relative_quotient(rho).gens()
(N[-1, -1, -2],)
sage: N = rho.lattice()
```

(continues on next page)

(continued from previous page)

```
sage: N_sigma == N.span(N_rho.gens() + tuple(q.lift()
....:               for q in sigma.relative_quotient(rho).gens()))
True
```

```
>>> from sage.all import *
>>> rho = Cone([(Integer(1), Integer(2), Integer(3)), (Integer(1), -
↪Integer(1), Integer(1))])
>>> sigma = Cone([(Integer(1), Integer(2), Integer(3)), (Integer(1), -
↪Integer(1), Integer(1)), (-Integer(1), Integer(1), Integer(1)), (-
↪Integer(1), -Integer(1), Integer(1))])
>>> N_sigma = sigma.sublattice()
>>> N_sigma
Sublattice <N(1, 2, 3), N(1, -1, 1), N(-1, -1, -2)>
>>> N_rho = rho.sublattice()
>>> N_rho
Sublattice <N(1, -1, 1), N(1, 2, 3)>
>>> sigma.relative_quotient(rho).gens()
(N[-1, -1, -2],)
>>> N = rho.lattice()
>>> N_sigma == N.span(N_rho.gens() + tuple(q.lift()
...               for q in sigma.relative_quotient(rho).gens()))
True
```

Sign choice in the codimension one case:

```
sage: sigma1 = Cone([(1, 2, 3), (1, -1, 1), (-1, 1, 1), (-1, -1, 1)])  # 3d
sage: sigma2 = Cone([(1, 1, -1), (1, 2, 3), (1, -1, 1), (1, -1, -1)])  # 3d
sage: rho = sigma1.intersection(sigma2)
sage: rho.sublattice()
Sublattice <N(1, -1, 1), N(1, 2, 3)>
sage: sigma1.relative_quotient(rho)
1-d lattice, quotient
of Sublattice <N(1, 2, 3), N(1, -1, 1), N(-1, -1, -2)>
by Sublattice <N(1, 2, 3), N(0, 3, 2)>
sage: sigma1.relative_quotient(rho).gens()
(N[-1, -1, -2],)
sage: sigma2.relative_quotient(rho).gens()
(N[0, 2, 1],)
```

```
>>> from sage.all import *
>>> sigma1 = Cone([(Integer(1), Integer(2), Integer(3)), (Integer(1), -
↪Integer(1), Integer(1)), (-Integer(1), Integer(1), Integer(1)), (-
↪Integer(1), -Integer(1), Integer(1))])  # 3d
>>> sigma2 = Cone([(Integer(1), Integer(1), -Integer(1)), (Integer(1),␣
↪Integer(2), Integer(3)), (Integer(1), -Integer(1), Integer(1)), (Integer(1),
↪ -Integer(1), -Integer(1))])  # 3d
>>> rho = sigma1.intersection(sigma2)
>>> rho.sublattice()
Sublattice <N(1, -1, 1), N(1, 2, 3)>
>>> sigma1.relative_quotient(rho)
1-d lattice, quotient
of Sublattice <N(1, 2, 3), N(1, -1, 1), N(-1, -1, -2)>
```

(continues on next page)

(continued from previous page)

```
by Sublattice <N(1, 2, 3), N(0, 3, 2)>
>>> sigma1.relative_quotient(rho).gens()
(N[-1, -1, -2],)
>>> sigma2.relative_quotient(rho).gens()
(N[0, 2, 1],)
```

semigroup_generators()

Return generators for the semigroup of lattice points of `self`.

OUTPUT:

- a *PointCollection* of lattice points generating the semigroup of lattice points contained in `self`.

> **Note**
>
> No attempt is made to return a minimal set of generators, see *Hilbert_basis()* for that.

EXAMPLES:

The following command ensures that the output ordering in the examples below is independent of TOPCOM, you don't have to use it:

```
sage: PointConfiguration.set_engine('internal')
```

```
>>> from sage.all import *
>>> PointConfiguration.set_engine('internal')
```

We start with a simple case of a non-smooth 2-dimensional cone:

```
sage: Cone([(1,0), (1,2)]).semigroup_generators()
N(1, 1),
N(1, 0),
N(1, 2)
in 2-d lattice N
```

```
>>> from sage.all import *
>>> Cone([(Integer(1),Integer(0)), (Integer(1),Integer(2))]).semigroup_
→generators()
N(1, 1),
N(1, 0),
N(1, 2)
in 2-d lattice N
```

A non-simplicial cone works, too:

```
sage: cone = Cone([(3,0,-1), (1,-1,0), (0,1,0), (0,0,1)])
sage: sorted(cone.semigroup_generators())
[N(0, 0, 1), N(0, 1, 0), N(1, -1, 0), N(1, 0, 0), N(3, 0, -1)]
```

```
>>> from sage.all import *
>>> cone = Cone([(Integer(3),Integer(0),-Integer(1)), (Integer(1),-Integer(1),
→Integer(0)), (Integer(0),Integer(1),Integer(0)), (Integer(0),Integer(0),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1))])
>>> sorted(cone.semigroup_generators())
[N(0, 0, 1), N(0, 1, 0), N(1, -1, 0), N(1, 0, 0), N(3, 0, -1)]
```

GAP's toric package thinks this is challenging:

```
sage: cone = Cone([[1,2,3,4], [0,1,0,7], [3,1,0,2], [0,0,1,0]]).dual()
sage: len(cone.semigroup_generators())
2806
```

```
>>> from sage.all import *
>>> cone = Cone([[Integer(1),Integer(2),Integer(3),Integer(4)], [Integer(0),
↪Integer(1),Integer(0),Integer(7)], [Integer(3),Integer(1),Integer(0),
↪Integer(2)], [Integer(0),Integer(0),Integer(1),Integer(0)]]).dual()
>>> len(cone.semigroup_generators())
2806
```

The cone need not be strictly convex:

```
sage: halfplane = Cone([(1,0), (2,1), (-1,0)])
sage: sorted(halfplane.semigroup_generators())
[N(-1, 0), N(0, 1), N(1, 0)]
sage: line = Cone([(1,1,1), (-1,-1,-1)])
sage: sorted(line.semigroup_generators())
[N(-1, -1, -1), N(1, 1, 1)]
sage: wedge = Cone([(1,0,0), (1,2,0), (0,0,1), (0,0,-1)])
sage: sorted(wedge.semigroup_generators())
[N(0, 0, -1), N(0, 0, 1), N(1, 0, 0), N(1, 1, 0), N(1, 2, 0)]
```

```
>>> from sage.all import *
>>> halfplane = Cone([(Integer(1),Integer(0)), (Integer(2),Integer(1)), (-
↪Integer(1),Integer(0))])
>>> sorted(halfplane.semigroup_generators())
[N(-1, 0), N(0, 1), N(1, 0)]
>>> line = Cone([(Integer(1),Integer(1),Integer(1)), (-Integer(1),-Integer(1),
↪-Integer(1))])
>>> sorted(line.semigroup_generators())
[N(-1, -1, -1), N(1, 1, 1)]
>>> wedge = Cone([(Integer(1),Integer(0),Integer(0)), (Integer(1),Integer(2),
↪Integer(0)), (Integer(0),Integer(0),Integer(1)), (Integer(0),Integer(0),-
↪Integer(1))])
>>> sorted(wedge.semigroup_generators())
[N(0, 0, -1), N(0, 0, 1), N(1, 0, 0), N(1, 1, 0), N(1, 2, 0)]
```

Nor does it have to be full-dimensional (see Issue #11312):

```
sage: Cone([(1,1,0), (-1,1,0)]).semigroup_generators()
N( 0, 1, 0),
N( 1, 1, 0),
N(-1, 1, 0)
in 3-d lattice N
```

```
>>> from sage.all import *
>>> Cone([(Integer(1),Integer(1),Integer(0)), (-Integer(1),Integer(1),
↪Integer(0))]).semigroup_generators()
N( 0, 1, 0),
N( 1, 1, 0),
N(-1, 1, 0)
in 3-d lattice N
```

Neither full-dimensional nor simplicial:

```
sage: A = matrix([(1, 3, 0), (-1, 0, 1), (1, 1, -2), (15, -2, 0)])
sage: A.elementary_divisors()
[1, 1, 1, 0]
sage: cone3d = Cone([(3,0,-1), (1,-1,0), (0,1,0), (0,0,1)])
sage: rays = (A*vector(v) for v in cone3d.rays())
sage: gens = Cone(rays).semigroup_generators(); sorted(gens)
[N(-2, -1, 0, 17),
 N(0, 1, -2, 0),
 N(1, -1, 1, 15),
 N(3, -4, 5, 45),
 N(3, 0, 1, -2)]
sage: set(map(tuple,gens)) == set(tuple(A*r) for r in cone3d.semigroup_
↪generators())
True
```

```
>>> from sage.all import *
>>> A = matrix([(Integer(1), Integer(3), Integer(0)), (-Integer(1),␣
↪Integer(0), Integer(1)), (Integer(1), Integer(1), -Integer(2)),␣
↪(Integer(15), -Integer(2), Integer(0))])
>>> A.elementary_divisors()
[1, 1, 1, 0]
>>> cone3d = Cone([(Integer(3),Integer(0),-Integer(1)), (Integer(1),-
↪Integer(1),Integer(0)), (Integer(0),Integer(1),Integer(0)), (Integer(0),
↪Integer(0),Integer(1))])
>>> rays = (A*vector(v) for v in cone3d.rays())
>>> gens = Cone(rays).semigroup_generators(); sorted(gens)
[N(-2, -1, 0, 17),
 N(0, 1, -2, 0),
 N(1, -1, 1, 15),
 N(3, -4, 5, 45),
 N(3, 0, 1, -2)]
>>> set(map(tuple,gens)) == set(tuple(A*r) for r in cone3d.semigroup_
↪generators())
True
```

ALGORITHM:

If the cone is not simplicial, it is first triangulated. Each simplicial subcone has the integral points of the spaned parallelotope as generators. This is the first step of the primal Normaliz algorithm, see [Normaliz]. For each simplicial cone (of dimension d), the integral points of the open parallelotope

$$par\langle x_1,\dots,x_d\rangle = \mathbf{Z}^n \cap \{q_1 x_1 + \dots + q_d x_d : \ 0 \leq q_i < 1\}$$

are then computed [BK2001].

Finally, the union of the generators of all simplicial subcones is returned.

solid_restriction()

Return a solid representation of this cone in terms of a basis of its *sublattice()*.

We define the **solid restriction** of a cone to be a representation of that cone in a basis of its own sublattice. Since a cone's sublattice is just large enough to hold the cone (by definition), the resulting solid restriction *is_solid()*. For convenience, the solid restriction lives in a new lattice (of the appropriate dimension) and not actually in the sublattice object returned by *sublattice()*.

OUTPUT:

A solid cone in a new lattice having the same dimension as this cone's *sublattice()*.

EXAMPLES:

The nonnegative quadrant in the plane is left after we take its solid restriction in space:

```
sage: K = Cone([(1,0,0), (0,1,0)])
sage: K.solid_restriction().rays()
N(0, 1),
N(1, 0)
in 2-d lattice N
```

```
>>> from sage.all import *
>>> K = Cone([(Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0))])
>>> K.solid_restriction().rays()
N(0, 1),
N(1, 0)
in 2-d lattice N
```

The solid restriction of a single ray has the same representation regardless of the ambient space:

```
sage: K = Cone([(1,0)])
sage: K.solid_restriction().rays()
N(1)
in 1-d lattice N
sage: K = Cone([(1,1,1)])
sage: K.solid_restriction().rays()
N(1)
in 1-d lattice N
```

```
>>> from sage.all import *
>>> K = Cone([(Integer(1),Integer(0))])
>>> K.solid_restriction().rays()
N(1)
in 1-d lattice N
>>> K = Cone([(Integer(1),Integer(1),Integer(1))])
>>> K.solid_restriction().rays()
N(1)
in 1-d lattice N
```

The solid restriction of the trivial cone lives in a trivial space:

```
sage: K = cones.trivial(0)
sage: K.solid_restriction()
0-d cone in 0-d lattice N
sage: K = cones.trivial(4)
sage: K.solid_restriction()
0-d cone in 0-d lattice N
```

```
>>> from sage.all import *
>>> K = cones.trivial(Integer(0))
>>> K.solid_restriction()
0-d cone in 0-d lattice N
>>> K = cones.trivial(Integer(4))
>>> K.solid_restriction()
0-d cone in 0-d lattice N
```

The solid restriction of a solid cone is itself:

```
sage: K = Cone([(1,1),(1,2)])
sage: K.solid_restriction() is K
True
```

```
>>> from sage.all import *
>>> K = Cone([(Integer(1),Integer(1)),(Integer(1),Integer(2))])
>>> K.solid_restriction() is K
True
```

strict_quotient()

Return the quotient of `self` by the linear subspace.

We define the **strict quotient** of a cone to be the image of this cone in the quotient of the ambient space by the linear subspace of the cone, i.e. it is the "complementary part" to the linear subspace.

OUTPUT: cone

EXAMPLES:

```
sage: halfplane = Cone([(1,0), (0,1), (-1,0)])
sage: ssc = halfplane.strict_quotient()
sage: ssc
1-d cone in 1-d lattice N
sage: ssc.rays()
N(1)
in 1-d lattice N
sage: line = Cone([(1,0), (-1,0)])
sage: ssc = line.strict_quotient()
sage: ssc
0-d cone in 1-d lattice N
sage: ssc.rays()
Empty collection
in 1-d lattice N
```

```
>>> from sage.all import *
>>> halfplane = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1)), (-
```

(continues on next page)

(continued from previous page)

```
↪Integer(1),Integer(0))])
>>> ssc = halfplane.strict_quotient()
>>> ssc
1-d cone in 1-d lattice N
>>> ssc.rays()
N(1)
in 1-d lattice N
>>> line = Cone([(Integer(1),Integer(0)), (-Integer(1),Integer(0))])
>>> ssc = line.strict_quotient()
>>> ssc
0-d cone in 1-d lattice N
>>> ssc.rays()
Empty collection
in 1-d lattice N
```

The quotient of the trivial cone is trivial:

```
sage: K = cones.trivial(0)
sage: K.strict_quotient()
0-d cone in 0-d lattice N
sage: K = Cone([(0,0,0,0)])
sage: K.strict_quotient()
0-d cone in 4-d lattice N
```

```
>>> from sage.all import *
>>> K = cones.trivial(Integer(0))
>>> K.strict_quotient()
0-d cone in 0-d lattice N
>>> K = Cone([(Integer(0),Integer(0),Integer(0),Integer(0))])
>>> K.strict_quotient()
0-d cone in 4-d lattice N
```

sublattice(**args*, ***kwds*)

The sublattice spanned by the cone.

Let σ be the given cone and $N =$ `self.lattice()` the ambient lattice. Then, in the notation of [Ful1993], this method returns the sublattice

$$N_\sigma \stackrel{\text{def}}{=} span(N \cap \sigma)$$

INPUT:

- either nothing or something that can be turned into an element of this lattice.

OUTPUT:

- if no arguments were given, a *toric sublattice*, otherwise the corresponding element of it.

Note

- The sublattice spanned by the cone is the saturation of the sublattice generated by the rays of the cone.
- If you only need a **Q**-basis, you may want to try the `basis()` method on the result of `rays()`.

- The returned lattice points are usually not rays of the cone. In fact, for a non-smooth cone the rays do not generate the sublattice N_σ, but only a finite index sublattice.

EXAMPLES:

```
sage: cone = Cone([(1, 1, 1), (1, -1, 1), (-1, -1, 1), (-1, 1, 1)])
sage: cone.rays().basis()
N( 1,  1, 1),
N( 1, -1, 1),
N(-1, -1, 1)
in 3-d lattice N
sage: cone.rays().basis().matrix().det()
-4
sage: cone.sublattice()
Sublattice <N(1, 1, 1), N(0, -1, 0), N(-1, -1, 0)>
sage: matrix( cone.sublattice().gens() ).det()
-1
```

```
>>> from sage.all import *
>>> cone = Cone([(Integer(1), Integer(1), Integer(1)), (Integer(1), -
↪Integer(1), Integer(1)), (-Integer(1), -Integer(1), Integer(1)), (-
↪Integer(1), Integer(1), Integer(1))])
>>> cone.rays().basis()
N( 1,  1, 1),
N( 1, -1, 1),
N(-1, -1, 1)
in 3-d lattice N
>>> cone.rays().basis().matrix().det()
-4
>>> cone.sublattice()
Sublattice <N(1, 1, 1), N(0, -1, 0), N(-1, -1, 0)>
>>> matrix( cone.sublattice().gens() ).det()
-1
```

Another example:

```
sage: c = Cone([(1,2,3), (4,-5,1)])
sage: c
2-d cone in 3-d lattice N
sage: c.rays()
N(1,  2, 3),
N(4, -5, 1)
in 3-d lattice N
sage: c.sublattice()
Sublattice <N(4, -5, 1), N(1, 2, 3)>
sage: c.sublattice(5, -3, 4)
N(5, -3, 4)
sage: c.sublattice(1, 0, 0)
Traceback (most recent call last):
...
TypeError: element [1, 0, 0] is not in free module
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(1),Integer(2),Integer(3)), (Integer(4),-Integer(5),
↪Integer(1))])
>>> c
2-d cone in 3-d lattice N
>>> c.rays()
N(1,  2, 3),
N(4, -5, 1)
in 3-d lattice N
>>> c.sublattice()
Sublattice <N(4, -5, 1), N(1, 2, 3)>
>>> c.sublattice(Integer(5), -Integer(3), Integer(4))
N(5, -3, 4)
>>> c.sublattice(Integer(1), Integer(0), Integer(0))
Traceback (most recent call last):
...
TypeError: element [1, 0, 0] is not in free module
```

sublattice_complement(**args*, ***kwds*)

A complement of the sublattice spanned by the cone.

In other words, `sublattice()` and `sublattice_complement()` together form a **Z**-basis for the ambient `lattice()`.

In the notation of [Ful1993], let σ be the given cone and $N=$ `self.lattice()` the ambient lattice. Then this method returns

$$N(\sigma) \stackrel{\text{def}}{=} N/N_\sigma$$

lifted (non-canonically) to a sublattice of N.

INPUT:

- either nothing or something that can be turned into an element of this lattice.

OUTPUT:

- if no arguments were given, a `toric sublattice`, otherwise the corresponding element of it.

EXAMPLES:

```
sage: C2_Z2 = Cone([(1,0), (1,2)])      # C^2/Z_2
sage: c1, c2 = C2_Z2.facets()                                             #␣
↪needs sage.graphs
sage: c2.sublattice()                                                     #␣
↪needs sage.graphs
Sublattice <N(1, 2)>
sage: c2.sublattice_complement()                                          #␣
↪needs sage.graphs
Sublattice <N(0, 1)>
```

```
>>> from sage.all import *
>>> C2_Z2 = Cone([(Integer(1),Integer(0)), (Integer(1),Integer(2))])      # C^
↪2/Z_2
>>> c1, c2 = C2_Z2.facets()                                             #␣
↪needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
>>> c2.sublattice()                                                         #␣
↪needs sage.graphs
Sublattice <N(1, 2)>
>>> c2.sublattice_complement()                                              #␣
↪needs sage.graphs
Sublattice <N(0, 1)>
```

A more complicated example:

```
sage: c = Cone([(1,2,3), (4,-5,1)])
sage: c.sublattice()
Sublattice <N(4, -5, 1), N(1, 2, 3)>
sage: c.sublattice_complement()
Sublattice <N(2, -3, 0)>
sage: m = matrix( c.sublattice().gens() + c.sublattice_complement().gens() )
sage: m
[ 4 -5  1]
[ 1  2  3]
[ 2 -3  0]
sage: m.det()
-1
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(1),Integer(2),Integer(3)), (Integer(4),-Integer(5),
↪Integer(1))])
>>> c.sublattice()
Sublattice <N(4, -5, 1), N(1, 2, 3)>
>>> c.sublattice_complement()
Sublattice <N(2, -3, 0)>
>>> m = matrix( c.sublattice().gens() + c.sublattice_complement().gens() )
>>> m
[ 4 -5  1]
[ 1  2  3]
[ 2 -3  0]
>>> m.det()
-1
```

sublattice_quotient(**args*, ***kwds*)

The quotient of the ambient lattice by the sublattice spanned by the cone.

INPUT:

- either nothing or something that can be turned into an element of this lattice.

OUTPUT:

- if no arguments were given, a `quotient of a toric lattice`, otherwise the corresponding element of it.

EXAMPLES:

```
sage: # needs sage.graphs
sage: C2_Z2 = Cone([(1,0), (1,2)])      # C^2/Z_2
sage: c1, c2 = C2_Z2.facets()
```

(continues on next page)

(continued from previous page)

```
sage: c2.sublattice_quotient()
1-d lattice, quotient of 2-d lattice N by Sublattice <N(1, 2)>
sage: N = C2_Z2.lattice()
sage: n = N(1,1)
sage: n_bar = c2.sublattice_quotient(n); n_bar
N[1, 1]
sage: n_bar.lift()
N(1, 1)
sage: vector(n_bar)
(-1)
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> C2_Z2 = Cone([(Integer(1),Integer(0)), (Integer(1),Integer(2))])      # C^
↪2/Z_2
>>> c1, c2 = C2_Z2.facets()
>>> c2.sublattice_quotient()
1-d lattice, quotient of 2-d lattice N by Sublattice <N(1, 2)>
>>> N = C2_Z2.lattice()
>>> n = N(Integer(1),Integer(1))
>>> n_bar = c2.sublattice_quotient(n); n_bar
N[1, 1]
>>> n_bar.lift()
N(1, 1)
>>> vector(n_bar)
(-1)
```

class `sage.geometry.cone.`**`IntegralRayCollection`**(*rays*, *lattice*)

Bases: `SageObject`, `Hashable`, `Iterable`

Create a collection of integral rays.

> **Warning**
>
> No correctness check or normalization is performed on the input data. This class is designed for internal operations and you probably should not use it directly.

This is a base class for *`convex rational polyhedral cones`* and *`fans`*.

Ray collections are immutable, but they cache most of the returned values.

INPUT:

- `rays` – list of immutable vectors in `lattice`;
- `lattice` – *`ToricLattice`*, $\mathbf{Z}^n$, or any other object that behaves like these. If `None`, it will be determined as `parent()` of the first ray. Of course, this cannot be done if there are no rays, so in this case you must give an appropriate `lattice` directly. Note that `None` is *not* the default value - you always *must* give this argument explicitly, even if it is `None`.

OUTPUT:

- collection of given integral rays.

ambient_dim()

Return the dimension of the ambient lattice of `self`.

An alias is `ambient_dim()`.

OUTPUT: integer

EXAMPLES:

```
sage: c = Cone([(1,0)])
sage: c.lattice_dim()
2
sage: c.dim()
1
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(1),Integer(0))])
>>> c.lattice_dim()
2
>>> c.dim()
1
```

ambient_vector_space(*base_field=None*)

Return the ambient vector space.

It is the ambient lattice (`lattice()`) tensored with a field.

INPUT:

- `base_field` – (default: the rationals) a field

EXAMPLES:

```
sage: c = Cone([(1,0)])
sage: c.ambient_vector_space()
Vector space of dimension 2 over Rational Field
sage: c.ambient_vector_space(AA)                                          #␣
↪needs sage.rings.number_field
Vector space of dimension 2 over Algebraic Real Field
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(1),Integer(0))])
>>> c.ambient_vector_space()
Vector space of dimension 2 over Rational Field
>>> c.ambient_vector_space(AA)                                          #␣
↪needs sage.rings.number_field
Vector space of dimension 2 over Algebraic Real Field
```

cartesian_product(*other*, *lattice=None*)

Return the Cartesian product of `self` with `other`.

INPUT:

- `other` – an `IntegralRayCollection`;
- `lattice` – (optional) the ambient lattice for the result. By default, the direct sum of the ambient lattices of `self` and `other` is constructed.

OUTPUT: an *IntegralRayCollection*

By the Cartesian product of ray collections $(r_0, \ldots, r_{n-1})$ and $(s_0, \ldots, s_{m-1})$ we understand the ray collection of the form $((r_0, 0), \ldots, (r_{n-1}, 0), (0, s_0), \ldots, (0, s_{m-1}))$, which is suitable for Cartesian products of cones and fans. The ray order is guaranteed to be as described.

EXAMPLES:

```
sage: c = Cone([(1,)])
sage: c.cartesian_product(c)    # indirect doctest
2-d cone in 2-d lattice N+N
sage: _.rays()
N+N(1, 0),
N+N(0, 1)
in 2-d lattice N+N
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(1),)])
>>> c.cartesian_product(c)    # indirect doctest
2-d cone in 2-d lattice N+N
>>> _.rays()
N+N(1, 0),
N+N(0, 1)
in 2-d lattice N+N
```

codim()

Return the codimension of `self`.

The codimension of a collection of rays (of a cone/fan) is the difference between the dimension of the ambient space and the dimension of the subspace spanned by those rays (of the cone/fan).

OUTPUT: nonnegative integer representing the codimension of `self`

See also

dim(), *lattice_dim()*

EXAMPLES:

The codimension of the nonnegative orthant is zero, since the span of its generators equals the entire ambient space:

```
sage: K = cones.nonnegative_orthant(3)
sage: K.codim()
0
```

```
>>> from sage.all import *
>>> K = cones.nonnegative_orthant(Integer(3))
>>> K.codim()
0
```

However, if we remove a ray so that the entire cone is contained within the x-y plane, then the resulting cone will have codimension one, because the z-axis is perpendicular to every element of the cone:

```
sage: K = Cone([(1,0,0), (0,1,0)])
sage: K.codim()
1
```

```
>>> from sage.all import *
>>> K = Cone([(Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0))])
>>> K.codim()
1
```

If our cone is all of $\mathbb{R}^2$, then its codimension is zero:

```
sage: K = Cone([(1,0), (-1,0), (0,1), (0,-1)])
sage: K.is_full_space()
True
sage: K.codim()
0
```

```
>>> from sage.all import *
>>> K = Cone([(Integer(1),Integer(0)), (-Integer(1),Integer(0)), (Integer(0),
↪Integer(1)), (Integer(0),-Integer(1))])
>>> K.is_full_space()
True
>>> K.codim()
0
```

And if the cone is trivial in any space, then its codimension is equal to the dimension of the ambient space:

```
sage: K = cones.trivial(0)
sage: K.lattice_dim()
0
sage: K.codim()
0

sage: K = cones.trivial(1)
sage: K.lattice_dim()
1
sage: K.codim()
1

sage: K = cones.trivial(2)
sage: K.lattice_dim()
2
sage: K.codim()
2
```

```
>>> from sage.all import *
>>> K = cones.trivial(Integer(0))
>>> K.lattice_dim()
0
>>> K.codim()
0
```

(continues on next page)

(continued from previous page)

```
>>> K = cones.trivial(Integer(1))
>>> K.lattice_dim()
1
>>> K.codim()
1

>>> K = cones.trivial(Integer(2))
>>> K.lattice_dim()
2
>>> K.codim()
2
```

codimension()

Return the codimension of `self`.

The codimension of a collection of rays (of a cone/fan) is the difference between the dimension of the ambient space and the dimension of the subspace spanned by those rays (of the cone/fan).

OUTPUT: nonnegative integer representing the codimension of `self`

See also

dim(), *lattice_dim()*

EXAMPLES:

The codimension of the nonnegative orthant is zero, since the span of its generators equals the entire ambient space:

```
sage: K = cones.nonnegative_orthant(3)
sage: K.codim()
0
```

```
>>> from sage.all import *
>>> K = cones.nonnegative_orthant(Integer(3))
>>> K.codim()
0
```

However, if we remove a ray so that the entire cone is contained within the x-y plane, then the resulting cone will have codimension one, because the z-axis is perpendicular to every element of the cone:

```
sage: K = Cone([(1,0,0), (0,1,0)])
sage: K.codim()
1
```

```
>>> from sage.all import *
>>> K = Cone([(Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0))])
>>> K.codim()
1
```

If our cone is all of $\mathbb{R}^2$, then its codimension is zero:

```
sage: K = Cone([(1,0), (-1,0), (0,1), (0,-1)])
sage: K.is_full_space()
True
sage: K.codim()
0
```

```
>>> from sage.all import *
>>> K = Cone([(Integer(1),Integer(0)), (-Integer(1),Integer(0)), (Integer(0),
↪Integer(1)), (Integer(0),-Integer(1))])
>>> K.is_full_space()
True
>>> K.codim()
0
```

And if the cone is trivial in any space, then its codimension is equal to the dimension of the ambient space:

```
sage: K = cones.trivial(0)
sage: K.lattice_dim()
0
sage: K.codim()
0

sage: K = cones.trivial(1)
sage: K.lattice_dim()
1
sage: K.codim()
1

sage: K = cones.trivial(2)
sage: K.lattice_dim()
2
sage: K.codim()
2
```

```
>>> from sage.all import *
>>> K = cones.trivial(Integer(0))
>>> K.lattice_dim()
0
>>> K.codim()
0

>>> K = cones.trivial(Integer(1))
>>> K.lattice_dim()
1
>>> K.codim()
1

>>> K = cones.trivial(Integer(2))
>>> K.lattice_dim()
2
>>> K.codim()
2
```

dim()

Return the dimension of the subspace spanned by rays of `self`.

OUTPUT: integer

EXAMPLES:

```
sage: c = Cone([(1,0)])
sage: c.lattice_dim()
2
sage: c.dim()
1
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(1),Integer(0))])
>>> c.lattice_dim()
2
>>> c.dim()
1
```

dual_lattice()

Return the dual of the ambient lattice of `self`.

OUTPUT:

- lattice. If possible (that is, if `lattice()` has a `dual()` method), the dual lattice is returned. Otherwise, $\mathbf{Z}^n$ is returned, where n is the dimension of `lattice()`.

EXAMPLES:

```
sage: c = Cone([(1,0)])
sage: c.dual_lattice()
2-d lattice M
sage: Cone([], ZZ^3).dual_lattice()
Ambient free module of rank 3
over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(1),Integer(0))])
>>> c.dual_lattice()
2-d lattice M
>>> Cone([], ZZ**Integer(3)).dual_lattice()
Ambient free module of rank 3
over the principal ideal domain Integer Ring
```

lattice()

Return the ambient lattice of `self`.

OUTPUT: lattice

EXAMPLES:

```
sage: c = Cone([(1,0)])
sage: c.lattice()
2-d lattice N
sage: Cone([], ZZ^3).lattice()
```

(continues on next page)

(continued from previous page)

```
Ambient free module of rank 3
over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(1),Integer(0))])
>>> c.lattice()
2-d lattice N
>>> Cone([], ZZ**Integer(3)).lattice()
Ambient free module of rank 3
over the principal ideal domain Integer Ring
```

lattice_dim()

Return the dimension of the ambient lattice of `self`.

An alias is `ambient_dim()`.

OUTPUT: integer

EXAMPLES:

```
sage: c = Cone([(1,0)])
sage: c.lattice_dim()
2
sage: c.dim()
1
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(1),Integer(0))])
>>> c.lattice_dim()
2
>>> c.dim()
1
```

nrays()

Return the number of rays of `self`.

OUTPUT: integer

EXAMPLES:

```
sage: c = Cone([(1,0), (0,1)])
sage: c.nrays()
2
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> c.nrays()
2
```

plot(***options*)

Plot `self`.

INPUT:

- any options for toric plots (see `toric_plotter.options`), none are mandatory.

OUTPUT: a plot

EXAMPLES:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: quadrant.plot()                                                    #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 9 graphics primitives
```

```
>>> from sage.all import *
>>> quadrant = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> quadrant.plot()                                                      #␣
↪needs sage.plot sage.symbolic
Graphics object consisting of 9 graphics primitives
```

ray(*n*)

Return the `n`-th ray of `self`.

INPUT:

- `n` – integer; an index of a ray of `self`. Enumeration of rays starts with zero

OUTPUT: ray; an element of the lattice of `self`

EXAMPLES:

```
sage: c = Cone([(1,0), (0,1)])
sage: c.ray(0)
N(1, 0)
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> c.ray(Integer(0))
N(1, 0)
```

rays(**args*)

Return (some of the) rays of `self`.

INPUT:

- `ray_list` – list of integers, the indices of the requested rays. If not specified, all rays of `self` will be returned

OUTPUT: a *PointCollection* of primitive integral ray generators

EXAMPLES:

```
sage: c = Cone([(1,0), (0,1), (-1, 0)])
sage: c.rays()
N( 0, 1),
N( 1, 0),
N(-1, 0)
in 2-d lattice N
sage: c.rays([0, 2])
N( 0, 1),
N(-1, 0)
in 2-d lattice N
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1)), (-Integer(1),
↪Integer(0))])
>>> c.rays()
N( 0, 1),
N( 1, 0),
N(-1, 0)
in 2-d lattice N
>>> c.rays([Integer(0), Integer(2)])
N( 0, 1),
N(-1, 0)
in 2-d lattice N
```

You can also give ray indices directly, without packing them into a list:

```
sage: c.rays(0, 2)
N( 0, 1),
N(-1, 0)
in 2-d lattice N
```

```
>>> from sage.all import *
>>> c.rays(Integer(0), Integer(2))
N( 0, 1),
N(-1, 0)
in 2-d lattice N
```

span (*base_ring=None*)

Return the span of `self`.

INPUT:

- `base_ring` – (default: from lattice) the base ring to use for the generated module

OUTPUT: a module spanned by the generators of `self`

EXAMPLES:

The span of a single ray is a one-dimensional sublattice:

```
sage: K1 = Cone([(1,)])
sage: K1.span()
Sublattice <N(1)>
sage: K2 = Cone([(1,0)])
sage: K2.span()
Sublattice <N(1, 0)>
```

```
>>> from sage.all import *
>>> K1 = Cone([(Integer(1),)])
>>> K1.span()
Sublattice <N(1)>
>>> K2 = Cone([(Integer(1),Integer(0))])
>>> K2.span()
Sublattice <N(1, 0)>
```

The span of the nonnegative orthant is the entire ambient lattice:

```
sage: K = cones.nonnegative_orthant(3)
sage: K.span() == K.lattice()
True
```

```
>>> from sage.all import *
>>> K = cones.nonnegative_orthant(Integer(3))
>>> K.span() == K.lattice()
True
```

By specifying a `base_ring`, we can obtain a vector space:

```
sage: K = Cone([(1,0,0),(0,1,0),(0,0,1)])
sage: K.span(base_ring=QQ)
Vector space of degree 3 and dimension 3 over Rational Field
Basis matrix:
[1 0 0]
[0 1 0]
[0 0 1]
```

```
>>> from sage.all import *
>>> K = Cone([(Integer(1),Integer(0),Integer(0)),(Integer(0),Integer(1),
↪Integer(0)),(Integer(0),Integer(0),Integer(1))])
>>> K.span(base_ring=QQ)
Vector space of degree 3 and dimension 3 over Rational Field
Basis matrix:
[1 0 0]
[0 1 0]
[0 0 1]
```

`sage.geometry.cone.`**`classify_cone_2d`**(*ray0*, *ray1*, *check=True*)

Return (d, k) classifying the lattice cone spanned by the two rays.

INPUT:

- `ray0`, `ray1` – two primitive integer vectors; the generators of the two rays generating the two-dimensional cone
- `check` – boolean (default: `True`); whether to check the input rays for consistency

OUTPUT:

A pair (d, k) of integers classifying the cone up to $GL(2, \mathbf{Z})$ equivalence. See Proposition 10.1.1 of [CLS2011] for the definition. We return the unique (d, k) with minimal k, see Proposition 10.1.3 of [CLS2011].

EXAMPLES:

```
sage: ray0 = vector([1,0])
sage: ray1 = vector([2,3])
sage: from sage.geometry.cone import classify_cone_2d
sage: classify_cone_2d(ray0, ray1)
(3, 2)

sage: ray0 = vector([2,4,5])
sage: ray1 = vector([5,19,11])
sage: classify_cone_2d(ray0, ray1)
```

(continues on next page)

(continued from previous page)

```
(3, 1)

sage: m = matrix(ZZ, [(19, -14, -115), (-2, 5, 25), (43, -42, -298)])
sage: m.det()    # check that it is in GL(3,ZZ)
-1
sage: classify_cone_2d(m*ray0, m*ray1)
(3, 1)
```

```
>>> from sage.all import *
>>> ray0 = vector([Integer(1),Integer(0)])
>>> ray1 = vector([Integer(2),Integer(3)])
>>> from sage.geometry.cone import classify_cone_2d
>>> classify_cone_2d(ray0, ray1)
(3, 2)

>>> ray0 = vector([Integer(2),Integer(4),Integer(5)])
>>> ray1 = vector([Integer(5),Integer(19),Integer(11)])
>>> classify_cone_2d(ray0, ray1)
(3, 1)

>>> m = matrix(ZZ, [(Integer(19), -Integer(14), -Integer(115)), (-Integer(2),␣
↪Integer(5), Integer(25)), (Integer(43), -Integer(42), -Integer(298))])
>>> m.det()    # check that it is in GL(3,ZZ)
-1
>>> classify_cone_2d(m*ray0, m*ray1)
(3, 1)
```

sage.geometry.cone.**integral_length**(*v*)

Compute the integral length of a given rational vector.

INPUT:

- `v` – any object which can be converted to a list of rationals

OUTPUT:

Rational number `r` such that `v = r * u`, where `u` is the primitive integral vector in the direction of `v`.

EXAMPLES:

```
sage: from sage.geometry.cone import integral_length
sage: integral_length([1, 2, 4])
1
sage: integral_length([2, 2, 4])
2
sage: integral_length([2/3, 2, 4])
2/3
```

```
>>> from sage.all import *
>>> from sage.geometry.cone import integral_length
>>> integral_length([Integer(1), Integer(2), Integer(4)])
1
>>> integral_length([Integer(2), Integer(2), Integer(4)])
2
```

(continues on next page)

(continued from previous page)

```
>>> integral_length([Integer(2)/Integer(3), Integer(2), Integer(4)])
2/3
```

sage.geometry.cone.**is_Cone**(*x*)

Check if `x` is a cone.

INPUT:

- `x` – anything

OUTPUT: `True` if `x` is a cone and `False` otherwise

EXAMPLES:

```
sage: from sage.geometry.cone import is_Cone
sage: is_Cone(1)
doctest:warning...
DeprecationWarning: is_Cone is deprecated, use isinstance instead
See https://github.com/sagemath/sage/issues/34307 for details.
False
sage: quadrant = Cone([(1,0), (0,1)])
sage: quadrant
2-d cone in 2-d lattice N
sage: is_Cone(quadrant)
True
```

```
>>> from sage.all import *
>>> from sage.geometry.cone import is_Cone
>>> is_Cone(Integer(1))
doctest:warning...
DeprecationWarning: is_Cone is deprecated, use isinstance instead
See https://github.com/sagemath/sage/issues/34307 for details.
False
>>> quadrant = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> quadrant
2-d cone in 2-d lattice N
>>> is_Cone(quadrant)
True
```

sage.geometry.cone.**normalize_rays**(*rays*, *lattice*)

Normalize a list of rational rays: make them primitive and immutable.

INPUT:

- `rays` – list of rays which can be converted to the rational extension of `lattice`;
- `lattice` – *ToricLattice*, $\mathbf{Z}^n$, or any other object that behaves like these. If `None`, an attempt will be made to determine an appropriate toric lattice automatically.

OUTPUT:

- list of immutable primitive vectors of the `lattice` in the same directions as original `rays`.

EXAMPLES:

```
sage: from sage.geometry.cone import normalize_rays
sage: normalize_rays([(0, 1), (0, 2), (3, 2), (5/7, 10/3)], None)
```

(continues on next page)

(continued from previous page)

```
[N(0, 1), N(0, 1), N(3, 2), N(3, 14)]
sage: L = ToricLattice(2, "L")
sage: normalize_rays([(0, 1), (0, 2), (3, 2), (5/7, 10/3)], L.dual())
[L*(0, 1), L*(0, 1), L*(3, 2), L*(3, 14)]
sage: ray_in_L = L(0,1)
sage: normalize_rays([ray_in_L, (0, 2), (3, 2), (5/7, 10/3)], None)
[L(0, 1), L(0, 1), L(3, 2), L(3, 14)]
sage: normalize_rays([(0, 1), (0, 2), (3, 2), (5/7, 10/3)], ZZ^2)
[(0, 1), (0, 1), (3, 2), (3, 14)]
sage: normalize_rays([(0, 1), (0, 2), (3, 2), (5/7, 10/3)], ZZ^3)
Traceback (most recent call last):
...
TypeError: cannot convert (0, 1) to
Vector space of dimension 3 over Rational Field!
sage: normalize_rays([], ZZ^3)
[]
```

```
>>> from sage.all import *
>>> from sage.geometry.cone import normalize_rays
>>> normalize_rays([(Integer(0), Integer(1)), (Integer(0), Integer(2)),
↪(Integer(3), Integer(2)), (Integer(5)/Integer(7), Integer(10)/Integer(3))],
↪None)
[N(0, 1), N(0, 1), N(3, 2), N(3, 14)]
>>> L = ToricLattice(Integer(2), "L")
>>> normalize_rays([(Integer(0), Integer(1)), (Integer(0), Integer(2)),
↪(Integer(3), Integer(2)), (Integer(5)/Integer(7), Integer(10)/Integer(3))], L.
↪dual())
[L*(0, 1), L*(0, 1), L*(3, 2), L*(3, 14)]
>>> ray_in_L = L(Integer(0),Integer(1))
>>> normalize_rays([ray_in_L, (Integer(0), Integer(2)), (Integer(3), Integer(2)),
↪(Integer(5)/Integer(7), Integer(10)/Integer(3))], None)
[L(0, 1), L(0, 1), L(3, 2), L(3, 14)]
>>> normalize_rays([(Integer(0), Integer(1)), (Integer(0), Integer(2)),
↪(Integer(3), Integer(2)), (Integer(5)/Integer(7), Integer(10)/Integer(3))],
↪ZZ**Integer(2))
[(0, 1), (0, 1), (3, 2), (3, 14)]
>>> normalize_rays([(Integer(0), Integer(1)), (Integer(0), Integer(2)),
↪(Integer(3), Integer(2)), (Integer(5)/Integer(7), Integer(10)/Integer(3))],
↪ZZ**Integer(3))
Traceback (most recent call last):
...
TypeError: cannot convert (0, 1) to
Vector space of dimension 3 over Rational Field!
>>> normalize_rays([], ZZ**Integer(3))
[]
```

`sage.geometry.cone.`**`random_cone`**(*lattice=None*, *min_ambient_dim=0*, *max_ambient_dim=None*, *min_rays=0*, *max_rays=None*, *strictly_convex=None*, *solid=None*)

Generate a random convex rational polyhedral cone.

Lower and upper bounds may be provided for both the dimension of the ambient space and the number of generating rays of the cone. If a lower bound is left unspecified, it defaults to zero. Unspecified upper bounds will be chosen randomly, unless you set `solid`, in which case they are chosen a little more wisely.

You may specify the ambient `lattice` for the returned cone. In that case, the `min_ambient_dim` and `max_ambient_dim` parameters are ignored.

You may also request that the returned cone be strictly convex (or not). Likewise you may request that it be (non-)solid.

> **Warning**
>
> If you request a large number of rays in a low-dimensional space, you might be waiting for a while. For example, in three dimensions, it is possible to obtain an octagon raised up to height one (all z-coordinates equal to one). But in practice, we usually generate the entire three-dimensional space with six rays before we get to the eight rays needed for an octagon. We therefore have to throw the cone out and start over from scratch. This process repeats until we get lucky.
>
> We also refrain from "adjusting" the min/max parameters given to us when a (non-)strictly convex or (non-)solid cone is requested. This means that it may take a long time to generate such a cone if the parameters are chosen unwisely.
>
> For example, you may want to set `min_rays` close to `min_ambient_dim` if you desire a solid cone. Or, if you desire a non-strictly-convex cone, then they all contain at least two generating rays. So that might be a good candidate for `min_rays`.

INPUT:

- `lattice` – (default: random) a `ToricLattice` object in which the returned cone will live. By default a new lattice will be constructed with a randomly-chosen rank (subject to `min_ambient_dim` and `max_ambient_dim`).
- `min_ambient_dim` – (default: zero) a nonnegative integer representing the minimum dimension of the ambient lattice
- `max_ambient_dim` – (default: random) a nonnegative integer representing the maximum dimension of the ambient lattice
- `min_rays` – (default: zero) a nonnegative integer representing the minimum number of generating rays of the cone
- `max_rays` – (default: random) a nonnegative integer representing the maximum number of generating rays of the cone
- `strictly_convex` – (default: random) whether or not to make the returned cone strictly convex. Specify `True` for a strictly convex cone, `False` for a non-strictly-convex cone, or `None` if you don't care.
- `solid` – (default: random) whether or not to make the returned cone solid. Specify `True` for a solid cone, `False` for a non-solid cone, or `None` if you don't care.

OUTPUT: a new, randomly generated cone

A `ValueError` will be thrown under the following conditions:

- Any of `min_ambient_dim`, `max_ambient_dim`, `min_rays`, or `max_rays` are negative.
- `max_ambient_dim` is less than `min_ambient_dim`.
- `max_rays` is less than `min_rays`.
- Both `max_ambient_dim` and `lattice` are specified.
- `min_rays` is greater than four but `max_ambient_dim` is less than three.
- `min_rays` is greater than four but `lattice` has dimension less than three.

- `min_rays` is greater than two but `max_ambient_dim` is less than two.
- `min_rays` is greater than two but `lattice` has dimension less than two.
- `min_rays` is positive but `max_ambient_dim` is zero.
- `min_rays` is positive but `lattice` has dimension zero.
- A trivial lattice is supplied and a non-strictly-convex cone is requested.
- A non-strictly-convex cone is requested but `max_rays` is less than two.
- A solid cone is requested but `max_rays` is less than `min_ambient_dim`.
- A solid cone is requested but `max_rays` is less than the dimension of `lattice`.
- A non-solid cone is requested but `max_ambient_dim` is zero.
- A non-solid cone is requested but `lattice` has dimension zero.
- A non-solid cone is requested but `min_rays` is so large that it guarantees a solid cone.

ALGORITHM:

First, a lattice is determined from `min_ambient_dim` and `max_ambient_dim` (or from the supplied `lattice`).

Then, lattice elements are generated one at a time and added to a cone. This continues until either the cone meets the user's requirements, or the cone is equal to the entire space (at which point it is futile to generate more).

We check whether or not the resulting cone meets the user's requirements; if it does, it is returned. If not, we throw it away and start over. This process repeats indefinitely until an appropriate cone is generated.

EXAMPLES:

Generate a trivial cone in a trivial space:

```
sage: random_cone(max_ambient_dim=0, max_rays=0)
0-d cone in 0-d lattice N
```

```
>>> from sage.all import *
>>> random_cone(max_ambient_dim=Integer(0), max_rays=Integer(0))
0-d cone in 0-d lattice N
```

We can predict the ambient dimension when `min_ambient_dim == max_ambient_dim`:

```
sage: K = random_cone(min_ambient_dim=4, max_ambient_dim=4)
sage: K.lattice_dim()
4
```

```
>>> from sage.all import *
>>> K = random_cone(min_ambient_dim=Integer(4), max_ambient_dim=Integer(4))
>>> K.lattice_dim()
4
```

Likewise for the number of rays when `min_rays == max_rays`:

```
sage: K = random_cone(min_rays=3, max_rays=3)
sage: K.nrays()
3
```

```
>>> from sage.all import *
>>> K = random_cone(min_rays=Integer(3), max_rays=Integer(3))
>>> K.nrays()
3
```

If we specify a lattice, then the returned cone will live in it:

```
sage: L = ToricLattice(5, "L")
sage: K = random_cone(lattice=L)
sage: K.lattice() is L
True
```

```
>>> from sage.all import *
>>> L = ToricLattice(Integer(5), "L")
>>> K = random_cone(lattice=L)
>>> K.lattice() is L
True
```

We can also request a strictly convex cone:

```
sage: K = random_cone(max_ambient_dim=8, max_rays=10,
....:                 strictly_convex=True)
sage: K.is_strictly_convex()
True
```

```
>>> from sage.all import *
>>> K = random_cone(max_ambient_dim=Integer(8), max_rays=Integer(10),
...                 strictly_convex=True)
>>> K.is_strictly_convex()
True
```

Or one that isn't strictly convex:

```
sage: K = random_cone(min_ambient_dim=5, min_rays=2,
....:                 strictly_convex=False)
sage: K.is_strictly_convex()
False
```

```
>>> from sage.all import *
>>> K = random_cone(min_ambient_dim=Integer(5), min_rays=Integer(2),
...                 strictly_convex=False)
>>> K.is_strictly_convex()
False
```

An example with all parameters set:

```
sage: K = random_cone(min_ambient_dim=4, max_ambient_dim=7,
....:                 min_rays=2, max_rays=10,
....:                 strictly_convex=False, solid=True)
sage: 4 <= K.lattice_dim() and K.lattice_dim() <= 7
True
sage: 2 <= K.nrays() and K.nrays() <= 10
True
```

(continues on next page)

(continued from previous page)

```
sage: K.is_strictly_convex()
False
sage: K.is_solid()
True
```

```
>>> from sage.all import *
>>> K = random_cone(min_ambient_dim=Integer(4), max_ambient_dim=Integer(7),
...                 min_rays=Integer(2), max_rays=Integer(10),
...                 strictly_convex=False, solid=True)
>>> Integer(4) <= K.lattice_dim() and K.lattice_dim() <= Integer(7)
True
>>> Integer(2) <= K.nrays() and K.nrays() <= Integer(10)
True
>>> K.is_strictly_convex()
False
>>> K.is_solid()
True
```

2.5.3 Catalog of common polyhedral convex cones

This module provides shortcut functions, grouped under the globally-available `cones` prefix, to create some common cones:

- The downward-monotone cone,
- The nonnegative orthant,
- The rearrangement cone of order `p`,
- The Schur cone,
- The trivial cone.

At the moment, only convex rational polyhedral cones are supported—specifically, those cones that can be built using the *Cone()* constructor. As a result, each shortcut method can be passed either an ambient dimension `ambient_dim`, or a toric `lattice` (from which the dimension can be inferred) to determine the ambient space.

Here are some typical usage examples:

```
sage: cones.downward_monotone(3).rays()
N( 1,  0,  0),
N( 1,  1,  0),
N( 1,  1,  1),
N(-1, -1, -1)
in 3-d lattice N
```

```
>>> from sage.all import *
>>> cones.downward_monotone(Integer(3)).rays()
N( 1,  0,  0),
N( 1,  1,  0),
N( 1,  1,  1),
N(-1, -1, -1)
in 3-d lattice N
```

```
sage: cones.nonnegative_orthant(2).rays()
N(1, 0),
N(0, 1)
in 2-d lattice N
```

```
>>> from sage.all import *
>>> cones.nonnegative_orthant(Integer(2)).rays()
N(1, 0),
N(0, 1)
in 2-d lattice N
```

```
sage: cones.rearrangement(2,2).rays()
N( 1,  0),
N( 1, -1),
N(-1,  1)
in 2-d lattice N
```

```
>>> from sage.all import *
>>> cones.rearrangement(Integer(2),Integer(2)).rays()
N( 1,  0),
N( 1, -1),
N(-1,  1)
in 2-d lattice N
```

```
sage: cones.schur(3).rays()
N(1, -1,  0),
N(0,  1, -1)
in 3-d lattice N
```

```
>>> from sage.all import *
>>> cones.schur(Integer(3)).rays()
N(1, -1,  0),
N(0,  1, -1)
in 3-d lattice N
```

```
sage: cones.trivial(3).rays()
Empty collection
in 3-d lattice N
```

```
>>> from sage.all import *
>>> cones.trivial(Integer(3)).rays()
Empty collection
in 3-d lattice N
```

To specify some other lattice, pass it as an argument to the function:

```
sage: K = cones.nonnegative_orthant(3)
sage: cones.schur(lattice=K.dual().lattice())
2-d cone in 3-d lattice M
```

```
>>> from sage.all import *
>>> K = cones.nonnegative_orthant(Integer(3))
>>> cones.schur(lattice=K.dual().lattice())
2-d cone in 3-d lattice M
```

For more information about these cones, see the documentation for the individual functions and the references therein.

sage.geometry.cone_catalog.**downward_monotone**(*ambient_dim=None*, *lattice=None*)

The downward-monotone cone in `ambient_dim` dimensions, or living in `lattice`.

The elements of the downward-monotone cone are vectors whose components are arranged in non-increasing order. Vectors whose entries are arranged in the reverse (non-decreasing) order are sometimes called isotone vectors, and are used in statistics for isotonic regression.

The downward-monotone cone is the dual of the Schur cone. It is also often referred to as the downward-monotone cone.

INPUT:

- `ambient_dim` – nonnegative integer (default: `None`); the dimension of the ambient space
- `lattice` – a toric lattice (default: `None`); the lattice in which the cone will live

If `ambient_dim` is omitted, then it will be inferred from the rank of `lattice`. If the `lattice` is omitted, then the default lattice of rank `ambient_dim` will be used.

A `ValueError` is raised if neither `ambient_dim` nor `lattice` are specified. It is also a `ValueError` to specify both `ambient_dim` and `lattice` unless the rank of `lattice` is equal to `ambient_dim`.

OUTPUT:

A *ConvexRationalPolyhedralCone* living in `lattice` whose elements' entries are arranged in nonincreasing order. Each generating ray has the integer ring as its base ring.

A `ValueError` can be raised if the inputs are incompatible or insufficient. See the INPUT documentation for details.

See also

schur()

REFERENCES:

- [GS2010], Section 3.1
- [Niez1998], Example 2.2

EXAMPLES:

The entries of the elements of the downward-monotone cone are in non-increasing order:

```
sage: ambient_dim = ZZ.random_element(10)
sage: K = cones.downward_monotone(ambient_dim)
sage: all( x[i] >= x[i + 1]
....:      for i in range(ambient_dim - 1)
....:      for x in K.rays() )
True
sage: x = K.random_element()
sage: all( x[i] >= x[i + 1] for i in range(ambient_dim - 1) )
True
```

```
>>> from sage.all import *
>>> ambient_dim = ZZ.random_element(Integer(10))
>>> K = cones.downward_monotone(ambient_dim)
>>> all( x[i] >= x[i + Integer(1)]
...      for i in range(ambient_dim - Integer(1))
...      for x in K.rays() )
True
>>> x = K.random_element()
>>> all( x[i] >= x[i + Integer(1)] for i in range(ambient_dim - Integer(1)) )
True
```

A nontrivial downward-monotone cone is solid but not proper, since it contains both the vector of all ones and its negation; that, however, is the only subspace it contains:

```
sage: ambient_dim = ZZ.random_element(1,10)
sage: K = cones.downward_monotone(ambient_dim)
sage: K.is_solid()
True
sage: K.is_proper()
False
sage: K.lineality()
1
```

```
>>> from sage.all import *
>>> ambient_dim = ZZ.random_element(Integer(1),Integer(10))
>>> K = cones.downward_monotone(ambient_dim)
>>> K.is_solid()
True
>>> K.is_proper()
False
>>> K.lineality()
1
```

The dual of the downward-monotone cone is the Schur cone [GS2010] that induces the majorization preordering:

```
sage: ambient_dim = ZZ.random_element(10)
sage: K = cones.downward_monotone(ambient_dim).dual()
sage: J = cones.schur(ambient_dim, K.lattice())
sage: K.is_equivalent(J)
True
```

```
>>> from sage.all import *
>>> ambient_dim = ZZ.random_element(Integer(10))
>>> K = cones.downward_monotone(ambient_dim).dual()
>>> J = cones.schur(ambient_dim, K.lattice())
>>> K.is_equivalent(J)
True
```

`sage.geometry.cone_catalog.`**`nonnegative_orthant`**(*ambient_dim=None*, *lattice=None*)

The nonnegative orthant in `ambient_dim` dimensions, or living in `lattice`.

The nonnegative orthant consists of all componentwise-nonnegative vectors. It is the convex-conic hull of the standard basis.

INPUT:

- `ambient_dim` – nonnegative integer (default: `None`); the dimension of the ambient space
- `lattice` – a toric lattice (default: `None`); the lattice in which the cone will live

If `ambient_dim` is omitted, then it will be inferred from the rank of `lattice`. If the `lattice` is omitted, then the default lattice of rank `ambient_dim` will be used.

A `ValueError` is raised if neither `ambient_dim` nor `lattice` are specified. It is also a `ValueError` to specify both `ambient_dim` and `lattice` unless the rank of `lattice` is equal to `ambient_dim`.

OUTPUT:

A *`ConvexRationalPolyhedralCone`* living in `lattice` and having `ambient_dim` standard basis vectors as its generators. Each generating ray has the integer ring as its base ring.

A `ValueError` can be raised if the inputs are incompatible or insufficient. See the INPUT documentation for details.

REFERENCES:

- Chapter 2 in [BV2009] (Examples 2.4, 2.14, and 2.23 in particular)

EXAMPLES:

```
sage: cones.nonnegative_orthant(3).rays()
N(1, 0, 0),
N(0, 1, 0),
N(0, 0, 1)
in 3-d lattice N
```

```
>>> from sage.all import *
>>> cones.nonnegative_orthant(Integer(3)).rays()
N(1, 0, 0),
N(0, 1, 0),
N(0, 0, 1)
in 3-d lattice N
```

`sage.geometry.cone_catalog.`**`rearrangement`**`(`*p*, *ambient_dim=None*, *lattice=None*`)`

The rearrangement cone of order `p` in `ambient_dim` dimensions, or living in `lattice`.

The rearrangement cone of order `p` in `ambient_dim` dimensions consists of all vectors of length `ambient_dim` whose smallest `p` components sum to a nonnegative number.

For example, the rearrangement cone of order one has its single smallest component nonnegative. This implies that all components are nonnegative, and that therefore the rearrangement cone of order one is the nonnegative orthant in its ambient space.

When `p` and `ambient_dim` are equal, all components of the cone's elements must sum to a nonnegative number. In other words, the rearrangement cone of order `ambient_dim` is a half-space.

INPUT:

- `p` – nonnegative integer; the number of components to "rearrange", between `1` and `ambient_dim` inclusive
- `ambient_dim` – nonnegative integer (default: `None`); the dimension of the ambient space
- `lattice` – a toric lattice (default: `None`); the lattice in which the cone will live

If `ambient_dim` is omitted, then it will be inferred from the rank of `lattice`. If the `lattice` is omitted, then the default lattice of rank `ambient_dim` will be used.

A `ValueError` is raised if neither `ambient_dim` nor `lattice` are specified. It is also a `ValueError` to specify both `ambient_dim` and `lattice` unless the rank of `lattice` is equal to `ambient_dim`.

It is also a `ValueError` to specify a non-integer `p`.

OUTPUT:

A *ConvexRationalPolyhedralCone* representing the rearrangement cone of order `p` living in `lattice`, with ambient dimension `ambient_dim`. Each generating ray has the integer ring as its base ring.

A `ValueError` can be raised if the inputs are incompatible or insufficient. See the INPUT documentation for details.

ALGORITHM:

Suppose that the ambient space is of dimension n. The extreme directions of the rearrangement cone for $1 \leq p \leq n-1$ are given by [Jeong2017] Theorem 5.2.3. When $2 \leq p \leq n-2$ (that is, if we ignore $p = 1$ and $p = n-1$), they consist of

- the standard basis $\{e_1, e_2, \ldots, e_n\}$ for the ambient space, and
- the n vectors $(1, 1, \ldots, 1)^T - pe_i$ for $i = 1, 2, \ldots, n$.

Special cases are then given for $p = 1$ and $p = n - 1$ in the theorem. However in SageMath we don't need conically-independent extreme directions. We only need a generating set, because the *Cone()* function will eliminate any redundant generators. And one can easily verify that the special-case extreme directions for $p = 1$ and $p = n-1$ are contained in the conic hull of the $2n$ generators just described. The half space resulting from $p = n$ is also covered by this set of generators, so for all valid p we simply take the conic hull of those $2n$ vectors.

REFERENCES:

- [GJ2016], Section 4
- [HS2010], Example 2.21
- [Jeong2017], Section 5.2

EXAMPLES:

The rearrangement cones of order one are nonnegative orthants:

```
sage: orthant = cones.nonnegative_orthant(6)
sage: cones.rearrangement(1,6).is_equivalent(orthant)
True
```

```
>>> from sage.all import *
>>> orthant = cones.nonnegative_orthant(Integer(6))
>>> cones.rearrangement(Integer(1),Integer(6)).is_equivalent(orthant)
True
```

When `p` and `ambient_dim` are equal, the rearrangement cone is a half-space, so we expect its lineality to be one less than `ambient_dim` because it will contain a hyperplane but is not the entire space:

```
sage: cones.rearrangement(5,5).lineality()
4
```

```
>>> from sage.all import *
>>> cones.rearrangement(Integer(5),Integer(5)).lineality()
4
```

Jeong's Proposition 5.2.1 [Jeong2017] states that all rearrangement cones are proper when `p` is less than `ambient_dim`:

```
sage: all( cones.rearrangement(p, ambient_dim).is_proper()
....:              for ambient_dim in range(10)
....:              for p in range(1, ambient_dim) )
True
```

```
>>> from sage.all import *
>>> all( cones.rearrangement(p, ambient_dim).is_proper()
...              for ambient_dim in range(Integer(10))
...              for p in range(Integer(1), ambient_dim) )
True
```

Jeong's Corollary 5.2.4 [Jeong2017] states that if $p = n - 1$ in an n-dimensional ambient space, then the Lyapunov rank of the rearrangement cone is n, and that for all other $p > 1$ its Lyapunov rank is one:

```
sage: all( cones.rearrangement(p, ambient_dim).lyapunov_rank()
....:       ==
....:       ambient_dim
....:              for ambient_dim in range(2, 10)
....:              for p in [ ambient_dim-1 ] )
True
sage: all( cones.rearrangement(p, ambient_dim).lyapunov_rank() == 1
....:              for ambient_dim in range(3, 10)
....:              for p in range(2, ambient_dim-1) )
True
```

```
>>> from sage.all import *
>>> all( cones.rearrangement(p, ambient_dim).lyapunov_rank()
...       ==
...       ambient_dim
...              for ambient_dim in range(Integer(2), Integer(10))
...              for p in [ ambient_dim-Integer(1) ] )
True
>>> all( cones.rearrangement(p, ambient_dim).lyapunov_rank() == Integer(1)
...              for ambient_dim in range(Integer(3), Integer(10))
...              for p in range(Integer(2), ambient_dim-Integer(1)) )
True
```

`sage.geometry.cone_catalog.`**`schur`**`(`*ambient_dim=None*, *lattice=None*`)`

The Schur cone in `ambient_dim` dimensions, or living in `lattice`.

The Schur cone in n dimensions induces the majorization preordering on the ambient space. If $\{e_1, e_2, \ldots, e_n\}$ is the standard basis for the space, then its generators are $\{e_i - e_{i+1} \mid 1 \leq i \leq n - 1\}$. Its dual is the downward monotone cone.

INPUT:

- `ambient_dim` – nonnegative integer (default: `None`); the dimension of the ambient space
- `lattice` – a toric lattice (default: `None`); the lattice in which the cone will live

If `ambient_dim` is omitted, then it will be inferred from the rank of `lattice`. If the `lattice` is omitted, then the default lattice of rank `ambient_dim` will be used.

A ValueError is raised if neither ambient_dim nor lattice are specified. It is also a ValueError to specify both ambient_dim and lattice unless the rank of lattice is equal to ambient_dim.

OUTPUT:

A *ConvexRationalPolyhedralCone* representing the Schur cone living in lattice, with ambient dimension ambient_dim. Each generating ray has the integer ring as its base ring.

A ValueError can be raised if the inputs are incompatible or insufficient. See the INPUT documentation for details.

See also

downward_monotone()

REFERENCES:

- [GS2010], Section 3.1
- [IS2005], Example 7.3
- [SS2016], Example 7.4

EXAMPLES:

Verify the claim [SS2016] that the maximal angle between any two generators of the Schur cone and the nonnegative orthant in dimension five is $(3/4)\pi$:

```
sage: # needs sage.rings.number_fields
sage: P = cones.schur(5)
sage: Q = cones.nonnegative_orthant(5)
sage: G = ( g.change_ring(QQbar).normalized() for g in P )
sage: H = ( h.change_ring(QQbar).normalized() for h in Q )
sage: actual = max(arccos(u.inner_product(v)) for u in G for v in H)
sage: expected = 3*pi/4
sage: abs(actual - expected).n() < 1e-12
True
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_fields
>>> P = cones.schur(Integer(5))
>>> Q = cones.nonnegative_orthant(Integer(5))
>>> G = ( g.change_ring(QQbar).normalized() for g in P )
>>> H = ( h.change_ring(QQbar).normalized() for h in Q )
>>> actual = max(arccos(u.inner_product(v)) for u in G for v in H)
>>> expected = Integer(3)*pi/Integer(4)
>>> abs(actual - expected).n() < RealNumber('1e-12')
True
```

The dual of the Schur cone is the downward-monotone cone [GS2010], whose elements' entries are in non-increasing order:

```
sage: ambient_dim = ZZ.random_element(10)
sage: K = cones.schur(ambient_dim).dual()
sage: J = cones.downward_monotone(ambient_dim, K.lattice())
sage: K.is_equivalent(J)
True
```

```
>>> from sage.all import *
>>> ambient_dim = ZZ.random_element(Integer(10))
>>> K = cones.schur(ambient_dim).dual()
>>> J = cones.downward_monotone(ambient_dim, K.lattice())
>>> K.is_equivalent(J)
True
```

sage.geometry.cone_catalog.**trivial**(*ambient_dim=None*, *lattice=None*)

The trivial cone with no nonzero generators in `ambient_dim` dimensions, or living in `lattice`.

INPUT:

- `ambient_dim` – nonnegative integer (default: `None`); the dimension of the ambient space
- `lattice` – a toric lattice (default: `None`); the lattice in which the cone will live

If `ambient_dim` is omitted, then it will be inferred from the rank of `lattice`. If the `lattice` is omitted, then the default lattice of rank `ambient_dim` will be used.

A `ValueError` is raised if neither `ambient_dim` nor `lattice` are specified. It is also a `ValueError` to specify both `ambient_dim` and `lattice` unless the rank of `lattice` is equal to `ambient_dim`.

OUTPUT:

A *ConvexRationalPolyhedralCone* representing the trivial cone with no nonzero generators living in `lattice`, with ambient dimension `ambient_dim`.

A `ValueError` can be raised if the inputs are incompatible or insufficient. See the INPUT documentation for details.

EXAMPLES:

Construct the trivial cone, containing only the origin, in three dimensions:

```
sage: cones.trivial(3)
0-d cone in 3-d lattice N
```

```
>>> from sage.all import *
>>> cones.trivial(Integer(3))
0-d cone in 3-d lattice N
```

If a `lattice` is given, the trivial cone will live in that lattice:

```
sage: L = ToricLattice(3, 'M')
sage: cones.trivial(3, lattice=L)
0-d cone in 3-d lattice M
```

```
>>> from sage.all import *
>>> L = ToricLattice(Integer(3), 'M')
>>> cones.trivial(Integer(3), lattice=L)
0-d cone in 3-d lattice M
```

2.5.4 Find maximal angles between polyhedral convex cones

> **Warning**
>
> This module is considered internal and its contents are subject to change at any time without (deprecation) warning. The stable interface is `sage.geometry.cone.ConvexRationalPolyhedralCone.max_angle()`.

Finding the maximal (or equivalently, the minimal) angle between two polyhedral convex cones is a hard nonconvex optimization problem. The problem for a single cone was introduced in [IS2005], and was later extended in [SS2016] to two cones as a generalization of the principal angle between two vector subspaces.

Seeger and Sossa proposed an algorithm in [SS2016] to find maximal angles, and [Or2020] elaborates on that algorithm. It is this latest improvement that is implemented (more or less) by this module. The fact that perturbations of pointed cones may not change the answer too much [Or2024] is taken into consideration to avoid pathological cases.

This module is internal to SageMath; the interface presented to users consists of a public method, `sage.geometry.cone.ConvexRationalPolyhedralCone.max_angle()` for polyhedral convex cones. Even though all of the functions in this module are internal, some are more internal than others. There are a few functions that are used only in doctests, and not by any code that an end-user would run. Breaking somewhat with tradition, only those methods have been prefixed with an underscore.

`sage.geometry.cone_critical_angles.`**`check_gevp_feasibility`**`(`*cos_theta*, *xi*, *eta*, *G_I*, *G_I_c_T*, *H_J*, *H_J_c_T*, *epsilon*`)`

Determine if a solution to the generalized eigenvalue problem in Theorem 3 [Or2020] is feasible.

Implementation detail: we take four matrices that we are capable of computing as parameters instead, because we will be called in a nested loop "for all $I\dots$ and for all $J\dots$" The data corresponding to I should be computed only once, which means that we can't do it here – it needs to be done outside of the J loop. For symmetry (and to avoid relying on too many cross-function implementation details), we also insist that the J data be passed in.

INPUT:

- `cos_theta` – an eigenvalue corresponding to (ξ, η)
- `xi` – first component of the (ξ, η) eigenvector
- `eta` – second component of the (ξ, η) eigenvector
- `G_I` – the submatrix of G with columns indexed by I
- `G_I_c_T` – a matrix whose rows are the non-I columns of G
- `H_J` – the submatrix of H with columns indexed by J
- `H_J_c_T` – a matrix whose rows are the non-J columns of H
- `epsilon` – the tolerance to use when making comparisons

OUTPUT:

A triple containing (in order),

- a boolean,
- a vector in the cone P (of the same length as `xi`), and
- a vector in the cone Q (of the same length as `eta`).

If (ξ, η) is feasible, we return `(True, u, v)` where u and v are the vectors in P and Q respectively that form the the angle θ.

If (ξ, η) is **not** feasible, then we return `(False, 0, 0)` where `0` should be interpreted to mean the zero vector in the appropriate space.

EXAMPLES:

If ξ has any components less than "zero," it isn't feasible:

```
sage: from sage.geometry.cone_critical_angles import (
....:   check_gevp_feasibility)
sage: xi = vector(QQ, [-1,1])
sage: eta = vector(QQ, [1,1,1])
sage: check_gevp_feasibility(0,xi,eta,None,None,None,None,0)
(False, (0, 0), (0, 0, 0))
```

```
>>> from sage.all import *
>>> from sage.geometry.cone_critical_angles import (
...   check_gevp_feasibility)
>>> xi = vector(QQ, [-Integer(1),Integer(1)])
>>> eta = vector(QQ, [Integer(1),Integer(1),Integer(1)])
>>> check_gevp_feasibility(Integer(0),xi,eta,None,None,None,None,Integer(0))
(False, (0, 0), (0, 0, 0))
```

If η has any components less than "zero," it isn't feasible:

```
sage: from sage.geometry.cone_critical_angles import (
....:   check_gevp_feasibility)
sage: xi = vector(QQ, [2])
sage: eta = vector(QQ, [1,-4,4,5])
sage: check_gevp_feasibility(0,xi,eta,None,None,None,None,0)
(False, (0), (0, 0, 0, 0))
```

```
>>> from sage.all import *
>>> from sage.geometry.cone_critical_angles import (
...   check_gevp_feasibility)
>>> xi = vector(QQ, [Integer(2)])
>>> eta = vector(QQ, [Integer(1),-Integer(4),Integer(4),Integer(5)])
>>> check_gevp_feasibility(Integer(0),xi,eta,None,None,None,None,Integer(0))
(False, (0), (0, 0, 0, 0))
```

If ξ and η are equal and if G_I and H_J are not, then the copy of η that's been scaled by the norm of $G_I\xi$ generally won't satisfy its norm-equality constraint:

```
sage: from sage.geometry.cone_critical_angles import (
....:   check_gevp_feasibility)
sage: xi = vector(QQ, [1,1])
sage: eta = xi
sage: G_I = matrix.identity(QQ,2)
sage: H_J = 2*G_I
sage: check_gevp_feasibility(0,xi,eta,G_I,None,H_J,None,0)
(False, (0, 0), (0, 0))
```

```
>>> from sage.all import *
>>> from sage.geometry.cone_critical_angles import (
...   check_gevp_feasibility)
>>> xi = vector(QQ, [Integer(1),Integer(1)])
>>> eta = xi
>>> G_I = matrix.identity(QQ,Integer(2))
>>> H_J = Integer(2)*G_I
```

(continues on next page)

(continued from previous page)

```
>>> check_gevp_feasibility(Integer(0),xi,eta,G_I,None,H_J,None,Integer(0))
(False, (0, 0), (0, 0))
```

When $\cos\theta$ is zero, the inequality (42) in Theorem 7.3 [SS2016] is just an inner product with v which we can make positive by ensuring that all of the entries of H_J are positive. So, if any of the rows of `G_I_c_T` contain a negative entry, (42) will fail:

```
sage: from sage.geometry.cone_critical_angles import (
....:   check_gevp_feasibility)
sage: xi = vector(QQ, [1/2,1/2,1/2,1/2])
sage: eta = xi
sage: G_I = matrix.identity(QQ,4)
sage: G_I_c_T = matrix(QQ, [[0,-1,0,0]])
sage: H_J = G_I
sage: check_gevp_feasibility(0,xi,eta,G_I,G_I_c_T,H_J,None,0)
(False, (0, 0, 0, 0), (0, 0, 0, 0))
```

```
>>> from sage.all import *
>>> from sage.geometry.cone_critical_angles import (
...   check_gevp_feasibility)
>>> xi = vector(QQ, [Integer(1)/Integer(2),Integer(1)/Integer(2),Integer(1)/
↪Integer(2),Integer(1)/Integer(2)])
>>> eta = xi
>>> G_I = matrix.identity(QQ,Integer(4))
>>> G_I_c_T = matrix(QQ, [[Integer(0),-Integer(1),Integer(0),Integer(0)]])
>>> H_J = G_I
>>> check_gevp_feasibility(Integer(0),xi,eta,G_I,G_I_c_T,H_J,None,Integer(0))
(False, (0, 0, 0, 0), (0, 0, 0, 0))
```

Likewise we can make (43) fail in exactly the same way:

```
sage: from sage.geometry.cone_critical_angles import (
....:   check_gevp_feasibility)
sage: xi = vector(QQ, [1/2,1/2,1/2,1/2])
sage: eta = xi
sage: G_I = matrix.identity(QQ,4)
sage: G_I_c_T = matrix(QQ, [[0,1,0,0]])
sage: H_J = G_I
sage: H_J_c_T = matrix(QQ, [[0,-1,0,0]])
sage: check_gevp_feasibility(0,xi,eta,G_I,G_I_c_T,H_J,H_J_c_T,0)
(False, (0, 0, 0, 0), (0, 0, 0, 0))
```

```
>>> from sage.all import *
>>> from sage.geometry.cone_critical_angles import (
...   check_gevp_feasibility)
>>> xi = vector(QQ, [Integer(1)/Integer(2),Integer(1)/Integer(2),Integer(1)/
↪Integer(2),Integer(1)/Integer(2)])
>>> eta = xi
>>> G_I = matrix.identity(QQ,Integer(4))
>>> G_I_c_T = matrix(QQ, [[Integer(0),Integer(1),Integer(0),Integer(0)]])
>>> H_J = G_I
>>> H_J_c_T = matrix(QQ, [[Integer(0),-Integer(1),Integer(0),Integer(0)]])
```

(continues on next page)

(continued from previous page)

```
>>> check_gevp_feasibility(Integer(0),xi,eta,G_I,G_I_c_T,H_J,H_J_c_T,Integer(0))
(False, (0, 0, 0, 0), (0, 0, 0, 0))
```

Finally, if we ensure that everything works, we get back a feasible result along with the vectors (scaled ξ and η) that worked:

```
sage: from sage.geometry.cone_critical_angles import (
....:   check_gevp_feasibility)
sage: xi = vector(QQ, [1/2,1/2,1/2,1/2])
sage: eta = xi
sage: G_I = matrix.identity(QQ,4)
sage: G_I_c_T = matrix(QQ, [[0,1,0,0]])
sage: H_J = G_I
sage: H_J_c_T = matrix(QQ, [[0,1,0,0]])
sage: check_gevp_feasibility(0,xi,eta,G_I,G_I_c_T,H_J,H_J_c_T,0)
(True, (1/2, 1/2, 1/2, 1/2), (1/2, 1/2, 1/2, 1/2))
```

```
>>> from sage.all import *
>>> from sage.geometry.cone_critical_angles import (
...   check_gevp_feasibility)
>>> xi = vector(QQ, [Integer(1)/Integer(2),Integer(1)/Integer(2),Integer(1)/
↪Integer(2),Integer(1)/Integer(2)])
>>> eta = xi
>>> G_I = matrix.identity(QQ,Integer(4))
>>> G_I_c_T = matrix(QQ, [[Integer(0),Integer(1),Integer(0),Integer(0)]])
>>> H_J = G_I
>>> H_J_c_T = matrix(QQ, [[Integer(0),Integer(1),Integer(0),Integer(0)]])
>>> check_gevp_feasibility(Integer(0),xi,eta,G_I,G_I_c_T,H_J,H_J_c_T,Integer(0))
(True, (1/2, 1/2, 1/2, 1/2), (1/2, 1/2, 1/2, 1/2))
```

`sage.geometry.cone_critical_angles.`**`compute_gevp_M`**(*gs*, *hs*)

Compute the matrix M whose (i, j)-th entry is the inner product of `gs[i]` and `hs[j]`.

This is the “generalized Gram matrix” appearing in Proposition 6 in [Or2020]. For efficiency, we also return the minimal pair, whose inner product is minimal among the entries of M. This allows our consumer to bail out immediately (knowing the optimal pair!) if it turns out that the maximal angle is acute; i.e. if the smallest entry of M is nonnegative.

INPUT:

- `gs` – a linearly independent list of unit-norm generators for the cone P
- `hs` – a linearly independent list of unit-norm generators for the cone Q

OUTPUT: a tuple containing four elements, in order:

- The matrix M described in Proposition 6
- The minimal entry in the matrix M
- A vector in `gs` that achieves that minimal inner product along with the next element of the tuple
- A vector in `hs` that achieves the minimal inner product along with the previous element in the tuple

EXAMPLES:

```
sage: from sage.geometry.cone_critical_angles import compute_gevp_M
sage: P = Cone([ (1,2,0), (3,4,0) ])
sage: Q = Cone([ (-1,4,1), (5,-2,-1),  (-1,-1,5) ])
sage: gs = [g.change_ring(QQ) for g in P]
sage: hs = [h.change_ring(QQ) for h in Q]
sage: M = compute_gevp_M(gs, hs)[0]
sage: all( M[i][j] == gs[i].inner_product(hs[j])
....:       for i in range(P.nrays())
....:       for j in range(Q.nrays()) )
True
```

```
>>> from sage.all import *
>>> from sage.geometry.cone_critical_angles import compute_gevp_M
>>> P = Cone([ (Integer(1),Integer(2),Integer(0)), (Integer(3),Integer(4),
↪Integer(0)) ])
>>> Q = Cone([ (-Integer(1),Integer(4),Integer(1)), (Integer(5),-Integer(2),-
↪Integer(1)),  (-Integer(1),-Integer(1),Integer(5)) ])
>>> gs = [g.change_ring(QQ) for g in P]
>>> hs = [h.change_ring(QQ) for h in Q]
>>> M = compute_gevp_M(gs, hs)[Integer(0)]
>>> all( M[i][j] == gs[i].inner_product(hs[j])
...       for i in range(P.nrays())
...       for j in range(Q.nrays()) )
True
```

sage.geometry.cone_critical_angles.**gevp_licis**(G)

Return all nonempty subsets of indices for the columns of G that correspond to linearly independent sets (of columns of G).

Mnemonic: linearly independent column-index subsets (LICIS).

The returned lists are all sorted in the same (the natural) order; and are returned as lists so that they may be used to index into the rows/columns of matrices.

INPUT:

- G – the matrix whose linearly independent column index sets we want

OUTPUT:

A generator that returns sorted lists of natural numbers. Each generated list I is a set of indices corresponding to columns of G that, when considered as a set, is linearly independent.

EXAMPLES:

The linearly independent subsets of the matrix corresponding to a line (with two generators pointing in opposite directions) are the one-element subsets, since the only two-element subset isn't linearly independent:

```
sage: from sage.geometry.cone_critical_angles import gevp_licis
sage: K = Cone([(1,0),(-1,0)])
sage: G = matrix.column(K.rays())
sage: list(gevp_licis(G))
[[0], [1]]
```

```
>>> from sage.all import *
>>> from sage.geometry.cone_critical_angles import gevp_licis
```

(continues on next page)

(continued from previous page)

```
>>> K = Cone([(Integer(1),Integer(0)),(-Integer(1),Integer(0))])
>>> G = matrix.column(K.rays())
>>> list(gevp_licis(G))
[[0], [1]]
```

The matrix for the trivial cone has no linearly independent subsets, since we require them to be nonempty:

```
sage: from sage.geometry.cone_critical_angles import gevp_licis
sage: trivial_cone = cones.trivial(0)
sage: trivial_cone.is_trivial()
True
sage: list(gevp_licis(matrix.column(trivial_cone.rays())))
[]
```

```
>>> from sage.all import *
>>> from sage.geometry.cone_critical_angles import gevp_licis
>>> trivial_cone = cones.trivial(Integer(0))
>>> trivial_cone.is_trivial()
True
>>> list(gevp_licis(matrix.column(trivial_cone.rays())))
[]
```

All rays in the nonnegative orthant of R^n are linearly independent, so we should get back $2^n - 1$ subsets after accounting for the absence of the empty set:

```
sage: from sage.geometry.cone_critical_angles import gevp_licis
sage: K = cones.nonnegative_orthant(3)
sage: G = matrix.column(K.rays())
sage: len(list(gevp_licis(G))) == 2^(K.nrays()) - 1
True
```

```
>>> from sage.all import *
>>> from sage.geometry.cone_critical_angles import gevp_licis
>>> K = cones.nonnegative_orthant(Integer(3))
>>> G = matrix.column(K.rays())
>>> len(list(gevp_licis(G))) == Integer(2)**(K.nrays()) - Integer(1)
True
```

sage.geometry.cone_critical_angles.**max_angle**(*P*, *Q*, *exact*, *epsilon*)

Find the maximal angle between the cones P and Q.

This implements *sage.geometry.cone.ConvexRationalPolyhedralCone.max_angle()*, which should be fully documented.

EXAMPLES:

For the sake of the user interface, the argument validation for this function is performed in the associated cone method; we can therefore crash it by feeding it invalid input like an inadmissible cone:

```
sage: from sage.geometry.cone_critical_angles import max_angle
sage: K = cones.trivial(3)
sage: max_angle(K,K,True,0)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
IndexError: list index out of range
```

```
>>> from sage.all import *
>>> from sage.geometry.cone_critical_angles import max_angle
>>> K = cones.trivial(Integer(3))
>>> max_angle(K,K,True,Integer(0))
Traceback (most recent call last):
...
IndexError: list index out of range
```

sage.geometry.cone_critical_angles.**solve_gevp_nonzero**(*GG*, *HH*, *M*, *I*, *J*)

Solve the generalized eigenvalue problem in Theorem 3 [Or2020] for a nonzero eigenvalue using Propositions 3 and 5 [Or2020].

INPUT:

- `GG` – the matrix whose (i, j)-th entry is the inner product of g_i and g_j, which are in turn the i-th and j-th columns of the matrix G in Theorem 3 [Or2020]
- `HH` – the matrix whose (i, j)-th entry is the inner product of h_i and h_j, which are in turn the i-th and j-th columns of the matrix H in Theorem 3 [Or2020]
- `M` – the matrix whose (i, j)-th entry is the inner product of g_i and h_j as in Proposition 6 in [Or2020]
- `I` – a linearly independent column-index set for the matrix G that appears in Theorem 3 [Or2020]
- `J` – a linearly independent column-index set for the matrix H that appears in Theorem 3 [Or2020]

OUTPUT:

A generator of `(eigenvalue, xi, eta, multiplicity)` quartets where

- `eigenvalue` is a real eigenvalue of the system
- `xi` is the first (length `len(I)`) component of an eigenvector associated with `eigenvalue`
- `eta` is the second (length `len(J)`) component of an eigenvector associated with `eigenvalue`
- `multiplicity` is the dimension of the eigenspace associated with `eigenvalue`

Note that we do not return a basis for each eigenspace along with its eigenvalue. For the application we have in mind, an eigenspace of dimension greater than one (so, `multiplicity > 1`) is an error. As such, our return value is optimized for convenience in the non-error case, where there is only one eigenvector (spanning a one-dimensional eigenspace) associated with each eigenvalue.

ALGORITHM:

According to Proposition 5 [Or2020], the solutions corresponding to nonzero eigenvalues can be found by solving a smaller eigenvalue problem in only the variable ξ. So, we do that, and then solve for η in terms of ξ as described in the proposition.

EXAMPLES:

When the zero solutions are included, this function returns the same solutions as the naive method on the Schur cone in three dimensions:

```
sage: from itertools import chain
sage: from sage.geometry.cone_critical_angles import (
....:   _normalize_gevp_solution,
```

(continues on next page)

(continued from previous page)

```
....:   _solve_gevp_naive,
....:   gevp_licis,
....:   solve_gevp_nonzero,
....:   solve_gevp_zero)
sage: K = cones.schur(3)
sage: gs = [g.change_ring(AA).normalized() for g in K]
sage: G = matrix.column(gs)
sage: GG = G.transpose() * G
sage: G_index_sets = list(gevp_licis(G))
sage: all(
....:   set(
....:     _normalize_gevp_solution(s)
....:     for s in
....:     chain(
....:       solve_gevp_zero(GG, I, J),
....:       solve_gevp_nonzero(GG, GG, GG, I, J)
....:     )
....:   )
....:   ==
....:   set(
....:     _normalize_gevp_solution(s)
....:     for s in
....:     _solve_gevp_naive(GG,GG,GG,I,J)
....:   )
....:   for I in G_index_sets
....:   for J in G_index_sets
....: )
True
```

```
>>> from sage.all import *
>>> from itertools import chain
>>> from sage.geometry.cone_critical_angles import (
...   _normalize_gevp_solution,
...   _solve_gevp_naive,
...   gevp_licis,
...   solve_gevp_nonzero,
...   solve_gevp_zero)
>>> K = cones.schur(Integer(3))
>>> gs = [g.change_ring(AA).normalized() for g in K]
>>> G = matrix.column(gs)
>>> GG = G.transpose() * G
>>> G_index_sets = list(gevp_licis(G))
>>> all(
...   set(
...     _normalize_gevp_solution(s)
...     for s in
...     chain(
...       solve_gevp_zero(GG, I, J),
...       solve_gevp_nonzero(GG, GG, GG, I, J)
...     )
...   )
...   ==
```

(continues on next page)

(continued from previous page)

```
...   set(
...     _normalize_gevp_solution(s)
...     for s in
...     _solve_gevp_naive(GG,GG,GG,I,J)
...   )
...   for I in G_index_sets
...   for J in G_index_sets
... )
True
```

sage.geometry.cone_critical_angles.**solve_gevp_zero**(M, I, J)

Solve the generalized eigenvalue problem in Theorem 3 [Or2020] for a zero eigenvalue using Propositions 3 and 4 [Or2020].

INPUT:

- `M` – the matrix whose (i, j)-th entry is the inner product of g_i and h_j as in Proposition 6 [Or2020]
- `I` – a linearly independent column-index set for the matrix G that appears in Theorem 3 [Or2020]
- `J` – a linearly independent column-index set for the matrix H that appears in Theorem 3 [Or2020]

OUTPUT:

A generator of `(eigenvalue, xi, eta, multiplicity)` quartets where

- `eigenvalue` is zero (the eigenvalue of the system)
- `xi` is the first (length `len(I)`) component of an eigenvector associated with `eigenvalue`
- `eta` is the second (length `len(J)`) component of an eigenvector associated with `eigenvalue`
- `multiplicity` is the dimension of the eigenspace associated with `eigenvalue`

ALGORITHM:

Proposition 4 in [Or2020] is used.

EXAMPLES:

This particular configuration results in the zero matrix in the eigenvalue problem, so the only solutions correspond to the eigenvalue zero:

```
sage: from sage.geometry.cone_critical_angles import solve_gevp_zero
sage: K = cones.nonnegative_orthant(2)
sage: G = matrix.column(K.rays())
sage: GG = G.transpose() * G
sage: I = [0]
sage: J = [1]
sage: list(solve_gevp_zero(GG, I, J))
[(0, (1), (0), 2), (0, (0), (1), 2)]
```

```
>>> from sage.all import *
>>> from sage.geometry.cone_critical_angles import solve_gevp_zero
>>> K = cones.nonnegative_orthant(Integer(2))
>>> G = matrix.column(K.rays())
>>> GG = G.transpose() * G
>>> I = [Integer(0)]
>>> J = [Integer(1)]
```

(continues on next page)

(continued from previous page)

```
>>> list(solve_gevp_zero(GG, I, J))
[(0, (1), (0), 2), (0, (0), (1), 2)]
```

2.5.5 Rational polyhedral fans

This module was designed as a part of the framework for toric varieties (`variety`, `fano_variety`). While the emphasis is on complete full-dimensional fans, arbitrary fans are supported. Work with distinct lattices. The default lattice is `ToricLattice` N of the appropriate dimension. The only case when you must specify lattice explicitly is creation of a 0-dimensional fan, where dimension of the ambient space cannot be guessed.

A **rational polyhedral fan** is a *finite* collection of *strictly* convex rational polyhedral cones, such that the intersection of any two cones of the fan is a face of each of them and each face of each cone is also a cone of the fan.

AUTHORS:

- Andrey Novoseltsev (2010-05-15): initial version.
- Andrey Novoseltsev (2010-06-17): substantial improvement during review by Volker Braun.

EXAMPLES:

Use `Fan()` to construct fans "explicitly":

```
sage: fan = Fan(cones=[(0,1), (1,2)],
....:           rays=[(1,0), (0,1), (-1,0)])
sage: fan
Rational polyhedral fan in 2-d lattice N
```

```
>>> from sage.all import *
>>> fan = Fan(cones=[(Integer(0),Integer(1)), (Integer(1),Integer(2))],
...           rays=[(Integer(1),Integer(0)), (Integer(0),Integer(1)), (-Integer(1),
↪Integer(0))])
>>> fan
Rational polyhedral fan in 2-d lattice N
```

In addition to giving such lists of cones and rays you can also create cones first using `Cone()` and then combine them into a fan. See the documentation of `Fan()` for details.

In 2 dimensions there is a unique maximal fan determined by rays, and you can use `Fan2d()` to construct it:

```
sage: fan2d = Fan2d(rays=[(1,0), (0,1), (-1,0)])
sage: fan2d.is_equivalent(fan)
True
```

```
>>> from sage.all import *
>>> fan2d = Fan2d(rays=[(Integer(1),Integer(0)), (Integer(0),Integer(1)), (-
↪Integer(1),Integer(0))])
>>> fan2d.is_equivalent(fan)
True
```

But keep in mind that in higher dimensions the cone data is essential and cannot be omitted. Instead of building a fan from scratch, for this tutorial we will use an easy way to get two fans associated to `lattice polytopes`: `FaceFan()` and `NormalFan()`:

```
sage: fan1 = FaceFan(lattice_polytope.cross_polytope(3))
sage: fan2 = NormalFan(lattice_polytope.cross_polytope(3))
```

```
>>> from sage.all import *
>>> fan1 = FaceFan(lattice_polytope.cross_polytope(Integer(3)))
>>> fan2 = NormalFan(lattice_polytope.cross_polytope(Integer(3)))
```

Given such “automatic” fans, you may wonder what are their rays and cones:

```
sage: fan1.rays()
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1)
in 3-d lattice M
sage: fan1.generating_cones()
(3-d cone of Rational polyhedral fan in 3-d lattice M,
 3-d cone of Rational polyhedral fan in 3-d lattice M,
 3-d cone of Rational polyhedral fan in 3-d lattice M,
 3-d cone of Rational polyhedral fan in 3-d lattice M,
 3-d cone of Rational polyhedral fan in 3-d lattice M,
 3-d cone of Rational polyhedral fan in 3-d lattice M,
 3-d cone of Rational polyhedral fan in 3-d lattice M,
 3-d cone of Rational polyhedral fan in 3-d lattice M)
```

```
>>> from sage.all import *
>>> fan1.rays()
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1)
in 3-d lattice M
>>> fan1.generating_cones()
(3-d cone of Rational polyhedral fan in 3-d lattice M,
 3-d cone of Rational polyhedral fan in 3-d lattice M,
 3-d cone of Rational polyhedral fan in 3-d lattice M,
 3-d cone of Rational polyhedral fan in 3-d lattice M,
 3-d cone of Rational polyhedral fan in 3-d lattice M,
 3-d cone of Rational polyhedral fan in 3-d lattice M,
 3-d cone of Rational polyhedral fan in 3-d lattice M,
 3-d cone of Rational polyhedral fan in 3-d lattice M)
```

The last output is not very illuminating. Let’s try to improve it:

```
sage: for cone in fan1: print(cone.rays())
M( 0, 1, 0),
M( 0, 0, 1),
M(-1, 0, 0)
in 3-d lattice M
M( 0,  0, 1),
M(-1,  0, 0),
M( 0, -1, 0)
```

(continues on next page)

(continued from previous page)

```
in 3-d lattice M
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1)
in 3-d lattice M
M( 0, 1,  0),
M(-1, 0,  0),
M( 0, 0, -1)
in 3-d lattice M
M(1, 0,  0),
M(0, 1,  0),
M(0, 0, -1)
in 3-d lattice M
M(1, 0, 0),
M(0, 1, 0),
M(0, 0, 1)
in 3-d lattice M
M(1,  0, 0),
M(0,  0, 1),
M(0, -1, 0)
in 3-d lattice M
M(1,  0,  0),
M(0, -1,  0),
M(0,  0, -1)
in 3-d lattice M
```

```
>>> from sage.all import *
>>> for cone in fan1: print(cone.rays())
M( 0, 1, 0),
M( 0, 0, 1),
M(-1, 0, 0)
in 3-d lattice M
M( 0,  0, 1),
M(-1,  0, 0),
M( 0, -1, 0)
in 3-d lattice M
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1)
in 3-d lattice M
M( 0, 1,  0),
M(-1, 0,  0),
M( 0, 0, -1)
in 3-d lattice M
M(1, 0,  0),
M(0, 1,  0),
M(0, 0, -1)
in 3-d lattice M
M(1, 0, 0),
M(0, 1, 0),
M(0, 0, 1)
in 3-d lattice M
```

(continues on next page)

(continued from previous page)

```
M(1,  0, 0),
M(0,  0, 1),
M(0, -1, 0)
in 3-d lattice M
M(1,  0,  0),
M(0, -1,  0),
M(0,  0, -1)
in 3-d lattice M
```

You can also do

```
sage: for cone in fan1: print(cone.ambient_ray_indices())
(1, 2, 3)
(2, 3, 4)
(3, 4, 5)
(1, 3, 5)
(0, 1, 5)
(0, 1, 2)
(0, 2, 4)
(0, 4, 5)
```

```
>>> from sage.all import *
>>> for cone in fan1: print(cone.ambient_ray_indices())
(1, 2, 3)
(2, 3, 4)
(3, 4, 5)
(1, 3, 5)
(0, 1, 5)
(0, 1, 2)
(0, 2, 4)
(0, 4, 5)
```

to see indices of rays of the fan corresponding to each cone.

While the above cycles were over "cones in fan", it is obvious that we did not get ALL the cones: every face of every cone in a fan must also be in the fan, but all of the above cones were of dimension three. The reason for this behaviour is that in many cases it is enough to work with generating cones of the fan, i.e. cones which are not faces of bigger cones. When you do need to work with lower dimensional cones, you can easily get access to them using `cones()`:

```
sage: [cone.ambient_ray_indices() for cone in fan1.cones(2)]
[(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (0, 4),
 (2, 4), (3, 4), (0, 5), (1, 5), (3, 5), (4, 5)]
```

```
>>> from sage.all import *
>>> [cone.ambient_ray_indices() for cone in fan1.cones(Integer(2))]
[(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (0, 4),
 (2, 4), (3, 4), (0, 5), (1, 5), (3, 5), (4, 5)]
```

In fact, you do not have to type `.cones`:

```
sage: [cone.ambient_ray_indices() for cone in fan1(2)]
[(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (0, 4),
 (2, 4), (3, 4), (0, 5), (1, 5), (3, 5), (4, 5)]
```

```
>>> from sage.all import *
>>> [cone.ambient_ray_indices() for cone in fan1(Integer(2))]
[(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (0, 4),
 (2, 4), (3, 4), (0, 5), (1, 5), (3, 5), (4, 5)]
```

You may also need to know the inclusion relations between all of the cones of the fan. In this case check out `cone_lattice()`:

```
sage: L = fan1.cone_lattice()
sage: L
Finite lattice containing 28 elements with distinguished linear extension
sage: L.bottom()
0-d cone of Rational polyhedral fan in 3-d lattice M
sage: L.top()
Rational polyhedral fan in 3-d lattice M
sage: cone = L.level_sets()[2][0]
sage: cone
2-d cone of Rational polyhedral fan in 3-d lattice M
sage: sorted(L.hasse_diagram().neighbors(cone))
[1-d cone of Rational polyhedral fan in 3-d lattice M,
 1-d cone of Rational polyhedral fan in 3-d lattice M,
 3-d cone of Rational polyhedral fan in 3-d lattice M,
 3-d cone of Rational polyhedral fan in 3-d lattice M]
```

```
>>> from sage.all import *
>>> L = fan1.cone_lattice()
>>> L
Finite lattice containing 28 elements with distinguished linear extension
>>> L.bottom()
0-d cone of Rational polyhedral fan in 3-d lattice M
>>> L.top()
Rational polyhedral fan in 3-d lattice M
>>> cone = L.level_sets()[Integer(2)][Integer(0)]
>>> cone
2-d cone of Rational polyhedral fan in 3-d lattice M
>>> sorted(L.hasse_diagram().neighbors(cone))
[1-d cone of Rational polyhedral fan in 3-d lattice M,
 1-d cone of Rational polyhedral fan in 3-d lattice M,
 3-d cone of Rational polyhedral fan in 3-d lattice M,
 3-d cone of Rational polyhedral fan in 3-d lattice M]
```

You can check how “good” a fan is:

```
sage: fan1.is_complete()
True
sage: fan1.is_simplicial()
True
sage: fan1.is_smooth()
True
```

```
>>> from sage.all import *
>>> fan1.is_complete()
True
```

(continues on next page)

(continued from previous page)

```
>>> fan1.is_simplicial()
True
>>> fan1.is_smooth()
True
```

The face fan of the octahedron is really good! Time to remember that we have also constructed its normal fan:

```
sage: fan2.is_complete()
True
sage: fan2.is_simplicial()
False
sage: fan2.is_smooth()
False
```

```
>>> from sage.all import *
>>> fan2.is_complete()
True
>>> fan2.is_simplicial()
False
>>> fan2.is_smooth()
False
```

This one does have some "problems," but we can fix them:

```
sage: fan3 = fan2.make_simplicial()
sage: fan3.is_simplicial()
True
sage: fan3.is_smooth()
False
```

```
>>> from sage.all import *
>>> fan3 = fan2.make_simplicial()
>>> fan3.is_simplicial()
True
>>> fan3.is_smooth()
False
```

Note that we had to save the result of `make_simplicial()` in a new fan. Fans in Sage are immutable, so any operation that does change them constructs a new fan.

We can also make `fan3` smooth, but it will take a bit more work:

```
sage: # needs palp
sage: cube = lattice_polytope.cross_polytope(3).polar()
sage: sk = cube.skeleton_points(2)
sage: rays = [cube.point(p) for p in sk]
sage: fan4 = fan3.subdivide(new_rays=rays)
sage: fan4.is_smooth()
True
```

```
>>> from sage.all import *
>>> # needs palp
```

(continues on next page)

(continued from previous page)

```
>>> cube = lattice_polytope.cross_polytope(Integer(3)).polar()
>>> sk = cube.skeleton_points(Integer(2))
>>> rays = [cube.point(p) for p in sk]
>>> fan4 = fan3.subdivide(new_rays=rays)
>>> fan4.is_smooth()
True
```

Let's see how "different" are `fan2` and `fan4`:

```
sage: fan2.ngenerating_cones()
6
sage: fan2.nrays()
8
sage: fan4.ngenerating_cones()                                          #␣
↪needs palp
48
sage: fan4.nrays()                                                      #␣
↪needs palp
26
```

```
>>> from sage.all import *
>>> fan2.ngenerating_cones()
6
>>> fan2.nrays()
8
>>> fan4.ngenerating_cones()                                          #␣
↪needs palp
48
>>> fan4.nrays()                                                      #␣
↪needs palp
26
```

Smoothness does not come for free!

Please take a look at the rest of the available functions below and their complete descriptions. If you need any features that are missing, feel free to suggest them. (Or implement them on your own and submit a patch to Sage for inclusion!)

class sage.geometry.fan.**Cone_of_fan**(*ambient*, *ambient_ray_indices*)

Bases: *ConvexRationalPolyhedralCone*

Construct a cone belonging to a fan.

> **Warning**
>
> This class does not check that the input defines a valid cone of a fan. You must not construct objects of this class directly.

In addition to all of the properties of "regular" *cones*, such cones know their relation to the fan.

INPUT:

- `ambient` – fan whose cone is constructed

- `ambient_ray_indices` – increasing list or tuple of integers, indices of rays of `ambient` generating this cone

OUTPUT: cone of `ambient`

EXAMPLES:

The intended way to get objects of this class is the following:

```
sage: # needs palp
sage: fan = toric_varieties.P1xP1().fan()
sage: cone = fan.generating_cone(0); cone
2-d cone of Rational polyhedral fan in 2-d lattice N
sage: cone.ambient_ray_indices()
(0, 2)
sage: cone.star_generator_indices()
(0,)
```

```
>>> from sage.all import *
>>> # needs palp
>>> fan = toric_varieties.P1xP1().fan()
>>> cone = fan.generating_cone(Integer(0)); cone
2-d cone of Rational polyhedral fan in 2-d lattice N
>>> cone.ambient_ray_indices()
(0, 2)
>>> cone.star_generator_indices()
(0,)
```

star_generator_indices()

Return indices of generating cones of the "ambient fan" containing `self`.

OUTPUT: increasing `tuple` of integers

EXAMPLES:

```
sage: P1xP1 = toric_varieties.P1xP1()                                    #␣
↪needs palp
sage: cone = P1xP1.fan().generating_cone(0)                              #␣
↪needs palp
sage: cone.star_generator_indices()                                      #␣
↪needs palp
(0,)
```

```
>>> from sage.all import *
>>> P1xP1 = toric_varieties.P1xP1()                                    #␣
↪needs palp
>>> cone = P1xP1.fan().generating_cone(Integer(0))                      ␣
↪      # needs palp
>>> cone.star_generator_indices()                                      #␣
↪needs palp
(0,)
```

star_generators()

Return indices of generating cones of the "ambient fan" containing `self`.

OUTPUT: increasing `tuple` of integers

EXAMPLES:

```
sage: P1xP1 = toric_varieties.P1xP1()                                           #␣
↪needs palp
sage: cone = P1xP1.fan().generating_cone(0)                                     #␣
↪needs palp
sage: cone.star_generators()                                                    #␣
↪needs palp
(2-d cone of Rational polyhedral fan in 2-d lattice N,)
```

```
>>> from sage.all import *
>>> P1xP1 = toric_varieties.P1xP1()                                           #␣
↪needs palp
>>> cone = P1xP1.fan().generating_cone(Integer(0))                             ␣
↪     # needs palp
>>> cone.star_generators()                                                    #␣
↪needs palp
(2-d cone of Rational polyhedral fan in 2-d lattice N,)
```

sage.geometry.fan.**FaceFan**(*polytope*, *lattice=None*)

Construct the face fan of the given rational `polytope`.

INPUT:

- `polytope` – a *polytope* over $\mathbf{Q}$ or a *lattice polytope*. A (not necessarily full-dimensional) polytope containing the origin in its `relative interior`.
- `lattice` – *ToricLattice*, $\mathbf{Z}^n$, or any other object that behaves like these. If not specified, an attempt will be made to determine an appropriate toric lattice automatically.

OUTPUT: *rational polyhedral fan*

See also *NormalFan()*.

EXAMPLES:

Let's construct the fan corresponding to the product of two projective lines:

```
sage: diamond = lattice_polytope.cross_polytope(2)
sage: P1xP1 = FaceFan(diamond)
sage: P1xP1.rays()
M( 1,  0),
M( 0,  1),
M(-1,  0),
M( 0, -1)
in 2-d lattice M
sage: for cone in P1xP1: print(cone.rays())
M(-1,  0),
M( 0, -1)
in 2-d lattice M
M( 0, 1),
M(-1, 0)
in 2-d lattice M
M(1, 0),
M(0, 1)
in 2-d lattice M
M(1,  0),
```

(continues on next page)

(continued from previous page)

```
M(0, -1)
in 2-d lattice M
```

```
>>> from sage.all import *
>>> diamond = lattice_polytope.cross_polytope(Integer(2))
>>> P1xP1 = FaceFan(diamond)
>>> P1xP1.rays()
M( 1,  0),
M( 0,  1),
M(-1,  0),
M( 0, -1)
in 2-d lattice M
>>> for cone in P1xP1: print(cone.rays())
M(-1,  0),
M( 0, -1)
in 2-d lattice M
M( 0, 1),
M(-1, 0)
in 2-d lattice M
M(1, 0),
M(0, 1)
in 2-d lattice M
M(1,  0),
M(0, -1)
in 2-d lattice M
```

`sage.geometry.fan.Fan`(*cones*, *rays=None*, *lattice=None*, *check=True*, *normalize=True*, *is_complete=None*, *virtual_rays=None*, *discard_faces=False*, *allow_arrangement=False*)

Construct a rational polyhedral fan.

Note

Approximate time to construct a fan consisting of n cones is $n^2/5$ seconds. That is half an hour for 100 cones. This time can be significantly reduced in the future, but it is still likely to be $\sim n^2$ (with, say, $/500$ instead of $/5$). If you know that your input does form a valid fan, use `check=False` option to skip consistency checks.

INPUT:

- `cones` – list of either *Cone* objects or lists of integers interpreted as indices of generating rays in `rays`. These must be only **maximal** cones of the fan, unless `discard_faces=True` or `allow_arrangement=True` option is specified;
- `rays` – list of rays given as list or vectors convertible to the rational extension of `lattice`. If `cones` are given by *Cone* objects `rays` may be determined automatically. You still may give them explicitly to ensure a particular order of rays in the fan. In this case you must list all rays that appear in `cones`. You can give "extra" ones if it is convenient (e.g. if you have a big list of rays for several fans), but all "extra" rays will be discarded;
- `lattice` – *ToricLattice*, $\mathbf{Z}^n$, or any other object that behaves like these. If not specified, an attempt will be made to determine an appropriate toric lattice automatically;
- `check` – by default the input data will be checked for correctness (e.g. that intersection of any two given cones is a face of each), unless `allow_arrangement=True` option is specified. If you know for sure that

the input is correct, you may significantly decrease construction time using `check=False` option;

- `normalize` – you can further speed up construction using `normalize=False` option. In this case `cones` must be a list of **sorted** `tuples` and `rays` must be immutable primitive vectors in `lattice`. In general, you should not use this option, it is designed for code optimization and does not give as drastic improvement in speed as the previous one;
- `is_complete` – every fan can determine on its own if it is complete or not, however it can take quite a bit of time for "big" fans with many generating cones. On the other hand, in some situations it is known in advance that a certain fan is complete. In this case you can pass `is_complete=True` option to speed up some computations. You may also pass `is_complete=False` option, although it is less likely to be beneficial. Of course, passing a wrong value can compromise the integrity of data structures of the fan and lead to wrong results, so you should be very careful if you decide to use this option;
- `virtual_rays` – (optional, computed automatically if needed) a list of ray generators to be used for `virtual_rays()`;
- `discard_faces` – by default, the fan constructor expects the list of **maximal** cones, unless `allow_arrangement=True` option is specified. If you provide "extra" ones and leave `allow_arrangement=False` (default) and `check=True` (default), an exception will be raised. If you provide "extra" cones and set `allow_arrangement=False` (default) and `check=False`, you may get wrong results as assumptions on internal data structures will be invalid. If you want the fan constructor to select the maximal cones from the given input, you may provide `discard_faces=True` option (it works both for `check=True` and `check=False`).
- `allow_arrangement` – by default (`allow_arrangement=False`), the fan constructor expects that the intersection of any two given cones is a face of each. If `allow_arrangement=True` option is specified, then construct a rational polyhedralfan from the cone arrangement, so that the union of the cones in the polyhedral fan equals to the union of the given cones, and each given cone is the union of some cones in the polyhedral fan.

OUTPUT: a *fan*

See also

In 2 dimensions you can cyclically order the rays. Hence the rays determine a unique maximal fan without having to specify the cones, and you can use `Fan2d()` to construct this fan from just the rays.

EXAMPLES:

Let's construct a fan corresponding to the projective plane in several ways:

```
sage: cone1 = Cone([(1,0), (0,1)])
sage: cone2 = Cone([(0,1), (-1,-1)])
sage: cone3 = Cone([(-1,-1), (1,0)])
sage: P2 = Fan([cone1, cone2, cone2])
Traceback (most recent call last):
...
ValueError: you have provided 3 cones, but only 2 of them are maximal!
Use discard_faces=True if you indeed need to construct a fan from
these cones.
```

```
>>> from sage.all import *
>>> cone1 = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> cone2 = Cone([(Integer(0),Integer(1)), (-Integer(1),-Integer(1))])
>>> cone3 = Cone([(-Integer(1),-Integer(1)), (Integer(1),Integer(0))])
>>> P2 = Fan([cone1, cone2, cone2])
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
ValueError: you have provided 3 cones, but only 2 of them are maximal!
Use discard_faces=True if you indeed need to construct a fan from
these cones.
```

Oops! There was a typo and `cone2` was listed twice as a generating cone of the fan. If it was intentional (e.g. the list of cones was generated automatically and it is possible that it contains repetitions or faces of other cones), use `discard_faces=True` option:

```
sage: P2 = Fan([cone1, cone2, cone2], discard_faces=True)
sage: P2.ngenerating_cones()
2
```

```
>>> from sage.all import *
>>> P2 = Fan([cone1, cone2, cone2], discard_faces=True)
>>> P2.ngenerating_cones()
2
```

However, in this case it was definitely a typo, since the fan of $\mathbb{P}^2$ has 3 maximal cones:

```
sage: P2 = Fan([cone1, cone2, cone3])
sage: P2.ngenerating_cones()
3
```

```
>>> from sage.all import *
>>> P2 = Fan([cone1, cone2, cone3])
>>> P2.ngenerating_cones()
3
```

Looks better. An alternative way is

```
sage: rays = [(1,0), (0,1), (-1,-1)]
sage: cones = [(0,1), (1,2), (2,0)]
sage: P2a = Fan(cones, rays)
sage: P2a.ngenerating_cones()
3
sage: P2 == P2a
False
```

```
>>> from sage.all import *
>>> rays = [(Integer(1),Integer(0)), (Integer(0),Integer(1)), (-Integer(1),-
↪Integer(1))]
>>> cones = [(Integer(0),Integer(1)), (Integer(1),Integer(2)), (Integer(2),
↪Integer(0))]
>>> P2a = Fan(cones, rays)
>>> P2a.ngenerating_cones()
3
>>> P2 == P2a
False
```

That may seem wrong, but it is not:

```
sage: P2.is_equivalent(P2a)
True
```

```
>>> from sage.all import *
>>> P2.is_equivalent(P2a)
True
```

See `is_equivalent()` for details.

Yet another way to construct this fan is

```
sage: P2b = Fan(cones, rays, check=False)
sage: P2b.ngenerating_cones()
3
sage: P2a == P2b
True
```

```
>>> from sage.all import *
>>> P2b = Fan(cones, rays, check=False)
>>> P2b.ngenerating_cones()
3
>>> P2a == P2b
True
```

If you try the above examples, you are likely to notice the difference in speed, so when you are sure that everything is correct, it is a good idea to use `check=False` option. On the other hand, it is usually **NOT** a good idea to use `normalize=False` option:

```
sage: P2c = Fan(cones, rays, check=False, normalize=False)
Traceback (most recent call last):
...
AttributeError: 'tuple' object has no attribute 'parent'...
```

```
>>> from sage.all import *
>>> P2c = Fan(cones, rays, check=False, normalize=False)
Traceback (most recent call last):
...
AttributeError: 'tuple' object has no attribute 'parent'...
```

Yet another way is to use functions `FaceFan()` and `NormalFan()` to construct fans from `lattice polytopes`.

We have not yet used `lattice` argument, since if was determined automatically:

```
sage: P2.lattice()
2-d lattice N
sage: P2b.lattice()
2-d lattice N
```

```
>>> from sage.all import *
>>> P2.lattice()
2-d lattice N
>>> P2b.lattice()
2-d lattice N
```

However, it is necessary to specify it explicitly if you want to construct a fan without rays or cones:

```
sage: Fan([], [])
Traceback (most recent call last):
...
ValueError: you must specify the lattice
when you construct a fan without rays and cones!
sage: F = Fan([], [], lattice=ToricLattice(2, "L"))
sage: F
Rational polyhedral fan in 2-d lattice L
sage: F.lattice_dim()
2
sage: F.dim()
0
```

```
>>> from sage.all import *
>>> Fan([], [])
Traceback (most recent call last):
...
ValueError: you must specify the lattice
when you construct a fan without rays and cones!
>>> F = Fan([], [], lattice=ToricLattice(Integer(2), "L"))
>>> F
Rational polyhedral fan in 2-d lattice L
>>> F.lattice_dim()
2
>>> F.dim()
0
```

In the following examples, we test the `allow_arrangement=True` option. See Issue #25122.

The intersection of the two cones is not a face of each. Therefore, they do not belong to the same rational polyhedral fan:

```
sage: c1 = Cone([(-2,-1,1), (-2,1,1), (2,1,1), (2,-1,1)])
sage: c2 = Cone([(-1,-2,1), (-1,2,1), (1,2,1), (1,-2,1)])
sage: c1.intersection(c2).is_face_of(c1)
False
sage: c1.intersection(c2).is_face_of(c2)
False
sage: Fan([c1, c2])
Traceback (most recent call last):
...
ValueError: these cones cannot belong to the same fan!
...
```

```
>>> from sage.all import *
>>> c1 = Cone([(-Integer(2),-Integer(1),Integer(1)), (-Integer(2),Integer(1),
↪Integer(1)), (Integer(2),Integer(1),Integer(1)), (Integer(2),-Integer(1),
↪Integer(1))])
>>> c2 = Cone([(-Integer(1),-Integer(2),Integer(1)), (-Integer(1),Integer(2),
↪Integer(1)), (Integer(1),Integer(2),Integer(1)), (Integer(1),-Integer(2),
↪Integer(1))])
>>> c1.intersection(c2).is_face_of(c1)
```

(continues on next page)

(continued from previous page)

```
False
>>> c1.intersection(c2).is_face_of(c2)
False
>>> Fan([c1, c2])
Traceback (most recent call last):
...
ValueError: these cones cannot belong to the same fan!
...
```

Let's construct the fan using `allow_arrangement=True` option:

```
sage: fan = Fan([c1, c2], allow_arrangement=True)
sage: fan.ngenerating_cones()
5
```

```
>>> from sage.all import *
>>> fan = Fan([c1, c2], allow_arrangement=True)
>>> fan.ngenerating_cones()
5
```

Another example where cone c2 is inside cone c1:

```
sage: c1 = Cone([(4, 0, 0), (0, 4, 0), (0, 0, 4)])
sage: c2 = Cone([(2, 1, 1), (1, 2, 1), (1, 1, 2)])
sage: fan = Fan([c1, c2], allow_arrangement=True)
sage: fan.ngenerating_cones()
7
sage: fan.plot()                                                            #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> c1 = Cone([(Integer(4), Integer(0), Integer(0)), (Integer(0), Integer(4),␣
↪Integer(0)), (Integer(0), Integer(0), Integer(4))])
>>> c2 = Cone([(Integer(2), Integer(1), Integer(1)), (Integer(1), Integer(2),␣
↪Integer(1)), (Integer(1), Integer(1), Integer(2))])
>>> fan = Fan([c1, c2], allow_arrangement=True)
>>> fan.ngenerating_cones()
7
>>> fan.plot()                                                              #␣
↪needs sage.plot
Graphics3d Object
```

Cones of different dimension:

```
sage: c1 = Cone([(1,0), (0,1)])
sage: c2 = Cone([(2,1)])
sage: c3 = Cone([(-1,-2)])
sage: fan = Fan([c1, c2, c3], allow_arrangement=True)
sage: for cone in sorted(fan.generating_cones()): print(sorted(cone.rays()))
[N(-1, -2)]
[N(0, 1), N(1, 2)]
```

(continues on next page)

(continued from previous page)

```
[N(1, 0), N(2, 1)]
[N(1, 2), N(2, 1)]
```

```
>>> from sage.all import *
>>> c1 = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> c2 = Cone([(Integer(2),Integer(1))])
>>> c3 = Cone([(-Integer(1),-Integer(2))])
>>> fan = Fan([c1, c2, c3], allow_arrangement=True)
>>> for cone in sorted(fan.generating_cones()): print(sorted(cone.rays()))
[N(-1, -2)]
[N(0, 1), N(1, 2)]
[N(1, 0), N(2, 1)]
[N(1, 2), N(2, 1)]
```

A 3-d cone and a 1-d cone:

```
sage: c3 = Cone([[0, 1, 1], [1, 0, 1], [0, -1, 1], [-1, 0, 1]])
sage: c1 = Cone([[0, 0, 1]])
sage: fan1 = Fan([c1, c3], allow_arrangement=True)
sage: fan1.plot()                                                          #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> c3 = Cone([[Integer(0), Integer(1), Integer(1)], [Integer(1), Integer(0),␣
↪Integer(1)], [Integer(0), -Integer(1), Integer(1)], [-Integer(1), Integer(0),␣
↪Integer(1)]])
>>> c1 = Cone([[Integer(0), Integer(0), Integer(1)]])
>>> fan1 = Fan([c1, c3], allow_arrangement=True)
>>> fan1.plot()                                                          #␣
↪needs sage.plot
Graphics3d Object
```

A 3-d cone and two 2-d cones:

```
sage: c2v = Cone([[0, 1, 1], [0, -1, 1]])
sage: c2h = Cone([[1, 0, 1], [-1, 0, 1]])
sage: fan2 = Fan([c2v, c2h, c3], allow_arrangement=True)
sage: fan2.is_simplicial()
True
sage: fan2.is_equivalent(fan1)
True
```

```
>>> from sage.all import *
>>> c2v = Cone([[Integer(0), Integer(1), Integer(1)], [Integer(0), -Integer(1),␣
↪Integer(1)]])
>>> c2h = Cone([[Integer(1), Integer(0), Integer(1)], [-Integer(1), Integer(0),␣
↪Integer(1)]])
>>> fan2 = Fan([c2v, c2h, c3], allow_arrangement=True)
>>> fan2.is_simplicial()
True
```

(continues on next page)

(continued from previous page)

```
>>> fan2.is_equivalent(fan1)
True
```

sage.geometry.fan.**Fan2d**(*rays*, *lattice=None*)

Construct the maximal 2-d fan with given `rays`.

In two dimensions we can uniquely construct a fan from just rays, just by cyclically ordering the rays and constructing as many cones as possible. This is why we implement a special constructor for this case.

INPUT:

- `rays` – list of rays given as list or vectors convertible to the rational extension of `lattice`. Duplicate rays are removed without changing the ordering of the remaining rays.
- `lattice` – *ToricLattice*, $\mathbf{Z}^n$, or any other object that behaves like these. If not specified, an attempt will be made to determine an appropriate toric lattice automatically.

EXAMPLES:

```
sage: Fan2d([(0,1), (1,0)])
Rational polyhedral fan in 2-d lattice N
sage: Fan2d([], lattice=ToricLattice(2, 'myN'))
Rational polyhedral fan in 2-d lattice myN
```

```
>>> from sage.all import *
>>> Fan2d([(Integer(0),Integer(1)), (Integer(1),Integer(0))])
Rational polyhedral fan in 2-d lattice N
>>> Fan2d([], lattice=ToricLattice(Integer(2), 'myN'))
Rational polyhedral fan in 2-d lattice myN
```

The ray order is as specified, even if it is not the cyclic order:

```
sage: fan1 = Fan2d([(0,1), (1,0)])
sage: fan1.rays()
N(0, 1),
N(1, 0)
in 2-d lattice N

sage: fan2 = Fan2d([(1,0), (0,1)])
sage: fan2.rays()
N(1, 0),
N(0, 1)
in 2-d lattice N

sage: fan1 == fan2, fan1.is_equivalent(fan2)
(False, True)

sage: fan = Fan2d([(1,1), (-1,-1), (1,-1), (-1,1)])
sage: [cone.ambient_ray_indices() for cone in fan]
[(2, 1), (1, 3), (3, 0), (0, 2)]
sage: fan.is_complete()
True
```

```
>>> from sage.all import *
>>> fan1 = Fan2d([(Integer(0),Integer(1)), (Integer(1),Integer(0))])
>>> fan1.rays()
N(0, 1),
N(1, 0)
in 2-d lattice N

>>> fan2 = Fan2d([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> fan2.rays()
N(1, 0),
N(0, 1)
in 2-d lattice N

>>> fan1 == fan2, fan1.is_equivalent(fan2)
(False, True)

>>> fan = Fan2d([(Integer(1),Integer(1)), (-Integer(1),-Integer(1)), (Integer(1),-
↪Integer(1)), (-Integer(1),Integer(1))])
>>> [cone.ambient_ray_indices() for cone in fan]
[(2, 1), (1, 3), (3, 0), (0, 2)]
>>> fan.is_complete()
True
```

sage.geometry.fan.**NormalFan**(*polytope*, *lattice=None*)

Construct the normal fan of the given rational `polytope`.

This returns the inner normal fan. For the outer normal fan, use `NormalFan(-P)`.

INPUT:

- `polytope` – a full-dimensional *polytope* over **Q** or:class:*latticepolytope* < *sage.geometry.lattice_polytope.LatticePolytopeClass* >.
- `lattice` – *ToricLattice*, $\mathbf{Z}^n$, or any other object that behaves like these. If not specified, an attempt will be made to determine an appropriate toric lattice automatically.

OUTPUT: *rational polyhedral fan*

See also *FaceFan()*.

EXAMPLES:

Let's construct the fan corresponding to the product of two projective lines:

```
sage: square = LatticePolytope([(1,1), (-1,1), (-1,-1), (1,-1)])
sage: P1xP1 = NormalFan(square)
sage: P1xP1.rays()
N( 1,  0),
N( 0,  1),
N(-1,  0),
N( 0, -1)
in 2-d lattice N
sage: for cone in P1xP1: print(cone.rays())
N(-1,  0),
N( 0, -1)
in 2-d lattice N
N(1,  0),
```

(continues on next page)

(continued from previous page)

```
N(0, -1)
in 2-d lattice N
N(1, 0),
N(0, 1)
in 2-d lattice N
N( 0, 1),
N(-1, 0)
in 2-d lattice N

sage: cuboctahed = polytopes.cuboctahedron()
sage: NormalFan(cuboctahed)
Rational polyhedral fan in 3-d lattice N
```

```
>>> from sage.all import *
>>> square = LatticePolytope([(Integer(1),Integer(1)), (-Integer(1),Integer(1)),␣
↪(-Integer(1),-Integer(1)), (Integer(1),-Integer(1))])
>>> P1xP1 = NormalFan(square)
>>> P1xP1.rays()
N( 1,  0),
N( 0,  1),
N(-1,  0),
N( 0, -1)
in 2-d lattice N
>>> for cone in P1xP1: print(cone.rays())
N(-1,  0),
N( 0, -1)
in 2-d lattice N
N(1,  0),
N(0, -1)
in 2-d lattice N
N(1, 0),
N(0, 1)
in 2-d lattice N
N( 0, 1),
N(-1, 0)
in 2-d lattice N

>>> cuboctahed = polytopes.cuboctahedron()
>>> NormalFan(cuboctahed)
Rational polyhedral fan in 3-d lattice N
```

class sage.geometry.fan.**RationalPolyhedralFan**(*cones*, *rays*, *lattice*, *is_complete=None*, *virtual_rays=None*)

Bases: *IntegralRayCollection*, Callable, Container

Create a rational polyhedral fan.

> **Warning**
>
> This class does not perform any checks of correctness of input nor does it convert input into the standard representation. Use `Fan()` to construct fans from "raw data" or `FaceFan()` and `NormalFan()` to get fans associated to polytopes.

Fans are immutable, but they cache most of the returned values.

INPUT:

- `cones` – list of generating cones of the fan, each cone given as a list of indices of its generating rays in `rays`;
- `rays` – list of immutable primitive vectors in `lattice` consisting of exactly the rays of the fan (i.e. no "extra" ones);
- `lattice` – `ToricLattice`, $\mathbf{Z}^n$, or any other object that behaves like these. If `None`, it will be determined as `parent()` of the first ray. Of course, this cannot be done if there are no rays, so in this case you must give an appropriate `lattice` directly;
- `is_complete` – if given, must be `True` or `False` depending on whether this fan is complete or not. By default, it will be determined automatically if necessary;
- `virtual_rays` – if given, must be a list of immutable primitive vectors in `lattice`, see `virtual_rays()` for details. By default, it will be determined automatically if necessary.

OUTPUT:

rational polyhedral fan

Gale_transform()

Return the Gale transform of `self`.

OUTPUT: a matrix over ZZ

EXAMPLES:

```
sage: fan = toric_varieties.P1xP1().fan()                                    #␣
↪needs palp
sage: fan.Gale_transform()                                                   #␣
↪needs palp
[ 1  1  0  0 -2]
[ 0  0  1  1 -2]
sage: _.base_ring()                                                          #␣
↪needs palp
Integer Ring
```

```
>>> from sage.all import *
>>> fan = toric_varieties.P1xP1().fan()                                     #␣
↪needs palp
>>> fan.Gale_transform()                                                    #␣
↪needs palp
[ 1  1  0  0 -2]
[ 0  0  1  1 -2]
>>> _.base_ring()                                                           #␣
↪needs palp
Integer Ring
```

Stanley_Reisner_ideal (*ring*)

Return the Stanley-Reisner ideal.

INPUT:

- A polynomial ring in `self.nrays()` variables.

OUTPUT: the Stanley-Reisner ideal in the given polynomial ring

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: fan.Stanley_Reisner_ideal(PolynomialRing(QQ, 5, 'A, B, C, D, E'))
Ideal (A*E, C*D, A*B*C, B*D*E) of
 Multivariate Polynomial Ring in A, B, C, D, E over Rational Field
```

```
>>> from sage.all import *
>>> fan = Fan([[Integer(0),Integer(1),Integer(3)], [Integer(3),Integer(4)],␣
↪[Integer(2),Integer(0)], [Integer(1),Integer(2),Integer(4)]],
...           [(-Integer(3), -Integer(2), Integer(1)), (Integer(0),␣
↪Integer(0), Integer(1)), (Integer(3), -Integer(2), Integer(1)), (-
↪Integer(1), -Integer(1), Integer(1)), (Integer(1), -Integer(1),␣
↪Integer(1))])
>>> fan.Stanley_Reisner_ideal(PolynomialRing(QQ, Integer(5), 'A, B, C, D, E'))
Ideal (A*E, C*D, A*B*C, B*D*E) of
 Multivariate Polynomial Ring in A, B, C, D, E over Rational Field
```

cartesian_product (*other*, *lattice=None*)

Return the Cartesian product of `self` with `other`.

INPUT:

- `other` – a *rational polyhedral fan*;
- `lattice` – (optional) the ambient lattice for the Cartesian product fan. By default, the direct sum of the ambient lattices of `self` and `other` is constructed.

OUTPUT:

a *fan* whose cones are all pairwise Cartesian products of the cones of `self` and `other`

EXAMPLES:

```
sage: K = ToricLattice(1, 'K')
sage: fan1 = Fan([[0],[1]], [(1,),(-1,)], lattice=K)
sage: L = ToricLattice(2, 'L')
sage: fan2 = Fan(rays=[(1,0), (0,1), (-1,-1)],
....:            cones=[[0,1], [1,2], [2,0]], lattice=L)
sage: fan1.cartesian_product(fan2)
Rational polyhedral fan in 3-d lattice K+L
sage: _.ngenerating_cones()
6
```

```
>>> from sage.all import *
>>> K = ToricLattice(Integer(1), 'K')
>>> fan1 = Fan([[Integer(0)],[Integer(1)]], [(Integer(1),),(-Integer(1),)],␣
```

(continues on next page)

(continued from previous page)

```
↪lattice=K)
>>> L = ToricLattice(Integer(2), 'L')
>>> fan2 = Fan(rays=[(Integer(1),Integer(0)), (Integer(0),Integer(1)), (-
↪Integer(1),-Integer(1))],
...           cones=[[Integer(0),Integer(1)], [Integer(1),Integer(2)],␣
↪[Integer(2),Integer(0)]], lattice=L)
>>> fan1.cartesian_product(fan2)
Rational polyhedral fan in 3-d lattice K+L
>>> _.ngenerating_cones()
6
```

common_refinement(*other*)

Return the common refinement of this fan and `other`.

INPUT:

- `other` – a *fan* in the same `lattice()` and with the same support as this fan

OUTPUT: a *fan*

EXAMPLES:

Refining a fan with itself gives itself:

```
sage: F0 = Fan2d([(1,0), (0,1), (-1,0), (0,-1)])
sage: F0.common_refinement(F0) == F0
True
```

```
>>> from sage.all import *
>>> F0 = Fan2d([(Integer(1),Integer(0)), (Integer(0),Integer(1)), (-
↪Integer(1),Integer(0)), (Integer(0),-Integer(1))])
>>> F0.common_refinement(F0) == F0
True
```

A more complex example with complete fans:

```
sage: F1 = Fan([[0],[1]], [(1,),(-1,)])
sage: F2 = Fan2d([(1,0), (1,1), (0,1), (-1,0), (0,-1)])
sage: F3 = F2.cartesian_product(F1)
sage: F4 = F1.cartesian_product(F2)
sage: FF = F3.common_refinement(F4)
sage: F3.ngenerating_cones()
10
sage: F4.ngenerating_cones()
10
sage: FF.ngenerating_cones()
13
```

```
>>> from sage.all import *
>>> F1 = Fan([[Integer(0)],[Integer(1)]], [(Integer(1),),(-Integer(1),)])
>>> F2 = Fan2d([(Integer(1),Integer(0)), (Integer(1),Integer(1)), (Integer(0),
↪Integer(1)), (-Integer(1),Integer(0)), (Integer(0),-Integer(1))])
>>> F3 = F2.cartesian_product(F1)
>>> F4 = F1.cartesian_product(F2)
```

(continues on next page)

(continued from previous page)

```
>>> FF = F3.common_refinement(F4)
>>> F3.ngenerating_cones()
10
>>> F4.ngenerating_cones()
10
>>> FF.ngenerating_cones()
13
```

An example with two non-complete fans with the same support:

```
sage: F5 = Fan2d([(1,0), (1,2), (0,1)])
sage: F6 = Fan2d([(1,0), (2,1), (0,1)])
sage: F5.common_refinement(F6).ngenerating_cones()
3
```

```
>>> from sage.all import *
>>> F5 = Fan2d([(Integer(1),Integer(0)), (Integer(1),Integer(2)), (Integer(0),
↪Integer(1))])
>>> F6 = Fan2d([(Integer(1),Integer(0)), (Integer(2),Integer(1)), (Integer(0),
↪Integer(1))])
>>> F5.common_refinement(F6).ngenerating_cones()
3
```

Both fans must live in the same lattice:

```
sage: F0.common_refinement(F1)
Traceback (most recent call last):
...
ValueError: the fans are not in the same lattice
```

```
>>> from sage.all import *
>>> F0.common_refinement(F1)
Traceback (most recent call last):
...
ValueError: the fans are not in the same lattice
```

complex(*base_ring=Integer Ring*, *extended=False*)

Return the chain complex of the fan.

To a d-dimensional fan Σ, one can canonically associate a chain complex $K^\bullet$

$$0 \longrightarrow \mathbf{Z}^{\Sigma(d)} \longrightarrow \mathbf{Z}^{\Sigma(d-1)} \longrightarrow \cdots \longrightarrow \mathbf{Z}^{\Sigma(0)} \longrightarrow 0$$

where the leftmost nonzero entry is in degree 0 and the rightmost entry in degree d. See [Kly1990], eq. (3.2). This complex computes the homology of $|\Sigma| \subset N_{\mathbf{R}}$ with arbitrary support,

$$H_i(K) = H_{d-i}(|\Sigma|, \mathbf{Z})_{\text{non-cpct}}$$

For a complete fan, this is just the non-compactly supported homology of $\mathbf{R}^d$. In this case, $H_0(K) = \mathbf{Z}$ and 0 in all nonzero degrees.

For a complete fan, there is an extended chain complex

$$0 \longrightarrow \mathbf{Z} \longrightarrow \mathbf{Z}^{\Sigma(d)} \longrightarrow \mathbf{Z}^{\Sigma(d-1)} \longrightarrow \cdots \longrightarrow \mathbf{Z}^{\Sigma(0)} \longrightarrow 0$$

where we take the first **Z** term to be in degree -1. This complex is an exact sequence, that is, all homology groups vanish.

The orientation of each cone is chosen as in `oriented_boundary()`.

INPUT:

- `extended` – boolean (default: `False`); whether to construct the extended complex, that is, including the **Z**-term at degree -1 or not
- `base_ring` – a ring (default: `ZZ`); the ring to use instead of **Z**

OUTPUT:

The complex associated to the fan as a `ChainComplex`. This raises a `ValueError` if the extended complex is requested for a non-complete fan.

EXAMPLES:

```
sage: # needs palp
sage: fan = toric_varieties.P(3).fan()
sage: K_normal = fan.complex(); K_normal
Chain complex with at most 4 nonzero terms over Integer Ring
sage: K_normal.homology()
{0: Z, 1: 0, 2: 0, 3: 0}
sage: K_extended = fan.complex(extended=True); K_extended
Chain complex with at most 5 nonzero terms over Integer Ring
sage: K_extended.homology()
{-1: 0, 0: 0, 1: 0, 2: 0, 3: 0}
```

```
>>> from sage.all import *
>>> # needs palp
>>> fan = toric_varieties.P(Integer(3)).fan()
>>> K_normal = fan.complex(); K_normal
Chain complex with at most 4 nonzero terms over Integer Ring
>>> K_normal.homology()
{0: Z, 1: 0, 2: 0, 3: 0}
>>> K_extended = fan.complex(extended=True); K_extended
Chain complex with at most 5 nonzero terms over Integer Ring
>>> K_extended.homology()
{-1: 0, 0: 0, 1: 0, 2: 0, 3: 0}
```

Homology computations are much faster over **Q** if you do not care about the torsion coefficients:

```
sage: toric_varieties.P2_123().fan().complex(extended=True,                    #␣
↪needs palp
....:                                        base_ring=QQ)
Chain complex with at most 4 nonzero terms over Rational Field
sage: _.homology()                                                              #␣
↪needs palp
{-1: Vector space of dimension 0 over Rational Field,
 0: Vector space of dimension 0 over Rational Field,
 1: Vector space of dimension 0 over Rational Field,
 2: Vector space of dimension 0 over Rational Field}
```

```
>>> from sage.all import *
>>> toric_varieties.P2_123().fan().complex(extended=True,                     #␣
```

(continues on next page)

(continued from previous page)

```
↪needs palp
...                                             base_ring=QQ)
Chain complex with at most 4 nonzero terms over Rational Field
>>> _.homology()                                                            #␣
↪needs palp
{-1: Vector space of dimension 0 over Rational Field,
 0: Vector space of dimension 0 over Rational Field,
 1: Vector space of dimension 0 over Rational Field,
 2: Vector space of dimension 0 over Rational Field}
```

The extended complex is only defined for complete fans:

```
sage: fan = Fan([Cone([(1,0)])])
sage: fan.is_complete()
False
sage: fan.complex(extended=True)
Traceback (most recent call last):
...
ValueError: The extended complex is only defined for complete fans!
```

```
>>> from sage.all import *
>>> fan = Fan([Cone([(Integer(1),Integer(0))])])
>>> fan.is_complete()
False
>>> fan.complex(extended=True)
Traceback (most recent call last):
...
ValueError: The extended complex is only defined for complete fans!
```

The definition of the complex does not refer to the ambient space of the fan, so it does not distinguish a fan from the same fan embedded in a subspace:

```
sage: K1 = Fan([Cone([(-1,)]), Cone([(1,)])]).complex()
sage: K2 = Fan([Cone([(-1,0,0)]), Cone([(1,0,0)])]).complex()
sage: K1 == K2
True
```

```
>>> from sage.all import *
>>> K1 = Fan([Cone([(-Integer(1),)]), Cone([(Integer(1),)])]).complex()
>>> K2 = Fan([Cone([(-Integer(1),Integer(0),Integer(0))]), Cone([(Integer(1),
↪Integer(0),Integer(0))])]).complex()
>>> K1 == K2
True
```

Things get more complicated for non-complete fans:

```
sage: fan = Fan([Cone([(1,1,1)]),
....:            Cone([(1,0,0), (0,1,0)]),
....:            Cone([(-1,0,0), (0,-1,0), (0,0,-1)])])
sage: fan.complex().homology()
{0: 0, 1: 0, 2: Z x Z, 3: 0}
sage: fan = Fan([Cone([(1,0,0), (0,1,0)]),
```

(continues on next page)

(continued from previous page)

```
....:              Cone([(-1,0,0), (0,-1,0), (0,0,-1)])])
sage: fan.complex().homology()
{0: 0, 1: 0, 2: Z, 3: 0}
sage: fan = Fan([Cone([(-1,0,0), (0,-1,0), (0,0,-1)])])
sage: fan.complex().homology()
{0: 0, 1: 0, 2: 0, 3: 0}
```

```
>>> from sage.all import *
>>> fan = Fan([Cone([(Integer(1),Integer(1),Integer(1))]),
...             Cone([(Integer(1),Integer(0),Integer(0)), (Integer(0),
↪Integer(1),Integer(0))]),
...             Cone([(-Integer(1),Integer(0),Integer(0)), (Integer(0),-
↪Integer(1),Integer(0)), (Integer(0),Integer(0),-Integer(1))])])
>>> fan.complex().homology()
{0: 0, 1: 0, 2: Z x Z, 3: 0}
>>> fan = Fan([Cone([(Integer(1),Integer(0),Integer(0)), (Integer(0),
↪Integer(1),Integer(0))]),
...             Cone([(-Integer(1),Integer(0),Integer(0)), (Integer(0),-
↪Integer(1),Integer(0)), (Integer(0),Integer(0),-Integer(1))])])
>>> fan.complex().homology()
{0: 0, 1: 0, 2: Z, 3: 0}
>>> fan = Fan([Cone([(-Integer(1),Integer(0),Integer(0)), (Integer(0),-
↪Integer(1),Integer(0)), (Integer(0),Integer(0),-Integer(1))])])
>>> fan.complex().homology()
{0: 0, 1: 0, 2: 0, 3: 0}
```

cone_containing(**points*)

Return the smallest cone of `self` containing all given points.

INPUT:

- either one or more indices of rays of `self`, or one or more objects representing points of the ambient space of `self`, or a list of such objects (you CANNOT give a list of indices).

OUTPUT:

A *cone of fan* whose ambient fan is `self`

> **Note**
>
> We think of the origin as of the smallest cone containing no rays at all. If there is no ray in `self` that contains all `rays`, a `ValueError` exception will be raised.

EXAMPLES:

```
sage: cone1 = Cone([(0,-1), (1,0)])
sage: cone2 = Cone([(1,0), (0,1)])
sage: f = Fan([cone1, cone2])
sage: f.rays()
N(0, -1),
N(0,  1),
N(1,  0)
in 2-d lattice N
```

(continues on next page)

(continued from previous page)

```
sage: f.cone_containing(0)  # ray index
1-d cone of Rational polyhedral fan in 2-d lattice N
sage: f.cone_containing(0, 1) # ray indices
Traceback (most recent call last):
...
ValueError: there is no cone in
Rational polyhedral fan in 2-d lattice N
containing all of the given rays! Ray indices: [0, 1]
sage: f.cone_containing(0, 2) # ray indices
2-d cone of Rational polyhedral fan in 2-d lattice N
sage: f.cone_containing((0,1))  # point
1-d cone of Rational polyhedral fan in 2-d lattice N
sage: f.cone_containing([(0,1)]) # point
1-d cone of Rational polyhedral fan in 2-d lattice N
sage: f.cone_containing((1,1))
2-d cone of Rational polyhedral fan in 2-d lattice N
sage: f.cone_containing((1,1), (1,0))
2-d cone of Rational polyhedral fan in 2-d lattice N
sage: f.cone_containing()
0-d cone of Rational polyhedral fan in 2-d lattice N
sage: f.cone_containing((0,0))
0-d cone of Rational polyhedral fan in 2-d lattice N
sage: f.cone_containing((-1,1))
Traceback (most recent call last):
...
ValueError: there is no cone in
Rational polyhedral fan in 2-d lattice N
containing all of the given points! Points: [N(-1, 1)]
```

```
>>> from sage.all import *
>>> cone1 = Cone([(Integer(0),-Integer(1)), (Integer(1),Integer(0))])
>>> cone2 = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> f = Fan([cone1, cone2])
>>> f.rays()
N(0, -1),
N(0,  1),
N(1,  0)
in 2-d lattice N
>>> f.cone_containing(Integer(0))  # ray index
1-d cone of Rational polyhedral fan in 2-d lattice N
>>> f.cone_containing(Integer(0), Integer(1)) # ray indices
Traceback (most recent call last):
...
ValueError: there is no cone in
Rational polyhedral fan in 2-d lattice N
containing all of the given rays! Ray indices: [0, 1]
>>> f.cone_containing(Integer(0), Integer(2)) # ray indices
2-d cone of Rational polyhedral fan in 2-d lattice N
>>> f.cone_containing((Integer(0),Integer(1)))  # point
1-d cone of Rational polyhedral fan in 2-d lattice N
>>> f.cone_containing([(Integer(0),Integer(1))]) # point
1-d cone of Rational polyhedral fan in 2-d lattice N
```

(continues on next page)

(continued from previous page)

```
>>> f.cone_containing((Integer(1),Integer(1)))
2-d cone of Rational polyhedral fan in 2-d lattice N
>>> f.cone_containing((Integer(1),Integer(1)), (Integer(1),Integer(0)))
2-d cone of Rational polyhedral fan in 2-d lattice N
>>> f.cone_containing()
0-d cone of Rational polyhedral fan in 2-d lattice N
>>> f.cone_containing((Integer(0),Integer(0)))
0-d cone of Rational polyhedral fan in 2-d lattice N
>>> f.cone_containing((-Integer(1),Integer(1)))
Traceback (most recent call last):
...
ValueError: there is no cone in
Rational polyhedral fan in 2-d lattice N
containing all of the given points! Points: [N(-1, 1)]
```

cone_lattice()

Return the cone lattice of `self`.

This lattice will have the origin as the bottom (we do not include the empty set as a cone) and the fan itself as the top.

OUTPUT:

`finite poset <sage.combinat.posets.posets.FinitePoset` of *cones of fan*, behaving like "regular" cones, but also containing the information about their relation to this fan, namely, the contained rays and containing generating cones. The top of the lattice will be this fan itself (*which is not a* *cone of fan*).

See also *cones()*.

EXAMPLES:

Cone lattices can be computed for arbitrary fans:

```
sage: cone1 = Cone([(1,0), (0,1)])
sage: cone2 = Cone([(-1,0)])
sage: fan = Fan([cone1, cone2])
sage: fan.rays()
N(-1, 0),
N( 0, 1),
N( 1, 0)
in 2-d lattice N
sage: for cone in fan: print(cone.ambient_ray_indices())
(1, 2)
(0,)
sage: L = fan.cone_lattice()
sage: L
Finite poset containing 6 elements with distinguished linear extension
```

```
>>> from sage.all import *
>>> cone1 = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> cone2 = Cone([(-Integer(1),Integer(0))])
>>> fan = Fan([cone1, cone2])
>>> fan.rays()
N(-1, 0),
N( 0, 1),
```

(continues on next page)

(continued from previous page)

```
N( 1, 0)
in 2-d lattice N
>>> for cone in fan: print(cone.ambient_ray_indices())
(1, 2)
(0,)
>>> L = fan.cone_lattice()
>>> L
Finite poset containing 6 elements with distinguished linear extension
```

These 6 elements are the origin, three rays, one two-dimensional cone, and the fan itself. Since we do add the fan itself as the largest face, you should be a little bit careful with this last element:

```
sage: for face in L: print(face.ambient_ray_indices())
Traceback (most recent call last):
...
AttributeError: 'RationalPolyhedralFan'
object has no attribute 'ambient_ray_indices'
sage: L.top()
Rational polyhedral fan in 2-d lattice N
```

```
>>> from sage.all import *
>>> for face in L: print(face.ambient_ray_indices())
Traceback (most recent call last):
...
AttributeError: 'RationalPolyhedralFan'
object has no attribute 'ambient_ray_indices'
>>> L.top()
Rational polyhedral fan in 2-d lattice N
```

For example, you can do

```
sage: for l in L.level_sets()[:-1]:
....:     print([f.ambient_ray_indices() for f in l])
[()]
[(0,), (1,), (2,)]
[(1, 2)]
```

```
>>> from sage.all import *
>>> for l in L.level_sets()[:-Integer(1)]:
...     print([f.ambient_ray_indices() for f in l])
[()]
[(0,), (1,), (2,)]
[(1, 2)]
```

If the fan is complete, its cone lattice is atomic and coatomic and can (and will!) be computed in a much more efficient way, but the interface is exactly the same:

```
sage: fan = toric_varieties.P1xP1().fan()                                   #␣
↪needs palp
sage: L = fan.cone_lattice()                                                #␣
↪needs palp
sage: for l in L.level_sets()[:-1]:                                         #␣
```

(continues on next page)

(continued from previous page)

```
↪needs palp
....:     print([f.ambient_ray_indices() for f in l])
[()]
[(0,), (1,), (2,), (3,)]
[(0, 2), (1, 2), (1, 3), (0, 3)]
```

```
>>> from sage.all import *
>>> fan = toric_varieties.P1xP1().fan()                                    #␣
↪needs palp
>>> L = fan.cone_lattice()                                                 #␣
↪needs palp
>>> for l in L.level_sets()[:-Integer(1)]:                                  ␣
↪     # needs palp
...     print([f.ambient_ray_indices() for f in l])
[()]
[(0,), (1,), (2,), (3,)]
[(0, 2), (1, 2), (1, 3), (0, 3)]
```

Let's also consider the cone lattice of a fan generated by a single cone:

```
sage: fan = Fan([cone1])
sage: L = fan.cone_lattice()
sage: L
Finite poset containing 5 elements with distinguished linear extension
```

```
>>> from sage.all import *
>>> fan = Fan([cone1])
>>> L = fan.cone_lattice()
>>> L
Finite poset containing 5 elements with distinguished linear extension
```

Here these 5 elements correspond to the origin, two rays, one generating cone of dimension two, and the whole fan. While this single cone "is" the whole fan, it is consistent and convenient to distinguish them in the cone lattice.

cones(*dim=None*, *codim=None*)

Return the specified cones of `self`.

INPUT:

- `dim` – dimension of the requested cones
- `codim` – codimension of the requested cones

> **Note**
>
> You can specify at most one input parameter.

OUTPUT:

`tuple` of cones of `self` of the specified (co)dimension, if either `dim` or `codim` is given. Otherwise `tuple` of such tuples for all existing dimensions.

EXAMPLES:

```
sage: cone1 = Cone([(1,0), (0,1)])
sage: cone2 = Cone([(-1,0)])
sage: fan = Fan([cone1, cone2])
sage: fan(dim=0)
(0-d cone of Rational polyhedral fan in 2-d lattice N,)
sage: fan(codim=2)
(0-d cone of Rational polyhedral fan in 2-d lattice N,)
sage: for cone in fan.cones(1): cone.ray(0)
N(-1, 0)
N(0, 1)
N(1, 0)
sage: fan.cones(2)
(2-d cone of Rational polyhedral fan in 2-d lattice N,)
```

```
>>> from sage.all import *
>>> cone1 = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> cone2 = Cone([(-Integer(1),Integer(0))])
>>> fan = Fan([cone1, cone2])
>>> fan(dim=Integer(0))
(0-d cone of Rational polyhedral fan in 2-d lattice N,)
>>> fan(codim=Integer(2))
(0-d cone of Rational polyhedral fan in 2-d lattice N,)
>>> for cone in fan.cones(Integer(1)): cone.ray(Integer(0))
N(-1, 0)
N(0, 1)
N(1, 0)
>>> fan.cones(Integer(2))
(2-d cone of Rational polyhedral fan in 2-d lattice N,)
```

You cannot specify both dimension and codimension, even if they "agree":

```
sage: fan(dim=1, codim=1)
Traceback (most recent call last):
...
ValueError: dimension and codimension
cannot be specified together!
```

```
>>> from sage.all import *
>>> fan(dim=Integer(1), codim=Integer(1))
Traceback (most recent call last):
...
ValueError: dimension and codimension
cannot be specified together!
```

But it is OK to ask for cones of too high or low (co)dimension:

```
sage: fan(-1)
()
sage: fan(3)
()
sage: fan(codim=4)
()
```

```
>>> from sage.all import *
>>> fan(-Integer(1))
()
>>> fan(Integer(3))
()
>>> fan(codim=Integer(4))
()
```

contains(*cone*)

Check if a given `cone` is equivalent to a cone of the fan.

INPUT:

- `cone` – anything

OUTPUT:

`False` if `cone` is not a cone or if `cone` is not equivalent to a cone of the fan, `True` otherwise

> **Note**
>
> Recall that a fan is a (finite) collection of cones. A cone is contained in a fan if it is equivalent to one of the cones of the fan. In particular, it is possible that all rays of the cone are in the fan, but the cone itself is not.
>
> If you want to know whether a point is in the support of the fan, you should use *support_contains()*.

EXAMPLES:

We first construct a simple fan:

```
sage: cone1 = Cone([(0,-1), (1,0)])
sage: cone2 = Cone([(1,0), (0,1)])
sage: f = Fan([cone1, cone2])
```

```
>>> from sage.all import *
>>> cone1 = Cone([(Integer(0),-Integer(1)), (Integer(1),Integer(0))])
>>> cone2 = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> f = Fan([cone1, cone2])
```

Now we check if some cones are in this fan. First, we make sure that the order of rays of the input cone does not matter (`check=False` option ensures that rays of these cones will be listed exactly as they are given):

```
sage: f.contains(Cone([(1,0), (0,1)], check=False))
True
sage: f.contains(Cone([(0,1), (1,0)], check=False))
True
```

```
>>> from sage.all import *
>>> f.contains(Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))],␣
↪check=False))
True
>>> f.contains(Cone([(Integer(0),Integer(1)), (Integer(1),Integer(0))],␣
↪check=False))
True
```

Now we check that a non-generating cone is in our fan:

```
sage: f.contains(Cone([(1,0)]))
True
sage: Cone([(1,0)]) in f   # equivalent to the previous command
True
```

```
>>> from sage.all import *
>>> f.contains(Cone([(Integer(1),Integer(0))]))
True
>>> Cone([(Integer(1),Integer(0))]) in f   # equivalent to the previous␣
↪command
True
```

Finally, we test some cones which are not in this fan:

```
sage: f.contains(Cone([(1,1)]))
False
sage: f.contains(Cone([(1,0), (-0,1)]))
True
```

```
>>> from sage.all import *
>>> f.contains(Cone([(Integer(1),Integer(1))]))
False
>>> f.contains(Cone([(Integer(1),Integer(0)), (-Integer(0),Integer(1))]))
True
```

A point is not a cone:

```
sage: n = f.lattice()(1,1); n
N(1, 1)
sage: f.contains(n)
False
```

```
>>> from sage.all import *
>>> n = f.lattice()(Integer(1),Integer(1)); n
N(1, 1)
>>> f.contains(n)
False
```

embed(*cone*)

Return the cone equivalent to the given one, but sitting in `self`.

You may need to use this method before calling methods of `cone` that depend on the ambient structure, such as *ambient_ray_indices()* or *facet_of()*. The cone returned by this method will have `self` as ambient. If `cone` does not represent a valid cone of `self`, `ValueError` exception is raised.

> **Note**
>
> This method is very quick if `self` is already the ambient structure of `cone`, so you can use without extra checks and performance hit even if `cone` is likely to sit in `self` but in principle may not.

INPUT:

- cone – a *cone*

OUTPUT:

a *cone of fan*, equivalent to cone but sitting inside self

EXAMPLES:

Let's take a 3-d fan generated by a cone on 4 rays:

```
sage: f = Fan([Cone([(1,0,1), (0,1,1), (-1,0,1), (0,-1,1)])])
```

```
>>> from sage.all import *
>>> f = Fan([Cone([(Integer(1),Integer(0),Integer(1)), (Integer(0),Integer(1),
↪Integer(1)), (-Integer(1),Integer(0),Integer(1)), (Integer(0),-Integer(1),
↪Integer(1))])])
```

Then any ray generates a 1-d cone of this fan, but if you construct such a cone directly, it will not "sit" inside the fan:

```
sage: ray = Cone([(0,-1,1)])
sage: ray
1-d cone in 3-d lattice N
sage: ray.ambient_ray_indices()
(0,)
sage: ray.adjacent()
()
sage: ray.ambient()
1-d cone in 3-d lattice N
```

```
>>> from sage.all import *
>>> ray = Cone([(Integer(0),-Integer(1),Integer(1))])
>>> ray
1-d cone in 3-d lattice N
>>> ray.ambient_ray_indices()
(0,)
>>> ray.adjacent()
()
>>> ray.ambient()
1-d cone in 3-d lattice N
```

If we want to operate with this ray as a part of the fan, we need to embed it first:

```
sage: e_ray = f.embed(ray)
sage: e_ray
1-d cone of Rational polyhedral fan in 3-d lattice N
sage: e_ray.rays()
N(0, -1, 1)
in 3-d lattice N
sage: e_ray is ray
False
sage: e_ray.is_equivalent(ray)
True
sage: e_ray.ambient_ray_indices()
(3,)
```

(continues on next page)

(continued from previous page)

```
sage: e_ray.adjacent()
(1-d cone of Rational polyhedral fan in 3-d lattice N,
 1-d cone of Rational polyhedral fan in 3-d lattice N)
sage: e_ray.ambient()
Rational polyhedral fan in 3-d lattice N
```

```
>>> from sage.all import *
>>> e_ray = f.embed(ray)
>>> e_ray
1-d cone of Rational polyhedral fan in 3-d lattice N
>>> e_ray.rays()
N(0, -1, 1)
in 3-d lattice N
>>> e_ray is ray
False
>>> e_ray.is_equivalent(ray)
True
>>> e_ray.ambient_ray_indices()
(3,)
>>> e_ray.adjacent()
(1-d cone of Rational polyhedral fan in 3-d lattice N,
 1-d cone of Rational polyhedral fan in 3-d lattice N)
>>> e_ray.ambient()
Rational polyhedral fan in 3-d lattice N
```

Not every cone can be embedded into a fixed fan:

```
sage: f.embed(Cone([(0,0,1)]))
Traceback (most recent call last):
...
ValueError: 1-d cone in 3-d lattice N does not belong
to Rational polyhedral fan in 3-d lattice N!
sage: f.embed(Cone([(1,0,1), (-1,0,1)]))
Traceback (most recent call last):
...
ValueError: 2-d cone in 3-d lattice N does not belong
to Rational polyhedral fan in 3-d lattice N!
```

```
>>> from sage.all import *
>>> f.embed(Cone([(Integer(0),Integer(0),Integer(1))]))
Traceback (most recent call last):
...
ValueError: 1-d cone in 3-d lattice N does not belong
to Rational polyhedral fan in 3-d lattice N!
>>> f.embed(Cone([(Integer(1),Integer(0),Integer(1)), (-Integer(1),Integer(0),
↪Integer(1))]))
Traceback (most recent call last):
...
ValueError: 2-d cone in 3-d lattice N does not belong
to Rational polyhedral fan in 3-d lattice N!
```

f_vector()

Return the f-vector of the fan.

This is the tuple $(f_0, f_1, \ldots, f_d)$ where f_i is the number of cones of dimension i.

EXAMPLES:

```
sage: F = ClusterAlgebra(['A',2]).cluster_fan()
sage: F.f_vector()
(1, 5, 5)
```

```
>>> from sage.all import *
>>> F = ClusterAlgebra(['A',Integer(2)]).cluster_fan()
>>> F.f_vector()
(1, 5, 5)
```

generating_cone(*n*)

Return the n-th generating cone of `self`.

INPUT:

- `n` – integer; the index of a generating cone

OUTPUT: *cone of fan*

EXAMPLES:

```
sage: fan = toric_varieties.P1xP1().fan()                                    #␣
↪needs palp
sage: fan.generating_cone(0)                                                 #␣
↪needs palp
2-d cone of Rational polyhedral fan in 2-d lattice N
```

```
>>> from sage.all import *
>>> fan = toric_varieties.P1xP1().fan()                                    #␣
↪needs palp
>>> fan.generating_cone(Integer(0))                                          ␣
↪     # needs palp
2-d cone of Rational polyhedral fan in 2-d lattice N
```

generating_cones()

Return generating cones of `self`.

OUTPUT: `tuple` of *cones of fan*

EXAMPLES:

```
sage: fan = toric_varieties.P1xP1().fan()                                    #␣
↪needs palp
sage: fan.generating_cones()                                                 #␣
↪needs palp
(2-d cone of Rational polyhedral fan in 2-d lattice N,
 2-d cone of Rational polyhedral fan in 2-d lattice N,
 2-d cone of Rational polyhedral fan in 2-d lattice N,
 2-d cone of Rational polyhedral fan in 2-d lattice N)
sage: cone1 = Cone([(1,0), (0,1)])
sage: cone2 = Cone([(-1,0)])
sage: fan = Fan([cone1, cone2])
```

(continues on next page)

(continued from previous page)

```
sage: fan.generating_cones()
(2-d cone of Rational polyhedral fan in 2-d lattice N,
 1-d cone of Rational polyhedral fan in 2-d lattice N)
```

```
>>> from sage.all import *
>>> fan = toric_varieties.P1xP1().fan()                                      #␣
↪needs palp
>>> fan.generating_cones()                                                   #␣
↪needs palp
(2-d cone of Rational polyhedral fan in 2-d lattice N,
 2-d cone of Rational polyhedral fan in 2-d lattice N,
 2-d cone of Rational polyhedral fan in 2-d lattice N,
 2-d cone of Rational polyhedral fan in 2-d lattice N)
>>> cone1 = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> cone2 = Cone([(-Integer(1),Integer(0))])
>>> fan = Fan([cone1, cone2])
>>> fan.generating_cones()
(2-d cone of Rational polyhedral fan in 2-d lattice N,
 1-d cone of Rational polyhedral fan in 2-d lattice N)
```

is_complete()

Check if `self` is complete.

A rational polyhedral fan is *complete* if its cones fill the whole space.

OUTPUT: `True` if `self` is complete and `False` otherwise

EXAMPLES:

```
sage: fan = toric_varieties.P1xP1().fan()                                      #␣
↪needs palp
sage: fan.is_complete()                                                        #␣
↪needs palp
True
sage: cone1 = Cone([(1,0), (0,1)])
sage: cone2 = Cone([(-1,0)])
sage: fan = Fan([cone1, cone2])
sage: fan.is_complete()
False
```

```
>>> from sage.all import *
>>> fan = toric_varieties.P1xP1().fan()                                      #␣
↪needs palp
>>> fan.is_complete()                                                        #␣
↪needs palp
True
>>> cone1 = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> cone2 = Cone([(-Integer(1),Integer(0))])
>>> fan = Fan([cone1, cone2])
>>> fan.is_complete()
False
```

is_equivalent(*other*)

Check if `self` is "mathematically" the same as `other`.

INPUT:

- `other` – fan

OUTPUT:

`True` if `self` and `other` define the same fans as collections of equivalent cones in the same lattice, `False` otherwise.

There are three different equivalences between fans F_1 and F_2 in the same lattice:

1. They have the same rays in the same order and the same generating cones in the same order. This is tested by `F1 == F2`.
2. They have the same rays and the same generating cones without taking into account any order. This is tested by `F1.is_equivalent(F2)`.
3. They are in the same orbit of $GL(n,\mathbf{Z})$ (and, therefore, correspond to isomorphic toric varieties). This is tested by `F1.is_isomorphic(F2)`.

Note that `virtual_rays()` are included into consideration for all of the above equivalences.

EXAMPLES:

```
sage: fan1 = Fan(cones=[(0,1), (1,2)],
....:            rays=[(1,0), (0,1), (-1,-1)],
....:            check=False)
sage: fan2 = Fan(cones=[(2,1), (0,2)],
....:            rays=[(1,0), (-1,-1), (0,1)],
....:            check=False)
sage: fan3 = Fan(cones=[(0,1), (1,2)],
....:            rays=[(1,0), (0,1), (-1,1)],
....:            check=False)
sage: fan1 == fan2
False
sage: fan1.is_equivalent(fan2)
True
sage: fan1 == fan3
False
sage: fan1.is_equivalent(fan3)
False
```

```
>>> from sage.all import *
>>> fan1 = Fan(cones=[(Integer(0),Integer(1)), (Integer(1),Integer(2))],
...            rays=[(Integer(1),Integer(0)), (Integer(0),Integer(1)), (-
↪Integer(1),-Integer(1))],
...            check=False)
>>> fan2 = Fan(cones=[(Integer(2),Integer(1)), (Integer(0),Integer(2))],
...            rays=[(Integer(1),Integer(0)), (-Integer(1),-Integer(1)),
↪(Integer(0),Integer(1))],
...            check=False)
>>> fan3 = Fan(cones=[(Integer(0),Integer(1)), (Integer(1),Integer(2))],
...            rays=[(Integer(1),Integer(0)), (Integer(0),Integer(1)), (-
↪Integer(1),Integer(1))],
...            check=False)
>>> fan1 == fan2
False
>>> fan1.is_equivalent(fan2)
```

(continues on next page)

(continued from previous page)

```
True
>>> fan1 == fan3
False
>>> fan1.is_equivalent(fan3)
False
```

is_isomorphic(*other*)

Check if `self` is in the same $GL(n,\mathbf{Z})$-orbit as `other`.

There are three different equivalences between fans F_1 and F_2 in the same lattice:

1. They have the same rays in the same order and the same generating cones in the same order. This is tested by `F1 == F2`.
2. They have the same rays and the same generating cones without taking into account any order. This is tested by `F1.is_equivalent(F2)`.
3. They are in the same orbit of $GL(n,\mathbf{Z})$ (and, therefore, correspond to isomorphic toric varieties). This is tested by `F1.is_isomorphic(F2)`.

Note that `virtual_rays()` are included into consideration for all of the above equivalences.

INPUT:

- `other` – a *fan*

OUTPUT:

`True` if `self` and `other` are in the same $GL(n,\mathbf{Z})$-orbit, `False` otherwise

See also

If you want to obtain the actual fan isomorphism, use `isomorphism()`.

EXAMPLES:

Here we pick an $SL(2,\mathbf{Z})$ matrix `m` and then verify that the image fan is isomorphic:

```
sage: rays = ((1, 1), (0, 1), (-1, -1), (1, 0))
sage: cones = [(0,1), (1,2), (2,3), (3,0)]
sage: fan1 = Fan(cones, rays)
sage: m = matrix([[-2,3], [1,-1]])
sage: fan2 = Fan(cones, [vector(r)*m for r in rays])
sage: fan1.is_isomorphic(fan2)
True
sage: fan1.is_equivalent(fan2)
False
sage: fan1 == fan2
False
```

```
>>> from sage.all import *
>>> rays = ((Integer(1), Integer(1)), (Integer(0), Integer(1)), (-Integer(1),␣
↪-Integer(1)), (Integer(1), Integer(0)))
>>> cones = [(Integer(0),Integer(1)), (Integer(1),Integer(2)), (Integer(2),
↪Integer(3)), (Integer(3),Integer(0))]
>>> fan1 = Fan(cones, rays)
```

(continues on next page)

(continued from previous page)

```
>>> m = matrix([[-Integer(2),Integer(3)], [Integer(1),-Integer(1)]])
>>> fan2 = Fan(cones, [vector(r)*m for r in rays])
>>> fan1.is_isomorphic(fan2)
True
>>> fan1.is_equivalent(fan2)
False
>>> fan1 == fan2
False
```

These fans are “mirrors” of each other:

```
sage: fan1 = Fan(cones=[(0,1), (1,2)],
....:            rays=[(1,0), (0,1), (-1,-1)],
....:            check=False)
sage: fan2 = Fan(cones=[(0,1), (1,2)],
....:            rays=[(1,0), (0,-1), (-1,1)],
....:            check=False)
sage: fan1 == fan2
False
sage: fan1.is_equivalent(fan2)
False
sage: fan1.is_isomorphic(fan2)
True
sage: fan1.is_isomorphic(fan1)
True
```

```
>>> from sage.all import *
>>> fan1 = Fan(cones=[(Integer(0),Integer(1)), (Integer(1),Integer(2))],
...            rays=[(Integer(1),Integer(0)), (Integer(0),Integer(1)), (-
↪Integer(1),-Integer(1))],
...            check=False)
>>> fan2 = Fan(cones=[(Integer(0),Integer(1)), (Integer(1),Integer(2))],
...            rays=[(Integer(1),Integer(0)), (Integer(0),-Integer(1)), (-
↪Integer(1),Integer(1))],
...            check=False)
>>> fan1 == fan2
False
>>> fan1.is_equivalent(fan2)
False
>>> fan1.is_isomorphic(fan2)
True
>>> fan1.is_isomorphic(fan1)
True
```

is_polytopal()

Check if `self` is the normal fan of a polytope.

A rational polyhedral fan is *polytopal* if it is the normal fan of a polytope. This is also called *regular*, or provides a *coherent* subdivision or leads to a *projective* toric variety.

OUTPUT: `True` if `self` is polytopal and `False` otherwise

EXAMPLES:

This is the mother of all examples (see Section 7.1.1 in [DLRS2010]):

```
sage: def mother(epsilon=0):
....:     rays = [(4-epsilon,epsilon,0),(0,4-epsilon,epsilon),(epsilon,0,4-
↪epsilon),(2,1,1),(1,2,1),(1,1,2),(-1,-1,-1)]
....:     L = [(0,1,4),(0,3,4),(1,2,5),(1,4,5),(0,2,3),(2,3,5),(3,4,5),(6,0,
↪1),(6,1,2),(6,2,0)]
....:     S1 = [Cone([rays[i] for i in indices]) for indices in L]
....:     return Fan(S1)
```

```
>>> from sage.all import *
>>> def mother(epsilon=Integer(0)):
...     rays = [(Integer(4)-epsilon,epsilon,Integer(0)),(Integer(0),
↪Integer(4)-epsilon,epsilon),(epsilon,Integer(0),Integer(4)-epsilon),
↪(Integer(2),Integer(1),Integer(1)),(Integer(1),Integer(2),Integer(1)),
↪(Integer(1),Integer(1),Integer(2)),(-Integer(1),-Integer(1),-Integer(1))]
...     L = [(Integer(0),Integer(1),Integer(4)),(Integer(0),Integer(3),
↪Integer(4)),(Integer(1),Integer(2),Integer(5)),(Integer(1),Integer(4),
↪Integer(5)),(Integer(0),Integer(2),Integer(3)),(Integer(2),Integer(3),
↪Integer(5)),(Integer(3),Integer(4),Integer(5)),(Integer(6),Integer(0),
↪Integer(1)),(Integer(6),Integer(1),Integer(2)),(Integer(6),Integer(2),
↪Integer(0))]
...     S1 = [Cone([rays[i] for i in indices]) for indices in L]
...     return Fan(S1)
```

When epsilon=0, it is not polytopal:

```
sage: epsilon = 0
sage: mother(epsilon).is_polytopal()
False
```

```
>>> from sage.all import *
>>> epsilon = Integer(0)
>>> mother(epsilon).is_polytopal()
False
```

Doing a slight perturbation makes the same subdivision polytopal:

```
sage: epsilon = 1/2
sage: mother(epsilon).is_polytopal()
True
```

```
>>> from sage.all import *
>>> epsilon = Integer(1)/Integer(2)
>>> mother(epsilon).is_polytopal()
True
```

See also

`is_projective()`.

is_simplicial()

Check if `self` is simplicial.

A rational polyhedral fan is **simplicial** if all of its cones are, i.e. primitive vectors along generating rays of every cone form a part of a *rational* basis of the ambient space.

OUTPUT: `True` if `self` is simplicial and `False` otherwise

EXAMPLES:

```
sage: fan = toric_varieties.P1xP1().fan()                                    #␣
↪needs palp
sage: fan.is_simplicial()                                                    #␣
↪needs palp
True
sage: cone1 = Cone([(1,0), (0,1)])
sage: cone2 = Cone([(-1,0)])
sage: fan = Fan([cone1, cone2])
sage: fan.is_simplicial()
True
```

```
>>> from sage.all import *
>>> fan = toric_varieties.P1xP1().fan()                                      #␣
↪needs palp
>>> fan.is_simplicial()                                                      #␣
↪needs palp
True
>>> cone1 = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> cone2 = Cone([(-Integer(1),Integer(0))])
>>> fan = Fan([cone1, cone2])
>>> fan.is_simplicial()
True
```

In fact, any fan in a two-dimensional ambient space is simplicial. This is no longer the case in dimension three:

```
sage: fan = NormalFan(lattice_polytope.cross_polytope(3))
sage: fan.is_simplicial()
False
sage: fan.generating_cone(0).nrays()
4
```

```
>>> from sage.all import *
>>> fan = NormalFan(lattice_polytope.cross_polytope(Integer(3)))
>>> fan.is_simplicial()
False
>>> fan.generating_cone(Integer(0)).nrays()
4
```

is_smooth(*codim=None*)

Check if `self` is smooth.

A rational polyhedral fan is **smooth** if all of its cones are, i.e. primitive vectors along generating rays of every cone form a part of an *integral* basis of the ambient space. In this case the corresponding toric variety is smooth.

A fan in an n-dimensional lattice is smooth up to codimension c if all cones of codimension greater than or equal to c are smooth, i.e. if all cones of dimension less than or equal to $n - c$ are smooth. In this case the singular set of the corresponding toric variety is of dimension less than c.

INPUT:

- `codim` – codimension in which smoothness has to be checked, by default complete smoothness will be checked

OUTPUT:

`True` if `self` is smooth (in codimension `codim`, if it was given) and `False` otherwise.

EXAMPLES:

```
sage: fan = toric_varieties.P1xP1().fan()                                    #␣
↪needs palp
sage: fan.is_smooth()                                                        #␣
↪needs palp
True
sage: cone1 = Cone([(1,0), (0,1)])
sage: cone2 = Cone([(-1,0)])
sage: fan = Fan([cone1, cone2])
sage: fan.is_smooth()
True
sage: fan = NormalFan(lattice_polytope.cross_polytope(2))
sage: fan.is_smooth()
False
sage: fan.is_smooth(codim=1)
True
sage: fan.generating_cone(0).rays()
N(-1, -1),
N(-1,  1)
in 2-d lattice N
sage: fan.generating_cone(0).rays().matrix().det()
-2
```

```
>>> from sage.all import *
>>> fan = toric_varieties.P1xP1().fan()                                     #␣
↪needs palp
>>> fan.is_smooth()                                                         #␣
↪needs palp
True
>>> cone1 = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> cone2 = Cone([(-Integer(1),Integer(0))])
>>> fan = Fan([cone1, cone2])
>>> fan.is_smooth()
True
>>> fan = NormalFan(lattice_polytope.cross_polytope(Integer(2)))
>>> fan.is_smooth()
False
>>> fan.is_smooth(codim=Integer(1))
True
>>> fan.generating_cone(Integer(0)).rays()
N(-1, -1),
N(-1,  1)
in 2-d lattice N
>>> fan.generating_cone(Integer(0)).rays().matrix().det()
-2
```

isomorphism(*other*)

Return a fan isomorphism from `self` to `other`.

INPUT:

- `other` – fan

OUTPUT:

A fan isomorphism. If no such isomorphism exists, a `FanNotIsomorphicError` is raised.

EXAMPLES:

```
sage: rays = ((1, 1), (0, 1), (-1, -1), (3, 1))
sage: cones = [(0,1), (1,2), (2,3), (3,0)]
sage: fan1 = Fan(cones, rays)
sage: m = matrix([[-2,3], [1,-1]])
sage: fan2 = Fan(cones, [vector(r)*m for r in rays])

sage: fan1.isomorphism(fan2)
Fan morphism defined by the matrix
[-2  3]
[ 1 -1]
Domain fan: Rational polyhedral fan in 2-d lattice N
Codomain fan: Rational polyhedral fan in 2-d lattice N

sage: fan2.isomorphism(fan1)
Fan morphism defined by the matrix
[1 3]
[1 2]
Domain fan: Rational polyhedral fan in 2-d lattice N
Codomain fan: Rational polyhedral fan in 2-d lattice N

sage: fan1.isomorphism(toric_varieties.P2().fan())                    #␣
↪needs palp
Traceback (most recent call last):
...
FanNotIsomorphicError
```

```
>>> from sage.all import *
>>> rays = ((Integer(1), Integer(1)), (Integer(0), Integer(1)), (-Integer(1),␣
↪-Integer(1)), (Integer(3), Integer(1)))
>>> cones = [(Integer(0),Integer(1)), (Integer(1),Integer(2)), (Integer(2),
↪Integer(3)), (Integer(3),Integer(0))]
>>> fan1 = Fan(cones, rays)
>>> m = matrix([[-Integer(2),Integer(3)], [Integer(1),-Integer(1)]])
>>> fan2 = Fan(cones, [vector(r)*m for r in rays])

>>> fan1.isomorphism(fan2)
Fan morphism defined by the matrix
[-2  3]
[ 1 -1]
Domain fan: Rational polyhedral fan in 2-d lattice N
Codomain fan: Rational polyhedral fan in 2-d lattice N
```

(continues on next page)

(continued from previous page)

```
>>> fan2.isomorphism(fan1)
Fan morphism defined by the matrix
[1 3]
[1 2]
Domain fan: Rational polyhedral fan in 2-d lattice N
Codomain fan: Rational polyhedral fan in 2-d lattice N

>>> fan1.isomorphism(toric_varieties.P2().fan())                          #␣
↪needs palp
Traceback (most recent call last):
...
FanNotIsomorphicError
```

linear_equivalence_ideal(*ring*)

Return the ideal generated by linear relations.

INPUT:

- A polynomial ring in `self.nrays()` variables.

OUTPUT:

Return the ideal, in the given `ring`, generated by the linear relations of the rays. In toric geometry, this corresponds to rational equivalence of divisors.

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: fan.linear_equivalence_ideal(PolynomialRing(QQ, 5, 'A, B, C, D, E'))
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
```

```
>>> from sage.all import *
>>> fan = Fan([[Integer(0),Integer(1),Integer(3)],[Integer(3),Integer(4)],
↪[Integer(2),Integer(0)],[Integer(1),Integer(2),Integer(4)]],
...           [(-Integer(3), -Integer(2), Integer(1)), (Integer(0),␣
↪Integer(0), Integer(1)), (Integer(3), -Integer(2), Integer(1)), (-
↪Integer(1), -Integer(1), Integer(1)), (Integer(1), -Integer(1),␣
↪Integer(1))])
>>> fan.linear_equivalence_ideal(PolynomialRing(QQ, Integer(5), 'A, B, C, D, E
↪'))
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
```

make_simplicial(***kwds*)

Construct a simplicial fan subdividing `self`.

It is a synonym for *subdivide()* with `make_simplicial=True` option.

INPUT:

- this functions accepts only keyword arguments. See *subdivide()* for documentation.

OUTPUT:

rational polyhedral fan

EXAMPLES:

```
sage: fan = NormalFan(lattice_polytope.cross_polytope(3))
sage: fan.is_simplicial()
False
sage: fan.ngenerating_cones()
6
sage: new_fan = fan.make_simplicial()
sage: new_fan.is_simplicial()
True
sage: new_fan.ngenerating_cones()
12
```

```
>>> from sage.all import *
>>> fan = NormalFan(lattice_polytope.cross_polytope(Integer(3)))
>>> fan.is_simplicial()
False
>>> fan.ngenerating_cones()
6
>>> new_fan = fan.make_simplicial()
>>> new_fan.is_simplicial()
True
>>> new_fan.ngenerating_cones()
12
```

ngenerating_cones()

Return the number of generating cones of `self`.

OUTPUT: integer

EXAMPLES:

```
sage: fan = toric_varieties.P1xP1().fan()                                   #␣
↪needs palp
sage: fan.ngenerating_cones()                                               #␣
↪needs palp
4
sage: cone1 = Cone([(1,0), (0,1)])
sage: cone2 = Cone([(-1,0)])
sage: fan = Fan([cone1, cone2])
sage: fan.ngenerating_cones()
2
```

```
>>> from sage.all import *
>>> fan = toric_varieties.P1xP1().fan()                                     #␣
↪needs palp
>>> fan.ngenerating_cones()                                                 #␣
↪needs palp
4
>>> cone1 = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> cone2 = Cone([(-Integer(1),Integer(0))])
>>> fan = Fan([cone1, cone2])
>>> fan.ngenerating_cones()
2
```

oriented_boundary(*cone*)

Return the facets bounding `cone` with their induced orientation.

INPUT:

- `cone` – a cone of the fan or the whole fan

OUTPUT:

The boundary cones of `cone` as a formal linear combination of cones with coefficients ± 1. Each summand is a facet of `cone` and the coefficient indicates whether their (chosen) orientation agrees or disagrees with the "outward normal first" boundary orientation. Note that the orientation of any individual cone is arbitrary. This method once and for all picks orientations for all cones and then computes the boundaries relative to that chosen orientation.

If `cone` is the fan itself, the generating cones with their orientation relative to the ambient space are returned.

See `complex()` for the associated chain complex. If you do not require the orientation, use `cone.facets()` instead.

EXAMPLES:

```
sage: # needs palp
sage: fan = toric_varieties.P(3).fan()
sage: cone = fan(2)[0]
sage: bdry = fan.oriented_boundary(cone);  bdry
-1-d cone of Rational polyhedral fan in 3-d lattice N
+ 1-d cone of Rational polyhedral fan in 3-d lattice N
sage: bdry[0]
(-1, 1-d cone of Rational polyhedral fan in 3-d lattice N)
sage: bdry[1]
(1, 1-d cone of Rational polyhedral fan in 3-d lattice N)
sage: fan.oriented_boundary(bdry[0][1])
-0-d cone of Rational polyhedral fan in 3-d lattice N
sage: fan.oriented_boundary(bdry[1][1])
-0-d cone of Rational polyhedral fan in 3-d lattice N
```

```
>>> from sage.all import *
>>> # needs palp
>>> fan = toric_varieties.P(Integer(3)).fan()
>>> cone = fan(Integer(2))[Integer(0)]
>>> bdry = fan.oriented_boundary(cone);  bdry
-1-d cone of Rational polyhedral fan in 3-d lattice N
+ 1-d cone of Rational polyhedral fan in 3-d lattice N
>>> bdry[Integer(0)]
(-1, 1-d cone of Rational polyhedral fan in 3-d lattice N)
>>> bdry[Integer(1)]
(1, 1-d cone of Rational polyhedral fan in 3-d lattice N)
>>> fan.oriented_boundary(bdry[Integer(0)][Integer(1)])
-0-d cone of Rational polyhedral fan in 3-d lattice N
>>> fan.oriented_boundary(bdry[Integer(1)][Integer(1)])
-0-d cone of Rational polyhedral fan in 3-d lattice N
```

If you pass the fan itself, this method returns the orientation of the generating cones which is determined by the order of the rays in `cone.ray_basis()`

```
sage: fan.oriented_boundary(fan)                                          #␣
↪needs palp
-3-d cone of Rational polyhedral fan in 3-d lattice N
+ 3-d cone of Rational polyhedral fan in 3-d lattice N
- 3-d cone of Rational polyhedral fan in 3-d lattice N
+ 3-d cone of Rational polyhedral fan in 3-d lattice N
sage: [cone.rays().basis().matrix().det()                                 #␣
↪needs palp
....:  for cone in fan.generating_cones()]
[-1, 1, -1, 1]
```

```
>>> from sage.all import *
>>> fan.oriented_boundary(fan)                                          #␣
↪needs palp
-3-d cone of Rational polyhedral fan in 3-d lattice N
+ 3-d cone of Rational polyhedral fan in 3-d lattice N
- 3-d cone of Rational polyhedral fan in 3-d lattice N
+ 3-d cone of Rational polyhedral fan in 3-d lattice N
>>> [cone.rays().basis().matrix().det()                                 #␣
↪needs palp
...  for cone in fan.generating_cones()]
[-1, 1, -1, 1]
```

A non-full dimensional fan:

```
sage: cone = Cone([(4,5)])
sage: fan = Fan([cone])
sage: fan.oriented_boundary(cone)
0-d cone of Rational polyhedral fan in 2-d lattice N
sage: fan.oriented_boundary(fan)
1-d cone of Rational polyhedral fan in 2-d lattice N
```

```
>>> from sage.all import *
>>> cone = Cone([(Integer(4),Integer(5))])
>>> fan = Fan([cone])
>>> fan.oriented_boundary(cone)
0-d cone of Rational polyhedral fan in 2-d lattice N
>>> fan.oriented_boundary(fan)
1-d cone of Rational polyhedral fan in 2-d lattice N
```

plot(***options*)

Plot `self`.

INPUT:

- any options for toric plots (see *toric_plotter.options*), none are mandatory.

OUTPUT: a plot

EXAMPLES:

```
sage: fan = toric_varieties.dP6().fan()                                   #␣
↪needs palp
sage: fan.plot()                                                          #␣
```

(continues on next page)

(continued from previous page)

```
↪needs palp sage.plot
Graphics object consisting of 31 graphics primitives
```

```
>>> from sage.all import *
>>> fan = toric_varieties.dP6().fan()                                          #␣
↪needs palp
>>> fan.plot()                                                                 #␣
↪needs palp sage.plot
Graphics object consisting of 31 graphics primitives
```

primitive_collections()

Return the primitive collections.

OUTPUT:

Return the subsets $\{i_1, \dots, i_k\} \subset \{1, \dots, n\}$ such that

- The points $\{p_{i_1}, \dots, p_{i_k}\}$ do not span a cone of the fan.
- If you remove any one p_{i_j} from the set, then they do span a cone of the fan.

> **Note**
>
> By replacing the multiindices $\{i_1, \dots, i_k\}$ of each primitive collection with the monomials $x_{i_1} \cdots x_{i_k}$ one generates the Stanley-Reisner ideal in $\mathbf{Z}[x_1, \dots]$.

REFERENCES:

- [Bat1991]

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: fan.primitive_collections()
[frozenset({0, 4}),
 frozenset({2, 3}),
 frozenset({0, 1, 2}),
 frozenset({1, 3, 4})]
```

```
>>> from sage.all import *
>>> fan = Fan([[Integer(0),Integer(1),Integer(3)], [Integer(3),Integer(4)],␣
↪[Integer(2),Integer(0)], [Integer(1),Integer(2),Integer(4)]],
...           [(-Integer(3), -Integer(2), Integer(1)), (Integer(0),␣
↪Integer(0), Integer(1)), (Integer(3), -Integer(2), Integer(1)), (-
↪Integer(1), -Integer(1), Integer(1)), (Integer(1), -Integer(1),␣
↪Integer(1))])
>>> fan.primitive_collections()
[frozenset({0, 4}),
 frozenset({2, 3}),
 frozenset({0, 1, 2}),
 frozenset({1, 3, 4})]
```

subdivide(*new_rays=None*, *make_simplicial=False*, *algorithm='default'*, *verbose=False*)

Construct a new fan subdividing `self`.

INPUT:

- `new_rays` – list of new rays to be added during subdivision, each ray must be a list or a vector. May be empty or `None` (default);
- `make_simplicial` – if `True`, the returned fan is guaranteed to be simplicial, default is `False`;
- `algorithm` – string with the name of the algorithm used for subdivision. Currently there is only one available algorithm called "default";
- `verbose` – if `True`, some timing information may be printed during the process of subdivision

OUTPUT:

rational polyhedral fan

Currently the "default" algorithm corresponds to iterative stellar subdivision for each ray in `new_rays`.

EXAMPLES:

```
sage: fan = NormalFan(lattice_polytope.cross_polytope(3))
sage: fan.is_simplicial()
False
sage: fan.ngenerating_cones()
6
sage: fan.nrays()
8
sage: new_fan = fan.subdivide(new_rays=[(1,0,0)])
sage: new_fan.is_simplicial()
False
sage: new_fan.ngenerating_cones()
9
sage: new_fan.nrays()
9
```

```
>>> from sage.all import *
>>> fan = NormalFan(lattice_polytope.cross_polytope(Integer(3)))
>>> fan.is_simplicial()
False
>>> fan.ngenerating_cones()
6
>>> fan.nrays()
8
>>> new_fan = fan.subdivide(new_rays=[(Integer(1),Integer(0),Integer(0))])
>>> new_fan.is_simplicial()
False
>>> new_fan.ngenerating_cones()
9
>>> new_fan.nrays()
9
```

support_contains(**args*)

Check if a point is contained in the support of the fan.

The support of a fan is the union of all cones of the fan. If you want to know whether the fan contains a given cone, you should use `contains()` instead.

INPUT:

- `*args` – an element of `self.lattice()` or something that can be converted to it (for example, a list of coordinates).

OUTPUT:

`True` if `point` is contained in the support of the fan, `False` otherwise

toric_variety(**args*, ***kwds*)

Return the associated toric variety.

INPUT:

Same arguments as `ToricVariety()`.

OUTPUT: a toric variety

This is equivalent to the command `ToricVariety(self)` and is provided only as a convenient alternative method to go from the fan to the associated toric variety.

EXAMPLES:

```
sage: Fan([Cone([(1,0)]), Cone([(0,1)])]).toric_variety()
2-d toric variety covered by 2 affine patches
```

```
>>> from sage.all import *
>>> Fan([Cone([(Integer(1),Integer(0))]), Cone([(Integer(0),Integer(1))])]).
↪toric_variety()
2-d toric variety covered by 2 affine patches
```

vertex_graph()

Return the graph of 1- and 2-cones.

OUTPUT:

An edge-colored graph. The vertices correspond to the 1-cones (i.e. rays) of the fan. Two vertices are joined by an edge iff the rays span a 2-cone of the fan. The edges are colored by pairs of integers that classify the 2-cones up to $GL(2, \mathbf{Z})$ transformation, see `classify_cone_2d()`.

EXAMPLES:

```
sage: # needs palp
sage: dP8 = toric_varieties.dP8()
sage: g = dP8.fan().vertex_graph(); g
Graph on 4 vertices
sage: set(dP8.fan(1)) == set(g.vertices(sort=False))
True
sage: g.edge_labels()   # all edge labels the same since every cone is smooth
[(1, 0), (1, 0), (1, 0), (1, 0)]

sage: g = toric_varieties.Cube_deformation(10).fan().vertex_graph()
sage: g.automorphism_group().order()                                    #␣
↪needs sage.groups
48
sage: g.automorphism_group(edge_labels=True).order()                    #␣
↪needs sage.groups
4
```

```
>>> from sage.all import *
>>> # needs palp
>>> dP8 = toric_varieties.dP8()
>>> g = dP8.fan().vertex_graph(); g
Graph on 4 vertices
>>> set(dP8.fan(Integer(1))) == set(g.vertices(sort=False))
True
>>> g.edge_labels()   # all edge labels the same since every cone is smooth
[(1, 0), (1, 0), (1, 0), (1, 0)]

>>> g = toric_varieties.Cube_deformation(Integer(10)).fan().vertex_graph()
>>> g.automorphism_group().order()                                          #␣
↪needs sage.groups
48
>>> g.automorphism_group(edge_labels=True).order()                          #␣
↪needs sage.groups
4
```

virtual_rays(**args*)

Return (some of the) virtual rays of `self`.

Let N be the D-dimensional *lattice()* of a d-dimensional fan Σ in $N_{\mathbf{R}}$. Then the corresponding toric variety is of the form $X \times (\mathbf{C}^*)^{D-d}$. The actual *rays()* of Σ give a canonical choice of homogeneous coordinates on X. This function returns an arbitrary but fixed choice of virtual rays corresponding to a (non-canonical) choice of homogeneous coordinates on the torus factor. Combinatorially primitive integral generators of virtual rays span the $D - d$ dimensions of $N_{\mathbf{Q}}$ "missed" by the actual rays. (In general addition of virtual rays is not sufficient to span N over $\mathbf{Z}$.)

Note

You may use a particular choice of virtual rays by passing optional argument `virtual_rays` to the *Fan()* constructor.

INPUT:

- `ray_list` – list of integers; the indices of the requested virtual rays. If not specified, all virtual rays of `self` will be returned.

OUTPUT:

a *PointCollection* of primitive integral ray generators. Usually (if the fan is full-dimensional) this will be empty.

EXAMPLES:

```
sage: f = Fan([Cone([(1,0,1,0), (0,1,1,0)])])
sage: f.virtual_rays()
N(1, 0, 0, 0),
N(0, 0, 0, 1)
in 4-d lattice N

sage: f.rays()
N(1, 0, 1, 0),
N(0, 1, 1, 0)
```

(continues on next page)

(continued from previous page)

```
in 4-d lattice N

sage: f.virtual_rays([0])
N(1, 0, 0, 0)
in 4-d lattice N
```

```
>>> from sage.all import *
>>> f = Fan([Cone([(Integer(1),Integer(0),Integer(1),Integer(0)), (Integer(0),
↪Integer(1),Integer(1),Integer(0))])])
>>> f.virtual_rays()
N(1, 0, 0, 0),
N(0, 0, 0, 1)
in 4-d lattice N

>>> f.rays()
N(1, 0, 1, 0),
N(0, 1, 1, 0)
in 4-d lattice N

>>> f.virtual_rays([Integer(0)])
N(1, 0, 0, 0)
in 4-d lattice N
```

You can also give virtual ray indices directly, without packing them into a list:

```
sage: f.virtual_rays(0)
N(1, 0, 0, 0)
in 4-d lattice N
```

```
>>> from sage.all import *
>>> f.virtual_rays(Integer(0))
N(1, 0, 0, 0)
in 4-d lattice N
```

Make sure that Issue #16344 is fixed and one can compute the virtual rays of fans in non-saturated lattices:

```
sage: N = ToricLattice(1)
sage: B = N.submodule([(2,)]).basis()
sage: f = Fan([Cone([B[0]])])
sage: len(f.virtual_rays())
0
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(1))
>>> B = N.submodule([(Integer(2),)]).basis()
>>> f = Fan([Cone([B[Integer(0)]])])
>>> len(f.virtual_rays())
0
```

sage.geometry.fan.**discard_faces**(*cones*)

Return the cones of the given list which are not faces of each other.

INPUT:

- `cones` – list of *cones*

OUTPUT:

a list of *cones*, sorted by dimension in decreasing order

EXAMPLES:

Consider all cones of a fan:

```
sage: Sigma = toric_varieties.P2().fan()                                  #␣
↪needs palp
sage: cones = flatten(Sigma.cones())                                      #␣
↪needs palp
sage: len(cones)                                                          #␣
↪needs palp
7
```

```
>>> from sage.all import *
>>> Sigma = toric_varieties.P2().fan()                                  #␣
↪needs palp
>>> cones = flatten(Sigma.cones())                                      #␣
↪needs palp
>>> len(cones)                                                          #␣
↪needs palp
7
```

Most of them are not necessary to generate this fan:

```
sage: from sage.geometry.fan import discard_faces
sage: len(discard_faces(cones))                                           #␣
↪needs palp
3
sage: Sigma.ngenerating_cones()                                           #␣
↪needs palp
3
```

```
>>> from sage.all import *
>>> from sage.geometry.fan import discard_faces
>>> len(discard_faces(cones))                                           #␣
↪needs palp
3
>>> Sigma.ngenerating_cones()                                           #␣
↪needs palp
3
```

`sage.geometry.fan.is_Fan(x)`

Check if `x` is a Fan.

INPUT:

- `x` – anything

OUTPUT: `True` if `x` is a fan and `False` otherwise

EXAMPLES:

```
sage: from sage.geometry.fan import is_Fan
sage: is_Fan(1)
doctest:warning...
DeprecationWarning: The function is_Fan is deprecated; use 'isinstance(...,␣
↪RationalPolyhedralFan)' instead.
See https://github.com/sagemath/sage/issues/38126 for details.
False
sage: fan = toric_varieties.P2().fan(); fan                                  #␣
↪needs palp
Rational polyhedral fan in 2-d lattice N
sage: is_Fan(fan)                                                             #␣
↪needs palp
True
```

```
>>> from sage.all import *
>>> from sage.geometry.fan import is_Fan
>>> is_Fan(Integer(1))
doctest:warning...
DeprecationWarning: The function is_Fan is deprecated; use 'isinstance(...,␣
↪RationalPolyhedralFan)' instead.
See https://github.com/sagemath/sage/issues/38126 for details.
False
>>> fan = toric_varieties.P2().fan(); fan                                    #␣
↪needs palp
Rational polyhedral fan in 2-d lattice N
>>> is_Fan(fan)                                                               #␣
↪needs palp
True
```

2.5.6 Morphisms between toric lattices compatible with fans

This module is a part of the framework for toric varieties (`variety`, `fano_variety`). Its main purpose is to provide support for working with lattice morphisms compatible with fans via *`FanMorphism`* class.

AUTHORS:

- Andrey Novoseltsev (2010-10-17): initial version.
- **Andrey Novoseltsev (2011-04-11): added tests for injectivity/surjectivity,**
  fibration, bundle, as well as some related methods.

EXAMPLES:

Let's consider the face and normal fans of the "diamond" and the projection to the x-axis:

```
sage: diamond = lattice_polytope.cross_polytope(2)
sage: face = FaceFan(diamond, lattice=ToricLattice(2))
sage: normal = NormalFan(diamond)
sage: N = face.lattice()
sage: H = End(N)
sage: phi = H([N.0, 0])
sage: phi
Free module morphism defined by the matrix
[1 0]
[0 0]
```

(continues on next page)

(continued from previous page)

```
Domain: 2-d lattice N
Codomain: 2-d lattice N
sage: FanMorphism(phi, normal, face)
Traceback (most recent call last):
...
ValueError: the image of generating cone #1 of the domain fan
is not contained in a single cone of the codomain fan!
```

```
>>> from sage.all import *
>>> diamond = lattice_polytope.cross_polytope(Integer(2))
>>> face = FaceFan(diamond, lattice=ToricLattice(Integer(2)))
>>> normal = NormalFan(diamond)
>>> N = face.lattice()
>>> H = End(N)
>>> phi = H([N.gen(0), Integer(0)])
>>> phi
Free module morphism defined by the matrix
[1 0]
[0 0]
Domain: 2-d lattice N
Codomain: 2-d lattice N
>>> FanMorphism(phi, normal, face)
Traceback (most recent call last):
...
ValueError: the image of generating cone #1 of the domain fan
is not contained in a single cone of the codomain fan!
```

Some of the cones of the normal fan fail to be mapped to a single cone of the face fan. We can rectify the situation in the following way:

```
sage: fm = FanMorphism(phi, normal, face, subdivide=True)
sage: fm
Fan morphism defined by the matrix
[1 0]
[0 0]
Domain fan: Rational polyhedral fan in 2-d lattice N
Codomain fan: Rational polyhedral fan in 2-d lattice N
sage: fm.domain_fan().rays()
N( 1,  1),
N( 1, -1),
N(-1, -1),
N(-1,  1),
N( 0, -1),
N( 0,  1)
in 2-d lattice N
sage: normal.rays()
N( 1,  1),
N( 1, -1),
N(-1, -1),
N(-1,  1)
in 2-d lattice N
```

```
>>> from sage.all import *
>>> fm = FanMorphism(phi, normal, face, subdivide=True)
>>> fm
Fan morphism defined by the matrix
[1 0]
[0 0]
Domain fan: Rational polyhedral fan in 2-d lattice N
Codomain fan: Rational polyhedral fan in 2-d lattice N
>>> fm.domain_fan().rays()
N( 1,  1),
N( 1, -1),
N(-1, -1),
N(-1,  1),
N( 0, -1),
N( 0,  1)
in 2-d lattice N
>>> normal.rays()
N( 1,  1),
N( 1, -1),
N(-1, -1),
N(-1,  1)
in 2-d lattice N
```

As you see, it was necessary to insert two new rays (to prevent "upper" and "lower" cones of the normal fan from being mapped to the whole x-axis).

class sage.geometry.fan_morphism.**FanMorphism**(*morphism*, *domain_fan*, *codomain=None*, *subdivide=False*, *check=True*, *verbose=False*)

Bases: `FreeModuleMorphism`

Create a fan morphism.

Let Σ_1 and Σ_2 be two fans in lattices N_1 and N_2 respectively. Let ϕ be a morphism (i.e. a linear map) from N_1 to N_2. We say that ϕ is *compatible* with Σ_1 and Σ_2 if every cone $\sigma_1 \in \Sigma_1$ is mapped by ϕ into a single cone $\sigma_2 \in \Sigma_2$, i.e. $\phi(\sigma_1) \subset \sigma_2$ (σ_2 may be different for different σ_1).

By a **fan morphism** we understand a morphism between two lattices compatible with specified fans in these lattices. Such morphisms behave in exactly the same way as "regular" morphisms between lattices, but:

- fan morphisms have a special constructor allowing some automatic adjustments to the initial fans (see below);
- fan morphisms are aware of the associated fans and they can be accessed via `codomain_fan()` and `domain_fan()`;
- fan morphisms can efficiently compute `image_cone()` of a given cone of the domain fan and `preimage_cones()` of a given cone of the codomain fan.

INPUT:

- `morphism` – either a morphism between domain and codomain, or an integral matrix defining such a morphism;
- `domain_fan` – a `fan` in the domain
- `codomain` – (default: `None`) either a codomain lattice or a fan in the codomain. If the codomain fan is not given, the image fan (fan generated by images of generating cones) of `domain_fan` will be used, if possible.
- `subdivide` – boolean (default: `False`); if `True` and `domain_fan` is not compatible with the codomain fan because it is too coarse, it will be automatically refined to become compatible (the minimal refinement is

canonical, so there are no choices involved)

- `check` – boolean (default: `True`); if `False`, given fans and morphism will be assumed to be compatible. Be careful when using this option, since wrong assumptions can lead to wrong and hard-to-detect errors. On the other hand, this option may save you some time.
- `verbose` – boolean (default: `False`); if `True`, some information may be printed during construction of the fan morphism

OUTPUT: a fan morphism

EXAMPLES:

Here we consider the face and normal fans of the “diamond” and the projection to the x-axis:

```
sage: diamond = lattice_polytope.cross_polytope(2)
sage: face = FaceFan(diamond, lattice=ToricLattice(2))
sage: normal = NormalFan(diamond)
sage: N = face.lattice()
sage: H = End(N)
sage: phi = H([N.0, 0])
sage: phi
Free module morphism defined by the matrix
[1 0]
[0 0]
Domain: 2-d lattice N
Codomain: 2-d lattice N
sage: fm = FanMorphism(phi, face, normal)
sage: fm.domain_fan() is face
True
```

```
>>> from sage.all import *
>>> diamond = lattice_polytope.cross_polytope(Integer(2))
>>> face = FaceFan(diamond, lattice=ToricLattice(Integer(2)))
>>> normal = NormalFan(diamond)
>>> N = face.lattice()
>>> H = End(N)
>>> phi = H([N.gen(0), Integer(0)])
>>> phi
Free module morphism defined by the matrix
[1 0]
[0 0]
Domain: 2-d lattice N
Codomain: 2-d lattice N
>>> fm = FanMorphism(phi, face, normal)
>>> fm.domain_fan() is face
True
```

Note, that since `phi` is compatible with these fans, the returned fan is exactly the same object as the initial `domain_fan`.

```
sage: FanMorphism(phi, normal, face)
Traceback (most recent call last):
...
ValueError: the image of generating cone #1 of the domain fan
is not contained in a single cone of the codomain fan!
```

(continues on next page)

(continued from previous page)

```
sage: fm = FanMorphism(phi, normal, face, subdivide=True)
sage: fm.domain_fan() is normal
False
sage: fm.domain_fan().ngenerating_cones()
6
```

```
>>> from sage.all import *
>>> FanMorphism(phi, normal, face)
Traceback (most recent call last):
...
ValueError: the image of generating cone #1 of the domain fan
is not contained in a single cone of the codomain fan!
>>> fm = FanMorphism(phi, normal, face, subdivide=True)
>>> fm.domain_fan() is normal
False
>>> fm.domain_fan().ngenerating_cones()
6
```

We had to subdivide two of the four cones of the normal fan, since they were mapped by `phi` into non-strictly convex cones.

It is possible to omit the codomain fan, in which case the image fan will be used instead of it:

```
sage: fm = FanMorphism(phi, face)
sage: fm.codomain_fan()
Rational polyhedral fan in 2-d lattice N
sage: fm.codomain_fan().rays()
N( 1, 0),
N(-1, 0)
in 2-d lattice N
```

```
>>> from sage.all import *
>>> fm = FanMorphism(phi, face)
>>> fm.codomain_fan()
Rational polyhedral fan in 2-d lattice N
>>> fm.codomain_fan().rays()
N( 1, 0),
N(-1, 0)
in 2-d lattice N
```

Now we demonstrate a more subtle example. We take the first quadrant as our domain fan. Then we divide the first quadrant into three cones, throw away the middle one and take the other two as our codomain fan. These fans are incompatible with the identity lattice morphism since the image of the domain fan is out of the support of the codomain fan:

```
sage: N = ToricLattice(2)
sage: phi = End(N).identity()
sage: F1 = Fan(cones=[(0,1)], rays=[(1,0), (0,1)])
sage: F2 = Fan(cones=[(0,1), (2,3)],
....:          rays=[(1,0), (2,1), (1,2), (0,1)])
sage: FanMorphism(phi, F1, F2)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: the image of generating cone #0 of the domain fan
is not contained in a single cone of the codomain fan!
sage: FanMorphism(phi, F1, F2, subdivide=True)
Traceback (most recent call last):
...
ValueError: morphism defined by
[1 0]
[0 1]
does not map
Rational polyhedral fan in 2-d lattice N
into the support of
Rational polyhedral fan in 2-d lattice N!
```

```
>>> from sage.all import *
>>> N = ToricLattice(Integer(2))
>>> phi = End(N).identity()
>>> F1 = Fan(cones=[(Integer(0),Integer(1))], rays=[(Integer(1),Integer(0)),
↪(Integer(0),Integer(1))])
>>> F2 = Fan(cones=[(Integer(0),Integer(1)), (Integer(2),Integer(3))],
...          rays=[(Integer(1),Integer(0)), (Integer(2),Integer(1)), (Integer(1),
↪Integer(2)), (Integer(0),Integer(1))])
>>> FanMorphism(phi, F1, F2)
Traceback (most recent call last):
...
ValueError: the image of generating cone #0 of the domain fan
is not contained in a single cone of the codomain fan!
>>> FanMorphism(phi, F1, F2, subdivide=True)
Traceback (most recent call last):
...
ValueError: morphism defined by
[1 0]
[0 1]
does not map
Rational polyhedral fan in 2-d lattice N
into the support of
Rational polyhedral fan in 2-d lattice N!
```

The problem was detected and handled correctly (i.e. an exception was raised). However, the used algorithm requires extra checks for this situation after constructing a potential subdivision and this can take significant time. You can save about half the time using `check=False` option, if you know in advance that it is possible to make fans compatible with the morphism by subdividing the domain fan. Of course, if your assumption was incorrect, the result will be wrong and you will get a fan which *does* map into the support of the codomain fan, but is **not** a subdivision of the domain fan. You can test it on the example above:

```
sage: fm = FanMorphism(phi, F1, F2, subdivide=True,
....:                  check=False, verbose=True)
Placing ray images (... ms)
Computing chambers (... ms)
Number of domain cones: 1.
Number of chambers: 2.
Cone 0 sits in chambers 0 1 (... ms)
```

(continues on next page)

(continued from previous page)

```
sage: fm.domain_fan().is_equivalent(F2)
True
```

```
>>> from sage.all import *
>>> fm = FanMorphism(phi, F1, F2, subdivide=True,
...                  check=False, verbose=True)
Placing ray images (... ms)
Computing chambers (... ms)
Number of domain cones: 1.
Number of chambers: 2.
Cone 0 sits in chambers 0 1 (... ms)
>>> fm.domain_fan().is_equivalent(F2)
True
```

codomain_fan(*dim=None*, *codim=None*)

Return the codomain fan of `self`.

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: quadrant = Cone([(1,0), (0,1)])
sage: quadrant = Fan([quadrant])
sage: quadrant_bl = quadrant.subdivide([(1,1)])
sage: fm = FanMorphism(identity_matrix(2), quadrant_bl, quadrant)
sage: fm.codomain_fan()
Rational polyhedral fan in 2-d lattice N
sage: fm.codomain_fan() is quadrant
True
```

```
>>> from sage.all import *
>>> quadrant = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> quadrant = Fan([quadrant])
>>> quadrant_bl = quadrant.subdivide([(Integer(1),Integer(1))])
>>> fm = FanMorphism(identity_matrix(Integer(2)), quadrant_bl, quadrant)
>>> fm.codomain_fan()
Rational polyhedral fan in 2-d lattice N
>>> fm.codomain_fan() is quadrant
True
```

domain_fan(*dim=None*, *codim=None*)

Return the codomain fan of `self`.

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: quadrant = Cone([(1,0), (0,1)])
sage: quadrant = Fan([quadrant])
sage: quadrant_bl = quadrant.subdivide([(1,1)])
sage: fm = FanMorphism(identity_matrix(2), quadrant_bl, quadrant)
sage: fm.domain_fan()
Rational polyhedral fan in 2-d lattice N
sage: fm.domain_fan() is quadrant_bl
True
```

```
>>> from sage.all import *
>>> quadrant = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> quadrant = Fan([quadrant])
>>> quadrant_bl = quadrant.subdivide([(Integer(1),Integer(1))])
>>> fm = FanMorphism(identity_matrix(Integer(2)), quadrant_bl, quadrant)
>>> fm.domain_fan()
Rational polyhedral fan in 2-d lattice N
>>> fm.domain_fan() is quadrant_bl
True
```

factor()

Factor `self` into injective * birational * surjective morphisms.

OUTPUT:

- a triple of *FanMorphism* (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 $\phi_i \circ \phi_b \circ \phi_s$.

Intermediate fans live in the saturation of the image of `self` as a map between lattices and are the image of the *domain_fan()* and the restriction of the *codomain_fan()*, i.e. if `self` maps $\Sigma \to \Sigma'$, then we have factorization into

$$\Sigma \twoheadrightarrow \Sigma_s \to \Sigma_i \hookrightarrow \Sigma.$$

Note

- Σ_s is the finest fan with the smallest support that is compatible with `self`: any fan morphism from Σ given by the same map of lattices as `self` factors through Σ_s.
- Σ_i is the coarsest fan of the largest support that is compatible with `self`: any fan morphism into Σ' given by the same map of lattices as `self` factors though Σ_i.

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)                                          #␣
```

(continues on next page)

(continued from previous page)

```
↪needs palp
sage: m = matrix([(2,0,0), (1,1,0)])
sage: phi = A2.hom(m, P3)                                                   #␣
↪needs palp
sage: phi.as_polynomial_map()                                               #␣
↪needs palp
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]
```

```
>>> from sage.all import *
>>> A2 = toric_varieties.A2()
>>> P3 = toric_varieties.P(Integer(3))                                       ␣
↪      # needs palp
>>> m = matrix([(Integer(2),Integer(0),Integer(0)), (Integer(1),Integer(1),
↪Integer(0))])
>>> phi = A2.hom(m, P3)                                                    #␣
↪needs palp
>>> phi.as_polynomial_map()                                                #␣
↪needs palp
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]
```

Now we will work with the underlying fan morphism:

```
sage: # needs palp
sage: phi = phi.fan_morphism(); phi
Fan morphism defined by the matrix
[2 0 0]
[1 1 0]
Domain fan: Rational polyhedral fan in 2-d lattice N
Codomain fan: Rational polyhedral fan in 3-d lattice N
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
```

```
>>> from sage.all import *
>>> # needs palp
>>> phi = phi.fan_morphism(); phi
Fan morphism defined by the matrix
[2 0 0]
[1 1 0]
Domain fan: Rational polyhedral fan in 2-d lattice N
```

(continues on next page)

(continued from previous page)

```
Codomain fan: Rational polyhedral fan in 3-d lattice N
>>> phi.is_surjective(), phi.is_birational(), phi.is_injective()
(False, False, False)
>>> phi_i, phi_b, phi_s = phi.factor()
>>> phi_s.is_surjective(), phi_b.is_birational(), phi_i.is_injective()
(True, True, True)
>>> prod(phi.factor()) == phi
True
```

Double cover (surjective):

```
sage: A2.fan().rays()
N(1, 0),
N(0, 1)
in 2-d lattice N
sage: phi_s                                                                  #␣
↪needs palp
Fan morphism defined by the matrix
[2 0]
[1 1]
Domain fan: Rational polyhedral fan in 2-d lattice N
Codomain fan: Rational polyhedral fan in Sublattice <N(1, 0, 0), N(0, 1, 0)>
sage: phi_s.codomain_fan().rays()                                            #␣
↪needs palp
N(1, 0, 0),
N(1, 1, 0)
in Sublattice <N(1, 0, 0), N(0, 1, 0)>
```

```
>>> from sage.all import *
>>> A2.fan().rays()
N(1, 0),
N(0, 1)
in 2-d lattice N
>>> phi_s                                                                  #␣
↪needs palp
Fan morphism defined by the matrix
[2 0]
[1 1]
Domain fan: Rational polyhedral fan in 2-d lattice N
Codomain fan: Rational polyhedral fan in Sublattice <N(1, 0, 0), N(0, 1, 0)>
>>> phi_s.codomain_fan().rays()                                            #␣
↪needs palp
N(1, 0, 0),
N(1, 1, 0)
in Sublattice <N(1, 0, 0), N(0, 1, 0)>
```

Blowup chart (birational):

```
sage: phi_b                                                                  #␣
↪needs palp
Fan morphism defined by the matrix
[1 0]
```

(continues on next page)

(continued from previous page)

```
[0 1]
Domain fan: Rational polyhedral fan in Sublattice <N(1, 0, 0), N(0, 1, 0)>
Codomain fan: Rational polyhedral fan in Sublattice <N(1, 0, 0), N(0, 1, 0)>
sage: phi_b.codomain_fan().rays()                                          #␣
↪needs palp
N(-1, -1, 0),
N( 0,  1, 0),
N( 1,  0, 0)
in Sublattice <N(1, 0, 0), N(0, 1, 0)>
```

```
>>> from sage.all import *
>>> phi_b                                                                 #␣
↪needs palp
Fan morphism defined by the matrix
[1 0]
[0 1]
Domain fan: Rational polyhedral fan in Sublattice <N(1, 0, 0), N(0, 1, 0)>
Codomain fan: Rational polyhedral fan in Sublattice <N(1, 0, 0), N(0, 1, 0)>
>>> phi_b.codomain_fan().rays()                                           #␣
↪needs palp
N(-1, -1, 0),
N( 0,  1, 0),
N( 1,  0, 0)
in Sublattice <N(1, 0, 0), N(0, 1, 0)>
```

Coordinate plane inclusion (injective):

```
sage: phi_i                                                                #␣
↪needs palp
Fan morphism defined by the matrix
[1 0 0]
[0 1 0]
Domain fan: Rational polyhedral fan in Sublattice <N(1, 0, 0), N(0, 1, 0)>
Codomain fan: Rational polyhedral fan in 3-d lattice N
sage: phi.codomain_fan().rays()                                            #␣
↪needs palp
N( 1,  0,  0),
N( 0,  1,  0),
N( 0,  0,  1),
N(-1, -1, -1)
in 3-d lattice N
```

```
>>> from sage.all import *
>>> phi_i                                                                 #␣
↪needs palp
Fan morphism defined by the matrix
[1 0 0]
[0 1 0]
Domain fan: Rational polyhedral fan in Sublattice <N(1, 0, 0), N(0, 1, 0)>
Codomain fan: Rational polyhedral fan in 3-d lattice N
>>> phi.codomain_fan().rays()                                             #␣
↪needs palp
```

(continues on next page)

(continued from previous page)

```
N( 1,  0,  0),
N( 0,  1,  0),
N( 0,  0,  1),
N(-1, -1, -1)
in 3-d lattice N
```

image_cone(*cone*)

Return the cone of the codomain fan containing the image of `cone`.

INPUT:

- `cone` – a *cone* equivalent to a cone of the *domain_fan()* of `self`.

OUTPUT:

- a *cone* of the *codomain_fan()* of `self`.

EXAMPLES:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: quadrant = Fan([quadrant])
sage: quadrant_bl = quadrant.subdivide([(1,1)])
sage: fm = FanMorphism(identity_matrix(2), quadrant_bl, quadrant)
sage: fm.image_cone(Cone([(1,0)]))
1-d cone of Rational polyhedral fan in 2-d lattice N
sage: fm.image_cone(Cone([(1,1)]))
2-d cone of Rational polyhedral fan in 2-d lattice N
```

```
>>> from sage.all import *
>>> quadrant = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> quadrant = Fan([quadrant])
>>> quadrant_bl = quadrant.subdivide([(Integer(1),Integer(1))])
>>> fm = FanMorphism(identity_matrix(Integer(2)), quadrant_bl, quadrant)
>>> fm.image_cone(Cone([(Integer(1),Integer(0))]))
1-d cone of Rational polyhedral fan in 2-d lattice N
>>> fm.image_cone(Cone([(Integer(1),Integer(1))]))
2-d cone of Rational polyhedral fan in 2-d lattice N
```

index(*cone=None*)

Return the index of `self` as a map between lattices.

INPUT:

- `cone` – (default: `None`) a *cone* of the *codomain_fan()* of `self`.

OUTPUT: integer, infinity, or `None`

If no cone was specified, this function computes the index of the image of `self` in the codomain. If a cone σ was given, the index of `self` over σ is computed in the sense of Definition 2.1.7 of [HLY2002]: if σ' is any cone of the *domain_fan()* of `self` whose relative interior is mapped to the relative interior of σ, it is the index of the image of $N'(\sigma')$ in $N(\sigma)$, where N' and N are domain and codomain lattices respectively. While that definition was formulated for the case of the finite index only, we extend it to the infinite one as well and return `None` if there is no σ' at all. See examples below for situations when such things happen. Note also that the index of `self` is the same as index over the trivial cone.

EXAMPLES:

```
sage: # needs palp
sage: Sigma = toric_varieties.dP8().fan()
sage: Sigma_p = toric_varieties.P1().fan()
sage: phi = FanMorphism(matrix([[1], [-1]]), Sigma, Sigma_p)
sage: phi.index()
1
sage: psi = FanMorphism(matrix([[2], [-2]]), Sigma, Sigma_p)
sage: psi.index()
2
sage: xi = FanMorphism(matrix([[1, 0]]), Sigma_p, Sigma)
sage: xi.index()
+Infinity
```

```
>>> from sage.all import *
>>> # needs palp
>>> Sigma = toric_varieties.dP8().fan()
>>> Sigma_p = toric_varieties.P1().fan()
>>> phi = FanMorphism(matrix([[Integer(1)], [-Integer(1)]]), Sigma, Sigma_p)
>>> phi.index()
1
>>> psi = FanMorphism(matrix([[Integer(2)], [-Integer(2)]]), Sigma, Sigma_p)
>>> psi.index()
2
>>> xi = FanMorphism(matrix([[Integer(1), Integer(0)]]), Sigma_p, Sigma)
>>> xi.index()
+Infinity
```

Infinite index in the last example indicates that the image has positive codimension in the codomain. Let's look at the rays of our fans:

```
sage: Sigma_p.rays()                                                    #␣
↪needs palp
N( 1),
N(-1)
in 1-d lattice N
sage: Sigma.rays()                                                      #␣
↪needs palp
N( 1,  1),
N( 0,  1),
N(-1, -1),
N( 1,  0)
in 2-d lattice N
sage: xi.factor()[0].domain_fan().rays()                                #␣
↪needs palp
N(-1, 0),
N( 1, 0)
in Sublattice <N(1, 0)>
```

```
>>> from sage.all import *
>>> Sigma_p.rays()                                                    #␣
↪needs palp
N( 1),
```

(continues on next page)

(continued from previous page)

```
N(-1)
in 1-d lattice N
>>> Sigma.rays()                                                            #␣
↪needs palp
N( 1,  1),
N( 0,  1),
N(-1, -1),
N( 1,  0)
in 2-d lattice N
>>> xi.factor()[Integer(0)].domain_fan().rays()                              ␣
↪      # needs palp
N(-1, 0),
N( 1, 0)
in Sublattice <N(1, 0)>
```

We see that one of the rays of the fan of `P1` is mapped to a ray, while the other one to the interior of some 2-d cone. Both rays correspond to single points on `P1`, yet one is mapped to the distinguished point of a torus invariant curve of `dP8` (with the rest of this curve being uncovered) and the other to a fixed point of `dP8` (thus completely covering this torus orbit in `dP8`).

We should therefore expect the following behaviour: all indices over 1-d cones are `None`, except for one which is infinite, and all indices over 2-d cones are `None`, except for one which is 1:

```
sage: [xi.index(cone) for cone in Sigma(1)]                                   #␣
↪needs palp
[None, None, None, +Infinity]
sage: [xi.index(cone) for cone in Sigma(2)]                                   #␣
↪needs palp
[None, 1, None, None]
```

```
>>> from sage.all import *
>>> [xi.index(cone) for cone in Sigma(Integer(1))]                             ␣
↪      # needs palp
[None, None, None, +Infinity]
>>> [xi.index(cone) for cone in Sigma(Integer(2))]                             ␣
↪      # needs palp
[None, 1, None, None]
```

is_birational()

Check if `self` is birational.

OUTPUT: `True` if `self` is birational, `False` otherwise

For fan morphisms this check is equivalent to `self.index() == 1` and means that the corresponding map between toric varieties is birational.

EXAMPLES:

```
sage: # needs palp
sage: Sigma = toric_varieties.dP8().fan()
sage: Sigma_p = toric_varieties.P1().fan()
sage: phi = FanMorphism(matrix([[1], [-1]]), Sigma, Sigma_p)
sage: psi = FanMorphism(matrix([[2], [-2]]), Sigma, Sigma_p)
sage: xi = FanMorphism(matrix([[1, 0]]), Sigma_p, Sigma)
```

(continues on next page)

(continued from previous page)

```
sage: phi.index(), psi.index(), xi.index()
(1, 2, +Infinity)
sage: phi.is_birational(), psi.is_birational(), xi.is_birational()
(True, False, False)
```

```
>>> from sage.all import *
>>> # needs palp
>>> Sigma = toric_varieties.dP8().fan()
>>> Sigma_p = toric_varieties.P1().fan()
>>> phi = FanMorphism(matrix([[Integer(1)], [-Integer(1)]]), Sigma, Sigma_p)
>>> psi = FanMorphism(matrix([[Integer(2)], [-Integer(2)]]), Sigma, Sigma_p)
>>> xi = FanMorphism(matrix([[Integer(1), Integer(0)]]), Sigma_p, Sigma)
>>> phi.index(), psi.index(), xi.index()
(1, 2, +Infinity)
>>> phi.is_birational(), psi.is_birational(), xi.is_birational()
(True, False, False)
```

is_bundle()

Check if `self` is a bundle.

OUTPUT: `True` if `self` is a bundle, `False` otherwise

Let $\phi : \Sigma \to \Sigma'$ be a fan morphism such that the underlying lattice morphism $\phi : N \to N'$ is surjective. Let Σ_0 be the kernel fan of ϕ. Then ϕ is a **bundle** (or splitting) if there is a subfan $\widehat{\Sigma}$ of Σ such that the following two conditions are satisfied:

1. Cones of Σ are precisely the cones of the form $\sigma_0 + \widehat{\sigma}$, where $\sigma_0 \in \Sigma_0$ and $\widehat{\sigma} \in \widehat{\Sigma}$.
2. Cones of $\widehat{\Sigma}$ are in bijection with cones of Σ' induced by ϕ and ϕ maps lattice points in every cone $\widehat{\sigma} \in \widehat{\Sigma}$ bijectively onto lattice points in $\phi(\widehat{\sigma})$.

If a fan morphism $\phi : \Sigma \to \Sigma'$ is a bundle, then X_Σ is a fiber bundle over $X_{\Sigma'}$ with fibers X_{Σ_0, N_0}, where N_0 is the kernel lattice of ϕ. See [CLS2011] for more details.

See also

is_fibration(), *kernel_fan()*.

EXAMPLES:

We consider several maps between fans of a del Pezzo surface and the projective line:

```
sage: # needs palp
sage: Sigma = toric_varieties.dP8().fan()
sage: Sigma_p = toric_varieties.P1().fan()
sage: phi = FanMorphism(matrix([[1], [-1]]), Sigma, Sigma_p)
sage: psi = FanMorphism(matrix([[2], [-2]]), Sigma, Sigma_p)
sage: xi = FanMorphism(matrix([[1, 0]]), Sigma_p, Sigma)
sage: phi.is_bundle()
True
sage: phi.is_fibration()
True
sage: phi.index()
1
```

(continues on next page)

(continued from previous page)

```
sage: psi.is_bundle()
False
sage: psi.is_fibration()
True
sage: psi.index()
2
sage: xi.is_fibration()
False
sage: xi.index()
+Infinity
```

```
>>> from sage.all import *
>>> # needs palp
>>> Sigma = toric_varieties.dP8().fan()
>>> Sigma_p = toric_varieties.P1().fan()
>>> phi = FanMorphism(matrix([[Integer(1)], [-Integer(1)]]), Sigma, Sigma_p)
>>> psi = FanMorphism(matrix([[Integer(2)], [-Integer(2)]]), Sigma, Sigma_p)
>>> xi = FanMorphism(matrix([[Integer(1), Integer(0)]]), Sigma_p, Sigma)
>>> phi.is_bundle()
True
>>> phi.is_fibration()
True
>>> phi.index()
1
>>> psi.is_bundle()
False
>>> psi.is_fibration()
True
>>> psi.index()
2
>>> xi.is_fibration()
False
>>> xi.index()
+Infinity
```

The first of these maps induces not only a fibration, but a fiber bundle structure. The second map is very similar, yet it fails to be a bundle, as its index is 2. The last map is not even a fibration.

is_dominant()

Return whether the fan morphism is dominant.

A fan morphism ϕ is dominant if it is surjective as a map of vector spaces. That is, $\phi_{\mathbf{R}} : N_{\mathbf{R}} \to N'_{\mathbf{R}}$ is surjective.

If the domain fan is *complete*, then this implies that the fan morphism is *surjective*.

If the fan morphism is dominant, then the associated morphism of toric varieties is dominant in the algebraic-geometric sense (that is, surjective onto a dense subset).

OUTPUT: boolean

EXAMPLES:

```
sage: P1 = toric_varieties.P1()
sage: A1 = toric_varieties.A1()
```

(continues on next page)

(continued from previous page)

```
sage: phi = FanMorphism(matrix([[1]]), A1.fan(), P1.fan())
sage: phi.is_dominant()
True
sage: phi.is_surjective()
False
```

```
>>> from sage.all import *
>>> P1 = toric_varieties.P1()
>>> A1 = toric_varieties.A1()
>>> phi = FanMorphism(matrix([[Integer(1)]]), A1.fan(), P1.fan())
>>> phi.is_dominant()
True
>>> phi.is_surjective()
False
```

is_fibration()

Check if `self` is a fibration.

OUTPUT: `True` if `self` is a fibration, `False` otherwise

A fan morphism $\phi : \Sigma \to \Sigma'$ is a **fibration** if for any cone $\sigma' \in \Sigma'$ and any primitive preimage cone $\sigma \in \Sigma$ corresponding to σ' the linear map of vector spaces $\phi_{\mathbf{R}}$ induces a bijection between σ and σ', and, in addition, ϕ is `dominant` (that is, $\phi_{\mathbf{R}} : N_{\mathbf{R}} \to N'_{\mathbf{R}}$ is surjective).

If a fan morphism $\phi : \Sigma \to \Sigma'$ is a fibration, then the associated morphism between toric varieties $\tilde{\phi} : X_\Sigma \to X_{\Sigma'}$ is a fibration in the sense that it is surjective and all of its fibers have the same dimension, namely $\dim X_\Sigma - \dim X_{\Sigma'}$. These fibers do *not* have to be isomorphic, i.e. a fibration is not necessarily a fiber bundle. See [HLY2002] for more details.

See also

`is_bundle()`, `primitive_preimage_cones()`.

EXAMPLES:

We consider several maps between fans of a del Pezzo surface and the projective line:

```
sage: # needs palp
sage: Sigma = toric_varieties.dP8().fan()
sage: Sigma_p = toric_varieties.P1().fan()
sage: phi = FanMorphism(matrix([[1], [-1]]), Sigma, Sigma_p)
sage: psi = FanMorphism(matrix([[2], [-2]]), Sigma, Sigma_p)
sage: xi = FanMorphism(matrix([[1, 0]]), Sigma_p, Sigma)
sage: phi.is_bundle()
True
sage: phi.is_fibration()
True
sage: phi.index()
1
sage: psi.is_bundle()
False
sage: psi.is_fibration()
True
```

(continues on next page)

(continued from previous page)

```
sage: psi.index()
2
sage: xi.is_fibration()
False
sage: xi.index()
+Infinity
```

```
>>> from sage.all import *
>>> # needs palp
>>> Sigma = toric_varieties.dP8().fan()
>>> Sigma_p = toric_varieties.P1().fan()
>>> phi = FanMorphism(matrix([[Integer(1)], [-Integer(1)]]), Sigma, Sigma_p)
>>> psi = FanMorphism(matrix([[Integer(2)], [-Integer(2)]]), Sigma, Sigma_p)
>>> xi = FanMorphism(matrix([[Integer(1), Integer(0)]]), Sigma_p, Sigma)
>>> phi.is_bundle()
True
>>> phi.is_fibration()
True
>>> phi.index()
1
>>> psi.is_bundle()
False
>>> psi.is_fibration()
True
>>> psi.index()
2
>>> xi.is_fibration()
False
>>> xi.index()
+Infinity
```

The first of these maps induces not only a fibration, but a fiber bundle structure. The second map is very similar, yet it fails to be a bundle, as its index is 2. The last map is not even a fibration.

is_injective()

Check if `self` is injective.

OUTPUT: `True` if `self` is injective, `False` otherwise

Let $\phi : \Sigma \to \Sigma'$ be a fan morphism such that the underlying lattice morphism $\phi : N \to N'$ bijectively maps N to a *saturated* sublattice of N'. Let $\psi : \Sigma \to \Sigma'_0$ be the restriction of ϕ to the image. Then ϕ is **injective** if the map between cones corresponding to ψ (injectively) maps each cone of Σ to a cone of the same dimension.

If a fan morphism $\phi : \Sigma \to \Sigma'$ is injective, then the associated morphism between toric varieties $\tilde{\phi} : X_\Sigma \to X_{\Sigma'}$ is injective.

See also

factor().

EXAMPLES:

Consider the fan of the affine plane:

```
sage: A2 = toric_varieties.A(2).fan()
```

```
>>> from sage.all import *
>>> A2 = toric_varieties.A(Integer(2)).fan()
```

We will map several fans consisting of a single ray into the interior of the 2-cone:

```
sage: Sigma = Fan([Cone([(1,1)])])
sage: m = identity_matrix(2)
sage: FanMorphism(m, Sigma, A2).is_injective()
False
```

```
>>> from sage.all import *
>>> Sigma = Fan([Cone([(Integer(1),Integer(1))])])
>>> m = identity_matrix(Integer(2))
>>> FanMorphism(m, Sigma, A2).is_injective()
False
```

This morphism was not injective since (in the toric varieties interpretation) the 1-dimensional orbit corresponding to the ray was mapped to the 0-dimensional orbit corresponding to the 2-cone.

```
sage: Sigma = Fan([Cone([(1,)])])
sage: m = matrix(1, 2, [1,1])
sage: FanMorphism(m, Sigma, A2).is_injective()
True
```

```
>>> from sage.all import *
>>> Sigma = Fan([Cone([(Integer(1),)])])
>>> m = matrix(Integer(1), Integer(2), [Integer(1),Integer(1)])
>>> FanMorphism(m, Sigma, A2).is_injective()
True
```

While the fans in this example are close to the previous one, here the ray corresponds to a 0-dimensional orbit.

```
sage: Sigma = Fan([Cone([(1,)])])
sage: m = matrix(1, 2, [2,2])
sage: FanMorphism(m, Sigma, A2).is_injective()
False
```

```
>>> from sage.all import *
>>> Sigma = Fan([Cone([(Integer(1),)])])
>>> m = matrix(Integer(1), Integer(2), [Integer(2),Integer(2)])
>>> FanMorphism(m, Sigma, A2).is_injective()
False
```

Here the problem is that `m` maps the domain lattice to a non-saturated sublattice of the codomain. The corresponding map of the toric varieties is a two-sheeted cover of its image.

We also embed the affine plane into the projective one:

```
sage: P2 = toric_varieties.P(2).fan()                                    #␣
↪needs palp
sage: m = identity_matrix(2)
```

(continues on next page)

(continued from previous page)

```
sage: FanMorphism(m, A2, P2).is_injective()                                   #␣
↪needs palp
True
```

```
>>> from sage.all import *
>>> P2 = toric_varieties.P(Integer(2)).fan()                                 ␣
↪     # needs palp
>>> m = identity_matrix(Integer(2))
>>> FanMorphism(m, A2, P2).is_injective()                                  #␣
↪needs palp
True
```

is_surjective()

Check if `self` is surjective.

OUTPUT: `True` if `self` is surjective, `False` otherwise

A fan morphism $\phi : \Sigma \to \Sigma'$ is **surjective** if the corresponding map between cones is surjective, i.e. for each cone $\sigma' \in \Sigma'$ there is at least one preimage cone $\sigma \in \Sigma$ such that the relative interior of σ is mapped to the relative interior of σ' and, in addition, $\phi_\mathbf{R} : N_\mathbf{R} \to N'_\mathbf{R}$ is surjective.

If a fan morphism $\phi : \Sigma \to \Sigma'$ is surjective, then the associated morphism between toric varieties $\tilde{\phi} : X_\Sigma \to X_{\Sigma'}$ is surjective.

See also

is_bundle(), *is_fibration()*, *preimage_cones()*, `is_complete()`.

EXAMPLES:

We check that the blow up of the affine plane at the origin is surjective:

```
sage: A2 = toric_varieties.A(2).fan()
sage: Bl = A2.subdivide([(1,1)])
sage: m = identity_matrix(2)
sage: FanMorphism(m, Bl, A2).is_surjective()
True
```

```
>>> from sage.all import *
>>> A2 = toric_varieties.A(Integer(2)).fan()
>>> Bl = A2.subdivide([(Integer(1),Integer(1))])
>>> m = identity_matrix(Integer(2))
>>> FanMorphism(m, Bl, A2).is_surjective()
True
```

It remains surjective if we throw away "south and north poles" of the exceptional divisor:

```
sage: FanMorphism(m, Fan(Bl.cones(1)), A2).is_surjective()
True
```

```
>>> from sage.all import *
>>> FanMorphism(m, Fan(Bl.cones(Integer(1))), A2).is_surjective()
True
```

But a single patch of the blow up does not cover the plane:

```
sage: F = Fan([Bl.generating_cone(0)])
sage: FanMorphism(m, F, A2).is_surjective()
False
```

```
>>> from sage.all import *
>>> F = Fan([Bl.generating_cone(Integer(0))])
>>> FanMorphism(m, F, A2).is_surjective()
False
```

kernel_fan()

Return the subfan of the domain fan mapped into the origin.

OUTPUT: a *fan*

Note

The lattice of the kernel fan is the `kernel()` sublattice of `self`.

See also

preimage_fan().

EXAMPLES:

```
sage: fan = Fan(rays=[(1,0), (1,1), (0,1)], cones=[(0,1), (1,2)])
sage: fm = FanMorphism(matrix(2, 1, [1,-1]), fan, ToricLattice(1))
sage: fm.kernel_fan()
Rational polyhedral fan in Sublattice <N(1, 1)>
sage: _.rays()
N(1, 1)
in Sublattice <N(1, 1)>
sage: fm.kernel_fan().cones()
((0-d cone of Rational polyhedral fan in Sublattice <N(1, 1)>,),
 (1-d cone of Rational polyhedral fan in Sublattice <N(1, 1)>,))
```

```
>>> from sage.all import *
>>> fan = Fan(rays=[(Integer(1),Integer(0)), (Integer(1),Integer(1)),␣
↪(Integer(0),Integer(1))], cones=[(Integer(0),Integer(1)), (Integer(1),
↪Integer(2))])
>>> fm = FanMorphism(matrix(Integer(2), Integer(1), [Integer(1),-Integer(1)]),
↪ fan, ToricLattice(Integer(1)))
>>> fm.kernel_fan()
Rational polyhedral fan in Sublattice <N(1, 1)>
>>> _.rays()
N(1, 1)
in Sublattice <N(1, 1)>
>>> fm.kernel_fan().cones()
((0-d cone of Rational polyhedral fan in Sublattice <N(1, 1)>,),
 (1-d cone of Rational polyhedral fan in Sublattice <N(1, 1)>,))
```

preimage_cones(*cone*)

Return cones of the domain fan whose *image_cone()* is cone.

INPUT:

- cone – a *cone* equivalent to a cone of the *codomain_fan()* of self.

OUTPUT:

- a tuple of *cones* of the *domain_fan()* of self, sorted by dimension.

See also

preimage_fan().

EXAMPLES:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: quadrant = Fan([quadrant])
sage: quadrant_bl = quadrant.subdivide([(1,1)])
sage: fm = FanMorphism(identity_matrix(2), quadrant_bl, quadrant)
sage: fm.preimage_cones(Cone([(1,0)]))
(1-d cone of Rational polyhedral fan in 2-d lattice N,)
sage: fm.preimage_cones(Cone([(1,0), (0,1)]))
(1-d cone of Rational polyhedral fan in 2-d lattice N,
 2-d cone of Rational polyhedral fan in 2-d lattice N,
 2-d cone of Rational polyhedral fan in 2-d lattice N)
```

```
>>> from sage.all import *
>>> quadrant = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> quadrant = Fan([quadrant])
>>> quadrant_bl = quadrant.subdivide([(Integer(1),Integer(1))])
>>> fm = FanMorphism(identity_matrix(Integer(2)), quadrant_bl, quadrant)
>>> fm.preimage_cones(Cone([(Integer(1),Integer(0))]))
(1-d cone of Rational polyhedral fan in 2-d lattice N,)
>>> fm.preimage_cones(Cone([(Integer(1),Integer(0)), (Integer(0),
↪Integer(1))]))
(1-d cone of Rational polyhedral fan in 2-d lattice N,
 2-d cone of Rational polyhedral fan in 2-d lattice N,
 2-d cone of Rational polyhedral fan in 2-d lattice N)
```

preimage_fan(*cone*)

Return the subfan of the domain fan mapped into cone.

INPUT:

- cone – a *cone* equivalent to a cone of the *codomain_fan()* of self

OUTPUT: a *fan*

Note

The preimage fan of cone consists of all cones of the *domain_fan()* which are mapped into cone, including those that are mapped into its boundary. So this fan is not necessarily generated by *preimage_cones()* of cone.

See also

kernel_fan(), *preimage_cones()*.

EXAMPLES:

```
sage: quadrant_cone = Cone([(1,0), (0,1)])
sage: quadrant_fan = Fan([quadrant_cone])
sage: quadrant_bl = quadrant_fan.subdivide([(1,1)])
sage: fm = FanMorphism(identity_matrix(2),
....:                  quadrant_bl, quadrant_fan)
sage: fm.preimage_fan(Cone([(1,0)])).cones()
((0-d cone of Rational polyhedral fan in 2-d lattice N,),
 (1-d cone of Rational polyhedral fan in 2-d lattice N,))
sage: fm.preimage_fan(quadrant_cone).ngenerating_cones()
2
sage: len(fm.preimage_cones(quadrant_cone))
3
```

```
>>> from sage.all import *
>>> quadrant_cone = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> quadrant_fan = Fan([quadrant_cone])
>>> quadrant_bl = quadrant_fan.subdivide([(Integer(1),Integer(1))])
>>> fm = FanMorphism(identity_matrix(Integer(2)),
...                  quadrant_bl, quadrant_fan)
>>> fm.preimage_fan(Cone([(Integer(1),Integer(0))])).cones()
((0-d cone of Rational polyhedral fan in 2-d lattice N,),
 (1-d cone of Rational polyhedral fan in 2-d lattice N,))
>>> fm.preimage_fan(quadrant_cone).ngenerating_cones()
2
>>> len(fm.preimage_cones(quadrant_cone))
3
```

primitive_preimage_cones(*cone*)

Return the primitive cones of the domain fan corresponding to `cone`.

INPUT:

- `cone` – a *cone* equivalent to a cone of the *codomain_fan()* of `self`

OUTPUT: a *cone*

Let $\phi : \Sigma \to \Sigma'$ be a fan morphism, let $\sigma \in \Sigma$, and let $\sigma' = \phi(\sigma)$. Then σ is a **primitive cone corresponding to** σ' if there is no proper face τ of σ such that $\phi(\tau) = \sigma'$.

Primitive cones play an important role for fibration morphisms.

See also

is_fibration(), *preimage_cones()*, *preimage_fan()*.

EXAMPLES:

Consider a projection of a del Pezzo surface onto the projective line:

```
sage: Sigma = toric_varieties.dP6().fan()                                    #␣
↪needs palp
sage: Sigma.rays()                                                           #␣
↪needs palp
N( 0,  1),
N(-1,  0),
N(-1, -1),
N( 0, -1),
N( 1,  0),
N( 1,  1)
in 2-d lattice N
sage: Sigma_p = toric_varieties.P1().fan()
sage: phi = FanMorphism(matrix([[1], [-1]]), Sigma, Sigma_p)                 #␣
↪needs palp
```

```
>>> from sage.all import *
>>> Sigma = toric_varieties.dP6().fan()                                      #␣
↪needs palp
>>> Sigma.rays()                                                             #␣
↪needs palp
N( 0,  1),
N(-1,  0),
N(-1, -1),
N( 0, -1),
N( 1,  0),
N( 1,  1)
in 2-d lattice N
>>> Sigma_p = toric_varieties.P1().fan()
>>> phi = FanMorphism(matrix([[Integer(1)], [-Integer(1)]]), Sigma, Sigma_p) ␣
↪              # needs palp
```

Under this map, one pair of rays is mapped to the origin, one in the positive direction, and one in the negative one. Also three 2-dimensional cones are mapped in the positive direction and three in the negative one, so there are 5 preimage cones corresponding to either of the rays of the codomain fan `Sigma_p`:

```
sage: len(phi.preimage_cones(Cone([(1,)])))                                  #␣
↪needs palp
5
```

```
>>> from sage.all import *
>>> len(phi.preimage_cones(Cone([(Integer(1),)])))                           ␣
↪     # needs palp
5
```

Yet only rays are primitive:

```
sage: phi.primitive_preimage_cones(Cone([(1,)]))                             #␣
↪needs palp
(1-d cone of Rational polyhedral fan in 2-d lattice N,
 1-d cone of Rational polyhedral fan in 2-d lattice N)
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> phi.primitive_preimage_cones(Cone([(Integer(1),)]))                     ␣
↪     # needs palp
(1-d cone of Rational polyhedral fan in 2-d lattice N,
 1-d cone of Rational polyhedral fan in 2-d lattice N)
```

Since all primitive cones are mapped onto their images bijectively, we get a fibration:

```
sage: phi.is_fibration()                                                    #␣
↪needs palp
True
```

```
>>> from sage.all import *
>>> phi.is_fibration()                                                      #␣
↪needs palp
True
```

But since there are several primitive cones corresponding to the same cone of the codomain fan, this map is not a bundle, even though its index is 1:

```
sage: phi.is_bundle()                                                       #␣
↪needs palp
False
sage: phi.index()                                                           #␣
↪needs palp
1
```

```
>>> from sage.all import *
>>> phi.is_bundle()                                                         #␣
↪needs palp
False
>>> phi.index()                                                             #␣
↪needs palp
1
```

relative_star_generators(*domain_cone*)

Return the relative star generators of `domain_cone`.

INPUT:

- `domain_cone` – a cone of the *domain_fan()* of `self`

OUTPUT:

- `star_generators()` of `domain_cone` viewed as a cone of *preimage_fan()* of *image_cone()* of `domain_cone`.

EXAMPLES:

```
sage: A2 = toric_varieties.A(2).fan()
sage: Bl = A2.subdivide([(1,1)])
sage: f = FanMorphism(identity_matrix(2), Bl, A2)
sage: for c1 in Bl(1):
....:     print(f.relative_star_generators(c1))
(1-d cone of Rational polyhedral fan in 2-d lattice N,)
```

(continues on next page)

(continued from previous page)

```
(1-d cone of Rational polyhedral fan in 2-d lattice N,)
(2-d cone of Rational polyhedral fan in 2-d lattice N,
 2-d cone of Rational polyhedral fan in 2-d lattice N)
```

```
>>> from sage.all import *
>>> A2 = toric_varieties.A(Integer(2)).fan()
>>> Bl = A2.subdivide([(Integer(1),Integer(1))])
>>> f = FanMorphism(identity_matrix(Integer(2)), Bl, A2)
>>> for c1 in Bl(Integer(1)):
...     print(f.relative_star_generators(c1))
(1-d cone of Rational polyhedral fan in 2-d lattice N,)
(1-d cone of Rational polyhedral fan in 2-d lattice N,)
(2-d cone of Rational polyhedral fan in 2-d lattice N,
 2-d cone of Rational polyhedral fan in 2-d lattice N)
```

2.5.7 Point collections

This module was designed as a part of framework for toric varieties (`variety`, `fano_variety`).

AUTHORS:

- Andrey Novoseltsev (2011-04-25): initial version, based on cone module.
- Andrey Novoseltsev (2012-03-06): additions and doctest changes while switching cones to use point collections.

EXAMPLES:

The idea behind *`point collections`* is to have a container for points of the same space that

- behaves like a tuple *without significant performance penalty*:

```
sage: c = Cone([(0,0,1), (1,0,1), (0,1,1), (1,1,1)]).rays()
sage: c[1]
N(1, 0, 1)
sage: for point in c: point
N(0, 0, 1)
N(1, 0, 1)
N(0, 1, 1)
N(1, 1, 1)
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(0),Integer(0),Integer(1)), (Integer(1),Integer(0),
↪Integer(1)), (Integer(0),Integer(1),Integer(1)), (Integer(1),Integer(1),
↪Integer(1))]).rays()
>>> c[Integer(1)]
N(1, 0, 1)
>>> for point in c: point
N(0, 0, 1)
N(1, 0, 1)
N(0, 1, 1)
N(1, 1, 1)
```

- prints in a convenient way and with clear indication of the ambient space:

```
sage: c
N(0, 0, 1),
N(1, 0, 1),
N(0, 1, 1),
N(1, 1, 1)
in 3-d lattice N
```

```
>>> from sage.all import *
>>> c
N(0, 0, 1),
N(1, 0, 1),
N(0, 1, 1),
N(1, 1, 1)
in 3-d lattice N
```

- allows (cached) access to alternative representations:

```
sage: c.set()
frozenset({N(0, 0, 1), N(0, 1, 1), N(1, 0, 1), N(1, 1, 1)})
```

```
>>> from sage.all import *
>>> c.set()
frozenset({N(0, 0, 1), N(0, 1, 1), N(1, 0, 1), N(1, 1, 1)})
```

- allows introduction of additional methods:

```
sage: c.basis()
N(0, 0, 1),
N(1, 0, 1),
N(0, 1, 1)
in 3-d lattice N
```

```
>>> from sage.all import *
>>> c.basis()
N(0, 0, 1),
N(1, 0, 1),
N(0, 1, 1)
in 3-d lattice N
```

Examples of natural point collections include ray and line generators of cones, vertices and points of polytopes, normals to facets, their subcollections, etc.

Using this class for all of the above cases allows for unified interface *and* cache sharing. Suppose that Δ is a reflexive polytope. Then the same point collection can be linked as

1. vertices of Δ;
2. facet normals of its polar Δ°;
3. ray generators of the face fan of Δ;
4. ray generators of the normal fan of Δ.

If all these objects are in use and, say, a matrix representation was computed for one of them, it becomes available to all others as well, eliminating the need to spend time and memory four times.

class sage.geometry.point_collection.**PointCollection**

Bases: SageObject

Create a point collection.

> **Warning**
>
> No correctness check or normalization is performed on the input data. This class is designed for internal operations and you probably should not use it directly.

Point collections are immutable, but cache most of the returned values.

INPUT:

- `points` – an iterable structure of immutable elements of `module`, if `points` are already accessible to you as a `tuple`, it is preferable to use it for speed and memory consumption reasons;
- `module` – an ambient module for `points`. If `None` (the default), it will be determined as `parent()` of the first point. Of course, this cannot be done if there are no points, so in this case you must give an appropriate `module` directly.

OUTPUT:

- a point collection.

basis()

Return a linearly independent subset of points of `self`.

OUTPUT:

- a *point collection* giving a random (but fixed) choice of an **R**-basis for the vector space spanned by the points of `self`.

EXAMPLES:

```
sage: c = Cone([(0,0,1), (1,0,1), (0,1,1), (1,1,1)]).rays()
sage: c.basis()
N(0, 0, 1),
N(1, 0, 1),
N(0, 1, 1)
in 3-d lattice N
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(0),Integer(0),Integer(1)), (Integer(1),Integer(0),
↪Integer(1)), (Integer(0),Integer(1),Integer(1)), (Integer(1),Integer(1),
↪Integer(1))]).rays()
>>> c.basis()
N(0, 0, 1),
N(1, 0, 1),
N(0, 1, 1)
in 3-d lattice N
```

Calling this method twice will always return *exactly the same* point collection:

```
sage: c.basis().basis() is c.basis()
True
```

```
>>> from sage.all import *
>>> c.basis().basis() is c.basis()
True
```

cardinality()

Return the number of points in `self`.

OUTPUT: integer

EXAMPLES:

```
sage: c = Cone([(0,0,1), (1,0,1), (0,1,1), (1,1,1)]).rays()
sage: c.cardinality()
4
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(0),Integer(0),Integer(1)), (Integer(1),Integer(0),
↪Integer(1)), (Integer(0),Integer(1),Integer(1)), (Integer(1),Integer(1),
↪Integer(1))]).rays()
>>> c.cardinality()
4
```

cartesian_product(*other*, *module=None*)

Return the Cartesian product of `self` with `other`.

INPUT:

- `other` – a *point collection*;
- `module` – (optional) the ambient module for the result. By default, the direct sum of the ambient modules of `self` and `other` is constructed.

OUTPUT: a *point collection*

EXAMPLES:

```
sage: c = Cone([(0,0,1), (1,1,1)]).rays()
sage: c.cartesian_product(c)
N+N(0, 0, 1, 0, 0, 1),
N+N(1, 1, 1, 0, 0, 1),
N+N(0, 0, 1, 1, 1, 1),
N+N(1, 1, 1, 1, 1, 1)
in 6-d lattice N+N
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(0),Integer(0),Integer(1)), (Integer(1),Integer(1),
↪Integer(1))]).rays()
>>> c.cartesian_product(c)
N+N(0, 0, 1, 0, 0, 1),
N+N(1, 1, 1, 0, 0, 1),
N+N(0, 0, 1, 1, 1, 1),
N+N(1, 1, 1, 1, 1, 1)
in 6-d lattice N+N
```

column_matrix()

Return a matrix whose columns are points of `self`.

OUTPUT: a `matrix`

EXAMPLES:

```
sage: c = Cone([(0,0,1), (1,0,1), (0,1,1), (1,1,1)]).rays()
sage: c.column_matrix()
[0 1 0 1]
[0 0 1 1]
[1 1 1 1]
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(0),Integer(0),Integer(1)), (Integer(1),Integer(0),
↪Integer(1)), (Integer(0),Integer(1),Integer(1)), (Integer(1),Integer(1),
↪Integer(1))]).rays()
>>> c.column_matrix()
[0 1 0 1]
[0 0 1 1]
[1 1 1 1]
```

dim()

Return the dimension of the space spanned by points of `self`.

> **Note**
>
> You can use either *dim()* or *dimension()*.

OUTPUT: integer

EXAMPLES:

```
sage: c = Cone([(0,0,1), (1,1,1)]).rays()
sage: c.dimension()
2
sage: c.dim()
2
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(0),Integer(0),Integer(1)), (Integer(1),Integer(1),
↪Integer(1))]).rays()
>>> c.dimension()
2
>>> c.dim()
2
```

dimension()

Return the dimension of the space spanned by points of `self`.

> **Note**
>
> You can use either *dim()* or *dimension()*.

OUTPUT: integer

EXAMPLES:

```
sage: c = Cone([(0,0,1), (1,1,1)]).rays()
sage: c.dimension()
2
sage: c.dim()
2
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(0),Integer(0),Integer(1)), (Integer(1),Integer(1),
↪Integer(1))]).rays()
>>> c.dimension()
2
>>> c.dim()
2
```

dual_module()

Return the dual of the ambient module of `self`.

OUTPUT:

- a `module`. If possible (that is, if the ambient *module()* M of `self` has a `dual()` method), the dual module is returned. Otherwise, R^n is returned, where n is the dimension of M and R is its base ring.

EXAMPLES:

```
sage: c = Cone([(0,0,1), (1,0,1), (0,1,1), (1,1,1)]).rays()
sage: c.dual_module()
3-d lattice M
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(0),Integer(0),Integer(1)), (Integer(1),Integer(0),
↪Integer(1)), (Integer(0),Integer(1),Integer(1)), (Integer(1),Integer(1),
↪Integer(1))]).rays()
>>> c.dual_module()
3-d lattice M
```

index(**args*)

Return the index of the first occurrence of `point` in `self`.

INPUT:

- `point` – a point of `self`
- `start` – (optional) an integer, if given, the search will start at this position
- `stop` – (optional) an integer, if given, the search will stop at this position

OUTPUT: an integer if `point` is in `self[start:stop]`, otherwise a `ValueError` exception is raised

EXAMPLES:

```
sage: c = Cone([(0,0,1), (1,0,1), (0,1,1), (1,1,1)]).rays()
sage: c.index((0,1,1))
Traceback (most recent call last):
...
ValueError: tuple.index(x): x not in tuple
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(0),Integer(0),Integer(1)), (Integer(1),Integer(0),
↪Integer(1)), (Integer(0),Integer(1),Integer(1)), (Integer(1),Integer(1),
↪Integer(1))]).rays()
>>> c.index((Integer(0),Integer(1),Integer(1)))
Traceback (most recent call last):
...
ValueError: tuple.index(x): x not in tuple
```

Note that this was not a mistake: the *tuple* `(0,1,1)` is *not* a point of `c`! We need to pass actual element of the ambient module of `c` to get their indices:

```
sage: N = c.module()
sage: c.index(N(0,1,1))
2
sage: c[2]
N(0, 1, 1)
```

```
>>> from sage.all import *
>>> N = c.module()
>>> c.index(N(Integer(0),Integer(1),Integer(1)))
2
>>> c[Integer(2)]
N(0, 1, 1)
```

matrix()

Return a matrix whose rows are points of `self`.

OUTPUT: a `matrix`

EXAMPLES:

```
sage: c = Cone([(0,0,1), (1,0,1), (0,1,1), (1,1,1)]).rays()
sage: c.matrix()
[0 0 1]
[1 0 1]
[0 1 1]
[1 1 1]
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(0),Integer(0),Integer(1)), (Integer(1),Integer(0),
↪Integer(1)), (Integer(0),Integer(1),Integer(1)), (Integer(1),Integer(1),
↪Integer(1))]).rays()
>>> c.matrix()
[0 0 1]
[1 0 1]
[0 1 1]
[1 1 1]
```

module()

Return the ambient module of `self`.

OUTPUT: a `module`

EXAMPLES:

```
sage: c = Cone([(0,0,1), (1,0,1), (0,1,1), (1,1,1)]).rays()
sage: c.module()
3-d lattice N
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(0),Integer(0),Integer(1)), (Integer(1),Integer(0),
↪Integer(1)), (Integer(0),Integer(1),Integer(1)), (Integer(1),Integer(1),
↪Integer(1))]).rays()
>>> c.module()
3-d lattice N
```

static output_format (*format=None*)

Return or set the output format for **ALL** point collections.

INPUT:

- **format – (optional) if given, must be one of the strings**
  - "default" – output one point per line with vertical alignment of coordinates in text mode, same as "tuple" for LaTeX;
  - "tuple" – output `tuple(self)` with lattice information;
  - "matrix" – output `matrix()` with lattice information;
  - "column matrix" – output `column_matrix()` with lattice information;
  - "separated column matrix" – same as "column matrix" for text mode, for LaTeX separate columns by lines (not shown by jsMath).

OUTPUT:

- a string with the current format (only if `format` was omitted).

This function affects both regular and LaTeX output.

EXAMPLES:

```
sage: c = Cone([(0,0,1), (1,0,1), (0,1,1), (1,1,1)]).rays()
sage: c
N(0, 0, 1),
N(1, 0, 1),
N(0, 1, 1),
N(1, 1, 1)
in 3-d lattice N
sage: c.output_format()
'default'
sage: c.output_format("tuple")
sage: c
(N(0, 0, 1), N(1, 0, 1), N(0, 1, 1), N(1, 1, 1))
in 3-d lattice N
sage: c.output_format("matrix")
sage: c
[0 0 1]
[1 0 1]
[0 1 1]
[1 1 1]
in 3-d lattice N
```

(continues on next page)

(continued from previous page)

```
sage: c.output_format("column matrix")
sage: c
[0 1 0 1]
[0 0 1 1]
[1 1 1 1]
in 3-d lattice N
sage: c.output_format("separated column matrix")
sage: c
[0 1 0 1]
[0 0 1 1]
[1 1 1 1]
in 3-d lattice N
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(0),Integer(0),Integer(1)), (Integer(1),Integer(0),
↪Integer(1)), (Integer(0),Integer(1),Integer(1)), (Integer(1),Integer(1),
↪Integer(1))]).rays()
>>> c
N(0, 0, 1),
N(1, 0, 1),
N(0, 1, 1),
N(1, 1, 1)
in 3-d lattice N
>>> c.output_format()
'default'
>>> c.output_format("tuple")
>>> c
(N(0, 0, 1), N(1, 0, 1), N(0, 1, 1), N(1, 1, 1))
in 3-d lattice N
>>> c.output_format("matrix")
>>> c
[0 0 1]
[1 0 1]
[0 1 1]
[1 1 1]
in 3-d lattice N
>>> c.output_format("column matrix")
>>> c
[0 1 0 1]
[0 0 1 1]
[1 1 1 1]
in 3-d lattice N
>>> c.output_format("separated column matrix")
>>> c
[0 1 0 1]
[0 0 1 1]
[1 1 1 1]
in 3-d lattice N
```

Note that the last two outputs are identical, separators are only inserted in the LaTeX mode:

```
sage: latex(c)
```

(continues on next page)

(continued from previous page)

```
\left(\begin{array}{r|r|r|r}
0 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 \\
1 & 1 & 1 & 1
\end{array}\right)_{N}
```

```
>>> from sage.all import *
>>> latex(c)
\left(\begin{array}{r|r|r|r}
0 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 \\
1 & 1 & 1 & 1
\end{array}\right)_{N}
```

Since this is a static method, you can call it for the class directly:

```
sage: from sage.geometry.point_collection import PointCollection
sage: PointCollection.output_format("default")
sage: c
N(0, 0, 1),
N(1, 0, 1),
N(0, 1, 1),
N(1, 1, 1)
in 3-d lattice N
```

```
>>> from sage.all import *
>>> from sage.geometry.point_collection import PointCollection
>>> PointCollection.output_format("default")
>>> c
N(0, 0, 1),
N(1, 0, 1),
N(0, 1, 1),
N(1, 1, 1)
in 3-d lattice N
```

set()

Return points of `self` as a `frozenset`.

OUTPUT: a `frozenset`

EXAMPLES:

```
sage: c = Cone([(0,0,1), (1,0,1), (0,1,1), (1,1,1)]).rays()
sage: c.set()
frozenset({N(0, 0, 1), N(0, 1, 1), N(1, 0, 1), N(1, 1, 1)})
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(0),Integer(0),Integer(1)), (Integer(1),Integer(0),
↪Integer(1)), (Integer(0),Integer(1),Integer(1)), (Integer(1),Integer(1),
↪Integer(1))]).rays()
>>> c.set()
frozenset({N(0, 0, 1), N(0, 1, 1), N(1, 0, 1), N(1, 1, 1)})
```

write_for_palp(*f*)

Write `self` into an open file `f` in PALP format.

INPUT:

- `f` – a file opened for writing

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: from io import StringIO
sage: f = StringIO()
sage: o.vertices().write_for_palp(f)
sage: print(f.getvalue())
6 3
1 0 0
0 1 0
0 0 1
-1 0 0
0 -1 0
0 0 -1
```

```
>>> from sage.all import *
>>> o = lattice_polytope.cross_polytope(Integer(3))
>>> from io import StringIO
>>> f = StringIO()
>>> o.vertices().write_for_palp(f)
>>> print(f.getvalue())
6 3
1 0 0
0 1 0
0 0 1
-1 0 0
0 -1 0
0 0 -1
```

sage.geometry.point_collection.**is_PointCollection**(*x*)

Check if `x` is a *point collection*.

INPUT:

- `x` – anything

OUTPUT: `True` if `x` is a point collection and `False` otherwise

EXAMPLES:

```
sage: from sage.geometry.point_collection import PointCollection
sage: isinstance(1, PointCollection)
False
sage: c = Cone([(0,0,1), (1,0,1), (0,1,1), (1,1,1)])
sage: isinstance(c.rays(), PointCollection)
True
```

```
>>> from sage.all import *
>>> from sage.geometry.point_collection import PointCollection
```

(continues on next page)

(continued from previous page)

```
>>> isinstance(Integer(1), PointCollection)
False
>>> c = Cone([(Integer(0),Integer(0),Integer(1)), (Integer(1),Integer(0),
↪Integer(1)), (Integer(0),Integer(1),Integer(1)), (Integer(1),Integer(1),
↪Integer(1))])
>>> isinstance(c.rays(), PointCollection)
True
```

sage.geometry.point_collection.**read_palp_point_collection**(*f*, *lattice=None*, *permutation=False*)

Read and return a point collection from an opened file.

Data must be in PALP format:

- the first input line starts with two integers m and n, the number of points and the number of components of each;
- the rest of the first line may contain a permutation;
- the next m lines contain n numbers each.

Note

If $m < n$, it is assumed (for compatibility with PALP) that the matrix is transposed, i.e. that each column is a point.

INPUT:

- `f` – an opened file with PALP output
- `lattice` – the lattice for points. If not given, the *toric lattice* M of dimension n will be used.
- `permutation` – boolean (default: `False`); if `True`, try to retrieve the permutation. This parameter makes sense only when PALP computed the normal form of a lattice polytope.

OUTPUT:

- a *point collection*, optionally followed by a permutation. `None` if EOF is reached.

EXAMPLES:

```
sage: data = "3 2 regular\n1 2\n3 4\n5 6\n2 3 transposed\n1 2 3\n4 5 6"
sage: print(data)
3 2 regular
1 2
3 4
5 6
2 3 transposed
1 2 3
4 5 6
sage: from io import StringIO
sage: f = StringIO(data)
sage: from sage.geometry.point_collection \
....:     import read_palp_point_collection
sage: read_palp_point_collection(f)
M(1, 2),
M(3, 4),
```

(continues on next page)

(continued from previous page)

```
M(5, 6)
in 2-d lattice M
sage: read_palp_point_collection(f)
M(1, 4),
M(2, 5),
M(3, 6)
in 2-d lattice M
sage: read_palp_point_collection(f) is None
True
```

```
>>> from sage.all import *
>>> data = "3 2 regular\n1 2\n3 4\n5 6\n2 3 transposed\n1 2 3\n4 5 6"
>>> print(data)
3 2 regular
1 2
3 4
5 6
2 3 transposed
1 2 3
4 5 6
>>> from io import StringIO
>>> f = StringIO(data)
>>> from sage.geometry.point_collection    import read_palp_point_collection
>>> read_palp_point_collection(f)
M(1, 2),
M(3, 4),
M(5, 6)
in 2-d lattice M
>>> read_palp_point_collection(f)
M(1, 4),
M(2, 5),
M(3, 6)
in 2-d lattice M
>>> read_palp_point_collection(f) is None
True
```

2.5.8 Toric plotter

This module provides a helper class `ToricPlotter` for producing plots of objects related to toric geometry. Default plotting objects can be adjusted using `options()` and reset using `reset_options()`.

AUTHORS:

- Andrey Novoseltsev (2010-10-03): initial version, using some code bits by Volker Braun.

EXAMPLES:

In most cases, this module is used indirectly, e.g.

```
sage: fan = toric_varieties.dP6().fan()                                       #␣
↪needs palp sage.graphs
sage: fan.plot()                                                              #␣
↪needs palp sage.graphs sage.plot
Graphics object consisting of 31 graphics primitives
```

```
>>> from sage.all import *
>>> fan = toric_varieties.dP6().fan()                                    #␣
↪needs palp sage.graphs
>>> fan.plot()                                                           #␣
↪needs palp sage.graphs sage.plot
Graphics object consisting of 31 graphics primitives
```

You may change default plotting options as follows:

```
sage: toric_plotter.options("show_rays")
True
sage: toric_plotter.options(show_rays=False)
sage: toric_plotter.options("show_rays")
False
sage: fan.plot()                                                          #␣
↪needs palp sage.graphs sage.plot
Graphics object consisting of 19 graphics primitives
sage: toric_plotter.reset_options()
sage: toric_plotter.options("show_rays")
True
sage: fan.plot()                                                          #␣
↪needs palp sage.graphs sage.plot
Graphics object consisting of 31 graphics primitives
```

```
>>> from sage.all import *
>>> toric_plotter.options("show_rays")
True
>>> toric_plotter.options(show_rays=False)
>>> toric_plotter.options("show_rays")
False
>>> fan.plot()                                                           #␣
↪needs palp sage.graphs sage.plot
Graphics object consisting of 19 graphics primitives
>>> toric_plotter.reset_options()
>>> toric_plotter.options("show_rays")
True
>>> fan.plot()                                                           #␣
↪needs palp sage.graphs sage.plot
Graphics object consisting of 31 graphics primitives
```

class `sage.geometry.toric_plotter.`**`ToricPlotter`**(*all_options*, *dimension*, *generators=None*)

Bases: `SageObject`

Create a toric plotter.

INPUT:

- `all_options` – a `dictionary`, containing any of the options related to toric objects (see *`options()`*) and any other options that will be passed to lower level plotting functions
- `dimension` – integer (1, 2, or 3); dimension of toric objects to be plotted
- `generators` – (optional) a list of ray generators; see examples for a detailed explanation of this argument

OUTPUT: a toric plotter

EXAMPLES:

In most cases there is no need to create and use *ToricPlotter* directly. Instead, use plotting method of the object which you want to plot, e.g.

```
sage: fan = toric_varieties.dP6().fan()                                    #␣
↪needs palp sage.graphs
sage: fan.plot()                                                           #␣
↪needs palp sage.graphs sage.plot
Graphics object consisting of 31 graphics primitives
sage: print(fan.plot())                                                    #␣
↪needs palp sage.graphs sage.plot
Graphics object consisting of 31 graphics primitives
```

```
>>> from sage.all import *
>>> fan = toric_varieties.dP6().fan()                                      #␣
↪needs palp sage.graphs
>>> fan.plot()                                                             #␣
↪needs palp sage.graphs sage.plot
Graphics object consisting of 31 graphics primitives
>>> print(fan.plot())                                                      #␣
↪needs palp sage.graphs sage.plot
Graphics object consisting of 31 graphics primitives
```

If you do want to create your own plotting function for some toric structure, the anticipated usage of toric plotters is the following:

- collect all necessary options in a dictionary;
- pass these options and `dimension` to *ToricPlotter*;
- call *include_points()* on ray generators and any other points that you want to be present on the plot (it will try to set appropriate cut-off bounds);
- call *adjust_options()* to choose "nice" default values for all options that were not set yet and ensure consistency of rectangular and spherical cut-off bounds;
- call *set_rays()* on ray generators to scale them to the cut-off bounds of the plot;
- call appropriate `plot_*` functions to actually construct the plot.

For example, the plot from the previous example can be obtained as follows:

```
sage: # needs palp sage.graphs sage.plot
sage: from sage.geometry.toric_plotter import ToricPlotter
sage: options = dict()  # use default for everything
sage: tp = ToricPlotter(options, fan.lattice().degree())
sage: tp.include_points(fan.rays())
sage: tp.adjust_options()
sage: tp.set_rays(fan.rays())
sage: result = tp.plot_lattice()
sage: result += tp.plot_rays()
sage: result += tp.plot_generators()
sage: result += tp.plot_walls(fan(2))
sage: result
Graphics object consisting of 31 graphics primitives
```

```
>>> from sage.all import *
>>> # needs palp sage.graphs sage.plot
```

(continues on next page)

(continued from previous page)

```
>>> from sage.geometry.toric_plotter import ToricPlotter
>>> options = dict()  # use default for everything
>>> tp = ToricPlotter(options, fan.lattice().degree())
>>> tp.include_points(fan.rays())
>>> tp.adjust_options()
>>> tp.set_rays(fan.rays())
>>> result = tp.plot_lattice()
>>> result += tp.plot_rays()
>>> result += tp.plot_generators()
>>> result += tp.plot_walls(fan(Integer(2)))
>>> result
Graphics object consisting of 31 graphics primitives
```

In most situations it is only necessary to include generators of rays, in this case they can be passed to the constructor as an optional argument. In the example above, the toric plotter can be completely set up using

```
sage: tp = ToricPlotter(options, fan.lattice().degree(), fan.rays())              #␣
↪needs palp sage.graphs sage.plot
```

```
>>> from sage.all import *
>>> tp = ToricPlotter(options, fan.lattice().degree(), fan.rays())              #␣
↪needs palp sage.graphs sage.plot
```

All options are exposed as attributes of toric plotters and can be modified after constructions, however you will have to manually call `adjust_options()` and `set_rays()` again if you decide to change the plotting mode and/or cut-off bounds. Otherwise plots maybe invalid.

adjust_options()

Adjust plotting options.

This function determines appropriate default values for those options, that were not specified by the user, based on the other options. See `ToricPlotter` for a detailed example.

OUTPUT: none

include_points(*points*, *force=False*)

Try to include `points` into the bounding box of `self`.

INPUT:

- `points` – list of points
- `force` – boolean (default: `False`); by default, only bounds that were not set before will be chosen to include `points`. Use `force=True` if you don't mind increasing existing bounding box.

OUTPUT: none

EXAMPLES:

```
sage: from sage.geometry.toric_plotter import ToricPlotter
sage: tp = ToricPlotter(dict(), 2)
sage: print(tp.radius)
None
sage: tp.include_points([(3, 4)])
sage: print(tp.radius)
5.5...
```

(continues on next page)

(continued from previous page)

```
sage: tp.include_points([(5, 12)])
sage: print(tp.radius)
5.5...
sage: tp.include_points([(5, 12)], force=True)
sage: print(tp.radius)
13.5...
```

```
>>> from sage.all import *
>>> from sage.geometry.toric_plotter import ToricPlotter
>>> tp = ToricPlotter(dict(), Integer(2))
>>> print(tp.radius)
None
>>> tp.include_points([(Integer(3), Integer(4))])
>>> print(tp.radius)
5.5...
>>> tp.include_points([(Integer(5), Integer(12))])
>>> print(tp.radius)
5.5...
>>> tp.include_points([(Integer(5), Integer(12))], force=True)
>>> print(tp.radius)
13.5...
```

plot_generators()

Plot ray generators.

Ray generators must be specified during construction or using *set_rays()* before calling this method.

OUTPUT: a plot

EXAMPLES:

```
sage: from sage.geometry.toric_plotter import ToricPlotter
sage: tp = ToricPlotter(dict(), 2, [(3,4)])
sage: tp.plot_generators()                                              #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> from sage.geometry.toric_plotter import ToricPlotter
>>> tp = ToricPlotter(dict(), Integer(2), [(Integer(3),Integer(4))])
>>> tp.plot_generators()                                                #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

plot_labels(*labels*, *positions*)

Plot `labels` at specified `positions`.

INPUT:

- `labels` – string or list of strings
- `positions` – list of points

OUTPUT: a plot

EXAMPLES:

```
sage: from sage.geometry.toric_plotter import ToricPlotter
sage: tp = ToricPlotter(dict(), 2)
sage: tp.plot_labels("u", [(1.5,0)])                                           #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> from sage.geometry.toric_plotter import ToricPlotter
>>> tp = ToricPlotter(dict(), Integer(2))
>>> tp.plot_labels("u", [(RealNumber('1.5'),Integer(0))])                      ␣
↪                    # needs sage.plot
Graphics object consisting of 1 graphics primitive
```

plot_lattice()

Plot the lattice (i.e. its points in the cut-off bounds of `self`).

OUTPUT: a plot

EXAMPLES:

```
sage: from sage.geometry.toric_plotter import ToricPlotter
sage: tp = ToricPlotter(dict(), 2)
sage: tp.adjust_options()
sage: tp.plot_lattice()                                                         #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> from sage.geometry.toric_plotter import ToricPlotter
>>> tp = ToricPlotter(dict(), Integer(2))
>>> tp.adjust_options()
>>> tp.plot_lattice()                                                         #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

plot_points(*points*)

Plot given `points`.

INPUT:

- `points` – list of points

OUTPUT: a plot

EXAMPLES:

```
sage: from sage.geometry.toric_plotter import ToricPlotter
sage: tp = ToricPlotter(dict(), 2)
sage: tp.adjust_options()
sage: tp.plot_points([(1,0), (0,1)])                                            #␣
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> from sage.geometry.toric_plotter import ToricPlotter
```

(continues on next page)

(continued from previous page)

```
>>> tp = ToricPlotter(dict(), Integer(2))
>>> tp.adjust_options()
>>> tp.plot_points([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
↪                                # needs sage.plot
Graphics object consisting of 1 graphics primitive
```

plot_ray_labels()

Plot ray labels.

Usually ray labels are plotted together with rays, but in some cases it is desirable to output them separately.

Ray generators must be specified during construction or using *set_rays()* before calling this method.

OUTPUT: a plot

EXAMPLES:

```
sage: from sage.geometry.toric_plotter import ToricPlotter
sage: tp = ToricPlotter(dict(), 2, [(3,4)])
sage: tp.plot_ray_labels()                                                    #
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

```
>>> from sage.all import *
>>> from sage.geometry.toric_plotter import ToricPlotter
>>> tp = ToricPlotter(dict(), Integer(2), [(Integer(3),Integer(4))])
>>> tp.plot_ray_labels()                                                    #
↪needs sage.plot
Graphics object consisting of 1 graphics primitive
```

plot_rays()

Plot rays and their labels.

Ray generators must be specified during construction or using *set_rays()* before calling this method.

OUTPUT: a plot

EXAMPLES:

```
sage: from sage.geometry.toric_plotter import ToricPlotter
sage: tp = ToricPlotter(dict(), 2, [(3,4)])
sage: tp.plot_rays()                                                          #
↪needs sage.plot
Graphics object consisting of 2 graphics primitives
```

```
>>> from sage.all import *
>>> from sage.geometry.toric_plotter import ToricPlotter
>>> tp = ToricPlotter(dict(), Integer(2), [(Integer(3),Integer(4))])
>>> tp.plot_rays()                                                          #
↪needs sage.plot
Graphics object consisting of 2 graphics primitives
```

plot_walls(*walls*)

Plot `walls`, i.e. 2-d cones, and their labels.

Ray generators must be specified during construction or using `set_rays()` before calling this method and these specified ray generators will be used in conjunction with `ambient_ray_indices()` of `walls`.

INPUT:

- `walls` – list of 2-d cones

OUTPUT: a plot

EXAMPLES:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: from sage.geometry.toric_plotter import ToricPlotter
sage: tp = ToricPlotter(dict(), 2, quadrant.rays())
sage: tp.plot_walls([quadrant])                                      #␣
↪needs sage.plot
Graphics object consisting of 2 graphics primitives
```

```
>>> from sage.all import *
>>> quadrant = Cone([(Integer(1),Integer(0)), (Integer(0),Integer(1))])
>>> from sage.geometry.toric_plotter import ToricPlotter
>>> tp = ToricPlotter(dict(), Integer(2), quadrant.rays())
>>> tp.plot_walls([quadrant])                                      #␣
↪needs sage.plot
Graphics object consisting of 2 graphics primitives
```

Let's also check that the truncating polyhedron is functioning correctly:

```
sage: tp = ToricPlotter({"mode": "box"}, 2, quadrant.rays())
sage: tp.plot_walls([quadrant])                                      #␣
↪needs sage.plot
Graphics object consisting of 2 graphics primitives
```

```
>>> from sage.all import *
>>> tp = ToricPlotter({"mode": "box"}, Integer(2), quadrant.rays())
>>> tp.plot_walls([quadrant])                                      #␣
↪needs sage.plot
Graphics object consisting of 2 graphics primitives
```

set_rays(*generators*)

Set up rays and their `generators` to be used by plotting functions.

As an alternative to using this method, you can pass `generators` to `ToricPlotter` constructor.

INPUT:

- `generators` – list of primitive nonzero ray generators

OUTPUT: none

EXAMPLES:

```
sage: from sage.geometry.toric_plotter import ToricPlotter
sage: tp = ToricPlotter(dict(), 2)
sage: tp.adjust_options()
sage: tp.plot_rays()                                                 #␣
↪needs sage.plot
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
AttributeError: 'ToricPlotter' object has no attribute 'rays'...
sage: tp.set_rays([(0,1)])
sage: tp.plot_rays()                                                    #␣
↪needs sage.plot
Graphics object consisting of 2 graphics primitives
```

```
>>> from sage.all import *
>>> from sage.geometry.toric_plotter import ToricPlotter
>>> tp = ToricPlotter(dict(), Integer(2))
>>> tp.adjust_options()
>>> tp.plot_rays()                                                      #␣
↪needs sage.plot
Traceback (most recent call last):
...
AttributeError: 'ToricPlotter' object has no attribute 'rays'...
>>> tp.set_rays([(Integer(0),Integer(1))])
>>> tp.plot_rays()                                                      #␣
↪needs sage.plot
Graphics object consisting of 2 graphics primitives
```

sage.geometry.toric_plotter.**color_list**(*color*, *n*)

Normalize a list of n colors.

INPUT:

- color – anything specifying a Color, a list of such specifications, or the string "rainbow";
- n – integer

OUTPUT: list of n colors

If color specified a single color, it is repeated n times. If it was a list of n colors, it is returned without changes. If it was "rainbow", the rainbow of n colors is returned.

EXAMPLES:

```
sage: # needs sage.plot
sage: from sage.geometry.toric_plotter import color_list
sage: color_list("grey", 1)
[RGB color (0.5019607843137255, 0.5019607843137255, 0.5019607843137255)]
sage: len(color_list("grey", 3))
3
sage: L = color_list("rainbow", 3)
sage: L
[RGB color (1.0, 0.0, 0.0),
 RGB color (0.0, 1.0, 0.0),
 RGB color (0.0, 0.0, 1.0)]
sage: color_list(L, 3)
[RGB color (1.0, 0.0, 0.0),
 RGB color (0.0, 1.0, 0.0),
 RGB color (0.0, 0.0, 1.0)]
sage: color_list(L, 4)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ValueError: expected 4 colors, got 3!
```

```
>>> from sage.all import *
>>> # needs sage.plot
>>> from sage.geometry.toric_plotter import color_list
>>> color_list("grey", Integer(1))
[RGB color (0.5019607843137255, 0.5019607843137255, 0.5019607843137255)]
>>> len(color_list("grey", Integer(3)))
3
>>> L = color_list("rainbow", Integer(3))
>>> L
[RGB color (1.0, 0.0, 0.0),
 RGB color (0.0, 1.0, 0.0),
 RGB color (0.0, 0.0, 1.0)]
>>> color_list(L, Integer(3))
[RGB color (1.0, 0.0, 0.0),
 RGB color (0.0, 1.0, 0.0),
 RGB color (0.0, 0.0, 1.0)]
>>> color_list(L, Integer(4))
Traceback (most recent call last):
...
ValueError: expected 4 colors, got 3!
```

`sage.geometry.toric_plotter.`**`label_list`**(*label*, *n*, *math_mode*, *index_set=None*)

Normalize a list of `n` labels.

INPUT:

- `label` – `None`, a string, or a list of string
- `n` – integer
- `math_mode` – boolean; if `True`, will produce LaTeX expressions for labels
- `index_set` – list of integers (default: `range(n)`) that will be used as subscripts for labels

OUTPUT: list of `n` labels

If `label` was a list of `n` entries, it is returned without changes. If `label` is `None`, a list of `n` `None`'s is returned. If `label` is a string, a list of strings of the form `$label_{i}$` is returned, where i ranges over `index_set`. (If `math_mode=False`, the form "label_i" is used instead.) If `n=1`, there is no subscript added, unless `index_set` was specified explicitly.

EXAMPLES:

```
sage: from sage.geometry.toric_plotter import label_list
sage: label_list("u", 3, False)
['u_0', 'u_1', 'u_2']
sage: label_list("u", 3, True)
['$u_{0}$', '$u_{1}$', '$u_{2}$']
sage: label_list("u", 1, True)
['$u$']
```

```
>>> from sage.all import *
>>> from sage.geometry.toric_plotter import label_list
>>> label_list("u", Integer(3), False)
['u_0', 'u_1', 'u_2']
>>> label_list("u", Integer(3), True)
['$u_{0}$', '$u_{1}$', '$u_{2}$']
>>> label_list("u", Integer(1), True)
['$u$']
```

`sage.geometry.toric_plotter.`**`options`**(*option=None*, ***kwds*)

Get or set options for plots of toric geometry objects.

Note

This function provides access to global default options. Any of these options can be overridden by passing them directly to plotting functions. See also *`reset_options()`*.

INPUT:

- None;

OR:

- `option` – string, name of the option whose value you wish to get;

OR:

- keyword arguments specifying new values for one or more options.

OUTPUT:

- if there was no input, the dictionary of current options for toric plots;
- if `option` argument was given, the current value of `option`;
- if other keyword arguments were given, none.

Name Conventions

To clearly distinguish parts of toric plots, in options and methods we use the following name conventions:

Generator
: A primitive integral vector generating a 1-dimensional cone, plotted as an arrow from the origin (or a line, if the head of the arrow is beyond cut-off bounds for the plot).

Ray
: A 1-dimensional cone, plotted as a line from the origin to the cut-off bounds for the plot.

Wall
: A 2-dimensional cone, plotted as a region between rays (in the above sense). Its exact shape depends on the plotting mode (see below).

Chamber
: A 3-dimensional cone, plotting is not implemented yet.

Plotting Modes

A plotting mode mostly determines the shape of the cut-off region (which is always relevant for toric plots except for trivial objects consisting of the origin only). The following options are available:

Box
The cut-off region is a box with edges parallel to coordinate axes.

Generators
The cut-off region is determined by primitive integral generators of rays. Note that this notion is well-defined only for rays and walls, in particular you should plot the lattice on your own (`plot_lattice()` will use box mode which is likely to be unsuitable). While this method may not be suitable for general fans, it is quite natural for fans of `CPR-Fano toric varieties. <sage.schemes.toric.fano_variety.CPRFanoToricVariety_field`

Round
The cut-off regions is a sphere centered at the origin.

Available Options

Default values for the following options can be set using this function:

- `mode` – "box", "generators", or "round", see above for descriptions;
- `show_lattice` – boolean, whether to show lattice points in the cut-off region or not;
- `show_rays` – boolean, whether to show rays or not;
- `show_generators` – boolean, whether to show rays or not;
- `show_walls` – boolean, whether to show rays or not;
- `generator_color` – a color for generators;
- `label_color` – a color for labels;
- `point_color` – a color for lattice points;
- `ray_color` – a color for rays, a list of colors (one for each ray), or the string "rainbow";
- `wall_color` – a color for walls, a list of colors (one for each wall), or the string "rainbow";
- `wall_alpha` – a number between 0 and 1, the alpha-value for walls (determining their transparency);
- `point_size` – integer; the size of lattice points
- `ray_thickness` – integer; the thickness of rays
- `generator_thickness` – integer; the thickness of generators
- `font_size` – integer; the size of font used for labels
- `ray_label` – string or list of strings used for ray labels; use `None` to hide labels
- `wall_label` – string or list of strings used for wall labels; use `None` to hide labels
- `radius` – a positive number, the radius of the cut-off region for "round" mode
- `xmin`, `xmax`, `ymin`, `ymax`, `zmin`, `zmax` – numbers determining the cut-off region for "box" mode. Note that you cannot exclude the origin - if you try to do so, bounds will be automatically expanded to include it.
- `lattice_filter` – a callable, taking as an argument a lattice point and returning `True` if this point should be included on the plot (useful, e.g. for plotting sublattices)
- `wall_zorder`, `ray_zorder`, `generator_zorder`, `point_zorder`, `label_zorder` – integers, z-orders for different classes of objects. By default all values are negative, so that you can add other graphic objects on top of a toric plot. You may need to adjust these parameters if you want to put a toric plot on top of something else or if you want to overlap several toric plots.

You can see the current default value of any options by typing, e.g.

```
sage: toric_plotter.options("show_rays")
True
```

```
>>> from sage.all import *
>>> toric_plotter.options("show_rays")
True
```

If the default value is `None`, it means that the actual default is determined later based on the known options. Note, that not all options can be determined in such a way, so you should not set options to `None` unless it was its original state. (You can always revert to this "original state" using *reset_options()*.)

EXAMPLES:

The following line will make all subsequent toric plotting commands to draw "rainbows" from walls:

```
sage: toric_plotter.options(wall_color='rainbow')
```

```
>>> from sage.all import *
>>> toric_plotter.options(wall_color='rainbow')
```

If you prefer a less colorful output (e.g. if you need black-and-white illustrations for a paper), you can use something like this:

```
sage: toric_plotter.options(wall_color='grey')
```

```
>>> from sage.all import *
>>> toric_plotter.options(wall_color='grey')
```

`sage.geometry.toric_plotter.`**`reset_options`**`()`

Reset options for plots of toric geometry objects.

OUTPUT: none

EXAMPLES:

```
sage: toric_plotter.options("show_rays")
True
sage: toric_plotter.options(show_rays=False)
sage: toric_plotter.options("show_rays")
False
```

```
>>> from sage.all import *
>>> toric_plotter.options("show_rays")
True
>>> toric_plotter.options(show_rays=False)
>>> toric_plotter.options("show_rays")
False
```

Now all toric plots will not show rays, unless explicitly requested. If you want to go back to "default defaults", use this method:

```
sage: toric_plotter.reset_options()
sage: toric_plotter.options("show_rays")
True
```

```
>>> from sage.all import *
>>> toric_plotter.reset_options()
>>> toric_plotter.options("show_rays")
True
```

`sage.geometry.toric_plotter.sector(ray1, ray2, **extra_options)`

Plot a sector between `ray1` and `ray2` centered at the origin.

Note

This function was intended for plotting strictly convex cones, so it plots the smaller sector between `ray1` and `ray2` and, therefore, they cannot be opposite. If you do want to use this function for bigger regions, split them into several parts.

Note

As of version 4.6 Sage does not have a graphic primitive for sectors in 3-dimensional space, so this function will actually approximate them using polygons (the number of vertices used depends on the angle between rays).

INPUT:

- `ray1, ray2` – rays in 2- or 3-dimensional space of the same length
- `extra_options` – dictionary of options that should be passed to lower level plotting functions

OUTPUT: a plot

EXAMPLES:

```
sage: from sage.geometry.toric_plotter import sector
sage: sector((1,0), (0,1))                                              #␣
↪needs sage.symbolic
Graphics object consisting of 1 graphics primitive
sage: sector((3,2,1), (1,2,3))                                          #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> from sage.geometry.toric_plotter import sector
>>> sector((Integer(1),Integer(0)), (Integer(0),Integer(1)))                     ␣
↪                                  # needs sage.symbolic
Graphics object consisting of 1 graphics primitive
>>> sector((Integer(3),Integer(2),Integer(1)), (Integer(1),Integer(2),
↪Integer(3)))                                                   # needs sage.plot
Graphics3d Object
```

2.5.9 Groebner Fans

Sage provides much of the functionality of `gfan`, which is a software package whose main function is to enumerate all reduced Groebner bases of a polynomial ideal. The reduced Groebner bases yield the maximal cones in the Groebner fan of the ideal. Several subcomputations can be issued and additional tools are included. Among these the highlights are:

- Commands for computing tropical varieties.

- Interactive walks in the Groebner fan of an ideal.
- Commands for graphical renderings of Groebner fans and monomial ideals.

AUTHORS:

- Anders Nedergaard Jensen: Wrote the `gfan` C++ program, which implements algorithms many of which were invented by Jensen, Komei Fukuda, and Rekha Thomas. All the underlying hard work of the Groebner fans functionality of Sage depends on this C++ program.
- William Stein (2006-04-20): Wrote first version of the Sage code for working with Groebner fans.
- Tristram Bogart: the design of the Sage interface to `gfan` is joint work with Tristram Bogart, who also supplied numerous examples.
- Marshall Hampton (2008-03-25): Rewrote various functions to use `gfan-0.3`. This is still a work in progress, comments are appreciated on sage-devel@googlegroups.com (or personally at hamptonio@gmail.com).

EXAMPLES:

```
sage: x,y = QQ['x,y'].gens()
sage: i = ideal(x^2 - y^2 + 1)
sage: g = i.groebner_fan()
sage: g.reduced_groebner_bases()
[[x^2 - y^2 + 1], [-x^2 + y^2 - 1]]
```

```
>>> from sage.all import *
>>> x,y = QQ['x,y'].gens()
>>> i = ideal(x**Integer(2) - y**Integer(2) + Integer(1))
>>> g = i.groebner_fan()
>>> g.reduced_groebner_bases()
[[x^2 - y^2 + 1], [-x^2 + y^2 - 1]]
```

REFERENCES:

- Anders N. Jensen; *Gfan, a software system for Groebner fans*; http://home.math.au.dk/jensen/software/gfan/gfan.html

class `sage.rings.polynomial.groebner_fan.`**`GroebnerFan`**(*I*, *is_groebner_basis=False*, *symmetry=None*, *verbose=False*)

Bases: `SageObject`

This class is used to access capabilities of the program `Gfan`.

In addition to computing Groebner fans, `Gfan` can compute other things in tropical geometry such as tropical prevarieties.

INPUT:

- `I` – ideal in a multivariate polynomial ring
- `is_groebner_basis` – boolean (default: `False`); if `True`, then I.gens() must be a Groebner basis with respect to the standard degree lexicographic term order
- `symmetry` – (default: `None`) if not `None`, describes symmetries of the ideal
- `verbose` – (default: `False`) if `True`, printout useful info during computations

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: I = R.ideal([x^2*y - z, y^2*z - x, z^2*x - y])
sage: G = I.groebner_fan(); G
Groebner fan of the ideal:
Ideal (x^2*y - z, y^2*z - x, x*z^2 - y) of Multivariate Polynomial Ring in x, y,␣
↪z over Rational Field
```

```
>>> from sage.all import *
>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> I = R.ideal([x**Integer(2)*y - z, y**Integer(2)*z - x, z**Integer(2)*x - y])
>>> G = I.groebner_fan(); G
Groebner fan of the ideal:
Ideal (x^2*y - z, y^2*z - x, x*z^2 - y) of Multivariate Polynomial Ring in x, y,␣
↪z over Rational Field
```

Here is an example of the use of the tropical_intersection command, and then using the RationalPolyhedralFan class to compute the Stanley-Reisner ideal of the tropical prevariety:

```
sage: R.<x,y,z> = QQ[]
sage: I = R.ideal([(x+y+z)^3-1,(x+y+z)^3-x,(x+y+z)-3])
sage: GF = I.groebner_fan()
sage: PF = GF.tropical_intersection()
sage: PF.rays()
[[-1, 0, 0], [0, -1, 0], [0, 0, -1], [1, 1, 1]]
sage: RPF = PF.to_RationalPolyhedralFan()
sage: RPF.Stanley_Reisner_ideal(PolynomialRing(QQ,4,'A, B, C, D'))
Ideal (A*B, A*C, B*C*D) of Multivariate Polynomial Ring in A, B, C, D over␣
↪Rational Field
```

```
>>> from sage.all import *
>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> I = R.ideal([(x+y+z)**Integer(3)-Integer(1),(x+y+z)**Integer(3)-x,(x+y+z)-
↪Integer(3)])
>>> GF = I.groebner_fan()
>>> PF = GF.tropical_intersection()
>>> PF.rays()
[[-1, 0, 0], [0, -1, 0], [0, 0, -1], [1, 1, 1]]
>>> RPF = PF.to_RationalPolyhedralFan()
>>> RPF.Stanley_Reisner_ideal(PolynomialRing(QQ,Integer(4),'A, B, C, D'))
Ideal (A*B, A*C, B*C*D) of Multivariate Polynomial Ring in A, B, C, D over␣
↪Rational Field
```

buchberger()

Return a lexicographic reduced Groebner basis for the ideal.

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: G = R.ideal([x - z^3, y^2 - x + x^2 - z^3*x]).groebner_fan()
sage: G.buchberger()
[-z^3 + y^2, -z^3 + x]
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ,Integer(3), names=('x', 'y', 'z',)); (x, y, z,) = R.
↪_first_ngens(3)
>>> G = R.ideal([x - z**Integer(3), y**Integer(2) - x + x**Integer(2) -␣
↪z**Integer(3)*x]).groebner_fan()
>>> G.buchberger()
[-z^3 + y^2, -z^3 + x]
```

characteristic()

Return the characteristic of the base ring.

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: i1 = ideal(x*z + 6*y*z - z^2, x*y + 6*x*z + y*z - z^2, y^2 + x*z + y*z)
sage: gf = i1.groebner_fan()
sage: gf.characteristic()
0
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ,Integer(3), names=('x', 'y', 'z',)); (x, y, z,) = R.
↪_first_ngens(3)
>>> i1 = ideal(x*z + Integer(6)*y*z - z**Integer(2), x*y + Integer(6)*x*z +␣
↪y*z - z**Integer(2), y**Integer(2) + x*z + y*z)
>>> gf = i1.groebner_fan()
>>> gf.characteristic()
0
```

dimension_of_homogeneity_space()

Return the dimension of the homogeneity space.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: G = R.ideal([y^3 - x^2, y^2 - 13*x]).groebner_fan()
sage: G.dimension_of_homogeneity_space()
0
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ,Integer(2), names=('x', 'y',)); (x, y,) = R._first_
↪ngens(2)
>>> G = R.ideal([y**Integer(3) - x**Integer(2), y**Integer(2) -␣
↪Integer(13)*x]).groebner_fan()
>>> G.dimension_of_homogeneity_space()
0
```

gfan(*cmd='bases'*, *I=None*, *format=None*)

Return the `gfan` output as a string given an input `cmd`.

The default is to produce the list of reduced Groebner bases in `gfan` format.

INPUT:

- `cmd` – string (default: `'bases'`); GFan command
- `I` – ideal (default: `None`)

- `format` – boolean (default: `None`); deprecated

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: gf = R.ideal([x^3-y,y^3-x-1]).groebner_fan()
sage: gf.gfan()
'Q[x,y]\n{{\ny^9-1-y+3*y^3-3*y^6,\nx+1-y^3}\n,\n{\nx^3-y,\ny^3-1-x}\n,\n{\nx^
↪9-1-x,\ny-x^3}\n}\n'
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ,Integer(2), names=('x', 'y',)); (x, y,) = R._first_
↪ngens(2)
>>> gf = R.ideal([x**Integer(3)-y,y**Integer(3)-x-Integer(1)]).groebner_fan()
>>> gf.gfan()
'Q[x,y]\n{{\ny^9-1-y+3*y^3-3*y^6,\nx+1-y^3}\n,\n{\nx^3-y,\ny^3-1-x}\n,\n{\nx^
↪9-1-x,\ny-x^3}\n}\n'
```

homogeneity_space()

Return the homogeneity space of a the list of polynomials that define this Groebner fan.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: G = R.ideal([y^3 - x^2, y^2 - 13*x]).groebner_fan()
sage: H = G.homogeneity_space()
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ,Integer(2), names=('x', 'y',)); (x, y,) = R._first_
↪ngens(2)
>>> G = R.ideal([y**Integer(3) - x**Integer(2), y**Integer(2) -␣
↪Integer(13)*x]).groebner_fan()
>>> H = G.homogeneity_space()
```

ideal()

Return the ideal the was used to define this Groebner fan.

EXAMPLES:

```
sage: R.<x1,x2> = PolynomialRing(QQ,2)
sage: gf = R.ideal([x1^3-x2,x2^3-2*x1-2]).groebner_fan()
sage: gf.ideal()
Ideal (x1^3 - x2, x2^3 - 2*x1 - 2) of Multivariate Polynomial Ring in x1, x2␣
↪over Rational Field
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ,Integer(2), names=('x1', 'x2',)); (x1, x2,) = R._
↪first_ngens(2)
>>> gf = R.ideal([x1**Integer(3)-x2,x2**Integer(3)-Integer(2)*x1-Integer(2)]).
↪groebner_fan()
>>> gf.ideal()
Ideal (x1^3 - x2, x2^3 - 2*x1 - 2) of Multivariate Polynomial Ring in x1, x2␣
↪over Rational Field
```

interactive(**args*, ***kwds*)

See the documentation for self[0].interactive(). This does not work with the notebook.

EXAMPLES:

```
sage: print("This is not easily doc-testable; please write a good one!")
This is not easily doc-testable; please write a good one!
```

```
>>> from sage.all import *
>>> print("This is not easily doc-testable; please write a good one!")
This is not easily doc-testable; please write a good one!
```

maximal_total_degree_of_a_groebner_basis()

Return the maximal total degree of any Groebner basis.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: G = R.ideal([y^3 - x^2, y^2 - 13*x]).groebner_fan()
sage: G.maximal_total_degree_of_a_groebner_basis()
4
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ,Integer(2), names=('x', 'y',)); (x, y,) = R._first_
↪ngens(2)
>>> G = R.ideal([y**Integer(3) - x**Integer(2), y**Integer(2) -␣
↪Integer(13)*x]).groebner_fan()
>>> G.maximal_total_degree_of_a_groebner_basis()
4
```

minimal_total_degree_of_a_groebner_basis()

Return the minimal total degree of any Groebner basis.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: G = R.ideal([y^3 - x^2, y^2 - 13*x]).groebner_fan()
sage: G.minimal_total_degree_of_a_groebner_basis()
2
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ,Integer(2), names=('x', 'y',)); (x, y,) = R._first_
↪ngens(2)
>>> G = R.ideal([y**Integer(3) - x**Integer(2), y**Integer(2) -␣
↪Integer(13)*x]).groebner_fan()
>>> G.minimal_total_degree_of_a_groebner_basis()
2
```

mixed_volume()

Return the mixed volume of the generators of this ideal.

This is not really an ideal property, it can depend on the generators used.

The generators must give a square system (as many polynomials as variables).

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: example_ideal = R.ideal([x^2-y-1,y^2-z-1,z^2-x-1])
sage: gf = example_ideal.groebner_fan()
sage: mv = gf.mixed_volume()
sage: mv
8

sage: R2.<x,y> = QQ[]
sage: g1 = 1 - x + x^7*y^3 + 2*x^8*y^4
sage: g2 = 2 + y + 3*x^7*y^3 + x^8*y^4
sage: example2 = R2.ideal([g1,g2])
sage: example2.groebner_fan().mixed_volume()
15
```

```
>>> from sage.all import *
>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> example_ideal = R.ideal([x**Integer(2)-y-Integer(1),y**Integer(2)-z-
↪Integer(1),z**Integer(2)-x-Integer(1)])
>>> gf = example_ideal.groebner_fan()
>>> mv = gf.mixed_volume()
>>> mv
8

>>> R2 = QQ['x, y']; (x, y,) = R2._first_ngens(2)
>>> g1 = Integer(1) - x + x**Integer(7)*y**Integer(3) +␣
↪Integer(2)*x**Integer(8)*y**Integer(4)
>>> g2 = Integer(2) + y + Integer(3)*x**Integer(7)*y**Integer(3) +␣
↪x**Integer(8)*y**Integer(4)
>>> example2 = R2.ideal([g1,g2])
>>> example2.groebner_fan().mixed_volume()
15
```

number_of_reduced_groebner_bases()

Return the number of reduced Groebner bases.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: G = R.ideal([y^3 - x^2, y^2 - 13*x]).groebner_fan()
sage: G.number_of_reduced_groebner_bases()
3
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ,Integer(2), names=('x', 'y',)); (x, y,) = R._first_
↪ngens(2)
>>> G = R.ideal([y**Integer(3) - x**Integer(2), y**Integer(2) -␣
↪Integer(13)*x]).groebner_fan()
>>> G.number_of_reduced_groebner_bases()
3
```

number_of_variables()

Return the number of variables.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: G = R.ideal([y^3 - x^2, y^2 - 13*x]).groebner_fan()
sage: G.number_of_variables()
2
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ,Integer(2), names=('x', 'y',)); (x, y,) = R._first_
↪ngens(2)
>>> G = R.ideal([y**Integer(3) - x**Integer(2), y**Integer(2) -␣
↪Integer(13)*x]).groebner_fan()
>>> G.number_of_variables()
2
```

```
sage: R = PolynomialRing(QQ,'x',10)
sage: R.inject_variables(globals())
Defining x0, x1, x2, x3, x4, x5, x6, x7, x8, x9
sage: G = ideal([x0 - x9, sum(R.gens())]).groebner_fan()
sage: G.number_of_variables()
10
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ,'x',Integer(10))
>>> R.inject_variables(globals())
Defining x0, x1, x2, x3, x4, x5, x6, x7, x8, x9
>>> G = ideal([x0 - x9, sum(R.gens())]).groebner_fan()
>>> G.number_of_variables()
10
```

polyhedralfan()

Return a polyhedral fan object corresponding to the reduced Groebner bases.

EXAMPLES:

```
sage: R3.<x,y,z> = PolynomialRing(QQ,3)
sage: gf = R3.ideal([x^8-y^4,y^4-z^2,z^2-1]).groebner_fan()
sage: pf = gf.polyhedralfan()
sage: pf.rays()
[[0, 0, 1], [0, 1, 0], [1, 0, 0]]
```

```
>>> from sage.all import *
>>> R3 = PolynomialRing(QQ,Integer(3), names=('x', 'y', 'z',)); (x, y, z,) =␣
↪R3._first_ngens(3)
>>> gf = R3.ideal([x**Integer(8)-y**Integer(4),y**Integer(4)-z**Integer(2),
↪z**Integer(2)-Integer(1)]).groebner_fan()
>>> pf = gf.polyhedralfan()
>>> pf.rays()
[[0, 0, 1], [0, 1, 0], [1, 0, 0]]
```

reduced_groebner_bases()

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ, 3, order='lex')
sage: G = R.ideal([x^2*y - z, y^2*z - x, z^2*x - y]).groebner_fan()
sage: X = G.reduced_groebner_bases()
sage: len(X)
33
sage: X[0]
[z^15 - z, x - z^9, y - z^11]
sage: X[0].ideal()
Ideal (z^15 - z, x - z^9, y - z^11) of Multivariate Polynomial Ring in x, y,␣
↪z over Rational Field
sage: X[:5]
[[z^15 - z, x - z^9, y - z^11],
[y^2 - z^8, x - z^9, y*z^4 - z, -y + z^11],
[y^3 - z^5, x - y^2*z, y^2*z^3 - y, y*z^4 - z, -y^2 + z^8],
[y^4 - z^2, x - y^2*z, y^2*z^3 - y, y*z^4 - z, -y^3 + z^5],
[y^9 - z, y^6*z - y, x - y^2*z, -y^4 + z^2]]
sage: R3.<x,y,z> = PolynomialRing(GF(2477),3)
sage: gf = R3.ideal([300*x^3-y,y^2-z,z^2-12]).groebner_fan()
sage: gf.reduced_groebner_bases()
[[z^2 - 12, y^2 - z, x^3 + 933*y],
[y^4 - 12, x^3 + 933*y, -y^2 + z],
[x^6 - 1062*z, z^2 - 12, -300*x^3 + y],
[x^12 + 200, -300*x^3 + y, -828*x^6 + z]]
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ, Integer(3), order='lex', names=('x', 'y', 'z',));␣
↪(x, y, z,) = R._first_ngens(3)
>>> G = R.ideal([x**Integer(2)*y - z, y**Integer(2)*z - x, z**Integer(2)*x -␣
↪y]).groebner_fan()
>>> X = G.reduced_groebner_bases()
>>> len(X)
33
>>> X[Integer(0)]
[z^15 - z, x - z^9, y - z^11]
>>> X[Integer(0)].ideal()
Ideal (z^15 - z, x - z^9, y - z^11) of Multivariate Polynomial Ring in x, y,␣
↪z over Rational Field
>>> X[:Integer(5)]
[[z^15 - z, x - z^9, y - z^11],
[y^2 - z^8, x - z^9, y*z^4 - z, -y + z^11],
[y^3 - z^5, x - y^2*z, y^2*z^3 - y, y*z^4 - z, -y^2 + z^8],
[y^4 - z^2, x - y^2*z, y^2*z^3 - y, y*z^4 - z, -y^3 + z^5],
[y^9 - z, y^6*z - y, x - y^2*z, -y^4 + z^2]]
>>> R3 = PolynomialRing(GF(Integer(2477)),Integer(3), names=('x', 'y', 'z',));
↪ (x, y, z,) = R3._first_ngens(3)
>>> gf = R3.ideal([Integer(300)*x**Integer(3)-y,y**Integer(2)-z,z**Integer(2)-
↪Integer(12)]).groebner_fan()
>>> gf.reduced_groebner_bases()
[[z^2 - 12, y^2 - z, x^3 + 933*y],
[y^4 - 12, x^3 + 933*y, -y^2 + z],
[x^6 - 1062*z, z^2 - 12, -300*x^3 + y],
[x^12 + 200, -300*x^3 + y, -828*x^6 + z]]
```

render (*file=None*, *larger=False*, *shift=0*, *rgbcolor=(0, 0, 0)*, *polyfill=True*, *scale_colors=True*)

Render a Groebner fan as sage graphics or save as an xfig file.

More precisely, the output is a drawing of the Groebner fan intersected with a triangle. The corners of the triangle are (1,0,0) to the right, (0,1,0) to the left and (0,0,1) at the top. If there are more than three variables in the ring we extend these coordinates with zeros.

INPUT:

- `file` – a filename if you prefer the output saved to a file; this will be in xfig format
- `shift` – shift the positions of the variables in the drawing. For example, with shift=1, the corners will be b (right), c (left), and d (top). The shifting is done modulo the number of variables in the polynomial ring. The default is 0.
- `larger` – boolean (default: `False`); if `True`, make the triangle larger so that the shape of the Groebner region appears. Affects the xfig file but probably not the sage graphics (?).
- `rgbcolor` – this will not affect the saved xfig file, only the sage graphics produced
- `polyfill` – whether or not to fill the cones with a color determined by the highest degree in each reduced Groebner basis for that cone
- `scale_colors` – if `True`, this will normalize color values to try to maximize the range

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: G = R.ideal([y^3 - x^2, y^2 - 13*x,z]).groebner_fan()
sage: test_render = G.render()                                              #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ,Integer(3), names=('x', 'y', 'z',)); (x, y, z,) = R.
↪_first_ngens(3)
>>> G = R.ideal([y**Integer(3) - x**Integer(2), y**Integer(2) - Integer(13)*x,
↪z]).groebner_fan()
>>> test_render = G.render()                                                #␣
↪needs sage.plot
```

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: G = R.ideal([x^2*y - z, y^2*z - x, z^2*x - y]).groebner_fan()
sage: test_render = G.render(larger=True)                                   #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ,Integer(3), names=('x', 'y', 'z',)); (x, y, z,) = R.
↪_first_ngens(3)
>>> G = R.ideal([x**Integer(2)*y - z, y**Integer(2)*z - x, z**Integer(2)*x -␣
↪y]).groebner_fan()
>>> test_render = G.render(larger=True)                                     #␣
↪needs sage.plot
```

render3d(*verbose=False*)

For a Groebner fan of an ideal in a ring with four variables, this function intersects the fan with the standard simplex perpendicular to (1,1,1,1), creating a 3d polytope, which is then projected into 3 dimensions. The edges of this projected polytope are returned as lines.

EXAMPLES:

```
sage: R4.<w,x,y,z> = PolynomialRing(QQ,4)
sage: gf = R4.ideal([w^2-x,x^2-y,y^2-z,z^2-x]).groebner_fan()
sage: three_d = gf.render3d()                                              #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> R4 = PolynomialRing(QQ,Integer(4), names=('w', 'x', 'y', 'z',)); (w, x, y,
↪ z,) = R4._first_ngens(4)
>>> gf = R4.ideal([w**Integer(2)-x,x**Integer(2)-y,y**Integer(2)-z,
↪z**Integer(2)-x]).groebner_fan()
>>> three_d = gf.render3d()                                              #␣
↪needs sage.plot
```

ring()

Return the multivariate polynomial ring.

EXAMPLES:

```
sage: R.<x1,x2> = PolynomialRing(QQ,2)
sage: gf = R.ideal([x1^3-x2,x2^3-x1-2]).groebner_fan()
sage: gf.ring()
Multivariate Polynomial Ring in x1, x2 over Rational Field
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ,Integer(2), names=('x1', 'x2',)); (x1, x2,) = R._
↪first_ngens(2)
>>> gf = R.ideal([x1**Integer(3)-x2,x2**Integer(3)-x1-Integer(2)]).groebner_
↪fan()
>>> gf.ring()
Multivariate Polynomial Ring in x1, x2 over Rational Field
```

tropical_basis(*check=True*, *verbose=False*)

Return a tropical basis for the tropical curve associated to this ideal.

INPUT:

- `check` – boolean (default: `True`); if `True` raises a `ValueError` exception if this ideal does not define a tropical curve (i.e., the condition that R/I has dimension equal to 1 + the dimension of the homogeneity space is not satisfied)

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3, order='lex')
sage: G = R.ideal([y^3-3*x^2, z^3-x-y-2*y^3+2*x^2]).groebner_fan()
sage: G
Groebner fan of the ideal:
Ideal (-3*x^2 + y^3, 2*x^2 - x - 2*y^3 - y + z^3) of Multivariate Polynomial␣
↪Ring in x, y, z over Rational Field
sage: G.tropical_basis()
[-3*x^2 + y^3, 2*x^2 - x - 2*y^3 - y + z^3, 3/4*x + y^3 + 3/4*y - 3/4*z^3]
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ,Integer(3), order='lex', names=('x', 'y', 'z',));␣
```

(continues on next page)

(continued from previous page)

```
↪(x, y, z,) = R._first_ngens(3)
>>> G = R.ideal([y**Integer(3)-Integer(3)*x**Integer(2), z**Integer(3)-x-y-
↪Integer(2)*y**Integer(3)+Integer(2)*x**Integer(2)]).groebner_fan()
>>> G
Groebner fan of the ideal:
Ideal (-3*x^2 + y^3, 2*x^2 - x - 2*y^3 - y + z^3) of Multivariate Polynomial␣
↪Ring in x, y, z over Rational Field
>>> G.tropical_basis()
[-3*x^2 + y^3, 2*x^2 - x - 2*y^3 - y + z^3, 3/4*x + y^3 + 3/4*y - 3/4*z^3]
```

tropical_intersection(*parameters=[]*, *symmetry_generators=[]*, **args*, ***kwds*)

Return information about the tropical intersection of the polynomials defining the ideal.

This is the common refinement of the outward-pointing normal fans of the Newton polytopes of the generators of the ideal. Note that some people use the inward-pointing normal fans.

INPUT:

- `parameters` – (optional) list of variables to be considered as parameters
- `symmetry_generators` – (optional) generators of the symmetry group

OUTPUT: a TropicalPrevariety object

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: I = R.ideal(x*z + 6*y*z - z^2, x*y + 6*x*z + y*z - z^2, y^2 + x*z + y*z)
sage: gf = I.groebner_fan()
sage: pf = gf.tropical_intersection()
sage: pf.rays()
[[-2, 1, 1]]

sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: f1 = x*y*z - 1
sage: f2 = f1*(x^2 + y^2 + z^2)
sage: f3 = f2*(x + y + z - 1)
sage: I = R.ideal([f1,f2,f3])
sage: gf = I.groebner_fan()
sage: pf = gf.tropical_intersection(symmetry_generators = '(1,2,0),(1,0,2)')
sage: pf.rays()
[[-2, 1, 1], [1, -2, 1], [1, 1, -2]]

sage: R.<x,y,z> = QQ[]
sage: I = R.ideal([(x+y+z)^2-1,(x+y+z)-x,(x+y+z)-3])
sage: GF = I.groebner_fan()
sage: TI = GF.tropical_intersection()
sage: TI.rays()
[[-1, 0, 0], [0, -1, -1], [1, 1, 1]]
sage: GF = I.groebner_fan()
sage: TI = GF.tropical_intersection(parameters=[y])
sage: TI.rays()
[[-1, 0, 0]]
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ,Integer(3), names=('x', 'y', 'z',)); (x, y, z,) = R.
↪_first_ngens(3)
>>> I = R.ideal(x*z + Integer(6)*y*z - z**Integer(2), x*y + Integer(6)*x*z +␣
↪y*z - z**Integer(2), y**Integer(2) + x*z + y*z)
>>> gf = I.groebner_fan()
>>> pf = gf.tropical_intersection()
>>> pf.rays()
[[-2, 1, 1]]

>>> R = PolynomialRing(QQ,Integer(3), names=('x', 'y', 'z',)); (x, y, z,) = R.
↪_first_ngens(3)
>>> f1 = x*y*z - Integer(1)
>>> f2 = f1*(x**Integer(2) + y**Integer(2) + z**Integer(2))
>>> f3 = f2*(x + y + z - Integer(1))
>>> I = R.ideal([f1,f2,f3])
>>> gf = I.groebner_fan()
>>> pf = gf.tropical_intersection(symmetry_generators = '(1,2,0),(1,0,2)')
>>> pf.rays()
[[-2, 1, 1], [1, -2, 1], [1, 1, -2]]

>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> I = R.ideal([(x+y+z)**Integer(2)-Integer(1),(x+y+z)-x,(x+y+z)-Integer(3)])
>>> GF = I.groebner_fan()
>>> TI = GF.tropical_intersection()
>>> TI.rays()
[[-1, 0, 0], [0, -1, -1], [1, 1, 1]]
>>> GF = I.groebner_fan()
>>> TI = GF.tropical_intersection(parameters=[y])
>>> TI.rays()
[[-1, 0, 0]]
```

weight_vectors()

Return the weight vectors corresponding to the reduced Groebner bases.

EXAMPLES:

```
sage: r3.<x,y,z> = PolynomialRing(QQ,3)
sage: g = r3.ideal([x^3+y,y^3-z,z^2-x]).groebner_fan()
sage: g.weight_vectors()
[(3, 7, 1), (5, 1, 2), (7, 1, 4), (5, 1, 4), (1, 1, 1), (1, 4, 8), (1, 4, 10)]
sage: r4.<x,y,z,w> = PolynomialRing(QQ,4)
sage: g4 = r4.ideal([x^3+y,y^3-z,z^2-x,z^3 - w]).groebner_fan()
sage: len(g4.weight_vectors())
23
```

```
>>> from sage.all import *
>>> r3 = PolynomialRing(QQ,Integer(3), names=('x', 'y', 'z',)); (x, y, z,) =␣
↪r3._first_ngens(3)
>>> g = r3.ideal([x**Integer(3)+y,y**Integer(3)-z,z**Integer(2)-x]).groebner_
↪fan()
>>> g.weight_vectors()
[(3, 7, 1), (5, 1, 2), (7, 1, 4), (5, 1, 4), (1, 1, 1), (1, 4, 8), (1, 4, 10)]
```

(continues on next page)

(continued from previous page)

```
>>> r4 = PolynomialRing(QQ,Integer(4), names=('x', 'y', 'z', 'w',)); (x, y, z,
↪ w,) = r4._first_ngens(4)
>>> g4 = r4.ideal([x**Integer(3)+y,y**Integer(3)-z,z**Integer(2)-x,
↪z**Integer(3) - w]).groebner_fan()
>>> len(g4.weight_vectors())
23
```

class sage.rings.polynomial.groebner_fan.**InitialForm**(*cone*, *rays*, *initial_forms*)

Bases: SageObject

A system of initial forms from a polynomial system.

To each form is associated a cone and a list of polynomials (the initial form system itself).

This class is intended for internal use inside of the *TropicalPrevariety* class.

EXAMPLES:

```
sage: from sage.rings.polynomial.groebner_fan import InitialForm
sage: R.<x,y> = QQ[]
sage: inform = InitialForm([0], [[-1, 0]], [y^2 - 1, y^2 - 2, y^2 - 3])
sage: inform._cone
[0]
```

```
>>> from sage.all import *
>>> from sage.rings.polynomial.groebner_fan import InitialForm
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> inform = InitialForm([Integer(0)], [[-Integer(1), Integer(0)]],␣
↪[y**Integer(2) - Integer(1), y**Integer(2) - Integer(2), y**Integer(2) -␣
↪Integer(3)])
>>> inform._cone
[0]
```

cone()

The cone associated with the initial form system.

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: I = R.ideal([(x+y)^2-1,(x+y)^2-2,(x+y)^2-3])
sage: GF = I.groebner_fan()
sage: PF = GF.tropical_intersection()
sage: pfi0 = PF.initial_form_systems()[0]
sage: pfi0.cone()
[0]
```

```
>>> from sage.all import *
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> I = R.ideal([(x+y)**Integer(2)-Integer(1),(x+y)**Integer(2)-Integer(2),
↪(x+y)**Integer(2)-Integer(3)])
>>> GF = I.groebner_fan()
>>> PF = GF.tropical_intersection()
>>> pfi0 = PF.initial_form_systems()[Integer(0)]
```

(continues on next page)

(continued from previous page)

```
>>> pfi0.cone()
[0]
```

initial_forms()

The initial forms (polynomials).

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: I = R.ideal([(x+y)^2-1,(x+y)^2-2,(x+y)^2-3])
sage: GF = I.groebner_fan()
sage: PF = GF.tropical_intersection()
sage: pfi0 = PF.initial_form_systems()[0]
sage: pfi0.initial_forms()
[y^2 - 1, y^2 - 2, y^2 - 3]
```

```
>>> from sage.all import *
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> I = R.ideal([(x+y)**Integer(2)-Integer(1),(x+y)**Integer(2)-Integer(2),
↪(x+y)**Integer(2)-Integer(3)])
>>> GF = I.groebner_fan()
>>> PF = GF.tropical_intersection()
>>> pfi0 = PF.initial_form_systems()[Integer(0)]
>>> pfi0.initial_forms()
[y^2 - 1, y^2 - 2, y^2 - 3]
```

internal_ray()

A ray internal to the cone associated with the initial form system.

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: I = R.ideal([(x+y)^2-1,(x+y)^2-2,(x+y)^2-3])
sage: GF = I.groebner_fan()
sage: PF = GF.tropical_intersection()
sage: pfi0 = PF.initial_form_systems()[0]
sage: pfi0.internal_ray()
(-1, 0)
```

```
>>> from sage.all import *
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> I = R.ideal([(x+y)**Integer(2)-Integer(1),(x+y)**Integer(2)-Integer(2),
↪(x+y)**Integer(2)-Integer(3)])
>>> GF = I.groebner_fan()
>>> PF = GF.tropical_intersection()
>>> pfi0 = PF.initial_form_systems()[Integer(0)]
>>> pfi0.internal_ray()
(-1, 0)
```

rays()

The rays of the cone associated with the initial form system.

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: I = R.ideal([(x+y)^2-1,(x+y)^2-2,(x+y)^2-3])
sage: GF = I.groebner_fan()
sage: PF = GF.tropical_intersection()
sage: pfi0 = PF.initial_form_systems()[0]
sage: pfi0.rays()
[[-1, 0]]
```

```
>>> from sage.all import *
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> I = R.ideal([(x+y)**Integer(2)-Integer(1),(x+y)**Integer(2)-Integer(2),
↪(x+y)**Integer(2)-Integer(3)])
>>> GF = I.groebner_fan()
>>> PF = GF.tropical_intersection()
>>> pfi0 = PF.initial_form_systems()[Integer(0)]
>>> pfi0.rays()
[[-1, 0]]
```

class sage.rings.polynomial.groebner_fan.**PolyhedralCone**(*gfan_polyhedral_cone*, *ring=Rational Field*)

Bases: SageObject

Convert polymake/gfan data on a polyhedral cone into a sage class.

Currently (18-03-2008) needs a lot of work.

EXAMPLES:

```
sage: R3.<x,y,z> = PolynomialRing(QQ,3)
sage: gf = R3.ideal([x^8-y^4,y^4-z^2,z^2-2]).groebner_fan()
sage: a = gf[0].groebner_cone()
sage: a.facets()
[[0, 0, 1], [0, 1, 0], [1, 0, 0]]
```

```
>>> from sage.all import *
>>> R3 = PolynomialRing(QQ,Integer(3), names=('x', 'y', 'z',)); (x, y, z,) = R3._
↪first_ngens(3)
>>> gf = R3.ideal([x**Integer(8)-y**Integer(4),y**Integer(4)-z**Integer(2),
↪z**Integer(2)-Integer(2)]).groebner_fan()
>>> a = gf[Integer(0)].groebner_cone()
>>> a.facets()
[[0, 0, 1], [0, 1, 0], [1, 0, 0]]
```

ambient_dim()

Return the ambient dimension of the Groebner cone.

EXAMPLES:

```
sage: R3.<x,y,z> = PolynomialRing(QQ,3)
sage: gf = R3.ideal([x^8-y^4,y^4-z^2,z^2-2]).groebner_fan()
sage: a = gf[0].groebner_cone()
sage: a.ambient_dim()
3
```

```
>>> from sage.all import *
>>> R3 = PolynomialRing(QQ,Integer(3), names=('x', 'y', 'z',)); (x, y, z,) =␣
↪R3._first_ngens(3)
>>> gf = R3.ideal([x**Integer(8)-y**Integer(4),y**Integer(4)-z**Integer(2),
↪z**Integer(2)-Integer(2)]).groebner_fan()
>>> a = gf[Integer(0)].groebner_cone()
>>> a.ambient_dim()
3
```

dim()

Return the dimension of the Groebner cone.

EXAMPLES:

```
sage: R3.<x,y,z> = PolynomialRing(QQ,3)
sage: gf = R3.ideal([x^8-y^4,y^4-z^2,z^2-2]).groebner_fan()
sage: a = gf[0].groebner_cone()
sage: a.dim()
3
```

```
>>> from sage.all import *
>>> R3 = PolynomialRing(QQ,Integer(3), names=('x', 'y', 'z',)); (x, y, z,) =␣
↪R3._first_ngens(3)
>>> gf = R3.ideal([x**Integer(8)-y**Integer(4),y**Integer(4)-z**Integer(2),
↪z**Integer(2)-Integer(2)]).groebner_fan()
>>> a = gf[Integer(0)].groebner_cone()
>>> a.dim()
3
```

facets()

Return the inward facet normals of the Groebner cone.

EXAMPLES:

```
sage: R3.<x,y,z> = PolynomialRing(QQ,3)
sage: gf = R3.ideal([x^8-y^4,y^4-z^2,z^2-2]).groebner_fan()
sage: a = gf[0].groebner_cone()
sage: a.facets()
[[0, 0, 1], [0, 1, 0], [1, 0, 0]]
```

```
>>> from sage.all import *
>>> R3 = PolynomialRing(QQ,Integer(3), names=('x', 'y', 'z',)); (x, y, z,) =␣
↪R3._first_ngens(3)
>>> gf = R3.ideal([x**Integer(8)-y**Integer(4),y**Integer(4)-z**Integer(2),
↪z**Integer(2)-Integer(2)]).groebner_fan()
>>> a = gf[Integer(0)].groebner_cone()
>>> a.facets()
[[0, 0, 1], [0, 1, 0], [1, 0, 0]]
```

lineality_dim()

Return the lineality dimension of the Groebner cone. This is just the difference between the ambient dimension and the dimension of the cone.

EXAMPLES:

```
sage: R3.<x,y,z> = PolynomialRing(QQ,3)
sage: gf = R3.ideal([x^8-y^4,y^4-z^2,z^2-2]).groebner_fan()
sage: a = gf[0].groebner_cone()
sage: a.lineality_dim()
0
```

```
>>> from sage.all import *
>>> R3 = PolynomialRing(QQ,Integer(3), names=('x', 'y', 'z',)); (x, y, z,) =␣
↪R3._first_ngens(3)
>>> gf = R3.ideal([x**Integer(8)-y**Integer(4),y**Integer(4)-z**Integer(2),
↪z**Integer(2)-Integer(2)]).groebner_fan()
>>> a = gf[Integer(0)].groebner_cone()
>>> a.lineality_dim()
0
```

relative_interior_point()

Return a point in the relative interior of the Groebner cone.

EXAMPLES:

```
sage: R3.<x,y,z> = PolynomialRing(QQ,3)
sage: gf = R3.ideal([x^8-y^4,y^4-z^2,z^2-2]).groebner_fan()
sage: a = gf[0].groebner_cone()
sage: a.relative_interior_point()
[1, 1, 1]
```

```
>>> from sage.all import *
>>> R3 = PolynomialRing(QQ,Integer(3), names=('x', 'y', 'z',)); (x, y, z,) =␣
↪R3._first_ngens(3)
>>> gf = R3.ideal([x**Integer(8)-y**Integer(4),y**Integer(4)-z**Integer(2),
↪z**Integer(2)-Integer(2)]).groebner_fan()
>>> a = gf[Integer(0)].groebner_cone()
>>> a.relative_interior_point()
[1, 1, 1]
```

class sage.rings.polynomial.groebner_fan.**PolyhedralFan**(*gfan_polyhedral_fan*, *parameter_indices=None*)

Bases: SageObject

Convert polymake/gfan data on a polyhedral fan into a sage class.

INPUT:

- gfan_polyhedral_fan – output from gfan of a polyhedral fan

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: i2 = ideal(x*z + 6*y*z - z^2, x*y + 6*x*z + y*z - z^2, y^2 + x*z + y*z)
sage: gf2 = i2.groebner_fan(verbose=False)
sage: pf = gf2.polyhedralfan()
sage: pf.rays()
[[-1, 0, 1], [-1, 1, 0], [1, -2, 1], [1, 1, -2], [2, -1, -1]]
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ,Integer(3), names=('x', 'y', 'z',)); (x, y, z,) = R._
↪first_ngens(3)
>>> i2 = ideal(x*z + Integer(6)*y*z - z**Integer(2), x*y + Integer(6)*x*z + y*z -␣
↪z**Integer(2), y**Integer(2) + x*z + y*z)
>>> gf2 = i2.groebner_fan(verbose=False)
>>> pf = gf2.polyhedralfan()
>>> pf.rays()
[[-1, 0, 1], [-1, 1, 0], [1, -2, 1], [1, 1, -2], [2, -1, -1]]
```

ambient_dim()

Return the ambient dimension of the Groebner fan.

EXAMPLES:

```
sage: R3.<x,y,z> = PolynomialRing(QQ,3)
sage: gf = R3.ideal([x^8-y^4,y^4-z^2,z^2-2]).groebner_fan()
sage: a = gf.polyhedralfan()
sage: a.ambient_dim()
3
```

```
>>> from sage.all import *
>>> R3 = PolynomialRing(QQ,Integer(3), names=('x', 'y', 'z',)); (x, y, z,) =␣
↪R3._first_ngens(3)
>>> gf = R3.ideal([x**Integer(8)-y**Integer(4),y**Integer(4)-z**Integer(2),
↪z**Integer(2)-Integer(2)]).groebner_fan()
>>> a = gf.polyhedralfan()
>>> a.ambient_dim()
3
```

cones()

A dictionary of cones in which the keys are the cone dimensions. For each dimension, the value is a list of the cones, where each element consists of a list of ray indices.

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: f = 1+x+y+x*y
sage: I = R.ideal([f+z*f, 2*f+z*f, 3*f+z^2*f])
sage: GF = I.groebner_fan()
sage: PF = GF.tropical_intersection()
sage: PF.cones()
{1: [[0], [1], [2], [3], [4], [5]], 2: [[0, 1], [0, 2], [0, 3], [0, 4], [1,␣
↪2], [1, 3], [2, 4], [3, 4], [1, 5], [2, 5], [3, 5], [4, 5]]}
```

```
>>> from sage.all import *
>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> f = Integer(1)+x+y+x*y
>>> I = R.ideal([f+z*f, Integer(2)*f+z*f, Integer(3)*f+z**Integer(2)*f])
>>> GF = I.groebner_fan()
>>> PF = GF.tropical_intersection()
>>> PF.cones()
```

(continues on next page)

(continued from previous page)

```
{1: [[0], [1], [2], [3], [4], [5]], 2: [[0, 1], [0, 2], [0, 3], [0, 4], [1,␣
↪2], [1, 3], [2, 4], [3, 4], [1, 5], [2, 5], [3, 5], [4, 5]]}
```

dim()

Return the dimension of the Groebner fan.

EXAMPLES:

```
sage: R3.<x,y,z> = PolynomialRing(QQ,3)
sage: gf = R3.ideal([x^8-y^4,y^4-z^2,z^2-2]).groebner_fan()
sage: a = gf.polyhedralfan()
sage: a.dim()
3
```

```
>>> from sage.all import *
>>> R3 = PolynomialRing(QQ,Integer(3), names=('x', 'y', 'z',)); (x, y, z,) =␣
↪R3._first_ngens(3)
>>> gf = R3.ideal([x**Integer(8)-y**Integer(4),y**Integer(4)-z**Integer(2),
↪z**Integer(2)-Integer(2)]).groebner_fan()
>>> a = gf.polyhedralfan()
>>> a.dim()
3
```

f_vector()

The f-vector of the fan.

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: f = 1+x+y+x*y
sage: I = R.ideal([f+z*f, 2*f+z*f, 3*f+z^2*f])
sage: GF = I.groebner_fan()
sage: PF = GF.tropical_intersection()
sage: PF.f_vector()
[1, 6, 12]
```

```
>>> from sage.all import *
>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> f = Integer(1)+x+y+x*y
>>> I = R.ideal([f+z*f, Integer(2)*f+z*f, Integer(3)*f+z**Integer(2)*f])
>>> GF = I.groebner_fan()
>>> PF = GF.tropical_intersection()
>>> PF.f_vector()
[1, 6, 12]
```

is_simplicial()

Whether the fan is simplicial or not.

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: f = 1+x+y+x*y
sage: I = R.ideal([f+z*f, 2*f+z*f, 3*f+z^2*f])
```

(continues on next page)

(continued from previous page)

```
sage: GF = I.groebner_fan()
sage: PF = GF.tropical_intersection()
sage: PF.is_simplicial()
True
```

```
>>> from sage.all import *
>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> f = Integer(1)+x+y+x*y
>>> I = R.ideal([f+z*f, Integer(2)*f+z*f, Integer(3)*f+z**Integer(2)*f])
>>> GF = I.groebner_fan()
>>> PF = GF.tropical_intersection()
>>> PF.is_simplicial()
True
```

lineality_dim()

Return the lineality dimension of the fan. This is the dimension of the largest subspace contained in the fan.

EXAMPLES:

```
sage: R3.<x,y,z> = PolynomialRing(QQ,3)
sage: gf = R3.ideal([x^8-y^4,y^4-z^2,z^2-2]).groebner_fan()
sage: a = gf.polyhedralfan()
sage: a.lineality_dim()
0
```

```
>>> from sage.all import *
>>> R3 = PolynomialRing(QQ,Integer(3), names=('x', 'y', 'z',)); (x, y, z,) =␣
↪R3._first_ngens(3)
>>> gf = R3.ideal([x**Integer(8)-y**Integer(4),y**Integer(4)-z**Integer(2),
↪z**Integer(2)-Integer(2)]).groebner_fan()
>>> a = gf.polyhedralfan()
>>> a.lineality_dim()
0
```

maximal_cones()

A dictionary of the maximal cones in which the keys are the cone dimensions. For each dimension, the value is a list of the maximal cones, where each element consists of a list of ray indices.

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: f = 1+x+y+x*y
sage: I = R.ideal([f+z*f, 2*f+z*f, 3*f+z^2*f])
sage: GF = I.groebner_fan()
sage: PF = GF.tropical_intersection()
sage: PF.maximal_cones()
{2: [[0, 1], [0, 2], [0, 3], [0, 4], [1, 2], [1, 3], [2, 4], [3, 4], [1, 5],␣
↪[2, 5], [3, 5], [4, 5]]}
```

```
>>> from sage.all import *
>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> f = Integer(1)+x+y+x*y
```

(continues on next page)

(continued from previous page)

```
>>> I = R.ideal([f+z*f, Integer(2)*f+z*f, Integer(3)*f+z**Integer(2)*f])
>>> GF = I.groebner_fan()
>>> PF = GF.tropical_intersection()
>>> PF.maximal_cones()
{2: [[0, 1], [0, 2], [0, 3], [0, 4], [1, 2], [1, 3], [2, 4], [3, 4], [1, 5],␣
↪[2, 5], [3, 5], [4, 5]]}
```

rays()

A list of rays of the polyhedral fan.

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: i2 = ideal(x*z + 6*y*z - z^2, x*y + 6*x*z + y*z - z^2, y^2 + x*z + y*z)
sage: gf2 = i2.groebner_fan(verbose=False)
sage: pf = gf2.polyhedralfan()
sage: pf.rays()
[[-1, 0, 1], [-1, 1, 0], [1, -2, 1], [1, 1, -2], [2, -1, -1]]
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ,Integer(3), names=('x', 'y', 'z',)); (x, y, z,) = R.
↪_first_ngens(3)
>>> i2 = ideal(x*z + Integer(6)*y*z - z**Integer(2), x*y + Integer(6)*x*z +␣
↪y*z - z**Integer(2), y**Integer(2) + x*z + y*z)
>>> gf2 = i2.groebner_fan(verbose=False)
>>> pf = gf2.polyhedralfan()
>>> pf.rays()
[[-1, 0, 1], [-1, 1, 0], [1, -2, 1], [1, 1, -2], [2, -1, -1]]
```

to_RationalPolyhedralFan()

Convert to the RationalPolyhedralFan class, which is more actively maintained. While the information in each class is essentially the same, the methods and implementation are different.

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: f = 1+x+y+x*y
sage: I = R.ideal([f+z*f, 2*f+z*f, 3*f+z^2*f])
sage: GF = I.groebner_fan()
sage: PF = GF.tropical_intersection()
sage: fan = PF.to_RationalPolyhedralFan()
sage: [tuple(q.facet_normals()) for q in fan]
[(M(0, -1, 0), M(-1, 0, 0)), (M(0, 0, -1), M(-1, 0, 0)), (M(0, 0, 1), M(-1, 0,
↪ 0)), (M(0, 1, 0), M(-1, 0, 0)), (M(0, 0, -1), M(0, -1, 0)), (M(0, 0, 1),␣
↪M(0, -1, 0)), (M(0, 1, 0), M(0, 0, -1)), (M(0, 1, 0), M(0, 0, 1)), (M(1, 0,␣
↪0), M(0, -1, 0)), (M(1, 0, 0), M(0, 0, -1)), (M(1, 0, 0), M(0, 0, 1)), (M(1,
↪ 0, 0), M(0, 1, 0))]
```

```
>>> from sage.all import *
>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> f = Integer(1)+x+y+x*y
>>> I = R.ideal([f+z*f, Integer(2)*f+z*f, Integer(3)*f+z**Integer(2)*f])
```

(continues on next page)

(continued from previous page)

```
>>> GF = I.groebner_fan()
>>> PF = GF.tropical_intersection()
>>> fan = PF.to_RationalPolyhedralFan()
>>> [tuple(q.facet_normals()) for q in fan]
[(M(0, -1, 0), M(-1, 0, 0)), (M(0, 0, -1), M(-1, 0, 0)), (M(0, 0, 1), M(-1, 0,
↪ 0)), (M(0, 1, 0), M(-1, 0, 0)), (M(0, 0, -1), M(0, -1, 0)), (M(0, 0, 1),␣
↪M(0, -1, 0)), (M(0, 1, 0), M(0, 0, -1)), (M(0, 1, 0), M(0, 0, 1)), (M(1, 0,␣
↪0), M(0, -1, 0)), (M(1, 0, 0), M(0, 0, -1)), (M(1, 0, 0), M(0, 0, 1)), (M(1,
↪ 0, 0), M(0, 1, 0))]
```

Here we use the RationalPolyhedralFan's Gale_transform method on a tropical prevariety.

```
sage: fan.Gale_transform()
[ 1  0  0  0  0  1 -2]
[ 0  1  0  0  1  0 -2]
[ 0  0  1  1  0  0 -2]
```

```
>>> from sage.all import *
>>> fan.Gale_transform()
[ 1  0  0  0  0  1 -2]
[ 0  1  0  0  1  0 -2]
[ 0  0  1  1  0  0 -2]
```

class sage.rings.polynomial.groebner_fan.**ReducedGroebnerBasis**(*groebner_fan*, *gens*, *gfan_gens*)

Bases: `SageObject`, `list`

A class for representing reduced Groebner bases as produced by `gfan`.

INPUT:

- `groebner_fan` – a GroebnerFan object from an ideal
- `gens` – the generators of the ideal
- `gfan_gens` – the generators as a gfan string

EXAMPLES:

```
sage: R.<a,b> = PolynomialRing(QQ,2)
sage: gf = R.ideal([a^2-b^2,b-a-1]).groebner_fan()
sage: from sage.rings.polynomial.groebner_fan import ReducedGroebnerBasis
sage: ReducedGroebnerBasis(gf,gf[0],gf[0]._gfan_gens())
[b - 1/2, a + 1/2]
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ,Integer(2), names=('a', 'b',)); (a, b,) = R._first_
↪ngens(2)
>>> gf = R.ideal([a**Integer(2)-b**Integer(2),b-a-Integer(1)]).groebner_fan()
>>> from sage.rings.polynomial.groebner_fan import ReducedGroebnerBasis
>>> ReducedGroebnerBasis(gf,gf[Integer(0)],gf[Integer(0)]._gfan_gens())
[b - 1/2, a + 1/2]
```

groebner_cone(*restrict=False*)

Return defining inequalities for the full-dimensional Groebner cone associated to this marked minimal reduced Groebner basis.

INPUT:

- `restrict` – boolean (default: `False`); if `True`, add an inequality for each coordinate, so that the cone is restricted to the positive orthant

OUTPUT: tuple of integer vectors

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: G = R.ideal([y^3 - x^2, y^2 - 13*x]).groebner_fan()
sage: poly_cone = G[1].groebner_cone()
sage: poly_cone.facets()
[[-1, 2], [1, -1]]
sage: [g.groebner_cone().facets() for g in G]
[[[0, 1], [1, -2]], [[-1, 2], [1, -1]], [[-1, 1], [1, 0]]]
sage: G[1].groebner_cone(restrict=True).facets()
[[-1, 2], [1, -1]]
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ,Integer(2), names=('x', 'y',)); (x, y,) = R._first_
↪ngens(2)
>>> G = R.ideal([y**Integer(3) - x**Integer(2), y**Integer(2) -␣
↪Integer(13)*x]).groebner_fan()
>>> poly_cone = G[Integer(1)].groebner_cone()
>>> poly_cone.facets()
[[-1, 2], [1, -1]]
>>> [g.groebner_cone().facets() for g in G]
[[[0, 1], [1, -2]], [[-1, 2], [1, -1]], [[-1, 1], [1, 0]]]
>>> G[Integer(1)].groebner_cone(restrict=True).facets()
[[-1, 2], [1, -1]]
```

ideal()

Return the ideal generated by this basis.

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: G = R.ideal([x - z^3, y^2 - 13*x]).groebner_fan()
sage: G[0].ideal()
Ideal (-13*z^3 + y^2, -z^3 + x) of Multivariate Polynomial Ring in x, y, z␣
↪over Rational Field
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ,Integer(3), names=('x', 'y', 'z',)); (x, y, z,) = R.
↪_first_ngens(3)
>>> G = R.ideal([x - z**Integer(3), y**Integer(2) - Integer(13)*x]).groebner_
↪fan()
>>> G[Integer(0)].ideal()
Ideal (-13*z^3 + y^2, -z^3 + x) of Multivariate Polynomial Ring in x, y, z␣
↪over Rational Field
```

interactive(*latex=False*, *flippable=False*, *wall=False*, *inequalities=False*, *weight=False*)

Do an interactive walk of the Groebner fan starting at this reduced Groebner basis.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: G = R.ideal([y^3 - x^2, y^2 - 13*x]).groebner_fan()
sage: G[0].interactive()      # not tested
Initializing gfan interactive mode
*********************************************
*     Press control-C to return to Sage     *
*********************************************
....
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ,Integer(2), names=('x', 'y',)); (x, y,) = R._first_
↪ngens(2)
>>> G = R.ideal([y**Integer(3) - x**Integer(2), y**Integer(2) -␣
↪Integer(13)*x]).groebner_fan()
>>> G[Integer(0)].interactive()      # not tested
Initializing gfan interactive mode
*********************************************
*     Press control-C to return to Sage     *
*********************************************
....
```

class sage.rings.polynomial.groebner_fan.**TropicalPrevariety**(*gfan_polyhedral_fan*, *polynomial_system*, *poly_ring*, *parameters=None*)

Bases: `PolyhedralFan`

This class is a subclass of the PolyhedralFan class, with some additional methods for tropical prevarieties.

INPUT:

- `gfan_polyhedral_fan` – output from `gfan` of a polyhedral fan
- `polynomial_system` – list of polynomials
- `poly_ring` – the polynomial ring of the list of polynomials
- `parameters` – (optional) list of variables to be considered as parameters

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: I = R.ideal([(x+y+z)^2-1,(x+y+z)-x,(x+y+z)-3])
sage: GF = I.groebner_fan()
sage: TI = GF.tropical_intersection()
sage: TI._polynomial_system
[x^2 + 2*x*y + y^2 + 2*x*z + 2*y*z + z^2 - 1, y + z, x + y + z - 3]
```

```
>>> from sage.all import *
>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> I = R.ideal([(x+y+z)**Integer(2)-Integer(1),(x+y+z)-x,(x+y+z)-Integer(3)])
>>> GF = I.groebner_fan()
>>> TI = GF.tropical_intersection()
>>> TI._polynomial_system
[x^2 + 2*x*y + y^2 + 2*x*z + 2*y*z + z^2 - 1, y + z, x + y + z - 3]
```

initial_form_systems()

Return a list of systems of initial forms for each cone in the tropical prevariety.

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: I = R.ideal([(x+y)^2-1,(x+y)^2-2,(x+y)^2-3])
sage: GF = I.groebner_fan()
sage: PF = GF.tropical_intersection()
sage: pfi = PF.initial_form_systems()
sage: for q in pfi:
....:     print(q.initial_forms())
[y^2 - 1, y^2 - 2, y^2 - 3]
[x^2 - 1, x^2 - 2, x^2 - 3]
[x^2 + 2*x*y + y^2, x^2 + 2*x*y + y^2, x^2 + 2*x*y + y^2]
```

```
>>> from sage.all import *
>>> R = QQ['x, y']; (x, y,) = R._first_ngens(2)
>>> I = R.ideal([(x+y)**Integer(2)-Integer(1),(x+y)**Integer(2)-Integer(2),
↪(x+y)**Integer(2)-Integer(3)])
>>> GF = I.groebner_fan()
>>> PF = GF.tropical_intersection()
>>> pfi = PF.initial_form_systems()
>>> for q in pfi:
...     print(q.initial_forms())
[y^2 - 1, y^2 - 2, y^2 - 3]
[x^2 - 1, x^2 - 2, x^2 - 3]
[x^2 + 2*x*y + y^2, x^2 + 2*x*y + y^2, x^2 + 2*x*y + y^2]
```

sage.rings.polynomial.groebner_fan.**ideal_to_gfan_format**(*input_ring*, *polys*)

Return the ideal in gfan's notation.

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: polys = [x^2*y - z, y^2*z - x, z^2*x - y]
sage: from sage.rings.polynomial.groebner_fan import ideal_to_gfan_format
sage: ideal_to_gfan_format(R, polys)
'Q[x, y, z]{x^2*y-z,y^2*z-x,x*z^2-y}'
```

```
>>> from sage.all import *
>>> R = PolynomialRing(QQ,Integer(3), names=('x', 'y', 'z',)); (x, y, z,)␣
↪= R._first_ngens(3)
>>> polys = [x**Integer(2)*y - z, y**Integer(2)*z - x, z**Integer(2)*x -␣
↪y]
>>> from sage.rings.polynomial.groebner_fan import ideal_to_gfan_format
>>> ideal_to_gfan_format(R, polys)
'Q[x, y, z]{x^2*y-z,y^2*z-x,x*z^2-y}'
```

sage.rings.polynomial.groebner_fan.**max_degree**(*list_of_polys*)

Compute the maximum degree of a list of polynomials.

EXAMPLES:

```
sage: from sage.rings.polynomial.groebner_fan import max_degree
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: p_list = [x^2-y,x*y^10-x]
sage: max_degree(p_list)
11.0
```

```
>>> from sage.all import *
>>> from sage.rings.polynomial.groebner_fan import max_degree
>>> R = PolynomialRing(QQ,Integer(2), names=('x', 'y',)); (x, y,) = R._first_
↪ngens(2)
>>> p_list = [x**Integer(2)-y,x*y**Integer(10)-x]
>>> max_degree(p_list)
11.0
```

sage.rings.polynomial.groebner_fan.**prefix_check**(*str_list*)

Check if any strings in a list are prefixes of another string in the list.

EXAMPLES:

```
sage: from sage.rings.polynomial.groebner_fan import prefix_check
sage: prefix_check(['z1','z1z1'])
False
sage: prefix_check(['z1','zz1'])
True
```

```
>>> from sage.all import *
>>> from sage.rings.polynomial.groebner_fan import prefix_check
>>> prefix_check(['z1','z1z1'])
False
>>> prefix_check(['z1','zz1'])
True
```

sage.rings.polynomial.groebner_fan.**ring_to_gfan_format**(*input_ring*)

Convert a ring to gfan's format.

EXAMPLES:

```
sage: R.<w,x,y,z> = QQ[]
sage: from sage.rings.polynomial.groebner_fan import ring_to_gfan_format
sage: ring_to_gfan_format(R)
'Q[w, x, y, z]'
sage: R2.<x,y> = GF(2)[]
sage: ring_to_gfan_format(R2)
'Z/2Z[x, y]'
```

```
>>> from sage.all import *
>>> R = QQ['w, x, y, z']; (w, x, y, z,) = R._first_ngens(4)
>>> from sage.rings.polynomial.groebner_fan import ring_to_gfan_format
>>> ring_to_gfan_format(R)
'Q[w, x, y, z]'
>>> R2 = GF(Integer(2))['x, y']; (x, y,) = R2._first_ngens(2)
>>> ring_to_gfan_format(R2)
'Z/2Z[x, y]'
```

sage.rings.polynomial.groebner_fan.**verts_for_normal**(*normal*, *poly*)

Return the exponents of the vertices of a Newton polytope that make up the supporting hyperplane for the given outward normal.

EXAMPLES:

```
sage: from sage.rings.polynomial.groebner_fan import verts_for_normal
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: f1 = x*y*z - 1
sage: f2 = f1*(x^2 + y^2 + 1)
sage: verts_for_normal([1,1,1],f2)
[(3, 1, 1), (1, 3, 1)]
```

```
>>> from sage.all import *
>>> from sage.rings.polynomial.groebner_fan import verts_for_normal
>>> R = PolynomialRing(QQ,Integer(3), names=('x', 'y', 'z',)); (x, y, z,) = R._
↪first_ngens(3)
>>> f1 = x*y*z - Integer(1)
>>> f2 = f1*(x**Integer(2) + y**Integer(2) + Integer(1))
>>> verts_for_normal([Integer(1),Integer(1),Integer(1)],f2)
[(3, 1, 1), (1, 3, 1)]
```

2.6 Base classes for polyhedra

2.6.1 Base class for polyhedra: Initialization and access to Vrepresentation and Hrepresentation

class sage.geometry.polyhedron.base0.**Polyhedron_base0**(*parent*, *Vrep*, *Hrep*, *Vrep_minimal=None*, *Hrep_minimal=None*, *pref_rep=None*, *mutable=False*, ***kwds*)

Bases: `Element`, *Polyhedron*

Initialization and basic access for polyhedra.

See *sage.geometry.polyhedron.base.Polyhedron_base*.

Hrep_generator()

Return an iterator over the objects of the H-representation (inequalities or equations).

EXAMPLES:

```
sage: p = polytopes.hypercube(3)
sage: next(p.Hrep_generator())
An inequality (-1, 0, 0) x + 1 >= 0
```

```
>>> from sage.all import *
>>> p = polytopes.hypercube(Integer(3))
>>> next(p.Hrep_generator())
An inequality (-1, 0, 0) x + 1 >= 0
```

Hrepresentation(*index=None*)

Return the objects of the H-representation. Each entry is either an inequality or a equation.

INPUT:

- `index` – either an integer or `None`

OUTPUT:

The optional argument is an index running from `0` to `self.n_Hrepresentation()-1`. If present, the H-representation object at the given index will be returned. Without an argument, returns the list of all H-representation objects.

EXAMPLES:

```
sage: p = polytopes.hypercube(3, backend='field')
sage: p.Hrepresentation(0)
An inequality (-1, 0, 0) x + 1 >= 0
sage: p.Hrepresentation(0) == p.Hrepresentation()[0]
True
```

```
>>> from sage.all import *
>>> p = polytopes.hypercube(Integer(3), backend='field')
>>> p.Hrepresentation(Integer(0))
An inequality (-1, 0, 0) x + 1 >= 0
>>> p.Hrepresentation(Integer(0)) == p.Hrepresentation()[Integer(0)]
True
```

Hrepresentation_str(*separator='\n'*, *latex=False*, *style='>='*, *align=None*, ***kwds*)

Return a human-readable string representation of the Hrepresentation of this polyhedron.

INPUT:

- `separator` – string (default: `'\n'`)
- `latex` – boolean (default: `False`)
- **`style` – either `'positive'` (making all coefficients positive)**
  or `'<='`, or `'>='`; default is `'>='`
- **`align` – boolean or `None''; default is ``None` in which case**
  `align` is `True` if `separator` is the newline character. If set, then the lines of the output string are aligned by the comparison symbol by padding blanks.

Keyword parameters of *`repr_pretty()`* are passed on:

- `prefix` – string
- `indices` – tuple or other iterable

OUTPUT: string

EXAMPLES:

```
sage: P = polytopes.permutahedron(3)
sage: print(P.Hrepresentation_str())
x0 + x1 + x2 ==  6
     x0 + x1 >=  3
    -x0 - x1 >= -5
          x1 >=  1
         -x0 >= -3
          x0 >=  1
         -x1 >= -3
```

(continues on next page)

(continued from previous page)

```
sage: print(P.Hrepresentation_str(style='<='))
-x0 - x1 - x2 == -6
     -x0 - x1 <= -3
      x0 + x1 <=  5
          -x1 <= -1
           x0 <=  3
          -x0 <= -1
           x1 <=  3

sage: print(P.Hrepresentation_str(style='positive'))
x0 + x1 + x2 == 6
     x0 + x1 >= 3
           5 >= x0 + x1
          x1 >= 1
           3 >= x0
          x0 >= 1
           3 >= x1

sage: print(P.Hrepresentation_str(latex=True))
\begin{array}{rcl}
x_{0} + x_{1} + x_{2} & =    &  6 \\
        x_{0} + x_{1} & \geq &  3 \\
       -x_{0} - x_{1} & \geq & -5 \\
                x_{1} & \geq &  1 \\
               -x_{0} & \geq & -3 \\
                x_{0} & \geq &  1 \\
               -x_{1} & \geq & -3
\end{array}

sage: print(P.Hrepresentation_str(align=False))
x0 + x1 + x2 == 6
x0 + x1 >= 3
-x0 - x1 >= -5
x1 >= 1
-x0 >= -3
x0 >= 1
-x1 >= -3

sage: c = polytopes.cube()
sage: c.Hrepresentation_str(separator=', ', style='positive')
'1 >= x0, 1 >= x1, 1 >= x2, 1 + x0 >= 0, 1 + x2 >= 0, 1 + x1 >= 0'
```

```
>>> from sage.all import *
>>> P = polytopes.permutahedron(Integer(3))
>>> print(P.Hrepresentation_str())
x0 + x1 + x2 ==  6
     x0 + x1 >=  3
    -x0 - x1 >= -5
          x1 >=  1
         -x0 >= -3
          x0 >=  1
         -x1 >= -3
```

(continues on next page)

(continued from previous page)

```
>>> print(P.Hrepresentation_str(style='<='))
-x0 - x1 - x2 == -6
     -x0 - x1 <= -3
      x0 + x1 <=  5
          -x1 <= -1
           x0 <=  3
          -x0 <= -1
           x1 <=  3

>>> print(P.Hrepresentation_str(style='positive'))
x0 + x1 + x2 == 6
     x0 + x1 >= 3
           5 >= x0 + x1
          x1 >= 1
           3 >= x0
          x0 >= 1
           3 >= x1

>>> print(P.Hrepresentation_str(latex=True))
\begin{array}{rcl}
x_{0} + x_{1} + x_{2} & =    &  6 \\
        x_{0} + x_{1} & \geq &  3 \\
       -x_{0} - x_{1} & \geq & -5 \\
                x_{1} & \geq &  1 \\
               -x_{0} & \geq & -3 \\
                x_{0} & \geq &  1 \\
               -x_{1} & \geq & -3
\end{array}

>>> print(P.Hrepresentation_str(align=False))
x0 + x1 + x2 == 6
x0 + x1 >= 3
-x0 - x1 >= -5
x1 >= 1
-x0 >= -3
x0 >= 1
-x1 >= -3

>>> c = polytopes.cube()
>>> c.Hrepresentation_str(separator=', ', style='positive')
'1 >= x0, 1 >= x1, 1 >= x2, 1 + x0 >= 0, 1 + x2 >= 0, 1 + x1 >= 0'
```

Vrep_generator()

Return an iterator over the objects of the V-representation (vertices, rays, and lines).

EXAMPLES:

```
sage: p = polytopes.cyclic_polytope(3,4)
sage: vg = p.Vrep_generator()
sage: next(vg)
A vertex at (0, 0, 0)
```

(continues on next page)

(continued from previous page)

```
sage: next(vg)
A vertex at (1, 1, 1)
```

```
>>> from sage.all import *
>>> p = polytopes.cyclic_polytope(Integer(3),Integer(4))
>>> vg = p.Vrep_generator()
>>> next(vg)
A vertex at (0, 0, 0)
>>> next(vg)
A vertex at (1, 1, 1)
```

Vrepresentation(*index=None*)

Return the objects of the V-representation. Each entry is either a vertex, a ray, or a line.

See *sage.geometry.polyhedron.constructor* for a definition of vertex/ray/line.

INPUT:

- `index` – either an integer or `None`

OUTPUT:

The optional argument is an index running from `0` to `self.n_Vrepresentation()-1`. If present, the V-representation object at the given index will be returned. Without an argument, returns the list of all V-representation objects.

EXAMPLES:

```
sage: p = polytopes.simplex(4, project=True)
sage: p.Vrepresentation(0)
A vertex at (0.7071067812, 0.4082482905, 0.2886751346, 0.2236067977)
sage: p.Vrepresentation(0) == p.Vrepresentation() [0]
True
```

```
>>> from sage.all import *
>>> p = polytopes.simplex(Integer(4), project=True)
>>> p.Vrepresentation(Integer(0))
A vertex at (0.7071067812, 0.4082482905, 0.2886751346, 0.2236067977)
>>> p.Vrepresentation(Integer(0)) == p.Vrepresentation() [Integer(0)]
True
```

backend()

Return the backend used.

OUTPUT:

The name of the backend used for computations. It will be one of the following backends:

- `ppl` the Parma Polyhedra Library
- `cdd` CDD
- `normaliz` normaliz
- `polymake` polymake
- `field` a generic Sage implementation

EXAMPLES:

```
sage: triangle = Polyhedron(vertices=[[1, 0], [0, 1], [1, 1]])
sage: triangle.backend()
'ppl'
sage: D = polytopes.dodecahedron()                                          #␣
↪needs sage.groups sage.rings.number_field
sage: D.backend()                                                           #␣
↪needs sage.groups sage.rings.number_field
'field'
sage: P = Polyhedron([[1.23]])
sage: P.backend()
'cdd'
```

```
>>> from sage.all import *
>>> triangle = Polyhedron(vertices=[[Integer(1), Integer(0)], [Integer(0),␣
↪Integer(1)], [Integer(1), Integer(1)]])
>>> triangle.backend()
'ppl'
>>> D = polytopes.dodecahedron()                                          #␣
↪needs sage.groups sage.rings.number_field
>>> D.backend()                                                           #␣
↪needs sage.groups sage.rings.number_field
'field'
>>> P = Polyhedron([[RealNumber('1.23')]])
>>> P.backend()
'cdd'
```

base_extend(*base_ring*, *backend=None*)

Return a new polyhedron over a larger base ring.

This method can also be used to change the backend.

INPUT:

- `base_ring` – the new base ring
- `backend` – the new backend, see *Polyhedron()* If `None` (the default), attempt to keep the same backend. Otherwise, use the same defaulting behavior as described there.

OUTPUT: the same polyhedron, but over a larger base ring and possibly with a changed backend

EXAMPLES:

```
sage: P = Polyhedron(vertices=[(1,0), (0,1)], rays=[(1,1)], base_ring=ZZ);  P
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices␣
↪and 1 ray
sage: P.base_extend(QQ)
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 2 vertices␣
↪and 1 ray
sage: P.base_extend(QQ) == P
True
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[(Integer(1),Integer(0)), (Integer(0),
↪Integer(1))], rays=[(Integer(1),Integer(1))], base_ring=ZZ);  P
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices␣
```

(continues on next page)

(continued from previous page)

```
↪and 1 ray
>>> P.base_extend(QQ)
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 2 vertices␣
↪and 1 ray
>>> P.base_extend(QQ) == P
True
```

base_ring()

Return the base ring.

OUTPUT:

The ring over which the polyhedron is defined. Must be a sub-ring of the reals to define a polyhedron, in particular comparison must be defined. Popular choices are

- `ZZ` (the ring of integers, lattice polytope),
- `QQ` (exact arithmetic using gmp),
- `RDF` (double precision floating-point arithmetic), or
- `AA` (real algebraic field).

EXAMPLES:

```
sage: triangle = Polyhedron(vertices = [[1,0],[0,1],[1,1]])
sage: triangle.base_ring() == ZZ
True
```

```
>>> from sage.all import *
>>> triangle = Polyhedron(vertices = [[Integer(1),Integer(0)],[Integer(0),
↪Integer(1)],[Integer(1),Integer(1)]])
>>> triangle.base_ring() == ZZ
True
```

cdd_Hrepresentation()

Write the inequalities/equations data of the polyhedron in cdd's H-representation format.

See also

write_cdd_Hrepresentation() – export the polyhedron as a H-representation to a file.

OUTPUT: string

EXAMPLES:

```
sage: p = polytopes.hypercube(2)
sage: print(p.cdd_Hrepresentation())
H-representation
begin
 4 3 rational
 1 -1 0
 1 0 -1
 1 1 0
 1 0 1
```

(continues on next page)

(continued from previous page)

```
end


sage: triangle = Polyhedron(vertices=[[1,0], [0,1], [1,1]], base_ring=AA)    #␣
↪needs sage.rings.number_field
sage: triangle.base_ring()                                                  #␣
↪needs sage.rings.number_field
Algebraic Real Field
sage: triangle.cdd_Hrepresentation()                                        #␣
↪needs sage.rings.number_field
Traceback (most recent call last):
...
TypeError: the base ring must be ZZ, QQ, or RDF
```

```
>>> from sage.all import *
>>> p = polytopes.hypercube(Integer(2))
>>> print(p.cdd_Hrepresentation())
H-representation
begin
 4 3 rational
 1 -1 0
 1 0 -1
 1 1 0
 1 0 1
end
<BLANKLINE>

>>> triangle = Polyhedron(vertices=[[Integer(1),Integer(0)], [Integer(0),
↪Integer(1)], [Integer(1),Integer(1)]], base_ring=AA)    # needs sage.rings.
↪number_field
>>> triangle.base_ring()                                                  #␣
↪needs sage.rings.number_field
Algebraic Real Field
>>> triangle.cdd_Hrepresentation()                                        #␣
↪needs sage.rings.number_field
Traceback (most recent call last):
...
TypeError: the base ring must be ZZ, QQ, or RDF
```

cdd_Vrepresentation()

Write the vertices/rays/lines data of the polyhedron in cdd's V-representation format.

See also

write_cdd_Vrepresentation() – export the polyhedron as a V-representation to a file.

OUTPUT: string

EXAMPLES:

```
sage: q = Polyhedron(vertices = [[1,1],[0,0],[1,0],[0,1]])
```

(continues on next page)

(continued from previous page)

```
sage: print(q.cdd_Vrepresentation())
V-representation
begin
 4 3 rational
 1 0 0
 1 0 1
 1 1 0
 1 1 1
end
```

```
>>> from sage.all import *
>>> q = Polyhedron(vertices = [[Integer(1),Integer(1)],[Integer(0),
↪Integer(0)],[Integer(1),Integer(0)],[Integer(0),Integer(1)]])
>>> print(q.cdd_Vrepresentation())
V-representation
begin
 4 3 rational
 1 0 0
 1 0 1
 1 1 0
 1 1 1
end
```

change_ring(*base_ring*, *backend=None*)

Return the polyhedron obtained by coercing the entries of the vertices/lines/rays of this polyhedron into the given ring.

This method can also be used to change the backend.

INPUT:

- `base_ring` – the new base ring
- `backend` – the new backend or `None` (default), see *Polyhedron()*. If `None` (the default), attempt to keep the same backend. Otherwise, use the same defaulting behavior as described there.

EXAMPLES:

```
sage: P = Polyhedron(vertices=[(1,0), (0,1)], rays=[(1,1)], base_ring=QQ); P
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 2 vertices␣
↪and 1 ray
sage: P.change_ring(ZZ)
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices␣
↪and 1 ray
sage: P.change_ring(ZZ) == P
True

sage: P = Polyhedron(vertices=[(-1.3,0), (0,2.3)], base_ring=RDF); P.
↪vertices()
(A vertex at (-1.3, 0.0), A vertex at (0.0, 2.3))
sage: P.change_ring(QQ).vertices()
(A vertex at (-13/10, 0), A vertex at (0, 23/10))
sage: P == P.change_ring(QQ)
True
```

(continues on next page)

(continued from previous page)

```
sage: P.change_ring(ZZ)
Traceback (most recent call last):
...
TypeError: cannot change the base ring to the Integer Ring

sage: P = polytopes.regular_polygon(3); P                                   #␣
↪needs sage.rings.number_field
A 2-dimensional polyhedron in AA^2 defined as the convex hull of 3 vertices
sage: P.vertices()                                                          #␣
↪needs sage.rings.number_field
(A vertex at (0.?e-16, 1.000000000000000?),
 A vertex at (0.866025403784439?, -0.500000000000000?),
 A vertex at (-0.866025403784439?, -0.500000000000000?))
sage: P.change_ring(QQ)                                                     #␣
↪needs sage.rings.number_field
Traceback (most recent call last):
...
TypeError: cannot change the base ring to the Rational Field
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[(Integer(1),Integer(0)), (Integer(0),
↪Integer(1))], rays=[(Integer(1),Integer(1))], base_ring=QQ); P
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 2 vertices␣
↪and 1 ray
>>> P.change_ring(ZZ)
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices␣
↪and 1 ray
>>> P.change_ring(ZZ) == P
True

>>> P = Polyhedron(vertices=[(-RealNumber('1.3'),Integer(0)), (Integer(0),
↪RealNumber('2.3'))], base_ring=RDF); P.vertices()
(A vertex at (-1.3, 0.0), A vertex at (0.0, 2.3))
>>> P.change_ring(QQ).vertices()
(A vertex at (-13/10, 0), A vertex at (0, 23/10))
>>> P == P.change_ring(QQ)
True
>>> P.change_ring(ZZ)
Traceback (most recent call last):
...
TypeError: cannot change the base ring to the Integer Ring

>>> P = polytopes.regular_polygon(Integer(3)); P                           ␣
↪     # needs sage.rings.number_field
A 2-dimensional polyhedron in AA^2 defined as the convex hull of 3 vertices
>>> P.vertices()                                                          #␣
↪needs sage.rings.number_field
(A vertex at (0.?e-16, 1.000000000000000?),
 A vertex at (0.866025403784439?, -0.500000000000000?),
 A vertex at (-0.866025403784439?, -0.500000000000000?))
>>> P.change_ring(QQ)                                                     #␣
↪needs sage.rings.number_field
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
TypeError: cannot change the base ring to the Rational Field
```

Warning

The base ring `RDF` should be used with care. As it is not an exact ring, certain computations may break or silently produce wrong results, for example changing the base ring from an exact ring into `RDF` may cause a loss of data:

```
sage: P = Polyhedron([[2/3,0],[6666666666666667/10^16,0]], base_ring=AA);␣
↪P         # needs sage.rings.number_field
A 1-dimensional polyhedron in AA^2 defined as the convex hull of 2 vertices
sage: Q = P.change_ring(RDF); Q                                          #␣
↪needs sage.rings.number_field
A 0-dimensional polyhedron in RDF^2 defined as the convex hull of 1 vertex
sage: P.n_vertices() == Q.n_vertices()                                   #␣
↪needs sage.rings.number_field
False
```

```
>>> from sage.all import *
>>> P = Polyhedron([[Integer(2)/Integer(3),Integer(0)],
↪[Integer(6666666666666667)/Integer(10)**Integer(16),Integer(0)]], base_
↪ring=AA); P         # needs sage.rings.number_field
A 1-dimensional polyhedron in AA^2 defined as the convex hull of 2 vertices
>>> Q = P.change_ring(RDF); Q                                          #␣
↪needs sage.rings.number_field
A 0-dimensional polyhedron in RDF^2 defined as the convex hull of 1 vertex
>>> P.n_vertices() == Q.n_vertices()                                   #␣
↪needs sage.rings.number_field
False
```

equation_generator()

Return a generator for the linear equations satisfied by the polyhedron.

EXAMPLES:

```
sage: p = polytopes.regular_polygon(8,base_ring=RDF)
sage: p3 = Polyhedron(vertices = [x+[0] for x in p.vertices()], base_ring=RDF)
sage: next(p3.equation_generator())
An equation (0.0, 0.0, 1.0) x + 0.0 == 0
```

```
>>> from sage.all import *
>>> p = polytopes.regular_polygon(Integer(8),base_ring=RDF)
>>> p3 = Polyhedron(vertices = [x+[Integer(0)] for x in p.vertices()], base_
↪ring=RDF)
>>> next(p3.equation_generator())
An equation (0.0, 0.0, 1.0) x + 0.0 == 0
```

equations()

Return all linear constraints of the polyhedron.

OUTPUT: a tuple of equations

EXAMPLES:

```
sage: test_p = Polyhedron(vertices = [[1,2,3,4],[2,1,3,4],[4,3,2,1],[3,4,1,
↪2]])
sage: test_p.equations()
(An equation (1, 1, 1, 1) x - 10 == 0,)
```

```
>>> from sage.all import *
>>> test_p = Polyhedron(vertices = [[Integer(1),Integer(2),Integer(3),
↪Integer(4)],[Integer(2),Integer(1),Integer(3),Integer(4)],[Integer(4),
↪Integer(3),Integer(2),Integer(1)],[Integer(3),Integer(4),Integer(1),
↪Integer(2)]])
>>> test_p.equations()
(An equation (1, 1, 1, 1) x - 10 == 0,)
```

equations_list()

Return the linear constraints of the polyhedron. As with inequalities, each constraint is given as [b -a1 -a2 … an] where for variables x1, x2,…, xn, the polyhedron satisfies the equation b = a1*x1 + a2*x2 + … + an*xn.

> **Note**
>
> It is recommended to use *equations()* or *equation_generator()* instead to iterate over the list of *Equation* objects.

EXAMPLES:

```
sage: test_p = Polyhedron(vertices = [[1,2,3,4],[2,1,3,4],[4,3,2,1],[3,4,1,
↪2]])
sage: test_p.equations_list()
[[-10, 1, 1, 1, 1]]
```

```
>>> from sage.all import *
>>> test_p = Polyhedron(vertices = [[Integer(1),Integer(2),Integer(3),
↪Integer(4)],[Integer(2),Integer(1),Integer(3),Integer(4)],[Integer(4),
↪Integer(3),Integer(2),Integer(1)],[Integer(3),Integer(4),Integer(1),
↪Integer(2)]])
>>> test_p.equations_list()
[[-10, 1, 1, 1, 1]]
```

inequalities()

Return all inequalities.

OUTPUT: a tuple of inequalities

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[0,0,0],[0,0,1],[0,1,0],[1,0,0],[2,2,2]])
sage: p.inequalities()[0:3]
(An inequality (1, 0, 0) x + 0 >= 0,
 An inequality (0, 1, 0) x + 0 >= 0,
 An inequality (0, 0, 1) x + 0 >= 0)
```

(continues on next page)

(continued from previous page)

```
sage: # needs sage.combinat
sage: p3 = Polyhedron(vertices=Permutations([1, 2, 3, 4]))
sage: ieqs = p3.inequalities()
sage: ieqs[0]
An inequality (0, 1, 1, 1) x - 6 >= 0
sage: list(_)
[-6, 0, 1, 1, 1]
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices = [[Integer(0),Integer(0),Integer(0)],[Integer(0),
↪Integer(0),Integer(1)],[Integer(0),Integer(1),Integer(0)],[Integer(1),
↪Integer(0),Integer(0)],[Integer(2),Integer(2),Integer(2)]])
>>> p.inequalities()[Integer(0):Integer(3)]
(An inequality (1, 0, 0) x + 0 >= 0,
 An inequality (0, 1, 0) x + 0 >= 0,
 An inequality (0, 0, 1) x + 0 >= 0)

>>> # needs sage.combinat
>>> p3 = Polyhedron(vertices=Permutations([Integer(1), Integer(2), Integer(3),
↪ Integer(4)]))
>>> ieqs = p3.inequalities()
>>> ieqs[Integer(0)]
An inequality (0, 1, 1, 1) x - 6 >= 0
>>> list(_)
[-6, 0, 1, 1, 1]
```

inequalities_list()

Return a list of inequalities as coefficient lists.

> **Note**
>
> It is recommended to use *inequalities()* or *inequality_generator()* instead to iterate over the list of *Inequality* objects.

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[0,0,0],[0,0,1],[0,1,0],[1,0,0],[2,2,2]])
sage: p.inequalities_list()[0:3]
[[0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]]

sage: # needs sage.combinat
sage: p3 = Polyhedron(vertices=Permutations([1, 2, 3, 4]))
sage: ieqs = p3.inequalities_list()
sage: ieqs[0]
[-6, 0, 1, 1, 1]
sage: ieqs[-1]
[-3, 0, 1, 0, 1]
sage: ieqs == [list(x) for x in p3.inequality_generator()]
True
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices = [[Integer(0),Integer(0),Integer(0)],[Integer(0),
↪Integer(0),Integer(1)],[Integer(0),Integer(1),Integer(0)],[Integer(1),
↪Integer(0),Integer(0)],[Integer(2),Integer(2),Integer(2)]])
>>> p.inequalities_list()[Integer(0):Integer(3)]
[[0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]]

>>> # needs sage.combinat
>>> p3 = Polyhedron(vertices=Permutations([Integer(1), Integer(2), Integer(3),
↪ Integer(4)]))
>>> ieqs = p3.inequalities_list()
>>> ieqs[Integer(0)]
[-6, 0, 1, 1, 1]
>>> ieqs[-Integer(1)]
[-3, 0, 1, 0, 1]
>>> ieqs == [list(x) for x in p3.inequality_generator()]
True
```

inequality_generator()

Return a generator for the defining inequalities of the polyhedron.

OUTPUT: a generator of the inequality Hrepresentation objects

EXAMPLES:

```
sage: triangle = Polyhedron(vertices=[[1,0],[0,1],[1,1]])
sage: for v in triangle.inequality_generator(): print(v)
An inequality (1, 1) x - 1 >= 0
An inequality (0, -1) x + 1 >= 0
An inequality (-1, 0) x + 1 >= 0
sage: [ v for v in triangle.inequality_generator() ]
[An inequality (1, 1) x - 1 >= 0,
 An inequality (0, -1) x + 1 >= 0,
 An inequality (-1, 0) x + 1 >= 0]
sage: [ [v.A(), v.b()] for v in triangle.inequality_generator() ]
[[(1, 1), -1], [(0, -1), 1], [(-1, 0), 1]]
```

```
>>> from sage.all import *
>>> triangle = Polyhedron(vertices=[[Integer(1),Integer(0)],[Integer(0),
↪Integer(1)],[Integer(1),Integer(1)]])
>>> for v in triangle.inequality_generator(): print(v)
An inequality (1, 1) x - 1 >= 0
An inequality (0, -1) x + 1 >= 0
An inequality (-1, 0) x + 1 >= 0
>>> [ v for v in triangle.inequality_generator() ]
[An inequality (1, 1) x - 1 >= 0,
 An inequality (0, -1) x + 1 >= 0,
 An inequality (-1, 0) x + 1 >= 0]
>>> [ [v.A(), v.b()] for v in triangle.inequality_generator() ]
[[(1, 1), -1], [(0, -1), 1], [(-1, 0), 1]]
```

is_compact()

Test for boundedness of the polytope.

EXAMPLES:

```
sage: p = polytopes.icosahedron()                                          #␣
↪needs sage.groups sage.rings.number_field
sage: p.is_compact()                                                       #␣
↪needs sage.groups sage.rings.number_field
True
sage: p = Polyhedron(ieqs=[[0,1,0,0],[0,0,1,0],[0,0,0,1],[1,-1,0,0]])
sage: p.is_compact()
False
```

```
>>> from sage.all import *
>>> p = polytopes.icosahedron()                                        #␣
↪needs sage.groups sage.rings.number_field
>>> p.is_compact()                                                     #␣
↪needs sage.groups sage.rings.number_field
True
>>> p = Polyhedron(ieqs=[[Integer(0),Integer(1),Integer(0),Integer(0)],
↪[Integer(0),Integer(0),Integer(1),Integer(0)],[Integer(0),Integer(0),
↪Integer(0),Integer(1)],[Integer(1),-Integer(1),Integer(0),Integer(0)]])
>>> p.is_compact()
False
```

is_immutable()

Return `True` if the polyhedron is immutable, i.e. it cannot be modified in place.

EXAMPLES:

```
sage: p = polytopes.cube(backend='field')
sage: p.is_immutable()
True
```

```
>>> from sage.all import *
>>> p = polytopes.cube(backend='field')
>>> p.is_immutable()
True
```

is_mutable()

Return `True` if the polyhedron is mutable, i.e. it can be modified in place.

EXAMPLES:

```
sage: p = polytopes.cube(backend='field')
sage: p.is_mutable()
False
```

```
>>> from sage.all import *
>>> p = polytopes.cube(backend='field')
>>> p.is_mutable()
False
```

line_generator()

Return a generator for the lines of the polyhedron.

EXAMPLES:

```
sage: pr = Polyhedron(rays = [[1,0],[-1,0],[0,1]], vertices = [[-1,-1]])
sage: next(pr.line_generator()).vector()
(1, 0)
```

```
>>> from sage.all import *
>>> pr = Polyhedron(rays = [[Integer(1),Integer(0)],[-Integer(1),Integer(0)],
↪[Integer(0),Integer(1)]], vertices = [[-Integer(1),-Integer(1)]])
>>> next(pr.line_generator()).vector()
(1, 0)
```

lines()

Return all lines of the polyhedron.

OUTPUT: a tuple of lines

EXAMPLES:

```
sage: p = Polyhedron(rays = [[1,0],[-1,0],[0,1],[1,1]], vertices = [[-2,-2],
↪[2,3]])
sage: p.lines()
(A line in the direction (1, 0),)
```

```
>>> from sage.all import *
>>> p = Polyhedron(rays = [[Integer(1),Integer(0)],[-Integer(1),Integer(0)],
↪[Integer(0),Integer(1)],[Integer(1),Integer(1)]], vertices = [[-Integer(2),-
↪Integer(2)],[Integer(2),Integer(3)]])
>>> p.lines()
(A line in the direction (1, 0),)
```

lines_list()

Return a list of lines of the polyhedron. The line data is given as a list of coordinates rather than as a Hrepresentation object.

> **Note**
>
> It is recommended to use *line_generator()* instead to iterate over the list of *Line* objects.

EXAMPLES:

```
sage: p = Polyhedron(rays = [[1,0],[-1,0],[0,1],[1,1]], vertices = [[-2,-2],
↪[2,3]])
sage: p.lines_list()
[[1, 0]]
sage: p.lines_list() == [list(x) for x in p.line_generator()]
True
```

```
>>> from sage.all import *
>>> p = Polyhedron(rays = [[Integer(1),Integer(0)],[-Integer(1),Integer(0)],
↪[Integer(0),Integer(1)],[Integer(1),Integer(1)]], vertices = [[-Integer(2),-
↪Integer(2)],[Integer(2),Integer(3)]])
>>> p.lines_list()
[[1, 0]]
```

(continues on next page)

(continued from previous page)

```
>>> p.lines_list() == [list(x) for x in p.line_generator()]
True
```

n_Hrepresentation()

Return the number of objects that make up the H-representation of the polyhedron.

OUTPUT: integer

EXAMPLES:

```
sage: p = polytopes.cross_polytope(4)
sage: p.n_Hrepresentation()
16
sage: p.n_Hrepresentation() == p.n_inequalities() + p.n_equations()
True
```

```
>>> from sage.all import *
>>> p = polytopes.cross_polytope(Integer(4))
>>> p.n_Hrepresentation()
16
>>> p.n_Hrepresentation() == p.n_inequalities() + p.n_equations()
True
```

n_Vrepresentation()

Return the number of objects that make up the V-representation of the polyhedron.

OUTPUT: integer

EXAMPLES:

```
sage: p = polytopes.simplex(4)
sage: p.n_Vrepresentation()
5
sage: p.n_Vrepresentation() == p.n_vertices() + p.n_rays() + p.n_lines()
True
```

```
>>> from sage.all import *
>>> p = polytopes.simplex(Integer(4))
>>> p.n_Vrepresentation()
5
>>> p.n_Vrepresentation() == p.n_vertices() + p.n_rays() + p.n_lines()
True
```

n_equations()

Return the number of equations. The representation will always be minimal, so the number of equations is the codimension of the polyhedron in the ambient space.

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[1,0,0],[0,1,0],[0,0,1]])
sage: p.n_equations()
1
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices = [[Integer(1),Integer(0),Integer(0)],[Integer(0),
↪Integer(1),Integer(0)],[Integer(0),Integer(0),Integer(1)]])
>>> p.n_equations()
1
```

n_facets()

Return the number of inequalities. The representation will always be minimal, so the number of inequalities is the number of facets of the polyhedron in the ambient space.

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[1,0,0],[0,1,0],[0,0,1]])
sage: p.n_inequalities()
3

sage: p = Polyhedron(vertices = [[t,t^2,t^3] for t in range(6)])
sage: p.n_facets()
8
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices = [[Integer(1),Integer(0),Integer(0)],[Integer(0),
↪Integer(1),Integer(0)],[Integer(0),Integer(0),Integer(1)]])
>>> p.n_inequalities()
3

>>> p = Polyhedron(vertices = [[t,t**Integer(2),t**Integer(3)] for t in␣
↪range(Integer(6))])
>>> p.n_facets()
8
```

n_inequalities()

Return the number of inequalities. The representation will always be minimal, so the number of inequalities is the number of facets of the polyhedron in the ambient space.

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[1,0,0],[0,1,0],[0,0,1]])
sage: p.n_inequalities()
3

sage: p = Polyhedron(vertices = [[t,t^2,t^3] for t in range(6)])
sage: p.n_facets()
8
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices = [[Integer(1),Integer(0),Integer(0)],[Integer(0),
↪Integer(1),Integer(0)],[Integer(0),Integer(0),Integer(1)]])
>>> p.n_inequalities()
3

>>> p = Polyhedron(vertices = [[t,t**Integer(2),t**Integer(3)] for t in␣
↪range(Integer(6))])
```

(continues on next page)

(continued from previous page)

```
>>> p.n_facets()
8
```

n_lines()

Return the number of lines. The representation will always be minimal.

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[0,0]], rays=[[0,1],[0,-1]])
sage: p.n_lines()
1
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices = [[Integer(0),Integer(0)]], rays=[[Integer(0),
↪Integer(1)],[Integer(0),-Integer(1)]])
>>> p.n_lines()
1
```

n_rays()

Return the number of rays. The representation will always be minimal.

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[1,0],[0,1]], rays=[[1,1]])
sage: p.n_rays()
1
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices = [[Integer(1),Integer(0)],[Integer(0),
↪Integer(1)]], rays=[[Integer(1),Integer(1)]])
>>> p.n_rays()
1
```

n_vertices()

Return the number of vertices. The representation will always be minimal.

Warning

If the polyhedron has lines, return the number of vertices in the `Vrepresentation`. As the represented polyhedron has no 0-dimensional faces (i.e. vertices), `n_vertices` corresponds to the number of k-faces, where k is the number of lines:

```
sage: P = Polyhedron(rays=[[1,0,0]],lines=[[0,1,0]])
sage: P.n_vertices()
1
sage: P.faces(0)
()
sage: P.f_vector()
(1, 0, 1, 1)

sage: P = Polyhedron(rays=[[1,0,0]],lines=[[0,1,0],[0,1,1]])
sage: P.n_vertices()
1
sage: P.f_vector()
(1, 0, 0, 1, 1)
```

```
>>> from sage.all import *
>>> P = Polyhedron(rays=[[Integer(1),Integer(0),Integer(0)]],
↪lines=[[Integer(0),Integer(1),Integer(0)]])
>>> P.n_vertices()
1
>>> P.faces(Integer(0))
()
>>> P.f_vector()
(1, 0, 1, 1)

>>> P = Polyhedron(rays=[[Integer(1),Integer(0),Integer(0)]],
↪lines=[[Integer(0),Integer(1),Integer(0)],[Integer(0),Integer(1),
↪Integer(1)]])
>>> P.n_vertices()
1
>>> P.f_vector()
(1, 0, 0, 1, 1)
```

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[1,0],[0,1],[1,1]], rays=[[1,1]])
sage: p.n_vertices()
2
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices = [[Integer(1),Integer(0)],[Integer(0),
↪Integer(1)],[Integer(1),Integer(1)]], rays=[[Integer(1),Integer(1)]])
>>> p.n_vertices()
2
```

ray_generator()

Return a generator for the rays of the polyhedron.

EXAMPLES:

```
sage: pi = Polyhedron(ieqs = [[1,1,0],[1,0,1]])
sage: pir = pi.ray_generator()
sage: [x.vector() for x in pir]
[(1, 0), (0, 1)]
```

```
>>> from sage.all import *
>>> pi = Polyhedron(ieqs = [[Integer(1),Integer(1),Integer(0)],[Integer(1),
↪Integer(0),Integer(1)]])
>>> pir = pi.ray_generator()
>>> [x.vector() for x in pir]
[(1, 0), (0, 1)]
```

rays()

Return a list of rays of the polyhedron.

OUTPUT: a tuple of rays

EXAMPLES:

```
sage: p = Polyhedron(ieqs = [[0,0,0,1],[0,0,1,0],[1,1,0,0]])
sage: p.rays()
(A ray in the direction (1, 0, 0),
 A ray in the direction (0, 1, 0),
 A ray in the direction (0, 0, 1))
```

```
>>> from sage.all import *
>>> p = Polyhedron(ieqs = [[Integer(0),Integer(0),Integer(0),Integer(1)],
↪[Integer(0),Integer(0),Integer(1),Integer(0)],[Integer(1),Integer(1),
↪Integer(0),Integer(0)]])
>>> p.rays()
(A ray in the direction (1, 0, 0),
 A ray in the direction (0, 1, 0),
 A ray in the direction (0, 0, 1))
```

rays_list()

Return a list of rays as coefficient lists.

> **Note**
>
> It is recommended to use *rays()* or *ray_generator()* instead to iterate over the list of *Ray* objects.

OUTPUT: list of rays as lists of coordinates

EXAMPLES:

```
sage: p = Polyhedron(ieqs = [[0,0,0,1],[0,0,1,0],[1,1,0,0]])
sage: p.rays_list()
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
sage: p.rays_list() == [list(r) for r in p.ray_generator()]
True
```

```
>>> from sage.all import *
>>> p = Polyhedron(ieqs = [[Integer(0),Integer(0),Integer(0),Integer(1)],
↪[Integer(0),Integer(0),Integer(1),Integer(0)],[Integer(1),Integer(1),
↪Integer(0),Integer(0)]])
>>> p.rays_list()
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
>>> p.rays_list() == [list(r) for r in p.ray_generator()]
True
```

vertex_generator()

Return a generator for the vertices of the polyhedron.

> **Warning**
>
> If the polyhedron has lines, return a generator for the vertices of the `Vrepresentation`. However, the represented polyhedron has no 0-dimensional faces (i.e. vertices):
>
> ```
> sage: P = Polyhedron(rays=[[1,0,0]],lines=[[0,1,0]])
> sage: list(P.vertex_generator())
> [A vertex at (0, 0, 0)]
> ```

```
sage: P.faces(0)
()
```

```
>>> from sage.all import *
>>> P = Polyhedron(rays=[[Integer(1),Integer(0),Integer(0)]],
↪lines=[[Integer(0),Integer(1),Integer(0)]])
>>> list(P.vertex_generator())
[A vertex at (0, 0, 0)]
>>> P.faces(Integer(0))
()
```

EXAMPLES:

```
sage: triangle = Polyhedron(vertices=[[1,0],[0,1],[1,1]])
sage: for v in triangle.vertex_generator(): print(v)
A vertex at (0, 1)
A vertex at (1, 0)
A vertex at (1, 1)
sage: v_gen = triangle.vertex_generator()
sage: next(v_gen)   # the first vertex
A vertex at (0, 1)
sage: next(v_gen)   # the second vertex
A vertex at (1, 0)
sage: next(v_gen)   # the third vertex
A vertex at (1, 1)
sage: try: next(v_gen)   # there are only three vertices
....: except StopIteration: print("STOP")
STOP
sage: type(v_gen)
<... 'generator'>
sage: [ v for v in triangle.vertex_generator() ]
[A vertex at (0, 1), A vertex at (1, 0), A vertex at (1, 1)]
```

```
>>> from sage.all import *
>>> triangle = Polyhedron(vertices=[[Integer(1),Integer(0)],[Integer(0),
↪Integer(1)],[Integer(1),Integer(1)]])
>>> for v in triangle.vertex_generator(): print(v)
A vertex at (0, 1)
A vertex at (1, 0)
A vertex at (1, 1)
>>> v_gen = triangle.vertex_generator()
>>> next(v_gen)   # the first vertex
A vertex at (0, 1)
>>> next(v_gen)   # the second vertex
A vertex at (1, 0)
>>> next(v_gen)   # the third vertex
A vertex at (1, 1)
>>> try: next(v_gen)   # there are only three vertices
... except StopIteration: print("STOP")
STOP
>>> type(v_gen)
<... 'generator'>
```

(continues on next page)

(continued from previous page)

```
>>> [ v for v in triangle.vertex_generator() ]
[A vertex at (0, 1), A vertex at (1, 0), A vertex at (1, 1)]
```

vertices()

Return all vertices of the polyhedron.

OUTPUT: a tuple of vertices

> **Warning**
>
> If the polyhedron has lines, return the vertices of the `Vrepresentation`. However, the represented polyhedron has no 0-dimensional faces (i.e. vertices):
>
> ```
> sage: P = Polyhedron(rays=[[1,0,0]],lines=[[0,1,0]])
> sage: P.vertices()
> (A vertex at (0, 0, 0),)
> sage: P.faces(0)
> ()
> ```
>
> ```
> >>> from sage.all import *
> >>> P = Polyhedron(rays=[[Integer(1),Integer(0),Integer(0)]],
> ↪lines=[[Integer(0),Integer(1),Integer(0)]])
> >>> P.vertices()
> (A vertex at (0, 0, 0),)
> >>> P.faces(Integer(0))
> ()
> ```

EXAMPLES:

```
sage: triangle = Polyhedron(vertices=[[1,0],[0,1],[1,1]])
sage: triangle.vertices()
(A vertex at (0, 1), A vertex at (1, 0), A vertex at (1, 1))
sage: a_simplex = Polyhedron(ieqs = [
....:          [0,1,0,0,0],[0,0,1,0,0],[0,0,0,1,0],[0,0,0,0,1]
....:      ], eqns = [[1,-1,-1,-1,-1]])
sage: a_simplex.vertices()
(A vertex at (1, 0, 0, 0), A vertex at (0, 1, 0, 0),
 A vertex at (0, 0, 1, 0), A vertex at (0, 0, 0, 1))
```

```
>>> from sage.all import *
>>> triangle = Polyhedron(vertices=[[Integer(1),Integer(0)],[Integer(0),
↪Integer(1)],[Integer(1),Integer(1)]])
>>> triangle.vertices()
(A vertex at (0, 1), A vertex at (1, 0), A vertex at (1, 1))
>>> a_simplex = Polyhedron(ieqs = [
...          [Integer(0),Integer(1),Integer(0),Integer(0),Integer(0)],
↪[Integer(0),Integer(0),Integer(1),Integer(0),Integer(0)],[Integer(0),
↪Integer(0),Integer(0),Integer(1),Integer(0)],[Integer(0),Integer(0),
↪Integer(0),Integer(0),Integer(1)]
...      ], eqns = [[Integer(1),-Integer(1),-Integer(1),-Integer(1),-
↪Integer(1)]])
```

(continues on next page)

(continued from previous page)

```
>>> a_simplex.vertices()
(A vertex at (1, 0, 0, 0), A vertex at (0, 1, 0, 0),
 A vertex at (0, 0, 1, 0), A vertex at (0, 0, 0, 1))
```

vertices_list()

Return a list of vertices of the polyhedron.

Note

It is recommended to use *vertex_generator()* instead to iterate over the list of *Vertex* objects.

Warning

If the polyhedron has lines, return the vertices of the Vrepresentation. However, the represented polyhedron has no 0-dimensional faces (i.e. vertices):

```
sage: P = Polyhedron(rays=[[1,0,0]],lines=[[0,1,0]])
sage: P.vertices_list()
[[0, 0, 0]]
sage: P.faces(0)
()
```

```
>>> from sage.all import *
>>> P = Polyhedron(rays=[[Integer(1),Integer(0),Integer(0)]],
↪lines=[[Integer(0),Integer(1),Integer(0)]])
>>> P.vertices_list()
[[0, 0, 0]]
>>> P.faces(Integer(0))
()
```

EXAMPLES:

```
sage: triangle = Polyhedron(vertices=[[1,0],[0,1],[1,1]])
sage: triangle.vertices_list()
[[0, 1], [1, 0], [1, 1]]
sage: a_simplex = Polyhedron(ieqs = [
....:          [0,1,0,0,0],[0,0,1,0,0],[0,0,0,1,0],[0,0,0,0,1]
....:      ], eqns = [[1,-1,-1,-1,-1]])
sage: a_simplex.vertices_list()
[[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]]
sage: a_simplex.vertices_list() == [list(v) for v in a_simplex.vertex_
↪generator()]
True
```

```
>>> from sage.all import *
>>> triangle = Polyhedron(vertices=[[Integer(1),Integer(0)],[Integer(0),
↪Integer(1)],[Integer(1),Integer(1)]])
>>> triangle.vertices_list()
[[0, 1], [1, 0], [1, 1]]
```

(continues on next page)

(continued from previous page)

```
>>> a_simplex = Polyhedron(ieqs = [
...           [Integer(0),Integer(1),Integer(0),Integer(0),Integer(0)],
↪[Integer(0),Integer(0),Integer(1),Integer(0),Integer(0)],[Integer(0),
↪Integer(0),Integer(0),Integer(1),Integer(0)],[Integer(0),Integer(0),
↪Integer(0),Integer(0),Integer(1)]
...       ], eqns = [[Integer(1),-Integer(1),-Integer(1),-Integer(1),-
↪Integer(1)]])
>>> a_simplex.vertices_list()
[[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]]
>>> a_simplex.vertices_list() == [list(v) for v in a_simplex.vertex_
↪generator()]
True
```

vertices_matrix(*base_ring=None*)

Return the coordinates of the vertices as the columns of a matrix.

INPUT:

- `base_ring` – a ring or `None` (default); the base ring of the returned matrix. If not specified, the base ring of the polyhedron is used.

OUTPUT:

A matrix over `base_ring` whose columns are the coordinates of the vertices. A `TypeError` is raised if the coordinates cannot be converted to `base_ring`.

Warning

If the polyhedron has lines, return the coordinates of the vertices of the `Vrepresentation`. However, the represented polyhedron has no 0-dimensional faces (i.e. vertices):

```
sage: P = Polyhedron(rays=[[1,0,0]],lines=[[0,1,0]])
sage: P.vertices_matrix()
[0]
[0]
[0]
sage: P.faces(0)
()
```

```
>>> from sage.all import *
>>> P = Polyhedron(rays=[[Integer(1),Integer(0),Integer(0)]],
↪lines=[[Integer(0),Integer(1),Integer(0)]])
>>> P.vertices_matrix()
[0]
[0]
[0]
>>> P.faces(Integer(0))
()
```

EXAMPLES:

```
sage: triangle = Polyhedron(vertices=[[1,0],[0,1],[1,1]])
sage: triangle.vertices_matrix()
```

(continues on next page)

(continued from previous page)

```
[0 1 1]
[1 0 1]
sage: (triangle/2).vertices_matrix()
[  0 1/2 1/2]
[1/2   0 1/2]
sage: (triangle/2).vertices_matrix(ZZ)
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

```
>>> from sage.all import *
>>> triangle = Polyhedron(vertices=[[Integer(1),Integer(0)],[Integer(0),
↪Integer(1)],[Integer(1),Integer(1)]])
>>> triangle.vertices_matrix()
[0 1 1]
[1 0 1]
>>> (triangle/Integer(2)).vertices_matrix()
[  0 1/2 1/2]
[1/2   0 1/2]
>>> (triangle/Integer(2)).vertices_matrix(ZZ)
Traceback (most recent call last):
...
TypeError: no conversion of this rational to integer
```

write_cdd_Hrepresentation(*filename*)

Export the polyhedron as a H-representation to a file.

INPUT:

- `filename` – the output file

See also

cdd_Hrepresentation() – return the H-representation of the polyhedron as a string.

EXAMPLES:

```
sage: from sage.misc.temporary_file import tmp_filename
sage: filename = tmp_filename(ext='.ext')
sage: polytopes.cube().write_cdd_Hrepresentation(filename)
```

```
>>> from sage.all import *
>>> from sage.misc.temporary_file import tmp_filename
>>> filename = tmp_filename(ext='.ext')
>>> polytopes.cube().write_cdd_Hrepresentation(filename)
```

write_cdd_Vrepresentation(*filename*)

Export the polyhedron as a V-representation to a file.

INPUT:

- `filename` – the output file

See also

cdd_Vrepresentation() – return the V-representation of the polyhedron as a string.

EXAMPLES:

```
sage: from sage.misc.temporary_file import tmp_filename
sage: filename = tmp_filename(ext='.ext')
sage: polytopes.cube().write_cdd_Vrepresentation(filename)
```

```
>>> from sage.all import *
>>> from sage.misc.temporary_file import tmp_filename
>>> filename = tmp_filename(ext='.ext')
>>> polytopes.cube().write_cdd_Vrepresentation(filename)
```

2.6.2 Base class for polyhedra: Implementation of the `ConvexSet_base` API

Define methods that exist for convex sets, but not constructions such as dilation or product.

class sage.geometry.polyhedron.base1.**Polyhedron_base1**(*parent*, *Vrep*, *Hrep*, *Vrep_minimal=None*, *Hrep_minimal=None*, *pref_rep=None*, *mutable=False*, ***kwds*)

Bases: *Polyhedron_base0*, *ConvexSet_closed*

Convex set methods for polyhedra, but not constructions such as dilation or product.

See *sage.geometry.polyhedron.base.Polyhedron_base*.

Hrepresentation_space()

Return the linear space containing the H-representation vectors.

OUTPUT: a free module over the base ring of dimension *ambient_dim()* + 1

EXAMPLES:

```
sage: poly_test = Polyhedron(vertices = [[1,0,0,0],[0,1,0,0]])
sage: poly_test.Hrepresentation_space()
Ambient free module of rank 5 over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> poly_test = Polyhedron(vertices = [[Integer(1),Integer(0),Integer(0),
↪Integer(0)],[Integer(0),Integer(1),Integer(0),Integer(0)]])
>>> poly_test.Hrepresentation_space()
Ambient free module of rank 5 over the principal ideal domain Integer Ring
```

Vrepresentation_space()

Return the ambient free module.

OUTPUT: a free module over the base ring of dimension *ambient_dim()*

EXAMPLES:

```
sage: poly_test = Polyhedron(vertices = [[1,0,0,0],[0,1,0,0]])
sage: poly_test.Vrepresentation_space()
Ambient free module of rank 4 over the principal ideal domain Integer Ring
```

(continues on next page)

(continued from previous page)

```
sage: poly_test.ambient_space() is poly_test.Vrepresentation_space()
True
```

```
>>> from sage.all import *
>>> poly_test = Polyhedron(vertices = [[Integer(1),Integer(0),Integer(0),
↪Integer(0)],[Integer(0),Integer(1),Integer(0),Integer(0)]])
>>> poly_test.Vrepresentation_space()
Ambient free module of rank 4 over the principal ideal domain Integer Ring
>>> poly_test.ambient_space() is poly_test.Vrepresentation_space()
True
```

a_maximal_chain()

Return a maximal chain of the face lattice in increasing order.

Subclasses must provide an implementation of this method.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.base1 import Polyhedron_base1
sage: P = polytopes.cube()
sage: Polyhedron_base1.a_maximal_chain
<abstract method a_maximal_chain at ...>
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.base1 import Polyhedron_base1
>>> P = polytopes.cube()
>>> Polyhedron_base1.a_maximal_chain
<abstract method a_maximal_chain at ...>
```

ambient(*base_field=None*)

Return the ambient vector space.

It is the ambient free module (*Vrepresentation_space()*) tensored with a field.

INPUT:

- `base_field` – a field (default: the fraction field of the base ring)

EXAMPLES:

```
sage: poly_test = Polyhedron(vertices = [[1,0,0,0],[0,1,0,0]])
sage: poly_test.ambient_vector_space()
Vector space of dimension 4 over Rational Field
sage: poly_test.ambient_vector_space() is poly_test.ambient()
True

sage: poly_test.ambient_vector_space(AA)                                    #␣
↪needs sage.rings.number_field
Vector space of dimension 4 over Algebraic Real Field
sage: poly_test.ambient_vector_space(RDF)
Vector space of dimension 4 over Real Double Field
sage: poly_test.ambient_vector_space(SR)                                    #␣
↪needs sage.symbolic
Vector space of dimension 4 over Symbolic Ring
```

```
>>> from sage.all import *
>>> poly_test = Polyhedron(vertices = [[Integer(1),Integer(0),Integer(0),
↪Integer(0)],[Integer(0),Integer(1),Integer(0),Integer(0)]])
>>> poly_test.ambient_vector_space()
Vector space of dimension 4 over Rational Field
>>> poly_test.ambient_vector_space() is poly_test.ambient()
True

>>> poly_test.ambient_vector_space(AA)                                     #␣
↪needs sage.rings.number_field
Vector space of dimension 4 over Algebraic Real Field
>>> poly_test.ambient_vector_space(RDF)
Vector space of dimension 4 over Real Double Field
>>> poly_test.ambient_vector_space(SR)                                     #␣
↪needs sage.symbolic
Vector space of dimension 4 over Symbolic Ring
```

ambient_dim()

Return the dimension of the ambient space.

EXAMPLES:

```
sage: poly_test = Polyhedron(vertices = [[1,0,0,0],[0,1,0,0]])
sage: poly_test.ambient_dim()
4
```

```
>>> from sage.all import *
>>> poly_test = Polyhedron(vertices = [[Integer(1),Integer(0),Integer(0),
↪Integer(0)],[Integer(0),Integer(1),Integer(0),Integer(0)]])
>>> poly_test.ambient_dim()
4
```

ambient_space()

Return the ambient free module.

OUTPUT: a free module over the base ring of dimension `ambient_dim()`

EXAMPLES:

```
sage: poly_test = Polyhedron(vertices = [[1,0,0,0],[0,1,0,0]])
sage: poly_test.Vrepresentation_space()
Ambient free module of rank 4 over the principal ideal domain Integer Ring
sage: poly_test.ambient_space() is poly_test.Vrepresentation_space()
True
```

```
>>> from sage.all import *
>>> poly_test = Polyhedron(vertices = [[Integer(1),Integer(0),Integer(0),
↪Integer(0)],[Integer(0),Integer(1),Integer(0),Integer(0)]])
>>> poly_test.Vrepresentation_space()
Ambient free module of rank 4 over the principal ideal domain Integer Ring
>>> poly_test.ambient_space() is poly_test.Vrepresentation_space()
True
```

ambient_vector_space(*base_field=None*)

Return the ambient vector space.

It is the ambient free module (*Vrepresentation_space()*) tensored with a field.

INPUT:

- `base_field` – a field (default: the fraction field of the base ring)

EXAMPLES:

```
sage: poly_test = Polyhedron(vertices = [[1,0,0,0],[0,1,0,0]])
sage: poly_test.ambient_vector_space()
Vector space of dimension 4 over Rational Field
sage: poly_test.ambient_vector_space() is poly_test.ambient()
True

sage: poly_test.ambient_vector_space(AA)                                    #␣
↪needs sage.rings.number_field
Vector space of dimension 4 over Algebraic Real Field
sage: poly_test.ambient_vector_space(RDF)
Vector space of dimension 4 over Real Double Field
sage: poly_test.ambient_vector_space(SR)                                    #␣
↪needs sage.symbolic
Vector space of dimension 4 over Symbolic Ring
```

```
>>> from sage.all import *
>>> poly_test = Polyhedron(vertices = [[Integer(1),Integer(0),Integer(0),
↪Integer(0)],[Integer(0),Integer(1),Integer(0),Integer(0)]])
>>> poly_test.ambient_vector_space()
Vector space of dimension 4 over Rational Field
>>> poly_test.ambient_vector_space() is poly_test.ambient()
True

>>> poly_test.ambient_vector_space(AA)                                    #␣
↪needs sage.rings.number_field
Vector space of dimension 4 over Algebraic Real Field
>>> poly_test.ambient_vector_space(RDF)
Vector space of dimension 4 over Real Double Field
>>> poly_test.ambient_vector_space(SR)                                    #␣
↪needs sage.symbolic
Vector space of dimension 4 over Symbolic Ring
```

an_affine_basis()

Return points in `self` that form a basis for the affine span of `self`.

This implementation of the method *an_affine_basis()* for polytopes guarantees the following:

- All points are vertices.
- The basis is obtained by considering a maximal chain of faces in the face lattice and picking for each cover relation one vertex that is in the difference. Thus this method is independent of the concrete realization of the polytope.

For unbounded polyhedra, the result may contain arbitrary points of `self`, not just vertices.

EXAMPLES:

```
sage: P = polytopes.cube()
sage: P.an_affine_basis()
[A vertex at (-1, -1, -1),
 A vertex at (1, -1, -1),
 A vertex at (1, -1, 1),
 A vertex at (1, 1, -1)]

sage: P = polytopes.permutahedron(5)
sage: P.an_affine_basis()
[A vertex at (1, 2, 3, 5, 4),
 A vertex at (2, 1, 3, 5, 4),
 A vertex at (1, 3, 2, 5, 4),
 A vertex at (4, 1, 3, 5, 2),
 A vertex at (4, 2, 5, 3, 1)]
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> P.an_affine_basis()
[A vertex at (-1, -1, -1),
 A vertex at (1, -1, -1),
 A vertex at (1, -1, 1),
 A vertex at (1, 1, -1)]

>>> P = polytopes.permutahedron(Integer(5))
>>> P.an_affine_basis()
[A vertex at (1, 2, 3, 5, 4),
 A vertex at (2, 1, 3, 5, 4),
 A vertex at (1, 3, 2, 5, 4),
 A vertex at (4, 1, 3, 5, 2),
 A vertex at (4, 2, 5, 3, 1)]
```

Unbounded polyhedra:

```
sage: p = Polyhedron(vertices=[(0, 0)], rays=[(1,0), (0,1)])
sage: p.an_affine_basis()
[A vertex at (0, 0), (1, 0), (0, 1)]
sage: p = Polyhedron(vertices=[(2, 1)], rays=[(1,0), (0,1)])
sage: p.an_affine_basis()
[A vertex at (2, 1), (3, 1), (2, 2)]
sage: p = Polyhedron(vertices=[(2, 1)], rays=[(1,0)], lines=[(0,1)])
sage: p.an_affine_basis()
[(2, 1), A vertex at (2, 0), (3, 0)]
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices=[(Integer(0), Integer(0))], rays=[(Integer(1),
↪Integer(0)), (Integer(0),Integer(1))])
>>> p.an_affine_basis()
[A vertex at (0, 0), (1, 0), (0, 1)]
>>> p = Polyhedron(vertices=[(Integer(2), Integer(1))], rays=[(Integer(1),
↪Integer(0)), (Integer(0),Integer(1))])
>>> p.an_affine_basis()
[A vertex at (2, 1), (3, 1), (2, 2)]
```

(continues on next page)

(continued from previous page)

```
>>> p = Polyhedron(vertices=[(Integer(2), Integer(1))], rays=[(Integer(1),
↪Integer(0))], lines=[(Integer(0),Integer(1))])
>>> p.an_affine_basis()
[(2, 1), A vertex at (2, 0), (3, 0)]
```

contains (*point*)

Test whether the polyhedron contains the given `point`.

> **See also**
>
> *interior_contains()*, *relative_interior_contains()*.

INPUT:

- `point` – coordinates of a point (an iterable)

OUTPUT: boolean

EXAMPLES:

```
sage: P = Polyhedron(vertices=[[1,1],[1,-1],[0,0]])
sage: P.contains( [1,0] )
True
sage: P.contains( P.center() )  # true for any convex set
True
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[[Integer(1),Integer(1)],[Integer(1),-Integer(1)],
↪[Integer(0),Integer(0)]])
>>> P.contains( [Integer(1),Integer(0)] )
True
>>> P.contains( P.center() )  # true for any convex set
True
```

As a shorthand, one may use the usual `in` operator:

```
sage: P.center() in P
True
sage: [-1,-1] in P
False
```

```
>>> from sage.all import *
>>> P.center() in P
True
>>> [-Integer(1),-Integer(1)] in P
False
```

The point need not have coordinates in the same field as the polyhedron:

```
sage: # needs sage.symbolic
sage: ray = Polyhedron(vertices=[(0,0)], rays=[(1,0)], base_ring=QQ)
sage: ray.contains([sqrt(2)/3,0])        # irrational coordinates are ok
```

(continues on next page)

(continued from previous page)

```
True
sage: a = var('a')
sage: ray.contains([a,0])                # a might be negative!
False
sage: assume(a>0)
sage: ray.contains([a,0])
True
sage: ray.contains(['hello', 'kitty'])   # no common ring for coordinates
False
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> ray = Polyhedron(vertices=[(Integer(0),Integer(0))], rays=[(Integer(1),
↪Integer(0))], base_ring=QQ)
>>> ray.contains([sqrt(Integer(2))/Integer(3),Integer(0)])        #␣
↪irrational coordinates are ok
True
>>> a = var('a')
>>> ray.contains([a,Integer(0)])                # a might be negative!
False
>>> assume(a>Integer(0))
>>> ray.contains([a,Integer(0)])
True
>>> ray.contains(['hello', 'kitty'])   # no common ring for coordinates
False
```

The empty polyhedron needs extra care, see Issue #10238:

```
sage: empty = Polyhedron(); empty
The empty polyhedron in ZZ^0
sage: empty.contains([])
False
sage: empty.contains([0])               # not a point in QQ^0
False
sage: full = Polyhedron(vertices=[()]); full
A 0-dimensional polyhedron in ZZ^0 defined as the convex hull of 1 vertex
sage: full.contains([])
True
sage: full.contains([0])
False
```

```
>>> from sage.all import *
>>> empty = Polyhedron(); empty
The empty polyhedron in ZZ^0
>>> empty.contains([])
False
>>> empty.contains([Integer(0)])               # not a point in QQ^0
False
>>> full = Polyhedron(vertices=[()]); full
A 0-dimensional polyhedron in ZZ^0 defined as the convex hull of 1 vertex
>>> full.contains([])
True
```

(continues on next page)

(continued from previous page)

```
>>> full.contains([Integer(0)])
False
```

dim()

Return the dimension of the polyhedron.

OUTPUT: -1 if the polyhedron is empty, otherwise a nonnegative integer

EXAMPLES:

```
sage: simplex = Polyhedron(vertices = [[1,0,0,0],[0,0,0,1],[0,1,0,0],[0,0,1,
↪0]])
sage: simplex.dim()
3
sage: simplex.ambient_dim()
4
```

```
>>> from sage.all import *
>>> simplex = Polyhedron(vertices = [[Integer(1),Integer(0),Integer(0),
↪Integer(0)],[Integer(0),Integer(0),Integer(0),Integer(1)],[Integer(0),
↪Integer(1),Integer(0),Integer(0)],[Integer(0),Integer(0),Integer(1),
↪Integer(0)]])
>>> simplex.dim()
3
>>> simplex.ambient_dim()
4
```

The empty set is a special case (Issue #12193):

```
sage: P1=Polyhedron(vertices=[[1,0,0],[0,1,0],[0,0,1]])
sage: P2=Polyhedron(vertices=[[2,0,0],[0,2,0],[0,0,2]])
sage: P12 = P1.intersection(P2)
sage: P12
The empty polyhedron in ZZ^3
sage: P12.dim()
-1
```

```
>>> from sage.all import *
>>> P1=Polyhedron(vertices=[[Integer(1),Integer(0),Integer(0)],[Integer(0),
↪Integer(1),Integer(0)],[Integer(0),Integer(0),Integer(1)]])
>>> P2=Polyhedron(vertices=[[Integer(2),Integer(0),Integer(0)],[Integer(0),
↪Integer(2),Integer(0)],[Integer(0),Integer(0),Integer(2)]])
>>> P12 = P1.intersection(P2)
>>> P12
The empty polyhedron in ZZ^3
>>> P12.dim()
-1
```

dimension()

Return the dimension of the polyhedron.

OUTPUT: -1 if the polyhedron is empty, otherwise a nonnegative integer

EXAMPLES:

```
sage: simplex = Polyhedron(vertices = [[1,0,0,0],[0,0,0,1],[0,1,0,0],[0,0,1,
↪0]])
sage: simplex.dim()
3
sage: simplex.ambient_dim()
4
```

```
>>> from sage.all import *
>>> simplex = Polyhedron(vertices = [[Integer(1),Integer(0),Integer(0),
↪Integer(0)],[Integer(0),Integer(0),Integer(0),Integer(1)],[Integer(0),
↪Integer(1),Integer(0),Integer(0)],[Integer(0),Integer(0),Integer(1),
↪Integer(0)]])
>>> simplex.dim()
3
>>> simplex.ambient_dim()
4
```

The empty set is a special case (Issue #12193):

```
sage: P1=Polyhedron(vertices=[[1,0,0],[0,1,0],[0,0,1]])
sage: P2=Polyhedron(vertices=[[2,0,0],[0,2,0],[0,0,2]])
sage: P12 = P1.intersection(P2)
sage: P12
The empty polyhedron in ZZ^3
sage: P12.dim()
-1
```

```
>>> from sage.all import *
>>> P1=Polyhedron(vertices=[[Integer(1),Integer(0),Integer(0)],[Integer(0),
↪Integer(1),Integer(0)],[Integer(0),Integer(0),Integer(1)]])
>>> P2=Polyhedron(vertices=[[Integer(2),Integer(0),Integer(0)],[Integer(0),
↪Integer(2),Integer(0)],[Integer(0),Integer(0),Integer(2)]])
>>> P12 = P1.intersection(P2)
>>> P12
The empty polyhedron in ZZ^3
>>> P12.dim()
-1
```

interior()

The interior of `self`.

OUTPUT:

- either an empty polyhedron or an instance of *RelativeInterior*

EXAMPLES:

If the polyhedron is full-dimensional, the result is the same as that of *relative_interior()*:

```
sage: P_full = Polyhedron(vertices=[[0,0],[1,1],[1,-1]])
sage: P_full.interior()
Relative interior of
 a 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 3 vertices
```

```
>>> from sage.all import *
>>> P_full = Polyhedron(vertices=[[Integer(0),Integer(0)],[Integer(1),
↪Integer(1)],[Integer(1),-Integer(1)]])
>>> P_full.interior()
Relative interior of
 a 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 3 vertices
```

If the polyhedron is of strictly smaller dimension than the ambient space, its interior is empty:

```
sage: P_lower = Polyhedron(vertices=[[0,1], [0,-1]])
sage: P_lower.interior()
The empty polyhedron in ZZ^2
```

```
>>> from sage.all import *
>>> P_lower = Polyhedron(vertices=[[Integer(0),Integer(1)], [Integer(0),-
↪Integer(1)]])
>>> P_lower.interior()
The empty polyhedron in ZZ^2
```

interior_contains(*point*)

Test whether the interior of the polyhedron contains the given `point`.

See also

contains(), *relative_interior_contains()*.

INPUT:

- `point` – coordinates of a point

OUTPUT: boolean

EXAMPLES:

```
sage: P = Polyhedron(vertices=[[0,0],[1,1],[1,-1]])
sage: P.contains( [1,0] )
True
sage: P.interior_contains( [1,0] )
False
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[[Integer(0),Integer(0)],[Integer(1),Integer(1)],
↪[Integer(1),-Integer(1)]])
>>> P.contains( [Integer(1),Integer(0)] )
True
>>> P.interior_contains( [Integer(1),Integer(0)] )
False
```

If the polyhedron is of strictly smaller dimension than the ambient space, its interior is empty:

```
sage: P = Polyhedron(vertices=[[0,1],[0,-1]])
sage: P.contains( [0,0] )
True
```

(continues on next page)

(continued from previous page)

```
sage: P.interior_contains( [0,0] )
False
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[[Integer(0),Integer(1)],[Integer(0),-
↪Integer(1)]])
>>> P.contains( [Integer(0),Integer(0)] )
True
>>> P.interior_contains( [Integer(0),Integer(0)] )
False
```

The empty polyhedron needs extra care, see Issue #10238:

```
sage: empty = Polyhedron(); empty
The empty polyhedron in ZZ^0
sage: empty.interior_contains([])
False
```

```
>>> from sage.all import *
>>> empty = Polyhedron(); empty
The empty polyhedron in ZZ^0
>>> empty.interior_contains([])
False
```

is_empty()

Test whether the polyhedron is the empty polyhedron.

OUTPUT: boolean

EXAMPLES:

```
sage: P = Polyhedron(vertices=[[1,0,0],[0,1,0],[0,0,1]]);  P
A 2-dimensional polyhedron in ZZ^3 defined as the convex hull of 3 vertices
sage: P.is_empty(), P.is_universe()
(False, False)

sage: Q = Polyhedron(vertices=());  Q
The empty polyhedron in ZZ^0
sage: Q.is_empty(), Q.is_universe()
(True, False)

sage: R = Polyhedron(lines=[(1,0),(0,1)]);  R
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex and␣
↪2 lines
sage: R.is_empty(), R.is_universe()
(False, True)
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[[Integer(1),Integer(0),Integer(0)],[Integer(0),
↪Integer(1),Integer(0)],[Integer(0),Integer(0),Integer(1)]]);  P
A 2-dimensional polyhedron in ZZ^3 defined as the convex hull of 3 vertices
>>> P.is_empty(), P.is_universe()
```

(continues on next page)

(continued from previous page)

```
(False, False)

>>> Q = Polyhedron(vertices=());  Q
The empty polyhedron in ZZ^0
>>> Q.is_empty(), Q.is_universe()
(True, False)

>>> R = Polyhedron(lines=[(Integer(1),Integer(0)),(Integer(0),Integer(1))]); ␣
↪R
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex and␣
↪2 lines
>>> R.is_empty(), R.is_universe()
(False, True)
```

is_relatively_open()

Return whether `self` is relatively open.

OUTPUT: boolean

EXAMPLES:

```
sage: P = Polyhedron(vertices=[(1,0), (-1,0)]); P
A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: P.is_relatively_open()
False

sage: P0 = Polyhedron(vertices=[[1, 2]]); P0
A 0-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex
sage: P0.is_relatively_open()
True

sage: Empty = Polyhedron(ambient_dim=2); Empty
The empty polyhedron in ZZ^2
sage: Empty.is_relatively_open()
True

sage: Line = Polyhedron(vertices=[(1, 1)], lines=[(1, 0)]); Line
A 1-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and␣
↪1 line
sage: Line.is_relatively_open()
True
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[(Integer(1),Integer(0)), (-Integer(1),
↪Integer(0))]); P
A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
>>> P.is_relatively_open()
False

>>> P0 = Polyhedron(vertices=[[Integer(1), Integer(2)]]); P0
A 0-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex
>>> P0.is_relatively_open()
True
```

(continues on next page)

(continued from previous page)

```
>>> Empty = Polyhedron(ambient_dim=Integer(2)); Empty
The empty polyhedron in ZZ^2
>>> Empty.is_relatively_open()
True

>>> Line = Polyhedron(vertices=[(Integer(1), Integer(1))], lines=[(Integer(1),
↪ Integer(0))]); Line
A 1-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and
↪1 line
>>> Line.is_relatively_open()
True
```

is_universe()

Test whether the polyhedron is the whole ambient space.

OUTPUT: boolean

EXAMPLES:

```
sage: P = Polyhedron(vertices=[[1,0,0],[0,1,0],[0,0,1]]);  P
A 2-dimensional polyhedron in ZZ^3 defined as the convex hull of 3 vertices
sage: P.is_empty(), P.is_universe()
(False, False)

sage: Q = Polyhedron(vertices=());  Q
The empty polyhedron in ZZ^0
sage: Q.is_empty(), Q.is_universe()
(True, False)

sage: R = Polyhedron(lines=[(1,0),(0,1)]);  R
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex and
↪2 lines
sage: R.is_empty(), R.is_universe()
(False, True)
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[[Integer(1),Integer(0),Integer(0)],[Integer(0),
↪Integer(1),Integer(0)],[Integer(0),Integer(0),Integer(1)]]);  P
A 2-dimensional polyhedron in ZZ^3 defined as the convex hull of 3 vertices
>>> P.is_empty(), P.is_universe()
(False, False)

>>> Q = Polyhedron(vertices=());  Q
The empty polyhedron in ZZ^0
>>> Q.is_empty(), Q.is_universe()
(True, False)

>>> R = Polyhedron(lines=[(Integer(1),Integer(0)),(Integer(0),Integer(1))]); 
↪R
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex and
↪2 lines
>>> R.is_empty(), R.is_universe()
```

(continues on next page)

(continued from previous page)

```
(False, True)
```

relative_interior()

Return the relative interior of `self`.

EXAMPLES:

```
sage: P = Polyhedron(vertices=[(1,0), (-1,0)])
sage: ri_P = P.relative_interior(); ri_P
Relative interior of
 a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: (0, 0) in ri_P
True
sage: (1, 0) in ri_P
False

sage: P0 = Polyhedron(vertices=[[1, 2]])
sage: P0.relative_interior() is P0
True

sage: Empty = Polyhedron(ambient_dim=2)
sage: Empty.relative_interior() is Empty
True

sage: Line = Polyhedron(vertices=[(1, 1)], lines=[(1, 0)])
sage: Line.relative_interior() is Line
True
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[(Integer(1),Integer(0)), (-Integer(1),
↪Integer(0))])
>>> ri_P = P.relative_interior(); ri_P
Relative interior of
 a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
>>> (Integer(0), Integer(0)) in ri_P
True
>>> (Integer(1), Integer(0)) in ri_P
False

>>> P0 = Polyhedron(vertices=[[Integer(1), Integer(2)]])
>>> P0.relative_interior() is P0
True

>>> Empty = Polyhedron(ambient_dim=Integer(2))
>>> Empty.relative_interior() is Empty
True

>>> Line = Polyhedron(vertices=[(Integer(1), Integer(1))], lines=[(Integer(1),
↪ Integer(0))])
>>> Line.relative_interior() is Line
True
```

relative_interior_contains(*point*)

Test whether the relative interior of the polyhedron contains the given `point`.

See also

contains(), *interior_contains()*.

INPUT:

- `point` – coordinates of a point

OUTPUT: boolean

EXAMPLES:

```
sage: P = Polyhedron(vertices=[(1,0), (-1,0)])
sage: P.contains( (0,0) )
True
sage: P.interior_contains( (0,0) )
False
sage: P.relative_interior_contains( (0,0) )
True
sage: P.relative_interior_contains( (1,0) )
False
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[(Integer(1),Integer(0)), (-Integer(1),
→Integer(0))])
>>> P.contains( (Integer(0),Integer(0)) )
True
>>> P.interior_contains( (Integer(0),Integer(0)) )
False
>>> P.relative_interior_contains( (Integer(0),Integer(0)) )
True
>>> P.relative_interior_contains( (Integer(1),Integer(0)) )
False
```

The empty polyhedron needs extra care, see Issue #10238:

```
sage: empty = Polyhedron(); empty
The empty polyhedron in ZZ^0
sage: empty.relative_interior_contains([])
False
```

```
>>> from sage.all import *
>>> empty = Polyhedron(); empty
The empty polyhedron in ZZ^0
>>> empty.relative_interior_contains([])
False
```

representative_point()

Return a "generic" point.

See also

sage.geometry.polyhedron.base.Polyhedron_base.center().

OUTPUT:

A point as a coordinate vector. The point is chosen to be interior if possible. If the polyhedron is not full-dimensional, the point is in the relative interior. If the polyhedron is zero-dimensional, its single point is returned.

EXAMPLES:

```
sage: p = Polyhedron(vertices=[(3,2)], rays=[(1,-1)])
sage: p.representative_point()
(4, 1)
sage: p.center()
(3, 2)

sage: Polyhedron(vertices=[(3,2)]).representative_point()
(3, 2)
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices=[(Integer(3),Integer(2))], rays=[(Integer(1),-
↪Integer(1))])
>>> p.representative_point()
(4, 1)
>>> p.center()
(3, 2)

>>> Polyhedron(vertices=[(Integer(3),Integer(2))]).representative_point()
(3, 2)
```

2.6.3 Base class for polyhedra: Methods related to lattice points

class sage.geometry.polyhedron.base2.**Polyhedron_base2**(*parent*, *Vrep*, *Hrep*, *Vrep_minimal=None*, *Hrep_minimal=None*, *pref_rep=None*, *mutable=False*, ***kwds*)

Bases: *Polyhedron_base1*

Methods related to lattice points.

See *sage.geometry.polyhedron.base.Polyhedron_base*.

generating_function_of_integral_points(***kwds*)

Return the multivariate generating function of the integral points of this polyhedron.

To be precise, this returns

$$\sum_{(r_0,\dots,r_{d-1})\in \mathit{polyhedron}\cap \mathbf{Z}^d} y_0^{r_0}\dots y_{d-1}^{r_{d-1}}.$$

This calls *generating_function_of_integral_points()*, so have a look at the documentation and examples there.

INPUT:

The following keyword arguments are passed to *generating_function_of_integral_points()*:

- `split` – boolean (default: `False`) or list
  - `split=False` computes the generating function directly, without any splitting.
  - When `split` is a list of disjoint polyhedra, then for each of these polyhedra, this polyhedron is intersected with it, its generating function computed and all these generating functions are summed up.
  - `split=True` splits into $d!$ disjoint polyhedra.
- `result_as_tuple` – (default: `None`) a boolean or `None`

  This specifies whether the output is a (partial) factorization (`result_as_tuple=False`) or a sum of such (partial) factorizations (`result_as_tuple=True`). By default (`result_as_tuple=None`), this is automatically determined. If the output is a sum, it is represented as a tuple whose entries are the summands.
- `indices` – (default: `None`) a list or tuple

  If this is `None`, this is automatically determined.
- `name` – (default: `'y'`) a string

  The variable names of the Laurent polynomial ring of the output are this string followed by an integer.
- `names` – list or tuple of names (strings), or a comma separated string

  `name` is extracted from `names`, therefore `names` has to contain exactly one variable name, and `name` and``names`` cannot be specified both at the same time.
- `Factorization_sort` (default: `False`) and `Factorization_simplify` (default: `True`) – booleans

  These are passed on to `sage.structure.factorization.Factorization` when creating the result.
- `sort_factors` – boolean (default: `False`)

  If set, then the factors of the output are sorted such that the numerator is first and only then all factors of the denominator. It is ensured that the sorting is always the same; use this for doctesting.

OUTPUT:

The generating function as a (partial) `Factorization` of the result whose factors are Laurent polynomials

The result might be a tuple of such factorizations (depending on the parameter `result_as_tuple`) as well.

Note

At the moment, only polyhedra with nonnegative coordinates (i.e. a polyhedron in the nonnegative orthant) are handled.

EXAMPLES:

```
sage: # needs sage.combinat
sage: P2 = (Polyhedron(ieqs=[(0, 0, 0, 1), (0, 0, 1, 0), (0, 1, 0, -1)]),
....:       Polyhedron(ieqs=[(0, -1, 0, 1), (0, 1, 0, 0), (0, 0, 1, 0)]))
sage: P2[0].generating_function_of_integral_points(sort_factors=True)
1 * (-y0 + 1)^-1 * (-y1 + 1)^-1 * (-y0*y2 + 1)^-1
sage: P2[1].generating_function_of_integral_points(sort_factors=True)
1 * (-y1 + 1)^-1 * (-y2 + 1)^-1 * (-y0*y2 + 1)^-1
```

(continues on next page)

(continued from previous page)

```
sage: (P2[0] & P2[1]).Hrepresentation()
(An equation (1, 0, -1) x + 0 == 0,
 An inequality (1, 0, 0) x + 0 >= 0,
 An inequality (0, 1, 0) x + 0 >= 0)
sage: (P2[0] & P2[1]).generating_function_of_integral_points(sort_
↪factors=True)
1 * (-y1 + 1)^-1 * (-y0*y2 + 1)^-1
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> P2 = (Polyhedron(ieqs=[(Integer(0), Integer(0), Integer(0), Integer(1)),␣
↪(Integer(0), Integer(0), Integer(1), Integer(0)), (Integer(0), Integer(1),␣
↪Integer(0), -Integer(1))]),
...       Polyhedron(ieqs=[(Integer(0), -Integer(1), Integer(0), Integer(1)),␣
↪(Integer(0), Integer(1), Integer(0), Integer(0)), (Integer(0), Integer(0),␣
↪Integer(1), Integer(0))]))
>>> P2[Integer(0)].generating_function_of_integral_points(sort_factors=True)
1 * (-y0 + 1)^-1 * (-y1 + 1)^-1 * (-y0*y2 + 1)^-1
>>> P2[Integer(1)].generating_function_of_integral_points(sort_factors=True)
1 * (-y1 + 1)^-1 * (-y2 + 1)^-1 * (-y0*y2 + 1)^-1
>>> (P2[Integer(0)] & P2[Integer(1)]).Hrepresentation()
(An equation (1, 0, -1) x + 0 == 0,
 An inequality (1, 0, 0) x + 0 >= 0,
 An inequality (0, 1, 0) x + 0 >= 0)
>>> (P2[Integer(0)] & P2[Integer(1)]).generating_function_of_integral_
↪points(sort_factors=True)
1 * (-y1 + 1)^-1 * (-y0*y2 + 1)^-1
```

The number of integer partitions $1 \leq r_0 \leq r_1 \leq r_2 \leq r_3 \leq r_4$:

```
sage: # needs sage.combinat
sage: P = Polyhedron(ieqs=[(-1, 1, 0, 0, 0, 0), (0, -1, 1, 0, 0, 0),
....:                      (0, 0, -1, 1, 0, 0), (0, 0, 0, -1, 1, 0),
....:                      (0, 0, 0, 0, -1, 1)])
sage: f = P.generating_function_of_integral_points(sort_factors=True); f
y0*y1*y2*y3*y4 * (-y4 + 1)^-1 * (-y3*y4 + 1)^-1 * (-y2*y3*y4 + 1)^-1 *
(-y1*y2*y3*y4 + 1)^-1 * (-y0*y1*y2*y3*y4 + 1)^-1
sage: f = f.value()
sage: P.<z> = PowerSeriesRing(ZZ)
sage: c = f.subs({y: z for y in f.parent().gens()}); c
z^5 + z^6 + 2*z^7 + 3*z^8 + 5*z^9 + 7*z^10 + 10*z^11 + 13*z^12 + 18*z^13 +
23*z^14 + 30*z^15 + 37*z^16 + 47*z^17 + 57*z^18 + 70*z^19 + 84*z^20 +
101*z^21 + 119*z^22 + 141*z^23 + 164*z^24 + O(z^25)
sage: ([Partitions(k, length=5).cardinality() for k in range(5,20)] ==
....:     c.truncate().coefficients(sparse=False)[5:20])
True
```

```
>>> from sage.all import *
>>> # needs sage.combinat
>>> P = Polyhedron(ieqs=[(-Integer(1), Integer(1), Integer(0), Integer(0),␣
↪Integer(0), Integer(0)), (Integer(0), -Integer(1), Integer(1), Integer(0),␣
↪Integer(0), Integer(0)),
```

(continues on next page)

(continued from previous page)

```
...                          (Integer(0), Integer(0), -Integer(1), Integer(1),␣
↪Integer(0), Integer(0)), (Integer(0), Integer(0), Integer(0), -Integer(1),␣
↪Integer(1), Integer(0)),
...                          (Integer(0), Integer(0), Integer(0), Integer(0), -
↪Integer(1), Integer(1))])
>>> f = P.generating_function_of_integral_points(sort_factors=True); f
y0*y1*y2*y3*y4 * (-y4 + 1)^-1 * (-y3*y4 + 1)^-1 * (-y2*y3*y4 + 1)^-1 *
(-y1*y2*y3*y4 + 1)^-1 * (-y0*y1*y2*y3*y4 + 1)^-1
>>> f = f.value()
>>> P = PowerSeriesRing(ZZ, names=('z',)); (z,) = P._first_ngens(1)
>>> c = f.subs({y: z for y in f.parent().gens()}); c
z^5 + z^6 + 2*z^7 + 3*z^8 + 5*z^9 + 7*z^10 + 10*z^11 + 13*z^12 + 18*z^13 +
23*z^14 + 30*z^15 + 37*z^16 + 47*z^17 + 57*z^18 + 70*z^19 + 84*z^20 +
101*z^21 + 119*z^22 + 141*z^23 + 164*z^24 + O(z^25)
>>> ([Partitions(k, length=Integer(5)).cardinality() for k in␣
↪range(Integer(5),Integer(20))] ==
...     c.truncate().coefficients(sparse=False)[Integer(5):Integer(20)])
True
```

See also

More examples can be found at *generating_function_of_integral_points()*.

get_integral_point(*index*, ***kwds*)

Return the index-th integral point in this polyhedron.

This is equivalent to `sorted(self.integral_points())[index]`. However, so long as *integral_points_count()* does not need to enumerate all integral points, neither does this method. Hence it can be significantly faster. If the polyhedron is not compact, a `ValueError` is raised.

INPUT:

- `index` – integer; the index of the integral point to be found. If this is not in [0, `self.integral_point_count()`), an `IndexError` is raised.
- `**kwds` – optional keyword parameters that are passed to *integral_points_count()*

ALGORITHM:

The function computes each of the components of the requested point in turn. To compute x_i, the i-th component, it bisects the upper and lower bounds on x_i given by the bounding box. At each bisection, it uses *integral_points_count()* to determine on which side of the bisecting hyperplane the requested point lies.

See also

integral_points_count().

EXAMPLES:

```
sage: P = Polyhedron(vertices=[(-1,-1),(1,0),(1,1),(0,1)])
sage: P.get_integral_point(1)
(0, 0)
```

(continues on next page)

(continued from previous page)

```
sage: P.get_integral_point(4)
(1, 1)
sage: sorted(P.integral_points())
[(-1, -1), (0, 0), (0, 1), (1, 0), (1, 1)]
sage: P.get_integral_point(5)
Traceback (most recent call last):
...
IndexError: ...

sage: Q = Polyhedron([(1,3), (2, 7), (9, 77)])
sage: [Q.get_integral_point(i) for i in range(Q.integral_points_count())] ==␣
↪sorted(Q.integral_points())
True
sage: Q.get_integral_point(0, explicit_enumeration_threshold=0, triangulation=
↪'cddlib')  # optional - latte_int
(1, 3)
sage: Q.get_integral_point(0, explicit_enumeration_threshold=0, triangulation=
↪'cddlib', foo=True)  # optional - latte_int
Traceback (most recent call last):
...
RuntimeError: ...

sage: R = Polyhedron(vertices=[[1/2, 1/3]], rays=[[1, 1]])
sage: R.get_integral_point(0)
Traceback (most recent call last):
...
ValueError: ...
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[(-Integer(1),-Integer(1)),(Integer(1),
↪Integer(0)),(Integer(1),Integer(1)),(Integer(0),Integer(1))])
>>> P.get_integral_point(Integer(1))
(0, 0)
>>> P.get_integral_point(Integer(4))
(1, 1)
>>> sorted(P.integral_points())
[(-1, -1), (0, 0), (0, 1), (1, 0), (1, 1)]
>>> P.get_integral_point(Integer(5))
Traceback (most recent call last):
...
IndexError: ...

>>> Q = Polyhedron([(Integer(1),Integer(3)), (Integer(2), Integer(7)),␣
↪(Integer(9), Integer(77))])
>>> [Q.get_integral_point(i) for i in range(Q.integral_points_count())] ==␣
↪sorted(Q.integral_points())
True
>>> Q.get_integral_point(Integer(0), explicit_enumeration_
↪threshold=Integer(0), triangulation='cddlib')  # optional - latte_int
(1, 3)
>>> Q.get_integral_point(Integer(0), explicit_enumeration_
↪threshold=Integer(0), triangulation='cddlib', foo=True)  # optional - latte_
```

(continues on next page)

(continued from previous page)

```
↪int
Traceback (most recent call last):
...
RuntimeError: ...

>>> R = Polyhedron(vertices=[[Integer(1)/Integer(2), Integer(1)/Integer(3)]],␣
↪rays=[[Integer(1), Integer(1)]])
>>> R.get_integral_point(Integer(0))
Traceback (most recent call last):
...
ValueError: ...
```

h_star_vector()

Return the h^*-vector of the lattice polytope.

The h^*-vector records the coefficients of the polynomial in the numerator of the Ehrhart series of a lattice polytope.

INPUT:

- `self` – a lattice polytope

OUTPUT:

A list whose entries give the h^*-vector.

> **Note**
>
> The backend of `self` should be `'normaliz'`. This function depends on Normaliz (i.e. the `'pynormaliz'` optional package). See the Normaliz documentation for further details.

EXAMPLES:

The h^*-vector of a unimodular simplex S (a simplex with volume = $\frac{1}{dim(S)!}$) is always 1. Here we test this on simplices up to dimension 3:

```
sage: # optional - pynormaliz
sage: s1 = polytopes.simplex(1,backend='normaliz')
sage: s2 = polytopes.simplex(2,backend='normaliz')
sage: s3 = polytopes.simplex(3,backend='normaliz')
sage: [s1.h_star_vector(), s2.h_star_vector(), s3.h_star_vector()]
[[1], [1], [1]]
```

```
>>> from sage.all import *
>>> # optional - pynormaliz
>>> s1 = polytopes.simplex(Integer(1),backend='normaliz')
>>> s2 = polytopes.simplex(Integer(2),backend='normaliz')
>>> s3 = polytopes.simplex(Integer(3),backend='normaliz')
>>> [s1.h_star_vector(), s2.h_star_vector(), s3.h_star_vector()]
[[1], [1], [1]]
```

For a less trivial example, we compute the h^*-vector of the $0/1$-cube, which has the Eulerian numbers $(3, i)$ for $i \in [0, 2]$ as an h^*-vector:

```
sage: cube = polytopes.cube(intervals='zero_one', backend='normaliz')    #␣
↪optional - pynormaliz
sage: cube.h_star_vector()                                               #␣
↪optional - pynormaliz
[1, 4, 1]
sage: from sage.combinat.combinat import eulerian_number
sage: [eulerian_number(3,i) for i in range(3)]
[1, 4, 1]
```

```
>>> from sage.all import *
>>> cube = polytopes.cube(intervals='zero_one', backend='normaliz')    #␣
↪optional - pynormaliz
>>> cube.h_star_vector()                                               #␣
↪optional - pynormaliz
[1, 4, 1]
>>> from sage.combinat.combinat import eulerian_number
>>> [eulerian_number(Integer(3),i) for i in range(Integer(3))]
[1, 4, 1]
```

integral_points(*threshold=100000*)

Return the integral points in the polyhedron.

Uses either the naive algorithm (iterate over a rectangular bounding box) or triangulation + Smith form.

INPUT:

- `threshold` – integer (default: 100000); use the naive algorithm as long as the bounding box is smaller than this

OUTPUT:

The list of integral points in the polyhedron. If the polyhedron is not compact, a `ValueError` is raised.

EXAMPLES:

```
sage: Polyhedron(vertices=[(-1,-1),(1,0),(1,1),(0,1)]).integral_points()
((-1, -1), (0, 0), (0, 1), (1, 0), (1, 1))

sage: simplex = Polyhedron([(1,2,3), (2,3,7), (-2,-3,-11)])
sage: simplex.integral_points()
((-2, -3, -11), (0, 0, -2), (1, 2, 3), (2, 3, 7))
```

```
>>> from sage.all import *
>>> Polyhedron(vertices=[(-Integer(1),-Integer(1)),(Integer(1),Integer(0)),
↪(Integer(1),Integer(1)),(Integer(0),Integer(1))]).integral_points()
((-1, -1), (0, 0), (0, 1), (1, 0), (1, 1))

>>> simplex = Polyhedron([(Integer(1),Integer(2),Integer(3)), (Integer(2),
↪Integer(3),Integer(7)), (-Integer(2),-Integer(3),-Integer(11))])
>>> simplex.integral_points()
((-2, -3, -11), (0, 0, -2), (1, 2, 3), (2, 3, 7))
```

The polyhedron need not be full-dimensional:

```
sage: simplex = Polyhedron([(1,2,3,5), (2,3,7,5), (-2,-3,-11,5)])
sage: simplex.integral_points()
((-2, -3, -11, 5), (0, 0, -2, 5), (1, 2, 3, 5), (2, 3, 7, 5))

sage: point = Polyhedron([(2,3,7)])
sage: point.integral_points()
((2, 3, 7),)

sage: empty = Polyhedron()
sage: empty.integral_points()
()
```

```
>>> from sage.all import *
>>> simplex = Polyhedron([(Integer(1),Integer(2),Integer(3),Integer(5)),␣
↪(Integer(2),Integer(3),Integer(7),Integer(5)), (-Integer(2),-Integer(3),-
↪Integer(11),Integer(5))])
>>> simplex.integral_points()
((-2, -3, -11, 5), (0, 0, -2, 5), (1, 2, 3, 5), (2, 3, 7, 5))

>>> point = Polyhedron([(Integer(2),Integer(3),Integer(7))])
>>> point.integral_points()
((2, 3, 7),)

>>> empty = Polyhedron()
>>> empty.integral_points()
()
```

Here is a simplex where the naive algorithm of running over all points in a rectangular bounding box no longer works fast enough:

```
sage: v = [(1,0,7,-1), (-2,-2,4,-3), (-1,-1,-1,4), (2,9,0,-5), (-2,-1,5,1)]
sage: simplex = Polyhedron(v); simplex
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 5 vertices
sage: len(simplex.integral_points())
49
```

```
>>> from sage.all import *
>>> v = [(Integer(1),Integer(0),Integer(7),-Integer(1)), (-Integer(2),-
↪Integer(2),Integer(4),-Integer(3)), (-Integer(1),-Integer(1),-Integer(1),
↪Integer(4)), (Integer(2),Integer(9),Integer(0),-Integer(5)), (-Integer(2),-
↪Integer(1),Integer(5),Integer(1))]
>>> simplex = Polyhedron(v); simplex
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 5 vertices
>>> len(simplex.integral_points())
49
```

A case where rounding in the right direction goes a long way:

```
sage: P = 1/10*polytopes.hypercube(14, backend='field')
sage: P.integral_points()
((0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0),)
```

```
>>> from sage.all import *
>>> P = Integer(1)/Integer(10)*polytopes.hypercube(Integer(14), backend='field
↪')
>>> P.integral_points()
((0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0),)
```

Finally, the 3-d reflexive polytope number 4078:

```
sage: v = [(1,0,0), (0,1,0), (0,0,1), (0,0,-1), (0,-2,1),
....:      (-1,2,-1), (-1,2,-2), (-1,1,-2), (-1,-1,2), (-1,-3,2)]
sage: P = Polyhedron(v)
sage: pts1 = P.integral_points()                     # Sage's own code
sage: all(P.contains(p) for p in pts1)
True
sage: pts2 = LatticePolytope(v).points()                                 #␣
↪needs palp
sage: for p in pts1: p.set_immutable()
sage: set(pts1) == set(pts2)                                             #␣
↪needs palp
True

sage: timeit('Polyhedron(v).integral_points()')   # not tested - random
625 loops, best of 3: 1.41 ms per loop
sage: timeit('LatticePolytope(v).points()')       # not tested - random
25 loops, best of 3: 17.2 ms per loop
```

```
>>> from sage.all import *
>>> v = [(Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0)), (Integer(0),Integer(0),Integer(1)), (Integer(0),Integer(0),-
↪Integer(1)), (Integer(0),-Integer(2),Integer(1)),
...      (-Integer(1),Integer(2),-Integer(1)), (-Integer(1),Integer(2),-
↪Integer(2)), (-Integer(1),Integer(1),-Integer(2)), (-Integer(1),-Integer(1),
↪Integer(2)), (-Integer(1),-Integer(3),Integer(2))]
>>> P = Polyhedron(v)
>>> pts1 = P.integral_points()                     # Sage's own code
>>> all(P.contains(p) for p in pts1)
True
>>> pts2 = LatticePolytope(v).points()                                 #␣
↪needs palp
>>> for p in pts1: p.set_immutable()
>>> set(pts1) == set(pts2)                                             #␣
↪needs palp
True

>>> timeit('Polyhedron(v).integral_points()')   # not tested - random
625 loops, best of 3: 1.41 ms per loop
>>> timeit('LatticePolytope(v).points()')       # not tested - random
25 loops, best of 3: 17.2 ms per loop
```

integral_points_count(***kwds*)

Return the number of integral points in the polyhedron.

This generic version of this method simply calls `integral_points()`.

EXAMPLES:

```
sage: P = polytopes.cube()
sage: P.integral_points_count()
27
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> P.integral_points_count()
27
```

We shrink the polyhedron a little bit:

```
sage: Q = P*(8/9)
sage: Q.integral_points_count()
1
```

```
>>> from sage.all import *
>>> Q = P*(Integer(8)/Integer(9))
>>> Q.integral_points_count()
1
```

Same for a polyhedron whose coordinates are not rationals. Note that the answer is an integer even though there are no guarantees for exactness:

```
sage: Q = P*RDF(8/9)
sage: Q.integral_points_count()
1
```

```
>>> from sage.all import *
>>> Q = P*RDF(Integer(8)/Integer(9))
>>> Q.integral_points_count()
1
```

Unbounded polyhedra (with or without lattice points) are not supported:

```
sage: P = Polyhedron(vertices=[[1/2, 1/3]], rays=[[1, 1]])
sage: P.integral_points_count()
Traceback (most recent call last):
...
NotImplementedError: ...
sage: P = Polyhedron(vertices=[[1, 1]], rays=[[1, 1]])
sage: P.integral_points_count()
Traceback (most recent call last):
...
NotImplementedError: ...
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[[Integer(1)/Integer(2), Integer(1)/Integer(3)]],␣
↪rays=[[Integer(1), Integer(1)]])
>>> P.integral_points_count()
Traceback (most recent call last):
...
```

(continues on next page)

(continued from previous page)

```
NotImplementedError: ...
>>> P = Polyhedron(vertices=[[Integer(1), Integer(1)]], rays=[[Integer(1),␣
↪Integer(1)]])
>>> P.integral_points_count()
Traceback (most recent call last):
...
NotImplementedError: ...
```

is_lattice_polytope()

Return whether the polyhedron is a lattice polytope.

OUTPUT:

`True` if the polyhedron is compact and has only integral vertices, `False` otherwise.

EXAMPLES:

```
sage: polytopes.cross_polytope(3).is_lattice_polytope()
True
sage: polytopes.regular_polygon(5).is_lattice_polytope()                    #␣
↪needs sage.rings.number_field
False
```

```
>>> from sage.all import *
>>> polytopes.cross_polytope(Integer(3)).is_lattice_polytope()
True
>>> polytopes.regular_polygon(Integer(5)).is_lattice_polytope()              ␣
↪     # needs sage.rings.number_field
False
```

lattice_polytope(*envelope=False*)

Return an encompassing lattice polytope.

INPUT:

- `envelope` – boolean (default: `False`); if the polyhedron has non-integral vertices, this option decides whether to return a strictly larger lattice polytope or raise a `ValueError`. This option has no effect if the polyhedron has already integral vertices.

OUTPUT:

A *LatticePolytope*. If the polyhedron is compact and has integral vertices, the lattice polytope equals the polyhedron. If the polyhedron is compact but has at least one non-integral vertex, a strictly larger lattice polytope is returned.

If the polyhedron is not compact, a `NotImplementedError` is raised.

If the polyhedron is not integral and `envelope=False`, a `ValueError` is raised.

ALGORITHM:

For each non-integral vertex, a bounding box of integral points is added and the convex hull of these integral points is returned.

EXAMPLES:

First, a polyhedron with integral vertices:

```
sage: P = Polyhedron(vertices=[(1, 0), (0, 1), (-1, 0), (0, -1)])
sage: lp = P.lattice_polytope(); lp
2-d reflexive polytope... in 2-d lattice M
sage: lp                                                              #␣
↪optional - polytopes_db, needs palp
2-d reflexive polytope #3 in 2-d lattice M
sage: lp.vertices()
M(-1,  0),
M( 0, -1),
M( 0,  1),
M( 1,  0)
in 2-d lattice M
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[(Integer(1), Integer(0)), (Integer(0),␣
↪Integer(1)), (-Integer(1), Integer(0)), (Integer(0), -Integer(1))])
>>> lp = P.lattice_polytope(); lp
2-d reflexive polytope... in 2-d lattice M
>>> lp                                                        # optional -
↪ polytopes_db, needs palp
2-d reflexive polytope #3 in 2-d lattice M
>>> lp.vertices()
M(-1,  0),
M( 0, -1),
M( 0,  1),
M( 1,  0)
in 2-d lattice M
```

Here is a polyhedron with non-integral vertices:

```
sage: P = Polyhedron( vertices = [(1/2, 1/2), (0, 1), (-1, 0), (0, -1)])
sage: lp = P.lattice_polytope()
Traceback (most recent call last):
...
ValueError: Some vertices are not integral. You probably want
to add the argument "envelope=True" to compute an enveloping
lattice polytope.
sage: lp = P.lattice_polytope(True)
sage: lp                                                              #␣
↪optional - polytopes_db, needs palp
2-d reflexive polytope #5 in 2-d lattice M
sage: lp.vertices()
M(-1,  0),
M( 0, -1),
M( 1,  1),
M( 0,  1),
M( 1,  0)
in 2-d lattice M
```

```
>>> from sage.all import *
>>> P = Polyhedron( vertices = [(Integer(1)/Integer(2), Integer(1)/
↪Integer(2)), (Integer(0), Integer(1)), (-Integer(1), Integer(0)),␣
```

(continues on next page)

(continued from previous page)

```
↪(Integer(0), -Integer(1))])
>>> lp = P.lattice_polytope()
Traceback (most recent call last):
...
ValueError: Some vertices are not integral. You probably want
to add the argument "envelope=True" to compute an enveloping
lattice polytope.
>>> lp = P.lattice_polytope(True)
>>> lp                                                          # optional -
↪ polytopes_db, needs palp
2-d reflexive polytope #5 in 2-d lattice M
>>> lp.vertices()
M(-1,  0),
M( 0, -1),
M( 1,  1),
M( 0,  1),
M( 1,  0)
in 2-d lattice M
```

random_integral_point(***kwds*)

Return an integral point in this polyhedron chosen uniformly at random.

INPUT:

- `**kwds` – optional keyword parameters that are passed to *get_integral_point()*

OUTPUT:

The integral point in the polyhedron chosen uniformly at random. If the polyhedron is not compact, a `ValueError` is raised. If the polyhedron does not contain any integral points, an `EmptySetError` is raised.

See also

get_integral_point().

EXAMPLES:

```
sage: P = Polyhedron(vertices=[(-1,-1),(1,0),(1,1),(0,1)])
sage: P.random_integral_point()                                 # random
(0, 0)
sage: P.random_integral_point() in P.integral_points()
True
sage: P.random_integral_point(explicit_enumeration_threshold=0,      # random,␣
↪optional - latte_int
....:                        triangulation='cddlib')
(1, 1)
sage: P.random_integral_point(explicit_enumeration_threshold=0,      #␣
↪optional - latte_int
....:                        triangulation='cddlib', foo=7)
Traceback (most recent call last):
...
RuntimeError: ...
```

(continues on next page)

(continued from previous page)

```
sage: Q = Polyhedron(vertices=[(2, 1/3)], rays=[(1, 2)])
sage: Q.random_integral_point()
Traceback (most recent call last):
...
ValueError: ...

sage: R = Polyhedron(vertices=[(1/2, 0), (1, 1/2), (0, 1/2)])
sage: R.random_integral_point()
Traceback (most recent call last):
...
EmptySetError: ...
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[(-Integer(1),-Integer(1)),(Integer(1),
↪Integer(0)),(Integer(1),Integer(1)),(Integer(0),Integer(1))])
>>> P.random_integral_point()                                    # random
(0, 0)
>>> P.random_integral_point() in P.integral_points()
True
>>> P.random_integral_point(explicit_enumeration_threshold=Integer(0),      #␣
↪random, optional - latte_int
...                             triangulation='cddlib')
(1, 1)
>>> P.random_integral_point(explicit_enumeration_threshold=Integer(0),      #␣
↪optional - latte_int
...                             triangulation='cddlib', foo=Integer(7))
Traceback (most recent call last):
...
RuntimeError: ...

>>> Q = Polyhedron(vertices=[(Integer(2), Integer(1)/Integer(3))],␣
↪rays=[(Integer(1), Integer(2))])
>>> Q.random_integral_point()
Traceback (most recent call last):
...
ValueError: ...

>>> R = Polyhedron(vertices=[(Integer(1)/Integer(2), Integer(0)), (Integer(1),
↪ Integer(1)/Integer(2)), (Integer(0), Integer(1)/Integer(2))])
>>> R.random_integral_point()
Traceback (most recent call last):
...
EmptySetError: ...
```

2.6.4 Base class for polyhedra: Methods regarding the combinatorics of a polyhedron

Excluding methods relying on `sage.graphs`.

class `sage.geometry.polyhedron.base3.`**`Polyhedron_base3`**(*parent*, *Vrep*, *Hrep*, *Vrep_minimal=None*, *Hrep_minimal=None*, *pref_rep=None*, *mutable=False*, ***kwds*)

Bases: *Polyhedron_base2*

Methods related to the combinatorics of a polyhedron.

See *sage.geometry.polyhedron.base.Polyhedron_base*.

a_maximal_chain()

Return a maximal chain of the face lattice in increasing order.

EXAMPLES:

```
sage: P = polytopes.cube()
sage: P.a_maximal_chain()
[A -1-dimensional face of a Polyhedron in ZZ^3,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 8␣
↪vertices]
sage: P = polytopes.cube()
sage: chain = P.a_maximal_chain(); chain
[A -1-dimensional face of a Polyhedron in ZZ^3,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 8␣
↪vertices]
sage: [face.ambient_V_indices() for face in chain]
[(), (5,), (0, 5), (0, 3, 4, 5), (0, 1, 2, 3, 4, 5, 6, 7)]
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> P.a_maximal_chain()
[A -1-dimensional face of a Polyhedron in ZZ^3,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 8␣
↪vertices]
>>> P = polytopes.cube()
>>> chain = P.a_maximal_chain(); chain
[A -1-dimensional face of a Polyhedron in ZZ^3,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
```

(continues on next page)

(continued from previous page)

```
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 8␣
↪vertices]
>>> [face.ambient_V_indices() for face in chain]
[(), (5,), (0, 5), (0, 3, 4, 5), (0, 1, 2, 3, 4, 5, 6, 7)]
```

adjacency_matrix(*algorithm=None*)

Return the binary matrix of vertex adjacencies.

INPUT:

- `algorithm` – string (optional); specify whether the face generator starts with facets or vertices:
  - `'primal'` – start with the facets
  - `'dual'` – start with the vertices
  - `None` – choose automatically

EXAMPLES:

```
sage: polytopes.simplex(4).vertex_adjacency_matrix()
[0 1 1 1 1]
[1 0 1 1 1]
[1 1 0 1 1]
[1 1 1 0 1]
[1 1 1 1 0]
```

```
>>> from sage.all import *
>>> polytopes.simplex(Integer(4)).vertex_adjacency_matrix()
[0 1 1 1 1]
[1 0 1 1 1]
[1 1 0 1 1]
[1 1 1 0 1]
[1 1 1 1 0]
```

The rows and columns of the vertex adjacency matrix correspond to the `Vrepresentation()` objects: vertices, rays, and lines. The (i, j) matrix entry equals 1 if the i-th and j-th V-representation object are adjacent.

Two vertices are adjacent if they are the endpoints of an edge, that is, a one-dimensional face. For unbounded polyhedra this clearly needs to be generalized and we define two V-representation objects (see `sage.geometry.polyhedron.constructor`) to be adjacent if they together generate a one-face. There are three possible combinations:

- Two vertices can bound a finite-length edge.
- A vertex and a ray can generate a half-infinite edge starting at the vertex and with the direction given by the ray.
- A vertex and a line can generate an infinite edge. The position of the vertex on the line is arbitrary in this case, only its transverse position matters. The direction of the edge is given by the line generator.

For example, take the half-plane:

```
sage: half_plane = Polyhedron(ieqs=[(0,1,0)])
sage: half_plane.Hrepresentation()
(An inequality (1, 0) x + 0 >= 0,)
```

```
>>> from sage.all import *
>>> half_plane = Polyhedron(ieqs=[(Integer(0),Integer(1),Integer(0))])
>>> half_plane.Hrepresentation()
(An inequality (1, 0) x + 0 >= 0,)
```

Its (non-unique) V-representation consists of a vertex, a ray, and a line. The only edge is spanned by the vertex and the line generator, so they are adjacent:

```
sage: half_plane.Vrepresentation()
(A line in the direction (0, 1), A ray in the direction (1, 0), A vertex at␣
↪(0, 0))
sage: half_plane.vertex_adjacency_matrix()
[0 0 1]
[0 0 0]
[1 0 0]
```

```
>>> from sage.all import *
>>> half_plane.Vrepresentation()
(A line in the direction (0, 1), A ray in the direction (1, 0), A vertex at␣
↪(0, 0))
>>> half_plane.vertex_adjacency_matrix()
[0 0 1]
[0 0 0]
[1 0 0]
```

In one dimension higher, that is for a half-space in 3 dimensions, there is no one-dimensional face. Hence nothing is adjacent:

```
sage: Polyhedron(ieqs=[(0,1,0,0)]).vertex_adjacency_matrix()
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
```

```
>>> from sage.all import *
>>> Polyhedron(ieqs=[(Integer(0),Integer(1),Integer(0),Integer(0))]).vertex_
↪adjacency_matrix()
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
```

EXAMPLES:

In a bounded polygon, every vertex has precisely two adjacent ones:

```
sage: P = Polyhedron(vertices=[(0, 1), (1, 0), (3, 0), (4, 1)])
sage: for v in P.Vrep_generator():
....:     print("{} {}".format(P.adjacency_matrix().row(v.index()), v))
```

(continues on next page)

(continued from previous page)

```
(0, 1, 0, 1) A vertex at (0, 1)
(1, 0, 1, 0) A vertex at (1, 0)
(0, 1, 0, 1) A vertex at (3, 0)
(1, 0, 1, 0) A vertex at (4, 1)
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[(Integer(0), Integer(1)), (Integer(1),␣
↪Integer(0)), (Integer(3), Integer(0)), (Integer(4), Integer(1))])
>>> for v in P.Vrep_generator():
...     print("{} {}".format(P.adjacency_matrix().row(v.index()), v))
(0, 1, 0, 1) A vertex at (0, 1)
(1, 0, 1, 0) A vertex at (1, 0)
(0, 1, 0, 1) A vertex at (3, 0)
(1, 0, 1, 0) A vertex at (4, 1)
```

If the V-representation of the polygon contains vertices and one ray, then each V-representation object is adjacent to two V-representation objects:

```
sage: P = Polyhedron(vertices=[(0, 1), (1, 0), (3, 0), (4, 1)],
....:                rays=[(0,1)])
sage: for v in P.Vrep_generator():
....:       print("{} {}".format(P.adjacency_matrix().row(v.index()), v))
(0, 1, 0, 0, 1) A ray in the direction (0, 1)
(1, 0, 1, 0, 0) A vertex at (0, 1)
(0, 1, 0, 1, 0) A vertex at (1, 0)
(0, 0, 1, 0, 1) A vertex at (3, 0)
(1, 0, 0, 1, 0) A vertex at (4, 1)
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[(Integer(0), Integer(1)), (Integer(1),␣
↪Integer(0)), (Integer(3), Integer(0)), (Integer(4), Integer(1))],
...                rays=[(Integer(0),Integer(1))])
>>> for v in P.Vrep_generator():
...       print("{} {}".format(P.adjacency_matrix().row(v.index()), v))
(0, 1, 0, 0, 1) A ray in the direction (0, 1)
(1, 0, 1, 0, 0) A vertex at (0, 1)
(0, 1, 0, 1, 0) A vertex at (1, 0)
(0, 0, 1, 0, 1) A vertex at (3, 0)
(1, 0, 0, 1, 0) A vertex at (4, 1)
```

If the V-representation of the polygon contains vertices and two distinct rays, then each vertex is adjacent to two V-representation objects (which can now be vertices or rays). The two rays are not adjacent to each other:

```
sage: P = Polyhedron(vertices=[(0, 1), (1, 0), (3, 0), (4, 1)],
....:                rays=[(0,1), (1,1)])
sage: for v in P.Vrep_generator():
....:     print("{} {}".format(P.adjacency_matrix().row(v.index()), v))
(0, 1, 0, 0, 0) A ray in the direction (0, 1)
(1, 0, 1, 0, 0) A vertex at (0, 1)
(0, 1, 0, 0, 1) A vertex at (1, 0)
(0, 0, 0, 0, 1) A ray in the direction (1, 1)
(0, 0, 1, 1, 0) A vertex at (3, 0)
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[(Integer(0), Integer(1)), (Integer(1),␣
↪Integer(0)), (Integer(3), Integer(0)), (Integer(4), Integer(1))],
...                  rays=[(Integer(0),Integer(1)), (Integer(1),Integer(1))])
>>> for v in P.Vrep_generator():
...     print("{} {}".format(P.adjacency_matrix().row(v.index()), v))
(0, 1, 0, 0, 0) A ray in the direction (0, 1)
(1, 0, 1, 0, 0) A vertex at (0, 1)
(0, 1, 0, 0, 1) A vertex at (1, 0)
(0, 0, 0, 0, 1) A ray in the direction (1, 1)
(0, 0, 1, 1, 0) A vertex at (3, 0)
```

The vertex adjacency matrix has base ring integers. This way one can express various counting questions:

```
sage: P = polytopes.cube()
sage: Q = P.stack(P.faces(2)[0])
sage: M = Q.vertex_adjacency_matrix()
sage: sum(M)
(4, 4, 3, 3, 4, 4, 4, 3, 3)
sage: G = Q.vertex_graph()                                              #␣
↪needs sage.graphs
sage: G.degree()                                                        #␣
↪needs sage.graphs
[4, 4, 3, 3, 4, 4, 4, 3, 3]
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> Q = P.stack(P.faces(Integer(2))[Integer(0)])
>>> M = Q.vertex_adjacency_matrix()
>>> sum(M)
(4, 4, 3, 3, 4, 4, 4, 3, 3)
>>> G = Q.vertex_graph()                                                #␣
↪needs sage.graphs
>>> G.degree()                                                          #␣
↪needs sage.graphs
[4, 4, 3, 3, 4, 4, 4, 3, 3]
```

bounded_edges()

Return the bounded edges (excluding rays and lines).

OUTPUT: a generator for pairs of vertices, one pair per edge

EXAMPLES:

```
sage: p = Polyhedron(vertices=[[1,0],[0,1]], rays=[[1,0],[0,1]])
sage: [ e for e in p.bounded_edges() ]
[(A vertex at (0, 1), A vertex at (1, 0))]
sage: for e in p.bounded_edges(): print(e)
(A vertex at (0, 1), A vertex at (1, 0))
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices=[[Integer(1),Integer(0)],[Integer(0),Integer(1)]],
↪ rays=[[Integer(1),Integer(0)],[Integer(0),Integer(1)]])
```

(continues on next page)

(continued from previous page)

```
>>> [ e for e in p.bounded_edges() ]
[(A vertex at (0, 1), A vertex at (1, 0))]
>>> for e in p.bounded_edges(): print(e)
(A vertex at (0, 1), A vertex at (1, 0))
```

combinatorial_polyhedron()

Return the combinatorial type of `self`.

See *sage.geometry.polyhedron.combinatorial_polyhedron.base.CombinatorialPolyhedron*.

EXAMPLES:

```
sage: polytopes.cube().combinatorial_polyhedron()
A 3-dimensional combinatorial polyhedron with 6 facets

sage: polytopes.cyclic_polytope(4,10).combinatorial_polyhedron()
A 4-dimensional combinatorial polyhedron with 35 facets

sage: Polyhedron(rays=[[0,1], [1,0]]).combinatorial_polyhedron()
A 2-dimensional combinatorial polyhedron with 2 facets
```

```
>>> from sage.all import *
>>> polytopes.cube().combinatorial_polyhedron()
A 3-dimensional combinatorial polyhedron with 6 facets

>>> polytopes.cyclic_polytope(Integer(4),Integer(10)).combinatorial_
↪polyhedron()
A 4-dimensional combinatorial polyhedron with 35 facets

>>> Polyhedron(rays=[[Integer(0),Integer(1)], [Integer(1),Integer(0)]]).
↪combinatorial_polyhedron()
A 2-dimensional combinatorial polyhedron with 2 facets
```

f_vector(*num_threads=None*, *parallelization_depth=None*, *algorithm=None*)

Return the f-vector.

INPUT:

- `num_threads` – integer (optional); specify the number of threads; otherwise determined by `ncpus()`
- `parallelization_depth` – integer (optional); specify how deep in the lattice the parallelization is done
- `algorithm` – string (optional); specify whether the face generator starts with facets or vertices:
  - `'primal'` – start with the facets
  - `'dual'` – start with the vertices
  - `None` – choose automatically

OUTPUT:

Return a vector whose i-th entry is the number of $i-2$-dimensional faces of the polytope.

Note

The `vertices` as given by `vertices()` do not need to correspond to 0-dimensional faces. If a polyhedron contains k lines they correspond to k-dimensional faces. See example below.

EXAMPLES:

```
sage: p = Polyhedron(vertices=[[1, 2, 3], [1, 3, 2],
....:     [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1], [0, 0, 0]])
sage: p.f_vector()
(1, 7, 12, 7, 1)

sage: polytopes.cyclic_polytope(4,10).f_vector()
(1, 10, 45, 70, 35, 1)

sage: polytopes.hypercube(5).f_vector()
(1, 32, 80, 80, 40, 10, 1)
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices=[[Integer(1), Integer(2), Integer(3)],␣
↪[Integer(1), Integer(3), Integer(2)],
...     [Integer(2), Integer(1), Integer(3)], [Integer(2), Integer(3),␣
↪Integer(1)], [Integer(3), Integer(1), Integer(2)], [Integer(3), Integer(2),␣
↪Integer(1)], [Integer(0), Integer(0), Integer(0)]])
>>> p.f_vector()
(1, 7, 12, 7, 1)

>>> polytopes.cyclic_polytope(Integer(4),Integer(10)).f_vector()
(1, 10, 45, 70, 35, 1)

>>> polytopes.hypercube(Integer(5)).f_vector()
(1, 32, 80, 80, 40, 10, 1)
```

Polyhedra with lines do not have 0-faces:

```
sage: Polyhedron(ieqs=[[1,-1,0,0],[1,1,0,0]]).f_vector()
(1, 0, 0, 2, 1)
```

```
>>> from sage.all import *
>>> Polyhedron(ieqs=[[Integer(1),-Integer(1),Integer(0),Integer(0)],
↪[Integer(1),Integer(1),Integer(0),Integer(0)]]).f_vector()
(1, 0, 0, 2, 1)
```

However, the method `Polyhedron_base.vertices()` returns two points that belong to the `Vrepresentation`:

```
sage: P = Polyhedron(ieqs=[[1,-1,0],[1,1,0]])
sage: P.vertices()
(A vertex at (1, 0), A vertex at (-1, 0))
sage: P.f_vector()
(1, 0, 2, 1)
```

```
>>> from sage.all import *
>>> P = Polyhedron(ieqs=[[Integer(1),-Integer(1),Integer(0)],[Integer(1),
↪Integer(1),Integer(0)]])
>>> P.vertices()
(A vertex at (1, 0), A vertex at (-1, 0))
>>> P.f_vector()
(1, 0, 2, 1)
```

face_generator(*face_dimension=None*, *algorithm=None*)

Return an iterator over the faces of given dimension.

If dimension is not specified return an iterator over all faces.

INPUT:

- `face_dimension` – integer (default: `None`); yield only faces of this dimension if specified
- `algorithm` – string (optional); specify whether to start with facets or vertices:
  - `'primal'` – start with the facets
  - `'dual'` – start with the vertices
  - `None` – choose automatically

OUTPUT:

A *FaceIterator_geom*. This class iterates over faces as *PolyhedronFace*. See *face* for details. The order is random but fixed.

EXAMPLES:

```
sage: P = polytopes.cube()
sage: it = P.face_generator()
sage: it
Iterator over the faces of a 3-dimensional polyhedron in ZZ^3
sage: list(it)
[A 3-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 8␣
↪vertices,
 A -1-dimensional face of a Polyhedron in ZZ^3,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
```

(continues on next page)

(continued from previous page)

```
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices]

sage: P = polytopes.hypercube(4)
sage: list(P.face_generator(2))[:4]
[A 2-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of␣
↪4 vertices,
 A 2-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of␣
↪4 vertices,
 A 2-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of␣
↪4 vertices,
 A 2-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of␣
↪4 vertices]
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> it = P.face_generator()
>>> it
Iterator over the faces of a 3-dimensional polyhedron in ZZ^3
>>> list(it)
```

(continues on next page)

(continued from previous page)

```
[A 3-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 8␣
↪vertices,
 A -1-dimensional face of a Polyhedron in ZZ^3,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
```

(continues on next page)

(continued from previous page)

```
↪vertex,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices]

>>> P = polytopes.hypercube(Integer(4))
>>> list(P.face_generator(Integer(2)))[:Integer(4)]
 [A 2-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of␣
↪4 vertices,
  A 2-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of␣
↪4 vertices,
  A 2-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of␣
↪4 vertices,
  A 2-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of␣
↪4 vertices]
```

If a polytope has more facets than vertices, the dual mode is chosen:

```
sage: P = polytopes.cross_polytope(3)
sage: list(P.face_generator())
[A 3-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 6␣
↪vertices,
 A -1-dimensional face of a Polyhedron in ZZ^3,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
```

(continues on next page)

(continued from previous page)

```
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices]
```

```
>>> from sage.all import *
>>> P = polytopes.cross_polytope(Integer(3))
>>> list(P.face_generator())
[A 3-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 6␣
↪vertices,
 A -1-dimensional face of a Polyhedron in ZZ^3,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3␣
```

(continues on next page)

(continued from previous page)

```
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices,
 A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3␣
↪vertices,
 A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices]
```

The face iterator can also be slightly modified. In non-dual mode we can skip subfaces of the current (proper) face:

```
sage: P = polytopes.cube()
sage: it = P.face_generator(algorithm='primal')
sage: _ = next(it), next(it)
sage: face = next(it)
sage: face.ambient_H_indices()
(5,)
sage: it.ignore_subfaces()
sage: face = next(it)
sage: face.ambient_H_indices()
(4,)
sage: it.ignore_subfaces()
sage: [face.ambient_H_indices() for face in it]
[(3,),
 (2,),
 (1,),
 (0,),
 (2, 3),
 (1, 3),
 (1, 2, 3),
```

(continues on next page)

(continued from previous page)

```
 (1, 2),
 (0, 2),
 (0, 1, 2),
 (0, 1)]
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> it = P.face_generator(algorithm='primal')
>>> _ = next(it), next(it)
>>> face = next(it)
>>> face.ambient_H_indices()
(5,)
>>> it.ignore_subfaces()
>>> face = next(it)
>>> face.ambient_H_indices()
(4,)
>>> it.ignore_subfaces()
>>> [face.ambient_H_indices() for face in it]
[(3,),
 (2,),
 (1,),
 (0,),
 (2, 3),
 (1, 3),
 (1, 2, 3),
 (1, 2),
 (0, 2),
 (0, 1, 2),
 (0, 1)]
```

In dual mode we can skip supfaces of the current (proper) face:

```
sage: P = polytopes.cube()
sage: it = P.face_generator(algorithm='dual')
sage: _ = next(it), next(it)
sage: face = next(it)
sage: face.ambient_V_indices()
(7,)
sage: it.ignore_supfaces()
sage: next(it)
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex
sage: face = next(it)
sage: face.ambient_V_indices()
(5,)
sage: it.ignore_supfaces()
sage: [face.ambient_V_indices() for face in it]
[(4,),
 (3,),
 (2,),
 (1,),
 (0,),
```

(continues on next page)

(continued from previous page)

```
 (1, 6),
 (3, 4),
 (2, 3),
 (0, 3),
 (0, 1, 2, 3),
 (1, 2),
 (0, 1)]
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> it = P.face_generator(algorithm='dual')
>>> _ = next(it), next(it)
>>> face = next(it)
>>> face.ambient_V_indices()
(7,)
>>> it.ignore_supfaces()
>>> next(it)
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex
>>> face = next(it)
>>> face.ambient_V_indices()
(5,)
>>> it.ignore_supfaces()
>>> [face.ambient_V_indices() for face in it]
[(4,),
 (3,),
 (2,),
 (1,),
 (0,),
 (1, 6),
 (3, 4),
 (2, 3),
 (0, 3),
 (0, 1, 2, 3),
 (1, 2),
 (0, 1)]
```

In non-dual mode, we cannot skip supfaces:

```
sage: it = P.face_generator(algorithm='primal')
sage: _ = next(it), next(it)
sage: next(it)
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices
sage: it.ignore_supfaces()
Traceback (most recent call last):
...
ValueError: only possible when in dual mode
```

```
>>> from sage.all import *
>>> it = P.face_generator(algorithm='primal')
>>> _ = next(it), next(it)
```

(continues on next page)

(continued from previous page)

```
>>> next(it)
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices
>>> it.ignore_supfaces()
Traceback (most recent call last):
...
ValueError: only possible when in dual mode
```

In dual mode, we cannot skip subfaces:

```
sage: it = P.face_generator(algorithm='dual')
sage: _ = next(it), next(it)
sage: next(it)
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex
sage: it.ignore_subfaces()
Traceback (most recent call last):
...
ValueError: only possible when not in dual mode
```

```
>>> from sage.all import *
>>> it = P.face_generator(algorithm='dual')
>>> _ = next(it), next(it)
>>> next(it)
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex
>>> it.ignore_subfaces()
Traceback (most recent call last):
...
ValueError: only possible when not in dual mode
```

We can only skip sub-/supfaces of proper faces:

```
sage: it = P.face_generator(algorithm='primal')
sage: next(it)
A 3-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 8␣
↪vertices
sage: it.ignore_subfaces()
Traceback (most recent call last):
...
ValueError: iterator not set to a face yet
```

```
>>> from sage.all import *
>>> it = P.face_generator(algorithm='primal')
>>> next(it)
A 3-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 8␣
↪vertices
>>> it.ignore_subfaces()
Traceback (most recent call last):
...
ValueError: iterator not set to a face yet
```

See also

FaceIterator_geom.

ALGORITHM:

See *FaceIterator*.

faces(*face_dimension*)

Return the faces of given dimension.

INPUT:

- `face_dimension` – integer; the dimension of the faces whose representation will be returned

OUTPUT:

A tuple of *PolyhedronFace*. See module *sage.geometry.polyhedron.face* for details. The order is random but fixed.

See also

face_generator(), `facet()`.

EXAMPLES:

Here we find the vertex and face indices of the eight three-dimensional facets of the four-dimensional hypercube:

```
sage: p = polytopes.hypercube(4)
sage: list(f.ambient_V_indices() for f in p.faces(3))
[(0, 5, 6, 7, 8, 9, 14, 15),
 (1, 4, 5, 6, 10, 13, 14, 15),
 (1, 2, 6, 7, 8, 10, 11, 15),
 (8, 9, 10, 11, 12, 13, 14, 15),
 (0, 3, 4, 5, 9, 12, 13, 14),
 (0, 2, 3, 7, 8, 9, 11, 12),
 (1, 2, 3, 4, 10, 11, 12, 13),
 (0, 1, 2, 3, 4, 5, 6, 7)]

sage: face = p.faces(3)[3]
sage: face.ambient_Hrepresentation()
(An inequality (1, 0, 0, 0) x + 1 >= 0,)
sage: face.vertices()
(A vertex at (-1, -1, 1, -1),
 A vertex at (-1, -1, 1, 1),
 A vertex at (-1, 1, -1, -1),
 A vertex at (-1, 1, 1, -1),
 A vertex at (-1, 1, 1, 1),
 A vertex at (-1, 1, -1, 1),
 A vertex at (-1, -1, -1, 1),
 A vertex at (-1, -1, -1, -1))
```

```
>>> from sage.all import *
>>> p = polytopes.hypercube(Integer(4))
```

(continues on next page)

(continued from previous page)

```
>>> list(f.ambient_V_indices() for f in p.faces(Integer(3)))
[(0, 5, 6, 7, 8, 9, 14, 15),
 (1, 4, 5, 6, 10, 13, 14, 15),
 (1, 2, 6, 7, 8, 10, 11, 15),
 (8, 9, 10, 11, 12, 13, 14, 15),
 (0, 3, 4, 5, 9, 12, 13, 14),
 (0, 2, 3, 7, 8, 9, 11, 12),
 (1, 2, 3, 4, 10, 11, 12, 13),
 (0, 1, 2, 3, 4, 5, 6, 7)]

>>> face = p.faces(Integer(3))[Integer(3)]
>>> face.ambient_Hrepresentation()
(An inequality (1, 0, 0, 0) x + 1 >= 0,)
>>> face.vertices()
(A vertex at (-1, -1, 1, -1),
 A vertex at (-1, -1, 1, 1),
 A vertex at (-1, 1, -1, -1),
 A vertex at (-1, 1, 1, -1),
 A vertex at (-1, 1, 1, 1),
 A vertex at (-1, 1, -1, 1),
 A vertex at (-1, -1, -1, 1),
 A vertex at (-1, -1, -1, -1))
```

You can use the *index()* method to enumerate vertices and inequalities:

```
sage: def get_idx(rep): return rep.index()
sage: [get_idx(_) for _ in face.ambient_Hrepresentation()]
[4]
sage: [get_idx(_) for _ in face.ambient_Vrepresentation()]
[8, 9, 10, 11, 12, 13, 14, 15]

sage: [ ([get_idx(_) for _ in face.ambient_Vrepresentation()],
....:    [get_idx(_) for _ in face.ambient_Hrepresentation()])
....:   for face in p.faces(3) ]
[([0, 5, 6, 7, 8, 9, 14, 15], [7]),
 ([1, 4, 5, 6, 10, 13, 14, 15], [6]),
 ([1, 2, 6, 7, 8, 10, 11, 15], [5]),
 ([8, 9, 10, 11, 12, 13, 14, 15], [4]),
 ([0, 3, 4, 5, 9, 12, 13, 14], [3]),
 ([0, 2, 3, 7, 8, 9, 11, 12], [2]),
 ([1, 2, 3, 4, 10, 11, 12, 13], [1]),
 ([0, 1, 2, 3, 4, 5, 6, 7], [0])]
```

```
>>> from sage.all import *
>>> def get_idx(rep): return rep.index()
>>> [get_idx(_) for _ in face.ambient_Hrepresentation()]
[4]
>>> [get_idx(_) for _ in face.ambient_Vrepresentation()]
[8, 9, 10, 11, 12, 13, 14, 15]

>>> [ ([get_idx(_) for _ in face.ambient_Vrepresentation()],
...    [get_idx(_) for _ in face.ambient_Hrepresentation()])
```

(continues on next page)

(continued from previous page)

```
...     for face in p.faces(Integer(3)) ]
[([0, 5, 6, 7, 8, 9, 14, 15], [7]),
 ([1, 4, 5, 6, 10, 13, 14, 15], [6]),
 ([1, 2, 6, 7, 8, 10, 11, 15], [5]),
 ([8, 9, 10, 11, 12, 13, 14, 15], [4]),
 ([0, 3, 4, 5, 9, 12, 13, 14], [3]),
 ([0, 2, 3, 7, 8, 9, 11, 12], [2]),
 ([1, 2, 3, 4, 10, 11, 12, 13], [1]),
 ([0, 1, 2, 3, 4, 5, 6, 7], [0])]
```

facet_adjacency_matrix(*algorithm=None*)

Return the adjacency matrix for the facets.

INPUT:

- `algorithm` – string (optional); specify whether the face generator starts with facets or vertices:
  - `'primal'` – start with the facets
  - `'dual'` – start with the vertices
  - `None` – choose automatically

EXAMPLES:

```
sage: s4 = polytopes.simplex(4, project=True)
sage: s4.facet_adjacency_matrix()
[0 1 1 1 1]
[1 0 1 1 1]
[1 1 0 1 1]
[1 1 1 0 1]
[1 1 1 1 0]

sage: p = Polyhedron(vertices=[(0,0),(1,0),(0,1)])
sage: p.facet_adjacency_matrix()
[0 1 1]
[1 0 1]
[1 1 0]
```

```
>>> from sage.all import *
>>> s4 = polytopes.simplex(Integer(4), project=True)
>>> s4.facet_adjacency_matrix()
[0 1 1 1 1]
[1 0 1 1 1]
[1 1 0 1 1]
[1 1 1 0 1]
[1 1 1 1 0]

>>> p = Polyhedron(vertices=[(Integer(0),Integer(0)),(Integer(1),Integer(0)),
↪(Integer(0),Integer(1))])
>>> p.facet_adjacency_matrix()
[0 1 1]
[1 0 1]
[1 1 0]
```

The facet adjacency matrix has base ring integers. This way one can express various counting questions:

```
sage: P = polytopes.cube()
sage: Q = P.stack(P.faces(2)[0])
sage: M = Q.facet_adjacency_matrix()
sage: sum(M)
(4, 4, 4, 4, 3, 3, 3, 3, 4)
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> Q = P.stack(P.faces(Integer(2))[Integer(0)])
>>> M = Q.facet_adjacency_matrix()
>>> sum(M)
(4, 4, 4, 4, 3, 3, 3, 3, 4)
```

facets()

Return the facets of the polyhedron.

Facets are the maximal nontrivial faces of polyhedra. The empty face and the polyhedron itself are trivial.

A facet of a d-dimensional polyhedron is a face of dimension $d - 1$. For $d \neq 0$ the converse is true as well.

OUTPUT:

A tuple of *PolyhedronFace*. See *face* for details. The order is random but fixed.

See also

facets()

EXAMPLES:

Here we find the eight three-dimensional facets of the four-dimensional hypercube:

```
sage: p = polytopes.hypercube(4)
sage: p.facets()
(A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices)
```

```
>>> from sage.all import *
>>> p = polytopes.hypercube(Integer(4))
```

(continues on next page)

(continued from previous page)

```
>>> p.facets()
(A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices)
```

This is the same result as explicitly finding the three-dimensional faces:

```
sage: dim = p.dimension()
sage: p.faces(dim-1)
(A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices)
```

```
>>> from sage.all import *
>>> dim = p.dimension()
>>> p.faces(dim-Integer(1))
(A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices,
```

(continues on next page)

(continued from previous page)

```
 A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices,
 A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8␣
↪vertices)
```

The 0-dimensional polyhedron does not have facets:

```
sage: P = Polyhedron([[0]])
sage: P.facets()
()
```

```
>>> from sage.all import *
>>> P = Polyhedron([[Integer(0)]])
>>> P.facets()
()
```

greatest_common_subface_of_Hrep(**Hrepresentatives*)

Return the largest face that is contained in `Hrepresentatives`.

INPUT:

- `Hrepresentatives` – facets or indices of Hrepresentatives; the indices are assumed to be the indices of the `Hrepresentation()`

OUTPUT: a `PolyhedronFace`

EXAMPLES:

```
sage: P = polytopes.permutahedron(5)
sage: P.meet_of_Hrep()
A 4-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of␣
↪120 vertices
sage: P.meet_of_Hrep(1)
A 3-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 24␣
↪vertices
sage: P.meet_of_Hrep(4)
A 3-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 12␣
↪vertices
sage: P.meet_of_Hrep(1,3,7)
A 1-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 2␣
↪vertices
sage: P.meet_of_Hrep(1,3,7).ambient_H_indices()
(0, 1, 3, 7)
```

```
>>> from sage.all import *
>>> P = polytopes.permutahedron(Integer(5))
>>> P.meet_of_Hrep()
A 4-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of␣
↪120 vertices
>>> P.meet_of_Hrep(Integer(1))
A 3-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 24␣
↪vertices
```

(continues on next page)

(continued from previous page)

```
>>> P.meet_of_Hrep(Integer(4))
A 3-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 12␣
↪vertices
>>> P.meet_of_Hrep(Integer(1),Integer(3),Integer(7))
A 1-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 2␣
↪vertices
>>> P.meet_of_Hrep(Integer(1),Integer(3),Integer(7)).ambient_H_indices()
(0, 1, 3, 7)
```

The indices are the indices of the *Hrepresentation()*. `0` corresponds to an equation and is ignored:

```
sage: P.meet_of_Hrep(0)
A 4-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of␣
↪120 vertices
```

```
>>> from sage.all import *
>>> P.meet_of_Hrep(Integer(0))
A 4-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of␣
↪120 vertices
```

The input is flexible:

```
sage: P.meet_of_Hrep(P.facets()[-1], P.inequalities()[2], 7)
A 1-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 2␣
↪vertices
```

```
>>> from sage.all import *
>>> P.meet_of_Hrep(P.facets()[-Integer(1)], P.inequalities()[Integer(2)],␣
↪Integer(7))
A 1-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 2␣
↪vertices
```

The `Hrepresentatives` must belong to `self`:

```
sage: P = polytopes.cube(backend='ppl')
sage: Q = polytopes.cube(backend='field')
sage: f = P.facets()[0]
sage: P.meet_of_Hrep(f)
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices
sage: Q.meet_of_Hrep(f)
Traceback (most recent call last):
...
ValueError: not a facet of ``self``
sage: f = P.inequalities()[0]
sage: P.meet_of_Hrep(f)
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices
sage: Q.meet_of_Hrep(f)
Traceback (most recent call last):
...
ValueError: not a facet of ``self``
```

```
>>> from sage.all import *
>>> P = polytopes.cube(backend='ppl')
>>> Q = polytopes.cube(backend='field')
>>> f = P.facets()[Integer(0)]
>>> P.meet_of_Hrep(f)
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices
>>> Q.meet_of_Hrep(f)
Traceback (most recent call last):
...
ValueError: not a facet of ``self``
>>> f = P.inequalities()[Integer(0)]
>>> P.meet_of_Hrep(f)
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices
>>> Q.meet_of_Hrep(f)
Traceback (most recent call last):
...
ValueError: not a facet of ``self``
```

incidence_matrix()

Return the incidence matrix.

Note

The columns correspond to inequalities/equations in the order *Hrepresentation()*, the rows correspond to vertices/rays/lines in the order *Vrepresentation()*.

See also

slack_matrix().

EXAMPLES:

```
sage: p = polytopes.cuboctahedron()
sage: p.incidence_matrix()
[0 0 1 1 0 1 0 0 0 0 1 0 0 0]
[0 0 0 1 0 0 1 0 1 0 1 0 0 0]
[0 0 1 1 1 0 0 1 0 0 0 0 0 0]
[1 0 0 1 1 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 1 1 1 0 0 0]
[0 0 1 0 0 1 0 1 0 0 0 1 0 0]
[1 0 0 0 0 0 1 0 1 0 0 0 1 0]
[1 0 0 0 1 0 0 1 0 0 0 0 0 1]
[0 1 0 0 0 1 0 0 0 1 0 1 0 0]
[0 1 0 0 0 0 0 0 1 1 0 0 1 0]
[0 1 0 0 0 0 0 1 0 0 0 1 0 1]
[1 1 0 0 0 0 0 0 0 0 0 0 1 1]
sage: v = p.Vrepresentation(0)
sage: v
```

(continues on next page)

(continued from previous page)

```
A vertex at (-1, -1, 0)
sage: h = p.Hrepresentation(2)
sage: h
An inequality (1, 1, -1) x + 2 >= 0
sage: h.eval(v)         # evaluation (1, 1, -1) * (-1/2, -1/2, 0) + 1
0
sage: h*v               # same as h.eval(v)
0
sage: p.incidence_matrix() [0,2]    # this entry is (v,h)
1
sage: h.contains(v)
True
sage: p.incidence_matrix() [2,0]    # note: not symmetric
0
```

```
>>> from sage.all import *
>>> p = polytopes.cuboctahedron()
>>> p.incidence_matrix()
[0 0 1 1 0 1 0 0 0 0 1 0 0 0]
[0 0 0 1 0 0 1 0 1 0 1 0 0 0]
[0 0 1 1 1 0 0 1 0 0 0 0 0 0]
[1 0 0 1 1 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 1 1 1 0 0 0]
[0 0 1 0 0 1 0 1 0 0 0 1 0 0]
[1 0 0 0 0 0 1 0 1 0 0 0 1 0]
[1 0 0 0 1 0 0 1 0 0 0 0 0 1]
[0 1 0 0 0 1 0 0 0 1 0 1 0 0]
[0 1 0 0 0 0 0 0 1 1 0 0 1 0]
[0 1 0 0 0 0 0 1 0 0 0 1 0 1]
[1 1 0 0 0 0 0 0 0 0 0 0 1 1]
>>> v = p.Vrepresentation(Integer(0))
>>> v
A vertex at (-1, -1, 0)
>>> h = p.Hrepresentation(Integer(2))
>>> h
An inequality (1, 1, -1) x + 2 >= 0
>>> h.eval(v)         # evaluation (1, 1, -1) * (-1/2, -1/2, 0) + 1
0
>>> h*v               # same as h.eval(v)
0
>>> p.incidence_matrix() [Integer(0),Integer(2)]    # this entry is (v,h)
1
>>> h.contains(v)
True
>>> p.incidence_matrix() [Integer(2),Integer(0)]    # note: not symmetric
0
```

The incidence matrix depends on the ambient dimension:

```
sage: simplex = polytopes.simplex(); simplex
A 3-dimensional polyhedron in ZZ^4 defined as the convex hull of 4 vertices
sage: simplex.incidence_matrix()
```

(continues on next page)

(continued from previous page)

```
[1 1 1 1 0]
[1 1 1 0 1]
[1 1 0 1 1]
[1 0 1 1 1]
sage: simplex = simplex.affine_hull_projection(); simplex
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 4 vertices
sage: simplex.incidence_matrix()
[1 1 1 0]
[1 1 0 1]
[1 0 1 1]
[0 1 1 1]
```

```
>>> from sage.all import *
>>> simplex = polytopes.simplex(); simplex
A 3-dimensional polyhedron in ZZ^4 defined as the convex hull of 4 vertices
>>> simplex.incidence_matrix()
[1 1 1 1 0]
[1 1 1 0 1]
[1 1 0 1 1]
[1 0 1 1 1]
>>> simplex = simplex.affine_hull_projection(); simplex
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 4 vertices
>>> simplex.incidence_matrix()
[1 1 1 0]
[1 1 0 1]
[1 0 1 1]
[0 1 1 1]
```

An incidence matrix does not determine a unique polyhedron:

```
sage: P = Polyhedron(vertices=[[0,1],[1,1],[1,0]])
sage: P.incidence_matrix()
[1 1 0]
[1 0 1]
[0 1 1]

sage: Q = Polyhedron(vertices=[[0,1], [1,0]], rays=[[1,1]])
sage: Q.incidence_matrix()
[1 1 0]
[1 0 1]
[0 1 1]
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[[Integer(0),Integer(1)],[Integer(1),Integer(1)],
↪[Integer(1),Integer(0)]])
>>> P.incidence_matrix()
[1 1 0]
[1 0 1]
[0 1 1]

>>> Q = Polyhedron(vertices=[[Integer(0),Integer(1)], [Integer(1),
↪Integer(0)]], rays=[[Integer(1),Integer(1)]])
```

(continues on next page)

(continued from previous page)

```
>>> Q.incidence_matrix()
[1 1 0]
[1 0 1]
[0 1 1]
```

An example of two polyhedra with isomorphic face lattices but different incidence matrices:

```
sage: Q.incidence_matrix()
[1 1 0]
[1 0 1]
[0 1 1]

sage: R = Polyhedron(vertices=[[0,1], [1,0]], rays=[[1,3/2], [3/2,1]])
sage: R.incidence_matrix()
[1 1 0]
[1 0 1]
[0 1 0]
[0 0 1]
```

```
>>> from sage.all import *
>>> Q.incidence_matrix()
[1 1 0]
[1 0 1]
[0 1 1]

>>> R = Polyhedron(vertices=[[Integer(0),Integer(1)], [Integer(1),
↪Integer(0)]], rays=[[Integer(1),Integer(3)/Integer(2)], [Integer(3)/
↪Integer(2),Integer(1)]])
>>> R.incidence_matrix()
[1 1 0]
[1 0 1]
[0 1 0]
[0 0 1]
```

The incidence matrix has base ring integers. This way one can express various counting questions:

```
sage: P = polytopes.twenty_four_cell()
sage: M = P.incidence_matrix()
sage: sum(sum(x) for x in M) == P.flag_f_vector(0, 3)                         #␣
↪needs sage.combinat
True
```

```
>>> from sage.all import *
>>> P = polytopes.twenty_four_cell()
>>> M = P.incidence_matrix()
>>> sum(sum(x) for x in M) == P.flag_f_vector(Integer(0), Integer(3))        ␣
↪               # needs sage.combinat
True
```

is_bipyramid(*certificate=False*)

Test whether the polytope is combinatorially equivalent to a bipyramid over some polytope.

INPUT:

- `certificate` – boolean (default: `False`); specifies whether to return two vertices of the polytope which are the apices of a bipyramid, if found

OUTPUT: if `certificate` is `True`, returns a tuple containing:

1. Boolean.
2. **`None` or a tuple containing:**
   a. The first apex.
   b. The second apex.

If `certificate` is `False` returns a boolean.

EXAMPLES:

```
sage: P = polytopes.octahedron()
sage: P.is_bipyramid()
True
sage: P.is_bipyramid(certificate=True)
(True, [A vertex at (1, 0, 0), A vertex at (-1, 0, 0)])
sage: Q = polytopes.cyclic_polytope(3,7)
sage: Q.is_bipyramid()
False
sage: R = Q.bipyramid()
sage: R.is_bipyramid(certificate=True)
(True, [A vertex at (1, 3, 13, 63), A vertex at (-1, 3, 13, 63)])
```

```
>>> from sage.all import *
>>> P = polytopes.octahedron()
>>> P.is_bipyramid()
True
>>> P.is_bipyramid(certificate=True)
(True, [A vertex at (1, 0, 0), A vertex at (-1, 0, 0)])
>>> Q = polytopes.cyclic_polytope(Integer(3),Integer(7))
>>> Q.is_bipyramid()
False
>>> R = Q.bipyramid()
>>> R.is_bipyramid(certificate=True)
(True, [A vertex at (1, 3, 13, 63), A vertex at (-1, 3, 13, 63)])
```

is_lawrence_polytope()

Return `True` if `self` is a Lawrence polytope.

A polytope is called a Lawrence polytope if it has a centrally symmetric (normalized) Gale diagram.

EXAMPLES:

```
sage: P = polytopes.hypersimplex(5,2)
sage: L = P.lawrence_polytope()
sage: L.is_lattice_polytope()
True

sage: egyptian_pyramid = polytopes.regular_polygon(4).pyramid()                #␣
↪needs sage.number_field
sage: egyptian_pyramid.is_lawrence_polytope()                                  #␣
↪needs sage.number_field
```

(continues on next page)

(continued from previous page)

```
True

sage: polytopes.octahedron().is_lawrence_polytope()
False
```

```
>>> from sage.all import *
>>> P = polytopes.hypersimplex(Integer(5),Integer(2))
>>> L = P.lawrence_polytope()
>>> L.is_lattice_polytope()
True

>>> egyptian_pyramid = polytopes.regular_polygon(Integer(4)).pyramid()      ␣
↪     # needs sage.number_field
>>> egyptian_pyramid.is_lawrence_polytope()                                #␣
↪needs sage.number_field
True

>>> polytopes.octahedron().is_lawrence_polytope()
False
```

REFERENCES:

For more information, see [BaSt1990].

is_neighborly(*k=None*)

Return whether the polyhedron is neighborly.

If the input `k` is provided, then return whether the polyhedron is `k`-neighborly

A polyhedron is neighborly if every set of n vertices forms a face for n up to floor of half the dimension of the polyhedron. It is k-neighborly if this is true for n up to k.

INPUT:

- `k` – the dimension up to which to check if every set of `k` vertices forms a face. If no `k` is provided, check up to floor of half the dimension of the polyhedron.

OUTPUT:

- `True` if every set of up to `k` vertices forms a face,
- `False` otherwise

See also

neighborliness()

EXAMPLES:

```
sage: cube = polytopes.hypercube(3)
sage: cube.is_neighborly()
True
sage: cube = polytopes.hypercube(4)
sage: cube.is_neighborly()
False
```

```
>>> from sage.all import *
>>> cube = polytopes.hypercube(Integer(3))
>>> cube.is_neighborly()
True
>>> cube = polytopes.hypercube(Integer(4))
>>> cube.is_neighborly()
False
```

Cyclic polytopes are neighborly:

```
sage: all(polytopes.cyclic_polytope(i, i + 1 + j).is_neighborly() for i in␣
↪range(5) for j in range(3))
True
```

```
>>> from sage.all import *
>>> all(polytopes.cyclic_polytope(i, i + Integer(1) + j).is_neighborly() for␣
↪i in range(Integer(5)) for j in range(Integer(3)))
True
```

The neighborliness of a polyhedron equals floor of dimension half (or larger in case of a simplex) if and only if the polyhedron is neighborly:

```
sage: testpolys = [polytopes.cube(), polytopes.cyclic_polytope(6, 9),␣
↪polytopes.simplex(6)]
sage: [(P.neighborliness() >= P.dim() // 2) == P.is_neighborly() for P in␣
↪testpolys]
[True, True, True]
```

```
>>> from sage.all import *
>>> testpolys = [polytopes.cube(), polytopes.cyclic_polytope(Integer(6),␣
↪Integer(9)), polytopes.simplex(Integer(6))]
>>> [(P.neighborliness() >= P.dim() // Integer(2)) == P.is_neighborly() for P␣
↪in testpolys]
[True, True, True]
```

is_prism(*certificate=False*)

Test whether the polytope is combinatorially equivalent to a prism of some polytope.

INPUT:

- `certificate` – boolean (default: `False`); specifies whether to return two facets of the polytope which are the bases of a prism, if found

OUTPUT: if `certificate` is `True`, returns a tuple containing:

1. Boolean.
2. **`None` or a tuple containing:**
   a. List of the vertices of the first base facet.
   b. List of the vertices of the second base facet.

If `certificate` is `False` returns a boolean.

EXAMPLES:

```
sage: P = polytopes.cube()
sage: P.is_prism()
True
sage: P.is_prism(certificate=True)
(True,
 [(A vertex at (1, -1, -1),
   A vertex at (1, -1, 1),
   A vertex at (-1, -1, 1),
   A vertex at (-1, -1, -1)),
  (A vertex at (1, 1, -1),
   A vertex at (1, 1, 1),
   A vertex at (-1, 1, -1),
   A vertex at (-1, 1, 1))])
sage: Q = polytopes.cyclic_polytope(3,8)
sage: Q.is_prism()
False
sage: R = Q.prism()
sage: R.is_prism(certificate=True)
(True,
 [(A vertex at (0, 3, 9, 27),
   A vertex at (0, 6, 36, 216),
   A vertex at (0, 0, 0, 0),
   A vertex at (0, 7, 49, 343),
   A vertex at (0, 5, 25, 125),
   A vertex at (0, 1, 1, 1),
   A vertex at (0, 2, 4, 8),
   A vertex at (0, 4, 16, 64)),
  (A vertex at (1, 6, 36, 216),
   A vertex at (1, 0, 0, 0),
   A vertex at (1, 7, 49, 343),
   A vertex at (1, 5, 25, 125),
   A vertex at (1, 1, 1, 1),
   A vertex at (1, 2, 4, 8),
   A vertex at (1, 4, 16, 64),
   A vertex at (1, 3, 9, 27))])
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> P.is_prism()
True
>>> P.is_prism(certificate=True)
(True,
 [(A vertex at (1, -1, -1),
   A vertex at (1, -1, 1),
   A vertex at (-1, -1, 1),
   A vertex at (-1, -1, -1)),
  (A vertex at (1, 1, -1),
   A vertex at (1, 1, 1),
   A vertex at (-1, 1, -1),
   A vertex at (-1, 1, 1))])
>>> Q = polytopes.cyclic_polytope(Integer(3),Integer(8))
>>> Q.is_prism()
```

(continues on next page)

(continued from previous page)

```
False
>>> R = Q.prism()
>>> R.is_prism(certificate=True)
(True,
 [(A vertex at (0, 3, 9, 27),
   A vertex at (0, 6, 36, 216),
   A vertex at (0, 0, 0, 0),
   A vertex at (0, 7, 49, 343),
   A vertex at (0, 5, 25, 125),
   A vertex at (0, 1, 1, 1),
   A vertex at (0, 2, 4, 8),
   A vertex at (0, 4, 16, 64)),
  (A vertex at (1, 6, 36, 216),
   A vertex at (1, 0, 0, 0),
   A vertex at (1, 7, 49, 343),
   A vertex at (1, 5, 25, 125),
   A vertex at (1, 1, 1, 1),
   A vertex at (1, 2, 4, 8),
   A vertex at (1, 4, 16, 64),
   A vertex at (1, 3, 9, 27))])
```

is_pyramid(*certificate=False*)

Test whether the polytope is a pyramid over one of its facets.

INPUT:

- `certificate` – boolean (default: `False`); specifies whether to return a vertex of the polytope which is the apex of a pyramid, if found

OUTPUT: if `certificate` is `True`, returns a tuple containing:

1. Boolean.
2. The apex of the pyramid or `None`.

If `certificate` is `False` returns a boolean.

EXAMPLES:

```
sage: P = polytopes.simplex(3)
sage: P.is_pyramid()
True
sage: P.is_pyramid(certificate=True)
(True, A vertex at (1, 0, 0, 0))
sage: egyptian_pyramid = polytopes.regular_polygon(4).pyramid()             #␣
↪needs sage.rings.number_field
sage: egyptian_pyramid.is_pyramid()                                          #␣
↪needs sage.rings.number_field
True
sage: Q = polytopes.octahedron()
sage: Q.is_pyramid()
False
```

```
>>> from sage.all import *
>>> P = polytopes.simplex(Integer(3))
```

(continues on next page)

(continued from previous page)

```
>>> P.is_pyramid()
True
>>> P.is_pyramid(certificate=True)
(True, A vertex at (1, 0, 0, 0))
>>> egyptian_pyramid = polytopes.regular_polygon(Integer(4)).pyramid()       ␣
↪     # needs sage.rings.number_field
>>> egyptian_pyramid.is_pyramid()                                           #␣
↪needs sage.rings.number_field
True
>>> Q = polytopes.octahedron()
>>> Q.is_pyramid()
False
```

For the 0-dimensional polyhedron, the output is `True`, but it cannot be constructed as a pyramid over the empty polyhedron:

```
sage: P = Polyhedron([[0]])
sage: P.is_pyramid()
True
sage: Polyhedron().pyramid()
Traceback (most recent call last):
...
ZeroDivisionError: rational division by zero
```

```
>>> from sage.all import *
>>> P = Polyhedron([[Integer(0)]])
>>> P.is_pyramid()
True
>>> Polyhedron().pyramid()
Traceback (most recent call last):
...
ZeroDivisionError: rational division by zero
```

is_simple()

Test for simplicity of a polytope.

See Wikipedia article Simple_polytope

EXAMPLES:

```
sage: p = Polyhedron([[0,0,0],[1,0,0],[0,1,0],[0,0,1]])
sage: p.is_simple()
True
sage: p = Polyhedron([[0,0,0],[4,4,0],[4,0,0],[0,4,0],[2,2,2]])
sage: p.is_simple()
False
```

```
>>> from sage.all import *
>>> p = Polyhedron([[Integer(0),Integer(0),Integer(0)],[Integer(1),Integer(0),
↪Integer(0)],[Integer(0),Integer(1),Integer(0)],[Integer(0),Integer(0),
↪Integer(1)]])
>>> p.is_simple()
True
```

(continues on next page)

(continued from previous page)

```
>>> p = Polyhedron([[Integer(0),Integer(0),Integer(0)],[Integer(4),Integer(4),
↪Integer(0)],[Integer(4),Integer(0),Integer(0)],[Integer(0),Integer(4),
↪Integer(0)],[Integer(2),Integer(2),Integer(2)]])
>>> p.is_simple()
False
```

is_simplex()

Return whether the polyhedron is a simplex.

A simplex is a bounded polyhedron with $d + 1$ vertices, where d is the dimension.

EXAMPLES:

```
sage: Polyhedron([(0,0,0), (1,0,0), (0,1,0)]).is_simplex()
True
sage: polytopes.simplex(3).is_simplex()
True
sage: polytopes.hypercube(3).is_simplex()
False
```

```
>>> from sage.all import *
>>> Polyhedron([(Integer(0),Integer(0),Integer(0)), (Integer(1),Integer(0),
↪Integer(0)), (Integer(0),Integer(1),Integer(0))]).is_simplex()
True
>>> polytopes.simplex(Integer(3)).is_simplex()
True
>>> polytopes.hypercube(Integer(3)).is_simplex()
False
```

is_simplicial()

Test if the polytope is simplicial.

A polytope is simplicial if every facet is a simplex.

See Wikipedia article Simplicial_polytope

EXAMPLES:

```
sage: p = polytopes.hypercube(3)
sage: p.is_simplicial()
False
sage: q = polytopes.simplex(5, project=True)
sage: q.is_simplicial()
True
sage: p = Polyhedron([[0,0,0],[1,0,0],[0,1,0],[0,0,1]])
sage: p.is_simplicial()
True
sage: q = Polyhedron([[1,1,1],[-1,1,1],[1,-1,1],[-1,-1,1],[1,1,-1]])
sage: q.is_simplicial()
False
sage: P = polytopes.simplex(); P
A 3-dimensional polyhedron in ZZ^4 defined as the convex hull of 4 vertices
sage: P.is_simplicial()
True
```

```
>>> from sage.all import *
>>> p = polytopes.hypercube(Integer(3))
>>> p.is_simplicial()
False
>>> q = polytopes.simplex(Integer(5), project=True)
>>> q.is_simplicial()
True
>>> p = Polyhedron([[Integer(0),Integer(0),Integer(0)],[Integer(1),Integer(0),
↪Integer(0)],[Integer(0),Integer(1),Integer(0)],[Integer(0),Integer(0),
↪Integer(1)]])
>>> p.is_simplicial()
True
>>> q = Polyhedron([[Integer(1),Integer(1),Integer(1)],[-Integer(1),
↪Integer(1),Integer(1)],[Integer(1),-Integer(1),Integer(1)],[-Integer(1),-
↪Integer(1),Integer(1)],[Integer(1),Integer(1),-Integer(1)]])
>>> q.is_simplicial()
False
>>> P = polytopes.simplex(); P
A 3-dimensional polyhedron in ZZ^4 defined as the convex hull of 4 vertices
>>> P.is_simplicial()
True
```

The method is not implemented for unbounded polyhedra:

```
sage: p = Polyhedron(vertices=[(0,0)],rays=[(1,0),(0,1)])
sage: p.is_simplicial()
Traceback (most recent call last):
...
NotImplementedError: this function is implemented for polytopes only
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices=[(Integer(0),Integer(0))],rays=[(Integer(1),
↪Integer(0)),(Integer(0),Integer(1))])
>>> p.is_simplicial()
Traceback (most recent call last):
...
NotImplementedError: this function is implemented for polytopes only
```

join_of_Vrep(**Vrepresentatives*)

Return the smallest face that contains `Vrepresentatives`.

INPUT:

- `Vrepresentatives` – vertices/rays/lines of `self` or indices of such

OUTPUT: a *PolyhedronFace*

> **Note**
>
> In the case of unbounded polyhedra, the join of rays etc. may not be well-defined.

EXAMPLES:

```
sage: P = polytopes.permutahedron(5)
sage: P.join_of_Vrep(1)
A 0-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 1␣
↪vertex
sage: P.join_of_Vrep()
A -1-dimensional face of a Polyhedron in ZZ^5
sage: P.join_of_Vrep(0,12,13).ambient_V_indices()
(0, 12, 13, 68)
```

```
>>> from sage.all import *
>>> P = polytopes.permutahedron(Integer(5))
>>> P.join_of_Vrep(Integer(1))
A 0-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 1␣
↪vertex
>>> P.join_of_Vrep()
A -1-dimensional face of a Polyhedron in ZZ^5
>>> P.join_of_Vrep(Integer(0),Integer(12),Integer(13)).ambient_V_indices()
(0, 12, 13, 68)
```

The input is flexible:

```
sage: P.join_of_Vrep(2, P.vertices()[3], P.Vrepresentation(4))
A 2-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 6␣
↪vertices
```

```
>>> from sage.all import *
>>> P.join_of_Vrep(Integer(2), P.vertices()[Integer(3)], P.
↪Vrepresentation(Integer(4)))
A 2-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 6␣
↪vertices
```

```
sage: P = polytopes.cube()
sage: a, b = P.faces(0)[:2]
sage: P.join_of_Vrep(a, b)
A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> a, b = P.faces(Integer(0))[:Integer(2)]
>>> P.join_of_Vrep(a, b)
A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices
```

In the case of an unbounded polyhedron, the join may not be well-defined:

```
sage: P = Polyhedron(vertices=[[1,0], [0,1]], rays=[[1,1]])
sage: P.join_of_Vrep(0)
A 0-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 1␣
↪vertex
sage: P.join_of_Vrep(0,1)
A 1-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 2␣
```

(continues on next page)

(continued from previous page)

```
↪vertices
sage: P.join_of_Vrep(0,2)
A 1-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 1␣
↪vertex and 1 ray
sage: P.join_of_Vrep(1,2)
A 1-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 1␣
↪vertex and 1 ray
sage: P.join_of_Vrep(2)
Traceback (most recent call last):
...
ValueError: the join is not well-defined
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[[Integer(1),Integer(0)], [Integer(0),
↪Integer(1)]], rays=[[Integer(1),Integer(1)]])
>>> P.join_of_Vrep(Integer(0))
A 0-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 1␣
↪vertex
>>> P.join_of_Vrep(Integer(0),Integer(1))
A 1-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 2␣
↪vertices
>>> P.join_of_Vrep(Integer(0),Integer(2))
A 1-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 1␣
↪vertex and 1 ray
>>> P.join_of_Vrep(Integer(1),Integer(2))
A 1-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 1␣
↪vertex and 1 ray
>>> P.join_of_Vrep(Integer(2))
Traceback (most recent call last):
...
ValueError: the join is not well-defined
```

The `Vrepresentatives` must be of `self`:

```
sage: P = polytopes.cube(backend='ppl')
sage: Q = polytopes.cube(backend='field')
sage: v = P.vertices()[0]
sage: P.join_of_Vrep(v)
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex
sage: Q.join_of_Vrep(v)
Traceback (most recent call last):
...
ValueError: not a Vrepresentative of ``self``
sage: f = P.faces(0)[0]
sage: P.join_of_Vrep(v)
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex
sage: Q.join_of_Vrep(v)
Traceback (most recent call last):
...
ValueError: not a Vrepresentative of ``self``
```

```
>>> from sage.all import *
>>> P = polytopes.cube(backend='ppl')
>>> Q = polytopes.cube(backend='field')
>>> v = P.vertices()[Integer(0)]
>>> P.join_of_Vrep(v)
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex
>>> Q.join_of_Vrep(v)
Traceback (most recent call last):
...
ValueError: not a Vrepresentative of ``self``
>>> f = P.faces(Integer(0))[Integer(0)]
>>> P.join_of_Vrep(v)
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex
>>> Q.join_of_Vrep(v)
Traceback (most recent call last):
...
ValueError: not a Vrepresentative of ``self``
```

least_common_superface_of_Vrep(**Vrepresentatives*)

Return the smallest face that contains `Vrepresentatives`.

INPUT:

- `Vrepresentatives` – vertices/rays/lines of `self` or indices of such

OUTPUT: a *PolyhedronFace*

> **Note**
>
> In the case of unbounded polyhedra, the join of rays etc. may not be well-defined.

EXAMPLES:

```
sage: P = polytopes.permutahedron(5)
sage: P.join_of_Vrep(1)
A 0-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 1␣
↪vertex
sage: P.join_of_Vrep()
A -1-dimensional face of a Polyhedron in ZZ^5
sage: P.join_of_Vrep(0,12,13).ambient_V_indices()
(0, 12, 13, 68)
```

```
>>> from sage.all import *
>>> P = polytopes.permutahedron(Integer(5))
>>> P.join_of_Vrep(Integer(1))
A 0-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 1␣
↪vertex
>>> P.join_of_Vrep()
A -1-dimensional face of a Polyhedron in ZZ^5
>>> P.join_of_Vrep(Integer(0),Integer(12),Integer(13)).ambient_V_indices()
(0, 12, 13, 68)
```

The input is flexible:

```
sage: P.join_of_Vrep(2, P.vertices()[3], P.Vrepresentation(4))
A 2-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 6␣
↪vertices
```

```
>>> from sage.all import *
>>> P.join_of_Vrep(Integer(2), P.vertices()[Integer(3)], P.
↪Vrepresentation(Integer(4)))
A 2-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 6␣
↪vertices
```

```
sage: P = polytopes.cube()
sage: a, b = P.faces(0)[:2]
sage: P.join_of_Vrep(a, b)
A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> a, b = P.faces(Integer(0))[:Integer(2)]
>>> P.join_of_Vrep(a, b)
A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2␣
↪vertices
```

In the case of an unbounded polyhedron, the join may not be well-defined:

```
sage: P = Polyhedron(vertices=[[1,0], [0,1]], rays=[[1,1]])
sage: P.join_of_Vrep(0)
A 0-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 1␣
↪vertex
sage: P.join_of_Vrep(0,1)
A 1-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 2␣
↪vertices
sage: P.join_of_Vrep(0,2)
A 1-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 1␣
↪vertex and 1 ray
sage: P.join_of_Vrep(1,2)
A 1-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 1␣
↪vertex and 1 ray
sage: P.join_of_Vrep(2)
Traceback (most recent call last):
...
ValueError: the join is not well-defined
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[[Integer(1),Integer(0)], [Integer(0),
↪Integer(1)]], rays=[[Integer(1),Integer(1)]])
>>> P.join_of_Vrep(Integer(0))
A 0-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 1␣
↪vertex
>>> P.join_of_Vrep(Integer(0),Integer(1))
A 1-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 2␣
```

(continues on next page)

(continued from previous page)

```
↪vertices
>>> P.join_of_Vrep(Integer(0),Integer(2))
A 1-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 1␣
↪vertex and 1 ray
>>> P.join_of_Vrep(Integer(1),Integer(2))
A 1-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 1␣
↪vertex and 1 ray
>>> P.join_of_Vrep(Integer(2))
Traceback (most recent call last):
...
ValueError: the join is not well-defined
```

The `Vrepresentatives` must be of `self`:

```
sage: P = polytopes.cube(backend='ppl')
sage: Q = polytopes.cube(backend='field')
sage: v = P.vertices()[0]
sage: P.join_of_Vrep(v)
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex
sage: Q.join_of_Vrep(v)
Traceback (most recent call last):
...
ValueError: not a Vrepresentative of ``self``
sage: f = P.faces(0)[0]
sage: P.join_of_Vrep(v)
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex
sage: Q.join_of_Vrep(v)
Traceback (most recent call last):
...
ValueError: not a Vrepresentative of ``self``
```

```
>>> from sage.all import *
>>> P = polytopes.cube(backend='ppl')
>>> Q = polytopes.cube(backend='field')
>>> v = P.vertices()[Integer(0)]
>>> P.join_of_Vrep(v)
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex
>>> Q.join_of_Vrep(v)
Traceback (most recent call last):
...
ValueError: not a Vrepresentative of ``self``
>>> f = P.faces(Integer(0))[Integer(0)]
>>> P.join_of_Vrep(v)
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1␣
↪vertex
>>> Q.join_of_Vrep(v)
Traceback (most recent call last):
...
ValueError: not a Vrepresentative of ``self``
```

meet_of_Hrep(**Hrepresentatives*)

Return the largest face that is contained in `Hrepresentatives`.

INPUT:

- `Hrepresentatives` – facets or indices of Hrepresentatives; the indices are assumed to be the indices of the `Hrepresentation()`

OUTPUT: a `PolyhedronFace`

EXAMPLES:

```
sage: P = polytopes.permutahedron(5)
sage: P.meet_of_Hrep()
A 4-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of␣
↪120 vertices
sage: P.meet_of_Hrep(1)
A 3-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 24␣
↪vertices
sage: P.meet_of_Hrep(4)
A 3-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 12␣
↪vertices
sage: P.meet_of_Hrep(1,3,7)
A 1-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 2␣
↪vertices
sage: P.meet_of_Hrep(1,3,7).ambient_H_indices()
(0, 1, 3, 7)
```

```
>>> from sage.all import *
>>> P = polytopes.permutahedron(Integer(5))
>>> P.meet_of_Hrep()
A 4-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of␣
↪120 vertices
>>> P.meet_of_Hrep(Integer(1))
A 3-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 24␣
↪vertices
>>> P.meet_of_Hrep(Integer(4))
A 3-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 12␣
↪vertices
>>> P.meet_of_Hrep(Integer(1),Integer(3),Integer(7))
A 1-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 2␣
↪vertices
>>> P.meet_of_Hrep(Integer(1),Integer(3),Integer(7)).ambient_H_indices()
(0, 1, 3, 7)
```

The indices are the indices of the `Hrepresentation()`. `0` corresponds to an equation and is ignored:

```
sage: P.meet_of_Hrep(0)
A 4-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of␣
↪120 vertices
```

```
>>> from sage.all import *
>>> P.meet_of_Hrep(Integer(0))
A 4-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of␣
↪120 vertices
```

The input is flexible:

```
sage: P.meet_of_Hrep(P.facets()[-1], P.inequalities()[2], 7)
A 1-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 2␣
↪vertices
```

```
>>> from sage.all import *
>>> P.meet_of_Hrep(P.facets()[-Integer(1)], P.inequalities()[Integer(2)],␣
↪Integer(7))
A 1-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 2␣
↪vertices
```

The `Hrepresentatives` must belong to `self`:

```
sage: P = polytopes.cube(backend='ppl')
sage: Q = polytopes.cube(backend='field')
sage: f = P.facets()[0]
sage: P.meet_of_Hrep(f)
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices
sage: Q.meet_of_Hrep(f)
Traceback (most recent call last):
...
ValueError: not a facet of ``self``
sage: f = P.inequalities()[0]
sage: P.meet_of_Hrep(f)
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices
sage: Q.meet_of_Hrep(f)
Traceback (most recent call last):
...
ValueError: not a facet of ``self``
```

```
>>> from sage.all import *
>>> P = polytopes.cube(backend='ppl')
>>> Q = polytopes.cube(backend='field')
>>> f = P.facets()[Integer(0)]
>>> P.meet_of_Hrep(f)
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices
>>> Q.meet_of_Hrep(f)
Traceback (most recent call last):
...
ValueError: not a facet of ``self``
>>> f = P.inequalities()[Integer(0)]
>>> P.meet_of_Hrep(f)
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4␣
↪vertices
>>> Q.meet_of_Hrep(f)
Traceback (most recent call last):
...
ValueError: not a facet of ``self``
```

neighborliness()

Return the largest k, such that the polyhedron is k-neighborly.

A polyhedron is k-neighborly if every set of n vertices forms a face for n up to k.

In case of the d-dimensional simplex, it returns $d + 1$.

See also

is_neighborly()

EXAMPLES:

```
sage: cube = polytopes.cube()
sage: cube.neighborliness()
1
sage: P = Polyhedron(); P
The empty polyhedron in ZZ^0
sage: P.neighborliness()
0
sage: P = Polyhedron([[0]]); P
A 0-dimensional polyhedron in ZZ^1 defined as the convex hull of 1 vertex
sage: P.neighborliness()
1
sage: S = polytopes.simplex(5); S
A 5-dimensional polyhedron in ZZ^6 defined as the convex hull of 6 vertices
sage: S.neighborliness()
6
sage: C = polytopes.cyclic_polytope(7,10); C
A 7-dimensional polyhedron in QQ^7 defined as the convex hull of 10 vertices
sage: C.neighborliness()
3
sage: C = polytopes.cyclic_polytope(6,11); C
A 6-dimensional polyhedron in QQ^6 defined as the convex hull of 11 vertices
sage: C.neighborliness()
3
sage: [polytopes.cyclic_polytope(5,n).neighborliness() for n in range(6,10)]
[6, 2, 2, 2]
```

```
>>> from sage.all import *
>>> cube = polytopes.cube()
>>> cube.neighborliness()
1
>>> P = Polyhedron(); P
The empty polyhedron in ZZ^0
>>> P.neighborliness()
0
>>> P = Polyhedron([[Integer(0)]]); P
A 0-dimensional polyhedron in ZZ^1 defined as the convex hull of 1 vertex
>>> P.neighborliness()
1
>>> S = polytopes.simplex(Integer(5)); S
A 5-dimensional polyhedron in ZZ^6 defined as the convex hull of 6 vertices
```

(continues on next page)

(continued from previous page)

```
>>> S.neighborliness()
6
>>> C = polytopes.cyclic_polytope(Integer(7),Integer(10)); C
A 7-dimensional polyhedron in QQ^7 defined as the convex hull of 10 vertices
>>> C.neighborliness()
3
>>> C = polytopes.cyclic_polytope(Integer(6),Integer(11)); C
A 6-dimensional polyhedron in QQ^6 defined as the convex hull of 11 vertices
>>> C.neighborliness()
3
>>> [polytopes.cyclic_polytope(Integer(5),n).neighborliness() for n in␣
↪range(Integer(6),Integer(10))]
[6, 2, 2, 2]
```

simpliciality()

Return the largest integer k such that the polytope is k-simplicial.

A polytope is k-simplicial, if every k-face is a simplex. If `self` is a simplex, returns its dimension.

EXAMPLES:

```
sage: polytopes.cyclic_polytope(10,4).simpliciality()
3
sage: polytopes.hypersimplex(5,2).simpliciality()
2
sage: polytopes.cross_polytope(4).simpliciality()
3
sage: polytopes.simplex(3).simpliciality()
3
sage: polytopes.simplex(1).simpliciality()
1
```

```
>>> from sage.all import *
>>> polytopes.cyclic_polytope(Integer(10),Integer(4)).simpliciality()
3
>>> polytopes.hypersimplex(Integer(5),Integer(2)).simpliciality()
2
>>> polytopes.cross_polytope(Integer(4)).simpliciality()
3
>>> polytopes.simplex(Integer(3)).simpliciality()
3
>>> polytopes.simplex(Integer(1)).simpliciality()
1
```

The method is not implemented for unbounded polyhedra:

```
sage: p = Polyhedron(vertices=[(0,0)],rays=[(1,0),(0,1)])
sage: p.simpliciality()
Traceback (most recent call last):
...
NotImplementedError: this function is implemented for polytopes only
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices=[(Integer(0),Integer(0))],rays=[(Integer(1),
↪Integer(0)),(Integer(0),Integer(1))])
>>> p.simpliciality()
Traceback (most recent call last):
...
NotImplementedError: this function is implemented for polytopes only
```

simplicity()

Return the largest integer k such that the polytope is k-simple.

A polytope P is k-simple, if every $(d-1-k)$-face is contained in exactly $k+1$ facets of P for $1 \leq k \leq d-1$. Equivalently it is k-simple if the polar/dual polytope is k-simplicial. If `self` is a simplex, it returns its dimension.

EXAMPLES:

```
sage: polytopes.hypersimplex(4,2).simplicity()
1
sage: polytopes.hypersimplex(5,2).simplicity()
2
sage: polytopes.hypersimplex(6,2).simplicity()
3
sage: polytopes.simplex(3).simplicity()
3
sage: polytopes.simplex(1).simplicity()
1
```

```
>>> from sage.all import *
>>> polytopes.hypersimplex(Integer(4),Integer(2)).simplicity()
1
>>> polytopes.hypersimplex(Integer(5),Integer(2)).simplicity()
2
>>> polytopes.hypersimplex(Integer(6),Integer(2)).simplicity()
3
>>> polytopes.simplex(Integer(3)).simplicity()
3
>>> polytopes.simplex(Integer(1)).simplicity()
1
```

The method is not implemented for unbounded polyhedra:

```
sage: p = Polyhedron(vertices=[(0,0)],rays=[(1,0),(0,1)])
sage: p.simplicity()
Traceback (most recent call last):
...
NotImplementedError: this function is implemented for polytopes only
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices=[(Integer(0),Integer(0))],rays=[(Integer(1),
↪Integer(0)),(Integer(0),Integer(1))])
>>> p.simplicity()
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
NotImplementedError: this function is implemented for polytopes only
```

slack_matrix()

Return the slack matrix.

The entries correspond to the evaluation of the Hrepresentation elements on the Vrepresentation elements.

Note: The columns correspond to inequalities/equations in the order *Hrepresentation()*, the rows correspond to vertices/rays/lines in the order *Vrepresentation()*.

See also:

incidence_matrix().

EXAMPLES:

```
sage: P = polytopes.cube()
sage: P.slack_matrix()
[0 2 2 2 0 0]
[0 0 2 2 0 2]
[0 0 0 2 2 2]
[0 2 0 2 2 0]
[2 2 0 0 2 0]
[2 2 2 0 0 0]
[2 0 2 0 0 2]
[2 0 0 0 2 2]

sage: P = polytopes.cube(intervals='zero_one')
sage: P.slack_matrix()
[0 1 1 1 0 0]
[0 0 1 1 0 1]
[0 0 0 1 1 1]
[0 1 0 1 1 0]
[1 1 0 0 1 0]
[1 1 1 0 0 0]
[1 0 1 0 0 1]
[1 0 0 0 1 1]

sage: # needs sage.rings.number_field
sage: P = polytopes.dodecahedron().faces(2)[0].as_polyhedron()
sage: P.slack_matrix()
[1/2*sqrt5 - 1/2                0                0                1 1/2*sqrt5 -
↪1/2                0]
[               0                0 1/2*sqrt5 - 1/2 1/2*sqrt5 - 1/2                
↪ 1                0]
[               0 1/2*sqrt5 - 1/2                1                0 1/2*sqrt5 -
↪1/2                0]
```

(continues on next page)

(continued from previous page)

```
[              1 1/2*sqrt5 - 1/2              0 1/2*sqrt5 - 1/2              ␣
↪ 0              0]
[1/2*sqrt5 - 1/2              1 1/2*sqrt5 - 1/2              0              ␣
↪ 0              0]

sage: P = Polyhedron(rays=[[1, 0], [0, 1]])
sage: P.slack_matrix()
[0 0]
[0 1]
[1 0]
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> P.slack_matrix()
[0 2 2 2 0 0]
[0 0 2 2 0 2]
[0 0 0 2 2 2]
[0 2 0 2 2 0]
[2 2 0 0 2 0]
[2 2 2 0 0 0]
[2 0 2 0 0 2]
[2 0 0 0 2 2]

>>> P = polytopes.cube(intervals='zero_one')
>>> P.slack_matrix()
[0 1 1 1 0 0]
[0 0 1 1 0 1]
[0 0 0 1 1 1]
[0 1 0 1 1 0]
[1 1 0 0 1 0]
[1 1 1 0 0 0]
[1 0 1 0 0 1]
[1 0 0 0 1 1]

>>> # needs sage.rings.number_field
>>> P = polytopes.dodecahedron().faces(Integer(2))[Integer(0)].as_polyhedron()
>>> P.slack_matrix()
[1/2*sqrt5 - 1/2              0              0              1 1/2*sqrt5 -␣
↪1/2              0]
[              0              0 1/2*sqrt5 - 1/2 1/2*sqrt5 - 1/2              ␣
↪ 1              0]
[              0 1/2*sqrt5 - 1/2              1              0 1/2*sqrt5 -␣
↪1/2              0]
[              1 1/2*sqrt5 - 1/2              0 1/2*sqrt5 - 1/2              ␣
↪ 0              0]
[1/2*sqrt5 - 1/2              1 1/2*sqrt5 - 1/2              0              ␣
↪ 0              0]

>>> P = Polyhedron(rays=[[Integer(1), Integer(0)], [Integer(0), Integer(1)]])
>>> P.slack_matrix()
[0 0]
[0 1]
```

(continues on next page)

(continued from previous page)

```
[1 0]
```

vertex_adjacency_matrix(*algorithm=None*)

Return the binary matrix of vertex adjacencies.

INPUT:

- `algorithm` – string (optional); specify whether the face generator starts with facets or vertices:
  - `'primal'` – start with the facets
  - `'dual'` – start with the vertices
  - `None` – choose automatically

EXAMPLES:

```
sage: polytopes.simplex(4).vertex_adjacency_matrix()
[0 1 1 1 1]
[1 0 1 1 1]
[1 1 0 1 1]
[1 1 1 0 1]
[1 1 1 1 0]
```

```
>>> from sage.all import *
>>> polytopes.simplex(Integer(4)).vertex_adjacency_matrix()
[0 1 1 1 1]
[1 0 1 1 1]
[1 1 0 1 1]
[1 1 1 0 1]
[1 1 1 1 0]
```

The rows and columns of the vertex adjacency matrix correspond to the `Vrepresentation()` objects: vertices, rays, and lines. The (i, j) matrix entry equals 1 if the i-th and j-th V-representation object are adjacent.

Two vertices are adjacent if they are the endpoints of an edge, that is, a one-dimensional face. For unbounded polyhedra this clearly needs to be generalized and we define two V-representation objects (see *sage.geometry.polyhedron.constructor*) to be adjacent if they together generate a one-face. There are three possible combinations:

- Two vertices can bound a finite-length edge.
- A vertex and a ray can generate a half-infinite edge starting at the vertex and with the direction given by the ray.
- A vertex and a line can generate an infinite edge. The position of the vertex on the line is arbitrary in this case, only its transverse position matters. The direction of the edge is given by the line generator.

For example, take the half-plane:

```
sage: half_plane = Polyhedron(ieqs=[(0,1,0)])
sage: half_plane.Hrepresentation()
(An inequality (1, 0) x + 0 >= 0,)
```

```
>>> from sage.all import *
>>> half_plane = Polyhedron(ieqs=[(Integer(0),Integer(1),Integer(0))])
```

(continues on next page)

(continued from previous page)

```
>>> half_plane.Hrepresentation()
(An inequality (1, 0) x + 0 >= 0,)
```

Its (non-unique) V-representation consists of a vertex, a ray, and a line. The only edge is spanned by the vertex and the line generator, so they are adjacent:

```
sage: half_plane.Vrepresentation()
(A line in the direction (0, 1), A ray in the direction (1, 0), A vertex at␣
↪(0, 0))
sage: half_plane.vertex_adjacency_matrix()
[0 0 1]
[0 0 0]
[1 0 0]
```

```
>>> from sage.all import *
>>> half_plane.Vrepresentation()
(A line in the direction (0, 1), A ray in the direction (1, 0), A vertex at␣
↪(0, 0))
>>> half_plane.vertex_adjacency_matrix()
[0 0 1]
[0 0 0]
[1 0 0]
```

In one dimension higher, that is for a half-space in 3 dimensions, there is no one-dimensional face. Hence nothing is adjacent:

```
sage: Polyhedron(ieqs=[(0,1,0,0)]).vertex_adjacency_matrix()
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
```

```
>>> from sage.all import *
>>> Polyhedron(ieqs=[(Integer(0),Integer(1),Integer(0),Integer(0))]).vertex_
↪adjacency_matrix()
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
```

EXAMPLES:

In a bounded polygon, every vertex has precisely two adjacent ones:

```
sage: P = Polyhedron(vertices=[(0, 1), (1, 0), (3, 0), (4, 1)])
sage: for v in P.Vrep_generator():
....:     print("{} {}".format(P.adjacency_matrix().row(v.index()), v))
(0, 1, 0, 1) A vertex at (0, 1)
(1, 0, 1, 0) A vertex at (1, 0)
(0, 1, 0, 1) A vertex at (3, 0)
(1, 0, 1, 0) A vertex at (4, 1)
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[(Integer(0), Integer(1)), (Integer(1),␣
↪Integer(0)), (Integer(3), Integer(0)), (Integer(4), Integer(1))])
>>> for v in P.Vrep_generator():
...     print("{} {}".format(P.adjacency_matrix().row(v.index()), v))
(0, 1, 0, 1) A vertex at (0, 1)
(1, 0, 1, 0) A vertex at (1, 0)
(0, 1, 0, 1) A vertex at (3, 0)
(1, 0, 1, 0) A vertex at (4, 1)
```

If the V-representation of the polygon contains vertices and one ray, then each V-representation object is adjacent to two V-representation objects:

```
sage: P = Polyhedron(vertices=[(0, 1), (1, 0), (3, 0), (4, 1)],
....:                rays=[(0,1)])
sage: for v in P.Vrep_generator():
....:       print("{} {}".format(P.adjacency_matrix().row(v.index()), v))
(0, 1, 0, 0, 1) A ray in the direction (0, 1)
(1, 0, 1, 0, 0) A vertex at (0, 1)
(0, 1, 0, 1, 0) A vertex at (1, 0)
(0, 0, 1, 0, 1) A vertex at (3, 0)
(1, 0, 0, 1, 0) A vertex at (4, 1)
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[(Integer(0), Integer(1)), (Integer(1),␣
↪Integer(0)), (Integer(3), Integer(0)), (Integer(4), Integer(1))],
...                rays=[(Integer(0),Integer(1))])
>>> for v in P.Vrep_generator():
...       print("{} {}".format(P.adjacency_matrix().row(v.index()), v))
(0, 1, 0, 0, 1) A ray in the direction (0, 1)
(1, 0, 1, 0, 0) A vertex at (0, 1)
(0, 1, 0, 1, 0) A vertex at (1, 0)
(0, 0, 1, 0, 1) A vertex at (3, 0)
(1, 0, 0, 1, 0) A vertex at (4, 1)
```

If the V-representation of the polygon contains vertices and two distinct rays, then each vertex is adjacent to two V-representation objects (which can now be vertices or rays). The two rays are not adjacent to each other:

```
sage: P = Polyhedron(vertices=[(0, 1), (1, 0), (3, 0), (4, 1)],
....:                rays=[(0,1), (1,1)])
sage: for v in P.Vrep_generator():
....:     print("{} {}".format(P.adjacency_matrix().row(v.index()), v))
(0, 1, 0, 0, 0) A ray in the direction (0, 1)
(1, 0, 1, 0, 0) A vertex at (0, 1)
(0, 1, 0, 0, 1) A vertex at (1, 0)
(0, 0, 0, 0, 1) A ray in the direction (1, 1)
(0, 0, 1, 1, 0) A vertex at (3, 0)
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[(Integer(0), Integer(1)), (Integer(1),␣
↪Integer(0)), (Integer(3), Integer(0)), (Integer(4), Integer(1))],
...                rays=[(Integer(0),Integer(1)), (Integer(1),Integer(1))])
>>> for v in P.Vrep_generator():
```

(continues on next page)

(continued from previous page)

```
...     print("{} {}".format(P.adjacency_matrix().row(v.index()), v))
(0, 1, 0, 0, 0) A ray in the direction (0, 1)
(1, 0, 1, 0, 0) A vertex at (0, 1)
(0, 1, 0, 0, 1) A vertex at (1, 0)
(0, 0, 0, 0, 1) A ray in the direction (1, 1)
(0, 0, 1, 1, 0) A vertex at (3, 0)
```

The vertex adjacency matrix has base ring integers. This way one can express various counting questions:

```
sage: P = polytopes.cube()
sage: Q = P.stack(P.faces(2)[0])
sage: M = Q.vertex_adjacency_matrix()
sage: sum(M)
(4, 4, 3, 3, 4, 4, 4, 3, 3)
sage: G = Q.vertex_graph()                                              #␣
↪needs sage.graphs
sage: G.degree()                                                        #␣
↪needs sage.graphs
[4, 4, 3, 3, 4, 4, 4, 3, 3]
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> Q = P.stack(P.faces(Integer(2))[Integer(0)])
>>> M = Q.vertex_adjacency_matrix()
>>> sum(M)
(4, 4, 3, 3, 4, 4, 4, 3, 3)
>>> G = Q.vertex_graph()                                                #␣
↪needs sage.graphs
>>> G.degree()                                                          #␣
↪needs sage.graphs
[4, 4, 3, 3, 4, 4, 4, 3, 3]
```

2.6.5 Base class for polyhedra: Graph-theoretic methods

Define methods relying on `sage.graphs`.

class `sage.geometry.polyhedron.base4.`**`Polyhedron_base4`**(*parent*, *Vrep*, *Hrep*, *Vrep_minimal=None*, *Hrep_minimal=None*, *pref_rep=None*, *mutable=False*, ***kwds*)

Bases: *Polyhedron_base3*

Methods relying on `sage.graphs`.

See *sage.geometry.polyhedron.base.Polyhedron_base*.

`combinatorial_automorphism_group`(*vertex_graph_only=False*)

Compute the combinatorial automorphism group.

If `vertex_graph_only` is `True`, the automorphism group of the vertex-edge graph of the polyhedron is returned. Otherwise the automorphism group of the vertex-facet graph, which is isomorphic to the automorphism group of the face lattice is returned.

INPUT:

- vertex_graph_only – boolean (default: False); whether to return the automorphism group of the vertex edges graph or of the lattice

OUTPUT:

A PermutationGroup that is isomorphic to the combinatorial automorphism group is returned.

- if vertex_graph_only is True: The automorphism group of the vertex-edge graph of the polyhedron
- if vertex_graph_only is False (default): The automorphism group of the vertex-facet graph of the polyhedron, see *vertex_facet_graph()*. This group is isomorphic to the automorphism group of the face lattice of the polyhedron.

NOTE:

Depending on vertex_graph_only, this method returns groups that are not necessarily isomorphic, see the examples below.

See also

is_combinatorially_isomorphic(), *graph()*, *vertex_facet_graph()*.

EXAMPLES:

```
sage: quadrangle = Polyhedron(vertices=[(0,0),(1,0),(0,1),(2,3)])
sage: quadrangle.combinatorial_automorphism_group().is_isomorphic(           #␣
↪needs sage.groups
....:     groups.permutation.Dihedral(4))
True
sage: quadrangle.restricted_automorphism_group()                            #␣
↪needs sage.groups
Permutation Group with generators [()]
```

```
>>> from sage.all import *
>>> quadrangle = Polyhedron(vertices=[(Integer(0),Integer(0)),(Integer(1),
↪Integer(0)),(Integer(0),Integer(1)),(Integer(2),Integer(3))])
>>> quadrangle.combinatorial_automorphism_group().is_isomorphic(          #␣
↪needs sage.groups
...     groups.permutation.Dihedral(Integer(4)))
True
>>> quadrangle.restricted_automorphism_group()                           #␣
↪needs sage.groups
Permutation Group with generators [()]
```

Permutations of the vertex graph only exchange vertices with vertices:

```
sage: P = Polyhedron(vertices=[(1,0), (1,1)], rays=[(1,0)])
sage: P.combinatorial_automorphism_group(vertex_graph_only=True)            #␣
↪needs sage.groups
Permutation Group with generators [(A vertex at (1,0),A vertex at (1,1))]
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[(Integer(1),Integer(0)), (Integer(1),
↪Integer(1))], rays=[(Integer(1),Integer(0))])
>>> P.combinatorial_automorphism_group(vertex_graph_only=True)           #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.groups
Permutation Group with generators [(A vertex at (1,0),A vertex at (1,1))]
```

This shows an example of two polytopes whose vertex-edge graphs are isomorphic, but their face lattices are not isomorphic:

```
sage: # needs sage.groups
sage: Q = Polyhedron([[-123984206864/2768850730773, -101701330976/
↪922950243591, -64154618668/2768850730773, -2748446474675/2768850730773],
....:                [-11083969050/98314591817, -4717557075/98314591817, -
↪32618537490/98314591817, -91960210208/98314591817],
....:                [-9690950/554883199, -73651220/554883199, 1823050/
↪554883199, -549885101/554883199],
....:                [-5174928/72012097, 5436288/72012097, -37977984/
↪72012097, 60721345/72012097],
....:                [-19184/902877, 26136/300959, -21472/902877, 899005/
↪902877],
....:                [53511524/1167061933, 88410344/1167061933, 621795064/
↪1167061933, 982203941/1167061933],
....:                [4674489456/83665171433, -4026061312/83665171433,␣
↪28596876672/83665171433, -78383796375/83665171433],
....:                [857794884940/98972360190089, -10910202223200/
↪98972360190089, 2974263671400/98972360190089, -98320463346111/
↪98972360190089]])
sage: C = polytopes.cyclic_polytope(4,8)
sage: C.is_combinatorially_isomorphic(Q)
False
sage: C.combinatorial_automorphism_group(vertex_graph_only=True).is_
↪isomorphic(
....:     Q.combinatorial_automorphism_group(vertex_graph_only=True))
True
sage: C.combinatorial_automorphism_group(vertex_graph_only=False).is_
↪isomorphic(
....:     Q.combinatorial_automorphism_group(vertex_graph_only=False))
False
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> Q = Polyhedron([[-Integer(123984206864)/Integer(2768850730773), -
↪Integer(101701330976)/Integer(922950243591), -Integer(64154618668)/
↪Integer(2768850730773), -Integer(2748446474675)/Integer(2768850730773)],
...                [-Integer(11083969050)/Integer(98314591817), -
↪Integer(4717557075)/Integer(98314591817), -Integer(32618537490)/
↪Integer(98314591817), -Integer(91960210208)/Integer(98314591817)],
...                [-Integer(9690950)/Integer(554883199), -Integer(73651220)/
↪Integer(554883199), Integer(1823050)/Integer(554883199), -
↪Integer(549885101)/Integer(554883199)],
...                [-Integer(5174928)/Integer(72012097), Integer(5436288)/
↪Integer(72012097), -Integer(37977984)/Integer(72012097), Integer(60721345)/
↪Integer(72012097)],
...                [-Integer(19184)/Integer(902877), Integer(26136)/
↪Integer(300959), -Integer(21472)/Integer(902877), Integer(899005)/
```

(continues on next page)

(continued from previous page)

```
↪Integer(902877)],
...                  [Integer(53511524)/Integer(1167061933), Integer(88410344)/
↪Integer(1167061933), Integer(621795064)/Integer(1167061933),␣
↪Integer(982203941)/Integer(1167061933)],
...                  [Integer(4674489456)/Integer(83665171433), -
↪Integer(4026061312)/Integer(83665171433), Integer(28596876672)/
↪Integer(83665171433), -Integer(78383796375)/Integer(83665171433)],
...                  [Integer(857794884940)/Integer(98972360190089), -
↪Integer(10910202223200)/Integer(98972360190089), Integer(2974263671400)/
↪Integer(98972360190089), -Integer(98320463346111)/Integer(98972360190089)]])
>>> C = polytopes.cyclic_polytope(Integer(4),Integer(8))
>>> C.is_combinatorially_isomorphic(Q)
False
>>> C.combinatorial_automorphism_group(vertex_graph_only=True).is_isomorphic(
...     Q.combinatorial_automorphism_group(vertex_graph_only=True))
True
>>> C.combinatorial_automorphism_group(vertex_graph_only=False).is_isomorphic(
...     Q.combinatorial_automorphism_group(vertex_graph_only=False))
False
```

The automorphism group of the face lattice is isomorphic to the combinatorial automorphism group:

```
sage: # needs sage.groups
sage: CG = C.hasse_diagram().automorphism_group()
sage: C.combinatorial_automorphism_group().is_isomorphic(CG)
True
sage: QG = Q.hasse_diagram().automorphism_group()
sage: Q.combinatorial_automorphism_group().is_isomorphic(QG)
True
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> CG = C.hasse_diagram().automorphism_group()
>>> C.combinatorial_automorphism_group().is_isomorphic(CG)
True
>>> QG = Q.hasse_diagram().automorphism_group()
>>> Q.combinatorial_automorphism_group().is_isomorphic(QG)
True
```

face_lattice()

Return the face-lattice poset.

OUTPUT:

A `FinitePoset`. Elements are given as *PolyhedronFace*.

In the case of a full-dimensional polytope, the faces are pairs (vertices, inequalities) of the spanning vertices and corresponding saturated inequalities. In general, a face is defined by a pair (V-rep. objects, H-rep. objects). The V-representation objects span the face, and the corresponding H-representation objects are those inequalities and equations that are saturated on the face.

The bottom-most element of the face lattice is the "empty face". It contains no V-representation object. All H-representation objects are incident.

The top-most element is the "full face". It is spanned by all V-representation objects. The incident H-representation objects are all equations and no inequalities.

In the case of a full-dimensional polytope, the "empty face" and the "full face" are the empty set (no vertices, all inequalities) and the full polytope (all vertices, no inequalities), respectively.

ALGORITHM:

See *sage.geometry.polyhedron.combinatorial_polyhedron.face_iterator*.

Note

The face lattice is not cached, as long as this creates a memory leak, see Issue #28982.

EXAMPLES:

```
sage: square = polytopes.hypercube(2)
sage: fl = square.face_lattice();fl
Finite lattice containing 10 elements
sage: list(f.ambient_V_indices() for f in fl)
[(), (0,), (1,), (0, 1), (2,), (1, 2), (3,), (0, 3), (2, 3), (0, 1, 2, 3)]
sage: poset_element = fl[5]
sage: a_face = poset_element
sage: a_face
A 1-dimensional face of a Polyhedron in ZZ^2 defined as the convex hull of 2␣
↪vertices
sage: a_face.ambient_V_indices()
(1, 2)
sage: set(a_face.ambient_Vrepresentation()) ==              ....: set([square.
↪Vrepresentation(1), square.Vrepresentation(2)])
True
sage: a_face.ambient_Vrepresentation()
(A vertex at (1, 1), A vertex at (-1, 1))
sage: a_face.ambient_Hrepresentation()
(An inequality (0, -1) x + 1 >= 0,)
```

```
>>> from sage.all import *
>>> square = polytopes.hypercube(Integer(2))
>>> fl = square.face_lattice();fl
Finite lattice containing 10 elements
>>> list(f.ambient_V_indices() for f in fl)
[(), (0,), (1,), (0, 1), (2,), (1, 2), (3,), (0, 3), (2, 3), (0, 1, 2, 3)]
>>> poset_element = fl[Integer(5)]
>>> a_face = poset_element
>>> a_face
A 1-dimensional face of a Polyhedron in ZZ^2 defined as the convex hull of 2␣
↪vertices
>>> a_face.ambient_V_indices()
(1, 2)
>>> set(a_face.ambient_Vrepresentation()) ==              Ellipsis.:␣
↪set([square.Vrepresentation(Integer(1)), square.
↪Vrepresentation(Integer(2))])
True
>>> a_face.ambient_Vrepresentation()
```

(continues on next page)

(continued from previous page)

```
(A vertex at (1, 1), A vertex at (-1, 1))
>>> a_face.ambient_Hrepresentation()
(An inequality (0, -1) x + 1 >= 0,)
```

A more complicated example:

```
sage: c5_10 = Polyhedron(vertices = [[i,i^2,i^3,i^4,i^5] for i in range(1,
↪11)])
sage: c5_10_fl = c5_10.face_lattice()
sage: [len(x) for x in c5_10_fl.level_sets()]
[1, 10, 45, 100, 105, 42, 1]
```

```
>>> from sage.all import *
>>> c5_10 = Polyhedron(vertices = [[i,i**Integer(2),i**Integer(3),
↪i**Integer(4),i**Integer(5)] for i in range(Integer(1),Integer(11))])
>>> c5_10_fl = c5_10.face_lattice()
>>> [len(x) for x in c5_10_fl.level_sets()]
[1, 10, 45, 100, 105, 42, 1]
```

Note that if the polyhedron contains lines then there is a dimension gap between the empty face and the first non-empty face in the face lattice:

```
sage: line = Polyhedron(vertices=[(0,)], lines=[(1,)])
sage: [ fl.dim() for fl in line.face_lattice() ]
[-1, 1]
```

```
>>> from sage.all import *
>>> line = Polyhedron(vertices=[(Integer(0),)], lines=[(Integer(1),)])
>>> [ fl.dim() for fl in line.face_lattice() ]
[-1, 1]
```

flag_f_vector(**args*)

Return the flag f-vector.

For each $-1 < i_0 < \cdots < i_n < d$ the flag f-vector counts the number of flags $F_0 \subset \cdots \subset F_n$ with F_j of dimension i_j for each $0 \leq j \leq n$, where d is the dimension of the polyhedron.

INPUT:

- `args` – integer (optional); specify an entry of the flag-f-vector; must be an increasing sequence of integers

OUTPUT: a dictionary, if no arguments were given

- an Integer, if arguments were given

EXAMPLES:

Obtain the entire flag-f-vector:

```
sage: P = polytopes.twenty_four_cell()
sage: P.flag_f_vector()
    {(-1,): 1,
     (0,): 24,
     (0, 1): 192,
     (0, 1, 2): 576,
```

(continues on next page)

(continued from previous page)

```
     (0, 1, 2, 3): 1152,
     (0, 1, 3): 576,
     (0, 2): 288,
     (0, 2, 3): 576,
     (0, 3): 144,
     (1,): 96,
     (1, 2): 288,
     (1, 2, 3): 576,
     (1, 3): 288,
     (2,): 96,
     (2, 3): 192,
     (3,): 24,
     (4,): 1}
```

```
>>> from sage.all import *
>>> P = polytopes.twenty_four_cell()
>>> P.flag_f_vector()
    {(-1,): 1,
     (0,): 24,
     (0, 1): 192,
     (0, 1, 2): 576,
     (0, 1, 2, 3): 1152,
     (0, 1, 3): 576,
     (0, 2): 288,
     (0, 2, 3): 576,
     (0, 3): 144,
     (1,): 96,
     (1, 2): 288,
     (1, 2, 3): 576,
     (1, 3): 288,
     (2,): 96,
     (2, 3): 192,
     (3,): 24,
     (4,): 1}
```

Specify an entry:

```
sage: P.flag_f_vector(0,3)
144
sage: P.flag_f_vector(2)
96
```

```
>>> from sage.all import *
>>> P.flag_f_vector(Integer(0),Integer(3))
144
>>> P.flag_f_vector(Integer(2))
96
```

Leading `-1` and trailing entry of dimension are allowed:

```
sage: P.flag_f_vector(-1,0,3)
144
```

(continues on next page)

(continued from previous page)

```
sage: P.flag_f_vector(-1,0,3,4)
144
```

```
>>> from sage.all import *
>>> P.flag_f_vector(-Integer(1),Integer(0),Integer(3))
144
>>> P.flag_f_vector(-Integer(1),Integer(0),Integer(3),Integer(4))
144
```

One can get the number of trivial faces:

```
sage: P.flag_f_vector(-1)
1
sage: P.flag_f_vector(4)
1
```

```
>>> from sage.all import *
>>> P.flag_f_vector(-Integer(1))
1
>>> P.flag_f_vector(Integer(4))
1
```

Polyhedra with lines, have `0` entries accordingly:

```
sage: P = (Polyhedron(lines=[[1]]) * polytopes.cross_polytope(3))
sage: P.flag_f_vector()
{(-1,): 1,
 (0, 1): 0,
 (0, 1, 2): 0,
 (0, 1, 3): 0,
 (0, 2): 0,
 (0, 2, 3): 0,
 (0, 3): 0,
 (0,): 0,
 (1, 2): 24,
 (1, 2, 3): 48,
 (1, 3): 24,
 (1,): 6,
 (2, 3): 24,
 (2,): 12,
 (3,): 8,
 4: 1}
```

```
>>> from sage.all import *
>>> P = (Polyhedron(lines=[[Integer(1)]]) * polytopes.cross_
↪polytope(Integer(3)))
>>> P.flag_f_vector()
{(-1,): 1,
 (0, 1): 0,
 (0, 1, 2): 0,
 (0, 1, 3): 0,
 (0, 2): 0,
```

(continues on next page)

(continued from previous page)

```
 (0, 2, 3): 0,
 (0, 3): 0,
 (0,): 0,
 (1, 2): 24,
 (1, 2, 3): 48,
 (1, 3): 24,
 (1,): 6,
 (2, 3): 24,
 (2,): 12,
 (3,): 8,
 4: 1}
```

If the arguments are not stricly increasing or out of range, a key error is raised:

```
sage: P.flag_f_vector(-1,0,3,6)
Traceback (most recent call last):
...
KeyError: (0, 3, 6)
sage: P.flag_f_vector(-1,3,0)
Traceback (most recent call last):
...
KeyError: (3, 0)
```

```
>>> from sage.all import *
>>> P.flag_f_vector(-Integer(1),Integer(0),Integer(3),Integer(6))
Traceback (most recent call last):
...
KeyError: (0, 3, 6)
>>> P.flag_f_vector(-Integer(1),Integer(3),Integer(0))
Traceback (most recent call last):
...
KeyError: (3, 0)
```

graph(***kwds*)

Return a graph in which the vertices correspond to vertices of the polyhedron, and edges to edges.

INPUT:

- `names` – boolean (default: `True`); if `False`, then the nodes of the graph are labeld by the indices of the Vrepresentation
- `algorithm` – string (optional); specify whether the face generator starts with facets or vertices:
  - `'primal'` – start with the facets
  - `'dual'` – start with the vertices
  - `None` – choose automatically

Note

The graph of a polyhedron with lines has no vertices, as the polyhedron has no vertices (0-faces).

The method `vertices()` returns the defining points in this case.

EXAMPLES:

```
sage: g3 = polytopes.hypercube(3).vertex_graph(); g3
Graph on 8 vertices
sage: g3.automorphism_group().cardinality()                                    ␣
↪          # needs sage.groups
48
sage: s4 = polytopes.simplex(4).vertex_graph(); s4
Graph on 5 vertices
sage: s4.is_eulerian()
True
```

```
>>> from sage.all import *
>>> g3 = polytopes.hypercube(Integer(3)).vertex_graph(); g3
Graph on 8 vertices
>>> g3.automorphism_group().cardinality()                                      ␣
↪        # needs sage.groups
48
>>> s4 = polytopes.simplex(Integer(4)).vertex_graph(); s4
Graph on 5 vertices
>>> s4.is_eulerian()
True
```

The graph of an unbounded polyhedron is the graph of the bounded complex:

```
sage: open_triangle = Polyhedron(vertices=[[1,0], [0,1]],
....:                            rays    =[[1,1]])
sage: open_triangle.vertex_graph()
Graph on 2 vertices
```

```
>>> from sage.all import *
>>> open_triangle = Polyhedron(vertices=[[Integer(1),Integer(0)], [Integer(0),
↪Integer(1)]],
...                            rays    =[[Integer(1),Integer(1)]])
>>> open_triangle.vertex_graph()
Graph on 2 vertices
```

The graph of a polyhedron with lines has no vertices:

```
sage: line = Polyhedron(lines=[[0,1]])
sage: line.vertex_graph()
Graph on 0 vertices
```

```
>>> from sage.all import *
>>> line = Polyhedron(lines=[[Integer(0),Integer(1)]])
>>> line.vertex_graph()
Graph on 0 vertices
```

hasse_diagram()

Return the Hasse diagram of the face lattice of `self`.

This is the Hasse diagram of the poset of the faces of `self`.

OUTPUT: a directed graph

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: P = polytopes.regular_polygon(4).pyramid()
sage: D = P.hasse_diagram(); D
Digraph on 20 vertices
sage: D.degree_polynomial()
x^5 + x^4*y + x*y^4 + y^5 + 4*x^3*y + 8*x^2*y^2 + 4*x*y^3
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> P = polytopes.regular_polygon(Integer(4)).pyramid()
>>> D = P.hasse_diagram(); D
Digraph on 20 vertices
>>> D.degree_polynomial()
x^5 + x^4*y + x*y^4 + y^5 + 4*x^3*y + 8*x^2*y^2 + 4*x*y^3
```

Faces of a mutable polyhedron are not hashable. Hence those are not suitable as vertices of the hasse diagram. Use the combinatorial polyhedron instead:

```
sage: # needs sage.rings.number_field
sage: P = polytopes.regular_polygon(4).pyramid()
sage: parent = P.parent()
sage: parent = parent.change_ring(QQ, backend='ppl')
sage: Q = parent._element_constructor_(P, mutable=True)
sage: Q.hasse_diagram()
Traceback (most recent call last):
...
TypeError: mutable polyhedra are unhashable
sage: C = Q.combinatorial_polyhedron()
sage: D = C.hasse_diagram()
sage: set(D.vertices(sort=False)) == set(range(20))
True
sage: def index_to_combinatorial_face(n):
....:     return C.face_by_face_lattice_index(n)
sage: D.relabel(index_to_combinatorial_face, inplace=True)
sage: D.vertices(sort=True)
[A -1-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 1-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 1-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 1-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 1-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 1-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 1-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 1-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 1-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
```

(continues on next page)

(continued from previous page)

```
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 3-dimensional face of a 3-dimensional combinatorial polyhedron]
sage: D.degree_polynomial()
x^5 + x^4*y + x*y^4 + y^5 + 4*x^3*y + 8*x^2*y^2 + 4*x*y^3
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> P = polytopes.regular_polygon(Integer(4)).pyramid()
>>> parent = P.parent()
>>> parent = parent.change_ring(QQ, backend='ppl')
>>> Q = parent._element_constructor_(P, mutable=True)
>>> Q.hasse_diagram()
Traceback (most recent call last):
...
TypeError: mutable polyhedra are unhashable
>>> C = Q.combinatorial_polyhedron()
>>> D = C.hasse_diagram()
>>> set(D.vertices(sort=False)) == set(range(Integer(20)))
True
>>> def index_to_combinatorial_face(n):
...     return C.face_by_face_lattice_index(n)
>>> D.relabel(index_to_combinatorial_face, inplace=True)
>>> D.vertices(sort=True)
[A -1-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 0-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 1-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 1-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 1-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 1-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 1-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 1-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 1-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 1-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 2-dimensional face of a 3-dimensional combinatorial polyhedron,
 A 3-dimensional face of a 3-dimensional combinatorial polyhedron]
>>> D.degree_polynomial()
x^5 + x^4*y + x*y^4 + y^5 + 4*x^3*y + 8*x^2*y^2 + 4*x*y^3
```

is_combinatorially_isomorphic(*other*, *algorithm='bipartite_graph'*)

Return whether the polyhedron is combinatorially isomorphic to another polyhedron.

We only consider bounded polyhedra. By definition, they are combinatorially isomorphic if their face lattices are isomorphic.

INPUT:

- `other` – a polyhedron object
- `algorithm` – (default: `'bipartite_graph'`) the algorithm to use; the other possible value is `'face_lattice'`

OUTPUT:

- `True` if the two polyhedra are combinatorially isomorphic
- `False` otherwise

See also

combinatorial_automorphism_group(), *vertex_facet_graph()*.

REFERENCES:

For the equivalence of the two algorithms see [KK1995], p. 877-878

EXAMPLES:

The square is combinatorially isomorphic to the 2-dimensional cube:

```
sage: polytopes.hypercube(2).is_combinatorially_isomorphic(polytopes.regular_
↪polygon(4))
True
```

```
>>> from sage.all import *
>>> polytopes.hypercube(Integer(2)).is_combinatorially_isomorphic(polytopes.
↪regular_polygon(Integer(4)))
True
```

All the faces of the 3-dimensional permutahedron are either combinatorially isomorphic to a square or a hexagon:

```
sage: H = polytopes.regular_polygon(6)                                  #␣
↪needs sage.rings.number_field
sage: S = polytopes.hypercube(2)
sage: P = polytopes.permutahedron(4)
sage: all(F.as_polyhedron().is_combinatorially_isomorphic(S)           #␣
↪needs sage.rings.number_field
....:       or F.as_polyhedron().is_combinatorially_isomorphic(H)
....:     for F in P.faces(2))
True
```

```
>>> from sage.all import *
>>> H = polytopes.regular_polygon(Integer(6))                          ␣
↪     # needs sage.rings.number_field
>>> S = polytopes.hypercube(Integer(2))
>>> P = polytopes.permutahedron(Integer(4))
>>> all(F.as_polyhedron().is_combinatorially_isomorphic(S)            #␣
↪needs sage.rings.number_field
...       or F.as_polyhedron().is_combinatorially_isomorphic(H)
...     for F in P.faces(Integer(2)))
True
```

Checking that a regular simplex intersected with its reflection through the origin is combinatorially isomorphic to the intersection of a cube with a hyperplane perpendicular to its long diagonal:

```
sage: def simplex_intersection(k):
....:   S1 = Polyhedron([vector(v)-vector(polytopes.simplex(k).center()) for␣
↪v in polytopes.simplex(k).vertices_list()])
....:   S2 = Polyhedron([-vector(v) for v in S1.vertices_list()])
....:   return S1.intersection(S2)
sage: def cube_intersection(k):
....:    C = polytopes.hypercube(k+1)
....:    H = Polyhedron(eqns=[[0]+[1 for i in range(k+1)]])
....:    return C.intersection(H)
sage: [simplex_intersection(k).is_combinatorially_isomorphic(cube_
↪intersection(k)) for k in range(2,5)]
[True, True, True]
sage: simplex_intersection(2).is_combinatorially_isomorphic(polytopes.regular_
↪polygon(6))                        # needs sage.rings.number_field
True
sage: simplex_intersection(3).is_combinatorially_isomorphic(polytopes.
↪octahedron())
True
```

```
>>> from sage.all import *
>>> def simplex_intersection(k):
...   S1 = Polyhedron([vector(v)-vector(polytopes.simplex(k).center()) for v␣
↪in polytopes.simplex(k).vertices_list()])
...   S2 = Polyhedron([-vector(v) for v in S1.vertices_list()])
...   return S1.intersection(S2)
>>> def cube_intersection(k):
...    C = polytopes.hypercube(k+Integer(1))
...    H = Polyhedron(eqns=[[Integer(0)]+[Integer(1) for i in␣
↪range(k+Integer(1))]])
...    return C.intersection(H)
>>> [simplex_intersection(k).is_combinatorially_isomorphic(cube_
↪intersection(k)) for k in range(Integer(2),Integer(5))]
[True, True, True]
>>> simplex_intersection(Integer(2)).is_combinatorially_isomorphic(polytopes.
↪regular_polygon(Integer(6)))                   # needs sage.rings.number_
↪field
True
>>> simplex_intersection(Integer(3)).is_combinatorially_isomorphic(polytopes.
↪octahedron())
True
```

Two polytopes with the same f-vector, but different combinatorial types:

```
sage: P = Polyhedron([[-605520/1525633, -605520/1525633, -1261500/1525633, -
↪52200/1525633, 11833/1525633],\
....: [-720/1769, -600/1769, 1500/1769, 0, -31/1769], [-216/749, 240/749, -
↪240/749, -432/749, 461/749], \
....: [-50/181, 50/181, 60/181, -100/181, -119/181], [-32/51, -16/51, -4/51,␣
↪12/17, 1/17],\
....: [1, 0, 0, 0, 0], [16/129, 128/129, 0, 0, 1/129], [64/267, -128/267, 24/
```

(continues on next page)

(continued from previous page)

```
↪89, -128/267, 57/89],\
....: [1200/3953, -1200/3953, -1440/3953, -360/3953, -3247/3953], [1512/5597,␣
↪1512/5597, 588/5597, 4704/5597, 2069/5597]])
sage: C = polytopes.cyclic_polytope(5,10)
sage: C.f_vector() == P.f_vector(); C.f_vector()
True
(1, 10, 45, 100, 105, 42, 1)
sage: C.is_combinatorially_isomorphic(P)
False

sage: S = polytopes.simplex(3)
sage: S = S.face_truncation(S.faces(0)[3])
sage: S = S.face_truncation(S.faces(0)[4])
sage: S = S.face_truncation(S.faces(0)[5])
sage: T = polytopes.simplex(3)
sage: T = T.face_truncation(T.faces(0)[3])
sage: T = T.face_truncation(T.faces(0)[4])
sage: T = T.face_truncation(T.faces(0)[4])
sage: T.is_combinatorially_isomorphic(S)
False
sage: T.f_vector(), S.f_vector()
((1, 10, 15, 7, 1), (1, 10, 15, 7, 1))

sage: C = polytopes.hypercube(5)
sage: C.is_combinatorially_isomorphic(C)
True
sage: C.is_combinatorially_isomorphic(C, algorithm='magic')
Traceback (most recent call last):
...
AssertionError: `algorithm` must be 'bipartite graph' or 'face_lattice'

sage: G = Graph()
sage: C.is_combinatorially_isomorphic(G)
Traceback (most recent call last):
...
AssertionError: input `other` must be a polyhedron

sage: H = Polyhedron(eqns=[[0,1,1,1,1]]); H
A 3-dimensional polyhedron in QQ^4 defined as the convex hull of 1 vertex and␣
↪3 lines
sage: C.is_combinatorially_isomorphic(H)
Traceback (most recent call last):
...
AssertionError: polyhedron `other` must be bounded
```

```
>>> from sage.all import *
>>> P = Polyhedron([[-Integer(605520)/Integer(1525633), -Integer(605520)/
↪Integer(1525633), -Integer(1261500)/Integer(1525633), -Integer(52200)/
↪Integer(1525633), Integer(11833)/Integer(1525633)],[-Integer(720)/
↪Integer(1769), -Integer(600)/Integer(1769), Integer(1500)/Integer(1769),␣
↪Integer(0), -Integer(31)/Integer(1769)], [-Integer(216)/Integer(749),␣
↪Integer(240)/Integer(749), -Integer(240)/Integer(749), -Integer(432)/
```

(continues on next page)

(continued from previous page)

```
↪Integer(749), Integer(461)/Integer(749)], [-Integer(50)/Integer(181),␣
↪Integer(50)/Integer(181), Integer(60)/Integer(181), -Integer(100)/
↪Integer(181), -Integer(119)/Integer(181)], [-Integer(32)/Integer(51), -
↪Integer(16)/Integer(51), -Integer(4)/Integer(51), Integer(12)/Integer(17),␣
↪Integer(1)/Integer(17)],[Integer(1), Integer(0), Integer(0), Integer(0),␣
↪Integer(0)], [Integer(16)/Integer(129), Integer(128)/Integer(129),␣
↪Integer(0), Integer(0), Integer(1)/Integer(129)], [Integer(64)/Integer(267),
↪ -Integer(128)/Integer(267), Integer(24)/Integer(89), -Integer(128)/
↪Integer(267), Integer(57)/Integer(89)],[Integer(1200)/Integer(3953), -
↪Integer(1200)/Integer(3953), -Integer(1440)/Integer(3953), -Integer(360)/
↪Integer(3953), -Integer(3247)/Integer(3953)], [Integer(1512)/Integer(5597),␣
↪Integer(1512)/Integer(5597), Integer(588)/Integer(5597), Integer(4704)/
↪Integer(5597), Integer(2069)/Integer(5597)]])
>>> C = polytopes.cyclic_polytope(Integer(5),Integer(10))
>>> C.f_vector() == P.f_vector(); C.f_vector()
True
(1, 10, 45, 100, 105, 42, 1)
>>> C.is_combinatorially_isomorphic(P)
False

>>> S = polytopes.simplex(Integer(3))
>>> S = S.face_truncation(S.faces(Integer(0))[Integer(3)])
>>> S = S.face_truncation(S.faces(Integer(0))[Integer(4)])
>>> S = S.face_truncation(S.faces(Integer(0))[Integer(5)])
>>> T = polytopes.simplex(Integer(3))
>>> T = T.face_truncation(T.faces(Integer(0))[Integer(3)])
>>> T = T.face_truncation(T.faces(Integer(0))[Integer(4)])
>>> T = T.face_truncation(T.faces(Integer(0))[Integer(4)])
>>> T.is_combinatorially_isomorphic(S)
False
>>> T.f_vector(), S.f_vector()
((1, 10, 15, 7, 1), (1, 10, 15, 7, 1))

>>> C = polytopes.hypercube(Integer(5))
>>> C.is_combinatorially_isomorphic(C)
True
>>> C.is_combinatorially_isomorphic(C, algorithm='magic')
Traceback (most recent call last):
...
AssertionError: `algorithm` must be 'bipartite graph' or 'face_lattice'

>>> G = Graph()
>>> C.is_combinatorially_isomorphic(G)
Traceback (most recent call last):
...
AssertionError: input `other` must be a polyhedron

>>> H = Polyhedron(eqns=[[Integer(0),Integer(1),Integer(1),Integer(1),
↪Integer(1)]]); H
A 3-dimensional polyhedron in QQ^4 defined as the convex hull of 1 vertex and␣
↪3 lines
>>> C.is_combinatorially_isomorphic(H)
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
AssertionError: polyhedron `other` must be bounded
```

is_self_dual()

Return whether the polytope is self-dual.

A polytope is self-dual if its face lattice is isomorphic to the face lattice of its dual polytope.

EXAMPLES:

```
sage: polytopes.simplex().is_self_dual()
True
sage: polytopes.twenty_four_cell().is_self_dual()
True
sage: polytopes.cube().is_self_dual()
False
sage: polytopes.hypersimplex(5,2).is_self_dual()                          #␣
↪needs sage.combinat
False
sage: P = Polyhedron(vertices=[[1/2, 1/3]], rays=[[1, 1]]).is_self_dual()
Traceback (most recent call last):
...
ValueError: polyhedron has to be compact
```

```
>>> from sage.all import *
>>> polytopes.simplex().is_self_dual()
True
>>> polytopes.twenty_four_cell().is_self_dual()
True
>>> polytopes.cube().is_self_dual()
False
>>> polytopes.hypersimplex(Integer(5),Integer(2)).is_self_dual()                ␣
↪              # needs sage.combinat
False
>>> P = Polyhedron(vertices=[[Integer(1)/Integer(2), Integer(1)/Integer(3)]],␣
↪rays=[[Integer(1), Integer(1)]]).is_self_dual()
Traceback (most recent call last):
...
ValueError: polyhedron has to be compact
```

restricted_automorphism_group(*output='abstract'*)

Return the restricted automorphism group.

First, let the linear automorphism group be the subgroup of the affine group $AGL(d, \mathbf{R}) = GL(d, \mathbf{R}) \ltimes \mathbf{R}^d$ preserving the d-dimensional polyhedron. The affine group acts in the usual way $\vec{x} \mapsto A\vec{x} + b$ on the ambient space.

The restricted automorphism group is the subgroup of the linear automorphism group generated by permutations of the generators of the same type. That is, vertices can only be permuted with vertices, ray generators with ray generators, and line generators with line generators.

For example, take the first quadrant

$$Q = \Big\{ (x, y) \Big| x \geq 0,\ y \geq 0 \Big\} \subset \mathbf{Q}^2$$

Then the linear automorphism group is

$$\mathrm{Aut}(Q) = \left\{ \begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}, \begin{pmatrix} 0 & c \\ d & 0 \end{pmatrix} : a, b, c, d \in \mathbf{Q}_{>0} \right\} \subset GL(2, \mathbf{Q}) \subset E(d)$$

Note that there are no translations that map the quadrant Q to itself, so the linear automorphism group is contained in the general linear group (the subgroup of transformations preserving the origin). The restricted automorphism group is

$$\mathrm{Aut}(Q) = \left\{ \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \right\} \simeq \mathbf{Z}_2$$

INPUT:

- `output` – how the group should be represented:
  - `'abstract'` – default; return an abstract permutation group without further meaning
  - `'permutation'` – return a permutation group on the indices of the polyhedron generators. For example, the permutation `(0,1)` would correspond to swapping `self.Vrepresentation(0)` and `self.Vrepresentation(1)`.
  - `'matrix'` – return a matrix group representing affine transformations. When acting on affine vectors, you should append a 1 to every vector. If the polyhedron is not full dimensional, the returned matrices act as the identity on the orthogonal complement of the affine space spanned by the polyhedron.
  - `'matrixlist'` – like `matrix`, but return the list of elements of the matrix group. Useful for fields without a good implementation of matrix groups or to avoid the overhead of creating the group.

OUTPUT:

- For `output="abstract"` and `output="permutation"`: a `PermutationGroup`.
- For `output="matrix"`: a `MatrixGroup()`.
- For `output="matrixlist"`: a list of matrices.

REFERENCES:

- [BSS2009]

EXAMPLES:

A cross-polytope example:

```
sage: # needs sage.groups
sage: P = polytopes.cross_polytope(3)
sage: P.restricted_automorphism_group() == PermutationGroup([[(3,4)], [(2,3),
↪(4,5)],[(2,5)],[(1,2),(5,6)],[(1,6)]])
True
sage: P.restricted_automorphism_group(output='permutation') ==␣
↪PermutationGroup([[(2,3)],[(1,2),(3,4)],[(1,4)],[(0,1),(4,5)],[(0,5)]])
True
sage: mgens = [[[1,0,0,0],[0,1,0,0],[0,0,-1,0],[0,0,0,1]], [[1,0,0,0],[0,0,1,
↪0],[0,1,0,0],[0,0,0,1]], [[0,1,0,0],[1,0,0,0],[0,0,1,0],[0,0,0,1]]]
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> P = polytopes.cross_polytope(Integer(3))
>>> P.restricted_automorphism_group() == PermutationGroup([[(Integer(3),
```

(continues on next page)

(continued from previous page)

```
↪Integer(4))], [(Integer(2),Integer(3)),(Integer(4),Integer(5))],
↪[(Integer(2),Integer(5))],[(Integer(1),Integer(2)),(Integer(5),Integer(6))],
↪[(Integer(1),Integer(6))]])
True
>>> P.restricted_automorphism_group(output='permutation') ==␣
↪PermutationGroup([[(Integer(2),Integer(3))],[(Integer(1),Integer(2)),
↪(Integer(3),Integer(4))],[(Integer(1),Integer(4))],[(Integer(0),Integer(1)),
↪(Integer(4),Integer(5))],[(Integer(0),Integer(5))]])
True
>>> mgens = [[[Integer(1),Integer(0),Integer(0),Integer(0)],[Integer(0),
↪Integer(1),Integer(0),Integer(0)],[Integer(0),Integer(0),-Integer(1),
↪Integer(0)],[Integer(0),Integer(0),Integer(0),Integer(1)]], [[Integer(1),
↪Integer(0),Integer(0),Integer(0)],[Integer(0),Integer(0),Integer(1),
↪Integer(0)],[Integer(0),Integer(1),Integer(0),Integer(0)],[Integer(0),
↪Integer(0),Integer(0),Integer(1)]], [[Integer(0),Integer(1),Integer(0),
↪Integer(0)],[Integer(1),Integer(0),Integer(0),Integer(0)],[Integer(0),
↪Integer(0),Integer(1),Integer(0)],[Integer(0),Integer(0),Integer(0),
↪Integer(1)]]]
```

We test groups for equality in a fool-proof way; they can have different generators, etc:

```
sage: # needs sage.groups
sage: poly_g = P.restricted_automorphism_group(output='matrix')
sage: matrix_g = MatrixGroup([matrix(QQ,t) for t in mgens])
sage: all(t.matrix() in poly_g for t in matrix_g.gens())
True
sage: all(t.matrix() in matrix_g for t in poly_g.gens())
True
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> poly_g = P.restricted_automorphism_group(output='matrix')
>>> matrix_g = MatrixGroup([matrix(QQ,t) for t in mgens])
>>> all(t.matrix() in poly_g for t in matrix_g.gens())
True
>>> all(t.matrix() in matrix_g for t in poly_g.gens())
True
```

24-cell example:

```
sage: # needs sage.groups
sage: P24 = polytopes.twenty_four_cell()
sage: AutP24 = P24.restricted_automorphism_group()
sage: PermutationGroup([
....:     '(1,20,2,24,5,23)(3,18,10,19,4,14)(6,21,11,22,7,15)(8,12,16,17,13,9)
↪',
....:     '(1,21,8,24,4,17)(2,11,6,15,9,13)(3,20)(5,22)(10,16,12,23,14,19)'
....: ]).is_isomorphic(AutP24)
True
sage: AutP24.order()
1152
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> P24 = polytopes.twenty_four_cell()
>>> AutP24 = P24.restricted_automorphism_group()
>>> PermutationGroup([
...     '(1,20,2,24,5,23)(3,18,10,19,4,14)(6,21,11,22,7,15)(8,12,16,17,13,9)',
...     '(1,21,8,24,4,17)(2,11,6,15,9,13)(3,20)(5,22)(10,16,12,23,14,19)'
... ]).is_isomorphic(AutP24)
True
>>> AutP24.order()
1152
```

Here is the quadrant example mentioned in the beginning:

```
sage: # needs sage.groups
sage: P = Polyhedron(rays=[(1,0),(0,1)])
sage: P.Vrepresentation()
(A vertex at (0, 0), A ray in the direction (0, 1), A ray in the direction (1,
↪ 0))
sage: P.restricted_automorphism_group(output='permutation')
Permutation Group with generators [(1,2)]
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> P = Polyhedron(rays=[(Integer(1),Integer(0)),(Integer(0),Integer(1))])
>>> P.Vrepresentation()
(A vertex at (0, 0), A ray in the direction (0, 1), A ray in the direction (1,
↪ 0))
>>> P.restricted_automorphism_group(output='permutation')
Permutation Group with generators [(1,2)]
```

Also, the polyhedron need not be full-dimensional:

```
sage: # needs sage.groups
sage: P = Polyhedron(vertices=[(1,2,3,4,5),(7,8,9,10,11)])
sage: P.restricted_automorphism_group()
Permutation Group with generators [(1,2)]
sage: G = P.restricted_automorphism_group(output='matrixlist'); G
(
[1 0 0 0 0 0]  [ -87/55  -82/55    -2/5   38/55   98/55   12/11]
[0 1 0 0 0 0]  [-142/55  -27/55    -2/5   38/55   98/55   12/11]
[0 0 1 0 0 0]  [-142/55  -82/55     3/5   38/55   98/55   12/11]
[0 0 0 1 0 0]  [-142/55  -82/55    -2/5   93/55   98/55   12/11]
[0 0 0 0 1 0]  [-142/55  -82/55    -2/5   38/55  153/55   12/11]
[0 0 0 0 0 1], [      0       0       0       0       0       1]
)
sage: g = AffineGroup(5, QQ)(G[1]); g
      [ -87/55  -82/55    -2/5   38/55   98/55]      [12/11]
      [-142/55  -27/55    -2/5   38/55   98/55]      [12/11]
x |-> [-142/55  -82/55     3/5   38/55   98/55] x + [12/11]
      [-142/55  -82/55    -2/5   93/55   98/55]      [12/11]
      [-142/55  -82/55    -2/5   38/55  153/55]      [12/11]
sage: g^2
```

(continues on next page)

(continued from previous page)

```
      [1 0 0 0 0]     [0]
      [0 1 0 0 0]     [0]
x |-> [0 0 1 0 0] x + [0]
      [0 0 0 1 0]     [0]
      [0 0 0 0 1]     [0]
sage: g(list(P.vertices()[0]))
(7, 8, 9, 10, 11)
sage: g(list(P.vertices()[1]))
(1, 2, 3, 4, 5)
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> P = Polyhedron(vertices=[(Integer(1),Integer(2),Integer(3),Integer(4),
↪Integer(5)),(Integer(7),Integer(8),Integer(9),Integer(10),Integer(11))])
>>> P.restricted_automorphism_group()
Permutation Group with generators [(1,2)]
>>> G = P.restricted_automorphism_group(output='matrixlist'); G
(
[1 0 0 0 0 0]  [ -87/55  -82/55    -2/5   38/55   98/55   12/11]
[0 1 0 0 0 0]  [-142/55  -27/55    -2/5   38/55   98/55   12/11]
[0 0 1 0 0 0]  [-142/55  -82/55     3/5   38/55   98/55   12/11]
[0 0 0 1 0 0]  [-142/55  -82/55    -2/5   93/55   98/55   12/11]
[0 0 0 0 1 0]  [-142/55  -82/55    -2/5   38/55  153/55   12/11]
[0 0 0 0 0 1], [      0       0       0       0       0       1]
)
>>> g = AffineGroup(Integer(5), QQ)(G[Integer(1)]); g
      [ -87/55  -82/55    -2/5   38/55   98/55]     [12/11]
      [-142/55  -27/55    -2/5   38/55   98/55]     [12/11]
x |-> [-142/55  -82/55     3/5   38/55   98/55] x + [12/11]
      [-142/55  -82/55    -2/5   93/55   98/55]     [12/11]
      [-142/55  -82/55    -2/5   38/55  153/55]     [12/11]
>>> g**Integer(2)
      [1 0 0 0 0]     [0]
      [0 1 0 0 0]     [0]
x |-> [0 0 1 0 0] x + [0]
      [0 0 0 1 0]     [0]
      [0 0 0 0 1]     [0]
>>> g(list(P.vertices()[Integer(0)]))
(7, 8, 9, 10, 11)
>>> g(list(P.vertices()[Integer(1)]))
(1, 2, 3, 4, 5)
```

Affine transformations do not change the restricted automorphism group. For example, any non-degenerate triangle has the dihedral group with 6 elements, D_6, as its automorphism group:

```
sage: # needs sage.groups
sage: initial_points = [vector([1,0]), vector([0,1]), vector([-2,-1])]
sage: points = initial_points
sage: Polyhedron(vertices=points).restricted_automorphism_group()
Permutation Group with generators [(2,3), (1,2)]
sage: points = [pt - initial_points[0] for pt in initial_points]
sage: Polyhedron(vertices=points).restricted_automorphism_group()
```

(continues on next page)

(continued from previous page)

```
Permutation Group with generators [(2,3), (1,2)]
sage: points = [pt - initial_points[1] for pt in initial_points]
sage: Polyhedron(vertices=points).restricted_automorphism_group()
Permutation Group with generators [(2,3), (1,2)]
sage: points = [pt - 2*initial_points[1] for pt in initial_points]
sage: Polyhedron(vertices=points).restricted_automorphism_group()
Permutation Group with generators [(2,3), (1,2)]
```

```
>>> from sage.all import *
>>> # needs sage.groups
>>> initial_points = [vector([Integer(1),Integer(0)]), vector([Integer(0),
↪Integer(1)]), vector([-Integer(2),-Integer(1)])]
>>> points = initial_points
>>> Polyhedron(vertices=points).restricted_automorphism_group()
Permutation Group with generators [(2,3), (1,2)]
>>> points = [pt - initial_points[Integer(0)] for pt in initial_points]
>>> Polyhedron(vertices=points).restricted_automorphism_group()
Permutation Group with generators [(2,3), (1,2)]
>>> points = [pt - initial_points[Integer(1)] for pt in initial_points]
>>> Polyhedron(vertices=points).restricted_automorphism_group()
Permutation Group with generators [(2,3), (1,2)]
>>> points = [pt - Integer(2)*initial_points[Integer(1)] for pt in initial_
↪points]
>>> Polyhedron(vertices=points).restricted_automorphism_group()
Permutation Group with generators [(2,3), (1,2)]
```

The `output="matrixlist"` can be used over fields without a complete implementation of matrix groups:

```
sage: # needs sage.groups sage.rings.number_field
sage: P = polytopes.dodecahedron(); P
A 3-dimensional polyhedron in (Number Field in sqrt5 with defining
 polynomial x^2 - 5 with sqrt5 = 2.236067977499790?)^3
 defined as the convex hull of 20 vertices
sage: G = P.restricted_automorphism_group(output='matrixlist')
sage: len(G)
120
```

```
>>> from sage.all import *
>>> # needs sage.groups sage.rings.number_field
>>> P = polytopes.dodecahedron(); P
A 3-dimensional polyhedron in (Number Field in sqrt5 with defining
 polynomial x^2 - 5 with sqrt5 = 2.236067977499790?)^3
 defined as the convex hull of 20 vertices
>>> G = P.restricted_automorphism_group(output='matrixlist')
>>> len(G)
120
```

Floating-point computations are supported with a simple fuzzy zero implementation:

```
sage: P = Polyhedron(vertices=[(1/3,0,0,1),(0,1/4,0,1),(0,0,1/5,1)],
....:                base_ring=RDF)
sage: P.restricted_automorphism_group()                                   #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.groups
Permutation Group with generators [(2,3), (1,2)]
sage: len(P.restricted_automorphism_group(output='matrixlist'))
6
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[(Integer(1)/Integer(3),Integer(0),Integer(0),
↪Integer(1)),(Integer(0),Integer(1)/Integer(4),Integer(0),Integer(1)),
↪(Integer(0),Integer(0),Integer(1)/Integer(5),Integer(1))],
...                base_ring=RDF)
>>> P.restricted_automorphism_group()                                    #␣
↪needs sage.groups
Permutation Group with generators [(2,3), (1,2)]
>>> len(P.restricted_automorphism_group(output='matrixlist'))
6
```

vertex_digraph(*f*, *increasing=True*)

Return the directed graph of the polyhedron according to a linear form.

The underlying undirected graph is the graph of vertices and edges.

INPUT:

- `f` – a linear form. The linear form can be provided as:
  - a vector space morphism with one-dimensional codomain, (see `sage.modules.vector_space_morphism.linear_transformation()` and `sage.modules.vector_space_morphism.VectorSpaceMorphism`)
  - a vector ; in this case the linear form is obtained by duality using the dot product: `f(v) = v.dot_product(f)`.
- `increasing` – boolean (default: `True`); whether to orient edges in the increasing or decreasing direction

By default, an edge is oriented from v to w if $f(v) \leq f(w)$.

If $f(v) = f(w)$, then two opposite edges are created.

EXAMPLES:

```
sage: penta = Polyhedron([[0,0],[1,0],[0,1],[1,2],[3,2]])
sage: G = penta.vertex_digraph(vector([1,1])); G
Digraph on 5 vertices
sage: G.sinks()
[A vertex at (3, 2)]

sage: A = matrix(ZZ, [[1], [-1]])
sage: f = linear_transformation(A)
sage: G = penta.vertex_digraph(f) ; G
Digraph on 5 vertices
sage: G.is_directed_acyclic()
False
```

```
>>> from sage.all import *
>>> penta = Polyhedron([[Integer(0),Integer(0)],[Integer(1),Integer(0)],
↪[Integer(0),Integer(1)],[Integer(1),Integer(2)],[Integer(3),Integer(2)]])
```

(continues on next page)

(continued from previous page)

```
>>> G = penta.vertex_digraph(vector([Integer(1),Integer(1)])); G
Digraph on 5 vertices
>>> G.sinks()
[A vertex at (3, 2)]

>>> A = matrix(ZZ, [[Integer(1)], [-Integer(1)]])
>>> f = linear_transformation(A)
>>> G = penta.vertex_digraph(f) ; G
Digraph on 5 vertices
>>> G.is_directed_acyclic()
False
```

See also

vertex_graph()

vertex_facet_graph(*labels=True*)

Return the vertex-facet graph.

This function constructs a directed bipartite graph. The nodes of the graph correspond to the vertices of the polyhedron and the facets of the polyhedron. There is a directed edge from a vertex to a face if and only if the vertex is incident to the face.

INPUT:

- `labels` – boolean (default: `True`); decide how the nodes of the graph are labelled. Either with the original vertices/facets of the Polyhedron or with integers.

OUTPUT:

- a bipartite DiGraph. If `labels` is `True`, then the nodes of the graph will actually be the vertices and facets of `self`, otherwise they will be integers.

See also

combinatorial_automorphism_group(), *is_combinatorially_isomorphic()*.

EXAMPLES:

```
sage: P = polytopes.cube()
sage: G = P.vertex_facet_graph(); G
Digraph on 14 vertices
sage: G.vertices(sort=True, key=lambda v: str(v))
[A vertex at (-1, -1, -1),
 A vertex at (-1, -1, 1),
 A vertex at (-1, 1, -1),
 A vertex at (-1, 1, 1),
 A vertex at (1, -1, -1),
 A vertex at (1, -1, 1),
 A vertex at (1, 1, -1),
 A vertex at (1, 1, 1),
 An inequality (-1, 0, 0) x + 1 >= 0,
```

(continues on next page)

(continued from previous page)

```
 An inequality (0, -1, 0) x + 1 >= 0,
 An inequality (0, 0, -1) x + 1 >= 0,
 An inequality (0, 0, 1) x + 1 >= 0,
 An inequality (0, 1, 0) x + 1 >= 0,
 An inequality (1, 0, 0) x + 1 >= 0]
sage: G.automorphism_group().is_isomorphic(P.hasse_diagram().automorphism_
↪group())       # needs sage.groups
True
sage: O = polytopes.octahedron(); O
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 6 vertices
sage: O.vertex_facet_graph()
Digraph on 14 vertices
sage: H = O.vertex_facet_graph()
sage: G.is_isomorphic(H)                                                    ␣
↪           # needs sage.groups
False
sage: G2 = copy(G)
sage: G2.reverse_edges(G2.edges(sort=True))
sage: G2.is_isomorphic(H)                                                   ␣
↪           # needs sage.groups
True
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> G = P.vertex_facet_graph(); G
Digraph on 14 vertices
>>> G.vertices(sort=True, key=lambda v: str(v))
[A vertex at (-1, -1, -1),
 A vertex at (-1, -1, 1),
 A vertex at (-1, 1, -1),
 A vertex at (-1, 1, 1),
 A vertex at (1, -1, -1),
 A vertex at (1, -1, 1),
 A vertex at (1, 1, -1),
 A vertex at (1, 1, 1),
 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 (0, 0, 1) x + 1 >= 0,
 An inequality (0, 1, 0) x + 1 >= 0,
 An inequality (1, 0, 0) x + 1 >= 0]
>>> G.automorphism_group().is_isomorphic(P.hasse_diagram().automorphism_
↪group())       # needs sage.groups
True
>>> O = polytopes.octahedron(); O
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 6 vertices
>>> O.vertex_facet_graph()
Digraph on 14 vertices
>>> H = O.vertex_facet_graph()
>>> G.is_isomorphic(H)                                                      ␣
↪         # needs sage.groups
False
```

(continues on next page)

(continued from previous page)

```
>>> G2 = copy(G)
>>> G2.reverse_edges(G2.edges(sort=True))
>>> G2.is_isomorphic(H)                                                    ␣
↪        # needs sage.groups
True
```

vertex_graph(***kwds*)

Return a graph in which the vertices correspond to vertices of the polyhedron, and edges to edges.

INPUT:

- `names` – boolean (default: `True`); if `False`, then the nodes of the graph are labeld by the indices of the Vrepresentation
- `algorithm` – string (optional); specify whether the face generator starts with facets or vertices:
  - `'primal'` – start with the facets
  - `'dual'` – start with the vertices
  - `None` – choose automatically

Note

The graph of a polyhedron with lines has no vertices, as the polyhedron has no vertices (0-faces).

The method *vertices()* returns the defining points in this case.

EXAMPLES:

```
sage: g3 = polytopes.hypercube(3).vertex_graph(); g3
Graph on 8 vertices
sage: g3.automorphism_group().cardinality()                                ␣
↪          # needs sage.groups
48
sage: s4 = polytopes.simplex(4).vertex_graph(); s4
Graph on 5 vertices
sage: s4.is_eulerian()
True
```

```
>>> from sage.all import *
>>> g3 = polytopes.hypercube(Integer(3)).vertex_graph(); g3
Graph on 8 vertices
>>> g3.automorphism_group().cardinality()                                  ␣
↪        # needs sage.groups
48
>>> s4 = polytopes.simplex(Integer(4)).vertex_graph(); s4
Graph on 5 vertices
>>> s4.is_eulerian()
True
```

The graph of an unbounded polyhedron is the graph of the bounded complex:

```
sage: open_triangle = Polyhedron(vertices=[[1,0], [0,1]],
....:                            rays    =[[1,1]])
sage: open_triangle.vertex_graph()
Graph on 2 vertices
```

```
>>> from sage.all import *
>>> open_triangle = Polyhedron(vertices=[[Integer(1),Integer(0)], [Integer(0),
↪Integer(1)]],
...                            rays    =[[Integer(1),Integer(1)]])
>>> open_triangle.vertex_graph()
Graph on 2 vertices
```

The graph of a polyhedron with lines has no vertices:

```
sage: line = Polyhedron(lines=[[0,1]])
sage: line.vertex_graph()
Graph on 0 vertices
```

```
>>> from sage.all import *
>>> line = Polyhedron(lines=[[Integer(0),Integer(1)]])
>>> line.vertex_graph()
Graph on 0 vertices
```

2.6.6 Base class for polyhedra: Methods for constructing new polyhedra

Except for affine hull and affine hull projection.

class sage.geometry.polyhedron.base5.**Polyhedron_base5**(*parent*, *Vrep*, *Hrep*, *Vrep_minimal=None*, *Hrep_minimal=None*, *pref_rep=None*, *mutable=False*, ***kwds*)

Bases: *Polyhedron_base4*

Methods constructing new polyhedra except for affine hull and affine hull projection.

See *sage.geometry.polyhedron.base.Polyhedron_base*.

bipyramid()

Return a polyhedron that is a bipyramid over the original.

EXAMPLES:

```
sage: octahedron = polytopes.cross_polytope(3)
sage: cross_poly_4d = octahedron.bipyramid()
sage: cross_poly_4d.n_vertices()
8
sage: q = [list(v) for v in cross_poly_4d.vertex_generator()]; q
[[-1, 0, 0, 0],
 [0, -1, 0, 0],
 [0, 0, -1, 0],
 [0, 0, 0, -1],
 [0, 0, 0, 1],
 [0, 0, 1, 0],
 [0, 1, 0, 0],
 [1, 0, 0, 0]]
```

```
>>> from sage.all import *
>>> octahedron = polytopes.cross_polytope(Integer(3))
>>> cross_poly_4d = octahedron.bipyramid()
>>> cross_poly_4d.n_vertices()
8
>>> q = [list(v) for v in cross_poly_4d.vertex_generator()]; q
[[-1, 0, 0, 0],
 [0, -1, 0, 0],
 [0, 0, -1, 0],
 [0, 0, 0, -1],
 [0, 0, 0, 1],
 [0, 0, 1, 0],
 [0, 1, 0, 0],
 [1, 0, 0, 0]]
```

Now check that bipyramids of cross-polytopes are cross-polytopes:

```
sage: q2 = [list(v) for v in polytopes.cross_polytope(4).vertex_generator()]
sage: [v in q2 for v in q]
[True, True, True, True, True, True, True, True]
```

```
>>> from sage.all import *
>>> q2 = [list(v) for v in polytopes.cross_polytope(Integer(4)).vertex_
↪generator()]
>>> [v in q2 for v in q]
[True, True, True, True, True, True, True, True]
```

cartesian_product(*other*)

Return the Cartesian product.

INPUT:

- other – a *Polyhedron_base*

OUTPUT:

The Cartesian product of self and other with a suitable base ring to encompass the two.

EXAMPLES:

```
sage: P1 = Polyhedron([[0], [1]], base_ring=ZZ)
sage: P2 = Polyhedron([[0], [1]], base_ring=QQ)
sage: P1.product(P2)
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 4 vertices
```

```
>>> from sage.all import *
>>> P1 = Polyhedron([[Integer(0)], [Integer(1)]], base_ring=ZZ)
>>> P2 = Polyhedron([[Integer(0)], [Integer(1)]], base_ring=QQ)
>>> P1.product(P2)
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 4 vertices
```

The Cartesian product is the product in the semiring of polyhedra:

```
sage: P1 * P1
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices
```

(continues on next page)

(continued from previous page)

```
sage: P1 * P2
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 4 vertices
sage: P2 * P2
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 4 vertices
sage: 2 * P1
A 1-dimensional polyhedron in ZZ^1 defined as the convex hull of 2 vertices
sage: P1 * 2.0
A 1-dimensional polyhedron in RDF^1 defined as the convex hull of 2 vertices
```

```
>>> from sage.all import *
>>> P1 * P1
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices
>>> P1 * P2
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 4 vertices
>>> P2 * P2
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 4 vertices
>>> Integer(2) * P1
A 1-dimensional polyhedron in ZZ^1 defined as the convex hull of 2 vertices
>>> P1 * RealNumber('2.0')
A 1-dimensional polyhedron in RDF^1 defined as the convex hull of 2 vertices
```

An alias is *cartesian_product()*:

```
sage: P1.cartesian_product(P2) == P1.product(P2)
True
```

```
>>> from sage.all import *
>>> P1.cartesian_product(P2) == P1.product(P2)
True
```

convex_hull(*other*)

Return the convex hull of the set-theoretic union of the two polyhedra.

INPUT:

- other – a *Polyhedron*

OUTPUT: the convex hull

EXAMPLES:

```
sage: a_simplex = polytopes.simplex(3, project=True)
sage: verts = a_simplex.vertices()
sage: verts = [[x[0]*3/5 + x[1]*4/5, -x[0]*4/5 + x[1]*3/5, x[2]] for x in␣
↪verts]
sage: another_simplex = Polyhedron(vertices=verts)
sage: simplex_union = a_simplex.convex_hull(another_simplex)
sage: simplex_union.n_vertices()
7
```

```
>>> from sage.all import *
>>> a_simplex = polytopes.simplex(Integer(3), project=True)
>>> verts = a_simplex.vertices()
```

(continues on next page)

(continued from previous page)

```
>>> verts = [[x[Integer(0)]*Integer(3)/Integer(5) + x[Integer(1)]*Integer(4)/
↪Integer(5), -x[Integer(0)]*Integer(4)/Integer(5) + x[Integer(1)]*Integer(3)/
↪Integer(5), x[Integer(2)]] for x in verts]
>>> another_simplex = Polyhedron(vertices=verts)
>>> simplex_union = a_simplex.convex_hull(another_simplex)
>>> simplex_union.n_vertices()
7
```

deformation_cone()

Return the deformation cone of `self`.

Let P be a d-polytope in $\mathbf{R}^r$ with n facets. The deformation cone is a polyhedron in $\mathbf{R}^n$ whose points are the right-hand side b in $Ax \leq b$ where A is the matrix of facet normals of `self`, so that the resulting polytope has a normal fan which is a coarsening of the normal fan of `self`.

EXAMPLES:

Let's examine the deformation cone of the square with one truncated vertex:

```
sage: tc = Polyhedron([(1, -1), (1/3, 1), (1, 1/3), (-1, 1), (-1, -1)])
sage: dc = tc.deformation_cone()
sage: dc.an_element()
(2, 1, 1, 0, 0)
sage: [_.A() for _ in tc.Hrepresentation()]
[(1, 0), (0, 1), (0, -1), (-3, -3), (-1, 0)]
sage: P = Polyhedron(rays=[(1, 0, 2), (0, 1, 1), (0, -1, 1), (-3, -3, 0), (-1,
↪ 0, 0)])
sage: P.rays()
(A ray in the direction (-1, -1, 0),
 A ray in the direction (-1, 0, 0),
 A ray in the direction (0, -1, 1),
 A ray in the direction (0, 1, 1),
 A ray in the direction (1, 0, 2))
```

```
>>> from sage.all import *
>>> tc = Polyhedron([(Integer(1), -Integer(1)), (Integer(1)/Integer(3),␣
↪Integer(1)), (Integer(1), Integer(1)/Integer(3)), (-Integer(1), Integer(1)),
↪ (-Integer(1), -Integer(1))])
>>> dc = tc.deformation_cone()
>>> dc.an_element()
(2, 1, 1, 0, 0)
>>> [_.A() for _ in tc.Hrepresentation()]
[(1, 0), (0, 1), (0, -1), (-3, -3), (-1, 0)]
>>> P = Polyhedron(rays=[(Integer(1), Integer(0), Integer(2)), (Integer(0),␣
↪Integer(1), Integer(1)), (Integer(0), -Integer(1), Integer(1)), (-
↪Integer(3), -Integer(3), Integer(0)), (-Integer(1), Integer(0),␣
↪Integer(0))])
>>> P.rays()
(A ray in the direction (-1, -1, 0),
 A ray in the direction (-1, 0, 0),
 A ray in the direction (0, -1, 1),
 A ray in the direction (0, 1, 1),
 A ray in the direction (1, 0, 2))
```

Now, let's compute the deformation cone of the pyramid over a square and verify that it is not full dimensional:

```
sage: py = Polyhedron([(0, -1, -1), (0, -1, 1), (0, 1, -1), (0, 1, 1), (1, 0,␣
↪0)])
sage: dc_py = py.deformation_cone(); dc_py
A 4-dimensional polyhedron in QQ^5 defined as the convex hull of 1 vertex, 1␣
↪ray, 3 lines
sage: [ineq.b() for ineq in py.Hrepresentation()]
[0, 1, 1, 1, 1]
sage: r = dc_py.rays()[0]
sage: l1,l2,l3 = dc_py.lines()
sage: r.vector()-l1.vector()/2-l2.vector()-l3.vector()/2
(0, 1, 1, 1, 1)
```

```
>>> from sage.all import *
>>> py = Polyhedron([(Integer(0), -Integer(1), -Integer(1)), (Integer(0), -
↪Integer(1), Integer(1)), (Integer(0), Integer(1), -Integer(1)), (Integer(0),
↪ Integer(1), Integer(1)), (Integer(1), Integer(0), Integer(0))])
>>> dc_py = py.deformation_cone(); dc_py
A 4-dimensional polyhedron in QQ^5 defined as the convex hull of 1 vertex, 1␣
↪ray, 3 lines
>>> [ineq.b() for ineq in py.Hrepresentation()]
[0, 1, 1, 1, 1]
>>> r = dc_py.rays()[Integer(0)]
>>> l1,l2,l3 = dc_py.lines()
>>> r.vector()-l1.vector()/Integer(2)-l2.vector()-l3.vector()/Integer(2)
(0, 1, 1, 1, 1)
```

See also

`Kaehler_cone()`

REFERENCES:

For more information, see Section 5.4 of [DLRS2010] and Section 2.2 of [ACEP2020].

dilation(*scalar*)

Return the dilated (uniformly stretched) polyhedron.

INPUT:

- `scalar` – a scalar, not necessarily in `base_ring()`

OUTPUT:

The polyhedron dilated by that scalar, possibly coerced to a bigger base ring.

EXAMPLES:

```
sage: p = Polyhedron(vertices=[[t,t^2,t^3] for t in srange(2,6)])
sage: next(p.vertex_generator())
A vertex at (2, 4, 8)
sage: p2 = p.dilation(2)
sage: next(p2.vertex_generator())
A vertex at (4, 8, 16)
```

(continues on next page)

(continued from previous page)

```
sage: p.dilation(2) == p * 2
True
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices=[[t,t**Integer(2),t**Integer(3)] for t in␣
↪srange(Integer(2),Integer(6))])
>>> next(p.vertex_generator())
A vertex at (2, 4, 8)
>>> p2 = p.dilation(Integer(2))
>>> next(p2.vertex_generator())
A vertex at (4, 8, 16)
>>> p.dilation(Integer(2)) == p * Integer(2)
True
```

direct_sum(*other*)

Return the direct sum of `self` and `other`.

The direct sum of two polyhedron is the subdirect sum of the two, when they have the origin in their interior. To avoid checking if the origin is contained in both, we place the affine subspace containing `other` at the center of `self`.

INPUT:

- `other` – a *Polyhedron_base*

EXAMPLES:

```
sage: P1 = Polyhedron([[1], [2]], base_ring=ZZ)
sage: P2 = Polyhedron([[3], [4]], base_ring=QQ)
sage: ds = P1.direct_sum(P2);ds
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 4 vertices
sage: ds.vertices()
(A vertex at (1, 0),
 A vertex at (2, 0),
 A vertex at (3/2, -1/2),
 A vertex at (3/2, 1/2))
```

```
>>> from sage.all import *
>>> P1 = Polyhedron([[Integer(1)], [Integer(2)]], base_ring=ZZ)
>>> P2 = Polyhedron([[Integer(3)], [Integer(4)]], base_ring=QQ)
>>> ds = P1.direct_sum(P2);ds
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 4 vertices
>>> ds.vertices()
(A vertex at (1, 0),
 A vertex at (2, 0),
 A vertex at (3/2, -1/2),
 A vertex at (3/2, 1/2))
```

See also

join() subdirect_sum()

face_split(*face*)

Return the face splitting of the face `face`.

Splitting a face correspond to the bipyramid (see *bipyramid()*) of `self` where the two new vertices are placed above and below the center of `face` instead of the center of the whole polyhedron. The two new vertices are placed in the new dimension at height -1 and 1.

INPUT:

- `face` – a PolyhedronFace or a Vertex

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: pentagon  = polytopes.regular_polygon(5)
sage: f = pentagon.faces(1)[0]
sage: fsplit_pentagon = pentagon.face_split(f)
sage: fsplit_pentagon.f_vector()
(1, 7, 14, 9, 1)
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> pentagon  = polytopes.regular_polygon(Integer(5))
>>> f = pentagon.faces(Integer(1))[Integer(0)]
>>> fsplit_pentagon = pentagon.face_split(f)
>>> fsplit_pentagon.f_vector()
(1, 7, 14, 9, 1)
```

See also

one_point_suspension()

face_truncation(*face*, *linear_coefficients=None*, *cut_frac=None*)

Return a new polyhedron formed by truncating a face by an hyperplane.

By default, the normal vector of the hyperplane used to truncate the polyhedron is obtained by taking the barycenter vector of the cone corresponding to the truncated face in the normal fan of the polyhedron. It is possible to change the direction using the option `linear_coefficients`.

To determine how deep the truncation is done, the method uses the parameter `cut_frac`. By default it is equal to $\frac{1}{3}$. Once the normal vector of the cutting hyperplane is chosen, the vertices of polyhedron are evaluated according to the corresponding linear function. The parameter $\frac{1}{3}$ means that the cutting hyperplane is placed $\frac{1}{3}$ of the way from the vertices of the truncated face to the next evaluated vertex.

INPUT:

- `face` – a *PolyhedronFace*
- `linear_coefficients` – tuple of integer. Specifies the coefficient of the normal vector of the cutting hyperplane used to truncate the face. The default direction is determined using the normal fan of the polyhedron.
- `cut_frac` – number between 0 and 1. Determines where the hyperplane cuts the polyhedron. A value close to 0 cuts very close to the face, whereas a value close to 1 cuts very close to the next vertex (according to the normal vector of the cutting hyperplane). Default is $\frac{1}{3}$.

OUTPUT: a Polyhedron object, truncated as described above

EXAMPLES:

```
sage: Cube = polytopes.hypercube(3)
sage: vertex_trunc1 = Cube.face_truncation(Cube.faces(0)[0])
sage: vertex_trunc1.f_vector()
(1, 10, 15, 7, 1)
sage: tuple(f.ambient_V_indices() for f in vertex_trunc1.faces(2))
((4, 5, 6, 7, 9),
 (0, 3, 4, 8, 9),
 (0, 1, 6, 7, 8),
 (7, 8, 9),
 (2, 3, 4, 5),
 (1, 2, 5, 6),
 (0, 1, 2, 3))
sage: vertex_trunc1.vertices()
(A vertex at (1, -1, -1),
 A vertex at (1, 1, -1),
 A vertex at (1, 1, 1),
 A vertex at (1, -1, 1),
 A vertex at (-1, -1, 1),
 A vertex at (-1, 1, 1),
 A vertex at (-1, 1, -1),
 A vertex at (-1, -1/3, -1),
 A vertex at (-1/3, -1, -1),
 A vertex at (-1, -1, -1/3))
sage: vertex_trunc2 = Cube.face_truncation(Cube.faces(0)[0], cut_frac=1/2)
sage: vertex_trunc2.f_vector()
(1, 10, 15, 7, 1)
sage: tuple(f.ambient_V_indices() for f in vertex_trunc2.faces(2))
((4, 5, 6, 7, 9),
 (0, 3, 4, 8, 9),
 (0, 1, 6, 7, 8),
 (7, 8, 9),
 (2, 3, 4, 5),
 (1, 2, 5, 6),
 (0, 1, 2, 3))
sage: vertex_trunc2.vertices()
(A vertex at (1, -1, -1),
 A vertex at (1, 1, -1),
 A vertex at (1, 1, 1),
 A vertex at (1, -1, 1),
 A vertex at (-1, -1, 1),
 A vertex at (-1, 1, 1),
 A vertex at (-1, 1, -1),
 A vertex at (-1, 0, -1),
 A vertex at (0, -1, -1),
 A vertex at (-1, -1, 0))
sage: vertex_trunc3 = Cube.face_truncation(Cube.faces(0)[0], cut_frac=0.3)
sage: vertex_trunc3.vertices()
(A vertex at (-1.0, -1.0, 1.0),
 A vertex at (-1.0, 1.0, -1.0),
 A vertex at (-1.0, 1.0, 1.0),
 A vertex at (1.0, 1.0, -1.0),
```

(continues on next page)

(continued from previous page)

```
 A vertex at (1.0, 1.0, 1.0),
 A vertex at (1.0, -1.0, 1.0),
 A vertex at (1.0, -1.0, -1.0),
 A vertex at (-0.4, -1.0, -1.0),
 A vertex at (-1.0, -0.4, -1.0),
 A vertex at (-1.0, -1.0, -0.4))
sage: edge_trunc = Cube.face_truncation(Cube.faces(1)[11])
sage: edge_trunc.f_vector()
(1, 10, 15, 7, 1)
sage: tuple(f.ambient_V_indices() for f in edge_trunc.faces(2))
((0, 5, 6, 7),
 (1, 4, 5, 6, 8),
 (6, 7, 8, 9),
 (0, 2, 3, 7, 9),
 (1, 2, 8, 9),
 (0, 3, 4, 5),
 (1, 2, 3, 4))
 sage: face_trunc = Cube.face_truncation(Cube.faces(2)[2])
 sage: face_trunc.vertices()
 (A vertex at (1, -1, -1),
  A vertex at (1, 1, -1),
  A vertex at (1, 1, 1),
  A vertex at (1, -1, 1),
  A vertex at (-1/3, -1, 1),
  A vertex at (-1/3, 1, 1),
  A vertex at (-1/3, 1, -1),
  A vertex at (-1/3, -1, -1))
 sage: face_trunc.face_lattice().is_isomorphic(Cube.face_lattice())          #␣
↪needs sage.combinat sage.graphs
 True
```

```
>>> from sage.all import *
>>> Cube = polytopes.hypercube(Integer(3))
>>> vertex_trunc1 = Cube.face_truncation(Cube.faces(Integer(0))[Integer(0)])
>>> vertex_trunc1.f_vector()
(1, 10, 15, 7, 1)
>>> tuple(f.ambient_V_indices() for f in vertex_trunc1.faces(Integer(2)))
((4, 5, 6, 7, 9),
 (0, 3, 4, 8, 9),
 (0, 1, 6, 7, 8),
 (7, 8, 9),
 (2, 3, 4, 5),
 (1, 2, 5, 6),
 (0, 1, 2, 3))
>>> vertex_trunc1.vertices()
(A vertex at (1, -1, -1),
 A vertex at (1, 1, -1),
 A vertex at (1, 1, 1),
 A vertex at (1, -1, 1),
 A vertex at (-1, -1, 1),
 A vertex at (-1, 1, 1),
 A vertex at (-1, 1, -1),
```

(continues on next page)

(continued from previous page)

```
 A vertex at (-1, -1/3, -1),
 A vertex at (-1/3, -1, -1),
 A vertex at (-1, -1, -1/3))
>>> vertex_trunc2 = Cube.face_truncation(Cube.faces(Integer(0))[Integer(0)],␣
↪cut_frac=Integer(1)/Integer(2))
>>> vertex_trunc2.f_vector()
(1, 10, 15, 7, 1)
>>> tuple(f.ambient_V_indices() for f in vertex_trunc2.faces(Integer(2)))
((4, 5, 6, 7, 9),
 (0, 3, 4, 8, 9),
 (0, 1, 6, 7, 8),
 (7, 8, 9),
 (2, 3, 4, 5),
 (1, 2, 5, 6),
 (0, 1, 2, 3))
>>> vertex_trunc2.vertices()
(A vertex at (1, -1, -1),
 A vertex at (1, 1, -1),
 A vertex at (1, 1, 1),
 A vertex at (1, -1, 1),
 A vertex at (-1, -1, 1),
 A vertex at (-1, 1, 1),
 A vertex at (-1, 1, -1),
 A vertex at (-1, 0, -1),
 A vertex at (0, -1, -1),
 A vertex at (-1, -1, 0))
>>> vertex_trunc3 = Cube.face_truncation(Cube.faces(Integer(0))[Integer(0)],␣
↪cut_frac=RealNumber('0.3'))
>>> vertex_trunc3.vertices()
(A vertex at (-1.0, -1.0, 1.0),
 A vertex at (-1.0, 1.0, -1.0),
 A vertex at (-1.0, 1.0, 1.0),
 A vertex at (1.0, 1.0, -1.0),
 A vertex at (1.0, 1.0, 1.0),
 A vertex at (1.0, -1.0, 1.0),
 A vertex at (1.0, -1.0, -1.0),
 A vertex at (-0.4, -1.0, -1.0),
 A vertex at (-1.0, -0.4, -1.0),
 A vertex at (-1.0, -1.0, -0.4))
>>> edge_trunc = Cube.face_truncation(Cube.faces(Integer(1))[Integer(11)])
>>> edge_trunc.f_vector()
(1, 10, 15, 7, 1)
>>> tuple(f.ambient_V_indices() for f in edge_trunc.faces(Integer(2)))
((0, 5, 6, 7),
 (1, 4, 5, 6, 8),
 (6, 7, 8, 9),
 (0, 2, 3, 7, 9),
 (1, 2, 8, 9),
 (0, 3, 4, 5),
 (1, 2, 3, 4))
>>> face_trunc = Cube.face_truncation(Cube.faces(Integer(2))[Integer(2)])
>>> face_trunc.vertices()
```

(continues on next page)

(continued from previous page)

```
(A vertex at (1, -1, -1),
 A vertex at (1, 1, -1),
 A vertex at (1, 1, 1),
 A vertex at (1, -1, 1),
 A vertex at (-1/3, -1, 1),
 A vertex at (-1/3, 1, 1),
 A vertex at (-1/3, 1, -1),
 A vertex at (-1/3, -1, -1))
>>> face_trunc.face_lattice().is_isomorphic(Cube.face_lattice())          #␣
↪needs sage.combinat sage.graphs
True
```

intersection(*other*)

Return the intersection of one polyhedron with another.

INPUT:

- other – a *Polyhedron*

OUTPUT: the intersection

Note that the intersection of two **Z**-polyhedra might not be a **Z**-polyhedron. In this case, a **Q**-polyhedron is returned.

EXAMPLES:

```
sage: cube = polytopes.hypercube(3)
sage: oct = polytopes.cross_polytope(3)
sage: cube.intersection(oct*2)
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 12 vertices
```

```
>>> from sage.all import *
>>> cube = polytopes.hypercube(Integer(3))
>>> oct = polytopes.cross_polytope(Integer(3))
>>> cube.intersection(oct*Integer(2))
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 12 vertices
```

As a shorthand, one may use:

```
sage: cube & oct*2
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 12 vertices
```

```
>>> from sage.all import *
>>> cube & oct*Integer(2)
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 12 vertices
```

The intersection of two **Z**-polyhedra is not necessarily a **Z**-polyhedron:

```
sage: P = Polyhedron([(0,0),(1,1)], base_ring=ZZ)
sage: P.intersection(P)
A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: Q = Polyhedron([(0,1),(1,0)], base_ring=ZZ)
sage: P.intersection(Q)
A 0-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex
```

(continues on next page)

(continued from previous page)

```
sage: _.Vrepresentation()
(A vertex at (1/2, 1/2),)
```

```
>>> from sage.all import *
>>> P = Polyhedron([(Integer(0),Integer(0)),(Integer(1),Integer(1))], base_
↪ring=ZZ)
>>> P.intersection(P)
A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
>>> Q = Polyhedron([(Integer(0),Integer(1)),(Integer(1),Integer(0))], base_
↪ring=ZZ)
>>> P.intersection(Q)
A 0-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex
>>> _.Vrepresentation()
(A vertex at (1/2, 1/2),)
```

join(*other*)

Return the join of `self` and `other`.

The join of two polyhedra is obtained by first placing the two objects in two non-intersecting affine subspaces V, and W whose affine hull is the whole ambient space, and finally by taking the convex hull of their union. The dimension of the join is the sum of the dimensions of the two polyhedron plus 1.

INPUT:

- `other` – a polyhedron

EXAMPLES:

```
sage: P1 = Polyhedron([[0],[1]], base_ring=ZZ)
sage: P2 = Polyhedron([[0],[1]], base_ring=QQ)
sage: P1.join(P2)
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 4 vertices
sage: P1.join(P1)
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 4 vertices
sage: P2.join(P2)
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 4 vertices
```

```
>>> from sage.all import *
>>> P1 = Polyhedron([[Integer(0)],[Integer(1)]], base_ring=ZZ)
>>> P2 = Polyhedron([[Integer(0)],[Integer(1)]], base_ring=QQ)
>>> P1.join(P2)
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 4 vertices
>>> P1.join(P1)
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 4 vertices
>>> P2.join(P2)
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 4 vertices
```

An unbounded example:

```
sage: R1 = Polyhedron(rays=[[1]])
sage: R1.join(R1)
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 2 vertices␣
↪and 2 rays
```

```
>>> from sage.all import *
>>> R1 = Polyhedron(rays=[[Integer(1)]])
>>> R1.join(R1)
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 2 vertices␣
↪and 2 rays
```

lawrence_extension(*v*)

Return the Lawrence extension of `self` on the point `v`.

Let P be a polytope and v be a vertex of P or a point outside P. The Lawrence extension of P on v is the convex hull of $(v, 1)$, $(v, 2)$ and $(u, 0)$ for all vertices u in P other than v if v is a vertex.

INPUT:

- `v` – a vertex of `self` or a point outside it

EXAMPLES:

```
sage: P = polytopes.cube()
sage: P.lawrence_extension(P.vertices()[0])
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 9 vertices
sage: P.lawrence_extension([-1,-1,-1])
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 9 vertices
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> P.lawrence_extension(P.vertices()[Integer(0)])
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 9 vertices
>>> P.lawrence_extension([-Integer(1),-Integer(1),-Integer(1)])
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 9 vertices
```

REFERENCES:

For more information, see Section 6.6 of [Zie2007].

lawrence_polytope()

Return the Lawrence polytope of `self`.

Let P be a d-polytope in $\mathbf{R}^r$ with n vertices. The Lawrence polytope of P is the polytope whose vertices are the columns of the following $(r + n)$-by-$2n$ matrix.

$$\begin{pmatrix} V & V \\ I_n & 2I_n \end{pmatrix},$$

where V is the r-by-n vertices matrix of P.

EXAMPLES:

```
sage: P = polytopes.octahedron()
sage: L = P.lawrence_polytope(); L
A 9-dimensional polyhedron in ZZ^9 defined as the convex hull of 12 vertices
sage: V = P.vertices_list()
sage: for i, v in enumerate(V):
....:     v = v + i*[0]
....:     P = P.lawrence_extension(v)
sage: P == L
True
```

```
>>> from sage.all import *
>>> P = polytopes.octahedron()
>>> L = P.lawrence_polytope(); L
A 9-dimensional polyhedron in ZZ^9 defined as the convex hull of 12 vertices
>>> V = P.vertices_list()
>>> for i, v in enumerate(V):
...     v = v + i*[Integer(0)]
...     P = P.lawrence_extension(v)
>>> P == L
True
```

REFERENCES:

For more information, see Section 6.6 of [Zie2007].

linear_transformation(*linear_transf*, *new_base_ring=None*)

Return the linear transformation of `self`.

INPUT:

- `linear_transf` – a matrix, not necessarily in `base_ring()`
- `new_base_ring` – ring (optional); specify the new base ring; may avoid coercion failure

OUTPUT:

The polyhedron transformed by that matrix, possibly coerced to a bigger base ring.

EXAMPLES:

```
sage: b3 = polytopes.Birkhoff_polytope(3)
sage: proj_mat = matrix([[0,1,0,0,0,0,0,0,0], [0,0,0,1,0,0,0,0,0],
....:                    [0,0,0,0,0,1,0,0,0], [0,0,0,0,0,0,0,1,0]])
sage: b3_proj = proj_mat * b3; b3_proj
A 3-dimensional polyhedron in ZZ^4 defined as the convex hull of 5 vertices

sage: # needs sage.rings.number_field
sage: square = polytopes.regular_polygon(4)
sage: square.vertices_list()
[[0, -1], [1, 0], [-1, 0], [0, 1]]
sage: transf = matrix([[1,1], [0,1]])
sage: sheared = transf * square
sage: sheared.vertices_list()
[[-1, -1], [1, 0], [-1, 0], [1, 1]]
sage: sheared == square.linear_transformation(transf)
True
```

```
>>> from sage.all import *
>>> b3 = polytopes.Birkhoff_polytope(Integer(3))
>>> proj_mat = matrix([[Integer(0),Integer(1),Integer(0),Integer(0),
↪Integer(0),Integer(0),Integer(0),Integer(0),Integer(0)], [Integer(0),
↪Integer(0),Integer(0),Integer(1),Integer(0),Integer(0),Integer(0),
↪Integer(0),Integer(0)],
...                    [Integer(0),Integer(0),Integer(0),Integer(0),
↪Integer(0),Integer(1),Integer(0),Integer(0),Integer(0)], [Integer(0),
↪Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),Integer(0),
```

(continues on next page)

(continued from previous page)

```
↪Integer(1),Integer(0)]])
>>> b3_proj = proj_mat * b3; b3_proj
A 3-dimensional polyhedron in ZZ^4 defined as the convex hull of 5 vertices

>>> # needs sage.rings.number_field
>>> square = polytopes.regular_polygon(Integer(4))
>>> square.vertices_list()
[[0, -1], [1, 0], [-1, 0], [0, 1]]
>>> transf = matrix([[Integer(1),Integer(1)], [Integer(0),Integer(1)]])
>>> sheared = transf * square
>>> sheared.vertices_list()
[[-1, -1], [1, 0], [-1, 0], [1, 1]]
>>> sheared == square.linear_transformation(transf)
True
```

Specifying the new base ring may avoid coercion failure:

```
sage: # needs sage.rings.number_field
sage: K.<sqrt2> = QuadraticField(2)
sage: L.<sqrt3> = QuadraticField(3)
sage: P = polytopes.cube()*sqrt2
sage: M = matrix([[sqrt3, 0, 0], [0, sqrt3, 0], [0, 0, 1]])
sage: P.linear_transformation(M, new_base_ring=K.composite_fields(L)[0])
A 3-dimensional polyhedron in
 (Number Field in sqrt2sqrt3 with defining polynomial x^4 - 10*x^2 + 1
  with sqrt2sqrt3 = 0.3178372451957823?)^3
 defined as the convex hull of 8 vertices
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> K = QuadraticField(Integer(2), names=('sqrt2',)); (sqrt2,) = K._first_
↪ngens(1)
>>> L = QuadraticField(Integer(3), names=('sqrt3',)); (sqrt3,) = L._first_
↪ngens(1)
>>> P = polytopes.cube()*sqrt2
>>> M = matrix([[sqrt3, Integer(0), Integer(0)], [Integer(0), sqrt3,␣
↪Integer(0)], [Integer(0), Integer(0), Integer(1)]])
>>> P.linear_transformation(M, new_base_ring=K.composite_
↪fields(L)[Integer(0)])
A 3-dimensional polyhedron in
 (Number Field in sqrt2sqrt3 with defining polynomial x^4 - 10*x^2 + 1
  with sqrt2sqrt3 = 0.3178372451957823?)^3
 defined as the convex hull of 8 vertices
```

Linear transformation without specified new base ring fails in this case:

```
sage: M*P                                                               #␣
↪needs sage.rings.number_field
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for *:
'Full MatrixSpace of 3 by 3 dense matrices over Number Field in sqrt3
```

(continues on next page)

(continued from previous page)

```
with defining polynomial x^2 - 3 with sqrt3 = 1.732050807568878?' and
'Full MatrixSpace of 3 by 8 dense matrices over Number Field in sqrt2
with defining polynomial x^2 - 2 with sqrt2 = 1.414213562373095?'
```

```
>>> from sage.all import *
>>> M*P                                                                  #␣
↪needs sage.rings.number_field
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for *:
'Full MatrixSpace of 3 by 3 dense matrices over Number Field in sqrt3
with defining polynomial x^2 - 3 with sqrt3 = 1.732050807568878?' and
'Full MatrixSpace of 3 by 8 dense matrices over Number Field in sqrt2
with defining polynomial x^2 - 2 with sqrt2 = 1.414213562373095?'
```

minkowski_difference(*other*)

Return the Minkowski difference.

Minkowski subtraction can equivalently be defined via Minkowski addition (see *minkowski_sum()*) or as set-theoretic intersection via

$$X \ominus Y = (X^c \oplus Y)^c = \bigcap_{y \in Y} (X - y)$$

where superscript-"c" means the complement in the ambient vector space. The Minkowski difference of convex sets is convex, and the difference of polyhedra is again a polyhedron. We only consider the case of polyhedra in the following. Note that it is not quite the inverse of addition. In fact:

- $(X + Y) - Y = X$ for any polyhedra X, Y.
- $(X - Y) + Y \subseteq X$
- $(X - Y) + Y = X$ if and only if Y is a Minkowski summand of X.

INPUT:

- `other` – a *Polyhedron_base*

OUTPUT:

The Minkowski difference of `self` and `other`. Also known as Minkowski subtraction of `other` from `self`.

EXAMPLES:

```
sage: X = polytopes.hypercube(3)
sage: Y = Polyhedron(vertices=[(0,0,0), (0,0,1), (0,1,0), (1,0,0)]) / 2
sage: (X+Y)-Y == X
True
sage: (X-Y)+Y < X
True
```

```
>>> from sage.all import *
>>> X = polytopes.hypercube(Integer(3))
>>> Y = Polyhedron(vertices=[(Integer(0),Integer(0),Integer(0)), (Integer(0),
↪Integer(0),Integer(1)), (Integer(0),Integer(1),Integer(0)), (Integer(1),
↪Integer(0),Integer(0))]) / Integer(2)
>>> (X+Y)-Y == X
```

(continues on next page)

(continued from previous page)

```
True
>>> (X-Y)+Y < X
True
```

The polyhedra need not be full-dimensional:

```
sage: X2 = Polyhedron(vertices=[(-1,-1,0), (1,-1,0), (-1,1,0), (1,1,0)])
sage: Y2 = Polyhedron(vertices=[(0,0,0), (0,1,0), (1,0,0)]) / 2
sage: (X2+Y2)-Y2 == X2
True
sage: (X2-Y2)+Y2 < X2
True
```

```
>>> from sage.all import *
>>> X2 = Polyhedron(vertices=[(-Integer(1),-Integer(1),Integer(0)),␣
↪(Integer(1),-Integer(1),Integer(0)), (-Integer(1),Integer(1),Integer(0)),␣
↪(Integer(1),Integer(1),Integer(0))])
>>> Y2 = Polyhedron(vertices=[(Integer(0),Integer(0),Integer(0)), (Integer(0),
↪Integer(1),Integer(0)), (Integer(1),Integer(0),Integer(0))]) / Integer(2)
>>> (X2+Y2)-Y2 == X2
True
>>> (X2-Y2)+Y2 < X2
True
```

Minus sign is really an alias for `minkowski_difference()`

```
sage: four_cube = polytopes.hypercube(4)
sage: four_simplex = Polyhedron(vertices=[[0, 0, 0, 1], [0, 0, 1, 0],
....:                                     [0, 1, 0, 0], [1, 0, 0, 0]])
sage: four_cube - four_simplex
A 4-dimensional polyhedron in QQ^4 defined as the convex hull of 16 vertices
sage: four_cube.minkowski_difference(four_simplex) == four_cube - four_simplex
True
```

```
>>> from sage.all import *
>>> four_cube = polytopes.hypercube(Integer(4))
>>> four_simplex = Polyhedron(vertices=[[Integer(0), Integer(0), Integer(0),␣
↪Integer(1)], [Integer(0), Integer(0), Integer(1), Integer(0)],
...                                     [Integer(0), Integer(1), Integer(0),␣
↪Integer(0)], [Integer(1), Integer(0), Integer(0), Integer(0)]])
>>> four_cube - four_simplex
A 4-dimensional polyhedron in QQ^4 defined as the convex hull of 16 vertices
>>> four_cube.minkowski_difference(four_simplex) == four_cube - four_simplex
True
```

Coercion of the base ring works:

```
sage: poly_spam = Polyhedron([[3,4,5,2], [1,0,0,1], [0,0,0,0],
....:                         [0,4,3,2], [-3,-3,-3,-3]], base_ring=ZZ)
sage: poly_eggs = Polyhedron([[5,4,5,4], [-4,5,-4,5],
....:                         [4,-5,4,-5], [0,0,0,0]], base_ring=QQ) / 100
sage: poly_spam - poly_eggs
```

(continues on next page)

(continued from previous page)

```
A 4-dimensional polyhedron in QQ^4 defined as the convex hull of 5 vertices
```

```
>>> from sage.all import *
>>> poly_spam = Polyhedron([[Integer(3),Integer(4),Integer(5),Integer(2)],␣
↪[Integer(1),Integer(0),Integer(0),Integer(1)], [Integer(0),Integer(0),
↪Integer(0),Integer(0)],
...                             [Integer(0),Integer(4),Integer(3),Integer(2)], [-
↪Integer(3),-Integer(3),-Integer(3),-Integer(3)]], base_ring=ZZ)
>>> poly_eggs = Polyhedron([[Integer(5),Integer(4),Integer(5),Integer(4)], [-
↪Integer(4),Integer(5),-Integer(4),Integer(5)],
...                             [Integer(4),-Integer(5),Integer(4),-Integer(5)],␣
↪[Integer(0),Integer(0),Integer(0),Integer(0)]], base_ring=QQ) / Integer(100)
>>> poly_spam - poly_eggs
A 4-dimensional polyhedron in QQ^4 defined as the convex hull of 5 vertices
```

minkowski_sum(*other*)

Return the Minkowski sum.

Minkowski addition of two subsets of a vector space is defined as

$$X \oplus Y = \bigcup_{y\in Y}(X+y) = \bigcup_{x\in X, y\in Y}(x+y)$$

See *minkowski_difference()* for a partial inverse operation.

INPUT:

- other – a *Polyhedron_base*

OUTPUT: the Minkowski sum of self and other

EXAMPLES:

```
sage: X = polytopes.hypercube(3)
sage: Y = Polyhedron(vertices=[(0,0,0), (0,0,1/2), (0,1/2,0), (1/2,0,0)])
sage: X+Y
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 13 vertices

sage: four_cube = polytopes.hypercube(4)
sage: four_simplex = Polyhedron(vertices=[[0, 0, 0, 1], [0, 0, 1, 0],
....:                                       [0, 1, 0, 0], [1, 0, 0, 0]])
sage: four_cube + four_simplex
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 36 vertices
sage: four_cube.minkowski_sum(four_simplex) == four_cube + four_simplex
True

sage: poly_spam = Polyhedron([[3,4,5,2], [1,0,0,1], [0,0,0,0],
....:                         [0,4,3,2], [-3,-3,-3,-3]], base_ring=ZZ)
sage: poly_eggs = Polyhedron([[5,4,5,4], [-4,5,-4,5],
....:                         [4,-5,4,-5], [0,0,0,0]], base_ring=QQ)
sage: poly_spam + poly_spam + poly_eggs
A 4-dimensional polyhedron in QQ^4 defined as the convex hull of 12 vertices
```

```
>>> from sage.all import *
>>> X = polytopes.hypercube(Integer(3))
>>> Y = Polyhedron(vertices=[(Integer(0),Integer(0),Integer(0)), (Integer(0),
↪Integer(0),Integer(1)/Integer(2)), (Integer(0),Integer(1)/Integer(2),
↪Integer(0)), (Integer(1)/Integer(2),Integer(0),Integer(0))])
>>> X+Y
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 13 vertices

>>> four_cube = polytopes.hypercube(Integer(4))
>>> four_simplex = Polyhedron(vertices=[[Integer(0), Integer(0), Integer(0),␣
↪Integer(1)], [Integer(0), Integer(0), Integer(1), Integer(0)],
...                                     [Integer(0), Integer(1), Integer(0),␣
↪Integer(0)], [Integer(1), Integer(0), Integer(0), Integer(0)]])
>>> four_cube + four_simplex
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 36 vertices
>>> four_cube.minkowski_sum(four_simplex) == four_cube + four_simplex
True

>>> poly_spam = Polyhedron([[Integer(3),Integer(4),Integer(5),Integer(2)],␣
↪[Integer(1),Integer(0),Integer(0),Integer(1)], [Integer(0),Integer(0),
↪Integer(0),Integer(0)],
...                         [Integer(0),Integer(4),Integer(3),Integer(2)], [-
↪Integer(3),-Integer(3),-Integer(3),-Integer(3)]], base_ring=ZZ)
>>> poly_eggs = Polyhedron([[Integer(5),Integer(4),Integer(5),Integer(4)], [-
↪Integer(4),Integer(5),-Integer(4),Integer(5)],
...                         [Integer(4),-Integer(5),Integer(4),-Integer(5)],␣
↪[Integer(0),Integer(0),Integer(0),Integer(0)]], base_ring=QQ)
>>> poly_spam + poly_spam + poly_eggs
A 4-dimensional polyhedron in QQ^4 defined as the convex hull of 12 vertices
```

one_point_suspension(*vertex*)

Return the one-point suspension of `self` by splitting the vertex `vertex`.

The resulting polyhedron has one more vertex and its dimension increases by one.

INPUT:

- `vertex` – a Vertex of `self`

EXAMPLES:

```
sage: cube = polytopes.cube()
sage: v = cube.vertices()[0]
sage: ops_cube = cube.one_point_suspension(v)
sage: ops_cube.f_vector()
(1, 9, 24, 24, 9, 1)

sage: # needs sage.rings.number_field
sage: pentagon  = polytopes.regular_polygon(5)
sage: v = pentagon.vertices()[0]
sage: ops_pentagon = pentagon.one_point_suspension(v)
sage: ops_pentagon.f_vector()
(1, 6, 12, 8, 1)
```

```
>>> from sage.all import *
>>> cube = polytopes.cube()
>>> v = cube.vertices()[Integer(0)]
>>> ops_cube = cube.one_point_suspension(v)
>>> ops_cube.f_vector()
(1, 9, 24, 24, 9, 1)

>>> # needs sage.rings.number_field
>>> pentagon  = polytopes.regular_polygon(Integer(5))
>>> v = pentagon.vertices()[Integer(0)]
>>> ops_pentagon = pentagon.one_point_suspension(v)
>>> ops_pentagon.f_vector()
(1, 6, 12, 8, 1)
```

It works with a polyhedral face as well:

```
sage: vv = cube.faces(0)[1]
sage: ops_cube2 = cube.one_point_suspension(vv)
sage: ops_cube == ops_cube2
True
```

```
>>> from sage.all import *
>>> vv = cube.faces(Integer(0))[Integer(1)]
>>> ops_cube2 = cube.one_point_suspension(vv)
>>> ops_cube == ops_cube2
True
```

See also:

face_split()

polar(*in_affine_span=False*)

Return the polar (dual) polytope.

The original vertices are translated so that their barycenter is at the origin, and then the vertices are used as the coefficients in the polar inequalities.

The polytope must be full-dimensional, unless `in_affine_span` is `True`. If `in_affine_span` is `True`, then the operation will be performed in the linear/affine span of the polyhedron (after translation).

EXAMPLES:

```
sage: p = Polyhedron(vertices=[[0,0,1], [0,1,0], [1,0,0], [0,0,0], [1,1,1]],
....:                base_ring=QQ); p
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 5 vertices
sage: p.polar()
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 6 vertices

sage: cube = polytopes.hypercube(3)
sage: octahedron = polytopes.cross_polytope(3)
sage: cube_dual = cube.polar()
sage: octahedron == cube_dual
True
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices=[[Integer(0),Integer(0),Integer(1)], [Integer(0),
↪Integer(1),Integer(0)], [Integer(1),Integer(0),Integer(0)], [Integer(0),
↪Integer(0),Integer(0)], [Integer(1),Integer(1),Integer(1)]],
...                    base_ring=QQ); p
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 5 vertices
>>> p.polar()
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 6 vertices

>>> cube = polytopes.hypercube(Integer(3))
>>> octahedron = polytopes.cross_polytope(Integer(3))
>>> cube_dual = cube.polar()
>>> octahedron == cube_dual
True
```

`in_affine_span` somewhat ignores equations, performing the polar in the spanned subspace (after translating barycenter to origin):

```
sage: P = polytopes.simplex(3, base_ring=QQ)
sage: P.polar(in_affine_span=True)
A 3-dimensional polyhedron in QQ^4 defined as the convex hull of 4 vertices
```

```
>>> from sage.all import *
>>> P = polytopes.simplex(Integer(3), base_ring=QQ)
>>> P.polar(in_affine_span=True)
A 3-dimensional polyhedron in QQ^4 defined as the convex hull of 4 vertices
```

Embedding the polytope in a higher dimension, commutes with polar in this case:

```
sage: point = Polyhedron([[0]])
sage: P = polytopes.cube().change_ring(QQ)
sage: (P*point).polar(in_affine_span=True) == P.polar()*point
True
```

```
>>> from sage.all import *
>>> point = Polyhedron([[Integer(0)]])
>>> P = polytopes.cube().change_ring(QQ)
>>> (P*point).polar(in_affine_span=True) == P.polar()*point
True
```

prism()

Return a prism of the original polyhedron.

EXAMPLES:

```
sage: square = polytopes.hypercube(2)
sage: cube = square.prism()
sage: cube
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 8 vertices
sage: hypercube = cube.prism()
sage: hypercube.n_vertices()
16
```

```
>>> from sage.all import *
>>> square = polytopes.hypercube(Integer(2))
>>> cube = square.prism()
>>> cube
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 8 vertices
>>> hypercube = cube.prism()
>>> hypercube.n_vertices()
16
```

product(*other*)

Return the Cartesian product.

INPUT:

- other – a *Polyhedron_base*

OUTPUT:

The Cartesian product of self and other with a suitable base ring to encompass the two.

EXAMPLES:

```
sage: P1 = Polyhedron([[0], [1]], base_ring=ZZ)
sage: P2 = Polyhedron([[0], [1]], base_ring=QQ)
sage: P1.product(P2)
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 4 vertices
```

```
>>> from sage.all import *
>>> P1 = Polyhedron([[Integer(0)], [Integer(1)]], base_ring=ZZ)
>>> P2 = Polyhedron([[Integer(0)], [Integer(1)]], base_ring=QQ)
>>> P1.product(P2)
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 4 vertices
```

The Cartesian product is the product in the semiring of polyhedra:

```
sage: P1 * P1
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices
sage: P1 * P2
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 4 vertices
sage: P2 * P2
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 4 vertices
sage: 2 * P1
A 1-dimensional polyhedron in ZZ^1 defined as the convex hull of 2 vertices
sage: P1 * 2.0
A 1-dimensional polyhedron in RDF^1 defined as the convex hull of 2 vertices
```

```
>>> from sage.all import *
>>> P1 * P1
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices
>>> P1 * P2
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 4 vertices
>>> P2 * P2
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 4 vertices
>>> Integer(2) * P1
A 1-dimensional polyhedron in ZZ^1 defined as the convex hull of 2 vertices
```

(continues on next page)

(continued from previous page)

```
>>> P1 * RealNumber('2.0')
A 1-dimensional polyhedron in RDF^1 defined as the convex hull of 2 vertices
```

An alias is `cartesian_product()`:

```
sage: P1.cartesian_product(P2) == P1.product(P2)
True
```

```
>>> from sage.all import *
>>> P1.cartesian_product(P2) == P1.product(P2)
True
```

pyramid()

Return a polyhedron that is a pyramid over the original.

EXAMPLES:

```
sage: square = polytopes.hypercube(2);  square
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices
sage: egyptian_pyramid = square.pyramid();  egyptian_pyramid
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 5 vertices
sage: egyptian_pyramid.n_vertices()
5
sage: for v in egyptian_pyramid.vertex_generator(): print(v)
A vertex at (0, -1, -1)
A vertex at (0, -1, 1)
A vertex at (0, 1, -1)
A vertex at (0, 1, 1)
A vertex at (1, 0, 0)
```

```
>>> from sage.all import *
>>> square = polytopes.hypercube(Integer(2));  square
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices
>>> egyptian_pyramid = square.pyramid();  egyptian_pyramid
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 5 vertices
>>> egyptian_pyramid.n_vertices()
5
>>> for v in egyptian_pyramid.vertex_generator(): print(v)
A vertex at (0, -1, -1)
A vertex at (0, -1, 1)
A vertex at (0, 1, -1)
A vertex at (0, 1, 1)
A vertex at (1, 0, 0)
```

stack(*face*, *position=None*)

Return a new polyhedron formed by stacking onto a `face`. Stacking a face adds a new vertex located slightly outside of the designated face.

INPUT:

- `face` – a PolyhedronFace
- `position` – a positive number. Determines a relative distance from the barycenter of `face`. A value close to 0 will place the new vertex close to the face and a large value further away. Default is 1. If the

given value is too large, an error is returned.

OUTPUT: a Polyhedron object

EXAMPLES:

```
sage: cube = polytopes.cube()
sage: square_face = cube.facets()[2]
sage: stacked_square = cube.stack(square_face)
sage: stacked_square.f_vector()
(1, 9, 16, 9, 1)

sage: edge_face = cube.faces(1)[3]
sage: stacked_edge = cube.stack(edge_face)
sage: stacked_edge.f_vector()
(1, 9, 17, 10, 1)

sage: cube.stack(cube.faces(0)[0])
Traceback (most recent call last):
...
ValueError: cannot stack onto a vertex

sage: stacked_square_half = cube.stack(square_face, position=1/2)
sage: stacked_square_half.f_vector()
(1, 9, 16, 9, 1)
sage: stacked_square_large = cube.stack(square_face, position=10)

sage: # needs sage.rings.number_field
sage: hexaprism = polytopes.regular_polygon(6).prism()
sage: hexaprism.f_vector()
(1, 12, 18, 8, 1)
sage: square_face = hexaprism.faces(2)[2]
sage: stacked_hexaprism = hexaprism.stack(square_face)
sage: stacked_hexaprism.f_vector()
(1, 13, 22, 11, 1)

sage: hexaprism.stack(square_face, position=4)                          #␣
↪needs sage.rings.number_field
Traceback (most recent call last):
...
ValueError: the chosen position is too large

sage: s = polytopes.simplex(7)
sage: f = s.faces(3)[69]
sage: sf = s.stack(f); sf
A 7-dimensional polyhedron in QQ^8 defined as the convex hull of 9 vertices
sage: sf.vertices()
(A vertex at (-4, -4, -4, -4, 17/4, 17/4, 17/4, 17/4),
 A vertex at (0, 0, 0, 0, 0, 0, 0, 1),
 A vertex at (0, 0, 0, 0, 0, 0, 1, 0),
 A vertex at (0, 0, 0, 0, 0, 1, 0, 0),
 A vertex at (0, 0, 0, 0, 1, 0, 0, 0),
 A vertex at (0, 0, 0, 1, 0, 0, 0, 0),
 A vertex at (0, 0, 1, 0, 0, 0, 0, 0),
```

(continues on next page)

(continued from previous page)

```
 A vertex at (0, 1, 0, 0, 0, 0, 0, 0),
 A vertex at (1, 0, 0, 0, 0, 0, 0, 0))
```

```
>>> from sage.all import *
>>> cube = polytopes.cube()
>>> square_face = cube.facets()[Integer(2)]
>>> stacked_square = cube.stack(square_face)
>>> stacked_square.f_vector()
(1, 9, 16, 9, 1)

>>> edge_face = cube.faces(Integer(1))[Integer(3)]
>>> stacked_edge = cube.stack(edge_face)
>>> stacked_edge.f_vector()
(1, 9, 17, 10, 1)

>>> cube.stack(cube.faces(Integer(0))[Integer(0)])
Traceback (most recent call last):
...
ValueError: cannot stack onto a vertex

>>> stacked_square_half = cube.stack(square_face, position=Integer(1)/
↪Integer(2))
>>> stacked_square_half.f_vector()
(1, 9, 16, 9, 1)
>>> stacked_square_large = cube.stack(square_face, position=Integer(10))

>>> # needs sage.rings.number_field
>>> hexaprism = polytopes.regular_polygon(Integer(6)).prism()
>>> hexaprism.f_vector()
(1, 12, 18, 8, 1)
>>> square_face = hexaprism.faces(Integer(2))[Integer(2)]
>>> stacked_hexaprism = hexaprism.stack(square_face)
>>> stacked_hexaprism.f_vector()
(1, 13, 22, 11, 1)

>>> hexaprism.stack(square_face, position=Integer(4))                     ␣
↪    # needs sage.rings.number_field
Traceback (most recent call last):
...
ValueError: the chosen position is too large

>>> s = polytopes.simplex(Integer(7))
>>> f = s.faces(Integer(3))[Integer(69)]
>>> sf = s.stack(f); sf
A 7-dimensional polyhedron in QQ^8 defined as the convex hull of 9 vertices
>>> sf.vertices()
(A vertex at (-4, -4, -4, -4, 17/4, 17/4, 17/4, 17/4),
 A vertex at (0, 0, 0, 0, 0, 0, 0, 1),
 A vertex at (0, 0, 0, 0, 0, 0, 1, 0),
 A vertex at (0, 0, 0, 0, 0, 1, 0, 0),
 A vertex at (0, 0, 0, 0, 1, 0, 0, 0),
 A vertex at (0, 0, 0, 1, 0, 0, 0, 0),
```

(continues on next page)

(continued from previous page)

```
 A vertex at (0, 0, 1, 0, 0, 0, 0, 0),
 A vertex at (0, 1, 0, 0, 0, 0, 0, 0),
 A vertex at (1, 0, 0, 0, 0, 0, 0, 0))
```

It is possible to stack on unbounded faces:

```
sage: Q = Polyhedron(vertices=[[0,1], [1,0]], rays=[[1,1]])
sage: E = Q.faces(1)
sage: Q.stack(E[0],1/2).Vrepresentation()
(A vertex at (0, 1),
 A vertex at (1, 0),
 A ray in the direction (1, 1),
 A vertex at (2, 0))
sage: Q.stack(E[1],1/2).Vrepresentation()
(A vertex at (0, 1),
 A vertex at (0, 2),
 A vertex at (1, 0),
 A ray in the direction (1, 1))
sage: Q.stack(E[2],1/2).Vrepresentation()
(A vertex at (0, 0),
 A vertex at (0, 1),
 A vertex at (1, 0),
 A ray in the direction (1, 1))
```

```
>>> from sage.all import *
>>> Q = Polyhedron(vertices=[[Integer(0),Integer(1)], [Integer(1),
↪Integer(0)]], rays=[[Integer(1),Integer(1)]])
>>> E = Q.faces(Integer(1))
>>> Q.stack(E[Integer(0)],Integer(1)/Integer(2)).Vrepresentation()
(A vertex at (0, 1),
 A vertex at (1, 0),
 A ray in the direction (1, 1),
 A vertex at (2, 0))
>>> Q.stack(E[Integer(1)],Integer(1)/Integer(2)).Vrepresentation()
(A vertex at (0, 1),
 A vertex at (0, 2),
 A vertex at (1, 0),
 A ray in the direction (1, 1))
>>> Q.stack(E[Integer(2)],Integer(1)/Integer(2)).Vrepresentation()
(A vertex at (0, 0),
 A vertex at (0, 1),
 A vertex at (1, 0),
 A ray in the direction (1, 1))
```

Stacking requires a proper face:

```
sage: Q.stack(Q.faces(2)[0])
Traceback (most recent call last):
...
ValueError: can only stack on proper face
```

```
>>> from sage.all import *
>>> Q.stack(Q.faces(Integer(2))[Integer(0)])
Traceback (most recent call last):
...
ValueError: can only stack on proper face
```

subdirect_sum(*other*)

Return the subdirect sum of `self` and `other`.

The subdirect sum of two polyhedron is a projection of the join of the two polytopes. It is obtained by placing the two objects in orthogonal subspaces intersecting at the origin.

INPUT:

- `other` – a *Polyhedron_base*

EXAMPLES:

```
sage: P1 = Polyhedron([[1], [2]], base_ring=ZZ)
sage: P2 = Polyhedron([[3], [4]], base_ring=QQ)
sage: sds = P1.subdirect_sum(P2); sds
A 2-dimensional polyhedron in QQ^2
 defined as the convex hull of 4 vertices
sage: sds.vertices()
(A vertex at (0, 3),
 A vertex at (0, 4),
 A vertex at (1, 0),
 A vertex at (2, 0))
```

```
>>> from sage.all import *
>>> P1 = Polyhedron([[Integer(1)], [Integer(2)]], base_ring=ZZ)
>>> P2 = Polyhedron([[Integer(3)], [Integer(4)]], base_ring=QQ)
>>> sds = P1.subdirect_sum(P2); sds
A 2-dimensional polyhedron in QQ^2
 defined as the convex hull of 4 vertices
>>> sds.vertices()
(A vertex at (0, 3),
 A vertex at (0, 4),
 A vertex at (1, 0),
 A vertex at (2, 0))
```

See also

join() *direct_sum()*

translation(*displacement*)

Return the translated polyhedron.

INPUT:

- `displacement` – a displacement vector or a list/tuple of coordinates that determines a displacement vector

OUTPUT: the translated polyhedron

EXAMPLES:

```
sage: P = Polyhedron([[0,0], [1,0], [0,1]], base_ring=ZZ)
sage: P.translation([2,1])
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 3 vertices
sage: P.translation(vector(QQ, [2,1]))
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices
```

```
>>> from sage.all import *
>>> P = Polyhedron([[Integer(0),Integer(0)], [Integer(1),Integer(0)],␣
↪[Integer(0),Integer(1)]], base_ring=ZZ)
>>> P.translation([Integer(2),Integer(1)])
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 3 vertices
>>> P.translation(vector(QQ, [Integer(2),Integer(1)]))
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices
```

truncation(*cut_frac=None*)

Return a new polyhedron formed from two points on each edge between two vertices.

INPUT:

- `cut_frac` – integer; how deeply to cut into the edge Default is $\frac{1}{3}$

OUTPUT: a Polyhedron object, truncated as described above

EXAMPLES:

```
sage: cube = polytopes.hypercube(3)
sage: trunc_cube = cube.truncation()
sage: trunc_cube.n_vertices()
24
sage: trunc_cube.n_inequalities()
14
```

```
>>> from sage.all import *
>>> cube = polytopes.hypercube(Integer(3))
>>> trunc_cube = cube.truncation()
>>> trunc_cube.n_vertices()
24
>>> trunc_cube.n_inequalities()
14
```

wedge(*face*, *width=1*)

Return the wedge over a `face` of the polytope `self`.

The wedge over a face F of a polytope P with width $w \neq 0$ is defined as:

$$(P \times \mathbb{R}) \cap \{a^\top x + |w x_{d+1}| \leq b\}$$

where $\{x | a^\top x = b\}$ is a supporting hyperplane defining F.

INPUT:

- `face` – a PolyhedronFace of `self`, the face which we take the wedge over
- `width` – a nonzero number (default: `1`); specifies how wide the wedge will be

OUTPUT:

A (bounded) polyhedron

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: P_4 = polytopes.regular_polygon(4)
sage: W1 = P_4.wedge(P_4.faces(1)[0]); W1
A 3-dimensional polyhedron in AA^3 defined as the convex hull of 6 vertices
sage: triangular_prism = polytopes.regular_polygon(3).prism()
sage: W1.is_combinatorially_isomorphic(triangular_prism)                    #␣
↪needs sage.graphs
True

sage: Q = polytopes.hypersimplex(4,2)
sage: W2 = Q.wedge(Q.faces(2)[7]); W2
A 4-dimensional polyhedron in QQ^5 defined as the convex hull of 9 vertices
sage: W2.vertices()
(A vertex at (1, 1, 0, 0, 1),
 A vertex at (1, 1, 0, 0, -1),
 A vertex at (1, 0, 1, 0, 1),
 A vertex at (1, 0, 1, 0, -1),
 A vertex at (1, 0, 0, 1, 1),
 A vertex at (1, 0, 0, 1, -1),
 A vertex at (0, 0, 1, 1, 0),
 A vertex at (0, 1, 1, 0, 0),
 A vertex at (0, 1, 0, 1, 0))

sage: W3 = Q.wedge(Q.faces(1)[11]); W3
A 4-dimensional polyhedron in QQ^5 defined as the convex hull of 10 vertices
sage: W3.vertices()
(A vertex at (1, 1, 0, 0, -2),
 A vertex at (1, 1, 0, 0, 2),
 A vertex at (1, 0, 1, 0, -2),
 A vertex at (1, 0, 1, 0, 2),
 A vertex at (1, 0, 0, 1, 1),
 A vertex at (1, 0, 0, 1, -1),
 A vertex at (0, 1, 0, 1, 0),
 A vertex at (0, 1, 1, 0, 1),
 A vertex at (0, 0, 1, 1, 0),
 A vertex at (0, 1, 1, 0, -1))

sage: C_3_7 = polytopes.cyclic_polytope(3,7)
sage: P_6 = polytopes.regular_polygon(6)                                    #␣
↪needs sage.rings.number_field
sage: W4 = P_6.wedge(P_6.faces(1)[0])                                       #␣
↪needs sage.rings.number_field
sage: W4.is_combinatorially_isomorphic(C_3_7.polar())                       #␣
↪needs sage.graphs sage.rings.number_field
True
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> P_4 = polytopes.regular_polygon(Integer(4))
>>> W1 = P_4.wedge(P_4.faces(Integer(1))[Integer(0)]); W1
A 3-dimensional polyhedron in AA^3 defined as the convex hull of 6 vertices
```

(continues on next page)

(continued from previous page)

```
>>> triangular_prism = polytopes.regular_polygon(Integer(3)).prism()
>>> W1.is_combinatorially_isomorphic(triangular_prism)                    #␣
↪needs sage.graphs
True

>>> Q = polytopes.hypersimplex(Integer(4),Integer(2))
>>> W2 = Q.wedge(Q.faces(Integer(2))[Integer(7)]); W2
A 4-dimensional polyhedron in QQ^5 defined as the convex hull of 9 vertices
>>> W2.vertices()
(A vertex at (1, 1, 0, 0, 1),
 A vertex at (1, 1, 0, 0, -1),
 A vertex at (1, 0, 1, 0, 1),
 A vertex at (1, 0, 1, 0, -1),
 A vertex at (1, 0, 0, 1, 1),
 A vertex at (1, 0, 0, 1, -1),
 A vertex at (0, 0, 1, 1, 0),
 A vertex at (0, 1, 1, 0, 0),
 A vertex at (0, 1, 0, 1, 0))

>>> W3 = Q.wedge(Q.faces(Integer(1))[Integer(11)]); W3
A 4-dimensional polyhedron in QQ^5 defined as the convex hull of 10 vertices
>>> W3.vertices()
(A vertex at (1, 1, 0, 0, -2),
 A vertex at (1, 1, 0, 0, 2),
 A vertex at (1, 0, 1, 0, -2),
 A vertex at (1, 0, 1, 0, 2),
 A vertex at (1, 0, 0, 1, 1),
 A vertex at (1, 0, 0, 1, -1),
 A vertex at (0, 1, 0, 1, 0),
 A vertex at (0, 1, 1, 0, 1),
 A vertex at (0, 0, 1, 1, 0),
 A vertex at (0, 1, 1, 0, -1))

>>> C_3_7 = polytopes.cyclic_polytope(Integer(3),Integer(7))
>>> P_6 = polytopes.regular_polygon(Integer(6))                              ␣
↪     # needs sage.rings.number_field
>>> W4 = P_6.wedge(P_6.faces(Integer(1))[Integer(0)])                         ␣
↪              # needs sage.rings.number_field
>>> W4.is_combinatorially_isomorphic(C_3_7.polar())                       #␣
↪needs sage.graphs sage.rings.number_field
True
```

REFERENCES:

For more information, see Chapter 15 of [HoDaCG17].

2.6.7 Base class for polyhedra: Methods for plotting and affine hull projection

class sage.geometry.polyhedron.base6.**Polyhedron_base6**(*parent*, *Vrep*, *Hrep*, *Vrep_minimal=None*, *Hrep_minimal=None*, *pref_rep=None*, *mutable=False*, ***kwds*)

Bases: `Polyhedron_base5`

Methods related to plotting including affine hull projection.

affine_hull(**args*, ***kwds*)

Return the affine hull of `self` as a polyhedron.

EXAMPLES:

```
sage: half_plane_in_space = Polyhedron(ieqs=[(0,1,0,0)], eqns=[(0,0,0,1)])
sage: half_plane_in_space.affine_hull().Hrepresentation()
(An equation (0, 0, 1) x + 0 == 0,)

sage: polytopes.cube().affine_hull().is_universe()
True
```

```
>>> from sage.all import *
>>> half_plane_in_space = Polyhedron(ieqs=[(Integer(0),Integer(1),Integer(0),
→Integer(0))], eqns=[(Integer(0),Integer(0),Integer(0),Integer(1))])
>>> half_plane_in_space.affine_hull().Hrepresentation()
(An equation (0, 0, 1) x + 0 == 0,)

>>> polytopes.cube().affine_hull().is_universe()
True
```

affine_hull_manifold(*name=None*, *latex_name=None*, *start_index=0*, *ambient_space=None*, *ambient_chart=None*, *names=None*, ***kwds*)

Return the affine hull of `self` as a manifold.

If `self` is full-dimensional, it is just the ambient Euclidean space. Otherwise, it is a Riemannian submanifold of the ambient Euclidean space.

INPUT:

- `ambient_space` – a `EuclideanSpace` of the ambient dimension (default: the manifold of `ambient_chart`, if provided; otherwise, a new instance of `EuclideanSpace`).
- `ambient_chart` – a chart on `ambient_space`
- `names` – names for the coordinates on the affine hull
- optional arguments accepted by *`affine_hull_projection()`*

The default chart is determined by the optional arguments of *`affine_hull_projection()`*.

EXAMPLES:

```
sage: # needs sage.symbolic
sage: triangle = Polyhedron([(1, 0, 0), (0, 1, 0), (0, 0, 1)]);  triangle
A 2-dimensional polyhedron in ZZ^3 defined as the convex hull of 3 vertices
sage: A = triangle.affine_hull_manifold(name='A'); A
2-dimensional Riemannian submanifold A embedded in the Euclidean space E^3
sage: A.embedding().display()
A → E^3
   (x0, x1) ↦ (x, y, z) = (t0 + x0, t0 + x1, t0 - x0 - x1 + 1)
sage: A.embedding().inverse().display()
E^3 → A
   (x, y, z) ↦ (x0, x1) = (x, y)
sage: A.adapted_chart()
[Chart (E^3, (x0_E3, x1_E3, t0_E3))]
```

(continues on next page)

(continued from previous page)

```
sage: A.normal().display()
n = 1/3*sqrt(3) e_x + 1/3*sqrt(3) e_y + 1/3*sqrt(3) e_z
sage: A.induced_metric()       # Need to call this before volume_form
Riemannian metric gamma on the
 2-dimensional Riemannian submanifold A embedded in the Euclidean space E^3
sage: A.volume_form()
2-form eps_gamma on the
 2-dimensional Riemannian submanifold A embedded in the Euclidean space E^3
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> triangle = Polyhedron([(Integer(1), Integer(0), Integer(0)), (Integer(0),␣
↪Integer(1), Integer(0)), (Integer(0), Integer(0), Integer(1))]);  triangle
A 2-dimensional polyhedron in ZZ^3 defined as the convex hull of 3 vertices
>>> A = triangle.affine_hull_manifold(name='A'); A
2-dimensional Riemannian submanifold A embedded in the Euclidean space E^3
>>> A.embedding().display()
A → E^3
   (x0, x1) ↦ (x, y, z) = (t0 + x0, t0 + x1, t0 - x0 - x1 + 1)
>>> A.embedding().inverse().display()
E^3 → A
   (x, y, z) ↦ (x0, x1) = (x, y)
>>> A.adapted_chart()
[Chart (E^3, (x0_E3, x1_E3, t0_E3))]
>>> A.normal().display()
n = 1/3*sqrt(3) e_x + 1/3*sqrt(3) e_y + 1/3*sqrt(3) e_z
>>> A.induced_metric()       # Need to call this before volume_form
Riemannian metric gamma on the
 2-dimensional Riemannian submanifold A embedded in the Euclidean space E^3
>>> A.volume_form()
2-form eps_gamma on the
 2-dimensional Riemannian submanifold A embedded in the Euclidean space E^3
```

Orthogonal version:

```
sage: A = triangle.affine_hull_manifold(name='A', orthogonal=True); A        #␣
↪needs sage.symbolic
2-dimensional Riemannian submanifold A embedded in the Euclidean space E^3
sage: A.embedding().display()                                                #␣
↪needs sage.symbolic
A → E^3
   (x0, x1) ↦ (x, y, z) = (t0 - 1/2*x0 - 1/3*x1 + 1,
                           t0 + 1/2*x0 - 1/3*x1, t0 + 2/3*x1)
sage: A.embedding().inverse().display()                                      #␣
↪needs sage.symbolic
E^3 → A
   (x, y, z) ↦ (x0, x1) = (-x + y + 1, -1/2*x - 1/2*y + z + 1/2)
```

```
>>> from sage.all import *
>>> A = triangle.affine_hull_manifold(name='A', orthogonal=True); A       #␣
↪needs sage.symbolic
2-dimensional Riemannian submanifold A embedded in the Euclidean space E^3
```

(continues on next page)

(continued from previous page)

```
>>> A.embedding().display()                                                  #␣
↪needs sage.symbolic
A → E^3
   (x0, x1) ↦ (x, y, z) = (t0 - 1/2*x0 - 1/3*x1 + 1,
                           t0 + 1/2*x0 - 1/3*x1, t0 + 2/3*x1)
>>> A.embedding().inverse().display()                                        #␣
↪needs sage.symbolic
E^3 → A
   (x, y, z) ↦ (x0, x1) = (-x + y + 1, -1/2*x - 1/2*y + z + 1/2)
```

Arrangement of affine hull of facets:

```
sage: # needs sage.rings.number_field sage.symbolic
sage: D = polytopes.dodecahedron()
sage: E3 = EuclideanSpace(3)
sage: submanifolds = [              # long time
....:     F.as_polyhedron().affine_hull_manifold(name=f'F{i}',
....:                                            orthogonal=True, ambient_
↪space=E3)
....:     for i, F in enumerate(D.facets())]
sage: sum(FM.plot({},               # long time, not tested                   #␣
↪needs sage.plot
....:             srange(-2, 2, 0.1), srange(-2, 2, 0.1),
....:             opacity=0.2)
....:     for FM in submanifolds) + D.plot()
Graphics3d Object
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field sage.symbolic
>>> D = polytopes.dodecahedron()
>>> E3 = EuclideanSpace(Integer(3))
>>> submanifolds = [              # long time
...     F.as_polyhedron().affine_hull_manifold(name=f'F{i}',
...                                            orthogonal=True, ambient_
↪space=E3)
...     for i, F in enumerate(D.facets())]
>>> sum(FM.plot({},               # long time, not tested                    #␣
↪needs sage.plot
...             srange(-Integer(2), Integer(2), RealNumber('0.1')), srange(-
↪Integer(2), Integer(2), RealNumber('0.1')),
...             opacity=RealNumber('0.2'))
...     for FM in submanifolds) + D.plot()
Graphics3d Object
```

Full-dimensional case:

```
sage: cube = polytopes.cube(); cube
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 8 vertices
sage: cube.affine_hull_manifold()                                             #␣
↪needs sage.symbolic
Euclidean space E^3
```

```
>>> from sage.all import *
>>> cube = polytopes.cube(); cube
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 8 vertices
>>> cube.affine_hull_manifold()                                             #␣
↪needs sage.symbolic
Euclidean space E^3
```

affine_hull_projection(*as_polyhedron*, *as_affine_map=None*, *orthogonal=False*, *orthonormal=False*, *extend=False*, *minimal=False*, *return_all_data=False*, *as_convex_set=False*)

Return the polyhedron projected into its affine hull.

Each polyhedron is contained in some smallest affine subspace (possibly the entire ambient space) – its affine hull. We provide an affine linear map that projects the ambient space of the polyhedron to the standard Euclidean space of dimension of the polyhedron, which restricts to a bijection from the affine hull.

The projection map is not unique; some parameters control the choice of the map. Other parameters control the output of the function.

INPUT:

- `as_polyhedron`, `as_convex_set` – boolean or the default `None`; one of the two to be set
- `as_affine_map` – boolean (default: `False`); control the output

  The default `as_polyhedron=None` translates to `as_polyhedron=not as_affine_map`, therefore to `as_polyhedron=True` if nothing is specified.

  If exactly one of either `as_polyhedron` or `as_affine_map` is set, then either a polyhedron or the affine transformation is returned. The affine transformation sends the embedded polytope to a fulldimensional one. It is given as a pair `(A, b)`, where A is a linear transformation and b is a vector, and the affine transformation sends `v` to `A(v)+b`.

  If both `as_polyhedron` and `as_affine_map` are set, then both are returned, encapsulated in an instance of `AffineHullProjectionData`.
- `return_all_data` – boolean (default: `False`)

  If set, then `as_polyhedron` and `as_affine_map` will set (possibly overridden) and additional (internal) data concerning the transformation is returned. Everything is encapsulated in an instance of `AffineHullProjectionData` in this case.
- `orthogonal` – boolean (default: `False`); if `True`, provide an orthogonal transformation
- `orthonormal` – boolean (default: `False`); if `True`, provide an orthonormal transformation. If the base ring does not provide the necessary square roots, the extend parameter needs to be set to `True`.
- `extend` – boolean (default: `False`); if `True`, allow base ring to be extended if necessary. This becomes relevant when requiring an orthonormal transformation.
- `minimal` – boolean (default: `False`); if `True`, when doing an extension, it computes the minimal base ring of the extension, otherwise the base ring is `AA`.

OUTPUT:

A full-dimensional polyhedron or an affine transformation, depending on the parameters `as_polyhedron` and `as_affine_map`, or an instance of `AffineHullProjectionData` containing all data (parameter `return_all_data`).

If the output is an instance of `AffineHullProjectionData`, the following fields may be set:

- `image` – the projection of the original polyhedron

- `projection_map` – the affine map as a pair whose first component is a linear transformation and its second component a shift; see above.
- `section_map` – an affine map as a pair whose first component is a linear transformation and its second component a shift. It maps the codomain of `affine_map` to the affine hull of `self`. It is a right inverse of `projection_map`.

Note that all of these data are compatible.

Todo

- make the parameters `orthogonal` and `orthonormal` work with unbounded polyhedra.

EXAMPLES:

```
sage: triangle = Polyhedron([(1,0,0), (0,1,0), (0,0,1)]);  triangle
A 2-dimensional polyhedron in ZZ^3 defined as the convex hull of 3 vertices
sage: triangle.affine_hull_projection()
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 3 vertices

sage: half3d = Polyhedron(vertices=[(3,2,1)], rays=[(1,0,0)])
sage: half3d.affine_hull_projection().Vrepresentation()
(A ray in the direction (1), A vertex at (3))
```

```
>>> from sage.all import *
>>> triangle = Polyhedron([(Integer(1),Integer(0),Integer(0)), (Integer(0),
↪Integer(1),Integer(0)), (Integer(0),Integer(0),Integer(1))]);  triangle
A 2-dimensional polyhedron in ZZ^3 defined as the convex hull of 3 vertices
>>> triangle.affine_hull_projection()
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 3 vertices

>>> half3d = Polyhedron(vertices=[(Integer(3),Integer(2),Integer(1))],␣
↪rays=[(Integer(1),Integer(0),Integer(0))])
>>> half3d.affine_hull_projection().Vrepresentation()
(A ray in the direction (1), A vertex at (3))
```

The resulting affine hulls depend on the parameter `orthogonal` and `orthonormal`:

```
sage: L = Polyhedron([[1,0], [0,1]]); L
A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: A = L.affine_hull_projection(); A
A 1-dimensional polyhedron in ZZ^1 defined as the convex hull of 2 vertices
sage: A.vertices()
(A vertex at (0), A vertex at (1))
sage: A = L.affine_hull_projection(orthogonal=True); A
A 1-dimensional polyhedron in QQ^1 defined as the convex hull of 2 vertices
sage: A.vertices()
(A vertex at (0), A vertex at (2))
sage: A = L.affine_hull_projection(orthonormal=True)                      #␣
↪needs sage.rings.number_field
Traceback (most recent call last):
...
ValueError: the base ring needs to be extended; try with "extend=True"
```

(continues on next page)

(continued from previous page)

```
sage: A = L.affine_hull_projection(orthonormal=True, extend=True); A         #␣
↪needs sage.rings.number_field
A 1-dimensional polyhedron in AA^1 defined as the convex hull of 2 vertices
sage: A.vertices()                                                            #␣
↪needs sage.rings.number_field
(A vertex at (1.414213562373095?), A vertex at (0.?e-18))
```

```
>>> from sage.all import *
>>> L = Polyhedron([[Integer(1),Integer(0)], [Integer(0),Integer(1)]]); L
A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
>>> A = L.affine_hull_projection(); A
A 1-dimensional polyhedron in ZZ^1 defined as the convex hull of 2 vertices
>>> A.vertices()
(A vertex at (0), A vertex at (1))
>>> A = L.affine_hull_projection(orthogonal=True); A
A 1-dimensional polyhedron in QQ^1 defined as the convex hull of 2 vertices
>>> A.vertices()
(A vertex at (0), A vertex at (2))
>>> A = L.affine_hull_projection(orthonormal=True)                          #␣
↪needs sage.rings.number_field
Traceback (most recent call last):
...
ValueError: the base ring needs to be extended; try with "extend=True"
>>> A = L.affine_hull_projection(orthonormal=True, extend=True); A         #␣
↪needs sage.rings.number_field
A 1-dimensional polyhedron in AA^1 defined as the convex hull of 2 vertices
>>> A.vertices()                                                            #␣
↪needs sage.rings.number_field
(A vertex at (1.414213562373095?), A vertex at (0.?e-18))
```

More generally:

```
sage: S = polytopes.simplex(); S
A 3-dimensional polyhedron in ZZ^4 defined as the convex hull of 4 vertices
sage: S.vertices()
(A vertex at (0, 0, 0, 1),
 A vertex at (0, 0, 1, 0),
 A vertex at (0, 1, 0, 0),
 A vertex at (1, 0, 0, 0))
sage: A = S.affine_hull_projection(); A
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 4 vertices
sage: A.vertices()
(A vertex at (0, 0, 0),
 A vertex at (0, 0, 1),
 A vertex at (0, 1, 0),
 A vertex at (1, 0, 0))
sage: A = S.affine_hull_projection(orthogonal=True); A
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 4 vertices
sage: A.vertices()
(A vertex at (0, 0, 0),
 A vertex at (2, 0, 0),
 A vertex at (1, 3/2, 0),
```

(continues on next page)

(continued from previous page)

```
 A vertex at (1, 1/2, 4/3))
sage: A = S.affine_hull_projection(orthonormal=True, extend=True); A        #␣
↪needs sage.rings.number_field
A 3-dimensional polyhedron in AA^3 defined as the convex hull of 4 vertices
sage: A.vertices()                                                          #␣
↪needs sage.rings.number_field
(A vertex at (0.7071067811865475?, 0.4082482904638630?, 1.154700538379252?),
 A vertex at (0.7071067811865475?, 1.224744871391589?, 0.?e-18),
 A vertex at (1.414213562373095?, 0.?e-18, 0.?e-18),
 A vertex at (0.?e-18, 0.?e-18, 0.?e-18))
```

```
>>> from sage.all import *
>>> S = polytopes.simplex(); S
A 3-dimensional polyhedron in ZZ^4 defined as the convex hull of 4 vertices
>>> S.vertices()
(A vertex at (0, 0, 0, 1),
 A vertex at (0, 0, 1, 0),
 A vertex at (0, 1, 0, 0),
 A vertex at (1, 0, 0, 0))
>>> A = S.affine_hull_projection(); A
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 4 vertices
>>> A.vertices()
(A vertex at (0, 0, 0),
 A vertex at (0, 0, 1),
 A vertex at (0, 1, 0),
 A vertex at (1, 0, 0))
>>> A = S.affine_hull_projection(orthogonal=True); A
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 4 vertices
>>> A.vertices()
(A vertex at (0, 0, 0),
 A vertex at (2, 0, 0),
 A vertex at (1, 3/2, 0),
 A vertex at (1, 1/2, 4/3))
>>> A = S.affine_hull_projection(orthonormal=True, extend=True); A        #␣
↪needs sage.rings.number_field
A 3-dimensional polyhedron in AA^3 defined as the convex hull of 4 vertices
>>> A.vertices()                                                          #␣
↪needs sage.rings.number_field
(A vertex at (0.7071067811865475?, 0.4082482904638630?, 1.154700538379252?),
 A vertex at (0.7071067811865475?, 1.224744871391589?, 0.?e-18),
 A vertex at (1.414213562373095?, 0.?e-18, 0.?e-18),
 A vertex at (0.?e-18, 0.?e-18, 0.?e-18))
```

With the parameter `minimal` one can get a minimal base ring:

```
sage: # needs sage.rings.number_field
sage: s = polytopes.simplex(3)
sage: s_AA = s.affine_hull_projection(orthonormal=True, extend=True)
sage: s_AA.base_ring()
Algebraic Real Field
sage: s_full = s.affine_hull_projection(orthonormal=True, extend=True,
....:                                   minimal=True)
```

(continues on next page)

(continued from previous page)

```
sage: s_full.base_ring()
Number Field in a with defining polynomial y^4 - 4*y^2 + 1
 with a = 0.5176380902050415?
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> s = polytopes.simplex(Integer(3))
>>> s_AA = s.affine_hull_projection(orthonormal=True, extend=True)
>>> s_AA.base_ring()
Algebraic Real Field
>>> s_full = s.affine_hull_projection(orthonormal=True, extend=True,
...                                   minimal=True)
>>> s_full.base_ring()
Number Field in a with defining polynomial y^4 - 4*y^2 + 1
 with a = 0.5176380902050415?
```

More examples with the `orthonormal` parameter:

```
sage: P = polytopes.permutahedron(3); P
A 2-dimensional polyhedron in ZZ^3 defined as the convex hull of 6 vertices
sage: set([F.as_polyhedron().affine_hull_projection(                        #␣
↪needs sage.combinat sage.rings.number_field
....:          orthonormal=True, extend=True).volume()
....:     for F in P.affine_hull_projection().faces(1)]) == {1, sqrt(AA(2))}
True
sage: set([F.as_polyhedron().affine_hull_projection(                        #␣
↪needs sage.combinat sage.rings.number_field
....:          orthonormal=True, extend=True).volume()
....:     for F in P.affine_hull_projection(
....:             orthonormal=True, extend=True).faces(1)]) == {sqrt(AA(2))}
True

sage: # needs sage.rings.number_field
sage: D = polytopes.dodecahedron()
sage: F = D.faces(2)[0].as_polyhedron()
sage: F.affine_hull_projection(orthogonal=True)
A 2-dimensional polyhedron in
 (Number Field in sqrt5 with defining polynomial x^2 - 5
  with sqrt5 = 2.236067977499790?)^2
 defined as the convex hull of 5 vertices
sage: F.affine_hull_projection(orthonormal=True, extend=True)
A 2-dimensional polyhedron in AA^2 defined as the convex hull of 5 vertices

sage: # needs sage.rings.number_field
sage: K.<sqrt2> = QuadraticField(2)
sage: P = Polyhedron([2*[K.zero()],2*[sqrt2]]); P
A 1-dimensional polyhedron in
 (Number Field in sqrt2 with defining polynomial x^2 - 2
  with sqrt2 = 1.414213562373095?)^2
 defined as the convex hull of 2 vertices
sage: P.vertices()
(A vertex at (0, 0), A vertex at (sqrt2, sqrt2))
```

(continues on next page)

(continued from previous page)

```
sage: A = P.affine_hull_projection(orthonormal=True); A
A 1-dimensional polyhedron in
 (Number Field in sqrt2 with defining polynomial x^2 - 2
  with sqrt2 = 1.414213562373095?)^1
 defined as the convex hull of 2 vertices
sage: A.vertices()
(A vertex at (0), A vertex at (2))

sage: # needs sage.rings.number_field
sage: K.<sqrt3> = QuadraticField(3)
sage: P = Polyhedron([2*[K.zero()], 2*[sqrt3]]); P
A 1-dimensional polyhedron in
 (Number Field in sqrt3 with defining polynomial x^2 - 3
  with sqrt3 = 1.732050807568878?)^2
 defined as the convex hull of 2 vertices
sage: P.vertices()
(A vertex at (0, 0), A vertex at (sqrt3, sqrt3))
sage: A = P.affine_hull_projection(orthonormal=True)
Traceback (most recent call last):
...
ValueError: the base ring needs to be extended; try with "extend=True"
sage: A = P.affine_hull_projection(orthonormal=True, extend=True); A
A 1-dimensional polyhedron in AA^1 defined as the convex hull of 2 vertices
sage: A.vertices()
(A vertex at (0), A vertex at (2.449489742783178?))
sage: sqrt(6).n()
2.44948974278318
```

```
>>> from sage.all import *
>>> P = polytopes.permutahedron(Integer(3)); P
A 2-dimensional polyhedron in ZZ^3 defined as the convex hull of 6 vertices
>>> set([F.as_polyhedron().affine_hull_projection(                          #␣
↪needs sage.combinat sage.rings.number_field
...           orthonormal=True, extend=True).volume()
...      for F in P.affine_hull_projection().faces(Integer(1))]) ==
↪{Integer(1), sqrt(AA(Integer(2)))}
True
>>> set([F.as_polyhedron().affine_hull_projection(                          #␣
↪needs sage.combinat sage.rings.number_field
...           orthonormal=True, extend=True).volume()
...      for F in P.affine_hull_projection(
...              orthonormal=True, extend=True).faces(Integer(1))]) ==
↪{sqrt(AA(Integer(2)))}
True

>>> # needs sage.rings.number_field
>>> D = polytopes.dodecahedron()
>>> F = D.faces(Integer(2))[Integer(0)].as_polyhedron()
>>> F.affine_hull_projection(orthogonal=True)
A 2-dimensional polyhedron in
 (Number Field in sqrt5 with defining polynomial x^2 - 5
  with sqrt5 = 2.236067977499790?)^2
```

(continues on next page)

(continued from previous page)

```
 defined as the convex hull of 5 vertices
>>> F.affine_hull_projection(orthonormal=True, extend=True)
A 2-dimensional polyhedron in AA^2 defined as the convex hull of 5 vertices

>>> # needs sage.rings.number_field
>>> K = QuadraticField(Integer(2), names=('sqrt2',)); (sqrt2,) = K._first_
↪ngens(1)
>>> P = Polyhedron([Integer(2)*[K.zero()],Integer(2)*[sqrt2]]); P
A 1-dimensional polyhedron in
 (Number Field in sqrt2 with defining polynomial x^2 - 2
  with sqrt2 = 1.414213562373095?)^2
 defined as the convex hull of 2 vertices
>>> P.vertices()
(A vertex at (0, 0), A vertex at (sqrt2, sqrt2))
>>> A = P.affine_hull_projection(orthonormal=True); A
A 1-dimensional polyhedron in
 (Number Field in sqrt2 with defining polynomial x^2 - 2
  with sqrt2 = 1.414213562373095?)^1
 defined as the convex hull of 2 vertices
>>> A.vertices()
(A vertex at (0), A vertex at (2))

>>> # needs sage.rings.number_field
>>> K = QuadraticField(Integer(3), names=('sqrt3',)); (sqrt3,) = K._first_
↪ngens(1)
>>> P = Polyhedron([Integer(2)*[K.zero()], Integer(2)*[sqrt3]]); P
A 1-dimensional polyhedron in
 (Number Field in sqrt3 with defining polynomial x^2 - 3
  with sqrt3 = 1.732050807568878?)^2
 defined as the convex hull of 2 vertices
>>> P.vertices()
(A vertex at (0, 0), A vertex at (sqrt3, sqrt3))
>>> A = P.affine_hull_projection(orthonormal=True)
Traceback (most recent call last):
...
ValueError: the base ring needs to be extended; try with "extend=True"
>>> A = P.affine_hull_projection(orthonormal=True, extend=True); A
A 1-dimensional polyhedron in AA^1 defined as the convex hull of 2 vertices
>>> A.vertices()
(A vertex at (0), A vertex at (2.449489742783178?))
>>> sqrt(Integer(6)).n()
2.44948974278318
```

The affine hull is combinatorially equivalent to the input:

```
sage: P.is_combinatorially_isomorphic(P.affine_hull_projection())             #␣
↪needs sage.rings.number_field
True
sage: P.is_combinatorially_isomorphic(P.affine_hull_projection(               #␣
↪needs sage.rings.number_field
....:     orthogonal=True))
True
```

(continues on next page)

(continued from previous page)

```
sage: P.is_combinatorially_isomorphic(P.affine_hull_projection(             #␣
↪needs sage.rings.number_field
....:     orthonormal=True, extend=True))
True
```

```
>>> from sage.all import *
>>> P.is_combinatorially_isomorphic(P.affine_hull_projection())           #␣
↪needs sage.rings.number_field
True
>>> P.is_combinatorially_isomorphic(P.affine_hull_projection(             #␣
↪needs sage.rings.number_field
...     orthogonal=True))
True
>>> P.is_combinatorially_isomorphic(P.affine_hull_projection(             #␣
↪needs sage.rings.number_field
...     orthonormal=True, extend=True))
True
```

The `orthonormal=True` parameter preserves volumes; it provides an isometric copy of the polyhedron:

```
sage: # needs sage.rings.number_field
sage: Pentagon = polytopes.dodecahedron().faces(2)[0].as_polyhedron()
sage: P = Pentagon.affine_hull_projection(orthonormal=True, extend=True)
sage: _, c= P.is_inscribed(certificate=True)
sage: c
(0.4721359549995794?, 0.6498393924658126?)
sage: circumradius = (c - vector(P.vertices()[0])).norm()
sage: p = polytopes.regular_polygon(5)
sage: p.volume()
2.377641290737884?
sage: P.volume()
1.53406271079097?
sage: p.volume()*circumradius^2
1.534062710790965?
sage: P.volume() == p.volume()*circumradius^2
True
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> Pentagon = polytopes.dodecahedron().faces(Integer(2))[Integer(0)].as_
↪polyhedron()
>>> P = Pentagon.affine_hull_projection(orthonormal=True, extend=True)
>>> _, c= P.is_inscribed(certificate=True)
>>> c
(0.4721359549995794?, 0.6498393924658126?)
>>> circumradius = (c - vector(P.vertices()[Integer(0)])).norm()
>>> p = polytopes.regular_polygon(Integer(5))
>>> p.volume()
2.377641290737884?
>>> P.volume()
1.53406271079097?
>>> p.volume()*circumradius**Integer(2)
```

(continues on next page)

(continued from previous page)

```
1.534062710790965?
>>> P.volume() == p.volume()*circumradius**Integer(2)
True
```

One can also use `orthogonal` parameter to calculate volumes; in this case we don't need to switch base rings. One has to divide by the square root of the determinant of the linear part of the affine transformation times its transpose:

```
sage: # needs sage.rings.number_field
sage: Pentagon = polytopes.dodecahedron().faces(2)[0].as_polyhedron()
sage: Pnormal = Pentagon.affine_hull_projection(orthonormal=True,
....:                                           extend=True)
sage: Pgonal  = Pentagon.affine_hull_projection(orthogonal=True)
sage: A, b    = Pentagon.affine_hull_projection(orthogonal=True,
....:                                           as_affine_map=True)
sage: Adet = (A.matrix().transpose()*A.matrix()).det()
sage: Pnormal.volume()
1.534062710790965?
sage: Pgonal.volume()/Adet.sqrt(extend=True)
-80*(55*sqrt(5) - 123)/sqrt(-6368*sqrt(5) + 14240)
sage: Pgonal.volume()/AA(Adet).sqrt().n(digits=20)
1.5340627107909646813
sage: AA(Pgonal.volume()^2) == (Pnormal.volume()^2)*AA(Adet)
True
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> Pentagon = polytopes.dodecahedron().faces(Integer(2))[Integer(0)].as_
↪polyhedron()
>>> Pnormal = Pentagon.affine_hull_projection(orthonormal=True,
...                                           extend=True)
>>> Pgonal  = Pentagon.affine_hull_projection(orthogonal=True)
>>> A, b    = Pentagon.affine_hull_projection(orthogonal=True,
...                                           as_affine_map=True)
>>> Adet = (A.matrix().transpose()*A.matrix()).det()
>>> Pnormal.volume()
1.534062710790965?
>>> Pgonal.volume()/Adet.sqrt(extend=True)
-80*(55*sqrt(5) - 123)/sqrt(-6368*sqrt(5) + 14240)
>>> Pgonal.volume()/AA(Adet).sqrt().n(digits=Integer(20))
1.5340627107909646813
>>> AA(Pgonal.volume()**Integer(2)) == (Pnormal.volume()**Integer(2))*AA(Adet)
True
```

Another example with `as_affine_map=True`:

```
sage: # needs sage.combinat sage.rings.number_field
sage: P = polytopes.permutahedron(4)
sage: Q    = P.affine_hull_projection(orthonormal=True, extend=True)
sage: A, b = P.affine_hull_projection(orthonormal=True, extend=True,
....:                                 as_affine_map=True)
sage: Q.center()
```

(continues on next page)

(continued from previous page)

```
(0.7071067811865475?, 1.224744871391589?, 1.732050807568878?)
sage: A(P.center()) + b == Q.center()
True
```

```
>>> from sage.all import *
>>> # needs sage.combinat sage.rings.number_field
>>> P = polytopes.permutahedron(Integer(4))
>>> Q    = P.affine_hull_projection(orthonormal=True, extend=True)
>>> A, b = P.affine_hull_projection(orthonormal=True, extend=True,
...                                 as_affine_map=True)
>>> Q.center()
(0.7071067811865475?, 1.224744871391589?, 1.732050807568878?)
>>> A(P.center()) + b == Q.center()
True
```

For unbounded, non full-dimensional polyhedra, the `orthogonal=True` and `orthonormal=True` is not implemented:

```
sage: P = Polyhedron(ieqs=[[0, 1, 0], [0, 0, 1], [0, 0, -1]]); P
A 1-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and␣
↪1 ray
sage: P.is_compact()
False
sage: P.is_full_dimensional()
False
sage: P.affine_hull_projection(orthogonal=True)
Traceback (most recent call last):
...
NotImplementedError: "orthogonal=True" and "orthonormal=True"
work only for compact polyhedra
sage: P.affine_hull_projection(orthonormal=True)
Traceback (most recent call last):
...
NotImplementedError: "orthogonal=True" and "orthonormal=True"
work only for compact polyhedra
```

```
>>> from sage.all import *
>>> P = Polyhedron(ieqs=[[Integer(0), Integer(1), Integer(0)], [Integer(0),␣
↪Integer(0), Integer(1)], [Integer(0), Integer(0), -Integer(1)]]); P
A 1-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and␣
↪1 ray
>>> P.is_compact()
False
>>> P.is_full_dimensional()
False
>>> P.affine_hull_projection(orthogonal=True)
Traceback (most recent call last):
...
NotImplementedError: "orthogonal=True" and "orthonormal=True"
work only for compact polyhedra
>>> P.affine_hull_projection(orthonormal=True)
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
NotImplementedError: "orthogonal=True" and "orthonormal=True"
work only for compact polyhedra
```

Setting `as_affine_map` to `True` without `orthogonal` or `orthonormal` set to `True`:

```
sage: S = polytopes.simplex()
sage: S.affine_hull_projection(as_affine_map=True)
(Vector space morphism represented by the matrix:
 [1 0 0]
 [0 1 0]
 [0 0 1]
 [0 0 0]
 Domain: Vector space of dimension 4 over Rational Field
 Codomain: Vector space of dimension 3 over Rational Field,
 (0, 0, 0))
```

```
>>> from sage.all import *
>>> S = polytopes.simplex()
>>> S.affine_hull_projection(as_affine_map=True)
(Vector space morphism represented by the matrix:
 [1 0 0]
 [0 1 0]
 [0 0 1]
 [0 0 0]
 Domain: Vector space of dimension 4 over Rational Field
 Codomain: Vector space of dimension 3 over Rational Field,
 (0, 0, 0))
```

If the polyhedron is full-dimensional, it is returned:

```
sage: polytopes.cube().affine_hull_projection()
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 8 vertices
sage: polytopes.cube().affine_hull_projection(as_affine_map=True)
(Vector space morphism represented by the matrix:
 [1 0 0]
 [0 1 0]
 [0 0 1]
 Domain: Vector space of dimension 3 over Rational Field
 Codomain: Vector space of dimension 3 over Rational Field,
 (0, 0, 0))
```

```
>>> from sage.all import *
>>> polytopes.cube().affine_hull_projection()
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 8 vertices
>>> polytopes.cube().affine_hull_projection(as_affine_map=True)
(Vector space morphism represented by the matrix:
 [1 0 0]
 [0 1 0]
 [0 0 1]
 Domain: Vector space of dimension 3 over Rational Field
 Codomain: Vector space of dimension 3 over Rational Field,
```

(continues on next page)

(continued from previous page)

```
 (0, 0, 0))
```

Return polyhedron and affine map:

```
sage: S = polytopes.simplex(2)
sage: data = S.affine_hull_projection(orthogonal=True,
....:                                   as_polyhedron=True,
....:                                   as_affine_map=True); data
AffineHullProjectionData(image=A 2-dimensional polyhedron in QQ^2
                                defined as the convex hull of 3 vertices,
    projection_linear_map=Vector space morphism represented by the matrix:
        [  -1 -1/2]
        [   1 -1/2]
        [   0    1]
        Domain:   Vector space of dimension 3 over Rational Field
        Codomain: Vector space of dimension 2 over Rational Field,
    projection_translation=(1, 1/2),
    section_linear_map=None,
    section_translation=None)
```

```
>>> from sage.all import *
>>> S = polytopes.simplex(Integer(2))
>>> data = S.affine_hull_projection(orthogonal=True,
...                                   as_polyhedron=True,
...                                   as_affine_map=True); data
AffineHullProjectionData(image=A 2-dimensional polyhedron in QQ^2
                                defined as the convex hull of 3 vertices,
    projection_linear_map=Vector space morphism represented by the matrix:
        [  -1 -1/2]
        [   1 -1/2]
        [   0    1]
        Domain:   Vector space of dimension 3 over Rational Field
        Codomain: Vector space of dimension 2 over Rational Field,
    projection_translation=(1, 1/2),
    section_linear_map=None,
    section_translation=None)
```

Return all data:

```
sage: data = S.affine_hull_projection(orthogonal=True, return_all_data=True);
↪data
AffineHullProjectionData(image=A 2-dimensional polyhedron in QQ^2
                                defined as the convex hull of 3 vertices,
    projection_linear_map=Vector space morphism represented by the matrix:
        [  -1 -1/2]
        [   1 -1/2]
        [   0    1]
        Domain:   Vector space of dimension 3 over Rational Field
        Codomain: Vector space of dimension 2 over Rational Field,
    projection_translation=(1, 1/2),
    section_linear_map=Vector space morphism represented by the matrix:
        [-1/2  1/2    0]
```

(continues on next page)

(continued from previous page)

```
        [-1/3 -1/3  2/3]
        Domain:   Vector space of dimension 2 over Rational Field
        Codomain: Vector space of dimension 3 over Rational Field,
    section_translation=(1, 0, 0))
```

```
>>> from sage.all import *
>>> data = S.affine_hull_projection(orthogonal=True, return_all_data=True);␣
↪data
AffineHullProjectionData(image=A 2-dimensional polyhedron in QQ^2
                              defined as the convex hull of 3 vertices,
    projection_linear_map=Vector space morphism represented by the matrix:
        [  -1 -1/2]
        [   1 -1/2]
        [   0    1]
        Domain:   Vector space of dimension 3 over Rational Field
        Codomain: Vector space of dimension 2 over Rational Field,
    projection_translation=(1, 1/2),
    section_linear_map=Vector space morphism represented by the matrix:
        [-1/2  1/2    0]
        [-1/3 -1/3  2/3]
        Domain:   Vector space of dimension 2 over Rational Field
        Codomain: Vector space of dimension 3 over Rational Field,
    section_translation=(1, 0, 0))
```

The section map is a right inverse of the projection map:

```
sage: mat = data.section_linear_map.matrix().transpose()
sage: data.image.linear_transformation(mat) + data.section_translation == S
True
```

```
>>> from sage.all import *
>>> mat = data.section_linear_map.matrix().transpose()
>>> data.image.linear_transformation(mat) + data.section_translation == S
True
```

Same without `orthogonal=True`:

```
sage: data = S.affine_hull_projection(return_all_data=True); data
AffineHullProjectionData(image=A 2-dimensional polyhedron in ZZ^2
                              defined as the convex hull of 3 vertices,
    projection_linear_map=Vector space morphism represented by the matrix:
        [1 0]
        [0 1]
        [0 0]
        Domain:   Vector space of dimension 3 over Rational Field
        Codomain: Vector space of dimension 2 over Rational Field,
    projection_translation=(0, 0),
    section_linear_map=Vector space morphism represented by the matrix:
        [ 1  0 -1]
        [ 0  1 -1]
        Domain:   Vector space of dimension 2 over Rational Field
        Codomain: Vector space of dimension 3 over Rational Field,
```

(continues on next page)

(continued from previous page)

```
    section_translation=(0, 0, 1))
sage: mat = data.section_linear_map.matrix().transpose()
sage: data.image.linear_transformation(mat) + data.section_translation == S
True
```

```
>>> from sage.all import *
>>> data = S.affine_hull_projection(return_all_data=True); data
AffineHullProjectionData(image=A 2-dimensional polyhedron in ZZ^2
                               defined as the convex hull of 3 vertices,
    projection_linear_map=Vector space morphism represented by the matrix:
        [1 0]
        [0 1]
        [0 0]
        Domain:   Vector space of dimension 3 over Rational Field
        Codomain: Vector space of dimension 2 over Rational Field,
    projection_translation=(0, 0),
    section_linear_map=Vector space morphism represented by the matrix:
        [ 1  0 -1]
        [ 0  1 -1]
        Domain:   Vector space of dimension 2 over Rational Field
        Codomain: Vector space of dimension 3 over Rational Field,
    section_translation=(0, 0, 1))
>>> mat = data.section_linear_map.matrix().transpose()
>>> data.image.linear_transformation(mat) + data.section_translation == S
True
```

```
sage: P0 = Polyhedron(
....:     ieqs=[(0, -1, 0, 1, 1, 1), (0, 1, 1, 0, -1, -1), (0, -1, 1, 1, 0,␣
↪0),
....:           (0, 1, 0, 0, 0, 0), (0, 0, 1, 1, -1, -1), (0, 0, 0, 0, 0, 1),
....:           (0, 0, 0, 0, 1, 0), (0, 0, 0, 1, 0, -1), (0, 0, 1, 0, 0, 0)])
sage: P = P0.intersection(Polyhedron(eqns=[(-1, 1, 1, 1, 1, 1)]))
sage: P.dim()
4
sage: P.affine_hull_projection(orthogonal=True, as_affine_map=True)[0]
Vector space morphism represented by the matrix:
[    0     0     0   1/3]
[ -2/3  -1/6     0 -1/12]
[  1/3  -1/6   1/2 -1/12]
[    0   1/2     0 -1/12]
[  1/3  -1/6  -1/2 -1/12]
Domain: Vector space of dimension 5 over Rational Field
Codomain: Vector space of dimension 4 over Rational Field
```

```
>>> from sage.all import *
>>> P0 = Polyhedron(
...     ieqs=[(Integer(0), -Integer(1), Integer(0), Integer(1), Integer(1),␣
↪Integer(1)), (Integer(0), Integer(1), Integer(1), Integer(0), -Integer(1), -
↪Integer(1)), (Integer(0), -Integer(1), Integer(1), Integer(1), Integer(0),␣
↪Integer(0)),
...           (Integer(0), Integer(1), Integer(0), Integer(0), Integer(0),␣
```

(continues on next page)

(continued from previous page)

```
↪Integer(0)), (Integer(0), Integer(0), Integer(1), Integer(1), -Integer(1), -
↪Integer(1)), (Integer(0), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(1)),
...          (Integer(0), Integer(0), Integer(0), Integer(0), Integer(1),␣
↪Integer(0)), (Integer(0), Integer(0), Integer(0), Integer(1), Integer(0), -
↪Integer(1)), (Integer(0), Integer(0), Integer(1), Integer(0), Integer(0),␣
↪Integer(0))])
>>> P = P0.intersection(Polyhedron(eqns=[(-Integer(1), Integer(1), Integer(1),
↪ Integer(1), Integer(1), Integer(1))]))
>>> P.dim()
4
>>> P.affine_hull_projection(orthogonal=True, as_affine_map=True)[Integer(0)]
Vector space morphism represented by the matrix:
[    0     0     0   1/3]
[ -2/3  -1/6     0 -1/12]
[  1/3  -1/6   1/2 -1/12]
[    0   1/2     0 -1/12]
[  1/3  -1/6  -1/2 -1/12]
Domain: Vector space of dimension 5 over Rational Field
Codomain: Vector space of dimension 4 over Rational Field
```

gale_transform()

Return the Gale transform of a polytope as described in the reference below.

OUTPUT:

A list of vectors, the Gale transform. The dimension is the dimension of the affine dependencies of the vertices of the polytope.

EXAMPLES:

This is from the reference, for a triangular prism:

```
sage: p = Polyhedron(vertices = [[0,0],[0,1],[1,0]])
sage: p2 = p.prism()
sage: p2.gale_transform()
((-1, 0), (0, -1), (1, 1), (-1, -1), (1, 0), (0, 1))
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices = [[Integer(0),Integer(0)],[Integer(0),
↪Integer(1)],[Integer(1),Integer(0)]])
>>> p2 = p.prism()
>>> p2.gale_transform()
((-1, 0), (0, -1), (1, 1), (-1, -1), (1, 0), (0, 1))
```

REFERENCES:

Lectures in Geometric Combinatorics, R.R.Thomas, 2006, AMS Press.

See also

`gale_transform_to_polyhedron()`.

plot(*point=None*, *line=None*, *polygon=None*, *wireframe='blue'*, *fill='green'*, *position=None*, *orthonormal=True*, ***kwds*)

Return a graphical representation.

INPUT:

- `point`, `line`, `polygon` – parameters to pass to point (0d), line (1d), and polygon (2d) plot commands. Allowed values are:
  - A Python dictionary to be passed as keywords to the plot commands.
  - A string or triple of numbers: The color. This is equivalent to passing the dictionary `{'color':...}`.
  - `False`: Switches off the drawing of the corresponding graphics object
- `wireframe`, `fill` – similar to `point`, `line`, and `polygon`, but `fill` is used for the graphics objects in the dimension of the polytope (or of dimension 2 for higher dimensional polytopes) and `wireframe` is used for all lower-dimensional graphics objects (default: 'green' for `fill` and 'blue' for `wireframe`)
- `position` – positive number; the position to take the projection point in Schlegel diagrams
- `orthonormal` – boolean (default: `True`); whether to use orthonormal projections
- `**kwds` – optional keyword parameters that are passed to all graphics objects

OUTPUT:

A (multipart) graphics object.

EXAMPLES:

```
sage: square = polytopes.hypercube(2)
sage: point = Polyhedron([[1,1]])
sage: line = Polyhedron([[1,1],[2,1]])
sage: cube = polytopes.hypercube(3)
sage: hypercube = polytopes.hypercube(4)
```

```
>>> from sage.all import *
>>> square = polytopes.hypercube(Integer(2))
>>> point = Polyhedron([[Integer(1),Integer(1)]])
>>> line = Polyhedron([[Integer(1),Integer(1)],[Integer(2),Integer(1)]])
>>> cube = polytopes.hypercube(Integer(3))
>>> hypercube = polytopes.hypercube(Integer(4))
```

By default, the wireframe is rendered in blue and the fill in green:

```
sage: # needs sage.plot
sage: square.plot()
Graphics object consisting of 6 graphics primitives
sage: point.plot()
Graphics object consisting of 1 graphics primitive
sage: line.plot()
Graphics object consisting of 2 graphics primitives
sage: cube.plot()
Graphics3d Object
sage: hypercube.plot()
Graphics3d Object
```

```
>>> from sage.all import *
>>> # needs sage.plot
>>> square.plot()
Graphics object consisting of 6 graphics primitives
>>> point.plot()
Graphics object consisting of 1 graphics primitive
>>> line.plot()
Graphics object consisting of 2 graphics primitives
>>> cube.plot()
Graphics3d Object
>>> hypercube.plot()
Graphics3d Object
```

Draw the lines in red and nothing else:

```
sage: # needs sage.plot
sage: square.plot(point=False, line='red', polygon=False)
Graphics object consisting of 4 graphics primitives
sage: point.plot(point=False, line='red', polygon=False)
Graphics object consisting of 0 graphics primitives
sage: line.plot(point=False, line='red', polygon=False)
Graphics object consisting of 1 graphics primitive
sage: cube.plot(point=False, line='red', polygon=False)
Graphics3d Object
sage: hypercube.plot(point=False, line='red', polygon=False)
Graphics3d Object
```

```
>>> from sage.all import *
>>> # needs sage.plot
>>> square.plot(point=False, line='red', polygon=False)
Graphics object consisting of 4 graphics primitives
>>> point.plot(point=False, line='red', polygon=False)
Graphics object consisting of 0 graphics primitives
>>> line.plot(point=False, line='red', polygon=False)
Graphics object consisting of 1 graphics primitive
>>> cube.plot(point=False, line='red', polygon=False)
Graphics3d Object
>>> hypercube.plot(point=False, line='red', polygon=False)
Graphics3d Object
```

Draw points in red, no lines, and a blue polygon:

```
sage: # needs sage.plot
sage: square.plot(point={'color':'red'}, line=False, polygon=(0,0,1))
Graphics object consisting of 2 graphics primitives
sage: point.plot(point={'color':'red'}, line=False, polygon=(0,0,1))
Graphics object consisting of 1 graphics primitive
sage: line.plot(point={'color':'red'}, line=False, polygon=(0,0,1))
Graphics object consisting of 1 graphics primitive
sage: cube.plot(point={'color':'red'}, line=False, polygon=(0,0,1))
Graphics3d Object
sage: hypercube.plot(point={'color':'red'}, line=False, polygon=(0,0,1))
Graphics3d Object
```

```
>>> from sage.all import *
>>> # needs sage.plot
>>> square.plot(point={'color':'red'}, line=False, polygon=(Integer(0),
↪Integer(0),Integer(1)))
Graphics object consisting of 2 graphics primitives
>>> point.plot(point={'color':'red'}, line=False, polygon=(Integer(0),
↪Integer(0),Integer(1)))
Graphics object consisting of 1 graphics primitive
>>> line.plot(point={'color':'red'}, line=False, polygon=(Integer(0),
↪Integer(0),Integer(1)))
Graphics object consisting of 1 graphics primitive
>>> cube.plot(point={'color':'red'}, line=False, polygon=(Integer(0),
↪Integer(0),Integer(1)))
Graphics3d Object
>>> hypercube.plot(point={'color':'red'}, line=False, polygon=(Integer(0),
↪Integer(0),Integer(1)))
Graphics3d Object
```

If we instead use the `fill` and `wireframe` options, the coloring depends on the dimension of the object:

```
sage: # needs sage.plot
sage: square.plot(fill='green', wireframe='red')
Graphics object consisting of 6 graphics primitives
sage: point.plot(fill='green', wireframe='red')
Graphics object consisting of 1 graphics primitive
sage: line.plot(fill='green', wireframe='red')
Graphics object consisting of 2 graphics primitives
sage: cube.plot(fill='green', wireframe='red')
Graphics3d Object
sage: hypercube.plot(fill='green', wireframe='red')
Graphics3d Object
```

```
>>> from sage.all import *
>>> # needs sage.plot
>>> square.plot(fill='green', wireframe='red')
Graphics object consisting of 6 graphics primitives
>>> point.plot(fill='green', wireframe='red')
Graphics object consisting of 1 graphics primitive
>>> line.plot(fill='green', wireframe='red')
Graphics object consisting of 2 graphics primitives
>>> cube.plot(fill='green', wireframe='red')
Graphics3d Object
>>> hypercube.plot(fill='green', wireframe='red')
Graphics3d Object
```

It is possible to draw polyhedra up to dimension 4, no matter what the ambient dimension is:

```
sage: hcube = polytopes.hypercube(5)
sage: facet = hcube.facets()[0].as_polyhedron(); facet
A 4-dimensional polyhedron in ZZ^5 defined as the convex hull of 16 vertices
sage: facet.plot()                                                          #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> hcube = polytopes.hypercube(Integer(5))
>>> facet = hcube.facets()[Integer(0)].as_polyhedron(); facet
A 4-dimensional polyhedron in ZZ^5 defined as the convex hull of 16 vertices
>>> facet.plot()                                                        #␣
↪needs sage.plot
Graphics3d Object
```

For a 3d plot, we may draw the polygons with rainbow colors, using any of the following ways:

```
sage: cube.plot(polygon='rainbow')                                        #␣
↪needs sage.plot
Graphics3d Object
sage: cube.plot(polygon={'color':'rainbow'})                              #␣
↪needs sage.plot
Graphics3d Object
sage: cube.plot(fill='rainbow')                                           #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> cube.plot(polygon='rainbow')                                        #␣
↪needs sage.plot
Graphics3d Object
>>> cube.plot(polygon={'color':'rainbow'})                              #␣
↪needs sage.plot
Graphics3d Object
>>> cube.plot(fill='rainbow')                                           #␣
↪needs sage.plot
Graphics3d Object
```

For a 3d plot, the size of a point, the thickness of a line and the width of an arrow are controlled by the respective parameters:

```
sage: prism = Polyhedron(vertices=[[0,0,0],[1,0,0],[0,1,0]], rays=[[0,0,1]])
sage: prism.plot(size=20, thickness=30, width=1)                          #␣
↪needs sage.plot
Graphics3d Object
sage: prism.plot(point={'size':20, 'color':'black'},                      #␣
↪needs sage.plot
....:            line={'thickness':30, 'width':1, 'color':'black'},
....:            polygon='rainbow')
Graphics3d Object
```

```
>>> from sage.all import *
>>> prism = Polyhedron(vertices=[[Integer(0),Integer(0),Integer(0)],
↪[Integer(1),Integer(0),Integer(0)],[Integer(0),Integer(1),Integer(0)]],␣
↪rays=[[Integer(0),Integer(0),Integer(1)]])
>>> prism.plot(size=Integer(20), thickness=Integer(30), width=Integer(1))    ␣
↪                          # needs sage.plot
Graphics3d Object
>>> prism.plot(point={'size':Integer(20), 'color':'black'},                  ␣
↪       # needs sage.plot
```

(continues on next page)

(continued from previous page)

```
...             line={'thickness':Integer(30), 'width':Integer(1), 'color':
↪'black'},
...             polygon='rainbow')
Graphics3d Object
```

projection(*projection=None*)

Return a projection object.

INPUT:

- `proj` – a projection function

OUTPUT:

The identity projection. This is useful for plotting polyhedra.

See also

schlegel_projection() for a more interesting projection.

EXAMPLES:

```
sage: p = polytopes.hypercube(3)
sage: proj = p.projection()
sage: proj
The projection of a polyhedron into 3 dimensions
```

```
>>> from sage.all import *
>>> p = polytopes.hypercube(Integer(3))
>>> proj = p.projection()
>>> proj
The projection of a polyhedron into 3 dimensions
```

render_solid(***kwds*)

Return a solid rendering of a 2- or 3-d polytope.

EXAMPLES:

```
sage: p = polytopes.hypercube(3)
sage: p_solid = p.render_solid(opacity=.7)                                     #␣
↪needs sage.plot
sage: type(p_solid)                                                            #␣
↪needs sage.plot
<class 'sage.plot.plot3d.index_face_set.IndexFaceSet'>
```

```
>>> from sage.all import *
>>> p = polytopes.hypercube(Integer(3))
>>> p_solid = p.render_solid(opacity=RealNumber('.7'))                         ␣
↪          # needs sage.plot
>>> type(p_solid)                                                            #␣
↪needs sage.plot
<class 'sage.plot.plot3d.index_face_set.IndexFaceSet'>
```

render_wireframe(***kwds*)

For polytopes in 2 or 3 dimensions, return the edges as a list of lines.

EXAMPLES:

```
sage: p = Polyhedron([[1,2,],[1,1],[0,0]])
sage: p_wireframe = p.render_wireframe()                                    #␣
↪needs sage.plot
sage: p_wireframe._objects                                                  #␣
↪needs sage.plot
[Line defined by 2 points, Line defined by 2 points, Line defined by 2 points]
```

```
>>> from sage.all import *
>>> p = Polyhedron([[Integer(1),Integer(2),],[Integer(1),Integer(1)],
↪[Integer(0),Integer(0)]])
>>> p_wireframe = p.render_wireframe()                                    #␣
↪needs sage.plot
>>> p_wireframe._objects                                                  #␣
↪needs sage.plot
[Line defined by 2 points, Line defined by 2 points, Line defined by 2 points]
```

schlegel_projection(*facet=None*, *position=None*)

Return the Schlegel projection.

- The facet is orthonormally transformed into its affine hull.
- The position specifies a point coming out of the barycenter of the facet from which the other vertices will be projected into the facet.

INPUT:

- `facet` – a *PolyhedronFace* The facet into which the Schlegel diagram is created. The default is the first facet
- `position` – a positive number. Determines a relative distance from the barycenter of `facet`. A value close to 0 will place the projection point close to the facet and a large value further away. Default is 1. If the given value is too large, an error is returned.

OUTPUT: a *Projection* object

EXAMPLES:

```
sage: p = polytopes.hypercube(3)
sage: sch_proj = p.schlegel_projection()
sage: schlegel_edge_indices = sch_proj.lines
sage: schlegel_edges = [sch_proj.coordinates_of(x) for x in schlegel_edge_
↪indices]
sage: len([x for x in schlegel_edges if x[0][0] > 0])
8
```

```
>>> from sage.all import *
>>> p = polytopes.hypercube(Integer(3))
>>> sch_proj = p.schlegel_projection()
>>> schlegel_edge_indices = sch_proj.lines
>>> schlegel_edges = [sch_proj.coordinates_of(x) for x in schlegel_edge_
↪indices]
```

(continues on next page)

(continued from previous page)

```
>>> len([x for x in schlegel_edges if x[Integer(0)][Integer(0)] > Integer(0)])
8
```

The Schlegel projection preserves the convexity of facets, see Issue #30015:

```
sage: fcube = polytopes.hypercube(4)
sage: tfcube = fcube.face_truncation(fcube.faces(0)[0])
sage: tfcube.facets()[-1]
A 3-dimensional face of a Polyhedron in QQ^4 defined as the convex hull of 8␣
↪vertices
sage: sp = tfcube.schlegel_projection(tfcube.facets()[-1])
sage: sp.plot()                                                              #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> fcube = polytopes.hypercube(Integer(4))
>>> tfcube = fcube.face_truncation(fcube.faces(Integer(0))[Integer(0)])
>>> tfcube.facets()[-Integer(1)]
A 3-dimensional face of a Polyhedron in QQ^4 defined as the convex hull of 8␣
↪vertices
>>> sp = tfcube.schlegel_projection(tfcube.facets()[-Integer(1)])
>>> sp.plot()                                                              #␣
↪needs sage.plot
Graphics3d Object
```

The same truncated cube but see inside the tetrahedral facet:

```
sage: tfcube.facets()[4]
A 3-dimensional face of a Polyhedron in QQ^4 defined as the convex hull of 4␣
↪vertices
sage: sp = tfcube.schlegel_projection(tfcube.facets()[4])                    #␣
↪needs sage.symbolic
sage: sp.plot()                                                              #␣
↪needs sage.plot sage.symbolic
Graphics3d Object
```

```
>>> from sage.all import *
>>> tfcube.facets()[Integer(4)]
A 3-dimensional face of a Polyhedron in QQ^4 defined as the convex hull of 4␣
↪vertices
>>> sp = tfcube.schlegel_projection(tfcube.facets()[Integer(4)])             ␣
↪     # needs sage.symbolic
>>> sp.plot()                                                              #␣
↪needs sage.plot sage.symbolic
Graphics3d Object
```

A different values of `position` changes the projection:

```
sage: # needs sage.symbolic
sage: sp = tfcube.schlegel_projection(tfcube.facets()[4], 1/2)
sage: sp.plot()                                                              #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.plot
Graphics3d Object
sage: sp = tfcube.schlegel_projection(tfcube.facets()[4], 4)
sage: sp.plot()                                                          #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> # needs sage.symbolic
>>> sp = tfcube.schlegel_projection(tfcube.facets()[Integer(4)], Integer(1)/
↪Integer(2))
>>> sp.plot()                                                           #␣
↪needs sage.plot
Graphics3d Object
>>> sp = tfcube.schlegel_projection(tfcube.facets()[Integer(4)], Integer(4))
>>> sp.plot()                                                           #␣
↪needs sage.plot
Graphics3d Object
```

A value which is too large give a projection point that sees more than one facet resulting in a error:

```
sage: sp = tfcube.schlegel_projection(tfcube.facets()[4], 5)
Traceback (most recent call last):
...
ValueError: the chosen position is too large
```

```
>>> from sage.all import *
>>> sp = tfcube.schlegel_projection(tfcube.facets()[Integer(4)], Integer(5))
Traceback (most recent call last):
...
ValueError: the chosen position is too large
```

show(***kwds*)

Display graphics immediately.

This method attempts to display the graphics immediately, without waiting for the currently running code (if any) to return to the command line. Be careful, calling it from within a loop will potentially launch a large number of external viewer programs.

INPUT:

- `kwds` – optional keyword arguments; see `plot()` for the description of available options

OUTPUT:

This method does not return anything. Use `plot()` if you want to generate a graphics object that can be saved or further transformed.

EXAMPLES:

```
sage: square = polytopes.hypercube(2)
sage: square.show(point='red')                                           #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> square = polytopes.hypercube(Integer(2))
>>> square.show(point='red')                                          #␣
↪needs sage.plot
```

tikz(*view=[0, 0, 1]*, *angle=0*, *scale=1*, *edge_color='blue!95!black'*, *facet_color='blue!95!black'*, *opacity=0.8*, *vertex_color='green'*, *axis=False*, *output_type=None*)

Return a tikz picture of `self` as a string or as a `TikzPicture` according to a projection `view` and an angle `angle` obtained via the threejs viewer. `self` must be bounded.

INPUT:

- `view` – list (default: [0,0,1]) representing the rotation axis (see note below)
- `angle` – integer (default: 0); angle of rotation in degree from 0 to 360 (see note below)
- `scale` – integer (default: 1); the scaling of the tikz picture
- `edge_color` – string (default: `'blue!95!black'`); representing colors which tikz recognizes
- `facet_color` – string (default: `'blue!95!black'`); representing colors which tikz recognizes
- `vertex_color` – string (default: `'green'`); representing colors which tikz recognizes
- `opacity` – real number (default: 0.8) between 0 and 1 giving the opacity of the front facets
- `axis` – boolean (default: `False`); draw the axes at the origin or not
- `output_type` – string (default: `None`); valid values are `None` (deprecated), `'LatexExpr'` and `'TikzPicture'`, whether to return a LatexExpr object (which inherits from Python str) or a `TikzPicture` object from module `sage.misc.latex_standalone`

OUTPUT: LatexExpr object or TikzPicture object

Note

This is a wrapper of a method of the projection object `self.projection()`. See `tikz()` for more detail.

The inputs `view` and `angle` can be obtained by visualizing it using `.show(aspect_ratio=1)`. This will open an interactive view in your default browser, where you can rotate the polytope. Once the desired view angle is found, click on the information icon in the lower right-hand corner and select *Get Viewpoint*. This will copy a string of the form '[x,y,z],angle' to your local clipboard. Go back to Sage and type `Img = P.tikz([x,y,z],angle)`.

The inputs `view` and `angle` can also be obtained from the viewer Jmol:

```
1) Right click on the image
2) Select ``Console``
3) Select the tab ``State``
4) Scroll to the line ``moveto``
```

It reads something like:

```
moveto 0.0 {x y z angle} Scale
```

The `view` is then [x,y,z] and `angle` is angle. The following number is the scale.

Jmol performs a rotation of `angle` degrees along the vector [x,y,z] and show the result from the z-axis.

EXAMPLES:

```
sage: # needs sage.plot
sage: co = polytopes.cuboctahedron()
sage: Img = co.tikz([0, 0, 1], 0, output_type='TikzPicture')
sage: Img
\documentclass[tikz]{standalone}
\begin{document}
\begin{tikzpicture}%
        [x={(1.000000cm, 0.000000cm)},
        y={(0.000000cm, 1.000000cm)},
        z={(0.000000cm, 0.000000cm)},
        scale=1.000000,
...
Use print to see the full content.
...
\node[vertex] at (1.00000, 0.00000, 1.00000)     {};
\node[vertex] at (1.00000, 1.00000, 0.00000)     {};
%%
%%
\end{tikzpicture}
\end{document}
sage: print('\n'.join(Img.content().splitlines()[12:21]))
%% with the command: ._tikz_3d_in_3d and parameters:
%% view = [0, 0, 1]
%% angle = 0
%% scale = 1
%% edge_color = blue!95!black
%% facet_color = blue!95!black
%% opacity = 0.8
%% vertex_color = green
%% axis = False
sage: print('\n'.join(Img.content().splitlines()[22:26]))
%% Coordinate of the vertices:
%%
\coordinate (-1.00000, -1.00000, 0.00000) at (-1.00000, -1.00000, 0.00000);
\coordinate (-1.00000, 0.00000, -1.00000) at (-1.00000, 0.00000, -1.00000);
```

```
>>> from sage.all import *
>>> # needs sage.plot
>>> co = polytopes.cuboctahedron()
>>> Img = co.tikz([Integer(0), Integer(0), Integer(1)], Integer(0), output_
↪type='TikzPicture')
>>> Img
\documentclass[tikz]{standalone}
\begin{document}
\begin{tikzpicture}%
        [x={(1.000000cm, 0.000000cm)},
        y={(0.000000cm, 1.000000cm)},
        z={(0.000000cm, 0.000000cm)},
        scale=1.000000,
...
Use print to see the full content.
...
```

(continues on next page)

(continued from previous page)

```
\node[vertex] at (1.00000, 0.00000, 1.00000)     {};
\node[vertex] at (1.00000, 1.00000, 0.00000)     {};
%%
%%
\end{tikzpicture}
\end{document}
>>> print('\n'.join(Img.content().splitlines()[Integer(12):Integer(21)]))
%% with the command: ._tikz_3d_in_3d and parameters:
%% view = [0, 0, 1]
%% angle = 0
%% scale = 1
%% edge_color = blue!95!black
%% facet_color = blue!95!black
%% opacity = 0.8
%% vertex_color = green
%% axis = False
>>> print('\n'.join(Img.content().splitlines()[Integer(22):Integer(26)]))
%% Coordinate of the vertices:
%%
\coordinate (-1.00000, -1.00000, 0.00000) at (-1.00000, -1.00000, 0.00000);
\coordinate (-1.00000, 0.00000, -1.00000) at (-1.00000, 0.00000, -1.00000);
```

When output type is a `sage.misc.latex_standalone.TikzPicture`:

```
sage: # needs sage.plot
sage: co = polytopes.cuboctahedron()
sage: t = co.tikz([674, 108, -731], 112, output_type='TikzPicture'); t
\documentclass[tikz]{standalone}
\begin{document}
\begin{tikzpicture}%
        [x={(0.249656cm, -0.577639cm)},
        y={(0.777700cm, -0.358578cm)},
        z={(-0.576936cm, -0.733318cm)},
        scale=1.000000,
...
Use print to see the full content.
...
\node[vertex] at (1.00000, 0.00000, 1.00000)     {};
\node[vertex] at (1.00000, 1.00000, 0.00000)     {};
%%
%%
\end{tikzpicture}
\end{document}
sage: path_to_file = t.pdf()         # not tested
```

```
>>> from sage.all import *
>>> # needs sage.plot
>>> co = polytopes.cuboctahedron()
>>> t = co.tikz([Integer(674), Integer(108), -Integer(731)], Integer(112),␣
↪output_type='TikzPicture'); t
\documentclass[tikz]{standalone}
\begin{document}
```

(continues on next page)

(continued from previous page)

```
\begin{tikzpicture}%
        [x={(0.249656cm, -0.577639cm)},
        y={(0.777700cm, -0.358578cm)},
        z={(-0.576936cm, -0.733318cm)},
        scale=1.000000,
...
Use print to see the full content.
...
\node[vertex] at (1.00000, 0.00000, 1.00000)     {};
\node[vertex] at (1.00000, 1.00000, 0.00000)     {};
%%
%%
\end{tikzpicture}
\end{document}
>>> path_to_file = t.pdf()        # not tested
```

2.6.8 Base class for polyhedra: Methods for triangulation and volume computation

class sage.geometry.polyhedron.base7.**Polyhedron_base7**(*parent*, *Vrep*, *Hrep*, *Vrep_minimal=None*, *Hrep_minimal=None*, *pref_rep=None*, *mutable=False*, ***kwds*)

Bases: *Polyhedron_base6*

Methods related to triangulation and volume.

centroid(*engine='auto'*, ***kwds*)

Return the center of the mass of the polytope.

The mass is taken with respect to the induced Lebesgue measure, see *volume()*.

If the polyhedron is not compact, a NotImplementedError is raised.

INPUT:

- engine – either 'auto' (default), 'internal', 'TOPCOM', or 'normaliz'. The 'internal' and 'TOPCOM' instruct this package to always use its own triangulation algorithms or TOPCOM's algorithms, respectively. By default ('auto'), TOPCOM is used if it is available and internal routines otherwise.
- **kwds – keyword arguments that are passed to the triangulation engine (see *triangulate()*)

OUTPUT: the centroid as vector

ALGORITHM:

We triangulate the polytope and find the barycenter of the simplices. We add the individual barycenters weighted by the fraction of the total mass.

EXAMPLES:

```
sage: P = polytopes.hypercube(2).pyramid()
sage: P.centroid()
(1/4, 0, 0)

sage: P = polytopes.associahedron(['A', 2])                                    #␣
↪needs sage.combinat
sage: P.centroid()                                                              #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.combinat
(2/21, 2/21)

sage: P = polytopes.permutahedron(4, backend='normaliz')     # optional -␣
↪pynormaliz
sage: P.centroid()                                           # optional -␣
↪pynormaliz
(5/2, 5/2, 5/2, 5/2)
```

```
>>> from sage.all import *
>>> P = polytopes.hypercube(Integer(2)).pyramid()
>>> P.centroid()
(1/4, 0, 0)

>>> P = polytopes.associahedron(['A', Integer(2)])                          ␣
↪     # needs sage.combinat
>>> P.centroid()                                                         #␣
↪needs sage.combinat
(2/21, 2/21)

>>> P = polytopes.permutahedron(Integer(4), backend='normaliz')     # optional␣
↪- pynormaliz
>>> P.centroid()                                                 # optional -␣
↪pynormaliz
(5/2, 5/2, 5/2, 5/2)
```

The method is not implemented for unbounded polyhedra:

```
sage: P = Polyhedron(vertices=[(0, 0)], rays=[(1, 0), (0, 1)])
sage: P.centroid()
Traceback (most recent call last):
...
NotImplementedError: the polyhedron is not compact
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[(Integer(0), Integer(0))], rays=[(Integer(1),␣
↪Integer(0)), (Integer(0), Integer(1))])
>>> P.centroid()
Traceback (most recent call last):
...
NotImplementedError: the polyhedron is not compact
```

The centroid of an empty polyhedron is not defined:

```
sage: Polyhedron().centroid()
Traceback (most recent call last):
...
ZeroDivisionError: rational division by zero
```

```
>>> from sage.all import *
>>> Polyhedron().centroid()
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
...
ZeroDivisionError: rational division by zero
```

integrate(*function*, *measure='ambient'*, ***kwds*)

Return the integral of `function` over this polytope.

INPUT:

- `self` – Polyhedron
- `function` – a multivariate polynomial or a valid LattE description string for polynomials
- `measure` – string, the measure to use

  Allowed values are:

  – `ambient` (default): Lebesgue measure of ambient space,

  – `induced`: Lebesgue measure of the affine hull,

  – `induced_nonnormalized`: Lebesgue measure of the affine hull without the normalization by $\sqrt{\det(A^\top A)}$ (with A being the affine transformation matrix; see `affine_hull()`).

- `**kwds` – additional keyword arguments that are passed to the engine

OUTPUT: the integral of the polynomial over the polytope

> **Note**
>
> The polytope triangulation algorithm is used. This function depends on LattE (i.e., the `latte_int` optional package).

EXAMPLES:

```
sage: P = polytopes.cube()
sage: x, y, z = polygens(QQ, 'x, y, z')
sage: P.integrate(x^2*y^2*z^2)                               # optional -␣
↪latte_int
8/27
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> x, y, z = polygens(QQ, 'x, y, z')
>>> P.integrate(x**Integer(2)*y**Integer(2)*z**Integer(2))                  ␣
↪          # optional - latte_int
8/27
```

If the polyhedron has floating point coordinates, an inexact result can be obtained if we transform to rational coordinates:

```
sage: P = 1.4142*polytopes.cube()
sage: P_QQ = Polyhedron(vertices=[[QQ(vi) for vi in v] for v in P.vertex_
↪generator()])
sage: RDF(P_QQ.integrate(x^2*y^2*z^2))                       # optional -␣
↪latte_int
6.703841212195228
```

```
>>> from sage.all import *
>>> P = RealNumber('1.4142')*polytopes.cube()
>>> P_QQ = Polyhedron(vertices=[[QQ(vi) for vi in v] for v in P.vertex_
↪generator()])
>>> RDF(P_QQ.integrate(x**Integer(2)*y**Integer(2)*z**Integer(2)))                ␣
↪           # optional - latte_int
6.703841212195228
```

Integral over a non full-dimensional polytope:

```
sage: x, y = polygens(QQ, 'x, y')
sage: P = Polyhedron(vertices=[[0,0], [1,1]])
sage: P.integrate(x*y)
0
sage: ixy = P.integrate(x*y, measure='induced'); ixy        # optional -␣
↪latte_int
0.4714045207910317?
sage: ixy.parent()                                          # optional -␣
↪latte_int
Algebraic Real Field
```

```
>>> from sage.all import *
>>> x, y = polygens(QQ, 'x, y')
>>> P = Polyhedron(vertices=[[Integer(0),Integer(0)], [Integer(1),
↪Integer(1)]])
>>> P.integrate(x*y)
0
>>> ixy = P.integrate(x*y, measure='induced'); ixy        # optional - latte_
↪int
0.4714045207910317?
>>> ixy.parent()                                          # optional - latte_
↪int
Algebraic Real Field
```

Convert to a symbolic expression:

```
sage: ixy.radical_expression()                              # optional -␣
↪latte_int
1/3*sqrt(2)
```

```
>>> from sage.all import *
>>> ixy.radical_expression()                              # optional - latte_
↪int
1/3*sqrt(2)
```

Another non full-dimensional polytope integration:

```
sage: R.<x, y, z> = QQ[]
sage: P = polytopes.simplex(2)
sage: V = AA(P.volume(measure='induced'))                                  #␣
↪needs sage.rings.number_field
sage: V.radical_expression()                                               #␣
↪needs sage.rings.number_field sage.symbolic
```

(continues on next page)

(continued from previous page)

```
1/2*sqrt(3)
sage: P.integrate(R(1), measure='induced') == V          # optional -␣
↪latte_int, needs sage.rings.number_field sage.symbolic
True
```

```
>>> from sage.all import *
>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> P = polytopes.simplex(Integer(2))
>>> V = AA(P.volume(measure='induced'))                              #␣
↪needs sage.rings.number_field
>>> V.radical_expression()                                           #␣
↪needs sage.rings.number_field sage.symbolic
1/2*sqrt(3)
>>> P.integrate(R(Integer(1)), measure='induced') == V          # optional␣
↪- latte_int, needs sage.rings.number_field sage.symbolic
True
```

Computing the mass center:

```
sage: (P.integrate(x, measure='induced')                 # optional -␣
↪latte_int, needs sage.rings.number_field sage.symbolic
....:      / V).radical_expression()
1/3
sage: (P.integrate(y, measure='induced')                 # optional -␣
↪latte_int, needs sage.rings.number_field sage.symbolic
....:      / V).radical_expression()
1/3
sage: (P.integrate(z, measure='induced')                 # optional -␣
↪latte_int, needs sage.rings.number_field sage.symbolic
....:      / V).radical_expression()
1/3
```

```
>>> from sage.all import *
>>> (P.integrate(x, measure='induced')                 # optional - latte_
↪int, needs sage.rings.number_field sage.symbolic
...      / V).radical_expression()
1/3
>>> (P.integrate(y, measure='induced')                 # optional - latte_
↪int, needs sage.rings.number_field sage.symbolic
...      / V).radical_expression()
1/3
>>> (P.integrate(z, measure='induced')                 # optional - latte_
↪int, needs sage.rings.number_field sage.symbolic
...      / V).radical_expression()
1/3
```

triangulate(*engine='auto'*, *connected=True*, *fine=False*, *regular=None*, *star=None*)

Return a triangulation of the polytope.

INPUT:

- `engine` – either 'auto' (default), 'internal', 'TOPCOM', or 'normaliz'. The 'internal' and 'TOPCOM' instruct this package to always use its own triangulation algorithms or TOPCOM's algorithms, respectively.

By default ('auto'), TOPCOM is used if it is available and internal routines otherwise.

The remaining keyword parameters are passed through to the *PointConfiguration* constructor:

- `connected` – boolean (default: `True`); whether the triangulations should be connected to the regular triangulations via bistellar flips. These are much easier to compute than all triangulations.
- `fine` – boolean (default: `False`); whether the triangulations must be fine, that is, make use of all points of the configuration
- `regular` – boolean or `None` (default: `None`); whether the triangulations must be regular. A regular triangulation is one that is induced by a piecewise-linear convex support function. In other words, the shadows of the faces of a polyhedron in one higher dimension.
  - `True`: Only regular triangulations.
  - `False`: Only non-regular triangulations.
  - `None` (default): Both kinds of triangulation.
- `star` – either `None` (default) or a point. Whether the triangulations must be star. A triangulation is star if all maximal simplices contain a common point. The central point can be specified by its index (an integer) in the given points or by its coordinates (anything iterable.)

OUTPUT:

A triangulation of the convex hull of the vertices as a *Triangulation*. The indices in the triangulation correspond to the *Vrepresentation()* objects.

EXAMPLES:

```
sage: cube = polytopes.hypercube(3)
sage: triangulation = cube.triangulate(
....:    engine='internal')  # to make doctest independent of TOPCOM
sage: triangulation
(<0,1,2,7>, <0,1,5,7>, <0,2,3,7>, <0,3,4,7>, <0,4,5,7>, <1,5,6,7>)
sage: simplex_indices = triangulation[0]; simplex_indices
(0, 1, 2, 7)
sage: simplex_vertices = [cube.Vrepresentation(i) for i in simplex_indices]
sage: simplex_vertices
[A vertex at (1, -1, -1),
 A vertex at (1, 1, -1),
 A vertex at (1, 1, 1),
 A vertex at (-1, 1, 1)]
sage: Polyhedron(simplex_vertices)
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 4 vertices
```

```
>>> from sage.all import *
>>> cube = polytopes.hypercube(Integer(3))
>>> triangulation = cube.triangulate(
...    engine='internal')  # to make doctest independent of TOPCOM
>>> triangulation
(<0,1,2,7>, <0,1,5,7>, <0,2,3,7>, <0,3,4,7>, <0,4,5,7>, <1,5,6,7>)
>>> simplex_indices = triangulation[Integer(0)]; simplex_indices
(0, 1, 2, 7)
>>> simplex_vertices = [cube.Vrepresentation(i) for i in simplex_indices]
>>> simplex_vertices
[A vertex at (1, -1, -1),
 A vertex at (1, 1, -1),
```

(continues on next page)

(continued from previous page)

```
 A vertex at (1, 1, 1),
 A vertex at (-1, 1, 1)]
>>> Polyhedron(simplex_vertices)
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 4 vertices
```

It is possible to use 'normaliz' as an engine. For this, the polyhedron should have the backend set to normaliz:

```
sage: P = Polyhedron(vertices=[[0,0,1], [1,0,1],            # optional -␣
↪pynormaliz
....:                          [0,1,1], [1,1,1]],
....:                backend='normaliz')
sage: P.triangulate(engine='normaliz')                      # optional -␣
↪pynormaliz
(<0,1,2>, <1,2,3>)

sage: P = Polyhedron(vertices=[[0,0,1], [1,0,1],
....:                          [0,1,1], [1,1,1]])
sage: P.triangulate(engine='normaliz')
Traceback (most recent call last):
...
TypeError: the polyhedron's backend should be 'normaliz'
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[[Integer(0),Integer(0),Integer(1)], [Integer(1),
↪Integer(0),Integer(1)],            # optional - pynormaliz
...                          [Integer(0),Integer(1),Integer(1)], [Integer(1),
↪Integer(1),Integer(1)]],
...                backend='normaliz')
>>> P.triangulate(engine='normaliz')                        # optional -␣
↪pynormaliz
(<0,1,2>, <1,2,3>)

>>> P = Polyhedron(vertices=[[Integer(0),Integer(0),Integer(1)], [Integer(1),
↪Integer(0),Integer(1)],
...                          [Integer(0),Integer(1),Integer(1)], [Integer(1),
↪Integer(1),Integer(1)]])
>>> P.triangulate(engine='normaliz')
Traceback (most recent call last):
...
TypeError: the polyhedron's backend should be 'normaliz'
```

The normaliz engine can triangulate pointed cones:

```
sage: # optional - pynormaliz
sage: C1 = Polyhedron(rays=[[0,0,1], [1,0,1],
....:                       [0,1,1], [1,1,1]],
....:                 backend='normaliz')
sage: C1.triangulate(engine='normaliz')
(<0,1,2>, <1,2,3>)
sage: C2 = Polyhedron(rays=[[1,0,1], [0,0,1],
....:                       [0,1,1], [1,1,10/9]],
```

(continues on next page)

(continued from previous page)

```
....:                    backend='normaliz')
sage: C2.triangulate(engine='normaliz')
(<0,1,2>, <1,2,3>)
```

```
>>> from sage.all import *
>>> # optional - pynormaliz
>>> C1 = Polyhedron(rays=[[Integer(0),Integer(0),Integer(1)], [Integer(1),
↪Integer(0),Integer(1)],
...                         [Integer(0),Integer(1),Integer(1)], [Integer(1),
↪Integer(1),Integer(1)]],
...                    backend='normaliz')
>>> C1.triangulate(engine='normaliz')
(<0,1,2>, <1,2,3>)
>>> C2 = Polyhedron(rays=[[Integer(1),Integer(0),Integer(1)], [Integer(0),
↪Integer(0),Integer(1)],
...                         [Integer(0),Integer(1),Integer(1)], [Integer(1),
↪Integer(1),Integer(10)/Integer(9)]],
...                    backend='normaliz')
>>> C2.triangulate(engine='normaliz')
(<0,1,2>, <1,2,3>)
```

They can also be affine cones:

```
sage: K = Polyhedron(vertices=[[1,1,1]],                     # optional -␣
↪pynormaliz
....:                rays=[[1,0,0], [0,1,0], [1,1,-1], [1,1,1]],
....:                backend='normaliz')
sage: K.triangulate(engine='normaliz')                       # optional -␣
↪pynormaliz
(<0,1,2>, <0,1,3>)
```

```
>>> from sage.all import *
>>> K = Polyhedron(vertices=[[Integer(1),Integer(1),Integer(1)]],              ␣
↪       # optional - pynormaliz
...                rays=[[Integer(1),Integer(0),Integer(0)], [Integer(0),
↪Integer(1),Integer(0)], [Integer(1),Integer(1),-Integer(1)], [Integer(1),
↪Integer(1),Integer(1)]],
...                backend='normaliz')
>>> K.triangulate(engine='normaliz')                      # optional -␣
↪pynormaliz
(<0,1,2>, <0,1,3>)
```

volume(*measure='ambient'*, *engine='auto'*, ***kwds*)

Return the volume of the polytope.

INPUT:

- `measure` – string. The measure to use. Allowed values are:
  - `ambient` (default): Lebesgue measure of ambient space (volume)
  - `induced`: Lebesgue measure of the affine hull (relative volume)
  - `induced_rational`: Scaling of the Lebesgue measure for rational polytopes, such that the unit hypercube has volume 1

  - `induced_lattice`: Scaling of the Lebesgue measure, such that the volume of the hypercube is factorial(n)
- `engine` – string. The backend to use. Allowed values are:
  - `'auto'` (default): choose engine according to measure
  - `'internal'`: see `triangulate()`
  - `'TOPCOM'`: see `triangulate()`
  - `'lrs'`: use David Avis's lrs program (optional)
  - `'latte'`: use LattE integrale program (optional)
  - `'normaliz'`: use Normaliz program (optional)
- `**kwds` – keyword arguments that are passed to the triangulation engine

OUTPUT: the volume of the polytope

EXAMPLES:

```
sage: polytopes.hypercube(3).volume()
8
sage: (polytopes.hypercube(3)*2).volume()
64
sage: polytopes.twenty_four_cell().volume()
2
```

```
>>> from sage.all import *
>>> polytopes.hypercube(Integer(3)).volume()
8
>>> (polytopes.hypercube(Integer(3))*Integer(2)).volume()
64
>>> polytopes.twenty_four_cell().volume()
2
```

Volume of the same polytopes, using the optional package lrslib (which requires a rational polytope):

```
sage: I3 = polytopes.hypercube(3)
sage: I3.volume(engine='lrs')                                    # optional -␣
↪lrslib
8
sage: C24 = polytopes.twenty_four_cell()
sage: C24.volume(engine='lrs')                                   # optional -␣
↪lrslib
2
```

```
>>> from sage.all import *
>>> I3 = polytopes.hypercube(Integer(3))
>>> I3.volume(engine='lrs')                                    # optional - lrslib
8
>>> C24 = polytopes.twenty_four_cell()
>>> C24.volume(engine='lrs')                                   # optional - lrslib
2
```

If the base ring is exact, the answer is exact:

```
sage: P5 = polytopes.regular_polygon(5)                                   #␣
↪needs sage.rings.number_field
sage: P5.volume()                                                         #␣
↪needs sage.rings.number_field
2.377641290737884?

sage: polytopes.icosahedron().volume()                                    #␣
↪needs sage.groups sage.rings.number_field
5/12*sqrt5 + 5/4
sage: numerical_approx(_)  # abs tol 1e9                                  #␣
↪needs sage.groups sage.rings.number_field
2.18169499062491
```

```
>>> from sage.all import *
>>> P5 = polytopes.regular_polygon(Integer(5))                              ␣
↪     # needs sage.rings.number_field
>>> P5.volume()                                                          #␣
↪needs sage.rings.number_field
2.377641290737884?

>>> polytopes.icosahedron().volume()                                     #␣
↪needs sage.groups sage.rings.number_field
5/12*sqrt5 + 5/4
>>> numerical_approx(_)  # abs tol 1e9                                   #␣
↪needs sage.groups sage.rings.number_field
2.18169499062491
```

When considering lower-dimensional polytopes, we can ask for the ambient (full-dimensional), the induced measure (of the affine hull) or, in the case of lattice polytopes, for the induced rational measure. This is controlled by the parameter `measure`. Different engines may have different ideas on the definition of volume of a lower-dimensional object:

```
sage: P = Polyhedron([[0, 0], [1, 1]])
sage: P.volume()
0
sage: P.volume(measure='induced')                                         #␣
↪needs sage.rings.number_field
1.414213562373095?
sage: P.volume(measure='induced_rational')                 # optional -␣
↪latte_int
1

sage: # needs sage.rings.number_field
sage: S = polytopes.regular_polygon(6); S
A 2-dimensional polyhedron in AA^2 defined as the convex hull of 6 vertices
sage: edge = S.faces(1)[4].as_polyhedron()
sage: edge.vertices()
(A vertex at (0.866025403784439?, 1/2), A vertex at (0, 1))
sage: edge.volume()
0
sage: edge.volume(measure='induced')
1
```

(continues on next page)

(continued from previous page)

```
sage: # optional - pynormaliz
sage: P = Polyhedron(backend='normaliz',
....:                vertices=[[1,0,0], [0,0,1],
....:                          [-1,1,1], [-1,2,0]])
sage: P.volume()
0
sage: P.volume(measure='induced')                                         #␣
↪needs sage.rings.number_field
2.598076211353316?
sage: P.volume(measure='induced', engine='normaliz')
2.598076211353316
sage: P.volume(measure='induced_rational')                    # optional -␣
↪latte_int
3/2
sage: P.volume(measure='induced_rational',
....:          engine='normaliz')
3/2
sage: P.volume(measure='induced_lattice')
3
```

```
>>> from sage.all import *
>>> P = Polyhedron([[Integer(0), Integer(0)], [Integer(1), Integer(1)]])
>>> P.volume()
0
>>> P.volume(measure='induced')                                          #␣
↪needs sage.rings.number_field
1.414213562373095?
>>> P.volume(measure='induced_rational')                    # optional - latte_
↪int
1

>>> # needs sage.rings.number_field
>>> S = polytopes.regular_polygon(Integer(6)); S
A 2-dimensional polyhedron in AA^2 defined as the convex hull of 6 vertices
>>> edge = S.faces(Integer(1))[Integer(4)].as_polyhedron()
>>> edge.vertices()
(A vertex at (0.866025403784439?, 1/2), A vertex at (0, 1))
>>> edge.volume()
0
>>> edge.volume(measure='induced')
1

>>> # optional - pynormaliz
>>> P = Polyhedron(backend='normaliz',
...                vertices=[[Integer(1),Integer(0),Integer(0)], [Integer(0),
↪Integer(0),Integer(1)],
...                          [-Integer(1),Integer(1),Integer(1)], [-
↪Integer(1),Integer(2),Integer(0)]])
>>> P.volume()
0
>>> P.volume(measure='induced')                                          #␣
↪needs sage.rings.number_field
```

(continues on next page)

(continued from previous page)

```
2.598076211353316?
>>> P.volume(measure='induced', engine='normaliz')
2.598076211353316
>>> P.volume(measure='induced_rational')                    # optional - latte_
↪int
3/2
>>> P.volume(measure='induced_rational',
...          engine='normaliz')
3/2
>>> P.volume(measure='induced_lattice')
3
```

The same polytope without normaliz backend:

```
sage: P = Polyhedron(vertices=[[1,0,0], [0,0,1], [-1,1,1], [-1,2,0]])
sage: P.volume(measure='induced_lattice', engine='latte')    # optional -␣
↪latte_int
3

sage: # needs sage.groups sage.rings.number_field
sage: Dexact = polytopes.dodecahedron()
sage: F0 = Dexact.faces(2)[0].as_polyhedron()
sage: v = F0.volume(measure='induced', engine='internal'); v
1.53406271079097?
sage: F4 = Dexact.faces(2)[4].as_polyhedron()
sage: v = F4.volume(measure='induced', engine='internal'); v
1.53406271079097?
sage: RDF(v)    # abs tol 1e-9
1.53406271079044

sage: # needs sage.groups
sage: Dinexact = polytopes.dodecahedron(exact=False)
sage: F2 = Dinexact.faces(2)[2].as_polyhedron()
sage: w = F2.volume(measure='induced', engine='internal')
sage: RDF(w)    # abs tol 1e-9
1.5340627082974878

sage: all(polytopes.simplex(d).volume(measure='induced')                   #␣
↪needs sage.rings.number_field sage.symbolic
....:         == sqrt(d+1)/factorial(d)
....:     for d in range(1,5))
True

sage: I = Polyhedron([[-3, 0], [0, 9]])
sage: I.volume(measure='induced')                                          #␣
↪needs sage.rings.number_field
9.48683298050514?
sage: I.volume(measure='induced_rational')                 # optional -␣
↪latte_int
3

sage: T = Polyhedron([[3, 0, 0], [0, 4, 0], [0, 0, 5]])
```

(continues on next page)

(continued from previous page)

```
sage: T.volume(measure='induced')                                           #␣
↪needs sage.rings.number_field
13.86542462386205?
sage: T.volume(measure='induced_rational')                  # optional -␣
↪latte_int
1/2

sage: Q = Polyhedron(vertices=[(0, 0, 1, 1), (0, 1, 1, 0), (1, 1, 0, 0)])
sage: Q.volume(measure='induced')
1
sage: Q.volume(measure='induced_rational')                  # optional -␣
↪latte_int
1/2
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[[Integer(1),Integer(0),Integer(0)], [Integer(0),
↪Integer(0),Integer(1)], [-Integer(1),Integer(1),Integer(1)], [-Integer(1),
↪Integer(2),Integer(0)]])
>>> P.volume(measure='induced_lattice', engine='latte')    # optional - latte_
↪int
3

>>> # needs sage.groups sage.rings.number_field
>>> Dexact = polytopes.dodecahedron()
>>> F0 = Dexact.faces(Integer(2))[Integer(0)].as_polyhedron()
>>> v = F0.volume(measure='induced', engine='internal'); v
1.53406271079097?
>>> F4 = Dexact.faces(Integer(2))[Integer(4)].as_polyhedron()
>>> v = F4.volume(measure='induced', engine='internal'); v
1.53406271079097?
>>> RDF(v)     # abs tol 1e-9
1.53406271079044

>>> # needs sage.groups
>>> Dinexact = polytopes.dodecahedron(exact=False)
>>> F2 = Dinexact.faces(Integer(2))[Integer(2)].as_polyhedron()
>>> w = F2.volume(measure='induced', engine='internal')
>>> RDF(w)     # abs tol 1e-9
1.5340627082974878

>>> all(polytopes.simplex(d).volume(measure='induced')                    #␣
↪needs sage.rings.number_field sage.symbolic
...         == sqrt(d+Integer(1))/factorial(d)
...     for d in range(Integer(1),Integer(5)))
True

>>> I = Polyhedron([[-Integer(3), Integer(0)], [Integer(0), Integer(9)]])
>>> I.volume(measure='induced')                                           #␣
↪needs sage.rings.number_field
9.48683298050514?
>>> I.volume(measure='induced_rational')                   # optional - latte_
↪int
```

(continues on next page)

(continued from previous page)

```
3

>>> T = Polyhedron([[Integer(3), Integer(0), Integer(0)], [Integer(0),␣
↪Integer(4), Integer(0)], [Integer(0), Integer(0), Integer(5)]])
>>> T.volume(measure='induced')                                     #␣
↪needs sage.rings.number_field
13.86542462386205?
>>> T.volume(measure='induced_rational')                # optional - latte_
↪int
1/2

>>> Q = Polyhedron(vertices=[(Integer(0), Integer(0), Integer(1), Integer(1)),
↪ (Integer(0), Integer(1), Integer(1), Integer(0)), (Integer(1), Integer(1),␣
↪Integer(0), Integer(0))])
>>> Q.volume(measure='induced')
1
>>> Q.volume(measure='induced_rational')                # optional - latte_
↪int
1/2
```

The volume of a full-dimensional unbounded polyhedron is infinity:

```
sage: P = Polyhedron(vertices=[[1, 0], [0, 1]], rays=[[1, 1]])
sage: P.volume()
+Infinity
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[[Integer(1), Integer(0)], [Integer(0),␣
↪Integer(1)]], rays=[[Integer(1), Integer(1)]])
>>> P.volume()
+Infinity
```

The volume of a non full-dimensional unbounded polyhedron depends on the measure used:

```
sage: P = Polyhedron(ieqs = [[1,1,1], [-1,-1,-1], [3,1,0]]); P
A 1-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and␣
↪1 ray
sage: P.volume()
0
sage: P.volume(measure='induced')
+Infinity
sage: P.volume(measure='ambient')
0
sage: P.volume(measure='induced_rational')                # optional -␣
↪pynormaliz
+Infinity
sage: P.volume(measure='induced_rational',engine='latte')
+Infinity
```

```
>>> from sage.all import *
>>> P = Polyhedron(ieqs = [[Integer(1),Integer(1),Integer(1)], [-Integer(1),-
↪Integer(1),-Integer(1)], [Integer(3),Integer(1),Integer(0)]]); P
```

(continues on next page)

(continued from previous page)

```
A 1-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and␣
↪1 ray
>>> P.volume()
0
>>> P.volume(measure='induced')
+Infinity
>>> P.volume(measure='ambient')
0
>>> P.volume(measure='induced_rational')                  # optional -␣
↪pynormaliz
+Infinity
>>> P.volume(measure='induced_rational',engine='latte')
+Infinity
```

The volume in 0-dimensional space is taken by counting measure:

```
sage: P = Polyhedron(vertices=[[]]); P
A 0-dimensional polyhedron in ZZ^0 defined as the convex hull of 1 vertex
sage: P.volume()
1
sage: P = Polyhedron(vertices=[]); P
The empty polyhedron in ZZ^0
sage: P.volume()
0
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[[]]); P
A 0-dimensional polyhedron in ZZ^0 defined as the convex hull of 1 vertex
>>> P.volume()
1
>>> P = Polyhedron(vertices=[]); P
The empty polyhedron in ZZ^0
>>> P.volume()
0
```

2.6.9 Base class for polyhedra: Miscellaneous methods

class sage.geometry.polyhedron.base.**Polyhedron_base**(*parent*, *Vrep*, *Hrep*, *Vrep_minimal=None*, *Hrep_minimal=None*, *pref_rep=None*, *mutable=False*, ***kwds*)

Bases: *Polyhedron_base7*

Base class for Polyhedron objects.

INPUT:

- `parent` – the parent, an instance of *Polyhedra*
- `Vrep` – list `[vertices, rays, lines]` or `None`. The V-representation of the polyhedron; if `None`, the polyhedron is determined by the H-representation
- `Hrep` – list `[ieqs, eqns]` or `None`. The H-representation of the polyhedron; if `None`, the polyhedron is determined by the V-representation
- `Vrep_minimal` – (optional) see below

- `Hrep_minimal` – (optional) see below
- `pref_rep` – string (default: `None`); one of `Vrep` or `Hrep` to pick this in case the backend cannot initialize from complete double description
- `mutable` – ignored

If both `Vrep` and `Hrep` are provided, then `Vrep_minimal` and `Hrep_minimal` must be set to `True`.

barycentric_subdivision(*subdivision_frac=None*)

Return the barycentric subdivision of a compact polyhedron.

DEFINITION:

The barycentric subdivision of a compact polyhedron is a standard way to triangulate its faces in such a way that maximal faces correspond to flags of faces of the starting polyhedron (i.e. a maximal chain in the face lattice of the polyhedron). As a simplicial complex, this is known as the order complex of the face lattice of the polyhedron.

REFERENCE:

See Wikipedia article Barycentric_subdivision

Section 6.6, Handbook of Convex Geometry, Volume A, edited by P.M. Gruber and J.M. Wills. 1993, North-Holland Publishing Co..

INPUT:

- `subdivision_frac` – number. Gives the proportion how far the new vertices are pulled out of the polytope. Default is $\frac{1}{3}$ and the value should be smaller than $\frac{1}{2}$. The subdivision is computed on the polar polyhedron.

OUTPUT: a Polyhedron object, subdivided as described above

EXAMPLES:

```
sage: P = polytopes.hypercube(3)
sage: P.barycentric_subdivision()
A 3-dimensional polyhedron in QQ^3 defined as the convex hull
of 26 vertices
sage: P = Polyhedron(vertices=[[0,0,0],[0,1,0],[1,0,0],[0,0,1]])
sage: P.barycentric_subdivision()
A 3-dimensional polyhedron in QQ^3 defined as the convex hull
of 14 vertices
sage: P = Polyhedron(vertices=[[0,1,0],[0,0,1],[1,0,0]])
sage: P.barycentric_subdivision()
A 2-dimensional polyhedron in QQ^3 defined as the convex hull
of 6 vertices
sage: P = polytopes.regular_polygon(4, base_ring=QQ)                        #␣
↪needs sage.rings.number_field
sage: P.barycentric_subdivision()                                           #␣
↪needs sage.rings.number_field
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 8
vertices
```

```
>>> from sage.all import *
>>> P = polytopes.hypercube(Integer(3))
>>> P.barycentric_subdivision()
A 3-dimensional polyhedron in QQ^3 defined as the convex hull
```

(continues on next page)

(continued from previous page)

```
of 26 vertices
>>> P = Polyhedron(vertices=[[Integer(0),Integer(0),Integer(0)],[Integer(0),
↪Integer(1),Integer(0)],[Integer(1),Integer(0),Integer(0)],[Integer(0),
↪Integer(0),Integer(1)]])
>>> P.barycentric_subdivision()
A 3-dimensional polyhedron in QQ^3 defined as the convex hull
of 14 vertices
>>> P = Polyhedron(vertices=[[Integer(0),Integer(1),Integer(0)],[Integer(0),
↪Integer(0),Integer(1)],[Integer(1),Integer(0),Integer(0)]])
>>> P.barycentric_subdivision()
A 2-dimensional polyhedron in QQ^3 defined as the convex hull
of 6 vertices
>>> P = polytopes.regular_polygon(Integer(4), base_ring=QQ)                  ␣
↪     # needs sage.rings.number_field
>>> P.barycentric_subdivision()                                             #␣
↪needs sage.rings.number_field
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 8
vertices
```

boundary_complex()

Return the simplicial complex given by the boundary faces of `self`, if it is simplicial.

OUTPUT:

A (spherical) simplicial complex

EXAMPLES:

The boundary complex of the octahedron:

```
sage: # needs sage.graphs
sage: oc = polytopes.octahedron()
sage: sc_oc = oc.boundary_complex()
sage: fl_oc = oc.face_lattice()                                             #␣
↪needs sage.combinat
sage: fl_sc = sc_oc.face_poset()                                            #␣
↪needs sage.combinat
sage: [len(x) for x in fl_oc.level_sets()]                                  #␣
↪needs sage.combinat
[1, 6, 12, 8, 1]
sage: [len(x) for x in fl_sc.level_sets()]                                  #␣
↪needs sage.combinat
[6, 12, 8]
sage: sc_oc.euler_characteristic()
2
sage: sc_oc.homology()
{0: 0, 1: 0, 2: Z}
```

```
>>> from sage.all import *
>>> # needs sage.graphs
>>> oc = polytopes.octahedron()
>>> sc_oc = oc.boundary_complex()
>>> fl_oc = oc.face_lattice()                                               #␣
↪needs sage.combinat
```

(continues on next page)

(continued from previous page)

```
>>> fl_sc = sc_oc.face_poset()                                          #␣
↪needs sage.combinat
>>> [len(x) for x in fl_oc.level_sets()]                                #␣
↪needs sage.combinat
[1, 6, 12, 8, 1]
>>> [len(x) for x in fl_sc.level_sets()]                                #␣
↪needs sage.combinat
[6, 12, 8]
>>> sc_oc.euler_characteristic()
2
>>> sc_oc.homology()
{0: 0, 1: 0, 2: Z}
```

The polyhedron should be simplicial:

```
sage: c = polytopes.cube()
sage: c.boundary_complex()
Traceback (most recent call last):
...
NotImplementedError: this function is only implemented for simplicial␣
↪polytopes
```

```
>>> from sage.all import *
>>> c = polytopes.cube()
>>> c.boundary_complex()
Traceback (most recent call last):
...
NotImplementedError: this function is only implemented for simplicial␣
↪polytopes
```

bounding_box(*integral=False*, *integral_hull=False*)

Return the coordinates of a rectangular box containing the non-empty polytope.

INPUT:

- `integral` – boolean (default: `False`); whether to only allow integral coordinates in the bounding box
- `integral_hull` – boolean (default: `False`); if `True`, return a box containing the integral points of the polytope, or `None, None` if it is known that the polytope has no integral points

OUTPUT:

A pair of tuples `(box_min, box_max)` where `box_min` are the coordinates of a point bounding the coordinates of the polytope from below and `box_max` bounds the coordinates from above.

EXAMPLES:

```
sage: Polyhedron([(1/3,2/3), (2/3, 1/3)]).bounding_box()
((1/3, 1/3), (2/3, 2/3))
sage: Polyhedron([(1/3,2/3), (2/3, 1/3)]).bounding_box(integral=True)
((0, 0), (1, 1))
sage: Polyhedron([(1/3,2/3), (2/3, 1/3)]).bounding_box(integral_hull=True)
(None, None)
sage: Polyhedron([(1/3,2/3), (3/3, 4/3)]).bounding_box(integral_hull=True)
((1, 1), (1, 1))
```

(continues on next page)

(continued from previous page)

```
sage: polytopes.buckyball(exact=False).bounding_box()                       #␣
↪needs sage.groups
((-0.8090169944, -0.8090169944, -0.8090169944),
 (0.8090169944, 0.8090169944, 0.8090169944))
```

```
>>> from sage.all import *
>>> Polyhedron([(Integer(1)/Integer(3),Integer(2)/Integer(3)), (Integer(2)/
↪Integer(3), Integer(1)/Integer(3))]).bounding_box()
((1/3, 1/3), (2/3, 2/3))
>>> Polyhedron([(Integer(1)/Integer(3),Integer(2)/Integer(3)), (Integer(2)/
↪Integer(3), Integer(1)/Integer(3))]).bounding_box(integral=True)
((0, 0), (1, 1))
>>> Polyhedron([(Integer(1)/Integer(3),Integer(2)/Integer(3)), (Integer(2)/
↪Integer(3), Integer(1)/Integer(3))]).bounding_box(integral_hull=True)
(None, None)
>>> Polyhedron([(Integer(1)/Integer(3),Integer(2)/Integer(3)), (Integer(3)/
↪Integer(3), Integer(4)/Integer(3))]).bounding_box(integral_hull=True)
((1, 1), (1, 1))
>>> polytopes.buckyball(exact=False).bounding_box()                       #␣
↪needs sage.groups
((-0.8090169944, -0.8090169944, -0.8090169944),
 (0.8090169944, 0.8090169944, 0.8090169944))
```

center()

Return the average of the vertices.

See also

sage.geometry.polyhedron.base1.Polyhedron_base1.representative_point().

OUTPUT:

The center of the polyhedron. All rays and lines are ignored. Raises a `ZeroDivisionError` for the empty polytope.

EXAMPLES:

```
sage: p = polytopes.hypercube(3)
sage: p = p + vector([1,0,0])
sage: p.center()
(1, 0, 0)
```

```
>>> from sage.all import *
>>> p = polytopes.hypercube(Integer(3))
>>> p = p + vector([Integer(1),Integer(0),Integer(0)])
>>> p.center()
(1, 0, 0)
```

face_fan()

Return the face fan of a compact rational polyhedron.

OUTPUT:

A fan of the ambient space as a *RationalPolyhedralFan*.

> **See also**
>
> *normal_fan()*.

EXAMPLES:

```
sage: T = polytopes.cuboctahedron()
sage: T.face_fan()
Rational polyhedral fan in 3-d lattice M
```

```
>>> from sage.all import *
>>> T = polytopes.cuboctahedron()
>>> T.face_fan()
Rational polyhedral fan in 3-d lattice M
```

The polytope should contain the origin in the interior:

```
sage: P = Polyhedron(vertices=[[1/2, 1], [1, 1/2]])
sage: P.face_fan()
Traceback (most recent call last):
...
ValueError: face fans are defined only for polytopes
containing the origin as an interior point!

sage: Q = Polyhedron(vertices=[[-1, 1/2], [1, -1/2]])
sage: Q.contains([0,0])
True
sage: FF = Q.face_fan(); FF
Rational polyhedral fan in 2-d lattice M
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[[Integer(1)/Integer(2), Integer(1)], [Integer(1),
↪ Integer(1)/Integer(2)]])
>>> P.face_fan()
Traceback (most recent call last):
...
ValueError: face fans are defined only for polytopes
containing the origin as an interior point!

>>> Q = Polyhedron(vertices=[[-Integer(1), Integer(1)/Integer(2)],␣
↪[Integer(1), -Integer(1)/Integer(2)]])
>>> Q.contains([Integer(0),Integer(0)])
True
>>> FF = Q.face_fan(); FF
Rational polyhedral fan in 2-d lattice M
```

The polytope has to have rational coordinates:

```
sage: S = polytopes.dodecahedron()                                          #␣
↪needs sage.groups sage.rings.number_field
```

(continues on next page)

(continued from previous page)

```
sage: S.face_fan()                                                          #␣
↪needs sage.groups sage.rings.number_field
Traceback (most recent call last):
...
NotImplementedError: face fan handles only polytopes over the rationals
```

```
>>> from sage.all import *
>>> S = polytopes.dodecahedron()                                          #␣
↪needs sage.groups sage.rings.number_field
>>> S.face_fan()                                                          #␣
↪needs sage.groups sage.rings.number_field
Traceback (most recent call last):
...
NotImplementedError: face fan handles only polytopes over the rationals
```

REFERENCES:

For more information, see Chapter 7 of [Zie2007].

hyperplane_arrangement()

Return the hyperplane arrangement defined by the equations and inequalities.

OUTPUT:

A *hyperplane arrangement* consisting of the hyperplanes defined by the *Hrepresentation()*. If the polytope is full-dimensional, this is the hyperplane arrangement spanned by the facets of the polyhedron.

EXAMPLES:

```
sage: p = polytopes.hypercube(2)
sage: p.hyperplane_arrangement()
Arrangement <-t0 + 1 | -t1 + 1 | t1 + 1 | t0 + 1>
```

```
>>> from sage.all import *
>>> p = polytopes.hypercube(Integer(2))
>>> p.hyperplane_arrangement()
Arrangement <-t0 + 1 | -t1 + 1 | t1 + 1 | t0 + 1>
```

is_inscribed(*certificate=False*)

This function tests whether the vertices of the polyhedron are inscribed on a sphere.

The polyhedron is expected to be compact and full-dimensional. A full-dimensional compact polytope is inscribed if there exists a point in space which is equidistant to all its vertices.

ALGORITHM:

The function first computes the circumsphere of a full-dimensional simplex with vertices of `self`. It is found by lifting the points on a paraboloid to find the hyperplane on which the circumsphere is lifted. Then, it checks if all other vertices are equidistant to the circumcenter of that simplex.

INPUT:

- `certificate` – boolean (default: `False`); specifies whether to return the circumcenter, if found

OUTPUT: if `certificate` is true, returns a tuple containing:

1. Boolean.

2. The circumcenter of the polytope or None.

If `certificate` is false:

- a Boolean.

EXAMPLES:

```
sage: q = Polyhedron(vertices=[[1,1,1,1],[-1,-1,1,1],[1,-1,-1,1],
....:                          [-1,1,-1,1],[1,1,1,-1],[-1,-1,1,-1],
....:                          [1,-1,-1,-1],[-1,1,-1,-1],[0,0,10/13,-24/13],
....:                          [0,0,-10/13,-24/13]])
sage: q.is_inscribed(certificate=True)
(True, (0, 0, 0, 0))

sage: cube = polytopes.cube()
sage: cube.is_inscribed()
True

sage: translated_cube = Polyhedron(vertices=[v.vector() + vector([1,2,3])
....:                                        for v in cube.vertices()])
sage: translated_cube.is_inscribed(certificate=True)
(True, (1, 2, 3))

sage: truncated_cube = cube.face_truncation(cube.faces(0)[0])
sage: truncated_cube.is_inscribed()
False
```

```
>>> from sage.all import *
>>> q = Polyhedron(vertices=[[Integer(1),Integer(1),Integer(1),Integer(1)],[-
↪Integer(1),-Integer(1),Integer(1),Integer(1)],[Integer(1),-Integer(1),-
↪Integer(1),Integer(1)],
...                          [-Integer(1),Integer(1),-Integer(1),
↪Integer(1)],[Integer(1),Integer(1),Integer(1),-Integer(1)],[-Integer(1),-
↪Integer(1),Integer(1),-Integer(1)],
...                          [Integer(1),-Integer(1),-Integer(1),-
↪Integer(1)],[-Integer(1),Integer(1),-Integer(1),-Integer(1)],[Integer(0),
↪Integer(0),Integer(10)/Integer(13),-Integer(24)/Integer(13)],
...                          [Integer(0),Integer(0),-Integer(10)/
↪Integer(13),-Integer(24)/Integer(13)]])
>>> q.is_inscribed(certificate=True)
(True, (0, 0, 0, 0))

>>> cube = polytopes.cube()
>>> cube.is_inscribed()
True

>>> translated_cube = Polyhedron(vertices=[v.vector() + vector([Integer(1),
↪Integer(2),Integer(3)])
...                                        for v in cube.vertices()])
>>> translated_cube.is_inscribed(certificate=True)
(True, (1, 2, 3))

>>> truncated_cube = cube.face_truncation(cube.faces(Integer(0))[Integer(0)])
>>> truncated_cube.is_inscribed()
```

(continues on next page)

(continued from previous page)

```
False
```

The method is not implemented for non-full-dimensional polytope or unbounded polyhedra:

```
sage: square = Polyhedron(vertices=[[1,0,0],[0,1,0],[1,1,0],[0,0,0]])
sage: square.is_inscribed()
Traceback (most recent call last):
...
NotImplementedError: this function is implemented for full-dimensional␣
↪polyhedra only

sage: p = Polyhedron(vertices=[(0,0)],rays=[(1,0),(0,1)])
sage: p.is_inscribed()
Traceback (most recent call last):
...
NotImplementedError: this function is not implemented for unbounded polyhedra
```

```
>>> from sage.all import *
>>> square = Polyhedron(vertices=[[Integer(1),Integer(0),Integer(0)],
↪[Integer(0),Integer(1),Integer(0)],[Integer(1),Integer(1),Integer(0)],
↪[Integer(0),Integer(0),Integer(0)]])
>>> square.is_inscribed()
Traceback (most recent call last):
...
NotImplementedError: this function is implemented for full-dimensional␣
↪polyhedra only

>>> p = Polyhedron(vertices=[(Integer(0),Integer(0))],rays=[(Integer(1),
↪Integer(0)),(Integer(0),Integer(1))])
>>> p.is_inscribed()
Traceback (most recent call last):
...
NotImplementedError: this function is not implemented for unbounded polyhedra
```

is_minkowski_summand(Y)

Test whether `Y` is a Minkowski summand.

See `minkowski_sum()`.

OUTPUT: boolean; whether there exists another polyhedron Z such that `self` can be written as $Y \oplus Z$

EXAMPLES:

```
sage: A = polytopes.hypercube(2)
sage: B = Polyhedron(vertices=[(0,1), (1/2,1)])
sage: C = Polyhedron(vertices=[(1,1)])
sage: A.is_minkowski_summand(B)
True
sage: A.is_minkowski_summand(C)
True
sage: B.is_minkowski_summand(C)
True
sage: B.is_minkowski_summand(A)
```

(continues on next page)

(continued from previous page)

```
False
sage: C.is_minkowski_summand(A)
False
sage: C.is_minkowski_summand(B)
False
```

```
>>> from sage.all import *
>>> A = polytopes.hypercube(Integer(2))
>>> B = Polyhedron(vertices=[(Integer(0),Integer(1)), (Integer(1)/Integer(2),
→Integer(1))])
>>> C = Polyhedron(vertices=[(Integer(1),Integer(1))])
>>> A.is_minkowski_summand(B)
True
>>> A.is_minkowski_summand(C)
True
>>> B.is_minkowski_summand(C)
True
>>> B.is_minkowski_summand(A)
False
>>> C.is_minkowski_summand(A)
False
>>> C.is_minkowski_summand(B)
False
```

normal_fan(*direction='inner'*)

Return the normal fan of a compact full-dimensional rational polyhedron.

This returns the inner normal fan of `self`. For the outer normal fan, use `direction='outer'`.

INPUT:

- `direction` – either `'inner'` (default) or `'outer'`; if set to `'inner'`, use the inner normal vectors to span the cones of the fan, if set to `'outer'`, use the outer normal vectors.

OUTPUT:

A complete fan of the ambient space as a *RationalPolyhedralFan*.

See also

face_fan().

EXAMPLES:

```
sage: S = Polyhedron(vertices=[[0, 0], [1, 0], [0, 1]])
sage: S.normal_fan()
Rational polyhedral fan in 2-d lattice N

sage: C = polytopes.hypercube(4)
sage: NF = C.normal_fan(); NF
Rational polyhedral fan in 4-d lattice N
```

```
>>> from sage.all import *
>>> S = Polyhedron(vertices=[[Integer(0), Integer(0)], [Integer(1),␣
↪Integer(0)], [Integer(0), Integer(1)]])
>>> S.normal_fan()
Rational polyhedral fan in 2-d lattice N

>>> C = polytopes.hypercube(Integer(4))
>>> NF = C.normal_fan(); NF
Rational polyhedral fan in 4-d lattice N
```

Currently, it is only possible to get the normal fan of a bounded rational polytope:

```
sage: P = Polyhedron(rays=[[1, 0], [0, 1]])
sage: P.normal_fan()
Traceback (most recent call last):
...
NotImplementedError: the normal fan is only supported for polytopes (compact␣
↪polyhedra).

sage: Q = Polyhedron(vertices=[[1, 0, 0], [0, 1, 0], [0, 0, 1]])
sage: Q.normal_fan()
Traceback (most recent call last):
...
ValueError: the normal fan is only defined for full-dimensional polytopes

sage: R = Polyhedron(vertices=[[0, 0],                                  #␣
↪needs sage.rings.number_field sage.symbolic
....:                          [AA(sqrt(2)), 0],
....:                          [0, AA(sqrt(2))]])
sage: R.normal_fan()                                                    #␣
↪needs sage.rings.number_field sage.symbolic
Traceback (most recent call last):
...
NotImplementedError: normal fan handles only polytopes over the rationals

sage: P = Polyhedron(vertices=[[0,0], [2,0], [0,2], [2,1], [1,2]])
sage: P.normal_fan(direction=None)
Traceback (most recent call last):
...
TypeError: the direction should be 'inner' or 'outer'

sage: inner_nf = P.normal_fan()
sage: inner_nf.rays()
N( 1,  0),
N( 0, -1),
N( 0,  1),
N(-1,  0),
N(-1, -1)
in 2-d lattice N

sage: outer_nf = P.normal_fan(direction='outer')
sage: outer_nf.rays()
N( 1,  0),
```

(continues on next page)

(continued from previous page)

```
N( 1,  1),
N( 0,  1),
N(-1,  0),
N( 0, -1)
in 2-d lattice N
```

```
>>> from sage.all import *
>>> P = Polyhedron(rays=[[Integer(1), Integer(0)], [Integer(0), Integer(1)]])
>>> P.normal_fan()
Traceback (most recent call last):
...
NotImplementedError: the normal fan is only supported for polytopes (compact␣
↪polyhedra).

>>> Q = Polyhedron(vertices=[[Integer(1), Integer(0), Integer(0)],␣
↪[Integer(0), Integer(1), Integer(0)], [Integer(0), Integer(0), Integer(1)]])
>>> Q.normal_fan()
Traceback (most recent call last):
...
ValueError: the normal fan is only defined for full-dimensional polytopes

>>> R = Polyhedron(vertices=[[Integer(0), Integer(0)],                          ␣
↪              # needs sage.rings.number_field sage.symbolic
...                          [AA(sqrt(Integer(2))), Integer(0)],
...                          [Integer(0), AA(sqrt(Integer(2)))]])
>>> R.normal_fan()                                                            #␣
↪needs sage.rings.number_field sage.symbolic
Traceback (most recent call last):
...
NotImplementedError: normal fan handles only polytopes over the rationals

>>> P = Polyhedron(vertices=[[Integer(0),Integer(0)], [Integer(2),Integer(0)],
↪ [Integer(0),Integer(2)], [Integer(2),Integer(1)], [Integer(1),Integer(2)]])
>>> P.normal_fan(direction=None)
Traceback (most recent call last):
...
TypeError: the direction should be 'inner' or 'outer'

>>> inner_nf = P.normal_fan()
>>> inner_nf.rays()
N( 1,  0),
N( 0, -1),
N( 0,  1),
N(-1,  0),
N(-1, -1)
in 2-d lattice N

>>> outer_nf = P.normal_fan(direction='outer')
>>> outer_nf.rays()
N( 1,  0),
N( 1,  1),
N( 0,  1),
```

(continues on next page)

(continued from previous page)

```
N(-1,  0),
N( 0, -1)
in 2-d lattice N
```

REFERENCES:

For more information, see Chapter 7 of [Zie2007].

permutations_to_matrices(*conj_class_reps*, *acting_group=None*, *additional_elts=None*)

Return a dictionary between different representations of elements in the `acting_group`, with group elements represented as permutations of the vertices of this polytope (keys) or matrices (values).

The dictionary has entries for the generators of the `acting_group` and the representatives of conjugacy classes in `conj_class_reps`. By default, the `acting_group` is the *restricted_automorphism_group()* of the polytope. Each element in `additional_elts` also becomes a key.

INPUT:

- `conj_class_reps` – list; a list of representatives of the conjugacy classes of the `acting_group`
- `acting_group` – a subgroup of polytope's *restricted_automorphism_group()*
- `additional_elts` – list (default: `None`); a subset of the *restricted_automorphism_group()* of the polytope expressed as permutations.

OUTPUT:

A dictionary between elements of the `acting_group` expressed as permutations (keys) and matrices (values).

EXAMPLES:

This example shows the dictionary between permutations and matrices for the generators of the `restricted_automorphism_group` of the ± 1 2-dimensional square. The permutations are written in terms of the vertices of the square:

```
sage: # optional - pynormaliz, needs sage.groups
sage: square = Polyhedron(vertices=[[1,1], [-1,1],
....:                               [-1,-1], [1,-1]],
....:                     backend='normaliz')
sage: square.vertices()
(A vertex at (-1, -1),
 A vertex at (-1, 1),
 A vertex at (1, -1),
 A vertex at (1, 1))
sage: aut_square = square.restricted_automorphism_group(output='permutation')
sage: conj_reps = aut_square.conjugacy_classes_representatives()
sage: gens_dict = square.permutations_to_matrices(conj_reps)
sage: rotation_180 = aut_square([(0,3),(1,2)])
sage: rotation_180, gens_dict[rotation_180]
(
             [-1  0  0]
             [ 0 -1  0]
(0,3)(1,2), [ 0  0  1]
)
```

```
>>> from sage.all import *
>>> # optional - pynormaliz, needs sage.groups
```

(continues on next page)

(continued from previous page)

```
>>> square = Polyhedron(vertices=[[Integer(1),Integer(1)], [-Integer(1),
↪Integer(1)],
...                                [-Integer(1),-Integer(1)], [Integer(1),-
↪Integer(1)]],
...                         backend='normaliz')
>>> square.vertices()
(A vertex at (-1, -1),
 A vertex at (-1, 1),
 A vertex at (1, -1),
 A vertex at (1, 1))
>>> aut_square = square.restricted_automorphism_group(output='permutation')
>>> conj_reps = aut_square.conjugacy_classes_representatives()
>>> gens_dict = square.permutations_to_matrices(conj_reps)
>>> rotation_180 = aut_square([(Integer(0),Integer(3)),(Integer(1),
↪Integer(2))])
>>> rotation_180, gens_dict[rotation_180]
(
            [-1  0  0]
            [ 0 -1  0]
(0,3)(1,2), [ 0  0  1]
)
```

This example tests the functionality for additional elements:

```
sage: # needs sage.groups sage.rings.real_mpfr
sage: C = polytopes.cross_polytope(2)
sage: G = C.restricted_automorphism_group(output='permutation')
sage: conj_reps = G.conjugacy_classes_representatives()
sage: add_elt = G([(0, 2, 3, 1)])
sage: dict = C.permutations_to_matrices(conj_reps,
....:                                   additional_elts=[add_elt])
sage: dict[add_elt]
 [ 0  1  0]
 [-1  0  0]
 [ 0  0  1]
```

```
>>> from sage.all import *
>>> # needs sage.groups sage.rings.real_mpfr
>>> C = polytopes.cross_polytope(Integer(2))
>>> G = C.restricted_automorphism_group(output='permutation')
>>> conj_reps = G.conjugacy_classes_representatives()
>>> add_elt = G([(Integer(0), Integer(2), Integer(3), Integer(1))])
>>> dict = C.permutations_to_matrices(conj_reps,
...                                   additional_elts=[add_elt])
>>> dict[add_elt]
 [ 0  1  0]
 [-1  0  0]
 [ 0  0  1]
```

radius()

Return the maximal distance from the center to a vertex. All rays and lines are ignored.

OUTPUT:

The radius for a rational polyhedron is, in general, not rational. use *radius_square()* if you need a rational distance measure.

EXAMPLES:

```
sage: p = polytopes.hypercube(4)
sage: p.radius()
2
```

```
>>> from sage.all import *
>>> p = polytopes.hypercube(Integer(4))
>>> p.radius()
2
```

radius_square()

Return the square of the maximal distance from the *center()* to a vertex. All rays and lines are ignored.

OUTPUT:

The square of the radius, which is in *base_ring()*.

EXAMPLES:

```
sage: p = polytopes.permutahedron(4, project = False)
sage: p.radius_square()
5
```

```
>>> from sage.all import *
>>> p = polytopes.permutahedron(Integer(4), project = False)
>>> p.radius_square()
5
```

to_linear_program(*solver=None*, *return_variable=False*, *base_ring=None*)

Return a linear optimization problem over the polyhedron in the form of a `MixedIntegerLinearProgram`.

INPUT:

- `solver` – select a solver (MIP backend). See the documentation of for `MixedIntegerLinearProgram`. Set to `None` by default.
- `return_variable` – boolean (default: `False`); if `True`, return a tuple `(p, x)`, where `p` is the `MixedIntegerLinearProgram` object and `x` is the vector-valued MIP variable in this problem, indexed from 0. If `False`, only return `p`.
- `base_ring` – select a field over which the linear program should be set up. Use `RDF` to request a fast inexact (floating point) solver even if `self` is exact.

Note that the `MixedIntegerLinearProgram` object will have the null function as an objective to be maximized.

See also

`polyhedron()` – return the polyhedron associated with a `MixedIntegerLinearProgram` object.

EXAMPLES:

Exact rational linear program:

```
sage: p = polytopes.cube()
sage: p.to_linear_program()
Linear Program (no objective, 3 variables, 6 constraints)
sage: lp, x = p.to_linear_program(return_variable=True)
sage: lp.set_objective(2*x[0] + 1*x[1] + 39*x[2])
sage: lp.solve()
42
sage: lp.get_values(x[0], x[1], x[2])
[1, 1, 1]
```

```
>>> from sage.all import *
>>> p = polytopes.cube()
>>> p.to_linear_program()
Linear Program (no objective, 3 variables, 6 constraints)
>>> lp, x = p.to_linear_program(return_variable=True)
>>> lp.set_objective(Integer(2)*x[Integer(0)] + Integer(1)*x[Integer(1)] +␣
↪Integer(39)*x[Integer(2)])
>>> lp.solve()
42
>>> lp.get_values(x[Integer(0)], x[Integer(1)], x[Integer(2)])
[1, 1, 1]
```

Floating-point linear program:

```
sage: lp, x = p.to_linear_program(return_variable=True, base_ring=RDF)
sage: lp.set_objective(2*x[0] + 1*x[1] + 39*x[2])
sage: lp.solve()
42.0
```

```
>>> from sage.all import *
>>> lp, x = p.to_linear_program(return_variable=True, base_ring=RDF)
>>> lp.set_objective(Integer(2)*x[Integer(0)] + Integer(1)*x[Integer(1)] +␣
↪Integer(39)*x[Integer(2)])
>>> lp.solve()
42.0
```

Irrational algebraic linear program over an embedded number field:

```
sage: # needs sage.groups sage.rings.number_field
sage: p = polytopes.icosahedron()
sage: lp, x = p.to_linear_program(return_variable=True)
sage: lp.set_objective(x[0] + x[1] + x[2])
sage: lp.solve()
1/4*sqrt5 + 3/4
```

```
>>> from sage.all import *
>>> # needs sage.groups sage.rings.number_field
>>> p = polytopes.icosahedron()
>>> lp, x = p.to_linear_program(return_variable=True)
>>> lp.set_objective(x[Integer(0)] + x[Integer(1)] + x[Integer(2)])
>>> lp.solve()
1/4*sqrt5 + 3/4
```

Same example with floating point:

```
sage: # needs sage.groups sage.rings.number_field
sage: lp, x = p.to_linear_program(return_variable=True, base_ring=RDF)
sage: lp.set_objective(x[0] + x[1] + x[2])
sage: lp.solve()                                                   # tol 1e-5
1.3090169943749475
```

```
>>> from sage.all import *
>>> # needs sage.groups sage.rings.number_field
>>> lp, x = p.to_linear_program(return_variable=True, base_ring=RDF)
>>> lp.set_objective(x[Integer(0)] + x[Integer(1)] + x[Integer(2)])
>>> lp.solve()                                                   # tol 1e-5
1.3090169943749475
```

Same example with a specific floating point solver:

```
sage: # needs sage.groups sage.rings.number_field
sage: lp, x = p.to_linear_program(return_variable=True, solver='GLPK')
sage: lp.set_objective(x[0] + x[1] + x[2])
sage: lp.solve()                                                   # tol 1e-8
1.3090169943749475
```

```
>>> from sage.all import *
>>> # needs sage.groups sage.rings.number_field
>>> lp, x = p.to_linear_program(return_variable=True, solver='GLPK')
>>> lp.set_objective(x[Integer(0)] + x[Integer(1)] + x[Integer(2)])
>>> lp.solve()                                                   # tol 1e-8
1.3090169943749475
```

Irrational algebraic linear program over AA:

```
sage: # needs sage.groups sage.rings.number_field
sage: p = polytopes.icosahedron(base_ring=AA)
sage: lp, x = p.to_linear_program(return_variable=True)
sage: lp.set_objective(x[0] + x[1] + x[2])
sage: lp.solve()                             # long time
1.309016994374948?
```

```
>>> from sage.all import *
>>> # needs sage.groups sage.rings.number_field
>>> p = polytopes.icosahedron(base_ring=AA)
>>> lp, x = p.to_linear_program(return_variable=True)
>>> lp.set_objective(x[Integer(0)] + x[Integer(1)] + x[Integer(2)])
>>> lp.solve()                             # long time
1.309016994374948?
```

sage.geometry.polyhedron.base.**is_Polyhedron**(X)

Test whether X is a Polyhedron.

INPUT:

- X – anything

OUTPUT: boolean

EXAMPLES:

```
sage: p = polytopes.hypercube(2)
sage: from sage.geometry.polyhedron.base import is_Polyhedron
sage: is_Polyhedron(p)
doctest:warning...
DeprecationWarning: is_Polyhedron is deprecated, use isinstance instead
See https://github.com/sagemath/sage/issues/34307 for details.
True
sage: is_Polyhedron(123456)
False
```

```
>>> from sage.all import *
>>> p = polytopes.hypercube(Integer(2))
>>> from sage.geometry.polyhedron.base import is_Polyhedron
>>> is_Polyhedron(p)
doctest:warning...
DeprecationWarning: is_Polyhedron is deprecated, use isinstance instead
See https://github.com/sagemath/sage/issues/34307 for details.
True
>>> is_Polyhedron(Integer(123456))
False
```

2.6.10 Base class for polyhedra over $\mathbf{Q}$

class sage.geometry.polyhedron.base_QQ.**Polyhedron_QQ**(*parent*, *Vrep*, *Hrep*, *Vrep_minimal=None*, *Hrep_minimal=None*, *pref_rep=None*, *mutable=False*, ***kwds*)

Bases: *Polyhedron_base*

Base class for Polyhedra over $\mathbf{Q}$.

Hstar_function(*acting_group=None*, *output=None*)

Return H^* as a rational function in t with coefficients in the ring of class functions of the `acting_group` of this polytope.

Here, $H^*(t) = \sum_m \chi_{m\text{self}} t^m \det(Id - \rho(t))$. The irreducible characters of `acting_group` form an orthonormal basis for the ring of class functions with values in $\mathbf{C}$. The coefficients of $H^*(t)$ are expressed in this basis.

INPUT:

- `acting_group` – (default=None) a permgroup object. A subgroup of the polytope's `restricted_automorphism_group`. If `None`, it is set to the full `restricted_automorphism_group` of the polytope. The acting group should always use `output='permutation'`.
- `output` – string. an output option. The allowed values are:
  - `None` (default): returns the rational function $H^*(t)$. H^* is a rational function in t with coefficients in the ring of class functions.
  - `'e_series_list'`: Returns a list of the ehrhart_series for the fixed_subpolytopes of each conjugacy class representative.
  - `'determinant_vec'`: Returns a list of the determinants of $Id - \rho * t$ for each conjugacy class representative.

– `'Hstar_as_lin_comb'`: Returns a vector of the coefficients of the irreducible representations in the expression of H^*.
– `'prod_det_es'`: Returns a vector of the product of determinants and the Ehrhart series.
– `'complete'`: Returns a list with `Hstar`, `Hstar_as_lin_comb`, character table of the acting group, and whether `Hstar` is effective.

OUTPUT:

The default output is the rational function H^*. H^* is a rational function in t with coefficients in the ring of class functions. There are several output options to see the intermediary outputs of the function.

EXAMPLES:

The H^*-polynomial of the standard $(d-1)$-dimensional simplex $S = conv(e_1, \ldots, e_d)$ under its `restricted_automorphism_group()` is equal to 1 = $\chi_{trivial}$ (Prop 6.1 [Stap2011]). Here is the computation for the 3-dimensional standard simplex:

```
sage: # optional - pynormaliz
sage: S = polytopes.simplex(3, backend='normaliz'); S
A 3-dimensional polyhedron in ZZ^4 defined as the convex hull of 4 vertices
sage: G = S.restricted_automorphism_group(
....:         output='permutation')
sage: G.is_isomorphic(SymmetricGroup(4))
True
sage: Hstar = S._Hstar_function_normaliz(G); Hstar
chi_4
sage: G.character_table()
[ 1 -1  1  1 -1]
[ 3 -1  0 -1  1]
[ 2  0 -1  2  0]
[ 3  1  0 -1 -1]
[ 1  1  1  1  1]
```

```
>>> from sage.all import *
>>> # optional - pynormaliz
>>> S = polytopes.simplex(Integer(3), backend='normaliz'); S
A 3-dimensional polyhedron in ZZ^4 defined as the convex hull of 4 vertices
>>> G = S.restricted_automorphism_group(
...         output='permutation')
>>> G.is_isomorphic(SymmetricGroup(Integer(4)))
True
>>> Hstar = S._Hstar_function_normaliz(G); Hstar
chi_4
>>> G.character_table()
[ 1 -1  1  1 -1]
[ 3 -1  0 -1  1]
[ 2  0 -1  2  0]
[ 3  1  0 -1 -1]
[ 1  1  1  1  1]
```

The next example is Example 7.6 in [Stap2011], and shows that H^* is not always a polynomial. Let P be the polytope with vertices $\pm(0,0,1), \pm(1,0,1), \pm(0,1,1), \pm(1,1,1)$ and let G = $\mathbf{Z}/2\mathbf{Z}$ act on P as follows:

```
sage: # optional - pynormaliz
sage: P = Polyhedron(vertices=[[0,0,1], [0,0,-1], [1,0,1],
....:                          [-1,0,-1], [0,1,1],
....:                          [0,-1,-1], [1,1,1], [-1,-1,-1]],
....:                backend='normaliz')
sage: K = P.restricted_automorphism_group(
....:         output='permutation')
sage: G = K.subgroup(gens=[K([(0,2),(1,3),(4,6),(5,7)])])
sage: conj_reps = G.conjugacy_classes_representatives()
sage: Dict = P.permutations_to_matrices(conj_reps,
....:                                   acting_group=G)
sage: list(Dict.keys())[0]
(0,2)(1,3)(4,6)(5,7)
sage: list(Dict.values())[0]
[-1  0  1  0]
[ 0  1  0  0]
[ 0  0  1  0]
[ 0  0  0  1]
sage: len(G)
2
sage: G.character_table()
[ 1  1]
[ 1 -1]
```

```
>>> from sage.all import *
>>> # optional - pynormaliz
>>> P = Polyhedron(vertices=[[Integer(0),Integer(0),Integer(1)], [Integer(0),
↪Integer(0),-Integer(1)], [Integer(1),Integer(0),Integer(1)],
...                          [-Integer(1),Integer(0),-Integer(1)],␣
↪[Integer(0),Integer(1),Integer(1)],
...                          [Integer(0),-Integer(1),-Integer(1)],␣
↪[Integer(1),Integer(1),Integer(1)], [-Integer(1),-Integer(1),-Integer(1)]],
...                backend='normaliz')
>>> K = P.restricted_automorphism_group(
...         output='permutation')
>>> G = K.subgroup(gens=[K([(Integer(0),Integer(2)),(Integer(1),Integer(3)),
↪(Integer(4),Integer(6)),(Integer(5),Integer(7))])])
>>> conj_reps = G.conjugacy_classes_representatives()
>>> Dict = P.permutations_to_matrices(conj_reps,
...                                   acting_group=G)
>>> list(Dict.keys())[Integer(0)]
(0,2)(1,3)(4,6)(5,7)
>>> list(Dict.values())[Integer(0)]
[-1  0  1  0]
[ 0  1  0  0]
[ 0  0  1  0]
[ 0  0  0  1]
>>> len(G)
2
>>> G.character_table()
[ 1  1]
[ 1 -1]
```

Then we calculate the rational function $H^*(t)$:

```
sage: Hst = P._Hstar_function_normaliz(G); Hst              # optional -␣
↪pynormaliz
(chi_0*t^4 + (3*chi_0 + 3*chi_1)*t^3
 + (8*chi_0 + 2*chi_1)*t^2 + (3*chi_0 + 3*chi_1)*t + chi_0)/(t + 1)
```

```
>>> from sage.all import *
>>> Hst = P._Hstar_function_normaliz(G); Hst              # optional -␣
↪pynormaliz
(chi_0*t^4 + (3*chi_0 + 3*chi_1)*t^3
 + (8*chi_0 + 2*chi_1)*t^2 + (3*chi_0 + 3*chi_1)*t + chi_0)/(t + 1)
```

To see the exact as written in [Stap2011], we can format it as `'Hstar_as_lin_comb'`. The first coordinate is the coefficient of the trivial character; the second is the coefficient of the sign character:

```
sage: lin = P._Hstar_function_normaliz(                     # optional -␣
↪pynormaliz
....:             G, output='Hstar_as_lin_comb'); lin
((t^4 + 3*t^3 + 8*t^2 + 3*t + 1)/(t + 1),
 (3*t^3 + 2*t^2 + 3*t)/(t + 1))
```

```
>>> from sage.all import *
>>> lin = P._Hstar_function_normaliz(                     # optional -␣
↪pynormaliz
...             G, output='Hstar_as_lin_comb'); lin
((t^4 + 3*t^3 + 8*t^2 + 3*t + 1)/(t + 1),
 (3*t^3 + 2*t^2 + 3*t)/(t + 1))
```

ehrhart_polynomial(*engine=None*, *variable='t'*, *verbose=False*, *dual=None*, *irrational_primal=None*, *irrational_all_primal=None*, *maxdet=None*, *no_decomposition=None*, *compute_vertex_cones=None*, *smith_form=None*, *dualization=None*, *triangulation=None*, *triangulation_max_height=None*, ***kwds*)

Return the Ehrhart polynomial of this polyhedron.

The polyhedron must be a lattice polytope. Let P be a lattice polytope in $\mathbf{R}^d$ and define $L(P,t) = \#(tP\cap \mathbf{Z}^d)$. Then E. Ehrhart proved in 1962 that L coincides with a rational polynomial of degree d for integer t. L is called the *Ehrhart polynomial* of P. For more information see the Wikipedia article Ehrhart_polynomial. The Ehrhart polynomial may be computed using either LattE Integrale or Normaliz by setting `engine` to 'latte' or 'normaliz' respectively.

INPUT:

- `engine` – string; the backend to use. Allowed values are:
  - `None` (default); When no input is given the Ehrhart polynomial is computed using LattE Integrale (optional)
  - `'latte'`; use LattE integrale program (optional)
  - `'normaliz'`; use Normaliz program (optional package pynormaliz). The backend of `self` must be set to `'normaliz'`.
- `variable` – string (default: `'t'`); the variable in which the Ehrhart polynomial should be expressed
- When the `engine` is `'latte'`, the additional input values are:
  - `verbose` – boolean (default: `False`); if `True`, print the whole output of the LattE command

The following options are passed to the LattE command, for details consult the LattE documentation:

- `dual` – boolean; triangulate and signed-decompose in the dual space
- `irrational_primal` – boolean; triangulate in the dual space, signed-decompose in the primal space using irrationalization.
- `irrational_all_primal` – boolean; triangulate and signed-decompose in the primal space using irrationalization.
- `maxdet` – integer; decompose down to an index (determinant) of `maxdet` instead of index 1 (unimodular cones).
- `no_decomposition` – boolean; do not signed-decompose simplicial cones.
- `compute_vertex_cones` – string; either 'cdd' or 'lrs' or '4ti2'
- `smith_form` – string; either 'ilio' or 'lidia'
- `dualization` – string; either 'cdd' or '4ti2'
- `triangulation` – string; 'cddlib', '4ti2' or 'topcom'
- `triangulation_max_height` – integer; use a uniform distribution of height from 1 to this number

OUTPUT: a univariate polynomial in `variable` over a rational field

See also:

`latte` the interface to LattE Integrale PyNormaliz

EXAMPLES:

To start, we find the Ehrhart polynomial of a three-dimensional `simplex`, first using `engine='latte'`. Leaving the engine unspecified sets the `engine` to `'latte'` by default:

```
sage: simplex = Polyhedron(vertices=[(0,0,0),(3,3,3),(-3,2,1),(1,-1,-2)])
sage: simplex = simplex.change_ring(QQ)
sage: poly = simplex.ehrhart_polynomial(engine='latte')      # optional -
↪latte_int
sage: poly                                                     # optional -
↪latte_int
7/2*t^3 + 2*t^2 - 1/2*t + 1
sage: poly(1)                                                  # optional -
↪latte_int
6
sage: len(simplex.integral_points())
6
sage: poly(2)                                                  # optional -
↪latte_int
36
sage: len((2*simplex).integral_points())
36
```

```
>>> from sage.all import *
>>> simplex = Polyhedron(vertices=[(Integer(0),Integer(0),Integer(0)),
↪(Integer(3),Integer(3),Integer(3)),(-Integer(3),Integer(2),Integer(1)),
```

(continues on next page)

(continued from previous page)

```
↪(Integer(1),-Integer(1),-Integer(2))])
>>> simplex = simplex.change_ring(QQ)
>>> poly = simplex.ehrhart_polynomial(engine='latte')      # optional - latte_
↪int
>>> poly                                                    # optional - latte_
↪int
7/2*t^3 + 2*t^2 - 1/2*t + 1
>>> poly(Integer(1))                                                  # optional␣
↪- latte_int
6
>>> len(simplex.integral_points())
6
>>> poly(Integer(2))                                                  # optional␣
↪- latte_int
36
>>> len((Integer(2)*simplex).integral_points())
36
```

Now we find the same Ehrhart polynomial, this time using `engine='normaliz'`. To use the Normaliz engine, the `simplex` must be defined with `backend='normaliz'`:

```
sage: # optional - pynormaliz
sage: simplex = Polyhedron(vertices=[(0,0,0), (3,3,3),
....:                                 (-3,2,1), (1,-1,-2)],
....:                      backend='normaliz')
sage: simplex = simplex.change_ring(QQ)
sage: poly = simplex.ehrhart_polynomial(engine='normaliz')
sage: poly
7/2*t^3 + 2*t^2 - 1/2*t + 1
```

```
>>> from sage.all import *
>>> # optional - pynormaliz
>>> simplex = Polyhedron(vertices=[(Integer(0),Integer(0),Integer(0)),␣
↪(Integer(3),Integer(3),Integer(3)),
...                                (-Integer(3),Integer(2),Integer(1)),␣
↪(Integer(1),-Integer(1),-Integer(2))],
...                      backend='normaliz')
>>> simplex = simplex.change_ring(QQ)
>>> poly = simplex.ehrhart_polynomial(engine='normaliz')
>>> poly
7/2*t^3 + 2*t^2 - 1/2*t + 1
```

If the `engine='normaliz'`, the backend should be `'normaliz'`, otherwise it returns an error:

```
sage: simplex = Polyhedron(vertices=[(0,0,0), (3,3,3),
....:                                 (-3,2,1), (1,-1,-2)])
sage: simplex = simplex.change_ring(QQ)
sage: simplex.ehrhart_polynomial(engine='normaliz')
Traceback (most recent call last):
...
TypeError: The backend of the polyhedron should be 'normaliz'
```

```
>>> from sage.all import *
>>> simplex = Polyhedron(vertices=[(Integer(0),Integer(0),Integer(0)),␣
↪(Integer(3),Integer(3),Integer(3)),
...                                  (-Integer(3),Integer(2),Integer(1)),␣
↪(Integer(1),-Integer(1),-Integer(2))])
>>> simplex = simplex.change_ring(QQ)
>>> simplex.ehrhart_polynomial(engine='normaliz')
Traceback (most recent call last):
...
TypeError: The backend of the polyhedron should be 'normaliz'
```

The polyhedron should be compact:

```
sage: C = Polyhedron(rays=[[1,2], [2,1]],                   # optional -␣
↪pynormaliz
....:                backend='normaliz')
sage: C = C.change_ring(QQ)                                  # optional -␣
↪pynormaliz
sage: C.ehrhart_polynomial()                                 # optional -␣
↪pynormaliz
Traceback (most recent call last):
...
ValueError: Ehrhart polynomial only defined for compact polyhedra
```

```
>>> from sage.all import *
>>> C = Polyhedron(rays=[[Integer(1),Integer(2)], [Integer(2),Integer(1)]],  ␣
↪                 # optional - pynormaliz
...                backend='normaliz')
>>> C = C.change_ring(QQ)                                    # optional -␣
↪pynormaliz
>>> C.ehrhart_polynomial()                                   # optional -␣
↪pynormaliz
Traceback (most recent call last):
...
ValueError: Ehrhart polynomial only defined for compact polyhedra
```

The polyhedron should have integral vertices:

```
sage: L = Polyhedron(vertices=[[0], [1/2]])
sage: L.ehrhart_polynomial()
Traceback (most recent call last):
...
TypeError: the polytope has nonintegral vertices, use ehrhart_quasipolynomial␣
↪with backend 'normaliz'
```

```
>>> from sage.all import *
>>> L = Polyhedron(vertices=[[Integer(0)], [Integer(1)/Integer(2)]])
>>> L.ehrhart_polynomial()
Traceback (most recent call last):
...
TypeError: the polytope has nonintegral vertices, use ehrhart_quasipolynomial␣
↪with backend 'normaliz'
```

ehrhart_quasipolynomial(*variable='t'*, *engine=None*, *verbose=False*, *dual=None*, *irrational_primal=None*, *irrational_all_primal=None*, *maxdet=None*, *no_decomposition=None*, *compute_vertex_cones=None*, *smith_form=None*, *dualization=None*, *triangulation=None*, *triangulation_max_height=None*, ***kwds*)

Compute the Ehrhart quasipolynomial of this polyhedron with rational vertices.

If the polyhedron is a lattice polytope, returns the Ehrhart polynomial, a univariate polynomial in `variable` over a rational field. If the polyhedron has rational, nonintegral vertices, returns a tuple of polynomials in `variable` over a rational field. The Ehrhart counting function of a polytope P with rational vertices is given by a *quasipolynomial*. That is, there exists a positive integer l and l polynomials $ehr_{P,i}$ for $i \in \{1, \dots, l\}$ such that if t is equivalent to i mod l then $tP \cap \mathbb{Z}^d = ehr_{P,i}(t)$.

INPUT:

- `variable` – string (default: `'t'`); the variable in which the Ehrhart polynomial should be expressed
- `engine` – string; the backend to use. Allowed values are:
  - `None` (default); When no input is given the Ehrhart polynomial is computed using Normaliz (optional)
  - `'latte'`; use LattE Integrale program (requires optional package 'latte_int')
  - `'normaliz'`; use the Normaliz program (requires optional package 'pynormaliz'). The backend of `self` must be set to 'normaliz'.
- When the `engine` is 'latte', the additional input values are:
  - `verbose` – boolean (default: `False`); if `True`, print the whole output of the LattE command

  The following options are passed to the LattE command, for details consult the LattE documentation:
  - `dual` – boolean; triangulate and signed-decompose in the dual space
  - `irrational_primal` – boolean; triangulate in the dual space, signed-decompose in the primal space using irrationalization.
  - `irrational_all_primal` – boolean; triangulate and signed-decompose in the primal space using irrationalization.
  - `maxdet` – integer; decompose down to an index (determinant) of `maxdet` instead of index 1 (unimodular cones).
  - `no_decomposition` – boolean; do not signed-decompose simplicial cones.
  - `compute_vertex_cones` – string; either `'cdd'` or `'lrs'` or `'4ti2'`
  - `smith_form` – string; either `'ilio'` or `'lidia'`
  - `dualization` – string; either `'cdd'` or `'4ti2'`
  - `triangulation` – string; `'cddlib'`, `'4ti2'` or `'topcom'`
  - `triangulation_max_height` – integer; use a uniform distribution of height from 1 to this number

OUTPUT:

A univariate polynomial over a rational field or a tuple of such polynomials.

See also

`latte` the interface to LattE Integrale PyNormaliz

> **Warning**
>
> If the polytope has rational, non integral vertices, it must have `backend='normaliz'`.

EXAMPLES:

As a first example, consider the line segment [0,1/2]. If we dilate this line segment by an even integral factor k, then the dilated line segment will contain $k/2+1$ lattice points. If k is odd then there will be $k/2+1/2$ lattice points in the dilated line segment. Note that it is necessary to set the backend of the polytope to 'normaliz':

```
sage: line_seg = Polyhedron(vertices=[[0], [1/2]],          # optional -␣
↪pynormaliz
....:                              backend='normaliz'); line_seg
A 1-dimensional polyhedron in QQ^1 defined as the convex hull of 2 vertices
sage: line_seg.ehrhart_quasipolynomial()                    # optional -␣
↪pynormaliz
(1/2*t + 1, 1/2*t + 1/2)
```

```
>>> from sage.all import *
>>> line_seg = Polyhedron(vertices=[[Integer(0)], [Integer(1)/Integer(2)]],  ␣
↪       # optional - pynormaliz
...                            backend='normaliz'); line_seg
A 1-dimensional polyhedron in QQ^1 defined as the convex hull of 2 vertices
>>> line_seg.ehrhart_quasipolynomial()                      # optional -␣
↪pynormaliz
(1/2*t + 1, 1/2*t + 1/2)
```

For a more exciting example, let us look at the subpolytope of the 3 dimensional permutahedron fixed by the reflection across the hyperplane $x_1 = x_4$:

```
sage: verts = [[3/2, 3, 4, 3/2],
....:          [3/2, 4, 3, 3/2],
....:          [5/2, 1, 4, 5/2],
....:          [5/2, 4, 1, 5/2],
....:          [7/2, 1, 2, 7/2],
....:          [7/2, 2, 1, 7/2]]
sage: subpoly = Polyhedron(vertices=verts,                  # optional -␣
↪pynormaliz
....:                             backend='normaliz')
sage: eq = subpoly.ehrhart_quasipolynomial(); eq            # optional -␣
↪pynormaliz
(4*t^2 + 3*t + 1, 4*t^2 + 2*t)
sage: eq = subpoly.ehrhart_quasipolynomial(); eq            # optional -␣
↪pynormaliz
(4*t^2 + 3*t + 1, 4*t^2 + 2*t)
sage: even_ep = eq[0]                                       # optional -␣
↪pynormaliz
sage: odd_ep  = eq[1]                                       # optional -␣
↪pynormaliz
sage: even_ep(2)                                            # optional -␣
↪pynormaliz
23
```

(continues on next page)

(continued from previous page)

```
sage: ts = 2*subpoly                                          # optional -␣
↪pynormaliz
sage: ts.integral_points_count()                              # optional -␣
↪pynormaliz latte_int
23
sage: odd_ep(1)                                               # optional -␣
↪pynormaliz
6
sage: subpoly.integral_points_count()                         # optional -␣
↪pynormaliz latte_int
6
```

```
>>> from sage.all import *
>>> verts = [[Integer(3)/Integer(2), Integer(3), Integer(4), Integer(3)/
↪Integer(2)],
...          [Integer(3)/Integer(2), Integer(4), Integer(3), Integer(3)/
↪Integer(2)],
...          [Integer(5)/Integer(2), Integer(1), Integer(4), Integer(5)/
↪Integer(2)],
...          [Integer(5)/Integer(2), Integer(4), Integer(1), Integer(5)/
↪Integer(2)],
...          [Integer(7)/Integer(2), Integer(1), Integer(2), Integer(7)/
↪Integer(2)],
...          [Integer(7)/Integer(2), Integer(2), Integer(1), Integer(7)/
↪Integer(2)]]
>>> subpoly = Polyhedron(vertices=verts,                      # optional -␣
↪pynormaliz
...                      backend='normaliz')
>>> eq = subpoly.ehrhart_quasipolynomial(); eq                # optional -␣
↪pynormaliz
(4*t^2 + 3*t + 1, 4*t^2 + 2*t)
>>> eq = subpoly.ehrhart_quasipolynomial(); eq                # optional -␣
↪pynormaliz
(4*t^2 + 3*t + 1, 4*t^2 + 2*t)
>>> even_ep = eq[Integer(0)]                                         # optional␣
↪- pynormaliz
>>> odd_ep  = eq[Integer(1)]                                         # optional␣
↪- pynormaliz
>>> even_ep(Integer(2))                                              # optional␣
↪- pynormaliz
23
>>> ts = Integer(2)*subpoly                                          # optional␣
↪- pynormaliz
>>> ts.integral_points_count()                                # optional -␣
↪pynormaliz latte_int
23
>>> odd_ep(Integer(1))                                               # optional␣
↪- pynormaliz
6
>>> subpoly.integral_points_count()                           # optional -␣
↪pynormaliz latte_int
6
```

A polytope with rational nonintegral vertices must have `backend='normaliz'`:

```
sage: line_seg = Polyhedron(vertices=[[0], [1/2]])
sage: line_seg.ehrhart_quasipolynomial()
Traceback (most recent call last):
...
TypeError: The backend of the polyhedron should be 'normaliz'
```

```
>>> from sage.all import *
>>> line_seg = Polyhedron(vertices=[[Integer(0)], [Integer(1)/Integer(2)]])
>>> line_seg.ehrhart_quasipolynomial()
Traceback (most recent call last):
...
TypeError: The backend of the polyhedron should be 'normaliz'
```

The polyhedron should be compact:

```
sage: C = Polyhedron(rays=[[1/2,2], [2,1]],                     # optional -␣
↪pynormaliz
....:                backend='normaliz')
sage: C.ehrhart_quasipolynomial()                               # optional -␣
↪pynormaliz
Traceback (most recent call last):
...
ValueError: Ehrhart quasipolynomial only defined for compact polyhedra
```

```
>>> from sage.all import *
>>> C = Polyhedron(rays=[[Integer(1)/Integer(2),Integer(2)], [Integer(2),
↪Integer(1)]],                     # optional - pynormaliz
...                backend='normaliz')
>>> C.ehrhart_quasipolynomial()                               # optional -␣
↪pynormaliz
Traceback (most recent call last):
...
ValueError: Ehrhart quasipolynomial only defined for compact polyhedra
```

If the polytope happens to be a lattice polytope, the Ehrhart polynomial is returned:

```
sage: # optional - pynormaliz
sage: simplex = Polyhedron(vertices=[(0,0,0), (3,3,3),
....:                                 (-3,2,1), (1,-1,-2)],
....:                      backend='normaliz')
sage: simplex = simplex.change_ring(QQ)
sage: poly = simplex.ehrhart_quasipolynomial(
....:     engine='normaliz'); poly
7/2*t^3 + 2*t^2 - 1/2*t + 1
sage: simplex.ehrhart_polynomial()                              # optional -␣
↪latte_int
7/2*t^3 + 2*t^2 - 1/2*t + 1
```

```
>>> from sage.all import *
>>> # optional - pynormaliz
>>> simplex = Polyhedron(vertices=[(Integer(0),Integer(0),Integer(0)),␣
```

(continues on next page)

(continued from previous page)

```
↪(Integer(3),Integer(3),Integer(3)),
...                                   (-Integer(3),Integer(2),Integer(1)),␣
↪(Integer(1),-Integer(1),-Integer(2))],
...                         backend='normaliz')
>>> simplex = simplex.change_ring(QQ)
>>> poly = simplex.ehrhart_quasipolynomial(
...     engine='normaliz'); poly
7/2*t^3 + 2*t^2 - 1/2*t + 1
>>> simplex.ehrhart_polynomial()                          # optional - latte_
↪int
7/2*t^3 + 2*t^2 - 1/2*t + 1
```

fixed_subpolytope(*vertex_permutation*)

Return the fixed subpolytope of this polytope by the cyclic action of `vertex_permutation`.

The fixed subpolytope of this polytope under the `vertex_permutation` is the subset of this polytope that is fixed pointwise.

INPUT:

- `vertex_permutation` – permutation; a permutation of the vertices of `self`

OUTPUT: a subpolytope of `self`

Note

The vertex_permutation is obtained as a permutation of the vertices represented as a permutation. For example, vertex_permutation = self.restricted_automorphism_group(output='permutation').

Requiring a lattice polytope as opposed to a rational polytope as input is purely conventional.

EXAMPLES:

The fixed subpolytopes of the cube can be obtained as follows:

```
sage: Cube = polytopes.cube(backend = 'normaliz')            # optional -␣
↪pynormaliz
sage: AG = Cube.restricted_automorphism_group(               # optional -␣
↪pynormaliz
....:           output='permutation')
sage: reprs = AG.conjugacy_classes_representatives()         # optional -␣
↪pynormaliz
```

```
>>> from sage.all import *
>>> Cube = polytopes.cube(backend = 'normaliz')            # optional -␣
↪pynormaliz
>>> AG = Cube.restricted_automorphism_group(               # optional -␣
↪pynormaliz
...           output='permutation')
>>> reprs = AG.conjugacy_classes_representatives()         # optional -␣
↪pynormaliz
```

The fixed subpolytope of the identity element of the group is the entire cube:

```
sage: reprs[0]                                              # optional -␣
↪pynormaliz
()
sage: Cube.fixed_subpolytope(vertex_permutation=reprs[0])   # optional -␣
↪pynormaliz
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 8
vertices
sage: _.vertices()                                          # optional -␣
↪pynormaliz
(A vertex at (-1, -1, -1),
 A vertex at (-1, -1, 1),
 A vertex at (-1, 1, -1),
 A vertex at (-1, 1, 1),
 A vertex at (1, -1, -1),
 A vertex at (1, -1, 1),
 A vertex at (1, 1, -1),
 A vertex at (1, 1, 1))
```

```
>>> from sage.all import *
>>> reprs[Integer(0)]                                              # optional␣
↪- pynormaliz
()
>>> Cube.fixed_subpolytope(vertex_permutation=reprs[Integer(0)])   # optional␣
↪- pynormaliz
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 8
vertices
>>> _.vertices()                                          # optional -␣
↪pynormaliz
(A vertex at (-1, -1, -1),
 A vertex at (-1, -1, 1),
 A vertex at (-1, 1, -1),
 A vertex at (-1, 1, 1),
 A vertex at (1, -1, -1),
 A vertex at (1, -1, 1),
 A vertex at (1, 1, -1),
 A vertex at (1, 1, 1))
```

You can obtain non-trivial examples:

```
sage: G = AG([(0,1),(2,3),(4,5),(6,7)])                     # optional -␣
↪pynormaliz
sage: fsp = Cube.fixed_subpolytope(G); fsp                  # optional -␣
↪pynormaliz
A 2-dimensional polyhedron in QQ^3 defined as the convex hull of 4 vertices
sage: fsp.vertices()                                        # optional -␣
↪pynormaliz
(A vertex at (-1, -1, 0),
 A vertex at (-1, 1, 0),
 A vertex at (1, -1, 0),
 A vertex at (1, 1, 0))
```

```
>>> from sage.all import *
```

(continues on next page)

(continued from previous page)

```
>>> G = AG([(Integer(0),Integer(1)),(Integer(2),Integer(3)),(Integer(4),
↪Integer(5)),(Integer(6),Integer(7))])                    # optional -
↪pynormaliz
>>> fsp = Cube.fixed_subpolytope(G); fsp                  # optional -
↪pynormaliz
A 2-dimensional polyhedron in QQ^3 defined as the convex hull of 4 vertices
>>> fsp.vertices()                                        # optional -
↪pynormaliz
(A vertex at (-1, -1, 0),
 A vertex at (-1, 1, 0),
 A vertex at (1, -1, 0),
 A vertex at (1, 1, 0))
```

The next example shows that `fixed_subpolytope()` works for rational polytopes:

```
sage: # optional - pynormaliz
sage: P = Polyhedron(vertices=[[0], [1/2]],
....:                backend='normaliz')
sage: P.vertices()
(A vertex at (0), A vertex at (1/2))
sage: G = P.restricted_automorphism_group(
....:         output='permutation'); G
Permutation Group with generators [(0,1)]
sage: len(G)
2
sage: fixed_set = P.fixed_subpolytope(G.gens()[0])
sage: fixed_set
A 0-dimensional polyhedron in QQ^1 defined as the convex hull of 1 vertex
sage: fixed_set.vertices_list()
[[1/4]]
```

```
>>> from sage.all import *
>>> # optional - pynormaliz
>>> P = Polyhedron(vertices=[[Integer(0)], [Integer(1)/Integer(2)]],
...                backend='normaliz')
>>> P.vertices()
(A vertex at (0), A vertex at (1/2))
>>> G = P.restricted_automorphism_group(
...         output='permutation'); G
Permutation Group with generators [(0,1)]
>>> len(G)
2
>>> fixed_set = P.fixed_subpolytope(G.gens()[Integer(0)])
>>> fixed_set
A 0-dimensional polyhedron in QQ^1 defined as the convex hull of 1 vertex
>>> fixed_set.vertices_list()
[[1/4]]
```

fixed_subpolytopes(*conj_class_reps*)

Return the fixed subpolytopes of this polytope under the actions of the given conjugacy class representatives.

The `conj_class_reps` are representatives of the conjugacy classes of a subgroup of the automorphism group of this polytope. For an element of the automorphism group, the fixed subpolytope is the subset of this polytope that is fixed pointwise.

INPUT:

- `conj_class_reps` – list of representatives of the conjugacy classes of the subgroup of the `restricted_automorphism_group()` of the polytope. Each element is written as a permutation of the vertices of the polytope.

OUTPUT:

A dictionary where the elements of `conj_class_reps` are keys and the fixed subpolytopes are values.

> **Note**
>
> Two elements in the same conjugacy class fix lattice-isomorphic subpolytopes.

EXAMPLES:

Here is an example for the square:

```
sage: # optional - pynormaliz, needs sage.groups
sage: p = polytopes.hypercube(2, backend='normaliz'); p
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices
sage: aut_p = p.restricted_automorphism_group(
....:                   output='permutation')
sage: aut_p.order()
8
sage: conj_list = aut_p.conjugacy_classes_representatives()
sage: fixedpolytopes_dict = p.fixed_subpolytopes(conj_list)
sage: fixedpolytopes_dict[aut_p([(0,3),(1,2)])]
A 0-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex
```

```
>>> from sage.all import *
>>> # optional - pynormaliz, needs sage.groups
>>> p = polytopes.hypercube(Integer(2), backend='normaliz'); p
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices
>>> aut_p = p.restricted_automorphism_group(
...                 output='permutation')
>>> aut_p.order()
8
>>> conj_list = aut_p.conjugacy_classes_representatives()
>>> fixedpolytopes_dict = p.fixed_subpolytopes(conj_list)
>>> fixedpolytopes_dict[aut_p([(Integer(0),Integer(3)),(Integer(1),
↪Integer(2))])]
A 0-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex
```

integral_points_count(*verbose=False*, *use_Hrepresentation=False*, *explicit_enumeration_threshold=1000*, *preprocess=True*, ***kwds*)

Return the number of integral points in the polyhedron.

This method uses the optional package `latte_int` if an estimate for lattice points based on bounding boxes exceeds `explicit_enumeration_threshold`.

INPUT:

- `verbose` – boolean (default: `False`); whether to display verbose output

- `use_Hrepresentation` – boolean (default: `False`); whether to send the H or V representation to LattE
- `preprocess` – boolean (default: `True`); whether, if the integral hull is known to lie in a coordinate hyperplane, to tighten bounds to reduce dimension

See also

`latte` the interface to LattE interfaces

EXAMPLES:

```
sage: P = polytopes.cube()
sage: P.integral_points_count()
27
sage: P.integral_points_count(explicit_enumeration_threshold=0)   # optional -␣
↪latte_int
27
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> P.integral_points_count()
27
>>> P.integral_points_count(explicit_enumeration_threshold=Integer(0))   #␣
↪optional - latte_int
27
```

We enlarge the polyhedron to force the use of the generating function methods implemented in LattE integrale, rather than explicit enumeration:

```
sage: (1000000000*P).integral_points_count(verbose=True)          # optional -␣
↪latte_int
This is LattE integrale...
...
Total time:...
8000000012000000006000000001
```

```
>>> from sage.all import *
>>> (Integer(1000000000)*P).integral_points_count(verbose=True)          #␣
↪optional - latte_int
This is LattE integrale...
...
Total time:...
8000000012000000006000000001
```

We shrink the polyhedron a little bit:

```
sage: Q = P*(8/9)
sage: Q.integral_points_count()
1
sage: Q.integral_points_count(explicit_enumeration_threshold=0)
1
```

```
>>> from sage.all import *
>>> Q = P*(Integer(8)/Integer(9))
>>> Q.integral_points_count()
1
>>> Q.integral_points_count(explicit_enumeration_threshold=Integer(0))
1
```

Unbounded polyhedra (with or without lattice points) are not supported:

```
sage: P = Polyhedron(vertices=[[1/2, 1/3]], rays=[[1, 1]])
sage: P.integral_points_count()
Traceback (most recent call last):
...
NotImplementedError: ...
sage: P = Polyhedron(vertices=[[1, 1]], rays=[[1, 1]])
sage: P.integral_points_count()
Traceback (most recent call last):
...
NotImplementedError: ...
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[[Integer(1)/Integer(2), Integer(1)/Integer(3)]],␣
↪rays=[[Integer(1), Integer(1)]])
>>> P.integral_points_count()
Traceback (most recent call last):
...
NotImplementedError: ...
>>> P = Polyhedron(vertices=[[Integer(1), Integer(1)]], rays=[[Integer(1),␣
↪Integer(1)]])
>>> P.integral_points_count()
Traceback (most recent call last):
...
NotImplementedError: ...
```

"Fibonacci" knapsacks (preprocessing helps a lot):

```
sage: def fibonacci_knapsack(d, b, backend=None):
....:     lp = MixedIntegerLinearProgram(base_ring=QQ)
....:     x = lp.new_variable(nonnegative=True)
....:     lp.add_constraint(lp.sum(fibonacci(i+3)*x[i] for i in range(d)) <=␣
↪b)
....:     return lp.polyhedron(backend=backend)
sage: fibonacci_knapsack(20, 12).integral_points_count()  # does not finish␣
↪with preprocess=False           # needs sage.combinat
33
```

```
>>> from sage.all import *
>>> def fibonacci_knapsack(d, b, backend=None):
...     lp = MixedIntegerLinearProgram(base_ring=QQ)
...     x = lp.new_variable(nonnegative=True)
...     lp.add_constraint(lp.sum(fibonacci(i+Integer(3))*x[i] for i in␣
↪range(d)) <= b)
...     return lp.polyhedron(backend=backend)
```

(continues on next page)

(continued from previous page)

```
>>> fibonacci_knapsack(Integer(20), Integer(12)).integral_points_count()   #␣
↪does not finish with preprocess=False           # needs sage.combinat
33
```

is_effective(*Hstar*, *Hstar_as_lin_comb*)

Test for the effectiveness of the `Hstar` series of this polytope.

The `Hstar` series of the polytope is determined by the action of a subgroup of the polytope's `restricted_automorphism_group()`. The `Hstar` series is effective if it is a polynomial in t and the coefficient of each t^i is an effective character in the ring of class functions of the acting group. A character ρ is effective if the coefficients of the irreducible representations in the expression of ρ are nonnegative integers.

INPUT:

- `Hstar` – a rational function in t with coefficients in the ring of class functions
- `Hstar_as_lin_comb` – vector. The coefficients of the irreducible representations of the acting group in the expression of `Hstar` as a linear combination of irreducible representations with coefficients in the field of rational functions in t.

OUTPUT: boolean; whether the `Hstar` series is effective

See also

Hstar_function()

EXAMPLES:

The H^* series of the two-dimensional permutahedron under the action of the symmetric group is effective:

```
sage: # optional - pynormaliz
sage: p3 = polytopes.permutahedron(3, backend='normaliz')
sage: G = p3.restricted_automorphism_group(
....:         output='permutation')
sage: reflection12 = G([(0,2),(1,4),(3,5)])
sage: reflection23 = G([(0,1),(2,3),(4,5)])
sage: S3 = G.subgroup(gens=[reflection12, reflection23])
sage: S3.is_isomorphic(SymmetricGroup(3))
True
sage: Hstar = p3.Hstar_function(S3)
sage: Hlin  = p3.Hstar_function(S3,
....:                           output='Hstar_as_lin_comb')
sage: p3.is_effective(Hstar, Hlin)
True
```

```
>>> from sage.all import *
>>> # optional - pynormaliz
>>> p3 = polytopes.permutahedron(Integer(3), backend='normaliz')
>>> G = p3.restricted_automorphism_group(
...         output='permutation')
>>> reflection12 = G([(Integer(0),Integer(2)),(Integer(1),Integer(4)),
↪(Integer(3),Integer(5))])
>>> reflection23 = G([(Integer(0),Integer(1)),(Integer(2),Integer(3)),
```

(continues on next page)

(continued from previous page)

```
↪(Integer(4),Integer(5))]])
>>> S3 = G.subgroup(gens=[reflection12, reflection23])
>>> S3.is_isomorphic(SymmetricGroup(Integer(3)))
True
>>> Hstar = p3.Hstar_function(S3)
>>> Hlin  = p3.Hstar_function(S3,
...                            output='Hstar_as_lin_comb')
>>> p3.is_effective(Hstar, Hlin)
True
```

If the H^*-series is not polynomial, then it is not effective:

```
sage: # optional - pynormaliz
sage: P = Polyhedron(vertices=[[0,0,1], [0,0,-1], [1,0,1],
....:                          [-1,0,-1], [0,1,1],
....:                          [0,-1,-1], [1,1,1], [-1,-1,-1]],
....:                backend='normaliz')
sage: G = P.restricted_automorphism_group(
....:        output='permutation')
sage: H = G.subgroup(gens=[G([(0,2),(1,3),(4,6),(5,7)])])
sage: Hstar = P.Hstar_function(H); Hstar
(chi_0*t^4 + (3*chi_0 + 3*chi_1)*t^3
  + (8*chi_0 + 2*chi_1)*t^2 + (3*chi_0 + 3*chi_1)*t + chi_0)/(t + 1)
sage: Hstar_lin = P.Hstar_function(H,
....:                               output='Hstar_as_lin_comb')
sage: P.is_effective(Hstar, Hstar_lin)
False
```

```
>>> from sage.all import *
>>> # optional - pynormaliz
>>> P = Polyhedron(vertices=[[Integer(0),Integer(0),Integer(1)], [Integer(0),
↪Integer(0),-Integer(1)], [Integer(1),Integer(0),Integer(1)],
...                          [-Integer(1),Integer(0),-Integer(1)],␣
↪[Integer(0),Integer(1),Integer(1)],
...                          [Integer(0),-Integer(1),-Integer(1)],␣
↪[Integer(1),Integer(1),Integer(1)], [-Integer(1),-Integer(1),-Integer(1)]],
...                backend='normaliz')
>>> G = P.restricted_automorphism_group(
...        output='permutation')
>>> H = G.subgroup(gens=[G([(Integer(0),Integer(2)),(Integer(1),Integer(3)),
↪(Integer(4),Integer(6)),(Integer(5),Integer(7))])])
>>> Hstar = P.Hstar_function(H); Hstar
(chi_0*t^4 + (3*chi_0 + 3*chi_1)*t^3
  + (8*chi_0 + 2*chi_1)*t^2 + (3*chi_0 + 3*chi_1)*t + chi_0)/(t + 1)
>>> Hstar_lin = P.Hstar_function(H,
...                               output='Hstar_as_lin_comb')
>>> P.is_effective(Hstar, Hstar_lin)
False
```

2.6.11 Base class for polyhedra over $\mathbf{Z}$

class sage.geometry.polyhedron.base_ZZ.**Polyhedron_ZZ**(*parent*, *Vrep*, *Hrep*, *Vrep_minimal=None*, *Hrep_minimal=None*, *pref_rep=None*, *mutable=False*, ***kwds*)

Bases: *Polyhedron_QQ*

Base class for Polyhedra over $\mathbf{Z}$.

ehrhart_polynomial(*engine=None*, *variable='t'*, *verbose=False*, *dual=None*, *irrational_primal=None*, *irrational_all_primal=None*, *maxdet=None*, *no_decomposition=None*, *compute_vertex_cones=None*, *smith_form=None*, *dualization=None*, *triangulation=None*, *triangulation_max_height=None*, ***kwds*)

Return the Ehrhart polynomial of this polyhedron.

Let P be a lattice polytope in $\mathbf{R}^d$ and define $L(P,t) = \#(tP \cap \mathbf{Z}^d)$. Then E. Ehrhart proved in 1962 that L coincides with a rational polynomial of degree d for integer t. L is called the *Ehrhart polynomial* of P. For more information see the Wikipedia article Ehrhart_polynomial.

The Ehrhart polynomial may be computed using either LattE Integrale or Normaliz by setting `engine` to 'latte' or 'normaliz' respectively.

INPUT:

- `engine` – string; the backend to use. Allowed values are:
  - `None` (default); When no input is given the Ehrhart polynomial is computed using LattE Integrale (optional)
  - `'latte'`; use LattE integrale program (optional)
  - `'normaliz'`; use Normaliz program (optional). The backend of `self` must be set to 'normaliz'.
- `variable` – string (default: `'t'`); the variable in which the Ehrhart polynomial should be expressed
- When the `engine` is 'latte' or None, the additional input values are:
  - `verbose` – boolean (default: `False`); if `True`, print the whole output of the LattE command.

  The following options are passed to the LattE command, for details consult the LattE documentation:
  - `dual` – boolean; triangulate and signed-decompose in the dual space
  - `irrational_primal` – boolean; triangulate in the dual space, signed-decompose in the primal space using irrationalization.
  - `irrational_all_primal` – boolean; triangulate and signed-decompose in the primal space using irrationalization.
  - `maxdet` – integer; decompose down to an index (determinant) of `maxdet` instead of index 1 (unimodular cones).
  - `no_decomposition` – boolean; do not signed-decompose simplicial cones.
  - `compute_vertex_cones` – string; either 'cdd' or 'lrs' or '4ti2'
  - `smith_form` – string; either 'ilio' or 'lidia'
  - `dualization` – string; either 'cdd' or '4ti2'
  - `triangulation` – string; 'cddlib', '4ti2' or 'topcom'
  - `triangulation_max_height` – integer; use a uniform distribution of height from 1 to this number

OUTPUT:

The Ehrhart polynomial as a univariate polynomial in `variable` over a rational field.

See also

`latte` the interface to LattE Integrale PyNormaliz

EXAMPLES:

To start, we find the Ehrhart polynomial of a three-dimensional `simplex`, first using `engine='latte'`. Leaving the engine unspecified sets the `engine` to 'latte' by default:

```
sage: simplex = Polyhedron(vertices=[(0,0,0),(3,3,3),(-3,2,1),(1,-1,-2)])
sage: poly = simplex.ehrhart_polynomial(engine = 'latte')  # optional - latte_
↪int
sage: poly                                                  # optional - latte_
↪int
7/2*t^3 + 2*t^2 - 1/2*t + 1
sage: poly(1)                                               # optional - latte_
↪int
6
sage: len(simplex.integral_points())
6
sage: poly(2)                                               # optional - latte_
↪int
36
sage: len((2*simplex).integral_points())
36
```

```
>>> from sage.all import *
>>> simplex = Polyhedron(vertices=[(Integer(0),Integer(0),Integer(0)),
↪(Integer(3),Integer(3),Integer(3)),(-Integer(3),Integer(2),Integer(1)),
↪(Integer(1),-Integer(1),-Integer(2))])
>>> poly = simplex.ehrhart_polynomial(engine = 'latte')  # optional - latte_
↪int
>>> poly                                                  # optional - latte_
↪int
7/2*t^3 + 2*t^2 - 1/2*t + 1
>>> poly(Integer(1))                                                # optional -
↪ latte_int
6
>>> len(simplex.integral_points())
6
>>> poly(Integer(2))                                                # optional -
↪ latte_int
36
>>> len((Integer(2)*simplex).integral_points())
36
```

Now we find the same Ehrhart polynomial, this time using `engine='normaliz'`. To use the Normaliz engine, the `simplex` must be defined with `backend='normaliz'`:

```
sage: simplex = Polyhedron(vertices=[(0,0,0),(3,3,3),(-3,2,1),(1,-1,-2)],␣
↪backend='normaliz') # optional - pynormaliz
sage: poly = simplex.ehrhart_polynomial(engine='normaliz') # optional -␣
↪pynormaliz
sage: poly                                                  # optional -␣
↪pynormaliz
7/2*t^3 + 2*t^2 - 1/2*t + 1
```

```
>>> from sage.all import *
>>> simplex = Polyhedron(vertices=[(Integer(0),Integer(0),Integer(0)),
↪(Integer(3),Integer(3),Integer(3)),(-Integer(3),Integer(2),Integer(1)),
↪(Integer(1),-Integer(1),-Integer(2))], backend='normaliz') # optional -␣
↪pynormaliz
>>> poly = simplex.ehrhart_polynomial(engine='normaliz') # optional -␣
↪pynormaliz
>>> poly                                                  # optional -␣
↪pynormaliz
7/2*t^3 + 2*t^2 - 1/2*t + 1
```

If the `engine='normaliz'`, the backend should be `'normaliz'`, otherwise it returns an error:

```
sage: simplex = Polyhedron(vertices=[(0,0,0),(3,3,3),(-3,2,1),(1,-1,-2)])
sage: simplex.ehrhart_polynomial(engine='normaliz')
Traceback (most recent call last):
...
TypeError: The polyhedron's backend should be 'normaliz'
```

```
>>> from sage.all import *
>>> simplex = Polyhedron(vertices=[(Integer(0),Integer(0),Integer(0)),
↪(Integer(3),Integer(3),Integer(3)),(-Integer(3),Integer(2),Integer(1)),
↪(Integer(1),-Integer(1),-Integer(2))])
>>> simplex.ehrhart_polynomial(engine='normaliz')
Traceback (most recent call last):
...
TypeError: The polyhedron's backend should be 'normaliz'
```

Now we find the Ehrhart polynomials of the unit hypercubes of dimensions three through six. They are computed first with `engine='latte'` and then with `engine='normaliz'`. The degree of the Ehrhart polynomial matches the dimension of the hypercube, and the coefficient of the leading monomial equals the volume of the unit hypercube:

```
sage: # optional - latte_int
sage: from itertools import product
sage: def hypercube(d):
....:     return Polyhedron(vertices=list(product([0,1],repeat=d)))
sage: hypercube(3).ehrhart_polynomial()
t^3 + 3*t^2 + 3*t + 1
sage: hypercube(4).ehrhart_polynomial()
t^4 + 4*t^3 + 6*t^2 + 4*t + 1
sage: hypercube(5).ehrhart_polynomial()
t^5 + 5*t^4 + 10*t^3 + 10*t^2 + 5*t + 1
sage: hypercube(6).ehrhart_polynomial()
t^6 + 6*t^5 + 15*t^4 + 20*t^3 + 15*t^2 + 6*t + 1
```

(continues on next page)

(continued from previous page)

```
sage: # optional - pynormaliz
sage: from itertools import product
sage: def hypercube(d):
....:     return Polyhedron(vertices=list(product([0,1],repeat=d)),backend=
↪'normaliz')
sage: hypercube(3).ehrhart_polynomial(engine='normaliz')
t^3 + 3*t^2 + 3*t + 1
sage: hypercube(4).ehrhart_polynomial(engine='normaliz')
t^4 + 4*t^3 + 6*t^2 + 4*t + 1
sage: hypercube(5).ehrhart_polynomial(engine='normaliz')
t^5 + 5*t^4 + 10*t^3 + 10*t^2 + 5*t + 1
sage: hypercube(6).ehrhart_polynomial(engine='normaliz')
t^6 + 6*t^5 + 15*t^4 + 20*t^3 + 15*t^2 + 6*t + 1
```

```
>>> from sage.all import *
>>> # optional - latte_int
>>> from itertools import product
>>> def hypercube(d):
...     return Polyhedron(vertices=list(product([Integer(0),Integer(1)],
↪repeat=d)))
>>> hypercube(Integer(3)).ehrhart_polynomial()
t^3 + 3*t^2 + 3*t + 1
>>> hypercube(Integer(4)).ehrhart_polynomial()
t^4 + 4*t^3 + 6*t^2 + 4*t + 1
>>> hypercube(Integer(5)).ehrhart_polynomial()
t^5 + 5*t^4 + 10*t^3 + 10*t^2 + 5*t + 1
>>> hypercube(Integer(6)).ehrhart_polynomial()
t^6 + 6*t^5 + 15*t^4 + 20*t^3 + 15*t^2 + 6*t + 1

>>> # optional - pynormaliz
>>> from itertools import product
>>> def hypercube(d):
...     return Polyhedron(vertices=list(product([Integer(0),Integer(1)],
↪repeat=d)),backend='normaliz')
>>> hypercube(Integer(3)).ehrhart_polynomial(engine='normaliz')
t^3 + 3*t^2 + 3*t + 1
>>> hypercube(Integer(4)).ehrhart_polynomial(engine='normaliz')
t^4 + 4*t^3 + 6*t^2 + 4*t + 1
>>> hypercube(Integer(5)).ehrhart_polynomial(engine='normaliz')
t^5 + 5*t^4 + 10*t^3 + 10*t^2 + 5*t + 1
>>> hypercube(Integer(6)).ehrhart_polynomial(engine='normaliz')
t^6 + 6*t^5 + 15*t^4 + 20*t^3 + 15*t^2 + 6*t + 1
```

An empty polyhedron:

```
sage: p = Polyhedron(ambient_dim=3, vertices=[])
sage: p.ehrhart_polynomial()
0
sage: parent(_)
Univariate Polynomial Ring in t over Rational Field
```

```
>>> from sage.all import *
>>> p = Polyhedron(ambient_dim=Integer(3), vertices=[])
>>> p.ehrhart_polynomial()
0
>>> parent(_)
Univariate Polynomial Ring in t over Rational Field
```

The polyhedron should be compact:

```
sage: C = Polyhedron(rays=[[1,2],[2,1]])
sage: C.ehrhart_polynomial()
Traceback (most recent call last):
...
ValueError: Ehrhart polynomial only defined for compact polyhedra
```

```
>>> from sage.all import *
>>> C = Polyhedron(rays=[[Integer(1),Integer(2)],[Integer(2),Integer(1)]])
>>> C.ehrhart_polynomial()
Traceback (most recent call last):
...
ValueError: Ehrhart polynomial only defined for compact polyhedra
```

fibration_generator(*dim*)

Generate the lattice polytope fibrations.

For the purposes of this function, a lattice polytope fiber is a sub-lattice polytope. Projecting the plane spanned by the subpolytope to a point yields another lattice polytope, the base of the fibration.

INPUT:

- `dim` – integer; the dimension of the lattice polytope fiber

OUTPUT:

A generator yielding the distinct lattice polytope fibers of given dimension.

EXAMPLES:

```
sage: P = Polyhedron(toric_varieties.P4_11169().fan().rays(), base_ring=ZZ)  ␣
↪          # needs palp sage.graphs
sage: list(P.fibration_generator(2))                                        #␣
↪needs palp sage.graphs
[A 2-dimensional polyhedron in ZZ^4 defined as the convex hull of 3 vertices]
```

```
>>> from sage.all import *
>>> P = Polyhedron(toric_varieties.P4_11169().fan().rays(), base_ring=ZZ)    ␣
↪         # needs palp sage.graphs
>>> list(P.fibration_generator(Integer(2)))                                 ␣
↪      # needs palp sage.graphs
[A 2-dimensional polyhedron in ZZ^4 defined as the convex hull of 3 vertices]
```

find_translation(*translated_polyhedron*)

Return the translation vector to `translated_polyhedron`.

INPUT:

- `translated_polyhedron` – a polyhedron

OUTPUT:

A $\mathbf{Z}$-vector that translates `self` to `translated_polyhedron`. A `ValueError` is raised if `translated_polyhedron` is not a translation of `self`, this can be used to check that two polyhedra are not translates of each other.

EXAMPLES:

```
sage: X = polytopes.cube()
sage: X.find_translation(X + vector([2,3,5]))
(2, 3, 5)
sage: X.find_translation(2*X)
Traceback (most recent call last):
...
ValueError: polyhedron is not a translation of self
```

```
>>> from sage.all import *
>>> X = polytopes.cube()
>>> X.find_translation(X + vector([Integer(2),Integer(3),Integer(5)]))
(2, 3, 5)
>>> X.find_translation(Integer(2)*X)
Traceback (most recent call last):
...
ValueError: polyhedron is not a translation of self
```

has_IP_property()

Test whether the polyhedron has the IP property.

The IP (interior point) property means that

- `self` is compact (a polytope).
- `self` contains the origin as an interior point.

This implies that

- `self` is full-dimensional.
- The dual polyhedron is again a polytope (that is, a compact polyhedron), though not necessarily a lattice polytope.

EXAMPLES:

```
sage: Polyhedron([(1,1),(1,0),(0,1)], base_ring=ZZ).has_IP_property()
False
sage: Polyhedron([(0,0),(1,0),(0,1)], base_ring=ZZ).has_IP_property()
False
sage: Polyhedron([(-1,-1),(1,0),(0,1)], base_ring=ZZ).has_IP_property()
True
```

```
>>> from sage.all import *
>>> Polyhedron([(Integer(1),Integer(1)),(Integer(1),Integer(0)),(Integer(0),
↪Integer(1))], base_ring=ZZ).has_IP_property()
False
>>> Polyhedron([(Integer(0),Integer(0)),(Integer(1),Integer(0)),(Integer(0),
↪Integer(1))], base_ring=ZZ).has_IP_property()
False
```

(continues on next page)

(continued from previous page)

```
>>> Polyhedron([(-Integer(1),-Integer(1)),(Integer(1),Integer(0)),(Integer(0),
↪Integer(1))], base_ring=ZZ).has_IP_property()
True
```

REFERENCES:

- [PALP]

is_lattice_polytope()

Return whether the polyhedron is a lattice polytope.

OUTPUT:

`True` if the polyhedron is compact and has only integral vertices, `False` otherwise.

EXAMPLES:

```
sage: polytopes.cross_polytope(3).is_lattice_polytope()
True
sage: polytopes.regular_polygon(5).is_lattice_polytope()                        #␣
↪needs sage.rings.number_field
False
```

```
>>> from sage.all import *
>>> polytopes.cross_polytope(Integer(3)).is_lattice_polytope()
True
>>> polytopes.regular_polygon(Integer(5)).is_lattice_polytope()                  ␣
↪     # needs sage.rings.number_field
False
```

is_reflexive()

A lattice polytope is reflexive if it contains the origin in its interior and its polar with respect to the origin is a lattice polytope.

Equivalently, it is reflexive if it is of the form $\{x \in \mathbb{R}^d : Ax \leq 1\}$ for some integer matrix A and d the ambient dimension.

EXAMPLES:

```
sage: p = Polyhedron(vertices=[(1,0,0),(0,1,0),(0,0,1),(-1,-1,-1)], base_
↪ring=ZZ)
sage: p.is_reflexive()
True
sage: polytopes.hypercube(4).is_reflexive()
True
sage: p = Polyhedron(vertices=[(1,0), (0,2), (-1,0), (0,-1)], base_ring=ZZ)
sage: p.is_reflexive()
False
sage: p = Polyhedron(vertices=[(1,0), (0,2), (-1,0)], base_ring=ZZ)
sage: p.is_reflexive()
False
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices=[(Integer(1),Integer(0),Integer(0)),(Integer(0),
↪Integer(1),Integer(0)),(Integer(0),Integer(0),Integer(1)),(-Integer(1),-
```

(continues on next page)

(continued from previous page)

```
↪Integer(1),-Integer(1))], base_ring=ZZ)
>>> p.is_reflexive()
True
>>> polytopes.hypercube(Integer(4)).is_reflexive()
True
>>> p = Polyhedron(vertices=[(Integer(1),Integer(0)), (Integer(0),Integer(2)),
↪ (-Integer(1),Integer(0)), (Integer(0),-Integer(1))], base_ring=ZZ)
>>> p.is_reflexive()
False
>>> p = Polyhedron(vertices=[(Integer(1),Integer(0)), (Integer(0),Integer(2)),
↪ (-Integer(1),Integer(0))], base_ring=ZZ)
>>> p.is_reflexive()
False
```

An error is raised, if the polyhedron is not compact:

```
sage: p = Polyhedron(rays=[(1,)], base_ring=ZZ)
sage: p.is_reflexive()
Traceback (most recent call last):
...
ValueError: the polyhedron is not compact
```

```
>>> from sage.all import *
>>> p = Polyhedron(rays=[(Integer(1),)], base_ring=ZZ)
>>> p.is_reflexive()
Traceback (most recent call last):
...
ValueError: the polyhedron is not compact
```

minkowski_decompositions()

Return all Minkowski sums that add up to the polyhedron.

OUTPUT:

A tuple consisting of pairs (X, Y) of $\mathbf{Z}$-polyhedra that add up to `self`. All pairs up to exchange of the summands are returned, that is, (Y, X) is not included if (X, Y) already is.

EXAMPLES:

```
sage: square = Polyhedron(vertices=[(0,0),(1,0),(0,1),(1,1)])
sage: square.minkowski_decompositions()
((A 0-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4␣
↪vertices),
 (A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices,
  A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2␣
↪vertices))
```

```
>>> from sage.all import *
>>> square = Polyhedron(vertices=[(Integer(0),Integer(0)),(Integer(1),
↪Integer(0)),(Integer(0),Integer(1)),(Integer(1),Integer(1))])
>>> square.minkowski_decompositions()
((A 0-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex,
```

(continues on next page)

(continued from previous page)

```
 A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4␣
↪vertices),
 (A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices,
  A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2␣
↪vertices))
```

Example from http://cgi.di.uoa.gr/~amantzaf/geo/

```
sage: Q = Polyhedron(vertices=[(4,0), (6,0), (0,3), (4,3)])
sage: R = Polyhedron(vertices=[(0,0), (5,0), (8,4), (3,2)])
sage: (Q+R).minkowski_decompositions()
((A 0-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 7␣
↪vertices),
 (A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4␣
↪vertices,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4␣
↪vertices),
 (A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2␣
↪vertices,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 7␣
↪vertices),
 (A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 5␣
↪vertices,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4␣
↪vertices),
 (A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2␣
↪vertices,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 7␣
↪vertices),
 (A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 5␣
↪vertices,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 3␣
↪vertices),
 (A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2␣
↪vertices,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 7␣
↪vertices),
 (A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2␣
↪vertices,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 6␣
↪vertices))

sage: [ len(square.dilation(i).minkowski_decompositions())
....:   for i in range(6) ]
[1, 2, 5, 8, 13, 18]
sage: [ integer_ceil((i^2 + 2*i - 1) / 2) + 1 for i in range(10) ]
[1, 2, 5, 8, 13, 18, 25, 32, 41, 50]
```

normal_form(*algorithm='palp_native'*, *permutation=False*)

Return the normal form of vertices of the lattice polytope `self`.

INPUT:

- `algorithm` – must be `'palp_native'`, the default
- `permutation` – boolean (default: `False`); if `True`, the permutation applied to vertices to obtain the normal form is returned as well

For more more detail, see `normal_form()`.

EXAMPLES:

We compute the normal form of the “diamond”:

```
sage: d = Polyhedron([(1,0), (0,1), (-1,0), (0,-1)])
sage: d.vertices()
(A vertex at (-1, 0),
 A vertex at (0, -1),
 A vertex at (0, 1),
 A vertex at (1, 0))
sage: d.normal_form()                                                        #␣
↪needs sage.groups
[(1, 0), (0, 1), (0, -1), (-1, 0)]
sage: d.lattice_polytope().normal_form("palp_native")                       #␣
↪needs sage.groups
M( 1,  0),
M( 0,  1),
M( 0, -1),
M(-1,  0)
in 2-d lattice M
```

```
>>> from sage.all import *
>>> d = Polyhedron([(Integer(1),Integer(0)), (Integer(0),Integer(1)), (-
↪Integer(1),Integer(0)), (Integer(0),-Integer(1))])
>>> d.vertices()
(A vertex at (-1, 0),
 A vertex at (0, -1),
 A vertex at (0, 1),
 A vertex at (1, 0))
>>> d.normal_form()                                                        #␣
↪needs sage.groups
[(1, 0), (0, 1), (0, -1), (-1, 0)]
>>> d.lattice_polytope().normal_form("palp_native")                       #␣
↪needs sage.groups
M( 1,  0),
M( 0,  1),
M( 0, -1),
M(-1,  0)
in 2-d lattice M
```

Using `permutation=True`:

```
sage: d.normal_form(permutation=True)                                      #␣
↪needs sage.groups
([(1, 0), (0, 1), (0, -1), (-1, 0)], ())
```

```
>>> from sage.all import *
>>> d.normal_form(permutation=True)                                        #␣
```

(continues on next page)

(continued from previous page)

```
↪needs sage.groups
([(1, 0), (0, 1), (0, -1), (-1, 0)], ())
```

It is not possible to compute normal forms for polytopes which do not span the space:

```
sage: p = Polyhedron([(1,0,0), (0,1,0), (-1,0,0), (0,-1,0)])
sage: p.normal_form()
Traceback (most recent call last):
...
ValueError: normal form is not defined for lower-dimensional polyhedra, got
A 2-dimensional polyhedron in ZZ^3 defined as the convex hull of 4 vertices
```

```
>>> from sage.all import *
>>> p = Polyhedron([(Integer(1),Integer(0),Integer(0)), (Integer(0),
↪Integer(1),Integer(0)), (-Integer(1),Integer(0),Integer(0)), (Integer(0),-
↪Integer(1),Integer(0))])
>>> p.normal_form()
Traceback (most recent call last):
...
ValueError: normal form is not defined for lower-dimensional polyhedra, got
A 2-dimensional polyhedron in ZZ^3 defined as the convex hull of 4 vertices
```

The normal form is also not defined for unbounded polyhedra:

```
sage: p = Polyhedron(vertices=[[1, 1]], rays=[[1, 0], [0, 1]], base_ring=ZZ)
sage: p.normal_form()
Traceback (most recent call last):
...
ValueError: normal form is not defined for unbounded polyhedra, got
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex and␣
↪2 rays
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices=[[Integer(1), Integer(1)]], rays=[[Integer(1),␣
↪Integer(0)], [Integer(0), Integer(1)]], base_ring=ZZ)
>>> p.normal_form()
Traceback (most recent call last):
...
ValueError: normal form is not defined for unbounded polyhedra, got
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex and␣
↪2 rays
```

See Issue #15280 for proposed extensions to these cases.

polar()

Return the polar (dual) polytope.

The polytope must have the IP-property (see *has_IP_property()*), that is, the origin must be an interior point. In particular, it must be full-dimensional.

OUTPUT:

The polytope whose vertices are the coefficient vectors of the inequalities of `self` with inhomogeneous term normalized to unity.

EXAMPLES:

```
sage: p = Polyhedron(vertices=[(1,0,0),(0,1,0),(0,0,1),(-1,-1,-1)], base_
↪ring=ZZ)
sage: p.polar()
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 4 vertices
sage: type(_)
<class 'sage.geometry.polyhedron.parent.Polyhedra_ZZ_ppl_with_category.
↪element_class'>
sage: p.polar().base_ring()
Integer Ring
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices=[(Integer(1),Integer(0),Integer(0)),(Integer(0),
↪Integer(1),Integer(0)),(Integer(0),Integer(0),Integer(1)),(-Integer(1),-
↪Integer(1),-Integer(1))], base_ring=ZZ)
>>> p.polar()
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 4 vertices
>>> type(_)
<class 'sage.geometry.polyhedron.parent.Polyhedra_ZZ_ppl_with_category.
↪element_class'>
>>> p.polar().base_ring()
Integer Ring
```

2.6.12 Base class for polyhedra over RDF

class sage.geometry.polyhedron.base_RDF.**Polyhedron_RDF**(*parent*, *Vrep*, *Hrep*, *Vrep_minimal=None*, *Hrep_minimal=None*, *pref_rep=None*, *mutable=False*, ***kwds*)

Bases: *Polyhedron_base*

Base class for polyhedra over RDF.

2.7 Backends for Polyhedra

2.7.1 The cdd backend for polyhedral computations

class sage.geometry.polyhedron.backend_cdd.**Polyhedron_QQ_cdd**(*parent*, *Vrep*, *Hrep*, ***kwds*)

Bases: *Polyhedron_cdd*, *Polyhedron_QQ*

Polyhedra over QQ with cdd.

INPUT:

- parent – the parent, an instance of *Polyhedra*
- Vrep – list [vertices, rays, lines] or None
- Hrep – list [ieqs, eqns] or None

EXAMPLES:

```
sage: from sage.geometry.polyhedron.parent import Polyhedra
sage: parent = Polyhedra(QQ, 2, backend='cdd')
sage: from sage.geometry.polyhedron.backend_cdd import Polyhedron_QQ_cdd
```

(continues on next page)

(continued from previous page)

```
sage: Polyhedron_QQ_cdd(parent, [ [(1,0),(0,1),(0,0)], [], []], None,␣
↪verbose=False)
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.parent import Polyhedra
>>> parent = Polyhedra(QQ, Integer(2), backend='cdd')
>>> from sage.geometry.polyhedron.backend_cdd import Polyhedron_QQ_cdd
>>> Polyhedron_QQ_cdd(parent, [ [(Integer(1),Integer(0)),(Integer(0),Integer(1)),
↪(Integer(0),Integer(0))], [], []], None, verbose=False)
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices
```

class sage.geometry.polyhedron.backend_cdd.**Polyhedron_cdd**(*parent*, *Vrep*, *Hrep*, *Vrep_minimal=None*, *Hrep_minimal=None*, *pref_rep=None*, *mutable=False*, ***kwds*)

Bases: *Polyhedron_base*

Base class for the cdd backend.

2.7.2 The cdd backend for polyhedral computations, floating point version

class sage.geometry.polyhedron.backend_cdd_rdf.**Polyhedron_RDF_cdd**(*parent*, *Vrep*, *Hrep*, ***kwds*)

Bases: *Polyhedron_cdd*, *Polyhedron_RDF*

Polyhedra over RDF with cdd.

INPUT:

- `ambient_dim` – integer; the dimension of the ambient space
- `Vrep` – list `[vertices, rays, lines]` or `None`
- `Hrep` – list `[ieqs, eqns]` or `None`

EXAMPLES:

```
sage: from sage.geometry.polyhedron.parent import Polyhedra
sage: parent = Polyhedra(RDF, 2, backend='cdd')
sage: from sage.geometry.polyhedron.backend_cdd_rdf import Polyhedron_RDF_cdd
sage: Polyhedron_RDF_cdd(parent, [ [(1,0),(0,1),(0,0)], [], []], None,␣
↪verbose=False)
A 2-dimensional polyhedron in RDF^2 defined as the convex hull of 3 vertices
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.parent import Polyhedra
>>> parent = Polyhedra(RDF, Integer(2), backend='cdd')
>>> from sage.geometry.polyhedron.backend_cdd_rdf import Polyhedron_RDF_cdd
>>> Polyhedron_RDF_cdd(parent, [ [(Integer(1),Integer(0)),(Integer(0),Integer(1)),
↪(Integer(0),Integer(0))], [], []], None, verbose=False)
A 2-dimensional polyhedron in RDF^2 defined as the convex hull of 3 vertices
```

2.7.3 The Python backend

While slower than specialized C/C++ implementations, the implementation is general and works with any exact field in Sage that allows you to define polyhedra.

EXAMPLES:

```
sage: # needs sage.rings.number_field
sage: p0 = (0, 0)
sage: p1 = (1, 0)
sage: p2 = (1/2, AA(3).sqrt()/2)
sage: equilateral_triangle = Polyhedron([p0, p1, p2])
sage: equilateral_triangle.vertices()
(A vertex at (0, 0),
 A vertex at (1, 0),
 A vertex at (0.500000000000000?, 0.866025403784439?))
sage: equilateral_triangle.inequalities()
(An inequality (-1, -0.5773502691896258?) x + 1 >= 0,
 An inequality (1, -0.5773502691896258?) x + 0 >= 0,
 An inequality (0, 1.154700538379252?) x + 0 >= 0)
```

```
>>> from sage.all import *
>>> # needs sage.rings.number_field
>>> p0 = (Integer(0), Integer(0))
>>> p1 = (Integer(1), Integer(0))
>>> p2 = (Integer(1)/Integer(2), AA(Integer(3)).sqrt()/Integer(2))
>>> equilateral_triangle = Polyhedron([p0, p1, p2])
>>> equilateral_triangle.vertices()
(A vertex at (0, 0),
 A vertex at (1, 0),
 A vertex at (0.500000000000000?, 0.866025403784439?))
>>> equilateral_triangle.inequalities()
(An inequality (-1, -0.5773502691896258?) x + 1 >= 0,
 An inequality (1, -0.5773502691896258?) x + 0 >= 0,
 An inequality (0, 1.154700538379252?) x + 0 >= 0)
```

class sage.geometry.polyhedron.backend_field.**Polyhedron_field**(*parent*, *Vrep*, *Hrep*, *Vrep_minimal=None*, *Hrep_minimal=None*, *pref_rep=None*, *mutable=False*, ***kwds*)

Bases: *Polyhedron_base*

Polyhedra over all fields supported by Sage.

INPUT:

- `Vrep` – list `[vertices, rays, lines]` or `None`
- `Hrep` – list `[ieqs, eqns]` or `None`

EXAMPLES:

```
sage: p = Polyhedron(vertices=[(0,0),(AA(2).sqrt(),0),(0,AA(3).sqrt())],          #␣
↪needs sage.rings.number_field
....:                rays=[(1,1)], lines=[], backend='field', base_ring=AA)
```

(continues on next page)

(continued from previous page)

```
sage: TestSuite(p).run()                                                    #␣
↪needs sage.rings.number_field
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices=[(Integer(0),Integer(0)),(AA(Integer(2)).sqrt(),
↪Integer(0)),(Integer(0),AA(Integer(3)).sqrt())],        # needs sage.rings.
↪number_field
...                rays=[(Integer(1),Integer(1))], lines=[], backend='field',␣
↪base_ring=AA)
>>> TestSuite(p).run()                                                      #␣
↪needs sage.rings.number_field
```

2.7.4 The Python backend, using number fields internally

class sage.geometry.polyhedron.backend_number_field.**Polyhedron_number_field**(*parent*, *Vrep*, *Hrep*, *Vrep_minimal=None*, *Hrep_minimal=None*, *pref_rep=None*, *mutable=False*, ***kwds*)

Bases: *Polyhedron_field*, Polyhedron_base_number_field

Polyhedra whose data can be converted to number field elements.

All computations are done internally using a fixed real embedded number field, which is determined automatically.

INPUT:

- Vrep – list [vertices, rays, lines] or None
- Hrep – list [ieqs, eqns] or None

EXAMPLES:

```
sage: P = Polyhedron(vertices=[[1], [sqrt(2)]], backend='number_field'); P      #␣
↪needs sage.rings.number_field sage.symbolic
A 1-dimensional polyhedron
 in (Symbolic Ring)^1 defined as the convex hull of 2 vertices
sage: P.vertices()                                                          #␣
↪needs sage.rings.number_field sage.symbolic
(A vertex at (1), A vertex at (sqrt(2)))

sage: P = polytopes.icosahedron(exact=True, backend='number_field')         #␣
↪needs sage.rings.number_field
sage: P                                                                     #␣
↪needs sage.rings.number_field
A 3-dimensional polyhedron
 in (Number Field in sqrt5 with defining polynomial x^2 - 5
     with sqrt5 = 2.236067977499790?)^3
 defined as the convex hull of 12 vertices
```

(continues on next page)

(continued from previous page)

```
sage: x = polygen(ZZ); P = Polyhedron(                                    #␣
↪needs sage.rings.number_field sage.symbolic
....:     vertices=[[sqrt(2)], [AA.polynomial_root(x^3-2, RIF(0,3))]],
....:     backend='number_field')
sage: P                                                                   #␣
↪needs sage.rings.number_field sage.symbolic
A 1-dimensional polyhedron
 in (Symbolic Ring)^1 defined as the convex hull of 2 vertices
sage: P.vertices()                                                        #␣
↪needs sage.rings.number_field sage.symbolic
(A vertex at (sqrt(2)), A vertex at (2^(1/3)))
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[[Integer(1)], [sqrt(Integer(2))]], backend='number_
↪field'); P      # needs sage.rings.number_field sage.symbolic
A 1-dimensional polyhedron
 in (Symbolic Ring)^1 defined as the convex hull of 2 vertices
>>> P.vertices()                                                        #␣
↪needs sage.rings.number_field sage.symbolic
(A vertex at (1), A vertex at (sqrt(2)))

>>> P = polytopes.icosahedron(exact=True, backend='number_field')       #␣
↪needs sage.rings.number_field
>>> P                                                                   #␣
↪needs sage.rings.number_field
A 3-dimensional polyhedron
 in (Number Field in sqrt5 with defining polynomial x^2 - 5
     with sqrt5 = 2.236067977499790?)^3
 defined as the convex hull of 12 vertices

>>> x = polygen(ZZ); P = Polyhedron(                                    #␣
↪needs sage.rings.number_field sage.symbolic
...     vertices=[[sqrt(Integer(2))], [AA.polynomial_root(x**Integer(3)-
↪Integer(2), RIF(Integer(0),Integer(3)))]],
...     backend='number_field')
>>> P                                                                   #␣
↪needs sage.rings.number_field sage.symbolic
A 1-dimensional polyhedron
 in (Symbolic Ring)^1 defined as the convex hull of 2 vertices
>>> P.vertices()                                                        #␣
↪needs sage.rings.number_field sage.symbolic
(A vertex at (sqrt(2)), A vertex at (2^(1/3)))
```

2.7.5 The Normaliz backend for polyhedral computations

Note

This backend requires PyNormaliz. To install PyNormaliz, type `sage -i pynormaliz` in the terminal.

AUTHORS:

- Matthias Köppe (2016-12): initial version
- Jean-Philippe Labbé (2019-04): Expose normaliz features and added functionalities

class sage.geometry.polyhedron.backend_normaliz.**Polyhedron_QQ_normaliz**(*parent*, *Vrep*, *Hrep*, *normaliz_cone=None*, *normaliz_data=None*, *internal_base_ring=None*, ***kwds*)

Bases: *Polyhedron_normaliz*, *Polyhedron_QQ*

Polyhedra over $\mathbf{Q}$ with normaliz.

INPUT:

- `Vrep` – list `[vertices, rays, lines]` or `None`
- `Hrep` – list `[ieqs, eqns]` or `None`

EXAMPLES:

```
sage: p = Polyhedron(vertices=[(0,0), (1,0), (0,1)],
....:                rays=[(1,1)], lines=[],
....:                backend='normaliz', base_ring=QQ)
sage: TestSuite(p).run()
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices=[(Integer(0),Integer(0)), (Integer(1),Integer(0)),
↪(Integer(0),Integer(1))],
...                rays=[(Integer(1),Integer(1))], lines=[],
...                backend='normaliz', base_ring=QQ)
>>> TestSuite(p).run()
```

ehrhart_series(*variable='t'*)

Return the Ehrhart series of a compact rational polyhedron.

The Ehrhart series is the generating function where the coefficient of t^k is number of integer lattice points inside the k-th dilation of the polytope.

INPUT:

- `variable` – string (default: `'t'`)

OUTPUT: a rational function

EXAMPLES:

```
sage: S = Polyhedron(vertices=[[0,1], [1,0]], backend='normaliz')
sage: ES = S.ehrhart_series()
sage: ES.numerator()
1
sage: ES.denominator().factor()
(t - 1)^2

sage: C = Polyhedron(vertices=[[0,0,0], [0,0,1], [0,1,0], [0,1,1],
....:                          [1,0,0], [1,0,1], [1,1,0], [1,1,1]],
....:                  backend='normaliz')
```

(continues on next page)

(continued from previous page)

```
sage: ES = C.ehrhart_series()
sage: ES.numerator()
t^2 + 4*t + 1
sage: ES.denominator().factor()
(t - 1)^4
```

```
>>> from sage.all import *
>>> S = Polyhedron(vertices=[[Integer(0),Integer(1)], [Integer(1),
↪Integer(0)]], backend='normaliz')
>>> ES = S.ehrhart_series()
>>> ES.numerator()
1
>>> ES.denominator().factor()
(t - 1)^2

>>> C = Polyhedron(vertices=[[Integer(0),Integer(0),Integer(0)], [Integer(0),
↪Integer(0),Integer(1)], [Integer(0),Integer(1),Integer(0)], [Integer(0),
↪Integer(1),Integer(1)],
...                           [Integer(1),Integer(0),Integer(0)], [Integer(1),
↪Integer(0),Integer(1)], [Integer(1),Integer(1),Integer(0)], [Integer(1),
↪Integer(1),Integer(1)]],
...                 backend='normaliz')
>>> ES = C.ehrhart_series()
>>> ES.numerator()
t^2 + 4*t + 1
>>> ES.denominator().factor()
(t - 1)^4
```

The following example is from the Normaliz manual contained in the file `rational.in`:

```
sage: rat_poly = Polyhedron(vertices=[[1/2,1/2], [-1/3,-1/3], [1/4,-1/2]],
....:                       backend='normaliz')
sage: ES = rat_poly.ehrhart_series()
sage: ES.numerator()
2*t^6 + 3*t^5 + 4*t^4 + 3*t^3 + t^2 + t + 1
sage: ES.denominator().factor()
(-1) * (t + 1)^2 * (t - 1)^3 * (t^2 + 1) * (t^2 + t + 1)
```

```
>>> from sage.all import *
>>> rat_poly = Polyhedron(vertices=[[Integer(1)/Integer(2),Integer(1)/
↪Integer(2)], [-Integer(1)/Integer(3),-Integer(1)/Integer(3)], [Integer(1)/
↪Integer(4),-Integer(1)/Integer(2)]],
...                       backend='normaliz')
>>> ES = rat_poly.ehrhart_series()
>>> ES.numerator()
2*t^6 + 3*t^5 + 4*t^4 + 3*t^3 + t^2 + t + 1
>>> ES.denominator().factor()
(-1) * (t + 1)^2 * (t - 1)^3 * (t^2 + 1) * (t^2 + t + 1)
```

The polyhedron should be compact:

```
sage: C = Polyhedron(rays=[[1,2], [2,1]], backend='normaliz')
sage: C.ehrhart_series()
Traceback (most recent call last):
...
NotImplementedError: Ehrhart series can only be computed for compact␣
↪polyhedron
```

```
>>> from sage.all import *
>>> C = Polyhedron(rays=[[Integer(1),Integer(2)], [Integer(2),Integer(1)]],␣
↪backend='normaliz')
>>> C.ehrhart_series()
Traceback (most recent call last):
...
NotImplementedError: Ehrhart series can only be computed for compact␣
↪polyhedron
```

See also

`hilbert_series()`

hilbert_series(*grading*, *variable='t'*)

Return the Hilbert series of the polyhedron with respect to `grading`.

INPUT:

- `grading` – vector. The grading to use to form the Hilbert series
- `variable` – string (default: `'t'`)

OUTPUT: a rational function

EXAMPLES:

```
sage: C = Polyhedron(backend='normaliz',
....:                rays=[[0,0,1], [0,1,1], [1,0,1], [1,1,1]])
sage: HS = C.hilbert_series([1,1,1])
sage: HS.numerator()
t^2 + 1
sage: HS.denominator().factor()
(-1) * (t + 1) * (t - 1)^3 * (t^2 + t + 1)
```

```
>>> from sage.all import *
>>> C = Polyhedron(backend='normaliz',
...                rays=[[Integer(0),Integer(0),Integer(1)], [Integer(0),
↪Integer(1),Integer(1)], [Integer(1),Integer(0),Integer(1)], [Integer(1),
↪Integer(1),Integer(1)]])
>>> HS = C.hilbert_series([Integer(1),Integer(1),Integer(1)])
>>> HS.numerator()
t^2 + 1
>>> HS.denominator().factor()
(-1) * (t + 1) * (t - 1)^3 * (t^2 + t + 1)
```

By changing the grading, you can get the Ehrhart series of the square lifted at height 1:

```
sage: C.hilbert_series([0,0,1])
(t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
```

```
>>> from sage.all import *
>>> C.hilbert_series([Integer(0),Integer(0),Integer(1)])
(t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
```

Here is an example `2cone.in` from the Normaliz manual:

```
sage: C = Polyhedron(backend='normaliz', rays=[[1,3], [2,1]])
sage: HS = C.hilbert_series([1,1])
sage: HS.numerator()
t^5 + t^4 + t^3 + t^2 + 1
sage: HS.denominator().factor()
(t + 1) * (t - 1)^2 * (t^2 + 1) * (t^2 + t + 1)

sage: HS = C.hilbert_series([1,2])
sage: HS.numerator()
t^8 + t^6 + t^5 + t^3 + 1
sage: HS.denominator().factor()
(t + 1) * (t - 1)^2 * (t^2 + 1) * (t^6 + t^5 + t^4 + t^3 + t^2 + t + 1)
```

```
>>> from sage.all import *
>>> C = Polyhedron(backend='normaliz', rays=[[Integer(1),Integer(3)],␣
→[Integer(2),Integer(1)]])
>>> HS = C.hilbert_series([Integer(1),Integer(1)])
>>> HS.numerator()
t^5 + t^4 + t^3 + t^2 + 1
>>> HS.denominator().factor()
(t + 1) * (t - 1)^2 * (t^2 + 1) * (t^2 + t + 1)

>>> HS = C.hilbert_series([Integer(1),Integer(2)])
>>> HS.numerator()
t^8 + t^6 + t^5 + t^3 + 1
>>> HS.denominator().factor()
(t + 1) * (t - 1)^2 * (t^2 + 1) * (t^6 + t^5 + t^4 + t^3 + t^2 + t + 1)
```

Here is the magic square example form the Normaliz manual:

```
sage: eq = [[0,1,1,1,-1,-1,-1, 0, 0, 0],
....:       [0,1,1,1, 0, 0, 0,-1,-1,-1],
....:       [0,0,1,1,-1, 0, 0,-1, 0, 0],
....:       [0,1,0,1, 0,-1, 0, 0,-1, 0],
....:       [0,1,1,0, 0, 0,-1, 0, 0,-1],
....:       [0,0,1,1, 0,-1, 0, 0, 0,-1],
....:       [0,1,1,0, 0,-1, 0,-1, 0, 0]]
sage: magic_square = (Polyhedron(eqns=eq, backend='normaliz')
....:                 & Polyhedron(rays=identity_matrix(9).rows()))
sage: grading = [1,1,1,0,0,0,0,0,0]
sage: magic_square.hilbert_series(grading)
(t^6 + 2*t^3 + 1)/(-t^9 + 3*t^6 - 3*t^3 + 1)
```

```
>>> from sage.all import *
>>> eq = [[Integer(0),Integer(1),Integer(1),Integer(1),-Integer(1),-
↪Integer(1),-Integer(1), Integer(0), Integer(0), Integer(0)],
...       [Integer(0),Integer(1),Integer(1),Integer(1), Integer(0),␣
↪Integer(0), Integer(0),-Integer(1),-Integer(1),-Integer(1)],
...       [Integer(0),Integer(0),Integer(1),Integer(1),-Integer(1),␣
↪Integer(0), Integer(0),-Integer(1), Integer(0), Integer(0)],
...       [Integer(0),Integer(1),Integer(0),Integer(1), Integer(0),-
↪Integer(1), Integer(0), Integer(0),-Integer(1), Integer(0)],
...       [Integer(0),Integer(1),Integer(1),Integer(0), Integer(0),␣
↪Integer(0),-Integer(1), Integer(0), Integer(0),-Integer(1)],
...       [Integer(0),Integer(0),Integer(1),Integer(1), Integer(0),-
↪Integer(1), Integer(0), Integer(0), Integer(0),-Integer(1)],
...       [Integer(0),Integer(1),Integer(1),Integer(0), Integer(0),-
↪Integer(1), Integer(0),-Integer(1), Integer(0), Integer(0)]]
>>> magic_square = (Polyhedron(eqns=eq, backend='normaliz')
...                 & Polyhedron(rays=identity_matrix(Integer(9)).rows()))
>>> grading = [Integer(1),Integer(1),Integer(1),Integer(0),Integer(0),
↪Integer(0),Integer(0),Integer(0),Integer(0)]
>>> magic_square.hilbert_series(grading)
(t^6 + 2*t^3 + 1)/(-t^9 + 3*t^6 - 3*t^3 + 1)
```

See also

`ehrhart_series()`

integral_points(*threshold=10000*)

Return the integral points in the polyhedron.

Uses either the naive algorithm (iterate over a rectangular bounding box) or triangulation + Smith form.

INPUT:

- `threshold` – integer (default: 10000); use the naïve algorithm as long as the bounding box is smaller than this

OUTPUT:

The list of integral points in the polyhedron. If the polyhedron is not compact, a `ValueError` is raised.

EXAMPLES:

```
sage: Polyhedron(vertices=[(-1,-1), (1,0), (1,1), (0,1)],
....:            backend='normaliz').integral_points()
((-1, -1), (0, 0), (0, 1), (1, 0), (1, 1))

sage: simplex = Polyhedron([(1,2,3), (2,3,7), (-2,-3,-11)],
....:                      backend='normaliz')
sage: simplex.integral_points()
((-2, -3, -11), (0, 0, -2), (1, 2, 3), (2, 3, 7))
```

```
>>> from sage.all import *
>>> Polyhedron(vertices=[(-Integer(1),-Integer(1)), (Integer(1),Integer(0)),␣
↪(Integer(1),Integer(1)), (Integer(0),Integer(1))],
```

(continues on next page)

(continued from previous page)

```
...             backend='normaliz').integral_points()
((-1, -1), (0, 0), (0, 1), (1, 0), (1, 1))

>>> simplex = Polyhedron([(Integer(1),Integer(2),Integer(3)), (Integer(2),
↪Integer(3),Integer(7)), (-Integer(2),-Integer(3),-Integer(11))],
...                      backend='normaliz')
>>> simplex.integral_points()
((-2, -3, -11), (0, 0, -2), (1, 2, 3), (2, 3, 7))
```

The polyhedron need not be full-dimensional:

```
sage: simplex = Polyhedron([(1,2,3,5), (2,3,7,5), (-2,-3,-11,5)],
....:                      backend='normaliz')
sage: simplex.integral_points()
((-2, -3, -11, 5), (0, 0, -2, 5), (1, 2, 3, 5), (2, 3, 7, 5))

sage: point = Polyhedron([(2,3,7)],
....:                    backend='normaliz')
sage: point.integral_points()
((2, 3, 7),)

sage: empty = Polyhedron(backend='normaliz')
sage: empty.integral_points()
()
```

```
>>> from sage.all import *
>>> simplex = Polyhedron([(Integer(1),Integer(2),Integer(3),Integer(5)),␣
↪(Integer(2),Integer(3),Integer(7),Integer(5)), (-Integer(2),-Integer(3),-
↪Integer(11),Integer(5))],
...                      backend='normaliz')
>>> simplex.integral_points()
((-2, -3, -11, 5), (0, 0, -2, 5), (1, 2, 3, 5), (2, 3, 7, 5))

>>> point = Polyhedron([(Integer(2),Integer(3),Integer(7))],
...                    backend='normaliz')
>>> point.integral_points()
((2, 3, 7),)

>>> empty = Polyhedron(backend='normaliz')
>>> empty.integral_points()
()
```

Here is a simplex where the naive algorithm of running over all points in a rectangular bounding box no longer works fast enough:

```
sage: v = [(1,0,7,-1), (-2,-2,4,-3), (-1,-1,-1,4), (2,9,0,-5), (-2,-1,5,1)]
sage: simplex = Polyhedron(v, backend='normaliz'); simplex
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 5 vertices
sage: len(simplex.integral_points())
49
```

```
>>> from sage.all import *
>>> v = [(Integer(1),Integer(0),Integer(7),-Integer(1)), (-Integer(2),-
↪Integer(2),Integer(4),-Integer(3)), (-Integer(1),-Integer(1),-Integer(1),
↪Integer(4)), (Integer(2),Integer(9),Integer(0),-Integer(5)), (-Integer(2),-
↪Integer(1),Integer(5),Integer(1))]
>>> simplex = Polyhedron(v, backend='normaliz'); simplex
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 5 vertices
>>> len(simplex.integral_points())
49
```

A rather thin polytope for which the bounding box method would be a very bad idea (note this is a rational (non-lattice) polytope, so the other backends use the bounding box method):

```
sage: P = Polyhedron(vertices=((0, 0), (178933,37121))) + 1/1000*polytopes.
↪hypercube(2)
sage: P = Polyhedron(vertices=P.vertices_list(),
....:                backend='normaliz')
sage: len(P.integral_points())
434
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=((Integer(0), Integer(0)), (Integer(178933),
↪Integer(37121)))) + Integer(1)/Integer(1000)*polytopes.hypercube(Integer(2))
>>> P = Polyhedron(vertices=P.vertices_list(),
...                backend='normaliz')
>>> len(P.integral_points())
434
```

Finally, the 3-d reflexive polytope number 4078:

```
sage: v = [(1,0,0), (0,1,0), (0,0,1), (0,0,-1), (0,-2,1),
....:      (-1,2,-1), (-1,2,-2), (-1,1,-2), (-1,-1,2), (-1,-3,2)]
sage: P = Polyhedron(v, backend='normaliz')
sage: pts1 = P.integral_points()
sage: all(P.contains(p) for p in pts1)
True
sage: pts2 = LatticePolytope(v).points()                                    #␣
↪needs palp
sage: for p in pts1: p.set_immutable()
sage: set(pts1) == set(pts2)                                                #␣
↪needs palp
True

sage: timeit('Polyhedron(v, backend='normaliz').integral_points()')   # not␣
↪tested - random
625 loops, best of 3: 1.41 ms per loop
sage: timeit('LatticePolytope(v).points()')                           # not␣
↪tested - random
25 loops, best of 3: 17.2 ms per loop
```

```
>>> from sage.all import *
>>> v = [(Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0)), (Integer(0),Integer(0),Integer(1)), (Integer(0),Integer(0),-
```

(continues on next page)

(continued from previous page)

```
↪Integer(1)), (Integer(0),-Integer(2),Integer(1)),
...       (-Integer(1),Integer(2),-Integer(1)), (-Integer(1),Integer(2),-
↪Integer(2)), (-Integer(1),Integer(1),-Integer(2)), (-Integer(1),-Integer(1),
↪Integer(2)), (-Integer(1),-Integer(3),Integer(2))]
>>> P = Polyhedron(v, backend='normaliz')
>>> pts1 = P.integral_points()
>>> all(P.contains(p) for p in pts1)
True
>>> pts2 = LatticePolytope(v).points()                                    #␣
↪needs palp
>>> for p in pts1: p.set_immutable()
>>> set(pts1) == set(pts2)                                                #␣
↪needs palp
True

>>> timeit('Polyhedron(v, backend='normaliz').integral_points()')   # not␣
↪tested - random
625 loops, best of 3: 1.41 ms per loop
>>> timeit('LatticePolytope(v).points()')                           # not␣
↪tested - random
25 loops, best of 3: 17.2 ms per loop
```

integral_points_generators()

Return the integral points generators of the polyhedron.

Every integral point in the polyhedron can be written as a (unique) nonnegative linear combination of integral points contained in the three defining parts of the polyhedron: the integral points (the compact part), the recession cone, and the lineality space.

OUTPUT:

A tuple consisting of the integral points, the Hilbert basis of the recession cone, and an integral basis for the lineality space.

EXAMPLES:

Normaliz gives a nonnegative integer basis of the lineality space:

```
sage: P = Polyhedron(backend='normaliz', lines=[[2,2]])
sage: P.integral_points_generators()
(((0, 0),), (), ((1, 1),))
```

```
>>> from sage.all import *
>>> P = Polyhedron(backend='normaliz', lines=[[Integer(2),Integer(2)]])
>>> P.integral_points_generators()
(((0, 0),), (), ((1, 1),))
```

A recession cone generated by two rays:

```
sage: C = Polyhedron(backend='normaliz', rays=[[1,2], [2,1]])
sage: C.integral_points_generators()
(((0, 0),), ((1, 1), (1, 2), (2, 1)), ())
```

```
>>> from sage.all import *
>>> C = Polyhedron(backend='normaliz', rays=[[Integer(1),Integer(2)],
↪[Integer(2),Integer(1)]])
>>> C.integral_points_generators()
(((0, 0),), ((1, 1), (1, 2), (2, 1)), ())
```

Empty polyhedron:

```
sage: P = Polyhedron(backend='normaliz')
sage: P.integral_points_generators()
((), (), ())
```

```
>>> from sage.all import *
>>> P = Polyhedron(backend='normaliz')
>>> P.integral_points_generators()
((), (), ())
```

class sage.geometry.polyhedron.backend_normaliz.**Polyhedron_ZZ_normaliz**(*parent*, *Vrep*, *Hrep*, *normaliz_cone=None*, *normaliz_data=None*, *internal_base_ring=None*, ***kwds*)

Bases: *Polyhedron_QQ_normaliz*, *Polyhedron_ZZ*

Polyhedra over $\mathbf{Z}$ with normaliz.

INPUT:

- `Vrep` – list `[vertices, rays, lines]` or `None`
- `Hrep` – list `[ieqs, eqns]` or `None`

EXAMPLES:

```
sage: p = Polyhedron(vertices=[(0,0), (1,0), (0,1)],
....:                rays=[(1,1)], lines=[],
....:                backend='normaliz', base_ring=ZZ)
sage: TestSuite(p).run()
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices=[(Integer(0),Integer(0)), (Integer(1),Integer(0)),
↪(Integer(0),Integer(1))],
...                rays=[(Integer(1),Integer(1))], lines=[],
...                backend='normaliz', base_ring=ZZ)
>>> TestSuite(p).run()
```

class sage.geometry.polyhedron.backend_normaliz.**Polyhedron_normaliz**(*parent*, *Vrep*, *Hrep*, *normaliz_cone=None*, *normaliz_data=None*, *internal_base_ring=None*, ***kwds*)

Bases: `Polyhedron_base_number_field`

Polyhedra with normaliz.

INPUT:

- `parent` – *Polyhedra* the parent
- `Vrep` – list `[vertices, rays, lines]` or `None`; the V-representation of the polyhedron; if `None`, the polyhedron is determined by the H-representation
- `Hrep` – list `[ieqs, eqns]` or `None`; the H-representation of the polyhedron; if `None`, the polyhedron is determined by the V-representation
- `normaliz_cone` – a PyNormaliz wrapper of a normaliz cone

Only one of `Vrep`, `Hrep`, or `normaliz_cone` can be different from `None`.

EXAMPLES:

```
sage: p = Polyhedron(vertices=[(0,0), (1,0), (0,1)],
....:                 rays=[(1,1)], lines=[],
....:                 backend='normaliz')
sage: TestSuite(p).run()
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices=[(Integer(0),Integer(0)), (Integer(1),Integer(0)),␣
↪(Integer(0),Integer(1))],
...                 rays=[(Integer(1),Integer(1))], lines=[],
...                 backend='normaliz')
>>> TestSuite(p).run()
```

Two ways to get the full space:

```
sage: Polyhedron(eqns=[[0, 0, 0]], backend='normaliz')
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2␣
↪lines
sage: Polyhedron(ieqs=[[0, 0, 0]], backend='normaliz')
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2␣
↪lines
```

```
>>> from sage.all import *
>>> Polyhedron(eqns=[[Integer(0), Integer(0), Integer(0)]], backend='normaliz')
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2␣
↪lines
>>> Polyhedron(ieqs=[[Integer(0), Integer(0), Integer(0)]], backend='normaliz')
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2␣
↪lines
```

A lower-dimensional affine cone; we test that there are no mysterious inequalities coming in from the homogenization:

```
sage: P = Polyhedron(vertices=[(1, 1)], rays=[(0, 1)],
....:                 backend='normaliz')
sage: P.n_inequalities()
1
sage: P.equations()
(An equation (1, 0) x - 1 == 0,)
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[(Integer(1), Integer(1))], rays=[(Integer(0),␣
↪Integer(1))],
...                 backend='normaliz')
>>> P.n_inequalities()
1
>>> P.equations()
(An equation (1, 0) x - 1 == 0,)
```

The empty polyhedron:

```
sage: P = Polyhedron(ieqs=[[-2, 1, 1], [-3, -1, -1], [-4, 1, -2]],
....:                backend='normaliz')
sage: P
The empty polyhedron in QQ^2
sage: P.Vrepresentation()
()
sage: P.Hrepresentation()
(An equation -1 == 0,)
```

```
>>> from sage.all import *
>>> P = Polyhedron(ieqs=[[-Integer(2), Integer(1), Integer(1)], [-Integer(3), -
↪Integer(1), -Integer(1)], [-Integer(4), Integer(1), -Integer(2)]],
...                backend='normaliz')
>>> P
The empty polyhedron in QQ^2
>>> P.Vrepresentation()
()
>>> P.Hrepresentation()
(An equation -1 == 0,)
```

integral_hull()

Return the integral hull in the polyhedron.

This is a new polyhedron that is the convex hull of all integral points.

EXAMPLES:

Unbounded example from Normaliz manual, “a dull polyhedron”:

```
sage: P = Polyhedron(ieqs=[[1, 0, 2], [3, 0, -2], [3, 2, -2]],
....:                backend='normaliz')
sage: PI = P.integral_hull()
sage: P.plot(color='yellow') + PI.plot(color='green')                    #␣
↪needs sage.plot
Graphics object consisting of 10 graphics primitives
sage: PI.Vrepresentation()
(A vertex at (-1, 0),
 A vertex at (0, 1),
 A ray in the direction (1, 0))
```

```
>>> from sage.all import *
>>> P = Polyhedron(ieqs=[[Integer(1), Integer(0), Integer(2)], [Integer(3),␣
↪Integer(0), -Integer(2)], [Integer(3), Integer(2), -Integer(2)]],
```

(continues on next page)

(continued from previous page)

```
...                  backend='normaliz')
>>> PI = P.integral_hull()
>>> P.plot(color='yellow') + PI.plot(color='green')                        #␣
↪needs sage.plot
Graphics object consisting of 10 graphics primitives
>>> PI.Vrepresentation()
(A vertex at (-1, 0),
 A vertex at (0, 1),
 A ray in the direction (1, 0))
```

Nonpointed case:

```
sage: P = Polyhedron(vertices=[[1/2, 1/3]], rays=[[1, 1]],
....:                lines=[[-1, 1]], backend='normaliz')
sage: PI = P.integral_hull()
sage: PI.Vrepresentation()
(A vertex at (1, 0),
 A ray in the direction (1, 0),
 A line in the direction (1, -1))
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[[Integer(1)/Integer(2), Integer(1)/Integer(3)]],␣
↪rays=[[Integer(1), Integer(1)]],
...                lines=[[-Integer(1), Integer(1)]], backend='normaliz')
>>> PI = P.integral_hull()
>>> PI.Vrepresentation()
(A vertex at (1, 0),
 A ray in the direction (1, 0),
 A line in the direction (1, -1))
```

Empty polyhedron:

```
sage: P = Polyhedron(backend='normaliz')
sage: PI = P.integral_hull()
sage: PI.Vrepresentation()
()
```

```
>>> from sage.all import *
>>> P = Polyhedron(backend='normaliz')
>>> PI = P.integral_hull()
>>> PI.Vrepresentation()
()
```

2.7.6 The polymake backend for polyhedral computations

Note

This backend requires polymake. To install it, type `sage -i polymake` in the terminal.

AUTHORS:

- Matthias Köppe (2017-03): initial version

class sage.geometry.polyhedron.backend_polymake.**Polyhedron_QQ_polymake**(*parent*, *Vrep*, *Hrep*, *polymake_polytope=None*, ***kwds*)

Bases: *Polyhedron_polymake*, *Polyhedron_QQ*

Polyhedra over **Q** with polymake.

INPUT:

- Vrep – list [vertices, rays, lines] or None
- Hrep – list [ieqs, eqns] or None

EXAMPLES:

```
sage: p = Polyhedron(vertices=[(0,0),(1,0),(0,1)],                # optional -
↪jupymake
....:                rays=[(1,1)], lines=[],
....:                backend='polymake', base_ring=QQ)
sage: TestSuite(p).run()                                          # optional -
↪jupymake
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices=[(Integer(0),Integer(0)),(Integer(1),Integer(0)),
↪(Integer(0),Integer(1))],                          # optional - jupymake
...                rays=[(Integer(1),Integer(1))], lines=[],
...                backend='polymake', base_ring=QQ)
>>> TestSuite(p).run()                                          # optional -
↪jupymake
```

class sage.geometry.polyhedron.backend_polymake.**Polyhedron_ZZ_polymake**(*parent*, *Vrep*, *Hrep*, *polymake_polytope=None*, ***kwds*)

Bases: *Polyhedron_polymake*, *Polyhedron_ZZ*

Polyhedra over **Z** with polymake.

INPUT:

- Vrep – list [vertices, rays, lines] or None
- Hrep – list [ieqs, eqns] or None

EXAMPLES:

```
sage: p = Polyhedron(vertices=[(0,0),(1,0),(0,1)],                # optional -
↪jupymake
....:                rays=[(1,1)], lines=[],
....:                backend='polymake', base_ring=ZZ)
sage: TestSuite(p).run()                                          # optional -
↪jupymake
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices=[(Integer(0),Integer(0)),(Integer(1),Integer(0)),
↪(Integer(0),Integer(1))],                          # optional - jupymake
...                rays=[(Integer(1),Integer(1))], lines=[],
```

(continues on next page)

(continued from previous page)

```
...                    backend='polymake', base_ring=ZZ)
>>> TestSuite(p).run()                                           # optional -
↪jupymake
```

class sage.geometry.polyhedron.backend_polymake.**Polyhedron_polymake**(*parent*, *Vrep*, *Hrep*, *polymake_polytope=None*, ***kwds*)

Bases: *Polyhedron_base*

Polyhedra with polymake.

INPUT:

- `parent` – *Polyhedra* the parent
- `Vrep` – list `[vertices, rays, lines]` or `None`; the V-representation of the polyhedron. If `None`, the polyhedron is determined by the H-representation.
- `Hrep` – list `[ieqs, eqns]` or `None`; the H-representation of the polyhedron. If `None`, the polyhedron is determined by the V-representation.
- `polymake_polytope` – a polymake polytope object

Only one of `Vrep`, `Hrep`, or `polymake_polytope` can be different from `None`.

EXAMPLES:

```
sage: p = Polyhedron(vertices=[(0,0),(1,0),(0,1)], rays=[(1,1)],    # optional -
↪jupymake
....:                    lines=[], backend='polymake')
sage: TestSuite(p).run()                                             # optional -
↪jupymake
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices=[(Integer(0),Integer(0)),(Integer(1),Integer(0)),
↪(Integer(0),Integer(1))], rays=[(Integer(1),Integer(1))],    # optional -
↪jupymake
...                    lines=[], backend='polymake')
>>> TestSuite(p).run()                                           # optional -
↪jupymake
```

A lower-dimensional affine cone; we test that there are no mysterious inequalities coming in from the homogenization:

```
sage: P = Polyhedron(vertices=[(1, 1)], rays=[(0, 1)],              # optional -
↪jupymake
....:                    backend='polymake')
sage: P.n_inequalities()                                             # optional -
↪jupymake
1
sage: P.equations()                                                  # optional -
↪jupymake
(An equation (1, 0) x - 1 == 0,)
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[(Integer(1), Integer(1))], rays=[(Integer(0),␣
↪Integer(1))],              # optional - jupymake
...                  backend='polymake')
>>> P.n_inequalities()                                           # optional -␣
↪jupymake
1
>>> P.equations()                                                # optional -␣
↪jupymake
(An equation (1, 0) x - 1 == 0,)
```

The empty polyhedron:

```
sage: Polyhedron(eqns=[[1, 0, 0]], backend='polymake')           # optional -␣
↪jupymake
The empty polyhedron in QQ^2
```

```
>>> from sage.all import *
>>> Polyhedron(eqns=[[Integer(1), Integer(0), Integer(0)]], backend='polymake')  ␣
↪          # optional - jupymake
The empty polyhedron in QQ^2
```

It can also be obtained differently:

```
sage: # optional - jupymake
sage: P=Polyhedron(ieqs=[[-2, 1, 1], [-3, -1, -1], [-4, 1, -2]],
....:                  backend='polymake')
sage: P
The empty polyhedron in QQ^2
sage: P.Vrepresentation()
()
sage: P.Hrepresentation()
(An equation -1 == 0,)
```

```
>>> from sage.all import *
>>> # optional - jupymake
>>> P=Polyhedron(ieqs=[[-Integer(2), Integer(1), Integer(1)], [-Integer(3), -
↪Integer(1), -Integer(1)], [-Integer(4), Integer(1), -Integer(2)]],
...                  backend='polymake')
>>> P
The empty polyhedron in QQ^2
>>> P.Vrepresentation()
()
>>> P.Hrepresentation()
(An equation -1 == 0,)
```

The full polyhedron:

```
sage: Polyhedron(eqns=[[0, 0, 0]], backend='polymake')           # optional -␣
↪jupymake
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2␣
↪lines
sage: Polyhedron(ieqs=[[0, 0, 0]], backend='polymake')           # optional -␣
```

(continues on next page)

(continued from previous page)

```
↪jupymake
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2␣
↪lines
```

```
>>> from sage.all import *
>>> Polyhedron(eqns=[[Integer(0), Integer(0), Integer(0)]], backend='polymake')  ␣
↪          # optional - jupymake
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2␣
↪lines
>>> Polyhedron(ieqs=[[Integer(0), Integer(0), Integer(0)]], backend='polymake')  ␣
↪          # optional - jupymake
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2␣
↪lines
```

Quadratic fields work:

```
sage: V = polytopes.dodecahedron().vertices_list()                              #␣
↪needs sage.groups sage.rings.number_field
sage: Polyhedron(vertices=V, backend='polymake')        # optional - jupymake,␣
↪needs sage.groups sage.rings.number_field
A 3-dimensional polyhedron
 in (Number Field in sqrt5 with defining polynomial x^2 - 5
     with sqrt5 = 2.236067977499790?)^3
 defined as the convex hull of 20 vertices
```

```
>>> from sage.all import *
>>> V = polytopes.dodecahedron().vertices_list()                              #␣
↪needs sage.groups sage.rings.number_field
>>> Polyhedron(vertices=V, backend='polymake')        # optional - jupymake,␣
↪needs sage.groups sage.rings.number_field
A 3-dimensional polyhedron
 in (Number Field in sqrt5 with defining polynomial x^2 - 5
     with sqrt5 = 2.236067977499790?)^3
 defined as the convex hull of 20 vertices
```

2.7.7 The PPL (Parma Polyhedra Library) backend for polyhedral computations

class sage.geometry.polyhedron.backend_ppl.**Polyhedron_QQ_ppl**(*parent*, *Vrep*, *Hrep*, *ppl_polyhedron=None*, *mutable=False*, ***kwds*)

Bases: *Polyhedron_ppl*, *Polyhedron_QQ*

Polyhedra over **Q** with ppl.

INPUT:

- `Vrep` – list `[vertices, rays, lines]` or `None`
- `Hrep` – list `[ieqs, eqns]` or `None`

EXAMPLES:

```
sage: p = Polyhedron(vertices=[(0,0),(1,0),(0,1)], rays=[(1,1)], lines=[],
....:                 backend='ppl', base_ring=QQ)
sage: TestSuite(p).run()
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices=[(Integer(0),Integer(0)),(Integer(1),Integer(0)),
↪(Integer(0),Integer(1))], rays=[(Integer(1),Integer(1))], lines=[],
...                 backend='ppl', base_ring=QQ)
>>> TestSuite(p).run()
```

class sage.geometry.polyhedron.backend_ppl.**Polyhedron_ZZ_ppl**(*parent*, *Vrep*, *Hrep*, *ppl_polyhedron=None*, *mutable=False*, ***kwds*)

Bases: *Polyhedron_ppl*, *Polyhedron_ZZ*

Polyhedra over **Z** with ppl.

INPUT:

- Vrep – list [vertices, rays, lines] or None
- Hrep – list [ieqs, eqns] or None

EXAMPLES:

```
sage: p = Polyhedron(vertices=[(0,0),(1,0),(0,1)], rays=[(1,1)], lines=[],
....:                 backend='ppl', base_ring=ZZ)
sage: TestSuite(p).run()
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices=[(Integer(0),Integer(0)),(Integer(1),Integer(0)),
↪(Integer(0),Integer(1))], rays=[(Integer(1),Integer(1))], lines=[],
...                 backend='ppl', base_ring=ZZ)
>>> TestSuite(p).run()
```

class sage.geometry.polyhedron.backend_ppl.**Polyhedron_ppl**(*parent*, *Vrep*, *Hrep*, *ppl_polyhedron=None*, *mutable=False*, ***kwds*)

Bases: Polyhedron_mutable

Polyhedra with ppl.

INPUT:

- Vrep – list [vertices, rays, lines] or None
- Hrep – list [ieqs, eqns] or None

EXAMPLES:

```
sage: p = Polyhedron(vertices=[(0,0),(1,0),(0,1)], rays=[(1,1)], lines=[],␣
↪backend='ppl')
sage: TestSuite(p).run()
```

```
>>> from sage.all import *
>>> p = Polyhedron(vertices=[(Integer(0),Integer(0)),(Integer(1),Integer(0)),
```

(continues on next page)

(continued from previous page)

```
↪(Integer(0),Integer(1))], rays=[(Integer(1),Integer(1))], lines=[], backend='ppl
↪')
>>> TestSuite(p).run()
```

Hrepresentation(*index=None*)

Return the objects of the H-representation. Each entry is either an inequality or a equation.

INPUT:

- `index` – either an integer or `None`

OUTPUT:

The optional argument is an index running from `0` to `self.n_Hrepresentation()-1`. If present, the H-representation object at the given index will be returned. Without an argument, returns the list of all H-representation objects.

EXAMPLES:

```
sage: p = polytopes.hypercube(3)
sage: p.Hrepresentation(0)
An inequality (-1, 0, 0) x + 1 >= 0
sage: p.Hrepresentation(0) == p.Hrepresentation()[0]
True
```

```
>>> from sage.all import *
>>> p = polytopes.hypercube(Integer(3))
>>> p.Hrepresentation(Integer(0))
An inequality (-1, 0, 0) x + 1 >= 0
>>> p.Hrepresentation(Integer(0)) == p.Hrepresentation()[Integer(0)]
True
```

```
sage: P = p.parent()
sage: p = P._element_constructor_(p, mutable=True)
sage: p.Hrepresentation(0)
An inequality (0, 0, -1) x + 1 >= 0
sage: p._clear_cache()
sage: p.Hrepresentation(0)
An inequality (0, 0, -1) x + 1 >= 0
sage: TestSuite(p).run()
```

```
>>> from sage.all import *
>>> P = p.parent()
>>> p = P._element_constructor_(p, mutable=True)
>>> p.Hrepresentation(Integer(0))
An inequality (0, 0, -1) x + 1 >= 0
>>> p._clear_cache()
>>> p.Hrepresentation(Integer(0))
An inequality (0, 0, -1) x + 1 >= 0
>>> TestSuite(p).run()
```

Vrepresentation(*index=None*)

Return the objects of the V-representation. Each entry is either a vertex, a ray, or a line.

See *sage.geometry.polyhedron.constructor* for a definition of vertex/ray/line.

INPUT:

- `index` – either an integer or `None`

OUTPUT:

The optional argument is an index running from `0` to `self.n_Vrepresentation()-1`. If present, the V-representation object at the given index will be returned. Without an argument, returns the list of all V-representation objects.

EXAMPLES:

```
sage: p = polytopes.cube()
sage: p.Vrepresentation(0)
A vertex at (1, -1, -1)
```

```
>>> from sage.all import *
>>> p = polytopes.cube()
>>> p.Vrepresentation(Integer(0))
A vertex at (1, -1, -1)
```

```
sage: P = p.parent()
sage: p = P._element_constructor_(p, mutable=True)
sage: p.Vrepresentation(0)
A vertex at (-1, -1, -1)
sage: p._clear_cache()
sage: p.Vrepresentation(0)
A vertex at (-1, -1, -1)
sage: TestSuite(p).run()
```

```
>>> from sage.all import *
>>> P = p.parent()
>>> p = P._element_constructor_(p, mutable=True)
>>> p.Vrepresentation(Integer(0))
A vertex at (-1, -1, -1)
>>> p._clear_cache()
>>> p.Vrepresentation(Integer(0))
A vertex at (-1, -1, -1)
>>> TestSuite(p).run()
```

set_immutable()

Make this polyhedron immutable. This operation cannot be undone.

EXAMPLES:

```
sage: p = Polyhedron([[1, 1]], mutable=True)
sage: p.is_mutable()
True
sage: hasattr(p, "_Vrepresentation")
False
sage: p.set_immutable()
sage: hasattr(p, "_Vrepresentation")
True
```

```
>>> from sage.all import *
>>> p = Polyhedron([[Integer(1), Integer(1)]], mutable=True)
>>> p.is_mutable()
True
>>> hasattr(p, "_Vrepresentation")
False
>>> p.set_immutable()
>>> hasattr(p, "_Vrepresentation")
True
```

2.7.8 Double Description Algorithm for Cones

This module implements the double description algorithm for extremal vertex enumeration in a pointed cone following [FP1996]. With a little bit of preprocessing (see `double_description_inhomogeneous`) this defines a backend for polyhedral computations. But as far as this module is concerned, *inequality* always means without a constant term and the origin is always a point of the cone.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.double_description import StandardAlgorithm
sage: A = matrix(QQ, [(1,0,1), (0,1,1), (-1,-1,1)])
sage: alg = StandardAlgorithm(A);  alg
Pointed cone with inequalities
(1, 0, 1)
(0, 1, 1)
(-1, -1, 1)
sage: DD, _ = alg.initial_pair();  DD
Double description pair (A, R) defined by
    [ 1  0  1]        [ 2/3 -1/3 -1/3]
A = [ 0  1  1],   R = [-1/3  2/3 -1/3]
    [-1 -1  1]        [ 1/3  1/3  1/3]
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.double_description import StandardAlgorithm
>>> A = matrix(QQ, [(Integer(1),Integer(0),Integer(1)), (Integer(0),Integer(1),
↪Integer(1)), (-Integer(1),-Integer(1),Integer(1))])
>>> alg = StandardAlgorithm(A);  alg
Pointed cone with inequalities
(1, 0, 1)
(0, 1, 1)
(-1, -1, 1)
>>> DD, _ = alg.initial_pair();  DD
Double description pair (A, R) defined by
    [ 1  0  1]        [ 2/3 -1/3 -1/3]
A = [ 0  1  1],   R = [-1/3  2/3 -1/3]
    [-1 -1  1]        [ 1/3  1/3  1/3]
```

The implementation works over any exact field that is embedded in **R**, for example:

```
sage: from sage.geometry.polyhedron.double_description import StandardAlgorithm
sage: A = matrix(AA, [(1,0,1), (0,1,1), (-AA(2).sqrt(),-AA(3).sqrt(),1),          #␣
↪needs sage.rings.number_field
....:                 (-AA(3).sqrt(),-AA(2).sqrt(),1)])
```

(continues on next page)

(continued from previous page)

```
sage: alg = StandardAlgorithm(A)
sage: alg.run().R                                                      #␣
↪needs sage.rings.number_field
[(-0.4177376677004119?, 0.5822623322995881?, 0.4177376677004119?),
 (-0.2411809548974793?, -0.2411809548974793?, 0.2411809548974793?),
 (0.07665629029830300?, 0.07665629029830300?, 0.2411809548974793?),
 (0.5822623322995881?, -0.4177376677004119?, 0.4177376677004119?)]
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.double_description import StandardAlgorithm
>>> A = matrix(AA, [(Integer(1),Integer(0),Integer(1)), (Integer(0),Integer(1),
↪Integer(1)), (-AA(Integer(2)).sqrt(),-AA(Integer(3)).sqrt(),Integer(1)),
↪# needs sage.rings.number_field
...                 (-AA(Integer(3)).sqrt(),-AA(Integer(2)).sqrt(),Integer(1))])
>>> alg = StandardAlgorithm(A)
>>> alg.run().R                                                      #␣
↪needs sage.rings.number_field
[(-0.4177376677004119?, 0.5822623322995881?, 0.4177376677004119?),
 (-0.2411809548974793?, -0.2411809548974793?, 0.2411809548974793?),
 (0.07665629029830300?, 0.07665629029830300?, 0.2411809548974793?),
 (0.5822623322995881?, -0.4177376677004119?, 0.4177376677004119?)]
```

class sage.geometry.polyhedron.double_description.**DoubleDescriptionPair**(*problem*, *A_rows*, *R_cols*)

Bases: `object`

Base class for a double description pair (A, R).

> **Warning**
>
> You should use the *Problem.initial_pair()* or `Problem.run()` to generate double description pairs for a set of inequalities, and not generate `DoubleDescriptionPair` instances directly.

INPUT:

- `problem` – instance of *Problem*
- `A_rows` – list of row vectors of the matrix A; these encode the inequalities
- `R_cols` – list of column vectors of the matrix R; these encode the rays

R_by_sign(a)

Classify the rays into those that are positive, zero, and negative on a.

INPUT:

- `a` – vector; coefficient vector of a homogeneous inequality

OUTPUT:

A triple consisting of the rays (columns of R) that are positive, zero, and negative on a. In that order.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.double_description import
↪StandardAlgorithm
sage: A = matrix(QQ, [(1,0,1), (0,1,1), (-1,-1,1)])
sage: DD, _ = StandardAlgorithm(A).initial_pair()
sage: DD.R_by_sign(vector([1,-1,0]))
([(2/3, -1/3, 1/3)], [(-1/3, -1/3, 1/3)], [(-1/3, 2/3, 1/3)])
sage: DD.R_by_sign(vector([1,1,1]))
([(2/3, -1/3, 1/3), (-1/3, 2/3, 1/3)], [], [(-1/3, -1/3, 1/3)])
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.double_description import StandardAlgorithm
>>> A = matrix(QQ, [(Integer(1),Integer(0),Integer(1)), (Integer(0),
↪Integer(1),Integer(1)), (-Integer(1),-Integer(1),Integer(1))])
>>> DD, _ = StandardAlgorithm(A).initial_pair()
>>> DD.R_by_sign(vector([Integer(1),-Integer(1),Integer(0)]))
([(2/3, -1/3, 1/3)], [(-1/3, -1/3, 1/3)], [(-1/3, 2/3, 1/3)])
>>> DD.R_by_sign(vector([Integer(1),Integer(1),Integer(1)]))
([(2/3, -1/3, 1/3), (-1/3, 2/3, 1/3)], [], [(-1/3, -1/3, 1/3)])
```

are_adjacent(*r1*, *r2*)

Return whether the two rays are adjacent.

INPUT:

- r1, r2 – two rays

OUTPUT: boolean; whether the two rays are adjacent

EXAMPLES:

```
sage: from sage.geometry.polyhedron.double_description import
↪StandardAlgorithm
sage: A = matrix(QQ, [(0,1,0), (1,0,0), (0,-1,1), (-1,0,1)])
sage: DD = StandardAlgorithm(A).run()
sage: DD.are_adjacent(DD.R[0], DD.R[1])
True
sage: DD.are_adjacent(DD.R[0], DD.R[2])
True
sage: DD.are_adjacent(DD.R[0], DD.R[3])
False
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.double_description import StandardAlgorithm
>>> A = matrix(QQ, [(Integer(0),Integer(1),Integer(0)), (Integer(1),
↪Integer(0),Integer(0)), (Integer(0),-Integer(1),Integer(1)), (-Integer(1),
↪Integer(0),Integer(1))])
>>> DD = StandardAlgorithm(A).run()
>>> DD.are_adjacent(DD.R[Integer(0)], DD.R[Integer(1)])
True
>>> DD.are_adjacent(DD.R[Integer(0)], DD.R[Integer(2)])
True
>>> DD.are_adjacent(DD.R[Integer(0)], DD.R[Integer(3)])
False
```

cone()

Return the cone defined by A.

This method is for debugging only. Assumes that the base ring is $\mathbf{Q}$.

OUTPUT:

The cone defined by the inequalities as a *Polyhedron()*, using the PPL backend.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.double_description import␣
↪StandardAlgorithm
sage: A = matrix(QQ, [(1,0,1), (0,1,1), (-1,-1,1)])
sage: DD, _ = StandardAlgorithm(A).initial_pair()
sage: DD.cone().Hrepresentation()
(An inequality (-1, -1, 1) x + 0 >= 0,
 An inequality (0, 1, 1) x + 0 >= 0,
 An inequality (1, 0, 1) x + 0 >= 0)
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.double_description import StandardAlgorithm
>>> A = matrix(QQ, [(Integer(1),Integer(0),Integer(1)), (Integer(0),
↪Integer(1),Integer(1)), (-Integer(1),-Integer(1),Integer(1))])
>>> DD, _ = StandardAlgorithm(A).initial_pair()
>>> DD.cone().Hrepresentation()
(An inequality (-1, -1, 1) x + 0 >= 0,
 An inequality (0, 1, 1) x + 0 >= 0,
 An inequality (1, 0, 1) x + 0 >= 0)
```

dual()

Return the dual.

OUTPUT:

For the double description pair (A, R) this method returns the dual double description pair (R^T, A^T)

EXAMPLES:

```
sage: from sage.geometry.polyhedron.double_description import Problem
sage: A = matrix(QQ, [(0,1,0), (1,0,0), (0,-1,1), (-1,0,1)])
sage: DD, _ = Problem(A).initial_pair()
sage: DD
Double description pair (A, R) defined by
    [ 0  1  0]        [0 1 0]
A = [ 1  0  0],   R = [1 0 0]
    [ 0 -1  1]        [1 0 1]
sage: DD.dual()
 Double description pair (A, R) defined by
    [0 1 1]        [ 0  1  0]
A = [1 0 0],   R = [ 1  0 -1]
    [0 0 1]        [ 0  0  1]
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.double_description import Problem
>>> A = matrix(QQ, [(Integer(0),Integer(1),Integer(0)), (Integer(1),
```

(continues on next page)

(continued from previous page)

```
↪Integer(0),Integer(0)), (Integer(0),-Integer(1),Integer(1)), (-Integer(1),
↪Integer(0),Integer(1))])
>>> DD, _ = Problem(A).initial_pair()
>>> DD
Double description pair (A, R) defined by
    [ 0  1  0]        [0 1 0]
A = [ 1  0  0],   R = [1 0 0]
    [ 0 -1  1]        [1 0 1]
>>> DD.dual()
 Double description pair (A, R) defined by
    [0 1 1]        [ 0  1  0]
A = [1 0 0],   R = [ 1  0 -1]
    [0 0 1]        [ 0  0  1]
```

first_coordinate_plane()

Restrict to the first coordinate plane.

OUTPUT:

A new double description pair with the constraint $x_0 = 0$ added.

EXAMPLES:

```
sage: A = matrix([(1, 1), (-1, 1)])
sage: from sage.geometry.polyhedron.double_description import␣
↪StandardAlgorithm
sage: DD, _ = StandardAlgorithm(A).initial_pair()
sage: DD
Double description pair (A, R) defined by
A = [ 1  1],   R = [ 1/2 -1/2]
    [-1  1]        [ 1/2  1/2]
sage: DD.first_coordinate_plane()
Double description pair (A, R) defined by
    [ 1  1]
A = [-1  1],   R = [  0]
    [-1  0]        [1/2]
    [ 1  0]
```

```
>>> from sage.all import *
>>> A = matrix([(Integer(1), Integer(1)), (-Integer(1), Integer(1))])
>>> from sage.geometry.polyhedron.double_description import StandardAlgorithm
>>> DD, _ = StandardAlgorithm(A).initial_pair()
>>> DD
Double description pair (A, R) defined by
A = [ 1  1],   R = [ 1/2 -1/2]
    [-1  1]        [ 1/2  1/2]
>>> DD.first_coordinate_plane()
Double description pair (A, R) defined by
    [ 1  1]
A = [-1  1],   R = [  0]
    [-1  0]        [1/2]
    [ 1  0]
```

inner_product_matrix()

Return the inner product matrix between the rows of A and the columns of R.

OUTPUT:

A matrix over the base ring. There is one row for each row of A and one column for each column of R.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.double_description import␣
↪StandardAlgorithm
sage: A = matrix(QQ, [(1,0,1), (0,1,1), (-1,-1,1)])
sage: alg = StandardAlgorithm(A)
sage: DD, _ = alg.initial_pair()
sage: DD.inner_product_matrix()
[1 0 0]
[0 1 0]
[0 0 1]
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.double_description import StandardAlgorithm
>>> A = matrix(QQ, [(Integer(1),Integer(0),Integer(1)), (Integer(0),
↪Integer(1),Integer(1)), (-Integer(1),-Integer(1),Integer(1))])
>>> alg = StandardAlgorithm(A)
>>> DD, _ = alg.initial_pair()
>>> DD.inner_product_matrix()
[1 0 0]
[0 1 0]
[0 0 1]
```

is_extremal(*ray*)

Test whether the ray is extremal.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.double_description import␣
↪StandardAlgorithm
sage: A = matrix(QQ, [(0,1,0), (1,0,0), (0,-1,1), (-1,0,1)])
sage: DD = StandardAlgorithm(A).run()
sage: DD.is_extremal(DD.R[0])
True
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.double_description import StandardAlgorithm
>>> A = matrix(QQ, [(Integer(0),Integer(1),Integer(0)), (Integer(1),
↪Integer(0),Integer(0)), (Integer(0),-Integer(1),Integer(1)), (-Integer(1),
↪Integer(0),Integer(1))])
>>> DD = StandardAlgorithm(A).run()
>>> DD.is_extremal(DD.R[Integer(0)])
True
```

matrix_space(*nrows*, *ncols*)

Return a matrix space of size `nrows` and `ncols` over the base ring of `self`.

These matrix spaces are cached to avoid their creation in the very demanding `add_inequality()` and more precisely `are_adjacent()`.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.double_description import Problem
sage: A = matrix(QQ, [(1,0,1), (0,1,1), (-1,-1,1)])
sage: DD, _ = Problem(A).initial_pair()
sage: DD.matrix_space(2,2)
Full MatrixSpace of 2 by 2 dense matrices over Rational Field
sage: DD.matrix_space(3,2)
Full MatrixSpace of 3 by 2 dense matrices over Rational Field

sage: # needs sage.rings.number_field
sage: K.<sqrt2> = QuadraticField(2)
sage: A = matrix([[1,sqrt2],[2,0]])
sage: DD, _  = Problem(A).initial_pair()
sage: DD.matrix_space(1,2)
Full MatrixSpace of 1 by 2 dense matrices
 over Number Field in sqrt2 with defining polynomial x^2 - 2 with sqrt2 = 1.
↪414213562373095?
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.double_description import Problem
>>> A = matrix(QQ, [(Integer(1),Integer(0),Integer(1)), (Integer(0),
↪Integer(1),Integer(1)), (-Integer(1),-Integer(1),Integer(1))])
>>> DD, _ = Problem(A).initial_pair()
>>> DD.matrix_space(Integer(2),Integer(2))
Full MatrixSpace of 2 by 2 dense matrices over Rational Field
>>> DD.matrix_space(Integer(3),Integer(2))
Full MatrixSpace of 3 by 2 dense matrices over Rational Field

>>> # needs sage.rings.number_field
>>> K = QuadraticField(Integer(2), names=('sqrt2',)); (sqrt2,) = K._first_
↪ngens(1)
>>> A = matrix([[Integer(1),sqrt2],[Integer(2),Integer(0)]])
>>> DD, _  = Problem(A).initial_pair()
>>> DD.matrix_space(Integer(1),Integer(2))
Full MatrixSpace of 1 by 2 dense matrices
 over Number Field in sqrt2 with defining polynomial x^2 - 2 with sqrt2 = 1.
↪414213562373095?
```

verify()

Validate the double description pair.

This method used the PPL backend to check that the double description pair is valid. An assertion is triggered if it is not. Does nothing if the base ring is not **Q**.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.double_description import \
....:     DoubleDescriptionPair, Problem
sage: A = matrix(QQ, [(1,0,1), (0,1,1), (-1,-1,1)])
sage: alg = Problem(A)
sage: DD = DoubleDescriptionPair(alg,
....:     [(1, 0, 3), (0, 1, 1), (-1, -1, 1)],
....:     [(2/3, -1/3, 1/3), (-1/3, 2/3, 1/3), (-1/3, -1/3, 1/3)])
```

(continues on next page)

(continued from previous page)

```
sage: DD.verify()
Traceback (most recent call last):
...
    assert A_cone == R_cone
AssertionError
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.double_description import ␣
↪DoubleDescriptionPair, Problem
>>> A = matrix(QQ, [(Integer(1),Integer(0),Integer(1)), (Integer(0),
↪Integer(1),Integer(1)), (-Integer(1),-Integer(1),Integer(1))])
>>> alg = Problem(A)
>>> DD = DoubleDescriptionPair(alg,
...     [(Integer(1), Integer(0), Integer(3)), (Integer(0), Integer(1),␣
↪Integer(1)), (-Integer(1), -Integer(1), Integer(1))],
...     [(Integer(2)/Integer(3), -Integer(1)/Integer(3), Integer(1)/
↪Integer(3)), (-Integer(1)/Integer(3), Integer(2)/Integer(3), Integer(1)/
↪Integer(3)), (-Integer(1)/Integer(3), -Integer(1)/Integer(3), Integer(1)/
↪Integer(3))])
>>> DD.verify()
Traceback (most recent call last):
...
    assert A_cone == R_cone
AssertionError
```

zero_set (*ray*)

Return the zero set (active set) $Z(r)$.

INPUT:

- `ray` – a ray vector

OUTPUT: a set containing the inequality vectors that are zero on `ray`

EXAMPLES:

```
sage: from sage.geometry.polyhedron.double_description import Problem
sage: A = matrix(QQ, [(1,0,1), (0,1,1), (-1,-1,1)])
sage: DD, _ = Problem(A).initial_pair()
sage: r = DD.R[0];  r
(2/3, -1/3, 1/3)
sage: DD.zero_set(r)
{(-1, -1, 1), (0, 1, 1)}
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.double_description import Problem
>>> A = matrix(QQ, [(Integer(1),Integer(0),Integer(1)), (Integer(0),
↪Integer(1),Integer(1)), (-Integer(1),-Integer(1),Integer(1))])
>>> DD, _ = Problem(A).initial_pair()
>>> r = DD.R[Integer(0)];  r
(2/3, -1/3, 1/3)
>>> DD.zero_set(r)
{(-1, -1, 1), (0, 1, 1)}
```

class sage.geometry.polyhedron.double_description.**Problem**(A)

Bases: object

Base class for implementations of the double description algorithm.

It does not make sense to instantiate the base class directly, it just provides helpers for implementations.

INPUT:

- A – a matrix; the rows of the matrix are interpreted as homogeneous inequalities $Ax \geq 0$. Must have maximal rank.

A()

Return the rows of the defining matrix A.

OUTPUT: the matrix A whose rows are the inequalities

EXAMPLES:

```
sage: A = matrix([(1, 1), (-1, 1)])
sage: from sage.geometry.polyhedron.double_description import Problem
sage: Problem(A).A()
((1, 1), (-1, 1))
```

```
>>> from sage.all import *
>>> A = matrix([(Integer(1), Integer(1)), (-Integer(1), Integer(1))])
>>> from sage.geometry.polyhedron.double_description import Problem
>>> Problem(A).A()
((1, 1), (-1, 1))
```

A_matrix()

Return the defining matrix A.

OUTPUT: matrix whose rows are the inequalities

EXAMPLES:

```
sage: A = matrix([(1, 1), (-1, 1)])
sage: from sage.geometry.polyhedron.double_description import Problem
sage: Problem(A).A_matrix()
[ 1  1]
[-1  1]
```

```
>>> from sage.all import *
>>> A = matrix([(Integer(1), Integer(1)), (-Integer(1), Integer(1))])
>>> from sage.geometry.polyhedron.double_description import Problem
>>> Problem(A).A_matrix()
[ 1  1]
[-1  1]
```

base_ring()

Return the base field.

OUTPUT: a field

EXAMPLES:

```
sage: A = matrix(AA, [(1, 1), (-1, 1)])                                    #␣
↪needs sage.rings.number_field
sage: from sage.geometry.polyhedron.double_description import Problem
sage: Problem(A).base_ring()                                               #␣
↪needs sage.rings.number_field
Algebraic Real Field
```

```
>>> from sage.all import *
>>> A = matrix(AA, [(Integer(1), Integer(1)), (-Integer(1), Integer(1))])    ␣
↪                              # needs sage.rings.number_field
>>> from sage.geometry.polyhedron.double_description import Problem
>>> Problem(A).base_ring()                                                 #␣
↪needs sage.rings.number_field
Algebraic Real Field
```

dim()

Return the ambient space dimension.

OUTPUT: integer; the ambient space dimension of the cone

EXAMPLES:

```
sage: A = matrix(QQ, [(1, 1), (-1, 1)])
sage: from sage.geometry.polyhedron.double_description import Problem
sage: Problem(A).dim()
2
```

```
>>> from sage.all import *
>>> A = matrix(QQ, [(Integer(1), Integer(1)), (-Integer(1), Integer(1))])
>>> from sage.geometry.polyhedron.double_description import Problem
>>> Problem(A).dim()
2
```

initial_pair()

Return an initial double description pair.

Picks an initial set of rays by selecting a basis. This is probably the most efficient way to select the initial set.

INPUT:

- `pair_class` – subclass of *DoubleDescriptionPair*

OUTPUT:

A pair consisting of a *DoubleDescriptionPair* instance and the tuple of remaining unused inequalities.

EXAMPLES:

```
sage: A = matrix([(-1, 1), (-1, 2), (1/2, -1/2), (1/2, 2)])
sage: from sage.geometry.polyhedron.double_description import Problem
sage: DD, remaining = Problem(A).initial_pair()
sage: DD.verify()
sage: remaining
[(1/2, -1/2), (1/2, 2)]
```

```
>>> from sage.all import *
>>> A = matrix([(-Integer(1), Integer(1)), (-Integer(1), Integer(2)),␣
↪(Integer(1)/Integer(2), -Integer(1)/Integer(2)), (Integer(1)/Integer(2),␣
↪Integer(2))])
>>> from sage.geometry.polyhedron.double_description import Problem
>>> DD, remaining = Problem(A).initial_pair()
>>> DD.verify()
>>> remaining
[(1/2, -1/2), (1/2, 2)]
```

pair_class

alias of *DoubleDescriptionPair*

class sage.geometry.polyhedron.double_description.**StandardAlgorithm**(A)

Bases: *Problem*

Standard implementation of the double description algorithm.

See [FP1996] for the definition of the "Standard Algorithm".

EXAMPLES:

```
sage: A = matrix(QQ, [(1, 1), (-1, 1)])
sage: from sage.geometry.polyhedron.double_description import StandardAlgorithm
sage: DD = StandardAlgorithm(A).run()
sage: DD.R    # the extremal rays
[(1/2, 1/2), (-1/2, 1/2)]
```

```
>>> from sage.all import *
>>> A = matrix(QQ, [(Integer(1), Integer(1)), (-Integer(1), Integer(1))])
>>> from sage.geometry.polyhedron.double_description import StandardAlgorithm
>>> DD = StandardAlgorithm(A).run()
>>> DD.R    # the extremal rays
[(1/2, 1/2), (-1/2, 1/2)]
```

pair_class

alias of *StandardDoubleDescriptionPair*

run()

Run the Standard Algorithm.

OUTPUT:

A double description pair (A, R) of all inequalities as a *DoubleDescriptionPair*. By virtue of the double description algorithm, the columns of R are the extremal rays.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.double_description import␣
↪StandardAlgorithm
sage: A = matrix(QQ, [(0,1,0), (1,0,0), (0,-1,1), (-1,0,1)])
sage: StandardAlgorithm(A).run()
Double description pair (A, R) defined by
    [ 0  1  0]        [0 0 1 1]
A = [ 1  0  0],   R = [1 0 1 0]
```

(continues on next page)

(continued from previous page)

```
    [ 0 -1  1]        [1 1 1 1]
    [-1  0  1]
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.double_description import StandardAlgorithm
>>> A = matrix(QQ, [(Integer(0),Integer(1),Integer(0)), (Integer(1),
↪Integer(0),Integer(0)), (Integer(0),-Integer(1),Integer(1)), (-Integer(1),
↪Integer(0),Integer(1))])
>>> StandardAlgorithm(A).run()
Double description pair (A, R) defined by
    [ 0  1  0]        [0 0 1 1]
A = [ 1  0  0],   R = [1 0 1 0]
    [ 0 -1  1]        [1 1 1 1]
    [-1  0  1]
```

class sage.geometry.polyhedron.double_description.**StandardDoubleDescriptionPair**(*problem*, *A_rows*, *R_cols*)

Bases: *DoubleDescriptionPair*

Double description pair for the "Standard Algorithm".

See *StandardAlgorithm*.

add_inequality(*a*)

Add the inequality a to the matrix A of the double description.

INPUT:

- a – vector; an inequality

EXAMPLES:

```
sage: A = matrix([(-1, 1, 0), (-1, 2, 1), (1/2, -1/2, -1)])
sage: from sage.geometry.polyhedron.double_description import␣
↪StandardAlgorithm
sage: DD, _ = StandardAlgorithm(A).initial_pair()
sage: DD.add_inequality(vector([1,0,0]))
sage: DD
Double description pair (A, R) defined by
    [  -1    1    0]        [   1    1    0    0]
A = [  -1    2    1],   R = [   1    1    1    1]
    [ 1/2 -1/2   -1]        [   0   -1 -1/2   -2]
    [   1    0    0]
```

```
>>> from sage.all import *
>>> A = matrix([(-Integer(1), Integer(1), Integer(0)), (-Integer(1),␣
↪Integer(2), Integer(1)), (Integer(1)/Integer(2), -Integer(1)/Integer(2), -
↪Integer(1))])
>>> from sage.geometry.polyhedron.double_description import StandardAlgorithm
>>> DD, _ = StandardAlgorithm(A).initial_pair()
>>> DD.add_inequality(vector([Integer(1),Integer(0),Integer(0)]))
>>> DD
Double description pair (A, R) defined by
```

(continues on next page)

(continued from previous page)

```
    [  -1    1    0]        [   1    1    0    0]
A = [  -1    2    1],   R = [   1    1    1    1]
    [ 1/2 -1/2   -1]        [   0   -1 -1/2   -2]
    [   1    0    0]
```

sage.geometry.polyhedron.double_description.**random_inequalities**(d, n)

Random collections of inequalities for testing purposes.

INPUT:

- d – integer; the dimension
- n – integer; the number of random inequalities to generate

OUTPUT: a random set of inequalities as a *StandardAlgorithm* instance

EXAMPLES:

```
sage: from sage.geometry.polyhedron.double_description import random_inequalities
sage: P = random_inequalities(5, 10)
sage: P.run().verify()
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.double_description import random_inequalities
>>> P = random_inequalities(Integer(5), Integer(10))
>>> P.run().verify()
```

2.7.9 Double Description for Arbitrary Polyhedra

This module is part of the python backend for polyhedra. It uses the double description method for cones *double_description* to find minimal H/V-representations of polyhedra. The latter works with cones only. This is sufficient to treat general polyhedra by the following construction: Any polyhedron can be embedded in one dimension higher in the hyperplane $(1, *, \dots, *)$. The cone over the embedded polyhedron will be called the *homogenized cone* in the following. Conversely, intersecting the homogenized cone with the hyperplane $x_0 = 1$ gives you back the original polyhedron.

While slower than specialized C/C++ implementations, the implementation is general and works with any field in Sage that allows you to define polyhedra.

Note

If you just want polyhedra over arbitrary fields then you should just use the *Polyhedron()* constructor.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.double_description_inhomogeneous \
....:     import Hrep2Vrep, Vrep2Hrep
sage: Hrep2Vrep(QQ, 2, [(1,2,3), (2,4,3)], [])
[-1/2|-1/2  1/2|]
[   0| 2/3 -1/3|]
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.double_description_inhomogeneous     import␣
↪Hrep2Vrep, Vrep2Hrep
```

(continues on next page)

(continued from previous page)

```
>>> Hrep2Vrep(QQ, Integer(2), [(Integer(1),Integer(2),Integer(3)), (Integer(2),
↪Integer(4),Integer(3))], [])
[-1/2|-1/2  1/2|]
[   0| 2/3 -1/3|]
```

Note that the columns of the printed matrix are the vertices, rays, and lines of the minimal V-representation. Dually, the rows of the following are the inequalities and equations:

```
sage: Vrep2Hrep(QQ, 2, [(-1/2,0)], [(-1/2,2/3), (1/2,-1/3)], [])
[1 2 3]
[2 4 3]
[-----]
```

```
>>> from sage.all import *
>>> Vrep2Hrep(QQ, Integer(2), [(-Integer(1)/Integer(2),Integer(0))], [(-Integer(1)/
↪Integer(2),Integer(2)/Integer(3)), (Integer(1)/Integer(2),-Integer(1)/Integer(3))],␣
↪[])
[1 2 3]
[2 4 3]
[-----]
```

class sage.geometry.polyhedron.double_description_inhomogeneous.**Hrep2Vrep**(*base_ring*, *dim*, *inequalities*, *equations*)

Bases: *PivotedInequalities*

Convert H-representation to a minimal V-representation.

INPUT:

- `base_ring` – a field
- `dim` – integer; the ambient space dimension
- `inequalities` – list of inequalities; each inequality is given as constant term, `dim` coefficients
- `equations` – list of equations; same notation as for inequalities

EXAMPLES:

```
sage: from sage.geometry.polyhedron.double_description_inhomogeneous import␣
↪Hrep2Vrep
sage: Hrep2Vrep(QQ, 2, [(1,2,3), (2,4,3)], [])
[-1/2|-1/2  1/2|]
[   0| 2/3 -1/3|]
sage: Hrep2Vrep(QQ, 2, [(1,2,3), (2,-2,-3)], [])
[   1 -1/2||    1]
[   0    0||-2/3]
sage: Hrep2Vrep(QQ, 2, [(1,2,3), (2,2,3)], [])
[-1/2| 1/2|    1]
[   0|    0|-2/3]
sage: Hrep2Vrep(QQ, 2, [(8,7,-2), (1,-4,3), (4,-3,-1)], [])
[ 1  0 -2||]
[ 1  4 -3||]
sage: Hrep2Vrep(QQ, 2, [(1,2,3), (2,4,3), (5,-1,-2)], [])
```

(continues on next page)

(continued from previous page)

```
[-19/5  -1/2| 2/33  1/11||]
[ 22/5     0|-1/33 -2/33||]
sage: Hrep2Vrep(QQ, 2, [(0,2,3), (0,4,3), (0,-1,-2)], [])
[   0| 1/2  1/3||]
[   0|-1/3 -1/6||]
sage: Hrep2Vrep(QQ, 2, [], [(1,2,3), (7,8,9)])
[-2||]
[ 1||]
sage: Hrep2Vrep(QQ, 2, [(1,0,0)], [])     # universe
[0||1 0]
[0||0 1]
sage: Hrep2Vrep(QQ, 2, [(-1,0,0)], [])    # empty
[]
sage: Hrep2Vrep(QQ, 2, [], [])    # universe
[0||1 0]
[0||0 1]
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.double_description_inhomogeneous import␣
↪Hrep2Vrep
>>> Hrep2Vrep(QQ, Integer(2), [(Integer(1),Integer(2),Integer(3)), (Integer(2),
↪Integer(4),Integer(3))], [])
[-1/2|-1/2  1/2|]
[   0| 2/3 -1/3|]
>>> Hrep2Vrep(QQ, Integer(2), [(Integer(1),Integer(2),Integer(3)), (Integer(2),-
↪Integer(2),-Integer(3))], [])
[   1 -1/2||   1]
[   0    0||-2/3]
>>> Hrep2Vrep(QQ, Integer(2), [(Integer(1),Integer(2),Integer(3)), (Integer(2),
↪Integer(2),Integer(3))], [])
[-1/2| 1/2|   1]
[   0|   0|-2/3]
>>> Hrep2Vrep(QQ, Integer(2), [(Integer(8),Integer(7),-Integer(2)), (Integer(1),-
↪Integer(4),Integer(3)), (Integer(4),-Integer(3),-Integer(1))], [])
[ 1  0 -2||]
[ 1  4 -3||]
>>> Hrep2Vrep(QQ, Integer(2), [(Integer(1),Integer(2),Integer(3)), (Integer(2),
↪Integer(4),Integer(3)), (Integer(5),-Integer(1),-Integer(2))], [])
[-19/5  -1/2| 2/33  1/11||]
[ 22/5     0|-1/33 -2/33||]
>>> Hrep2Vrep(QQ, Integer(2), [(Integer(0),Integer(2),Integer(3)), (Integer(0),
↪Integer(4),Integer(3)), (Integer(0),-Integer(1),-Integer(2))], [])
[   0| 1/2  1/3||]
[   0|-1/3 -1/6||]
>>> Hrep2Vrep(QQ, Integer(2), [], [(Integer(1),Integer(2),Integer(3)),␣
↪(Integer(7),Integer(8),Integer(9))])
[-2||]
[ 1||]
>>> Hrep2Vrep(QQ, Integer(2), [(Integer(1),Integer(0),Integer(0))], [])     #␣
↪universe
[0||1 0]
[0||0 1]
```

(continues on next page)

(continued from previous page)

```
>>> Hrep2Vrep(QQ, Integer(2), [(-Integer(1),Integer(0),Integer(0))], [])    # empty
[]
>>> Hrep2Vrep(QQ, Integer(2), [], [])    # universe
[0||1 0]
[0||0 1]
```

verify (*inequalities*, *equations*)

Compare result to PPL if the base ring is QQ.

This method is for debugging purposes and compares the computation with another backend if available.

INPUT:

- `inequalities`, `equations` – see *Hrep2Vrep*

EXAMPLES:

```
sage: from sage.geometry.polyhedron.double_description_inhomogeneous import␣
↪Hrep2Vrep
sage: H = Hrep2Vrep(QQ, 1, [(1,2)], [])
sage: H.verify([(1,2)], [])
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.double_description_inhomogeneous import␣
↪Hrep2Vrep
>>> H = Hrep2Vrep(QQ, Integer(1), [(Integer(1),Integer(2))], [])
>>> H.verify([(Integer(1),Integer(2))], [])
```

class sage.geometry.polyhedron.double_description_inhomogeneous.**PivotedInequalities** (*base_ring*, *dim*)

Bases: `SageObject`

Base class for inequalities that may contain linear subspaces.

INPUT:

- `base_ring` – a field
- `dim` – integer; the ambient space dimension

EXAMPLES:

```
sage: from sage.geometry.polyhedron.double_description_inhomogeneous                .
↪...:     import PivotedInequalities
sage: piv = PivotedInequalities(QQ, 2)
sage: piv._pivot_inequalities(matrix([(1,1,3), (5,5,7)]))
[1 3]
[5 7]
sage: piv._pivots
(0, 2)
sage: piv._linear_subspace
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[ 1 -1  0]
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.double_description_inhomogeneous ␣
↪Ellipsis.:     import PivotedInequalities
>>> piv = PivotedInequalities(QQ, Integer(2))
>>> piv._pivot_inequalities(matrix([(Integer(1),Integer(1),Integer(3)),␣
↪(Integer(5),Integer(5),Integer(7))]))
[1 3]
[5 7]
>>> piv._pivots
(0, 2)
>>> piv._linear_subspace
Free module of degree 3 and rank 1 over Integer Ring
Echelon basis matrix:
[ 1 -1  0]
```

class sage.geometry.polyhedron.double_description_inhomogeneous.**Vrep2Hrep**(*base_ring*, *dim*, *vertices*, *rays*, *lines*)

Bases: *PivotedInequalities*

Convert V-representation to a minimal H-representation.

INPUT:

- `base_ring` – a field
- `dim` – integer; the ambient space dimension
- `vertices` – list of vertices; each vertex is given as list of `dim` coordinates
- `rays` – list of rays; each ray is given as list of `dim` coordinates, not all zero
- `lines` – list of line generators; each line is given as list of `dim` coordinates, not all zero

EXAMPLES:

```
sage: from sage.geometry.polyhedron.double_description_inhomogeneous import␣
↪Vrep2Hrep
sage: Vrep2Hrep(QQ, 2, [(-1/2,0)], [(-1/2,2/3), (1/2,-1/3)], [])
[1 2 3]
[2 4 3]
[-----]

sage: Vrep2Hrep(QQ, 2, [(1,0), (-1/2,0)], [], [(1,-2/3)])
[ 1/3  2/3    1]
[ 2/3 -2/3   -1]
[--------------]

sage: Vrep2Hrep(QQ, 2, [(-1/2,0)], [(1/2,0)], [(1,-2/3)])
[1 2 3]
[-----]

sage: Vrep2Hrep(QQ, 2, [(1,1), (0,4), (-2,-3)], [], [])
[ 8/13  7/13 -2/13]
[ 1/13 -4/13  3/13]
[ 4/13 -3/13 -1/13]
```

(continues on next page)

(continued from previous page)

```
[-----------------]

sage: Vrep2Hrep(QQ, 2, [(-19/5,22/5), (-1/2,0)], [(2/33,-1/33), (1/11,-2/33)], [])
[10/11 -2/11 -4/11]
[ 66/5 132/5  99/5]
[ 2/11  4/11  6/11]
[-----------------]

sage: Vrep2Hrep(QQ, 2, [(0,0)], [(1/2,-1/3), (1/3,-1/6)], [])
[  0  -6 -12]
[  0  12  18]
[-----------]

sage: Vrep2Hrep(QQ, 2, [(-1/2,0)], [], [(1,-2/3)])
[-----]
[1 2 3]

sage: Vrep2Hrep(QQ, 2, [(-1/2,0)], [], [(1,-2/3), (1,0)])
[]
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.double_description_inhomogeneous import␣
↪Vrep2Hrep
>>> Vrep2Hrep(QQ, Integer(2), [(-Integer(1)/Integer(2),Integer(0))], [(-
↪Integer(1)/Integer(2),Integer(2)/Integer(3)), (Integer(1)/Integer(2),-
↪Integer(1)/Integer(3))], [])
[1 2 3]
[2 4 3]
[-----]

>>> Vrep2Hrep(QQ, Integer(2), [(Integer(1),Integer(0)), (-Integer(1)/Integer(2),
↪Integer(0))], [], [(Integer(1),-Integer(2)/Integer(3))])
[ 1/3  2/3    1]
[ 2/3 -2/3   -1]
[--------------]

>>> Vrep2Hrep(QQ, Integer(2), [(-Integer(1)/Integer(2),Integer(0))], [(Integer(1)/
↪Integer(2),Integer(0))], [(Integer(1),-Integer(2)/Integer(3))])
[1 2 3]
[-----]

>>> Vrep2Hrep(QQ, Integer(2), [(Integer(1),Integer(1)), (Integer(0),Integer(4)),␣
↪(-Integer(2),-Integer(3))], [], [])
[ 8/13  7/13 -2/13]
[ 1/13 -4/13  3/13]
[ 4/13 -3/13 -1/13]
[-----------------]

>>> Vrep2Hrep(QQ, Integer(2), [(-Integer(19)/Integer(5),Integer(22)/Integer(5)),␣
↪(-Integer(1)/Integer(2),Integer(0))], [(Integer(2)/Integer(33),-Integer(1)/
↪Integer(33)), (Integer(1)/Integer(11),-Integer(2)/Integer(33))], [])
[10/11 -2/11 -4/11]
```

(continues on next page)

(continued from previous page)

```
[ 66/5 132/5  99/5]
[ 2/11  4/11  6/11]
[-----------------]

>>> Vrep2Hrep(QQ, Integer(2), [(Integer(0),Integer(0))], [(Integer(1)/Integer(2),-
↪Integer(1)/Integer(3)), (Integer(1)/Integer(3),-Integer(1)/Integer(6))], [])
[  0  -6 -12]
[  0  12  18]
[-----------]

>>> Vrep2Hrep(QQ, Integer(2), [(-Integer(1)/Integer(2),Integer(0))], [],
↪[(Integer(1),-Integer(2)/Integer(3))])
[-----]
[1 2 3]

>>> Vrep2Hrep(QQ, Integer(2), [(-Integer(1)/Integer(2),Integer(0))], [],
↪[(Integer(1),-Integer(2)/Integer(3)), (Integer(1),Integer(0))])
[]
```

verify(*vertices*, *rays*, *lines*)

Compare result to PPL if the base ring is QQ.

This method is for debugging purposes and compares the computation with another backend if available.

INPUT:

- `vertices`, `rays`, `lines` – see *Vrep2Hrep*

EXAMPLES:

```
sage: from sage.geometry.polyhedron.double_description_inhomogeneous import
↪Vrep2Hrep
sage: vertices = [(-1/2,0)]
sage: rays = [(-1/2,2/3), (1/2,-1/3)]
sage: lines = []
sage: V2H = Vrep2Hrep(QQ, 2, vertices, rays, lines)
sage: V2H.verify(vertices, rays, lines)
```

```
>>> from sage.all import *
>>> from sage.geometry.polyhedron.double_description_inhomogeneous import
↪Vrep2Hrep
>>> vertices = [(-Integer(1)/Integer(2),Integer(0))]
>>> rays = [(-Integer(1)/Integer(2),Integer(2)/Integer(3)), (Integer(1)/
↪Integer(2),-Integer(1)/Integer(3))]
>>> lines = []
>>> V2H = Vrep2Hrep(QQ, Integer(2), vertices, rays, lines)
>>> V2H.verify(vertices, rays, lines)
```

CHAPTER

THREE

TRIANGULATIONS

3.1 Triangulations of a point configuration

A point configuration is a finite set of points in Euclidean space or, more generally, in projective space. A triangulation is a simplicial decomposition of the convex hull of a given point configuration such that all vertices of the simplices end up lying on points of the configuration. That is, there are no new vertices apart from the initial points.

Note that points that are not vertices of the convex hull need not be used in the triangulation. A triangulation that does make use of all points of the configuration is called fine, and you can restrict yourself to such triangulations if you want. See `PointConfiguration` and `restrict_to_fine_triangulations()` for more details.

Finding a single triangulation and listing all connected triangulations is implemented natively in this package. However, for more advanced options [TOPCOM] needs to be installed; see topcom: Compute triangulations of point configurations and oriented matroids.

Note

TOPCOM and the internal algorithms tend to enumerate triangulations in a different order. This is why we always explicitly specify the engine as `engine='topcom'` or `engine='internal'` in the doctests. In your own applications, you do not need to specify the engine. By default, TOPCOM is used if it is available and the internal algorithms are used otherwise.

EXAMPLES:

First, we select the internal implementation for enumerating triangulations:

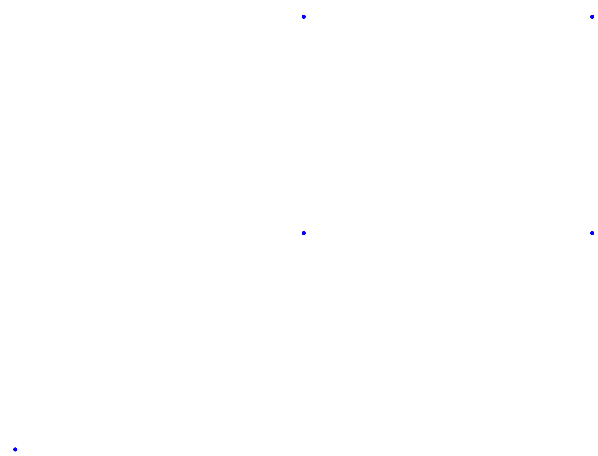
```
sage: PointConfiguration.set_engine('internal')   # to make doctests independent of␣
↪TOPCOM
```

```
>>> from sage.all import *
>>> PointConfiguration.set_engine('internal')   # to make doctests independent of␣
↪TOPCOM
```

A 2-dimensional point configuration:

```
sage: p = PointConfiguration([[0,0], [0,1], [1,0], [1,1], [-1,-1]]); p
A point configuration in affine 2-space over Integer Ring consisting
of 5 points. The triangulations of this point configuration are
assumed to be connected, not necessarily fine, not necessarily regular.
```

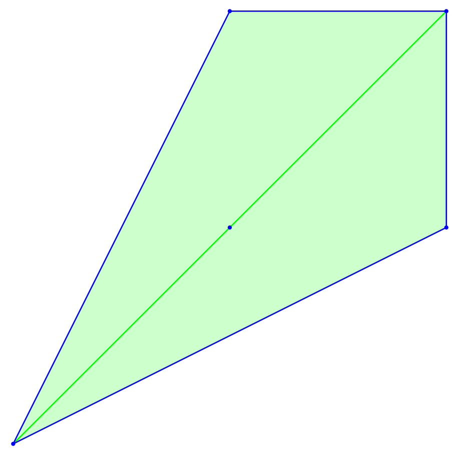
```
>>> from sage.all import *
>>> p = PointConfiguration([[Integer(0),Integer(0)], [Integer(0),Integer(1)],␣
↪[Integer(1),Integer(0)], [Integer(1),Integer(1)], [-Integer(1),-Integer(1)]]); p
A point configuration in affine 2-space over Integer Ring consisting
of 5 points. The triangulations of this point configuration are
assumed to be connected, not necessarily fine, not necessarily regular.
```



A triangulation of it:

```
sage: t = p.triangulate(); t  # a single triangulation
(<1,3,4>, <2,3,4>)
sage: len(t)
2
sage: t[0]
(1, 3, 4)
sage: t[1]
(2, 3, 4)
sage: list(t)
[(1, 3, 4), (2, 3, 4)]
sage: t.plot(axes=False)                                                    #␣
↪needs sage.plot
Graphics object consisting of 12 graphics primitives
```

```
>>> from sage.all import *
>>> t = p.triangulate(); t  # a single triangulation
(<1,3,4>, <2,3,4>)
>>> len(t)
2
>>> t[Integer(0)]
(1, 3, 4)
>>> t[Integer(1)]
(2, 3, 4)
>>> list(t)
[(1, 3, 4), (2, 3, 4)]
>>> t.plot(axes=False)                                                  # needs␣
↪sage.plot
Graphics object consisting of 12 graphics primitives
```



List triangulations of it:

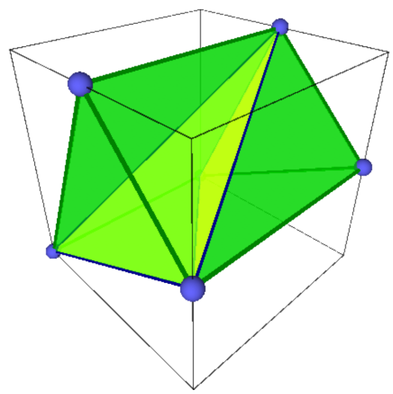
```
sage: list(p.triangulations())
[(<1,3,4>, <2,3,4>),
 (<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>),
 (<1,2,3>, <1,2,4>),
 (<0,1,2>, <0,1,4>, <0,2,4>, <1,2,3>)]
sage: p_fine = p.restrict_to_fine_triangulations(); p_fine
A point configuration in affine 2-space over Integer Ring consisting
of 5 points. The triangulations of this point configuration are
assumed to be connected, fine, not necessarily regular.
sage: list(p_fine.triangulations())
[(<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>),
 (<0,1,2>, <0,1,4>, <0,2,4>, <1,2,3>)]
```

```
>>> from sage.all import *
>>> list(p.triangulations())
[(<1,3,4>, <2,3,4>),
 (<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>),
 (<1,2,3>, <1,2,4>),
 (<0,1,2>, <0,1,4>, <0,2,4>, <1,2,3>)]
>>> p_fine = p.restrict_to_fine_triangulations(); p_fine
A point configuration in affine 2-space over Integer Ring consisting
of 5 points. The triangulations of this point configuration are
assumed to be connected, fine, not necessarily regular.
>>> list(p_fine.triangulations())
[(<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>),
 (<0,1,2>, <0,1,4>, <0,2,4>, <1,2,3>)]
```

A 3-dimensional point configuration:

```
sage: p = [[0,-1,-1], [0,0,1], [0,1,0], [1,-1,-1], [1,0,1], [1,1,0]]
sage: points = PointConfiguration(p)
sage: triang = points.triangulate()
sage: triang.plot(axes=False)                                          # needs␣
↪sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> p = [[Integer(0),-Integer(1),-Integer(1)], [Integer(0),Integer(0),Integer(1)],
↪[Integer(0),Integer(1),Integer(0)], [Integer(1),-Integer(1),-Integer(1)],
↪[Integer(1),Integer(0),Integer(1)], [Integer(1),Integer(1),Integer(0)]]
>>> points = PointConfiguration(p)
>>> triang = points.triangulate()
>>> triang.plot(axes=False)                                          # needs
↪sage.plot
Graphics3d Object
```



The standard example of a non-regular triangulation (requires TOPCOM):

```
sage: # optional - topcom
sage: PointConfiguration.set_engine('topcom')
sage: p = PointConfiguration([[-1,-5/9], [0,10/9], [1,-5/9],
....:                          [-2,-10/9], [0,20/9], [2,-10/9]])
sage: p_regular = p.restrict_to_regular_triangulations(True)
sage: regular = p_regular.triangulations_list()
sage: p_nonregular = p.restrict_to_regular_triangulations(False)
sage: nonregular = p_nonregular.triangulations_list()
sage: len(regular)
16
sage: len(nonregular)
2
sage: nonregular[0].plot(aspect_ratio=1, axes=False)                   # needs
↪sage.plot
Graphics object consisting of 25 graphics primitives
sage: PointConfiguration.set_engine('internal')   # to make doctests independent of
↪TOPCOM
```

```
>>> from sage.all import *
>>> # optional - topcom
>>> PointConfiguration.set_engine('topcom')
>>> p = PointConfiguration([[-Integer(1),-Integer(5)/Integer(9)], [Integer(0),
↪Integer(10)/Integer(9)], [Integer(1),-Integer(5)/Integer(9)],
...                          [-Integer(2),-Integer(10)/Integer(9)], [Integer(0),
↪Integer(20)/Integer(9)], [Integer(2),-Integer(10)/Integer(9)]])
```

(continues on next page)

(continued from previous page)

```
>>> p_regular = p.restrict_to_regular_triangulations(True)
>>> regular = p_regular.triangulations_list()
>>> p_nonregular = p.restrict_to_regular_triangulations(False)
>>> nonregular = p_nonregular.triangulations_list()
>>> len(regular)
16
>>> len(nonregular)
2
>>> nonregular[Integer(0)].plot(aspect_ratio=Integer(1), axes=False)
↪        # needs sage.plot
Graphics object consisting of 25 graphics primitives
>>> PointConfiguration.set_engine('internal')   # to make doctests independent of
↪TOPCOM
```

Note that the points need not be in general position. That is, the points may lie in a hyperplane and the linear dependencies will be removed before passing the data to TOPCOM which cannot handle it:

```
sage: points = [[0,0,0,1], [0,3,0,1], [3,0,0,1], [0,0,1,1],
....:           [0,3,1,1], [3,0,1,1], [1,1,2,1]]
sage: points = [p + [1,2,3] for p in points]
sage: pc = PointConfiguration(points)
sage: pc.ambient_dim()
7
sage: pc.dim()
3
sage: pc.triangulate()
(<0,1,2,6>, <0,1,3,6>, <0,2,3,6>, <1,2,4,6>, <1,3,4,6>, <2,3,5,6>, <2,4,5,6>)
sage: _ in pc.triangulations()
True
sage: len(pc.triangulations_list())
26
```

```
>>> from sage.all import *
>>> points = [[Integer(0),Integer(0),Integer(0),Integer(1)], [Integer(0),Integer(3),
↪Integer(0),Integer(1)], [Integer(3),Integer(0),Integer(0),Integer(1)], [Integer(0),
↪Integer(0),Integer(1),Integer(1)],
...           [Integer(0),Integer(3),Integer(1),Integer(1)], [Integer(3),Integer(0),
↪Integer(1),Integer(1)], [Integer(1),Integer(1),Integer(2),Integer(1)]]
>>> points = [p + [Integer(1),Integer(2),Integer(3)] for p in points]
>>> pc = PointConfiguration(points)
>>> pc.ambient_dim()
7
>>> pc.dim()
3
>>> pc.triangulate()
(<0,1,2,6>, <0,1,3,6>, <0,2,3,6>, <1,2,4,6>, <1,3,4,6>, <2,3,5,6>, <2,4,5,6>)
>>> _ in pc.triangulations()
True
>>> len(pc.triangulations_list())
26
```

AUTHORS:

- Volker Braun: initial version, 2010
- Josh Whitney: added functionality for computing volumes and secondary polytopes of PointConfigurations
- Marshall Hampton: improved documentation and doctest coverage
- Volker Braun: rewrite using Parent/Element and categories. Added a Point class. More doctests. Less zombies.
- Volker Braun: Cythonized parts of it, added a C++ implementation of the bistellar flip algorithm to enumerate all connected triangulations.
- Volker Braun 2011: switched the triangulate() method to the placing triangulation (faster).

class `sage.geometry.triangulation.point_configuration.`**`PointConfiguration`**(*points*, *connected*, *fine*, *regular*, *star*, *defined_affine*)

Bases: `UniqueRepresentation`, *`PointConfiguration_base`*

A collection of points in Euclidean (or projective) space.

This is the parent class for the triangulations of the point configuration. There are a few options to specifically select what kind of triangulations are admissible.

INPUT:

The constructor accepts the following arguments:

- `points` – the points; technically, any iterable of iterables will do. In particular, a *`PointConfiguration`* can be passed.
- `projective` – boolean (default: `False`); whether the point coordinates should be interpreted as projective (`True`) or affine (`False`) coordinates. If necessary, points are projectivized by setting the last homogeneous coordinate to one and/or affine patches are chosen internally.
- `connected` – boolean (default: `True`); whether the triangulations should be connected to the regular triangulations via bistellar flips. These are much easier to compute than all triangulations.
- `fine` – boolean (default: `False`); whether the triangulations must be fine, that is, make use of all points of the configuration
- `regular` – boolean or `None` (default: `None`); whether the triangulations must be regular. A regular triangulation is one that is induced by a piecewise-linear convex support function. In other words, the shadows of the faces of a polyhedron in one higher dimension.
  - `True`: Only regular triangulations.
  - `False`: Only non-regular triangulations.
  - `None` (default): Both kinds of triangulation.
- `star` – either `None` or a point; whether the triangulations must be star. A triangulation is star if all maximal simplices contain a common point. The central point can be specified by its index (an integer) in the given points or by its coordinates (anything iterable.)

EXAMPLES:

```
sage: p = PointConfiguration([[0,0], [0,1], [1,0], [1,1], [-1,-1]]); p
A point configuration in affine 2-space over Integer Ring
consisting of 5 points. The triangulations of this point
configuration are assumed to be connected, not necessarily fine,
not necessarily regular.
sage: p.triangulate()   # a single triangulation
(<1,3,4>, <2,3,4>)
```

```
>>> from sage.all import *
>>> p = PointConfiguration([[Integer(0),Integer(0)], [Integer(0),Integer(1)],
↪[Integer(1),Integer(0)], [Integer(1),Integer(1)], [-Integer(1),-Integer(1)]]); p
A point configuration in affine 2-space over Integer Ring
consisting of 5 points. The triangulations of this point
configuration are assumed to be connected, not necessarily fine,
not necessarily regular.
>>> p.triangulate()   # a single triangulation
(<1,3,4>, <2,3,4>)
```

Element

alias of *Triangulation*

Gale_transform(*points=None*, *homogenize=True*)

Return the Gale transform of `self`.

INPUT:

- `points` – tuple of points or point indices or `None` (default). A subset of points for which to compute the Gale transform. By default, all points are used.
- `homogenize` – boolean (default: `True`); whether to add a row of 1's before taking the transform.

OUTPUT: a matrix over `base_ring()`

EXAMPLES:

```
sage: pc = PointConfiguration([(0,0), (1,0), (2,1), (1,1), (0,1)])
sage: pc.Gale_transform()
[ 1 -1  0  1 -1]
[ 0  0  1 -2  1]

sage: pc.Gale_transform((0,1,3,4))
[ 1 -1  1 -1]

sage: points = (pc.point(0), pc.point(1), pc.point(3), pc.point(4))
sage: pc.Gale_transform(points)
[ 1 -1  1 -1]
```

```
>>> from sage.all import *
>>> pc = PointConfiguration([(Integer(0),Integer(0)), (Integer(1),Integer(0)),
↪ (Integer(2),Integer(1)), (Integer(1),Integer(1)), (Integer(0),Integer(1))])
>>> pc.Gale_transform()
[ 1 -1  0  1 -1]
[ 0  0  1 -2  1]

>>> pc.Gale_transform((Integer(0),Integer(1),Integer(3),Integer(4)))
[ 1 -1  1 -1]

>>> points = (pc.point(Integer(0)), pc.point(Integer(1)), pc.
↪point(Integer(3)), pc.point(Integer(4)))
>>> pc.Gale_transform(points)
[ 1 -1  1 -1]
```

It is possible to take the inverse of the Gale transform, by specifying whether to homogenize or not:

```
sage: pc2 = PointConfiguration([[0,0],[3,0],[0,3],[3,3],[1,1]])
sage: pc2.Gale_transform(homogenize=False)
[ 1  0  0  0  0]
[ 0  1  1  0 -3]
[ 0  0  0  1 -3]
sage: pc2.Gale_transform(homogenize=True)
[ 1  1  1  0 -3]
[ 0  2  2 -1 -3]
```

```
>>> from sage.all import *
>>> pc2 = PointConfiguration([[Integer(0),Integer(0)],[Integer(3),Integer(0)],
↪[Integer(0),Integer(3)],[Integer(3),Integer(3)],[Integer(1),Integer(1)]])
>>> pc2.Gale_transform(homogenize=False)
[ 1  0  0  0  0]
[ 0  1  1  0 -3]
[ 0  0  0  1 -3]
>>> pc2.Gale_transform(homogenize=True)
[ 1  1  1  0 -3]
[ 0  2  2 -1 -3]
```

It might not affect the result (when acyclic):

```
sage: PC = PointConfiguration([[4,0,0],[0,4,0],[0,0,4],[2,1,1],[1,2,1],[1,1,
↪2]])
sage: GT = PC.Gale_transform(homogenize=False);GT
[ 1  0  0 -3  1  1]
[ 0  1  0  1 -3  1]
[ 0  0  1  1  1 -3]
sage: GT = PC.Gale_transform(homogenize=True);GT
[ 1  0  0 -3  1  1]
[ 0  1  0  1 -3  1]
[ 0  0  1  1  1 -3]
```

```
>>> from sage.all import *
>>> PC = PointConfiguration([[Integer(4),Integer(0),Integer(0)],[Integer(0),
↪Integer(4),Integer(0)],[Integer(0),Integer(0),Integer(4)],[Integer(2),
↪Integer(1),Integer(1)],[Integer(1),Integer(2),Integer(1)],[Integer(1),
↪Integer(1),Integer(2)]])
>>> GT = PC.Gale_transform(homogenize=False);GT
[ 1  0  0 -3  1  1]
[ 0  1  0  1 -3  1]
[ 0  0  1  1  1 -3]
>>> GT = PC.Gale_transform(homogenize=True);GT
[ 1  0  0 -3  1  1]
[ 0  1  0  1 -3  1]
[ 0  0  1  1  1 -3]
```

The following point configuration is totally cyclic (the cone spanned by the vectors is equal to the vector space spanned by the points), hence its Gale dual is acyclic (there is a linear functional that is positive in all the points of the configuration) when not homogenized:

```
sage: pc3 = PointConfiguration([[-1, -1, -1], [-1, 0, 0], [0, -1, 0], [0, 0, -
↪1], [1, 0, 0], [0, 0, 1], [0, 1, 0]])
```

(continues on next page)

(continued from previous page)

```
sage: g_hom = pc3.Gale_transform(homogenize=True);g_hom
[ 1  0  0 -2  1 -1  1]
[ 0  1  0 -1  1 -1  0]
[ 0  0  1 -1  0 -1  1]
sage: g_inhom = pc3.Gale_transform(homogenize=False);g_inhom
[1 0 0 0 1 1 1]
[0 1 0 0 1 0 0]
[0 0 1 0 0 0 1]
[0 0 0 1 0 1 0]
sage: Polyhedron(rays=g_hom.columns())
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 1 vertex and
↪3 lines
sage: Polyhedron(rays=g_inhom.columns())
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 1 vertex and
↪4 rays
```

```
>>> from sage.all import *
>>> pc3 = PointConfiguration([[-Integer(1), -Integer(1), -Integer(1)], [-
↪Integer(1), Integer(0), Integer(0)], [Integer(0), -Integer(1), Integer(0)],
↪[Integer(0), Integer(0), -Integer(1)], [Integer(1), Integer(0), Integer(0)],
↪ [Integer(0), Integer(0), Integer(1)], [Integer(0), Integer(1),
↪Integer(0)]])
>>> g_hom = pc3.Gale_transform(homogenize=True);g_hom
[ 1  0  0 -2  1 -1  1]
[ 0  1  0 -1  1 -1  0]
[ 0  0  1 -1  0 -1  1]
>>> g_inhom = pc3.Gale_transform(homogenize=False);g_inhom
[1 0 0 0 1 1 1]
[0 1 0 0 1 0 0]
[0 0 1 0 0 0 1]
[0 0 0 1 0 1 0]
>>> Polyhedron(rays=g_hom.columns())
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 1 vertex and
↪3 lines
>>> Polyhedron(rays=g_inhom.columns())
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 1 vertex and
↪4 rays
```

an_element()

Synonymous for *triangulate()*.

bistellar_flips()

Return the bistellar flips.

OUTPUT:

The bistellar flips as a tuple. Each flip is a pair (T_+, T_-) where T_+ and T_- are partial triangulations of the point configuration.

EXAMPLES:

```
sage: pc = PointConfiguration([(0,0),(1,0),(0,1),(1,1)])
sage: pc.bistellar_flips()
```

(continues on next page)

(continued from previous page)

```
(((<0,1,3>, <0,2,3>), (<0,1,2>, <1,2,3>)),)
sage: Tpos, Tneg = pc.bistellar_flips()[0]
sage: Tpos.plot(axes=False)                                          #␣
↪needs sage.plot
Graphics object consisting of 11 graphics primitives
sage: Tneg.plot(axes=False)                                          #␣
↪needs sage.plot
Graphics object consisting of 11 graphics primitives
```

```
>>> from sage.all import *
>>> pc = PointConfiguration([(Integer(0),Integer(0)),(Integer(1),Integer(0)),
↪(Integer(0),Integer(1)),(Integer(1),Integer(1))])
>>> pc.bistellar_flips()
(((<0,1,3>, <0,2,3>), (<0,1,2>, <1,2,3>)),)
>>> Tpos, Tneg = pc.bistellar_flips()[Integer(0)]
>>> Tpos.plot(axes=False)                                        # needs␣
↪sage.plot
Graphics object consisting of 11 graphics primitives
>>> Tneg.plot(axes=False)                                        # needs␣
↪sage.plot
Graphics object consisting of 11 graphics primitives
```

The 3d analog:

```
sage: pc = PointConfiguration([(0,0,0),(0,2,0),(0,0,2),(-1,0,0),(1,1,1)])
sage: pc.bistellar_flips()
(((<0,1,2,3>, <0,1,2,4>), (<0,1,3,4>, <0,2,3,4>, <1,2,3,4>)),)
```

```
>>> from sage.all import *
>>> pc = PointConfiguration([(Integer(0),Integer(0),Integer(0)),(Integer(0),
↪Integer(2),Integer(0)),(Integer(0),Integer(0),Integer(2)),(-Integer(1),
↪Integer(0),Integer(0)),(Integer(1),Integer(1),Integer(1))])
>>> pc.bistellar_flips()
(((<0,1,2,3>, <0,1,2,4>), (<0,1,3,4>, <0,2,3,4>, <1,2,3,4>)),)
```

A 2d flip on the base of the pyramid over a square:

```
sage: pc = PointConfiguration([(0,0,0),(0,2,0),(0,0,2),(0,2,2),(1,1,1)])
sage: pc.bistellar_flips()
(((<0,1,3>, <0,2,3>), (<0,1,2>, <1,2,3>)),)
sage: Tpos, Tneg = pc.bistellar_flips()[0]
sage: Tpos.plot(axes=False)                                          #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> pc = PointConfiguration([(Integer(0),Integer(0),Integer(0)),(Integer(0),
↪Integer(2),Integer(0)),(Integer(0),Integer(0),Integer(2)),(Integer(0),
↪Integer(2),Integer(2)),(Integer(1),Integer(1),Integer(1))])
>>> pc.bistellar_flips()
(((<0,1,3>, <0,2,3>), (<0,1,2>, <1,2,3>)),)
>>> Tpos, Tneg = pc.bistellar_flips()[Integer(0)]
```

(continues on next page)

(continued from previous page)

```
>>> Tpos.plot(axes=False)                                               # needs␣
↪sage.plot
Graphics3d Object
```

circuits()

Return the circuits of the point configuration.

Roughly, a circuit is a minimal linearly dependent subset of the points. That is, a circuit is a partition

$$\{0, 1, \dots, n-1\} = C_+ \cup C_0 \cup C_-$$

such that there is an (unique up to an overall normalization) affine relation

$$\sum_{i \in C_+} \alpha_i \vec{p}_i = \sum_{j \in C_-} \alpha_j \vec{p}_j$$

with all positive (or all negative) coefficients, where $\vec{p}_i = (p_1, \dots, p_k, 1)$ are the projective coordinates of the i-th point.

OUTPUT:

The list of (unsigned) circuits as triples (C_+, C_0, C_-). The swapped circuit (C_-, C_0, C_+) is not returned separately.

EXAMPLES:

```
sage: p = PointConfiguration([(0,0), (+1,0), (-1,0), (0,+1), (0,-1)])
sage: sorted(p.circuits())
[((0,), (1, 2), (3, 4)), ((0,), (3, 4), (1, 2)), ((1, 2), (0,), (3, 4))]
```

```
>>> from sage.all import *
>>> p = PointConfiguration([(Integer(0),Integer(0)), (+Integer(1),Integer(0)),
↪ (-Integer(1),Integer(0)), (Integer(0),+Integer(1)), (Integer(0),-
↪Integer(1))])
>>> sorted(p.circuits())
[((0,), (1, 2), (3, 4)), ((0,), (3, 4), (1, 2)), ((1, 2), (0,), (3, 4))]
```

circuits_support()

A generator for the supports of the circuits of the point configuration.

See `circuits()` for details.

OUTPUT:

A generator for the supports $C_- \cup C_+$ (returned as a Python tuple) for all circuits of the point configuration.

EXAMPLES:

```
sage: p = PointConfiguration([(0,0), (+1,0), (-1,0), (0,+1), (0,-1)])
sage: sorted(p.circuits_support())
[(0, 1, 2), (0, 3, 4), (1, 2, 3, 4)]
```

```
>>> from sage.all import *
>>> p = PointConfiguration([(Integer(0),Integer(0)), (+Integer(1),Integer(0)),
↪ (-Integer(1),Integer(0)), (Integer(0),+Integer(1)), (Integer(0),-
↪Integer(1))])
```

(continues on next page)

(continued from previous page)

```
>>> sorted(p.circuits_support())
[(0, 1, 2), (0, 3, 4), (1, 2, 3, 4)]
```

contained_simplex(*large=True*, *initial_point=None*, *point_order=None*)

Return a simplex contained in the point configuration.

INPUT:

- `large` – boolean; whether to attempt to return a large simplex
- `initial_point` – a *Point* or `None` (default). A specific point to start with when picking the simplex vertices.
- `point_order` – list or tuple of (some or all) *Point* s or `None` (default)

OUTPUT:

A tuple of points that span a simplex of dimension `dim()`. If `large==True`, the simplex is constructed by successively picking the farthest point. This will ensure that the simplex is not unnecessarily small, but will in general not return a maximal simplex. If a `point_order` is specified, the simplex is greedily constructed by considering the points in this order. The `large` option and `initial_point` is ignored in this case. The `point_order` may contain only a subset of the points; in this case, the dimension of the simplex will be the dimension of this subset.

EXAMPLES:

```
sage: pc = PointConfiguration([(0,0), (1,0), (2,1), (1,1), (0,1)])
sage: pc.contained_simplex()
(P(0, 1), P(2, 1), P(1, 0))
sage: pc.contained_simplex(large=False)
(P(0, 1), P(1, 1), P(1, 0))
sage: pc.contained_simplex(initial_point=pc.point(2))
(P(2, 1), P(0, 0), P(1, 0))

sage: pc = PointConfiguration([[0,0], [0,1], [1,0], [1,1], [-1,-1]])
sage: pc.contained_simplex()
(P(-1, -1), P(1, 1), P(0, 1))
sage: pc.contained_simplex(point_order=[pc[1], pc[3], pc[4], pc[2], pc[0]])
(P(0, 1), P(1, 1), P(-1, -1))
```

```
>>> from sage.all import *
>>> pc = PointConfiguration([(Integer(0),Integer(0)), (Integer(1),Integer(0)),
→ (Integer(2),Integer(1)), (Integer(1),Integer(1)), (Integer(0),Integer(1))])
>>> pc.contained_simplex()
(P(0, 1), P(2, 1), P(1, 0))
>>> pc.contained_simplex(large=False)
(P(0, 1), P(1, 1), P(1, 0))
>>> pc.contained_simplex(initial_point=pc.point(Integer(2)))
(P(2, 1), P(0, 0), P(1, 0))

>>> pc = PointConfiguration([[Integer(0),Integer(0)], [Integer(0),Integer(1)],
→ [Integer(1),Integer(0)], [Integer(1),Integer(1)], [-Integer(1),-
→Integer(1)]])
>>> pc.contained_simplex()
(P(-1, -1), P(1, 1), P(0, 1))
```

(continues on next page)

(continued from previous page)

```
>>> pc.contained_simplex(point_order=[pc[Integer(1)], pc[Integer(3)],
↪pc[Integer(4)], pc[Integer(2)], pc[Integer(0)]])
(P(0, 1), P(1, 1), P(-1, -1))
```

Lower-dimensional example:

```
sage: pc.contained_simplex(point_order=[pc[0], pc[3], pc[4]])
(P(0, 0), P(1, 1))
```

```
>>> from sage.all import *
>>> pc.contained_simplex(point_order=[pc[Integer(0)], pc[Integer(3)],
↪pc[Integer(4)]])
(P(0, 0), P(1, 1))
```

convex_hull()

Return the convex hull of the point configuration.

EXAMPLES:

```
sage: p = PointConfiguration([[0,0], [0,1], [1,0], [1,1], [-1,-1]])
sage: p.convex_hull()
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices
```

```
>>> from sage.all import *
>>> p = PointConfiguration([[Integer(0),Integer(0)], [Integer(0),Integer(1)],
↪[Integer(1),Integer(0)], [Integer(1),Integer(1)], [-Integer(1),-
↪Integer(1)]])
>>> p.convex_hull()
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices
```

deformation_cone(*collection*)

Return the deformation cone for the `collection` of subconfigurations of `self`.

INPUT:

- `collection` – a collection of subconfigurations of `self`. Subconfigurations are given as indices

OUTPUT: a polyhedron. It contains the liftings of the point configuration making the collection a regular (or coherent, or projective, or polytopal) subdivision.

EXAMPLES:

```
sage: PC = PointConfiguration([(-1, -1), (-1, 0), (0, -1), (1, 0), (0, 1)])
sage: coll = [(1, 4), (0, 2), (0, 1), (2, 3), (3, 4)]
sage: dc = PC.deformation_cone(coll);dc
A 5-dimensional polyhedron in QQ^5 defined as the convex hull of 1 vertex, 3
↪rays, 2 lines
sage: dc.rays()
(A ray in the direction (1, 0, 1, 0, 0),
 A ray in the direction (1, 1, 0, 0, 0),
 A ray in the direction (1, 1, 1, 0, 0))
sage: dc.lines()
(A line in the direction (1, 0, 1, 0, -1),
 A line in the direction (1, 1, 0, -1, 0))
```

(continues on next page)

(continued from previous page)

```
sage: dc.an_element()
(3, 2, 2, 0, 0)
```

```
>>> from sage.all import *
>>> PC = PointConfiguration([(-Integer(1), -Integer(1)), (-Integer(1),␣
↪Integer(0)), (Integer(0), -Integer(1)), (Integer(1), Integer(0)),␣
↪(Integer(0), Integer(1))])
>>> coll = [(Integer(1), Integer(4)), (Integer(0), Integer(2)), (Integer(0),␣
↪Integer(1)), (Integer(2), Integer(3)), (Integer(3), Integer(4))]
>>> dc = PC.deformation_cone(coll);dc
A 5-dimensional polyhedron in QQ^5 defined as the convex hull of 1 vertex, 3␣
↪rays, 2 lines
>>> dc.rays()
(A ray in the direction (1, 0, 1, 0, 0),
 A ray in the direction (1, 1, 0, 0, 0),
 A ray in the direction (1, 1, 1, 0, 0))
>>> dc.lines()
(A line in the direction (1, 0, 1, 0, -1),
 A line in the direction (1, 1, 0, -1, 0))
>>> dc.an_element()
(3, 2, 2, 0, 0)
```

We add to the interior element the first line and we verify that the given rays are defining rays of the lower hull:

```
sage: P = Polyhedron(rays=[(-1, -1, 4), (-1, 0, 3), (0, -1, 2), (1, 0, -1),␣
↪(0, 1, 0)])
sage: P.rays()
(A ray in the direction (-1, -1, 4),
 A ray in the direction (-1, 0, 3),
 A ray in the direction (0, -1, 2),
 A ray in the direction (0, 1, 0),
 A ray in the direction (1, 0, -1))
```

```
>>> from sage.all import *
>>> P = Polyhedron(rays=[(-Integer(1), -Integer(1), Integer(4)), (-Integer(1),
↪ Integer(0), Integer(3)), (Integer(0), -Integer(1), Integer(2)),␣
↪(Integer(1), Integer(0), -Integer(1)), (Integer(0), Integer(1),␣
↪Integer(0))])
>>> P.rays()
(A ray in the direction (-1, -1, 4),
 A ray in the direction (-1, 0, 3),
 A ray in the direction (0, -1, 2),
 A ray in the direction (0, 1, 0),
 A ray in the direction (1, 0, -1))
```

Let's verify the mother of all examples explained in Section 7.1.1 of [DLRS2010]:

```
sage: def mother(epsilon=0):
....:     return PointConfiguration([(4-epsilon,epsilon,0),(0,4-epsilon,
↪epsilon),(epsilon,0,4-epsilon),(2,1,1),(1,2,1),(1,1,2)])
```

(continues on next page)

(continued from previous page)

```
sage: epsilon = 0
sage: m = mother(0)
sage: m.points()
(P(4, 0, 0), P(0, 4, 0), P(0, 0, 4), P(2, 1, 1), P(1, 2, 1), P(1, 1, 2))
sage: S1 = [(0,1,4),(0,3,4),(1,2,5),(1,4,5),(0,2,3),(2,3,5)]
sage: S2 = [(0,1,3),(1,3,4),(1,2,4),(2,4,5),(0,2,5),(0,3,5)]
```

```
>>> from sage.all import *
>>> def mother(epsilon=Integer(0)):
...     return PointConfiguration([(Integer(4)-epsilon,epsilon,Integer(0)),
↪(Integer(0),Integer(4)-epsilon,epsilon),(epsilon,Integer(0),Integer(4)-
↪epsilon),(Integer(2),Integer(1),Integer(1)),(Integer(1),Integer(2),
↪Integer(1)),(Integer(1),Integer(1),Integer(2))])

>>> epsilon = Integer(0)
>>> m = mother(Integer(0))
>>> m.points()
(P(4, 0, 0), P(0, 4, 0), P(0, 0, 4), P(2, 1, 1), P(1, 2, 1), P(1, 1, 2))
>>> S1 = [(Integer(0),Integer(1),Integer(4)),(Integer(0),Integer(3),
↪Integer(4)),(Integer(1),Integer(2),Integer(5)),(Integer(1),Integer(4),
↪Integer(5)),(Integer(0),Integer(2),Integer(3)),(Integer(2),Integer(3),
↪Integer(5))]
>>> S2 = [(Integer(0),Integer(1),Integer(3)),(Integer(1),Integer(3),
↪Integer(4)),(Integer(1),Integer(2),Integer(4)),(Integer(2),Integer(4),
↪Integer(5)),(Integer(0),Integer(2),Integer(5)),(Integer(0),Integer(3),
↪Integer(5))]
```

Both subdivisions $S1$ and $S2$ are not regular:

```
sage: mother_dc1 = m.deformation_cone(S1)
sage: mother_dc1
A 4-dimensional polyhedron in QQ^6 defined as the convex hull of 1 vertex, 1␣
↪ray, 3 lines
sage: mother_dc2 = m.deformation_cone(S2)
sage: mother_dc2
A 4-dimensional polyhedron in QQ^6 defined as the convex hull of 1 vertex, 1␣
↪ray, 3 lines
```

```
>>> from sage.all import *
>>> mother_dc1 = m.deformation_cone(S1)
>>> mother_dc1
A 4-dimensional polyhedron in QQ^6 defined as the convex hull of 1 vertex, 1␣
↪ray, 3 lines
>>> mother_dc2 = m.deformation_cone(S2)
>>> mother_dc2
A 4-dimensional polyhedron in QQ^6 defined as the convex hull of 1 vertex, 1␣
↪ray, 3 lines
```

Notice that they have a ray which provides a degenerate lifting which only provides a coarsening of the subdivision from the lower hull (it has 5 facets, and should have 8):

```
sage: result = Polyhedron([vector(list(m.points()[_])+[mother_dc1.rays()[0][_
↪]]) for _ in range(len(m.points()))])
sage: result.f_vector()
(1, 6, 9, 5, 1)
```

```
>>> from sage.all import *
>>> result = Polyhedron([vector(list(m.points()[_])+[mother_dc1.
↪rays()[Integer(0)][_]]) for _ in range(len(m.points()))])
>>> result.f_vector()
(1, 6, 9, 5, 1)
```

But if we use epsilon to perturb the configuration, suddenly $S1$ becomes regular:

```
sage: epsilon = 1/2
sage: mp = mother(epsilon)
sage: mp.points()
(P(7/2, 1/2, 0),
 P(0, 7/2, 1/2),
 P(1/2, 0, 7/2),
 P(2, 1, 1),
 P(1, 2, 1),
 P(1, 1, 2))
sage: mother_dc1 = mp.deformation_cone(S1);mother_dc1
A 6-dimensional polyhedron in QQ^6 defined as the convex hull of 1 vertex, 3␣
↪rays, 3 lines
sage: mother_dc2 = mp.deformation_cone(S2);mother_dc2
A 3-dimensional polyhedron in QQ^6 defined as the convex hull of 1 vertex and␣
↪3 lines
```

```
>>> from sage.all import *
>>> epsilon = Integer(1)/Integer(2)
>>> mp = mother(epsilon)
>>> mp.points()
(P(7/2, 1/2, 0),
 P(0, 7/2, 1/2),
 P(1/2, 0, 7/2),
 P(2, 1, 1),
 P(1, 2, 1),
 P(1, 1, 2))
>>> mother_dc1 = mp.deformation_cone(S1);mother_dc1
A 6-dimensional polyhedron in QQ^6 defined as the convex hull of 1 vertex, 3␣
↪rays, 3 lines
>>> mother_dc2 = mp.deformation_cone(S2);mother_dc2
A 3-dimensional polyhedron in QQ^6 defined as the convex hull of 1 vertex and␣
↪3 lines
```

See also

`Kaehler_cone()`

REFERENCES:

For more information, see Section 5.4 of [DLRS2010] and Section 2.2 of [ACEP2020].

distance(*x*, *y*)

Return the distance between two points.

INPUT:

- x, y – two points of the point configuration

OUTPUT:

The distance between x and y, measured either with *distance_affine()* or *distance_FS()* depending on whether the point configuration is defined by affine or projective points. These are related, but not equal to the usual flat and Fubini-Study distance.

EXAMPLES:

```
sage: pc = PointConfiguration([(0,0), (1,0), (2,1), (1,2), (0,1)])
sage: [pc.distance(pc.point(0), p) for p in pc.points()]
[0, 1, 5, 5, 1]

sage: pc = PointConfiguration([(0,0,1), (1,0,1), (2,1,1), (1,2,1), (0,1,1)],
....:                         projective=True)
sage: [pc.distance(pc.point(0), p) for p in pc.points()]
[0, 1/2, 5/6, 5/6, 1/2]
```

```
>>> from sage.all import *
>>> pc = PointConfiguration([(Integer(0),Integer(0)), (Integer(1),Integer(0)),
↪ (Integer(2),Integer(1)), (Integer(1),Integer(2)), (Integer(0),Integer(1))])
>>> [pc.distance(pc.point(Integer(0)), p) for p in pc.points()]
[0, 1, 5, 5, 1]

>>> pc = PointConfiguration([(Integer(0),Integer(0),Integer(1)), (Integer(1),
↪Integer(0),Integer(1)), (Integer(2),Integer(1),Integer(1)), (Integer(1),
↪Integer(2),Integer(1)), (Integer(0),Integer(1),Integer(1))],
...                         projective=True)
>>> [pc.distance(pc.point(Integer(0)), p) for p in pc.points()]
[0, 1/2, 5/6, 5/6, 1/2]
```

distance_FS(*x*, *y*)

Return the distance between two points.

The distance function used in this method is $1 - \cos d_{FS}(x, y)^2$, where d_{FS} is the Fubini-Study distance of projective points. Recall the Fubini-Studi distance function

$$d_{FS}(x, y) = \arccos \sqrt{\frac{(x \cdot y)^2}{|x|^2 |y|^2}}$$

INPUT:

- x, y – two points of the point configuration

OUTPUT:

The distance $1 - \cos d_{FS}(x, y)^2$. Note that this distance lies in the same field as the entries of x, y. That is, the distance of rational points will be rational and so on.

EXAMPLES:

```
sage: pc = PointConfiguration([(0,0), (1,0), (2,1), (1,2), (0,1)])
sage: [pc.distance_FS(pc.point(0), p) for p in pc.points()]
[0, 1/2, 5/6, 5/6, 1/2]
```

```
>>> from sage.all import *
>>> pc = PointConfiguration([(Integer(0),Integer(0)), (Integer(1),Integer(0)),
↪ (Integer(2),Integer(1)), (Integer(1),Integer(2)), (Integer(0),Integer(1))])
>>> [pc.distance_FS(pc.point(Integer(0)), p) for p in pc.points()]
[0, 1/2, 5/6, 5/6, 1/2]
```

distance_affine(*x*, *y*)

Return the distance between two points.

The distance function used in this method is $d_{aff}(x, y)^2$, the square of the usual affine distance function

$$d_{aff}(x, y) = |x - y|$$

INPUT:

- `x`, `y` – two points of the point configuration

OUTPUT:

The metric distance-square $d_{aff}(x, y)^2$. Note that this distance lies in the same field as the entries of `x`, `y`. That is, the distance of rational points will be rational and so on.

EXAMPLES:

```
sage: pc = PointConfiguration([(0,0),(1,0),(2,1),(1,2),(0,1)])
sage: [pc.distance_affine(pc.point(0), p) for p in pc.points()]
[0, 1, 5, 5, 1]
```

```
>>> from sage.all import *
>>> pc = PointConfiguration([(Integer(0),Integer(0)),(Integer(1),Integer(0)),
↪(Integer(2),Integer(1)),(Integer(1),Integer(2)),(Integer(0),Integer(1))])
>>> [pc.distance_affine(pc.point(Integer(0)), p) for p in pc.points()]
[0, 1, 5, 5, 1]
```

exclude_points(*point_idx_list*)

Return a new point configuration with the given points removed.

INPUT:

- `point_idx_list` – list of integers; the indices of points to exclude

OUTPUT:

A new *`PointConfiguration`* with the given points removed.

EXAMPLES:

```
sage: p = PointConfiguration([[-1,0], [0,0], [1,-1], [1,0], [1,1]])
sage: list(p)
[P(-1, 0), P(0, 0), P(1, -1), P(1, 0), P(1, 1)]
sage: q = p.exclude_points([3])
sage: list(q)
[P(-1, 0), P(0, 0), P(1, -1), P(1, 1)]
```

(continues on next page)

(continued from previous page)

```
sage: p.exclude_points(p.face_interior(codim=1)).points()
(P(-1, 0), P(0, 0), P(1, -1), P(1, 1))
```

```
>>> from sage.all import *
>>> p = PointConfiguration([[-Integer(1),Integer(0)], [Integer(0),Integer(0)],
↪ [Integer(1),-Integer(1)], [Integer(1),Integer(0)], [Integer(1),
↪Integer(1)]])
>>> list(p)
[P(-1, 0), P(0, 0), P(1, -1), P(1, 0), P(1, 1)]
>>> q = p.exclude_points([Integer(3)])
>>> list(q)
[P(-1, 0), P(0, 0), P(1, -1), P(1, 1)]
>>> p.exclude_points(p.face_interior(codim=Integer(1))).points()
(P(-1, 0), P(0, 0), P(1, -1), P(1, 1))
```

face_codimension(*point*)

Return the smallest $d \in \mathbf{Z}$ such that `point` is contained in the interior of a codimension-d face.

EXAMPLES:

```
sage: triangle = PointConfiguration([[0,0], [1,-1], [1,0], [1,1]])
sage: triangle.point(2)
P(1, 0)
sage: triangle.face_codimension(2)
1
sage: triangle.face_codimension([1,0])
1
```

```
>>> from sage.all import *
>>> triangle = PointConfiguration([[Integer(0),Integer(0)], [Integer(1),-
↪Integer(1)], [Integer(1),Integer(0)], [Integer(1),Integer(1)]])
>>> triangle.point(Integer(2))
P(1, 0)
>>> triangle.face_codimension(Integer(2))
1
>>> triangle.face_codimension([Integer(1),Integer(0)])
1
```

This also works for degenerate cases like the tip of the pyramid over a square (which saturates four inequalities):

```
sage: pyramid = PointConfiguration([[1,0,0], [0,1,1], [0,1,-1],
....:                                [0,-1,-1], [0,-1,1]])
sage: pyramid.face_codimension(0)
3
```

```
>>> from sage.all import *
>>> pyramid = PointConfiguration([[Integer(1),Integer(0),Integer(0)],␣
↪[Integer(0),Integer(1),Integer(1)], [Integer(0),Integer(1),-Integer(1)],
...                                [Integer(0),-Integer(1),-Integer(1)],␣
↪[Integer(0),-Integer(1),Integer(1)]])
```

(continues on next page)

(continued from previous page)

```
>>> pyramid.face_codimension(Integer(0))
3
```

face_interior(*dim=None*, *codim=None*)

Return points by the codimension of the containing face in the convex hull.

EXAMPLES:

```
sage: triangle = PointConfiguration([[-1,0], [0,0], [1,-1], [1,0], [1,1]])
sage: triangle.face_interior()
((1,), (3,), (0, 2, 4))
sage: triangle.face_interior(dim=0)    # the vertices of the convex hull
(0, 2, 4)
sage: triangle.face_interior(codim=1)  # interior of facets
(3,)
```

```
>>> from sage.all import *
>>> triangle = PointConfiguration([[-Integer(1),Integer(0)], [Integer(0),
↪Integer(0)], [Integer(1),-Integer(1)], [Integer(1),Integer(0)], [Integer(1),
↪Integer(1)]])
>>> triangle.face_interior()
((1,), (3,), (0, 2, 4))
>>> triangle.face_interior(dim=Integer(0))    # the vertices of the convex␣
↪hull
(0, 2, 4)
>>> triangle.face_interior(codim=Integer(1))  # interior of facets
(3,)
```

farthest_point(*points*, *among=None*)

Return the point with the most distance from `points`.

INPUT:

- `points` – list of points
- `among` – list of points or `None` (default); the set of points from which to pick the farthest one. By default, all points of the configuration are considered.

OUTPUT:

A *Point* with largest minimal distance from all given `points`.

EXAMPLES:

```
sage: pc = PointConfiguration([(0,0), (1,0), (1,1), (0,1)])
sage: pc.farthest_point([pc.point(0)])
P(1, 1)
```

```
>>> from sage.all import *
>>> pc = PointConfiguration([(Integer(0),Integer(0)), (Integer(1),Integer(0)),
↪ (Integer(1),Integer(1)), (Integer(0),Integer(1))])
>>> pc.farthest_point([pc.point(Integer(0))])
P(1, 1)
```

lexicographic_triangulation()

Return the lexicographic triangulation.

The algorithm was taken from [PUNTOS].

EXAMPLES:

```
sage: p = PointConfiguration([(0,0), (+1,0), (-1,0), (0,+1), (0,-1)])
sage: p.lexicographic_triangulation()
(<1,3,4>, <2,3,4>)
```

```
>>> from sage.all import *
>>> p = PointConfiguration([(Integer(0),Integer(0)), (+Integer(1),Integer(0)),
↪ (-Integer(1),Integer(0)), (Integer(0),+Integer(1)), (Integer(0),-
↪Integer(1))])
>>> p.lexicographic_triangulation()
(<1,3,4>, <2,3,4>)
```

placing_triangulation(*point_order=None*)

Construct the placing (pushing) triangulation.

INPUT:

- `point_order` – list of points or integers. The order in which the points are to be placed. If not given, the points will be placed in some arbitrary order that attempts to produce a small number of simplices.

OUTPUT: a `Triangulation`

EXAMPLES:

```
sage: pc = PointConfiguration([(0,0), (1,0), (2,1), (1,2), (0,1)])
sage: pc.placing_triangulation()
(<0,1,2>, <0,2,4>, <2,3,4>)
sage: pc.placing_triangulation(point_order=(3,2,1,4,0))
(<0,1,4>, <1,2,3>, <1,3,4>)
sage: pc.placing_triangulation(point_order=[pc[1], pc[3], pc[4], pc[0]])
(<0,1,4>, <1,3,4>)
sage: U = matrix([
....:    [ 0, 0, 0, 0, 0, 2, 4,-1, 1, 1, 0, 0, 1, 0],
....:    [ 0, 0, 0, 1, 0, 0,-1, 0, 0, 0, 0, 0, 0, 0],
....:    [ 0, 2, 0, 0, 0, 0,-1, 0, 1, 0, 1, 0, 0, 1],
....:    [ 0, 1, 1, 0, 0, 1, 0,-2, 1, 0, 0,-1, 1, 1],
....:    [ 0, 0, 0, 0, 1, 0,-1, 0, 0, 0, 0, 0, 0, 0]
....: ])
sage: p = PointConfiguration(U.columns())
sage: triangulation = p.placing_triangulation();  triangulation
(<0,2,3,4,6,7>, <0,2,3,4,6,12>, <0,2,3,4,7,13>, <0,2,3,4,12,13>,
 <0,2,3,6,7,13>, <0,2,3,6,12,13>, <0,2,4,6,7,13>, <0,2,4,6,12,13>,
 <0,3,4,6,7,12>, <0,3,4,7,12,13>, <0,3,6,7,12,13>, <0,4,6,7,12,13>,
 <1,3,4,5,6,12>, <1,3,4,6,11,12>, <1,3,4,7,11,13>, <1,3,4,11,12,13>,
 <1,3,6,7,11,13>, <1,3,6,11,12,13>, <1,4,6,7,11,13>, <1,4,6,11,12,13>,
 <3,4,6,7,11,12>, <3,4,7,11,12,13>, <3,6,7,11,12,13>, <4,6,7,11,12,13>)
sage: sum(p.volume(t) for t in triangulation)
42
sage: p0 = PointConfiguration([(0,0), (+1,0), (-1,0), (0,+1), (0,-1)])
sage: p0.pushing_triangulation(point_order=[1,2,0,3,4])
```

(continues on next page)

(continued from previous page)

```
(<1,2,3>, <1,2,4>)
sage: p0.pushing_triangulation(point_order=[0,1,2,3,4])
(<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>)
```

```
>>> from sage.all import *
>>> pc = PointConfiguration([(Integer(0),Integer(0)), (Integer(1),Integer(0)),
↪ (Integer(2),Integer(1)), (Integer(1),Integer(2)), (Integer(0),Integer(1))])
>>> pc.placing_triangulation()
(<0,1,2>, <0,2,4>, <2,3,4>)
>>> pc.placing_triangulation(point_order=(Integer(3),Integer(2),Integer(1),
↪Integer(4),Integer(0)))
(<0,1,4>, <1,2,3>, <1,3,4>)
>>> pc.placing_triangulation(point_order=[pc[Integer(1)], pc[Integer(3)],␣
↪pc[Integer(4)], pc[Integer(0)]])
(<0,1,4>, <1,3,4>)
>>> U = matrix([
...     [ Integer(0), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(2), Integer(4),-Integer(1), Integer(1), Integer(1), Integer(0),␣
↪Integer(0), Integer(1), Integer(0)],
...     [ Integer(0), Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(0),-Integer(1), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(0), Integer(0), Integer(0)],
...     [ Integer(0), Integer(2), Integer(0), Integer(0), Integer(0),␣
↪Integer(0),-Integer(1), Integer(0), Integer(1), Integer(0), Integer(1),␣
↪Integer(0), Integer(0), Integer(1)],
...     [ Integer(0), Integer(1), Integer(1), Integer(0), Integer(0),␣
↪Integer(1), Integer(0),-Integer(2), Integer(1), Integer(0), Integer(0),-
↪Integer(1), Integer(1), Integer(1)],
...     [ Integer(0), Integer(0), Integer(0), Integer(0), Integer(1),␣
↪Integer(0),-Integer(1), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(0), Integer(0), Integer(0)]
... ])
>>> p = PointConfiguration(U.columns())
>>> triangulation = p.placing_triangulation();  triangulation
(<0,2,3,4,6,7>, <0,2,3,4,6,12>, <0,2,3,4,7,13>, <0,2,3,4,12,13>,
 <0,2,3,6,7,13>, <0,2,3,6,12,13>, <0,2,4,6,7,13>, <0,2,4,6,12,13>,
 <0,3,4,6,7,12>, <0,3,4,7,12,13>, <0,3,6,7,12,13>, <0,4,6,7,12,13>,
 <1,3,4,5,6,12>, <1,3,4,6,11,12>, <1,3,4,7,11,13>, <1,3,4,11,12,13>,
 <1,3,6,7,11,13>, <1,3,6,11,12,13>, <1,4,6,7,11,13>, <1,4,6,11,12,13>,
 <3,4,6,7,11,12>, <3,4,7,11,12,13>, <3,6,7,11,12,13>, <4,6,7,11,12,13>)
>>> sum(p.volume(t) for t in triangulation)
42
>>> p0 = PointConfiguration([(Integer(0),Integer(0)), (+Integer(1),
↪Integer(0)), (-Integer(1),Integer(0)), (Integer(0),+Integer(1)),␣
↪(Integer(0),-Integer(1))])
>>> p0.pushing_triangulation(point_order=[Integer(1),Integer(2),Integer(0),
↪Integer(3),Integer(4)])
(<1,2,3>, <1,2,4>)
>>> p0.pushing_triangulation(point_order=[Integer(0),Integer(1),Integer(2),
↪Integer(3),Integer(4)])
(<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>)
```

The same triangulation with renumbered points 0->4, 1->0, etc:

```
sage: p1 = PointConfiguration([(+1,0), (-1,0), (0,+1), (0,-1), (0,0)])
sage: p1.pushing_triangulation(point_order=[4,0,1,2,3])
(<0,2,4>, <0,3,4>, <1,2,4>, <1,3,4>)
```

```
>>> from sage.all import *
>>> p1 = PointConfiguration([(+Integer(1),Integer(0)), (-Integer(1),
↪Integer(0)), (Integer(0),+Integer(1)), (Integer(0),-Integer(1)),␣
↪(Integer(0),Integer(0))])
>>> p1.pushing_triangulation(point_order=[Integer(4),Integer(0),Integer(1),
↪Integer(2),Integer(3)])
(<0,2,4>, <0,3,4>, <1,2,4>, <1,3,4>)
```

plot(***kwds*)

Produce a graphical representation of the point configuration.

EXAMPLES:

```
sage: p = PointConfiguration([[0,0], [0,1], [1,0], [1,1], [-1,-1]])
sage: p.plot(axes=False)                                                  #␣
↪needs sage.plot
Graphics object consisting of 5 graphics primitives
```

```
>>> from sage.all import *
>>> p = PointConfiguration([[Integer(0),Integer(0)], [Integer(0),Integer(1)],␣
↪[Integer(1),Integer(0)], [Integer(1),Integer(1)], [-Integer(1),-
↪Integer(1)]])
>>> p.plot(axes=False)                                                  #␣
↪needs sage.plot
Graphics object consisting of 5 graphics primitives
```


positive_circuits(**negative*)

Return the positive part of circuits with fixed negative part.

A circuit is a pair (C_+, C_-), each consisting of a subset (actually, an ordered tuple) of point indices.

INPUT:

- `*negative` – integer; the indices of points

OUTPUT: a tuple of all circuits with C_{-} = `negative`

EXAMPLES:

```
sage: p = PointConfiguration([(1,0,0), (0,1,0), (0,0,1), (-2,0,-1), (-2,-1,0),
....:                          (-3,-1,-1), (1,1,1), (-1,0,0), (0,0,0)])
sage: sorted(p.positive_circuits(8))
[(0, 1, 2, 5), (0, 1, 4), (0, 2, 3), (0, 3, 4, 6), (0, 5, 6), (0, 7)]
sage: p.positive_circuits(0,5,6)
((8,),)
```

```
>>> from sage.all import *
>>> p = PointConfiguration([(Integer(1),Integer(0),Integer(0)), (Integer(0),
↪Integer(1),Integer(0)), (Integer(0),Integer(0),Integer(1)), (-Integer(2),
↪Integer(0),-Integer(1)), (-Integer(2),-Integer(1),Integer(0)),
...                          (-Integer(3),-Integer(1),-Integer(1)),␣
↪(Integer(1),Integer(1),Integer(1)), (-Integer(1),Integer(0),Integer(0)),␣
↪(Integer(0),Integer(0),Integer(0))])
>>> sorted(p.positive_circuits(Integer(8)))
[(0, 1, 2, 5), (0, 1, 4), (0, 2, 3), (0, 3, 4, 6), (0, 5, 6), (0, 7)]
>>> p.positive_circuits(Integer(0),Integer(5),Integer(6))
((8,),)
```

pushing_triangulation(*point_order=None*)

Construct the placing (pushing) triangulation.

INPUT:

- `point_order` – list of points or integers. The order in which the points are to be placed. If not given, the points will be placed in some arbitrary order that attempts to produce a small number of simplices.

OUTPUT: a `Triangulation`

EXAMPLES:

```
sage: pc = PointConfiguration([(0,0), (1,0), (2,1), (1,2), (0,1)])
sage: pc.placing_triangulation()
(<0,1,2>, <0,2,4>, <2,3,4>)
sage: pc.placing_triangulation(point_order=(3,2,1,4,0))
(<0,1,4>, <1,2,3>, <1,3,4>)
sage: pc.placing_triangulation(point_order=[pc[1], pc[3], pc[4], pc[0]])
(<0,1,4>, <1,3,4>)
sage: U = matrix([
....:     [ 0, 0, 0, 0, 0, 2, 4,-1, 1, 1, 0, 0, 1, 0],
....:     [ 0, 0, 0, 1, 0, 0,-1, 0, 0, 0, 0, 0, 0, 0],
....:     [ 0, 2, 0, 0, 0, 0,-1, 0, 1, 0, 1, 0, 0, 1],
....:     [ 0, 1, 1, 0, 0, 1, 0,-2, 1, 0, 0,-1, 1, 1],
....:     [ 0, 0, 0, 0, 1, 0,-1, 0, 0, 0, 0, 0, 0, 0]
....: ])
sage: p = PointConfiguration(U.columns())
sage: triangulation = p.placing_triangulation();  triangulation
(<0,2,3,4,6,7>, <0,2,3,4,6,12>, <0,2,3,4,7,13>, <0,2,3,4,12,13>,
 <0,2,3,6,7,13>, <0,2,3,6,12,13>, <0,2,4,6,7,13>, <0,2,4,6,12,13>,
 <0,3,4,6,7,12>, <0,3,4,7,12,13>, <0,3,6,7,12,13>, <0,4,6,7,12,13>,
 <1,3,4,5,6,12>, <1,3,4,6,11,12>, <1,3,4,7,11,13>, <1,3,4,11,12,13>,
 <1,3,6,7,11,13>, <1,3,6,11,12,13>, <1,4,6,7,11,13>, <1,4,6,11,12,13>,
```

(continues on next page)

(continued from previous page)

```
 <3,4,6,7,11,12>, <3,4,7,11,12,13>, <3,6,7,11,12,13>, <4,6,7,11,12,13>)
sage: sum(p.volume(t) for t in triangulation)
42
sage: p0 = PointConfiguration([(0,0), (+1,0), (-1,0), (0,+1), (0,-1)])
sage: p0.pushing_triangulation(point_order=[1,2,0,3,4])
(<1,2,3>, <1,2,4>)
sage: p0.pushing_triangulation(point_order=[0,1,2,3,4])
(<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>)
```

```
>>> from sage.all import *
>>> pc = PointConfiguration([(Integer(0),Integer(0)), (Integer(1),Integer(0)),
↪ (Integer(2),Integer(1)), (Integer(1),Integer(2)), (Integer(0),Integer(1))])
>>> pc.placing_triangulation()
(<0,1,2>, <0,2,4>, <2,3,4>)
>>> pc.placing_triangulation(point_order=(Integer(3),Integer(2),Integer(1),
↪Integer(4),Integer(0)))
(<0,1,4>, <1,2,3>, <1,3,4>)
>>> pc.placing_triangulation(point_order=[pc[Integer(1)], pc[Integer(3)],␣
↪pc[Integer(4)], pc[Integer(0)]])
(<0,1,4>, <1,3,4>)
>>> U = matrix([
...    [ Integer(0), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(2), Integer(4),-Integer(1), Integer(1), Integer(1), Integer(0),␣
↪Integer(0), Integer(1), Integer(0)],
...    [ Integer(0), Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(0),-Integer(1), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(0), Integer(0), Integer(0)],
...    [ Integer(0), Integer(2), Integer(0), Integer(0), Integer(0),␣
↪Integer(0),-Integer(1), Integer(0), Integer(1), Integer(0), Integer(1),␣
↪Integer(0), Integer(0), Integer(1)],
...    [ Integer(0), Integer(1), Integer(1), Integer(0), Integer(0),␣
↪Integer(1), Integer(0),-Integer(2), Integer(1), Integer(0), Integer(0),-
↪Integer(1), Integer(1), Integer(1)],
...    [ Integer(0), Integer(0), Integer(0), Integer(0), Integer(1),␣
↪Integer(0),-Integer(1), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(0), Integer(0), Integer(0)]
... ])
>>> p = PointConfiguration(U.columns())
>>> triangulation = p.placing_triangulation();  triangulation
(<0,2,3,4,6,7>, <0,2,3,4,6,12>, <0,2,3,4,7,13>, <0,2,3,4,12,13>,
 <0,2,3,6,7,13>, <0,2,3,6,12,13>, <0,2,4,6,7,13>, <0,2,4,6,12,13>,
 <0,3,4,6,7,12>, <0,3,4,7,12,13>, <0,3,6,7,12,13>, <0,4,6,7,12,13>,
 <1,3,4,5,6,12>, <1,3,4,6,11,12>, <1,3,4,7,11,13>, <1,3,4,11,12,13>,
 <1,3,6,7,11,13>, <1,3,6,11,12,13>, <1,4,6,7,11,13>, <1,4,6,11,12,13>,
 <3,4,6,7,11,12>, <3,4,7,11,12,13>, <3,6,7,11,12,13>, <4,6,7,11,12,13>)
>>> sum(p.volume(t) for t in triangulation)
42
>>> p0 = PointConfiguration([(Integer(0),Integer(0)), (+Integer(1),
↪Integer(0)), (-Integer(1),Integer(0)), (Integer(0),+Integer(1)),␣
↪(Integer(0),-Integer(1))])
>>> p0.pushing_triangulation(point_order=[Integer(1),Integer(2),Integer(0),
↪Integer(3),Integer(4)])
```

(continues on next page)

(continued from previous page)

```
(<1,2,3>, <1,2,4>)
>>> p0.pushing_triangulation(point_order=[Integer(0),Integer(1),Integer(2),
↪Integer(3),Integer(4)])
(<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>)
```

The same triangulation with renumbered points 0->4, 1->0, etc:

```
sage: p1 = PointConfiguration([(+1,0), (-1,0), (0,+1), (0,-1), (0,0)])
sage: p1.pushing_triangulation(point_order=[4,0,1,2,3])
(<0,2,4>, <0,3,4>, <1,2,4>, <1,3,4>)
```

```
>>> from sage.all import *
>>> p1 = PointConfiguration([(+Integer(1),Integer(0)), (-Integer(1),
↪Integer(0)), (Integer(0),+Integer(1)), (Integer(0),-Integer(1)),
↪(Integer(0),Integer(0))])
>>> p1.pushing_triangulation(point_order=[Integer(4),Integer(0),Integer(1),
↪Integer(2),Integer(3)])
(<0,2,4>, <0,3,4>, <1,2,4>, <1,3,4>)
```

restrict_to_connected_triangulations(*connected=True*)

Restrict to connected triangulations.

NOTE:

Finding non-connected triangulations requires the optional TOPCOM package.

INPUT:

- `connected` – boolean; whether to restrict to triangulations that are connected by bistellar flips to the regular triangulations

OUTPUT:

A new *PointConfiguration* with the same points, but whose triangulations will all be in the connected component. See *PointConfiguration* for details.

EXAMPLES:

```
sage: p = PointConfiguration([[0,0], [0,1], [1,0], [1,1], [-1,-1]]); p
A point configuration in affine 2-space over Integer Ring
consisting of 5 points. The triangulations of this point
configuration are assumed to be connected, not necessarily
fine, not necessarily regular.
sage: len(p.triangulations_list())
4
sage: PointConfiguration.set_engine('topcom')
sage: p_all = p.restrict_to_connected_triangulations(connected=False)   #
↪optional - topcom
sage: len(p_all.triangulations_list())                                  #
↪optional - topcom
4
sage: p == p_all.restrict_to_connected_triangulations(connected=True)   #
↪optional - topcom
True
sage: PointConfiguration.set_engine('internal')
```

```
>>> from sage.all import *
>>> p = PointConfiguration([[Integer(0),Integer(0)], [Integer(0),Integer(1)],␣
→[Integer(1),Integer(0)], [Integer(1),Integer(1)], [-Integer(1),-
→Integer(1)]]); p
A point configuration in affine 2-space over Integer Ring
consisting of 5 points. The triangulations of this point
configuration are assumed to be connected, not necessarily
fine, not necessarily regular.
>>> len(p.triangulations_list())
4
>>> PointConfiguration.set_engine('topcom')
>>> p_all = p.restrict_to_connected_triangulations(connected=False)   #␣
→optional - topcom
>>> len(p_all.triangulations_list())                                  #␣
→optional - topcom
4
>>> p == p_all.restrict_to_connected_triangulations(connected=True)   #␣
→optional - topcom
True
>>> PointConfiguration.set_engine('internal')
```

restrict_to_fine_triangulations(*fine=True*)

Restrict to fine triangulations.

INPUT:

- `fine` – boolean; whether to restrict to fine triangulations

OUTPUT:

A new `PointConfiguration` with the same points, but whose triangulations will all be fine. See `PointConfiguration` for details.

EXAMPLES:

```
sage: p = PointConfiguration([[0,0], [0,1], [1,0], [1,1], [-1,-1]])
sage: p
A point configuration in affine 2-space over Integer Ring
consisting of 5 points. The triangulations of this point
configuration are assumed to be connected, not necessarily
fine, not necessarily regular.

sage: len(p.triangulations_list())
4
sage: p_fine = p.restrict_to_fine_triangulations()
sage: len(p.triangulations_list())
4
sage: p == p_fine.restrict_to_fine_triangulations(fine=False)
True
```

```
>>> from sage.all import *
>>> p = PointConfiguration([[Integer(0),Integer(0)], [Integer(0),Integer(1)],␣
→[Integer(1),Integer(0)], [Integer(1),Integer(1)], [-Integer(1),-
→Integer(1)]])
>>> p
```

(continues on next page)

(continued from previous page)

```
A point configuration in affine 2-space over Integer Ring
consisting of 5 points. The triangulations of this point
configuration are assumed to be connected, not necessarily
fine, not necessarily regular.

>>> len(p.triangulations_list())
4
>>> p_fine = p.restrict_to_fine_triangulations()
>>> len(p.triangulations_list())
4
>>> p == p_fine.restrict_to_fine_triangulations(fine=False)
True
```

restrict_to_regular_triangulations(*regular=True*)

Restrict to regular triangulations.

NOTE:

Regularity testing requires the optional TOPCOM package.

INPUT:

- `regular` – `True`, `False`, or `None`; whether to restrict to regular triangulations, irregular triangulations, or lift any restrictions on regularity

OUTPUT:

A new `PointConfiguration` with the same points, but whose triangulations will all be regular as specified. See `PointConfiguration` for details.

EXAMPLES:

```
sage: p = PointConfiguration([[0,0], [0,1], [1,0], [1,1], [-1,-1]]); p
A point configuration in affine 2-space over Integer Ring
consisting of 5 points. The triangulations of this point
configuration are assumed to be connected, not necessarily
fine, not necessarily regular.
sage: len(p.triangulations_list())
4
sage: PointConfiguration.set_engine('topcom')
sage: p_regular = p.restrict_to_regular_triangulations() # optional - topcom
sage: len(p_regular.triangulations_list())               # optional - topcom
4
sage: p == p_regular.restrict_to_regular_triangulations(regular=None) #␣
↪optional - topcom
True
sage: PointConfiguration.set_engine('internal')
```

```
>>> from sage.all import *
>>> p = PointConfiguration([[Integer(0),Integer(0)], [Integer(0),Integer(1)],␣
↪[Integer(1),Integer(0)], [Integer(1),Integer(1)], [-Integer(1),-
↪Integer(1)]]); p
A point configuration in affine 2-space over Integer Ring
consisting of 5 points. The triangulations of this point
configuration are assumed to be connected, not necessarily
```

(continues on next page)

(continued from previous page)

```
fine, not necessarily regular.
>>> len(p.triangulations_list())
4
>>> PointConfiguration.set_engine('topcom')
>>> p_regular = p.restrict_to_regular_triangulations() # optional - topcom
>>> len(p_regular.triangulations_list())                # optional - topcom
4
>>> p == p_regular.restrict_to_regular_triangulations(regular=None) #␣
↪optional - topcom
True
>>> PointConfiguration.set_engine('internal')
```

restrict_to_star_triangulations(*star*)

Restrict to star triangulations with the given point as the center.

INPUT:

- `origin` – `None` or an integer or the coordinates of a point. An integer denotes the index of the central point. If `None` is passed, any restriction on the starshape will be removed.

OUTPUT:

A new *PointConfiguration* with the same points, but whose triangulations will all be star. See *PointConfiguration* for details.

EXAMPLES:

```
sage: p = PointConfiguration([[0,0], [0,1], [1,0], [1,1], [-1,-1]])
sage: len(list(p.triangulations()))
4
sage: p_star =  p.restrict_to_star_triangulations(0)
sage: p_star is p.restrict_to_star_triangulations((0,0))
True
sage: p_star.triangulations_list()
[(<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>)]
sage: p_newstar = p_star.restrict_to_star_triangulations(1)  # pick different␣
↪origin
sage: p_newstar.triangulations_list()
[(<1,2,3>, <1,2,4>)]
sage: p == p_star.restrict_to_star_triangulations(star=None)
True
```

```
>>> from sage.all import *
>>> p = PointConfiguration([[Integer(0),Integer(0)], [Integer(0),Integer(1)],␣
↪[Integer(1),Integer(0)], [Integer(1),Integer(1)], [-Integer(1),-
↪Integer(1)]])
>>> len(list(p.triangulations()))
4
>>> p_star =  p.restrict_to_star_triangulations(Integer(0))
>>> p_star is p.restrict_to_star_triangulations((Integer(0),Integer(0)))
True
>>> p_star.triangulations_list()
[(<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>)]
>>> p_newstar = p_star.restrict_to_star_triangulations(Integer(1))  # pick␣
```

(continues on next page)

(continued from previous page)

```
↪different origin
>>> p_newstar.triangulations_list()
[(<1,2,3>, <1,2,4>)]
>>> p == p_star.restrict_to_star_triangulations(star=None)
True
```

restricted_automorphism_group()

Return the restricted automorphism group.

First, let the linear automorphism group be the subgroup of the affine group $AGL(d, \mathbf{R}) = GL(d, \mathbf{R}) \ltimes \mathbf{R}^d$ preserving the d-dimensional point configuration. The affine group acts in the usual way $\vec{x} \mapsto A\vec{x} + b$ on the ambient space.

The restricted automorphism group is the subgroup of the linear automorphism group generated by permutations of points. See [BSS2009] for more details and a description of the algorithm.

OUTPUT:

A `PermutationGroup` that is isomorphic to the restricted automorphism group is returned.

Note that in Sage, permutation groups always act on positive integers while lists etc. are indexed by nonnegative integers. The indexing of the permutation group is chosen to be shifted by `+1`. That is, the transposition `(i,j)` in the permutation group corresponds to exchange of `self[i-1]` and `self[j-1]`.

EXAMPLES:

```
sage: pyramid = PointConfiguration([[1,0,0], [0,1,1], [0,1,-1],
....:                                [0,-1,-1], [0,-1,1]])
sage: G = pyramid.restricted_automorphism_group()                        #␣
↪needs sage.graphs sage.groups
sage: G == PermutationGroup([[(3,5)], [(2,3),(4,5)], [(2,4)]])           #␣
↪needs sage.graphs sage.groups
True
sage: DihedralGroup(4).is_isomorphic(G)                                  #␣
↪needs sage.graphs sage.groups
True
```

```
>>> from sage.all import *
>>> pyramid = PointConfiguration([[Integer(1),Integer(0),Integer(0)],␣
↪[Integer(0),Integer(1),Integer(1)], [Integer(0),Integer(1),-Integer(1)],
...                              [Integer(0),-Integer(1),-Integer(1)],␣
↪[Integer(0),-Integer(1),Integer(1)]])
>>> G = pyramid.restricted_automorphism_group()                     # needs␣
↪sage.graphs sage.groups
>>> G == PermutationGroup([[(Integer(3),Integer(5))], [(Integer(2),
↪Integer(3)),(Integer(4),Integer(5))], [(Integer(2),Integer(4))]])        #␣
↪needs sage.graphs sage.groups
True
>>> DihedralGroup(Integer(4)).is_isomorphic(G)
↪# needs sage.graphs sage.groups
True
```

The square with an off-center point in the middle. Note that the middle point breaks the restricted automorphism group D_4 of the convex hull:

```
sage: square = PointConfiguration([(3/4,3/4), (1,1), (1,-1), (-1,-1), (-1,1)])
sage: square.restricted_automorphism_group()                                #␣
↪needs sage.graphs sage.groups
Permutation Group with generators [(3,5)]
sage: DihedralGroup(1).is_isomorphic(_)                                     #␣
↪needs sage.graphs sage.groups
True
```

```
>>> from sage.all import *
>>> square = PointConfiguration([(Integer(3)/Integer(4),Integer(3)/
↪Integer(4)), (Integer(1),Integer(1)), (Integer(1),-Integer(1)), (-
↪Integer(1),-Integer(1)), (-Integer(1),Integer(1))])
>>> square.restricted_automorphism_group()                            # needs␣
↪sage.graphs sage.groups
Permutation Group with generators [(3,5)]
>>> DihedralGroup(Integer(1)).is_isomorphic(_)
↪# needs sage.graphs sage.groups
True
```

secondary_polytope()

Calculate the secondary polytope of the point configuration.

For a definition of the secondary polytope, see [GKZ1994] page 220 Definition 1.6.

Note that if you restricted the admissible triangulations of the point configuration then the output will be the corresponding face of the whole secondary polytope.

OUTPUT:

The secondary polytope of the point configuration as an instance of *Polyhedron_base*.

EXAMPLES:

```
sage: p = PointConfiguration([[0,0], [1,0], [2,1], [1,2], [0,1]])
sage: poly = p.secondary_polytope()
sage: poly.vertices_matrix()
[1 1 3 3 5]
[3 5 1 4 1]
[4 2 5 2 4]
[2 4 2 5 4]
[5 3 4 1 1]
sage: poly.Vrepresentation()
(A vertex at (1, 3, 4, 2, 5),
 A vertex at (1, 5, 2, 4, 3),
 A vertex at (3, 1, 5, 2, 4),
 A vertex at (3, 4, 2, 5, 1),
 A vertex at (5, 1, 4, 4, 1))
sage: poly.Hrepresentation()
(An equation (0, 0, 1, 2, 1) x - 13 == 0,
 An equation (1, 0, 0, 2, 2) x - 15 == 0,
 An equation (0, 1, 0, -3, -2) x + 13 == 0,
 An inequality (0, 0, 0, -1, -1) x + 7 >= 0,
 An inequality (0, 0, 0, 1, 0) x - 2 >= 0,
 An inequality (0, 0, 0, -2, -1) x + 11 >= 0,
```

(continues on next page)

(continued from previous page)

```
 An inequality (0, 0, 0, 0, 1) x - 1 >= 0,
 An inequality (0, 0, 0, 3, 2) x - 14 >= 0)
```

```
>>> from sage.all import *
>>> p = PointConfiguration([[Integer(0),Integer(0)], [Integer(1),Integer(0)],␣
↪[Integer(2),Integer(1)], [Integer(1),Integer(2)], [Integer(0),Integer(1)]])
>>> poly = p.secondary_polytope()
>>> poly.vertices_matrix()
[1 1 3 3 5]
[3 5 1 4 1]
[4 2 5 2 4]
[2 4 2 5 4]
[5 3 4 1 1]
>>> poly.Vrepresentation()
(A vertex at (1, 3, 4, 2, 5),
 A vertex at (1, 5, 2, 4, 3),
 A vertex at (3, 1, 5, 2, 4),
 A vertex at (3, 4, 2, 5, 1),
 A vertex at (5, 1, 4, 4, 1))
>>> poly.Hrepresentation()
(An equation (0, 0, 1, 2, 1) x - 13 == 0,
 An equation (1, 0, 0, 2, 2) x - 15 == 0,
 An equation (0, 1, 0, -3, -2) x + 13 == 0,
 An inequality (0, 0, 0, -1, -1) x + 7 >= 0,
 An inequality (0, 0, 0, 1, 0) x - 2 >= 0,
 An inequality (0, 0, 0, -2, -1) x + 11 >= 0,
 An inequality (0, 0, 0, 0, 1) x - 1 >= 0,
 An inequality (0, 0, 0, 3, 2) x - 14 >= 0)
```

classmethod set_engine(*engine='auto'*)

Set the engine used to compute triangulations.

INPUT:

- `engine` – either `'auto'` (default), `'internal'`, or `'topcom'`. The latter two instruct this package to always use its own triangulation algorithms or TOPCOM's algorithms, respectively. By default (`'auto'`), internal routines are used.

EXAMPLES:

```
sage: # optional - topcom
sage: p = PointConfiguration([[0,0], [0,1], [1,0], [1,1], [-1,-1]])
sage: p.set_engine('internal')   # to make doctests independent of TOPCOM
sage: p.triangulate()
(<1,3,4>, <2,3,4>)
sage: p.set_engine('topcom')
sage: p.triangulate()
(<0,1,2>, <0,1,4>, <0,2,4>, <1,2,3>)
sage: p.set_engine('internal')
```

```
>>> from sage.all import *
>>> # optional - topcom
>>> p = PointConfiguration([[Integer(0),Integer(0)], [Integer(0),Integer(1)],␣
```

(continues on next page)

(continued from previous page)

```
↪[Integer(1),Integer(0)], [Integer(1),Integer(1)], [-Integer(1),-
↪Integer(1)]])
>>> p.set_engine('internal')   # to make doctests independent of TOPCOM
>>> p.triangulate()
(<1,3,4>, <2,3,4>)
>>> p.set_engine('topcom')
>>> p.triangulate()
(<0,1,2>, <0,1,4>, <0,2,4>, <1,2,3>)
>>> p.set_engine('internal')
```

star_center()

Return the center used for star triangulations.

See also

restrict_to_star_triangulations().

OUTPUT:

A *Point* if a distinguished star central point has been fixed. `ValueError` exception is raised otherwise.

EXAMPLES:

```
sage: pc = PointConfiguration([(1,0), (-1,0), (0,1), (0,2)], star=(0,1)); pc
A point configuration in affine 2-space over Integer Ring
consisting of 4 points. The triangulations of this point
configuration are assumed to be connected, not necessarily
fine, not necessarily regular, and star with center P(0, 1).
sage: pc.star_center()
P(0, 1)

sage: pc_nostar = pc.restrict_to_star_triangulations(None); pc_nostar
A point configuration in affine 2-space over Integer Ring
consisting of 4 points. The triangulations of this point
configuration are assumed to be connected, not necessarily
fine, not necessarily regular.
sage: pc_nostar.star_center()
Traceback (most recent call last):
...
ValueError: The point configuration has no star center defined.
```

```
>>> from sage.all import *
>>> pc = PointConfiguration([(Integer(1),Integer(0)), (-Integer(1),
↪Integer(0)), (Integer(0),Integer(1)), (Integer(0),Integer(2))],␣
↪star=(Integer(0),Integer(1))); pc
A point configuration in affine 2-space over Integer Ring
consisting of 4 points. The triangulations of this point
configuration are assumed to be connected, not necessarily
fine, not necessarily regular, and star with center P(0, 1).
>>> pc.star_center()
P(0, 1)
```

(continues on next page)

(continued from previous page)

```
>>> pc_nostar = pc.restrict_to_star_triangulations(None); pc_nostar
A point configuration in affine 2-space over Integer Ring
consisting of 4 points. The triangulations of this point
configuration are assumed to be connected, not necessarily
fine, not necessarily regular.
>>> pc_nostar.star_center()
Traceback (most recent call last):
...
ValueError: The point configuration has no star center defined.
```

triangulate(*verbose=False*)

Return one (in no particular order) triangulation.

INPUT:

- `verbose` – boolean; whether to print out the TOPCOM interaction, if any

OUTPUT:

A *Triangulation* satisfying all restrictions imposed. This raises a `ValueError` if no such triangulation exists.

EXAMPLES:

```
sage: p = PointConfiguration([[0,0], [0,1], [1,0], [1,1], [-1,-1]])
sage: p.triangulate()
(<1,3,4>, <2,3,4>)
sage: list( p.triangulate() )
[(1, 3, 4), (2, 3, 4)]
```

```
>>> from sage.all import *
>>> p = PointConfiguration([[Integer(0),Integer(0)], [Integer(0),Integer(1)],
→[Integer(1),Integer(0)], [Integer(1),Integer(1)], [-Integer(1),-
→Integer(1)]])
>>> p.triangulate()
(<1,3,4>, <2,3,4>)
>>> list( p.triangulate() )
[(1, 3, 4), (2, 3, 4)]
```

Using TOPCOM yields a different, but equally good, triangulation:

```
sage: # optional - topcom
sage: p.set_engine('topcom')
sage: p.triangulate()
(<0,1,2>, <0,1,4>, <0,2,4>, <1,2,3>)
sage: list(p.triangulate())
[(0, 1, 2), (0, 1, 4), (0, 2, 4), (1, 2, 3)]
sage: p.set_engine('internal')
```

```
>>> from sage.all import *
>>> # optional - topcom
>>> p.set_engine('topcom')
>>> p.triangulate()
(<0,1,2>, <0,1,4>, <0,2,4>, <1,2,3>)
```

(continues on next page)

(continued from previous page)

```
>>> list(p.triangulate())
[(0, 1, 2), (0, 1, 4), (0, 2, 4), (1, 2, 3)]
>>> p.set_engine('internal')
```

triangulations(*verbose=False*)

Return all triangulations.

- `verbose` – boolean (default: `False`); whether to print out the TOPCOM interaction, if any

OUTPUT:

A generator for the triangulations satisfying all the restrictions imposed. Each triangulation is returned as a `Triangulation` object.

EXAMPLES:

```
    sage: p = PointConfiguration([[0,0], [0,1], [1,0], [1,1], [-1,-1]])
    sage: iter = p.triangulations()
    sage: next(iter)
    (<1,3,4>, <2,3,4>)
    sage: next(iter)
    (<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>)
    sage: next(iter)
    (<1,2,3>, <1,2,4>)
    sage: next(iter)
    (<0,1,2>, <0,1,4>, <0,2,4>, <1,2,3>)
    sage: p.triangulations_list()
    [(<1,3,4>, <2,3,4>),
     (<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>),
     (<1,2,3>, <1,2,4>),
     (<0,1,2>, <0,1,4>, <0,2,4>, <1,2,3>)]
    sage: p_fine = p.restrict_to_fine_triangulations()
    sage: p_fine.triangulations_list()
    [(<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>),
     (<0,1,2>, <0,1,4>, <0,2,4>, <1,2,3>)]

Note that we explicitly asked the internal algorithm to
compute the triangulations. Using TOPCOM, we obtain the same
triangulations but in a different order::

    sage: # optional - topcom
    sage: p.set_engine('topcom')
    sage: iter = p.triangulations()
    sage: next(iter)
    (<0,1,2>, <0,1,4>, <0,2,4>, <1,2,3>)
    sage: next(iter)
    (<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>)
    sage: next(iter)
    (<1,2,3>, <1,2,4>)
    sage: next(iter)
    (<1,3,4>, <2,3,4>)
    sage: p.triangulations_list()
    [(<0,1,2>, <0,1,4>, <0,2,4>, <1,2,3>),
     (<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>),
```

(continues on next page)

(continued from previous page)

```
 (<1,2,3>, <1,2,4>),
 (<1,3,4>, <2,3,4>)]
sage: p_fine = p.restrict_to_fine_triangulations()
sage: p_fine.set_engine('topcom')
sage: p_fine.triangulations_list()
[(<0,1,2>, <0,1,4>, <0,2,4>, <1,2,3>),
 (<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>)]
sage: p.set_engine('internal')
```

triangulations_list(*verbose=False*)

Return all triangulations.

INPUT:

- `verbose` – boolean; whether to print out the TOPCOM interaction, if any

OUTPUT:

A list of triangulations (see *Triangulation*) satisfying all restrictions imposed previously.

EXAMPLES:

```
sage: p = PointConfiguration([[0,0], [0,1], [1,0], [1,1]])
sage: p.triangulations_list()
[(<0,1,2>, <1,2,3>), (<0,1,3>, <0,2,3>)]
sage: list(map(list, p.triangulations_list()))
[[(0, 1, 2), (1, 2, 3)], [(0, 1, 3), (0, 2, 3)]]
sage: p.set_engine('topcom')
sage: p.triangulations_list()      # optional - topcom
[(<0,1,2>, <1,2,3>), (<0,1,3>, <0,2,3>)]
sage: p.set_engine('internal')
```

```
>>> from sage.all import *
>>> p = PointConfiguration([[Integer(0),Integer(0)], [Integer(0),Integer(1)],
↪[Integer(1),Integer(0)], [Integer(1),Integer(1)]])
>>> p.triangulations_list()
[(<0,1,2>, <1,2,3>), (<0,1,3>, <0,2,3>)]
>>> list(map(list, p.triangulations_list()))
[[(0, 1, 2), (1, 2, 3)], [(0, 1, 3), (0, 2, 3)]]
>>> p.set_engine('topcom')
>>> p.triangulations_list()      # optional - topcom
[(<0,1,2>, <1,2,3>), (<0,1,3>, <0,2,3>)]
>>> p.set_engine('internal')
```

volume(*simplex=None*)

Find $n!$ times the n-volume of a simplex of dimension n.

INPUT:

- `simplex` – (optional argument) a simplex from a triangulation T specified as a list of point indices

OUTPUT:

- If a simplex was passed as an argument: $n!$ * (volume of `simplex`).
- Without argument: $n!$ * (the total volume of the convex hull).

EXAMPLES:

The volume of the standard simplex should always be 1:

```
sage: p = PointConfiguration([[0,0], [1,0], [0,1], [1,1]])
sage: p.volume([0,1,2])
1
sage: simplex = p.triangulate()[0]  # first simplex of triangulation
sage: p.volume(simplex)
1
```

```
>>> from sage.all import *
>>> p = PointConfiguration([[Integer(0),Integer(0)], [Integer(1),Integer(0)],␣
↪[Integer(0),Integer(1)], [Integer(1),Integer(1)]])
>>> p.volume([Integer(0),Integer(1),Integer(2)])
1
>>> simplex = p.triangulate()[Integer(0)]  # first simplex of triangulation
>>> p.volume(simplex)
1
```

The square can be triangulated into two minimal simplices, so in the "integral" normalization its volume equals two:

```
sage: p.volume()
2
```

```
>>> from sage.all import *
>>> p.volume()
2
```

Note

We return $n!$ * (metric volume of the simplex) to ensure that the volume is an integer. Essentially, this normalizes things so that the volume of the standard n-simplex is 1. See [GKZ1994] page 182.

3.2 Base classes for triangulations

We provide (fast) cython implementations here.

AUTHORS:

- Volker Braun (2010-09-14): initial version.

class sage.geometry.triangulation.base.**ConnectedTriangulationsIterator**

Bases: SageObject

A Python shim for the C++-class 'triangulations'.

INPUT:

- point_configuration – a *PointConfiguration*
- seed – a regular triangulation or None (default). In the latter case, a suitable triangulation is generated automatically. Otherwise, you can explicitly specify the seed triangulation as
  - A *Triangulation* object, or

  - an iterable of iterables specifying the vertices of the simplices, or
  - an iterable of integers, which are then considered the enumerated simplices (see `simplex_to_int()`.

- `star` – either `None` (default) or an integer. If an integer is passed, all returned triangulations will be star with respect to the
- `fine` – boolean (default: `False`); whether to return only fine triangulations, that is, simplicial decompositions that make use of all the points of the configuration.

OUTPUT:

An iterator. The generated values are tuples of integers, which encode simplices of the triangulation. The output is a suitable input to `Triangulation`.

EXAMPLES:

```
sage: p = PointConfiguration([[0,0],[0,1],[1,0],[1,1],[-1,-1]])
sage: from sage.geometry.triangulation.base import ConnectedTriangulationsIterator
sage: ci = ConnectedTriangulationsIterator(p)
sage: next(ci)
(9, 10)
sage: next(ci)
(2, 3, 4, 5)
sage: next(ci)
(7, 8)
sage: next(ci)
(1, 3, 5, 7)
sage: next(ci)
Traceback (most recent call last):
...
StopIteration
```

```
>>> from sage.all import *
>>> p = PointConfiguration([[Integer(0),Integer(0)],[Integer(0),Integer(1)],
↪[Integer(1),Integer(0)],[Integer(1),Integer(1)],[-Integer(1),-Integer(1)]])
>>> from sage.geometry.triangulation.base import ConnectedTriangulationsIterator
>>> ci = ConnectedTriangulationsIterator(p)
>>> next(ci)
(9, 10)
>>> next(ci)
(2, 3, 4, 5)
>>> next(ci)
(7, 8)
>>> next(ci)
(1, 3, 5, 7)
>>> next(ci)
Traceback (most recent call last):
...
StopIteration
```

You can reconstruct the triangulation from the compressed output via:

```
sage: from sage.geometry.triangulation.element import Triangulation
sage: Triangulation((2, 3, 4, 5), p)
(<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>)
```

```
>>> from sage.all import *
>>> from sage.geometry.triangulation.element import Triangulation
>>> Triangulation((Integer(2), Integer(3), Integer(4), Integer(5)), p)
(<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>)
```

How to use the restrictions:

```
sage: ci = ConnectedTriangulationsIterator(p, fine=True)
sage: list(ci)
[(2, 3, 4, 5), (1, 3, 5, 7)]
sage: ci = ConnectedTriangulationsIterator(p, star=1)
sage: list(ci)
[(7, 8)]
sage: ci = ConnectedTriangulationsIterator(p, star=1, fine=True)
sage: list(ci)
[]
```

```
>>> from sage.all import *
>>> ci = ConnectedTriangulationsIterator(p, fine=True)
>>> list(ci)
[(2, 3, 4, 5), (1, 3, 5, 7)]
>>> ci = ConnectedTriangulationsIterator(p, star=Integer(1))
>>> list(ci)
[(7, 8)]
>>> ci = ConnectedTriangulationsIterator(p, star=Integer(1), fine=True)
>>> list(ci)
[]
```

class sage.geometry.triangulation.base.**Point**

Bases: `SageObject`

A point of a point configuration.

Note that the coordinates of the points of a point configuration are somewhat arbitrary. What counts are the abstract linear relations between the points, for example encoded by the `circuits()`.

> **Warning**
>
> You should not create `Point` objects manually. The constructor of `PointConfiguration_base` takes care of this for you.

INPUT:

- `point_configuration` – `PointConfiguration_base`; the point configuration to which the point belongs
- `i` – integer; the index of the point in the point configuration
- `projective` – the projective coordinates of the point
- `affine` – the affine coordinates of the point
- `reduced` – the reduced (with linearities removed) coordinates of the point

EXAMPLES:

```
sage: pc = PointConfiguration([(0,0)])
sage: from sage.geometry.triangulation.base import Point
sage: Point(pc, 123, (0,0,1), (0,0), ())
P(0, 0)
```

```
>>> from sage.all import *
>>> pc = PointConfiguration([(Integer(0),Integer(0))])
>>> from sage.geometry.triangulation.base import Point
>>> Point(pc, Integer(123), (Integer(0),Integer(0),Integer(1)), (Integer(0),
→Integer(0)), ())
P(0, 0)
```

affine()

Return the affine coordinates of the point in the ambient space.

OUTPUT: a tuple containing the coordinates

EXAMPLES:

```
sage: pc = PointConfiguration([[10, 0, 1], [10, 0, 0], [10, 2, 3]])
sage: p = pc.point(2); p
P(10, 2, 3)
sage: p.affine()
(10, 2, 3)
sage: p.projective()
(10, 2, 3, 1)
sage: p.reduced_affine()
(2, 2)
sage: p.reduced_projective()
(2, 2, 1)
sage: p.reduced_affine_vector()
(2, 2)
```

```
>>> from sage.all import *
>>> pc = PointConfiguration([[Integer(10), Integer(0), Integer(1)],␣
→[Integer(10), Integer(0), Integer(0)], [Integer(10), Integer(2),␣
→Integer(3)]])
>>> p = pc.point(Integer(2)); p
P(10, 2, 3)
>>> p.affine()
(10, 2, 3)
>>> p.projective()
(10, 2, 3, 1)
>>> p.reduced_affine()
(2, 2)
>>> p.reduced_projective()
(2, 2, 1)
>>> p.reduced_affine_vector()
(2, 2)
```

index()

Return the index of the point in the point configuration.

EXAMPLES:

```
sage: pc = PointConfiguration([[0, 1], [0, 0], [1, 0]])
sage: p = pc.point(2); p
P(1, 0)
sage: p.index()
2
```

```
>>> from sage.all import *
>>> pc = PointConfiguration([[Integer(0), Integer(1)], [Integer(0),␣
↪Integer(0)], [Integer(1), Integer(0)]])
>>> p = pc.point(Integer(2)); p
P(1, 0)
>>> p.index()
2
```

point_configuration()

Return the point configuration to which the point belongs.

OUTPUT: a *PointConfiguration*

EXAMPLES:

```
sage: pc = PointConfiguration([ (0,0), (1,0), (0,1) ])
sage: p = pc.point(0)
sage: p is pc.point(0)
True
sage: p.point_configuration() is pc
True
```

```
>>> from sage.all import *
>>> pc = PointConfiguration([ (Integer(0),Integer(0)), (Integer(1),
↪Integer(0)), (Integer(0),Integer(1)) ])
>>> p = pc.point(Integer(0))
>>> p is pc.point(Integer(0))
True
>>> p.point_configuration() is pc
True
```

projective()

Return the projective coordinates of the point in the ambient space.

OUTPUT: a tuple containing the coordinates

EXAMPLES:

```
sage: pc = PointConfiguration([[10, 0, 1], [10, 0, 0], [10, 2, 3]])
sage: p = pc.point(2); p
P(10, 2, 3)
sage: p.affine()
(10, 2, 3)
sage: p.projective()
(10, 2, 3, 1)
sage: p.reduced_affine()
(2, 2)
sage: p.reduced_projective()
```

(continues on next page)

(continued from previous page)

```
(2, 2, 1)
sage: p.reduced_affine_vector()
(2, 2)
```

```
>>> from sage.all import *
>>> pc = PointConfiguration([[Integer(10), Integer(0), Integer(1)],
↪[Integer(10), Integer(0), Integer(0)], [Integer(10), Integer(2),
↪Integer(3)]])
>>> p = pc.point(Integer(2)); p
P(10, 2, 3)
>>> p.affine()
(10, 2, 3)
>>> p.projective()
(10, 2, 3, 1)
>>> p.reduced_affine()
(2, 2)
>>> p.reduced_projective()
(2, 2, 1)
>>> p.reduced_affine_vector()
(2, 2)
```

reduced_affine()

Return the affine coordinates of the point on the hyperplane spanned by the point configuration.

OUTPUT: a tuple containing the coordinates

EXAMPLES:

```
sage: pc = PointConfiguration([[10, 0, 1], [10, 0, 0], [10, 2, 3]])
sage: p = pc.point(2); p
P(10, 2, 3)
sage: p.affine()
(10, 2, 3)
sage: p.projective()
(10, 2, 3, 1)
sage: p.reduced_affine()
(2, 2)
sage: p.reduced_projective()
(2, 2, 1)
sage: p.reduced_affine_vector()
(2, 2)
```

```
>>> from sage.all import *
>>> pc = PointConfiguration([[Integer(10), Integer(0), Integer(1)],
↪[Integer(10), Integer(0), Integer(0)], [Integer(10), Integer(2),
↪Integer(3)]])
>>> p = pc.point(Integer(2)); p
P(10, 2, 3)
>>> p.affine()
(10, 2, 3)
>>> p.projective()
(10, 2, 3, 1)
```

(continues on next page)

(continued from previous page)

```
>>> p.reduced_affine()
(2, 2)
>>> p.reduced_projective()
(2, 2, 1)
>>> p.reduced_affine_vector()
(2, 2)
```

reduced_affine_vector()

Return the affine coordinates of the point on the hyperplane spanned by the point configuration.

OUTPUT: a tuple containing the coordinates

EXAMPLES:

```
sage: pc = PointConfiguration([[10, 0, 1], [10, 0, 0], [10, 2, 3]])
sage: p = pc.point(2); p
P(10, 2, 3)
sage: p.affine()
(10, 2, 3)
sage: p.projective()
(10, 2, 3, 1)
sage: p.reduced_affine()
(2, 2)
sage: p.reduced_projective()
(2, 2, 1)
sage: p.reduced_affine_vector()
(2, 2)
```

```
>>> from sage.all import *
>>> pc = PointConfiguration([[Integer(10), Integer(0), Integer(1)],␣
↪[Integer(10), Integer(0), Integer(0)], [Integer(10), Integer(2),␣
↪Integer(3)]])
>>> p = pc.point(Integer(2)); p
P(10, 2, 3)
>>> p.affine()
(10, 2, 3)
>>> p.projective()
(10, 2, 3, 1)
>>> p.reduced_affine()
(2, 2)
>>> p.reduced_projective()
(2, 2, 1)
>>> p.reduced_affine_vector()
(2, 2)
```

reduced_projective()

Return the projective coordinates of the point on the hyperplane spanned by the point configuration.

OUTPUT: a tuple containing the coordinates

EXAMPLES:

```
sage: pc = PointConfiguration([[10, 0, 1], [10, 0, 0], [10, 2, 3]])
sage: p = pc.point(2); p
```

(continues on next page)

(continued from previous page)

```
P(10, 2, 3)
sage: p.affine()
(10, 2, 3)
sage: p.projective()
(10, 2, 3, 1)
sage: p.reduced_affine()
(2, 2)
sage: p.reduced_projective()
(2, 2, 1)
sage: p.reduced_affine_vector()
(2, 2)
```

```
>>> from sage.all import *
>>> pc = PointConfiguration([[Integer(10), Integer(0), Integer(1)],
↪[Integer(10), Integer(0), Integer(0)], [Integer(10), Integer(2),
↪Integer(3)]])
>>> p = pc.point(Integer(2)); p
P(10, 2, 3)
>>> p.affine()
(10, 2, 3)
>>> p.projective()
(10, 2, 3, 1)
>>> p.reduced_affine()
(2, 2)
>>> p.reduced_projective()
(2, 2, 1)
>>> p.reduced_affine_vector()
(2, 2)
```

reduced_projective_vector()

Return the affine coordinates of the point on the hyperplane spanned by the point configuration.

OUTPUT: a tuple containing the coordinates

EXAMPLES:

```
sage: pc = PointConfiguration([[10, 0, 1], [10, 0, 0], [10, 2, 3]])
sage: p = pc.point(2); p
P(10, 2, 3)
sage: p.affine()
(10, 2, 3)
sage: p.projective()
(10, 2, 3, 1)
sage: p.reduced_affine()
(2, 2)
sage: p.reduced_projective()
(2, 2, 1)
sage: p.reduced_affine_vector()
(2, 2)
sage: type(p.reduced_affine_vector())
<class 'sage.modules.vector_rational_dense.Vector_rational_dense'>
```

```
>>> from sage.all import *
>>> pc = PointConfiguration([[Integer(10), Integer(0), Integer(1)],␣
↪[Integer(10), Integer(0), Integer(0)], [Integer(10), Integer(2),␣
↪Integer(3)]])
>>> p = pc.point(Integer(2)); p
P(10, 2, 3)
>>> p.affine()
(10, 2, 3)
>>> p.projective()
(10, 2, 3, 1)
>>> p.reduced_affine()
(2, 2)
>>> p.reduced_projective()
(2, 2, 1)
>>> p.reduced_affine_vector()
(2, 2)
>>> type(p.reduced_affine_vector())
<class 'sage.modules.vector_rational_dense.Vector_rational_dense'>
```

class sage.geometry.triangulation.base.**PointConfiguration_base**

Bases: Parent

The cython abstract base class for PointConfiguration.

> **Warning**
>
> You should not instantiate this base class, but only its derived class *PointConfiguration*.

ambient_dim()

Return the dimension of the ambient space of the point configuration.

See also dimension()

EXAMPLES:

```
sage: p = PointConfiguration([[0,0,0]])
sage: p.ambient_dim()
3
sage: p.dim()
0
```

```
>>> from sage.all import *
>>> p = PointConfiguration([[Integer(0),Integer(0),Integer(0)]])
>>> p.ambient_dim()
3
>>> p.dim()
0
```

base_ring()

Return the base ring, that is, the ring containing the coordinates of the points.

OUTPUT: a ring

EXAMPLES:

```
sage: p = PointConfiguration([(0,0)])
sage: p.base_ring()
Integer Ring

sage: p = PointConfiguration([(1/2,3)])
sage: p.base_ring()
Rational Field

sage: p = PointConfiguration([(0.2, 5)])
sage: p.base_ring()
Real Field with 53 bits of precision
```

```
>>> from sage.all import *
>>> p = PointConfiguration([(Integer(0),Integer(0))])
>>> p.base_ring()
Integer Ring

>>> p = PointConfiguration([(Integer(1)/Integer(2),Integer(3))])
>>> p.base_ring()
Rational Field

>>> p = PointConfiguration([(RealNumber('0.2'), Integer(5))])
>>> p.base_ring()
Real Field with 53 bits of precision
```

dim()

Return the actual dimension of the point configuration.

See also *ambient_dim()*

EXAMPLES:

```
sage: p = PointConfiguration([[0,0,0]])
sage: p.ambient_dim()
3
sage: p.dim()
0
```

```
>>> from sage.all import *
>>> p = PointConfiguration([[Integer(0),Integer(0),Integer(0)]])
>>> p.ambient_dim()
3
>>> p.dim()
0
```

int_to_simplex(*s*)

Reverse the enumeration of possible simplices in *simplex_to_int()*.

The enumeration is compatible with [PUNTOS].

INPUT:

- s – integer that uniquely specifies a simplex

OUTPUT:

An ordered tuple consisting of the indices of the vertices of the simplex.

EXAMPLES:

```
sage: U=matrix([
....:    [ 0, 0, 0, 0, 0, 2, 4,-1, 1, 1, 0, 0, 1, 0],
....:    [ 0, 0, 0, 1, 0, 0,-1, 0, 0, 0, 0, 0, 0, 0],
....:    [ 0, 2, 0, 0, 0, 0,-1, 0, 1, 0, 1, 0, 0, 1],
....:    [ 0, 1, 1, 0, 0, 1, 0,-2, 1, 0, 0,-1, 1, 1],
....:    [ 0, 0, 0, 0, 1, 0,-1, 0, 0, 0, 0, 0, 0, 0]
....: ])
sage: pc = PointConfiguration(U.columns())
sage: pc.simplex_to_int([1,3,4,7,10,13])
1678
sage: pc.int_to_simplex(1678)
(1, 3, 4, 7, 10, 13)
```

```
>>> from sage.all import *
>>> U=matrix([
...    [ Integer(0), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(2), Integer(4),-Integer(1), Integer(1), Integer(1), Integer(0),␣
↪Integer(0), Integer(1), Integer(0)],
...    [ Integer(0), Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(0),-Integer(1), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(0), Integer(0), Integer(0)],
...    [ Integer(0), Integer(2), Integer(0), Integer(0), Integer(0),␣
↪Integer(0),-Integer(1), Integer(0), Integer(1), Integer(0), Integer(1),␣
↪Integer(0), Integer(0), Integer(1)],
...    [ Integer(0), Integer(1), Integer(1), Integer(0), Integer(0),␣
↪Integer(1), Integer(0),-Integer(2), Integer(1), Integer(0), Integer(0),-
↪Integer(1), Integer(1), Integer(1)],
...    [ Integer(0), Integer(0), Integer(0), Integer(0), Integer(1),␣
↪Integer(0),-Integer(1), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(0), Integer(0), Integer(0)]
... ])
>>> pc = PointConfiguration(U.columns())
>>> pc.simplex_to_int([Integer(1),Integer(3),Integer(4),Integer(7),
↪Integer(10),Integer(13)])
1678
>>> pc.int_to_simplex(Integer(1678))
(1, 3, 4, 7, 10, 13)
```

is_affine()

Return whether the configuration is defined by affine points.

OUTPUT: boolean; if true, the homogeneous coordinates all have 1 as their last entry

EXAMPLES:

```
sage: p = PointConfiguration([(0.2, 5), (3, 0.1)])
sage: p.is_affine()
True

sage: p = PointConfiguration([(0.2, 5, 1), (3, 0.1, 1)], projective=True)
```

(continues on next page)

(continued from previous page)

```
sage: p.is_affine()
False
```

```
>>> from sage.all import *
>>> p = PointConfiguration([(RealNumber('0.2'), Integer(5)), (Integer(3),␣
↪RealNumber('0.1'))])
>>> p.is_affine()
True

>>> p = PointConfiguration([(RealNumber('0.2'), Integer(5), Integer(1)),␣
↪(Integer(3), RealNumber('0.1'), Integer(1))], projective=True)
>>> p.is_affine()
False
```

n_points()

Return the number of points.

Same as `len(self)`.

EXAMPLES:

```
sage: p = PointConfiguration([[0,0],[0,1],[1,0],[1,1],[-1,-1]])
sage: p
A point configuration in affine 2-space over Integer Ring
consisting of 5 points. The triangulations of this point
configuration are assumed to be connected, not necessarily
fine, not necessarily regular.
sage: len(p)
5
sage: p.n_points()
5
```

```
>>> from sage.all import *
>>> p = PointConfiguration([[Integer(0),Integer(0)],[Integer(0),Integer(1)],
↪[Integer(1),Integer(0)],[Integer(1),Integer(1)],[-Integer(1),-Integer(1)]])
>>> p
A point configuration in affine 2-space over Integer Ring
consisting of 5 points. The triangulations of this point
configuration are assumed to be connected, not necessarily
fine, not necessarily regular.
>>> len(p)
5
>>> p.n_points()
5
```

point(i)

Return the i-th point of the configuration.

Same as `__getitem__()`

INPUT:

- `i` – integer

OUTPUT: a point of the point configuration

EXAMPLES:

```
sage: pconfig = PointConfiguration([[0,0],[0,1],[1,0],[1,1],[-1,-1]])
sage: list(pconfig)
[P(0, 0), P(0, 1), P(1, 0), P(1, 1), P(-1, -1)]
sage: [ p for p in pconfig.points() ]
[P(0, 0), P(0, 1), P(1, 0), P(1, 1), P(-1, -1)]
sage: pconfig.point(0)
P(0, 0)
sage: pconfig[0]
P(0, 0)
sage: pconfig.point(1)
P(0, 1)
sage: pconfig.point( pconfig.n_points()-1 )
P(-1, -1)
```

```
>>> from sage.all import *
>>> pconfig = PointConfiguration([[Integer(0),Integer(0)],[Integer(0),
→Integer(1)],[Integer(1),Integer(0)],[Integer(1),Integer(1)],[-Integer(1),-
→Integer(1)]])
>>> list(pconfig)
[P(0, 0), P(0, 1), P(1, 0), P(1, 1), P(-1, -1)]
>>> [ p for p in pconfig.points() ]
[P(0, 0), P(0, 1), P(1, 0), P(1, 1), P(-1, -1)]
>>> pconfig.point(Integer(0))
P(0, 0)
>>> pconfig[Integer(0)]
P(0, 0)
>>> pconfig.point(Integer(1))
P(0, 1)
>>> pconfig.point( pconfig.n_points()-Integer(1) )
P(-1, -1)
```

points()

Return a list of the points.

OUTPUT:

A list of the points. See also the `__iter__()` method, which returns the corresponding generator.

EXAMPLES:

```
sage: pconfig = PointConfiguration([[0,0],[0,1],[1,0],[1,1],[-1,-1]])
sage: list(pconfig)
[P(0, 0), P(0, 1), P(1, 0), P(1, 1), P(-1, -1)]
sage: [ p for p in pconfig.points() ]
[P(0, 0), P(0, 1), P(1, 0), P(1, 1), P(-1, -1)]
sage: pconfig.point(0)
P(0, 0)
sage: pconfig.point(1)
P(0, 1)
sage: pconfig.point( pconfig.n_points()-1 )
P(-1, -1)
```

```
>>> from sage.all import *
>>> pconfig = PointConfiguration([[Integer(0),Integer(0)],[Integer(0),
↪Integer(1)],[Integer(1),Integer(0)],[Integer(1),Integer(1)],[-Integer(1),-
↪Integer(1)]])
>>> list(pconfig)
[P(0, 0), P(0, 1), P(1, 0), P(1, 1), P(-1, -1)]
>>> [ p for p in pconfig.points() ]
[P(0, 0), P(0, 1), P(1, 0), P(1, 1), P(-1, -1)]
>>> pconfig.point(Integer(0))
P(0, 0)
>>> pconfig.point(Integer(1))
P(0, 1)
>>> pconfig.point( pconfig.n_points()-Integer(1) )
P(-1, -1)
```

reduced_affine_vector_space()

Return the vector space that contains the affine points.

OUTPUT: a vector space over the fraction field of *base_ring()*

EXAMPLES:

```
sage: p = PointConfiguration([[0,0,0], [1,2,3]])
sage: p.base_ring()
Integer Ring
sage: p.reduced_affine_vector_space()
Vector space of dimension 1 over Rational Field
sage: p.reduced_projective_vector_space()
Vector space of dimension 2 over Rational Field
```

```
>>> from sage.all import *
>>> p = PointConfiguration([[Integer(0),Integer(0),Integer(0)], [Integer(1),
↪Integer(2),Integer(3)]])
>>> p.base_ring()
Integer Ring
>>> p.reduced_affine_vector_space()
Vector space of dimension 1 over Rational Field
>>> p.reduced_projective_vector_space()
Vector space of dimension 2 over Rational Field
```

reduced_projective_vector_space()

Return the vector space that is spanned by the homogeneous coordinates.

OUTPUT: a vector space over the fraction field of *base_ring()*

EXAMPLES:

```
sage: p = PointConfiguration([[0,0,0], [1,2,3]])
sage: p.base_ring()
Integer Ring
sage: p.reduced_affine_vector_space()
Vector space of dimension 1 over Rational Field
sage: p.reduced_projective_vector_space()
Vector space of dimension 2 over Rational Field
```

```
>>> from sage.all import *
>>> p = PointConfiguration([[Integer(0),Integer(0),Integer(0)], [Integer(1),
↪Integer(2),Integer(3)]])
>>> p.base_ring()
Integer Ring
>>> p.reduced_affine_vector_space()
Vector space of dimension 1 over Rational Field
>>> p.reduced_projective_vector_space()
Vector space of dimension 2 over Rational Field
```

simplex_to_int(*simplex*)

Return an integer that uniquely identifies the given simplex.

See also the inverse method `int_to_simplex()`.

The enumeration is compatible with [PUNTOS].

INPUT:

- `simplex` – iterable, for example a list. The elements are the vertex indices of the simplex

OUTPUT: integer that uniquely specifies the simplex

EXAMPLES:

```
sage: U=matrix([
....:    [ 0, 0, 0, 0, 0, 2, 4,-1, 1, 1, 0, 0, 1, 0],
....:    [ 0, 0, 0, 1, 0, 0,-1, 0, 0, 0, 0, 0, 0, 0],
....:    [ 0, 2, 0, 0, 0, 0,-1, 0, 1, 0, 1, 0, 0, 1],
....:    [ 0, 1, 1, 0, 0, 1, 0,-2, 1, 0, 0,-1, 1, 1],
....:    [ 0, 0, 0, 0, 1, 0,-1, 0, 0, 0, 0, 0, 0, 0]
....: ])
sage: pc = PointConfiguration(U.columns())
sage: pc.simplex_to_int([1,3,4,7,10,13])
1678
sage: pc.int_to_simplex(1678)
(1, 3, 4, 7, 10, 13)
```

```
>>> from sage.all import *
>>> U=matrix([
...     [ Integer(0), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(2), Integer(4),-Integer(1), Integer(1), Integer(1), Integer(0),␣
↪Integer(0), Integer(1), Integer(0)],
...     [ Integer(0), Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(0),-Integer(1), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(0), Integer(0), Integer(0)],
...     [ Integer(0), Integer(2), Integer(0), Integer(0), Integer(0),␣
↪Integer(0),-Integer(1), Integer(0), Integer(1), Integer(0), Integer(1),␣
↪Integer(0), Integer(0), Integer(1)],
...     [ Integer(0), Integer(1), Integer(1), Integer(0), Integer(0),␣
↪Integer(1), Integer(0),-Integer(2), Integer(1), Integer(0), Integer(0),-
↪Integer(1), Integer(1), Integer(1)],
...     [ Integer(0), Integer(0), Integer(0), Integer(0), Integer(1),␣
↪Integer(0),-Integer(1), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(0), Integer(0), Integer(0)]
```

(continues on next page)

(continued from previous page)

```
... ])
>>> pc = PointConfiguration(U.columns())
>>> pc.simplex_to_int([Integer(1),Integer(3),Integer(4),Integer(7),
↪Integer(10),Integer(13)])
1678
>>> pc.int_to_simplex(Integer(1678))
(1, 3, 4, 7, 10, 13)
```

3.3 A triangulation

In Sage, the `PointConfiguration` and `Triangulation` satisfy a parent/element relationship. In particular, each triangulation refers back to its point configuration. If you want to triangulate a point configuration, you should construct a point configuration first and then use one of its methods to triangulate it according to your requirements. You should never have to construct a `Triangulation` object directly.

EXAMPLES:

First, we select the internal implementation for enumerating triangulations:

```
sage: PointConfiguration.set_engine('internal')   # to make doctests independent of␣
↪TOPCOM
```

```
>>> from sage.all import *
>>> PointConfiguration.set_engine('internal')   # to make doctests independent of␣
↪TOPCOM
```

Here is a simple example of how to triangulate a point configuration:

```
sage: p = [[0,-1,-1],[0,0,1],[0,1,0], [1,-1,-1],[1,0,1],[1,1,0]]
sage: points = PointConfiguration(p)
sage: triang = points.triangulate();  triang
(<0,1,2,5>, <0,1,3,5>, <1,3,4,5>)
sage: triang.plot(axes=False)                                                  #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> p = [[Integer(0),-Integer(1),-Integer(1)],[Integer(0),Integer(0),Integer(1)],
↪[Integer(0),Integer(1),Integer(0)], [Integer(1),-Integer(1),-Integer(1)],
↪[Integer(1),Integer(0),Integer(1)],[Integer(1),Integer(1),Integer(0)]]
>>> points = PointConfiguration(p)
>>> triang = points.triangulate();  triang
(<0,1,2,5>, <0,1,3,5>, <1,3,4,5>)
>>> triang.plot(axes=False)                                                    #␣
↪needs sage.plot
Graphics3d Object
```

See `sage.geometry.triangulation.point_configuration` for more details.

class `sage.geometry.triangulation.element.`**`Triangulation`**(*triangulation*, *parent*, *check=True*)

Bases: `Element`

A triangulation of a `PointConfiguration`.

Warning

You should never create `Triangulation` objects manually. See `triangulate()` and `triangulations()` to triangulate point configurations.

adjacency_graph()

Return a graph showing which simplices are adjacent in the triangulation.

OUTPUT:

A graph consisting of vertices referring to the simplices in the triangulation, and edges showing which simplices are adjacent to each other.

See also

- To obtain the triangulation's 1-skeleton, use `SimplicialComplex.graph()` through `MyTriangulation.simplicial_complex().graph()`.

AUTHORS:

- Stephen Farley (2013-08-10): initial version

EXAMPLES:

```
sage: p = PointConfiguration([[1,0,0], [0,1,0], [0,0,1], [-1,0,1],
....:                         [1,0,-1], [-1,0,0], [0,-1,0], [0,0,-1]])
sage: t = p.triangulate()
sage: t.adjacency_graph()                                                   #␣
↪needs sage.graphs
Graph on 8 vertices
```

```
>>> from sage.all import *
>>> p = PointConfiguration([[Integer(1),Integer(0),Integer(0)], [Integer(0),
↪Integer(1),Integer(0)], [Integer(0),Integer(0),Integer(1)], [-Integer(1),
↪Integer(0),Integer(1)],
...                         [Integer(1),Integer(0),-Integer(1)], [-Integer(1),
↪Integer(0),Integer(0)], [Integer(0),-Integer(1),Integer(0)], [Integer(0),
↪Integer(0),-Integer(1)]])
>>> t = p.triangulate()
>>> t.adjacency_graph()                                                     #␣
↪needs sage.graphs
Graph on 8 vertices
```

boundary()

Return the boundary of the triangulation.

OUTPUT:

The outward-facing boundary simplices (of dimension $d-1$) of the d-dimensional triangulation as a set. Each boundary is returned by a tuple of point indices.

EXAMPLES:

```
sage: triangulation = polytopes.cube().triangulate(engine='internal')
sage: triangulation
(<0,1,2,7>, <0,1,5,7>, <0,2,3,7>, <0,3,4,7>, <0,4,5,7>, <1,5,6,7>)
sage: triangulation.boundary()
frozenset({(0, 1, 2),
           (0, 1, 5),
           (0, 2, 3),
           (0, 3, 4),
           (0, 4, 5),
           (1, 2, 7),
           (1, 5, 6),
           (1, 6, 7),
           (2, 3, 7),
           (3, 4, 7),
           (4, 5, 7),
           (5, 6, 7)})
sage: triangulation.interior_facets()
frozenset({(0, 1, 7), (0, 2, 7), (0, 3, 7), (0, 4, 7), (0, 5, 7), (1, 5, 7)})
```

```
>>> from sage.all import *
>>> triangulation = polytopes.cube().triangulate(engine='internal')
>>> triangulation
(<0,1,2,7>, <0,1,5,7>, <0,2,3,7>, <0,3,4,7>, <0,4,5,7>, <1,5,6,7>)
>>> triangulation.boundary()
frozenset({(0, 1, 2),
           (0, 1, 5),
           (0, 2, 3),
           (0, 3, 4),
           (0, 4, 5),
           (1, 2, 7),
           (1, 5, 6),
           (1, 6, 7),
           (2, 3, 7),
           (3, 4, 7),
           (4, 5, 7),
           (5, 6, 7)})
>>> triangulation.interior_facets()
frozenset({(0, 1, 7), (0, 2, 7), (0, 3, 7), (0, 4, 7), (0, 5, 7), (1, 5, 7)})
```

boundary_polyhedral_complex(***kwds*)

Return the boundary of `self` as a `PolyhedralComplex`.

OUTPUT:

A `PolyhedralComplex` whose maximal cells are the simplices of the boundary of `self`.

EXAMPLES:

```
sage: P = polytopes.cube()
sage: pc = PointConfiguration(P.vertices())
sage: T = pc.placing_triangulation(); T
(<0,1,2,7>, <0,1,5,7>, <0,2,3,7>, <0,3,4,7>, <0,4,5,7>, <1,5,6,7>)
sage: bd_C = T.boundary_polyhedral_complex(); bd_C                    #␣
↪needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
Polyhedral complex with 12 maximal cells
sage: [P.vertices_list() for P in bd_C.maximal_cells_sorted()]             #␣
↪needs sage.graphs
[[[-1, -1, -1], [-1, -1, 1], [-1, 1, 1]],
 [[-1, -1, -1], [-1, -1, 1], [1, -1, -1]],
 [[-1, -1, -1], [-1, 1, -1], [-1, 1, 1]],
 [[-1, -1, -1], [-1, 1, -1], [1, 1, -1]],
 [[-1, -1, -1], [1, -1, -1], [1, 1, -1]],
 [[-1, -1, 1], [-1, 1, 1], [1, -1, 1]],
 [[-1, -1, 1], [1, -1, -1], [1, -1, 1]],
 [[-1, 1, -1], [-1, 1, 1], [1, 1, -1]],
 [[-1, 1, 1], [1, -1, 1], [1, 1, 1]],
 [[-1, 1, 1], [1, 1, -1], [1, 1, 1]],
 [[1, -1, -1], [1, -1, 1], [1, 1, 1]],
 [[1, -1, -1], [1, 1, -1], [1, 1, 1]]]
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> pc = PointConfiguration(P.vertices())
>>> T = pc.placing_triangulation(); T
(<0,1,2,7>, <0,1,5,7>, <0,2,3,7>, <0,3,4,7>, <0,4,5,7>, <1,5,6,7>)
>>> bd_C = T.boundary_polyhedral_complex(); bd_C                          #␣
↪needs sage.graphs
Polyhedral complex with 12 maximal cells
>>> [P.vertices_list() for P in bd_C.maximal_cells_sorted()]              #␣
↪needs sage.graphs
[[[-1, -1, -1], [-1, -1, 1], [-1, 1, 1]],
 [[-1, -1, -1], [-1, -1, 1], [1, -1, -1]],
 [[-1, -1, -1], [-1, 1, -1], [-1, 1, 1]],
 [[-1, -1, -1], [-1, 1, -1], [1, 1, -1]],
 [[-1, -1, -1], [1, -1, -1], [1, 1, -1]],
 [[-1, -1, 1], [-1, 1, 1], [1, -1, 1]],
 [[-1, -1, 1], [1, -1, -1], [1, -1, 1]],
 [[-1, 1, -1], [-1, 1, 1], [1, 1, -1]],
 [[-1, 1, 1], [1, -1, 1], [1, 1, 1]],
 [[-1, 1, 1], [1, 1, -1], [1, 1, 1]],
 [[1, -1, -1], [1, -1, 1], [1, 1, 1]],
 [[1, -1, -1], [1, 1, -1], [1, 1, 1]]]
```

It is a subcomplex of `self` as a *polyhedral_complex()*:

```
sage: C = T.polyhedral_complex()                                            #␣
↪needs sage.graphs
sage: bd_C.is_subcomplex(C)                                                 #␣
↪needs sage.graphs
True
```

```
>>> from sage.all import *
>>> C = T.polyhedral_complex()                                             #␣
↪needs sage.graphs
>>> bd_C.is_subcomplex(C)                                                  #␣
↪needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
True
```

boundary_simplicial_complex()

Return the boundary of `self` as an (abstract) simplicial complex.

OUTPUT: a `SimplicialComplex`

EXAMPLES:

```
sage: p = polytopes.cuboctahedron()
sage: triangulation = p.triangulate(engine='internal')
sage: bd_sc = triangulation.boundary_simplicial_complex(); bd_sc             #␣
↪needs sage.graphs
Simplicial complex with 12 vertices and 20 facets
```

```
>>> from sage.all import *
>>> p = polytopes.cuboctahedron()
>>> triangulation = p.triangulate(engine='internal')
>>> bd_sc = triangulation.boundary_simplicial_complex(); bd_sc             #␣
↪needs sage.graphs
Simplicial complex with 12 vertices and 20 facets
```

The boundary of every convex set is a topological sphere, so it has spherical homology:

```
sage: bd_sc.homology()                                                       #␣
↪needs sage.graphs
{0: 0, 1: 0, 2: Z}
```

```
>>> from sage.all import *
>>> bd_sc.homology()                                                       #␣
↪needs sage.graphs
{0: 0, 1: 0, 2: Z}
```

It is a subcomplex of `self` as a *simplicial_complex()*:

```
sage: sc = triangulation.simplicial_complex()                                #␣
↪needs sage.graphs
sage: all(f in sc for f in bd_sc.maximal_faces())                            #␣
↪needs sage.graphs
True
```

```
>>> from sage.all import *
>>> sc = triangulation.simplicial_complex()                                #␣
↪needs sage.graphs
>>> all(f in sc for f in bd_sc.maximal_faces())                            #␣
↪needs sage.graphs
True
```

enumerate_simplices()

Return the enumerated simplices.

OUTPUT: a tuple of integers that uniquely specifies the triangulation

EXAMPLES:

```
sage: pc = PointConfiguration(matrix([
....:     [ 0, 0, 0, 0, 0, 2, 4,-1, 1, 1, 0, 0, 1, 0],
....:     [ 0, 0, 0, 1, 0, 0,-1, 0, 0, 0, 0, 0, 0, 0],
....:     [ 0, 2, 0, 0, 0, 0,-1, 0, 1, 0, 1, 0, 0, 1],
....:     [ 0, 1, 1, 0, 0, 1, 0,-2, 1, 0, 0,-1, 1, 1],
....:     [ 0, 0, 0, 0, 1, 0,-1, 0, 0, 0, 0, 0, 0, 0]
....: ]).columns())
sage: triangulation = pc.lexicographic_triangulation()
sage: triangulation.enumerate_simplices()
(1678, 1688, 1769, 1779, 1895, 1905, 2112, 2143, 2234, 2360, 2555, 2580,
 2610, 2626, 2650, 2652, 2654, 2661, 2663, 2667, 2685, 2755, 2757, 2759,
 2766, 2768, 2772, 2811, 2881, 2883, 2885, 2892, 2894, 2898)
```

```
>>> from sage.all import *
>>> pc = PointConfiguration(matrix([
...     [ Integer(0), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(2), Integer(4),-Integer(1), Integer(1), Integer(1), Integer(0),␣
↪Integer(0), Integer(1), Integer(0)],
...     [ Integer(0), Integer(0), Integer(0), Integer(1), Integer(0),␣
↪Integer(0),-Integer(1), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(0), Integer(0), Integer(0)],
...     [ Integer(0), Integer(2), Integer(0), Integer(0), Integer(0),␣
↪Integer(0),-Integer(1), Integer(0), Integer(1), Integer(0), Integer(1),␣
↪Integer(0), Integer(0), Integer(1)],
...     [ Integer(0), Integer(1), Integer(1), Integer(0), Integer(0),␣
↪Integer(1), Integer(0),-Integer(2), Integer(1), Integer(0), Integer(0),-
↪Integer(1), Integer(1), Integer(1)],
...     [ Integer(0), Integer(0), Integer(0), Integer(0), Integer(1),␣
↪Integer(0),-Integer(1), Integer(0), Integer(0), Integer(0), Integer(0),␣
↪Integer(0), Integer(0), Integer(0)]
... ]).columns())
>>> triangulation = pc.lexicographic_triangulation()
>>> triangulation.enumerate_simplices()
(1678, 1688, 1769, 1779, 1895, 1905, 2112, 2143, 2234, 2360, 2555, 2580,
 2610, 2626, 2650, 2652, 2654, 2661, 2663, 2667, 2685, 2755, 2757, 2759,
 2766, 2768, 2772, 2811, 2881, 2883, 2885, 2892, 2894, 2898)
```

You can recreate the triangulation from this list by passing it to the constructor:

```
sage: from sage.geometry.triangulation.point_configuration import␣
↪Triangulation
sage: Triangulation([1678, 1688, 1769, 1779, 1895, 1905, 2112, 2143,
....:  2234, 2360, 2555, 2580, 2610, 2626, 2650, 2652, 2654, 2661, 2663,
....:  2667, 2685, 2755, 2757, 2759, 2766, 2768, 2772, 2811, 2881, 2883,
....:  2885, 2892, 2894, 2898], pc)
(<1,3,4,7,10,13>, <1,3,4,8,10,13>, <1,3,6,7,10,13>, <1,3,6,8,10,13>,
 <1,4,6,7,10,13>, <1,4,6,8,10,13>, <2,3,4,6,7,12>, <2,3,4,7,12,13>,
 <2,3,6,7,12,13>, <2,4,6,7,12,13>, <3,4,5,6,9,12>, <3,4,5,8,9,12>,
 <3,4,6,7,11,12>, <3,4,6,9,11,12>, <3,4,7,10,11,13>, <3,4,7,11,12,13>,
 <3,4,8,9,10,12>, <3,4,8,10,12,13>, <3,4,9,10,11,12>, <3,4,10,11,12,13>,
 <3,5,6,8,9,12>, <3,6,7,10,11,13>, <3,6,7,11,12,13>, <3,6,8,9,10,12>,
 <3,6,8,10,12,13>, <3,6,9,10,11,12>, <3,6,10,11,12,13>, <4,5,6,8,9,12>,
 <4,6,7,10,11,13>, <4,6,7,11,12,13>, <4,6,8,9,10,12>, <4,6,8,10,12,13>,
```

(continues on next page)

(continued from previous page)

```
 <4,6,9,10,11,12>, <4,6,10,11,12,13>)
```

```
>>> from sage.all import *
>>> from sage.geometry.triangulation.point_configuration import Triangulation
>>> Triangulation([Integer(1678), Integer(1688), Integer(1769), Integer(1779),
↪ Integer(1895), Integer(1905), Integer(2112), Integer(2143),
...  Integer(2234), Integer(2360), Integer(2555), Integer(2580),␣
↪Integer(2610), Integer(2626), Integer(2650), Integer(2652), Integer(2654),␣
↪Integer(2661), Integer(2663),
...  Integer(2667), Integer(2685), Integer(2755), Integer(2757),␣
↪Integer(2759), Integer(2766), Integer(2768), Integer(2772), Integer(2811),␣
↪Integer(2881), Integer(2883),
...  Integer(2885), Integer(2892), Integer(2894), Integer(2898)], pc)
(<1,3,4,7,10,13>, <1,3,4,8,10,13>, <1,3,6,7,10,13>, <1,3,6,8,10,13>,
 <1,4,6,7,10,13>, <1,4,6,8,10,13>, <2,3,4,6,7,12>, <2,3,4,7,12,13>,
 <2,3,6,7,12,13>, <2,4,6,7,12,13>, <3,4,5,6,9,12>, <3,4,5,8,9,12>,
 <3,4,6,7,11,12>, <3,4,6,9,11,12>, <3,4,7,10,11,13>, <3,4,7,11,12,13>,
 <3,4,8,9,10,12>, <3,4,8,10,12,13>, <3,4,9,10,11,12>, <3,4,10,11,12,13>,
 <3,5,6,8,9,12>, <3,6,7,10,11,13>, <3,6,7,11,12,13>, <3,6,8,9,10,12>,
 <3,6,8,10,12,13>, <3,6,9,10,11,12>, <3,6,10,11,12,13>, <4,5,6,8,9,12>,
 <4,6,7,10,11,13>, <4,6,7,11,12,13>, <4,6,8,9,10,12>, <4,6,8,10,12,13>,
 <4,6,9,10,11,12>, <4,6,10,11,12,13>)
```

fan(*origin=None*)

Construct the fan of cones over the simplices of the triangulation.

INPUT:

- `origin` – `None` (default) or coordinates of a point. The common apex of all cones of the fan. If `None`, the triangulation must be a star triangulation and the distinguished central point is used as the origin.

OUTPUT:

A *`RationalPolyhedralFan`*. The coordinates of the points are shifted so that the apex of the fan is the origin of the coordinate system.

> **Note**
>
> If the set of cones over the simplices is not a fan, a suitable exception is raised.

EXAMPLES:

```
sage: pc = PointConfiguration([(0,0), (1,0), (0,1), (-1,-1)], star=0,␣
↪fine=True)
sage: triangulation = pc.triangulate()
sage: fan = triangulation.fan(); fan
Rational polyhedral fan in 2-d lattice N
sage: fan.is_equivalent(toric_varieties.P2().fan())                    #␣
↪needs palp sage.graphs
True
```

```
>>> from sage.all import *
>>> pc = PointConfiguration([(Integer(0),Integer(0)), (Integer(1),Integer(0)),
↪ (Integer(0),Integer(1)), (-Integer(1),-Integer(1))], star=Integer(0),␣
↪fine=True)
>>> triangulation = pc.triangulate()
>>> fan = triangulation.fan(); fan
Rational polyhedral fan in 2-d lattice N
>>> fan.is_equivalent(toric_varieties.P2().fan())                      #␣
↪needs palp sage.graphs
True
```

Toric diagrams (the $\mathbf{Z}_5$ hyperconifold):

```
sage: vertices=[(0, 1, 0), (0, 3, 1), (0, 2, 3), (0, 0, 2)]
sage: interior=[(0, 1, 1), (0, 1, 2), (0, 2, 1), (0, 2, 2)]
sage: points = vertices + interior
sage: pc = PointConfiguration(points, fine=True)
sage: triangulation = pc.triangulate()
sage: fan = triangulation.fan((-1,0,0)); fan
Rational polyhedral fan in 3-d lattice N
sage: fan.rays()
N(1, 1, 0),
N(1, 3, 1),
N(1, 2, 3),
N(1, 0, 2),
N(1, 1, 1),
N(1, 1, 2),
N(1, 2, 1),
N(1, 2, 2)
in 3-d lattice N
```

```
>>> from sage.all import *
>>> vertices=[(Integer(0), Integer(1), Integer(0)), (Integer(0), Integer(3),␣
↪Integer(1)), (Integer(0), Integer(2), Integer(3)), (Integer(0), Integer(0),␣
↪Integer(2))]
>>> interior=[(Integer(0), Integer(1), Integer(1)), (Integer(0), Integer(1),␣
↪Integer(2)), (Integer(0), Integer(2), Integer(1)), (Integer(0), Integer(2),␣
↪Integer(2))]
>>> points = vertices + interior
>>> pc = PointConfiguration(points, fine=True)
>>> triangulation = pc.triangulate()
>>> fan = triangulation.fan((-Integer(1),Integer(0),Integer(0))); fan
Rational polyhedral fan in 3-d lattice N
>>> fan.rays()
N(1, 1, 0),
N(1, 3, 1),
N(1, 2, 3),
N(1, 0, 2),
N(1, 1, 1),
N(1, 1, 2),
N(1, 2, 1),
N(1, 2, 2)
in 3-d lattice N
```

gkz_phi()

Calculate the GKZ phi vector of the triangulation.

The phi vector is a vector of length equals to the number of points in the point configuration. For a fixed triangulation T, the entry corresponding to the i-th point p_i is

$$\phi_T(p_i) = \sum_{t \in T, t \ni p_i} Vol(t)$$

that is, the total volume of all simplices containing p_i. See also [GKZ1994] page 220 equation 1.4.

OUTPUT: the phi vector of self

EXAMPLES:

```
sage: p = PointConfiguration([[0,0],[1,0],[2,1],[1,2],[0,1]])
sage: p.triangulate().gkz_phi()
(3, 1, 5, 2, 4)
sage: p.lexicographic_triangulation().gkz_phi()
(1, 3, 4, 2, 5)
```

```
>>> from sage.all import *
>>> p = PointConfiguration([[Integer(0),Integer(0)],[Integer(1),Integer(0)],
↪[Integer(2),Integer(1)],[Integer(1),Integer(2)],[Integer(0),Integer(1)]])
>>> p.triangulate().gkz_phi()
(3, 1, 5, 2, 4)
>>> p.lexicographic_triangulation().gkz_phi()
(1, 3, 4, 2, 5)
```

interior_facets()

Return the interior facets of the triangulation.

OUTPUT:

The inward-facing boundary simplices (of dimension $d-1$) of the d-dimensional triangulation as a set. Each boundary is returned by a tuple of point indices.

EXAMPLES:

```
sage: triangulation = polytopes.cube().triangulate(engine='internal')
sage: triangulation
(<0,1,2,7>, <0,1,5,7>, <0,2,3,7>, <0,3,4,7>, <0,4,5,7>, <1,5,6,7>)
sage: triangulation.boundary()
frozenset({(0, 1, 2),
           (0, 1, 5),
           (0, 2, 3),
           (0, 3, 4),
           (0, 4, 5),
           (1, 2, 7),
           (1, 5, 6),
           (1, 6, 7),
           (2, 3, 7),
           (3, 4, 7),
           (4, 5, 7),
           (5, 6, 7)})
sage: triangulation.interior_facets()
frozenset({(0, 1, 7), (0, 2, 7), (0, 3, 7), (0, 4, 7), (0, 5, 7), (1, 5, 7)})
```

```
>>> from sage.all import *
>>> triangulation = polytopes.cube().triangulate(engine='internal')
>>> triangulation
(<0,1,2,7>, <0,1,5,7>, <0,2,3,7>, <0,3,4,7>, <0,4,5,7>, <1,5,6,7>)
>>> triangulation.boundary()
frozenset({(0, 1, 2),
           (0, 1, 5),
           (0, 2, 3),
           (0, 3, 4),
           (0, 4, 5),
           (1, 2, 7),
           (1, 5, 6),
           (1, 6, 7),
           (2, 3, 7),
           (3, 4, 7),
           (4, 5, 7),
           (5, 6, 7)})
>>> triangulation.interior_facets()
frozenset({(0, 1, 7), (0, 2, 7), (0, 3, 7), (0, 4, 7), (0, 5, 7), (1, 5, 7)})
```

normal_cone()

Return the (closure of the) normal cone of the triangulation.

Recall that a regular triangulation is one that equals the "crease lines" of a convex piecewise-linear function. This support function is not unique, for example, you can scale it by a positive constant. The set of all piecewise-linear functions with fixed creases forms an open cone. This cone can be interpreted as the cone of normal vectors at a point of the secondary polytope, which is why we call it normal cone. See [GKZ1994] Section 7.1 for details.

OUTPUT:

The closure of the normal cone. The i-th entry equals the value of the piecewise-linear function at the i-th point of the configuration.

For an irregular triangulation, the normal cone is empty. In this case, a single point (the origin) is returned.

EXAMPLES:

```
sage: triangulation = polytopes.hypercube(2).triangulate(engine='internal')
sage: triangulation
(<0,1,3>, <1,2,3>)
sage: N = triangulation.normal_cone();  N
4-d cone in 4-d lattice
sage: N.rays()
( 0,  0,  0, -1),
( 0,  0,  1,  1),
( 0,  0, -1, -1),
( 1,  0,  0,  1),
(-1,  0,  0, -1),
( 0,  1,  0, -1),
( 0, -1,  0,  1)
in Ambient free module of rank 4
over the principal ideal domain Integer Ring
sage: N.dual().rays()
(1, -1, 1, -1)
```

(continues on next page)

(continued from previous page)

```
in Ambient free module of rank 4
over the principal ideal domain Integer Ring
```

```
>>> from sage.all import *
>>> triangulation = polytopes.hypercube(Integer(2)).triangulate(engine=
↪'internal')
>>> triangulation
(<0,1,3>, <1,2,3>)
>>> N = triangulation.normal_cone();  N
4-d cone in 4-d lattice
>>> N.rays()
( 0,  0,  0, -1),
( 0,  0,  1,  1),
( 0,  0, -1, -1),
( 1,  0,  0,  1),
(-1,  0,  0, -1),
( 0,  1,  0, -1),
( 0, -1,  0,  1)
in Ambient free module of rank 4
over the principal ideal domain Integer Ring
>>> N.dual().rays()
(1, -1, 1, -1)
in Ambient free module of rank 4
over the principal ideal domain Integer Ring
```

plot(***kwds*)

Produce a graphical representation of the triangulation.

EXAMPLES:

```
sage: p = PointConfiguration([[0,0],[0,1],[1,0],[1,1],[-1,-1]])
sage: triangulation = p.triangulate()
sage: triangulation
(<1,3,4>, <2,3,4>)
sage: triangulation.plot(axes=False)                                        #␣
↪needs sage.plot
Graphics object consisting of 12 graphics primitives
```

```
>>> from sage.all import *
>>> p = PointConfiguration([[Integer(0),Integer(0)],[Integer(0),Integer(1)],
↪[Integer(1),Integer(0)],[Integer(1),Integer(1)],[-Integer(1),-Integer(1)]])
>>> triangulation = p.triangulate()
>>> triangulation
(<1,3,4>, <2,3,4>)
>>> triangulation.plot(axes=False)                                          #␣
↪needs sage.plot
Graphics object consisting of 12 graphics primitives
```

point_configuration()

Return the point configuration underlying the triangulation.

EXAMPLES:

```
sage: pconfig = PointConfiguration([[0,0],[0,1],[1,0]])
sage: pconfig
A point configuration in affine 2-space over Integer Ring
consisting of 3 points. The triangulations of this point
configuration are assumed to be connected, not necessarily
fine, not necessarily regular.
sage: triangulation = pconfig.triangulate()
sage: triangulation
(<0,1,2>)
sage: triangulation.point_configuration()
A point configuration in affine 2-space over Integer Ring
consisting of 3 points. The triangulations of this point
configuration are assumed to be connected, not necessarily
fine, not necessarily regular.
sage: pconfig == triangulation.point_configuration()
True
```

```
>>> from sage.all import *
>>> pconfig = PointConfiguration([[Integer(0),Integer(0)],[Integer(0),
↪Integer(1)],[Integer(1),Integer(0)]])
>>> pconfig
A point configuration in affine 2-space over Integer Ring
consisting of 3 points. The triangulations of this point
configuration are assumed to be connected, not necessarily
fine, not necessarily regular.
>>> triangulation = pconfig.triangulate()
>>> triangulation
(<0,1,2>)
>>> triangulation.point_configuration()
A point configuration in affine 2-space over Integer Ring
consisting of 3 points. The triangulations of this point
configuration are assumed to be connected, not necessarily
fine, not necessarily regular.
>>> pconfig == triangulation.point_configuration()
True
```

polyhedral_complex(***kwds*)

Return `self` as a *PolyhedralComplex*.

OUTPUT:

A *PolyhedralComplex* whose maximal cells are the simplices of the triangulation.

EXAMPLES:

```
sage: P = polytopes.cube()
sage: pc = PointConfiguration(P.vertices())
sage: T = pc.placing_triangulation(); T
(<0,1,2,7>, <0,1,5,7>, <0,2,3,7>, <0,3,4,7>, <0,4,5,7>, <1,5,6,7>)
sage: C = T.polyhedral_complex(); C                                     #␣
↪needs sage.graphs
Polyhedral complex with 6 maximal cells
sage: [P.vertices_list() for P in C.maximal_cells_sorted()]             #␣
↪needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
[[[-1, -1, -1], [-1, -1, 1], [-1, 1, 1], [1, -1, -1]],
 [[-1, -1, -1], [-1, 1, -1], [-1, 1, 1], [1, 1, -1]],
 [[-1, -1, -1], [-1, 1, 1], [1, -1, -1], [1, 1, -1]],
 [[-1, -1, 1], [-1, 1, 1], [1, -1, -1], [1, -1, 1]],
 [[-1, 1, 1], [1, -1, -1], [1, -1, 1], [1, 1, 1]],
 [[-1, 1, 1], [1, -1, -1], [1, 1, -1], [1, 1, 1]]]
```

```
>>> from sage.all import *
>>> P = polytopes.cube()
>>> pc = PointConfiguration(P.vertices())
>>> T = pc.placing_triangulation(); T
(<0,1,2,7>, <0,1,5,7>, <0,2,3,7>, <0,3,4,7>, <0,4,5,7>, <1,5,6,7>)
>>> C = T.polyhedral_complex(); C                                    #␣
↪needs sage.graphs
Polyhedral complex with 6 maximal cells
>>> [P.vertices_list() for P in C.maximal_cells_sorted()]            #␣
↪needs sage.graphs
[[[-1, -1, -1], [-1, -1, 1], [-1, 1, 1], [1, -1, -1]],
 [[-1, -1, -1], [-1, 1, -1], [-1, 1, 1], [1, 1, -1]],
 [[-1, -1, -1], [-1, 1, 1], [1, -1, -1], [1, 1, -1]],
 [[-1, -1, 1], [-1, 1, 1], [1, -1, -1], [1, -1, 1]],
 [[-1, 1, 1], [1, -1, -1], [1, -1, 1], [1, 1, 1]],
 [[-1, 1, 1], [1, -1, -1], [1, 1, -1], [1, 1, 1]]]
```

simplicial_complex()

Return `self` as an (abstract) simplicial complex.

OUTPUT: a `SimplicialComplex`

EXAMPLES:

```
sage: p = polytopes.cuboctahedron()
sage: sc = p.triangulate(engine='internal').simplicial_complex(); sc     #␣
↪needs sage.graphs
Simplicial complex with 12 vertices and 16 facets
```

```
>>> from sage.all import *
>>> p = polytopes.cuboctahedron()
>>> sc = p.triangulate(engine='internal').simplicial_complex(); sc     #␣
↪needs sage.graphs
Simplicial complex with 12 vertices and 16 facets
```

Any convex set is contractable, so its reduced homology groups vanish:

```
sage: sc.homology()                                                    #␣
↪needs sage.graphs
{0: 0, 1: 0, 2: 0, 3: 0}
```

```
>>> from sage.all import *
>>> sc.homology()                                                      #␣
↪needs sage.graphs
{0: 0, 1: 0, 2: 0, 3: 0}
```

sage.geometry.triangulation.element.**triangulation_render_2d**(*triangulation*, ***kwds*)

Return a graphical representation of a 2-d triangulation.

INPUT:

- `triangulation` – a *Triangulation*
- `**kwds` – keywords that are passed on to the graphics primitives

OUTPUT: a 2-d graphics object

EXAMPLES:

```
sage: points = PointConfiguration([[0,0],[0,1],[1,0],[1,1],[-1,-1]])
sage: triang = points.triangulate()
sage: triang.plot(axes=False, aspect_ratio=1)    # indirect doctest              #␣
↪needs sage.plot
Graphics object consisting of 12 graphics primitives
```

```
>>> from sage.all import *
>>> points = PointConfiguration([[Integer(0),Integer(0)],[Integer(0),Integer(1)],
↪[Integer(1),Integer(0)],[Integer(1),Integer(1)],[-Integer(1),-Integer(1)]])
>>> triang = points.triangulate()
>>> triang.plot(axes=False, aspect_ratio=Integer(1))    # indirect doctest        ␣
↪      # needs sage.plot
Graphics object consisting of 12 graphics primitives
```

sage.geometry.triangulation.element.**triangulation_render_3d**(*triangulation*, ***kwds*)

Return a graphical representation of a 3-d triangulation.

INPUT:

- `triangulation` – a *Triangulation*
- `**kwds` – keywords that are passed on to the graphics primitives

OUTPUT: a 3-d graphics object

EXAMPLES:

```
sage: p = [[0,-1,-1],[0,0,1],[0,1,0], [1,-1,-1],[1,0,1],[1,1,0]]
sage: points = PointConfiguration(p)
sage: triang = points.triangulate()
sage: triang.plot(axes=False)      # indirect doctest                             #␣
↪needs sage.plot
Graphics3d Object
```

```
>>> from sage.all import *
>>> p = [[Integer(0),-Integer(1),-Integer(1)],[Integer(0),Integer(0),Integer(1)],
↪[Integer(0),Integer(1),Integer(0)], [Integer(1),-Integer(1),-Integer(1)],
↪[Integer(1),Integer(0),Integer(1)],[Integer(1),Integer(1),Integer(0)]]
>>> points = PointConfiguration(p)
>>> triang = points.triangulate()
>>> triang.plot(axes=False)      # indirect doctest                             #␣
↪needs sage.plot
Graphics3d Object
```

CHAPTER

FOUR

MISCELLANEOUS

4.1 Abstract base classes for classes in geometry

class sage.geometry.abc.**ConvexRationalPolyhedralCone**

Bases: object

Abstract base class for *ConvexRationalPolyhedralCone*

This class is defined for the purpose of isinstance tests. It should not be instantiated.

EXAMPLES:

```
sage: import sage.geometry.abc
sage: C = cones.nonnegative_orthant(2)                                     #␣
↪needs sage.geometry.polyhedron
sage: isinstance(C, sage.geometry.abc.ConvexRationalPolyhedralCone)       #␣
↪needs sage.geometry.polyhedron
True
```

```
>>> from sage.all import *
>>> import sage.geometry.abc
>>> C = cones.nonnegative_orthant(Integer(2))                              ␣
↪     # needs sage.geometry.polyhedron
>>> isinstance(C, sage.geometry.abc.ConvexRationalPolyhedralCone)        #␣
↪needs sage.geometry.polyhedron
True
```

By design, there is a unique direct subclass:

```
sage: sage.geometry.abc.ConvexRationalPolyhedralCone.__subclasses__()     #␣
↪needs sage.geometry.polyhedron
[<class 'sage.geometry.cone.ConvexRationalPolyhedralCone'>]

sage: len(sage.geometry.abc.Polyhedron.__subclasses__()) <= 1
True
```

```
>>> from sage.all import *
>>> sage.geometry.abc.ConvexRationalPolyhedralCone.__subclasses__()      #␣
↪needs sage.geometry.polyhedron
[<class 'sage.geometry.cone.ConvexRationalPolyhedralCone'>]
```

(continues on next page)

(continued from previous page)

```
>>> len(sage.geometry.abc.Polyhedron.__subclasses__()) <= Integer(1)
True
```

class sage.geometry.abc.**LatticePolytope**

Bases: object

Abstract base class for *LatticePolytopeClass*

This class is defined for the purpose of isinstance tests. It should not be instantiated.

EXAMPLES:

```
sage: import sage.geometry.abc
sage: P = LatticePolytope([(1,2,3), (4,5,6)])                            #␣
↪needs sage.geometry.polyhedron
sage: isinstance(P, sage.geometry.abc.LatticePolytope)                   #␣
↪needs sage.geometry.polyhedron
True
```

```
>>> from sage.all import *
>>> import sage.geometry.abc
>>> P = LatticePolytope([(Integer(1),Integer(2),Integer(3)), (Integer(4),
↪Integer(5),Integer(6))])                                  # needs sage.
↪geometry.polyhedron
>>> isinstance(P, sage.geometry.abc.LatticePolytope)                    #␣
↪needs sage.geometry.polyhedron
True
```

By design, there is a unique direct subclass:

```
sage: sage.geometry.abc.LatticePolytope.__subclasses__()                 #␣
↪needs sage.geometry.polyhedron
[<class 'sage.geometry.lattice_polytope.LatticePolytopeClass'>]

sage: len(sage.geometry.abc.Polyhedron.__subclasses__()) <= 1
True
```

```
>>> from sage.all import *
>>> sage.geometry.abc.LatticePolytope.__subclasses__()                  #␣
↪needs sage.geometry.polyhedron
[<class 'sage.geometry.lattice_polytope.LatticePolytopeClass'>]

>>> len(sage.geometry.abc.Polyhedron.__subclasses__()) <= Integer(1)
True
```

class sage.geometry.abc.**Polyhedron**

Bases: object

Abstract base class for *Polyhedron_base*

This class is defined for the purpose of isinstance tests. It should not be instantiated.

EXAMPLES:

```
sage: import sage.geometry.abc
sage: P = polytopes.cube()                                                    #␣
↪needs sage.geometry.polyhedron
sage: isinstance(P, sage.geometry.abc.Polyhedron)                             #␣
↪needs sage.geometry.polyhedron
True
```

```
>>> from sage.all import *
>>> import sage.geometry.abc
>>> P = polytopes.cube()                                                    #␣
↪needs sage.geometry.polyhedron
>>> isinstance(P, sage.geometry.abc.Polyhedron)                             #␣
↪needs sage.geometry.polyhedron
True
```

By design, there is a unique direct subclass:

```
sage: sage.geometry.abc.Polyhedron.__subclasses__()                           #␣
↪needs sage.geometry.polyhedron
[<class 'sage.geometry.polyhedron.base0.Polyhedron_base0'>]

sage: len(sage.geometry.abc.Polyhedron.__subclasses__()) <= 1
True
```

```
>>> from sage.all import *
>>> sage.geometry.abc.Polyhedron.__subclasses__()                           #␣
↪needs sage.geometry.polyhedron
[<class 'sage.geometry.polyhedron.base0.Polyhedron_base0'>]

>>> len(sage.geometry.abc.Polyhedron.__subclasses__()) <= Integer(1)
True
```

4.2 Convex Sets

class sage.geometry.convex_set.**AffineHullProjectionData**(*image: Any = None*, *projection_linear_map: Any = None*, *projection_translation: Any = None*, *section_linear_map: Any = None*, *section_translation: Any = None*)

Bases: object

image: Any = None

projection_linear_map: Any = None

projection_translation: Any = None

section_linear_map: Any = None

section_translation: Any = None

class sage.geometry.convex_set.**ConvexSet_base**

Bases: SageObject, Set_base

Abstract base class for convex sets.

affine_hull(**args*, ***kwds*)

Return the affine hull of `self` as a polyhedron.

EXAMPLES:

```
sage: from sage.geometry.convex_set import ConvexSet_compact
sage: class EmbeddedDisk(ConvexSet_compact):
....:     def an_affine_basis(self):
....:         return [vector([1, 0, 0]), vector([1, 1, 0]), vector([1, 0, 1])]
sage: O = EmbeddedDisk()
sage: O.dim()
2
sage: O.affine_hull()
A 2-dimensional polyhedron in QQ^3 defined as the convex hull of 1 vertex and␣
↪2 lines
```

```
>>> from sage.all import *
>>> from sage.geometry.convex_set import ConvexSet_compact
>>> class EmbeddedDisk(ConvexSet_compact):
...     def an_affine_basis(self):
...         return [vector([Integer(1), Integer(0), Integer(0)]),␣
↪vector([Integer(1), Integer(1), Integer(0)]), vector([Integer(1),␣
↪Integer(0), Integer(1)])]
>>> O = EmbeddedDisk()
>>> O.dim()
2
>>> O.affine_hull()
A 2-dimensional polyhedron in QQ^3 defined as the convex hull of 1 vertex and␣
↪2 lines
```

affine_hull_projection(*as_convex_set=None*, *as_affine_map=False*, *orthogonal=False*, *orthonormal=False*, *extend=False*, *minimal=False*, *return_all_data=False*, ***kwds*)

Return `self` projected into its affine hull.

Each convex set is contained in some smallest affine subspace (possibly the entire ambient space) – its affine hull. We provide an affine linear map that projects the ambient space of the convex set to the standard Euclidean space of dimension of the convex set, which restricts to a bijection from the affine hull.

The projection map is not unique; some parameters control the choice of the map. Other parameters control the output of the function.

EXAMPLES:

```
sage: P = Polyhedron(vertices=[[1, 0], [0, 1]])
sage: ri_P = P.relative_interior(); ri_P
Relative interior of a 1-dimensional polyhedron in ZZ^2 defined as the convex␣
↪hull of 2 vertices
sage: ri_P.affine_hull_projection(as_affine_map=True)
(Vector space morphism represented by the matrix:
[1]
```

(continues on next page)

(continued from previous page)

```
[0]
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 1 over Rational Field,
(0))
sage: P_aff = P.affine_hull_projection(); P_aff
A 1-dimensional polyhedron in ZZ^1 defined as the convex hull of 2 vertices
sage: ri_P_aff = ri_P.affine_hull_projection(); ri_P_aff
Relative interior of a 1-dimensional polyhedron in QQ^1 defined as the convex␣
↪hull of 2 vertices
sage: ri_P_aff.closure() == P_aff
True
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[[Integer(1), Integer(0)], [Integer(0),␣
↪Integer(1)]])
>>> ri_P = P.relative_interior(); ri_P
Relative interior of a 1-dimensional polyhedron in ZZ^2 defined as the convex␣
↪hull of 2 vertices
>>> ri_P.affine_hull_projection(as_affine_map=True)
(Vector space morphism represented by the matrix:
[1]
[0]
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 1 over Rational Field,
(0))
>>> P_aff = P.affine_hull_projection(); P_aff
A 1-dimensional polyhedron in ZZ^1 defined as the convex hull of 2 vertices
>>> ri_P_aff = ri_P.affine_hull_projection(); ri_P_aff
Relative interior of a 1-dimensional polyhedron in QQ^1 defined as the convex␣
↪hull of 2 vertices
>>> ri_P_aff.closure() == P_aff
True
```

ambient()

Return the ambient convex set or space.

The default implementation delegates to *ambient_vector_space()*.

EXAMPLES:

```
sage: from sage.geometry.convex_set import ConvexSet_base
sage: class ExampleSet(ConvexSet_base):
....:     def ambient_vector_space(self, base_field=None):
....:         return (base_field or QQ)^2001
sage: ExampleSet().ambient()
Vector space of dimension 2001 over Rational Field
```

```
>>> from sage.all import *
>>> from sage.geometry.convex_set import ConvexSet_base
>>> class ExampleSet(ConvexSet_base):
...     def ambient_vector_space(self, base_field=None):
...         return (base_field or QQ)**Integer(2001)
```

(continues on next page)

(continued from previous page)

```
>>> ExampleSet().ambient()
Vector space of dimension 2001 over Rational Field
```

ambient_dim()

Return the dimension of the ambient convex set or space.

The default implementation obtains it from `ambient()`.

EXAMPLES:

```
sage: from sage.geometry.convex_set import ConvexSet_base
sage: class ExampleSet(ConvexSet_base):
....:     def ambient(self):
....:         return QQ^7
sage: ExampleSet().ambient_dim()
7
```

```
>>> from sage.all import *
>>> from sage.geometry.convex_set import ConvexSet_base
>>> class ExampleSet(ConvexSet_base):
...     def ambient(self):
...         return QQ**Integer(7)
>>> ExampleSet().ambient_dim()
7
```

ambient_dimension()

Return the dimension of the ambient convex set or space.

This is the same as `ambient_dim()`.

EXAMPLES:

```
sage: from sage.geometry.convex_set import ConvexSet_base
sage: class ExampleSet(ConvexSet_base):
....:     def ambient_dim(self):
....:         return 91
sage: ExampleSet().ambient_dimension()
91
```

```
>>> from sage.all import *
>>> from sage.geometry.convex_set import ConvexSet_base
>>> class ExampleSet(ConvexSet_base):
...     def ambient_dim(self):
...         return Integer(91)
>>> ExampleSet().ambient_dimension()
91
```

ambient_vector_space(*base_field=None*)

Return the ambient vector space.

Subclasses must provide an implementation of this method.

The default implementations of `ambient()`, `ambient_dim()`, `ambient_dimension()` use this method.

EXAMPLES:

```
sage: from sage.geometry.convex_set import ConvexSet_base
sage: C = ConvexSet_base()
sage: C.ambient_vector_space()
Traceback (most recent call last):
...
NotImplementedError: <abstract method ambient_vector_space at ...>
```

```
>>> from sage.all import *
>>> from sage.geometry.convex_set import ConvexSet_base
>>> C = ConvexSet_base()
>>> C.ambient_vector_space()
Traceback (most recent call last):
...
NotImplementedError: <abstract method ambient_vector_space at ...>
```

an_affine_basis()

Return points that form an affine basis for the affine hull.

The points are guaranteed to lie in the topological closure of `self`.

EXAMPLES:

```
sage: from sage.geometry.convex_set import ConvexSet_base
sage: C = ConvexSet_base()
sage: C.an_affine_basis()
Traceback (most recent call last):
...
TypeError: 'NotImplementedType' object is not callable
```

```
>>> from sage.all import *
>>> from sage.geometry.convex_set import ConvexSet_base
>>> C = ConvexSet_base()
>>> C.an_affine_basis()
Traceback (most recent call last):
...
TypeError: 'NotImplementedType' object is not callable
```

an_element()

Return a point of `self`.

If `self` is empty, an `EmptySetError` will be raised.

The default implementation delegates to `_some_elements_()`.

EXAMPLES:

```
sage: from sage.geometry.convex_set import ConvexSet_compact
sage: class BlueBox(ConvexSet_compact):
....:     def _some_elements_(self):
....:         yield 'blue'
....:         yield 'cyan'
sage: BlueBox().an_element()
'blue'
```

```
>>> from sage.all import *
>>> from sage.geometry.convex_set import ConvexSet_compact
>>> class BlueBox(ConvexSet_compact):
...     def _some_elements_(self):
...         yield 'blue'
...         yield 'cyan'
>>> BlueBox().an_element()
'blue'
```

cardinality()

Return the cardinality of this set.

OUTPUT: either an integer or `Infinity`

EXAMPLES:

```
sage: p = LatticePolytope([], lattice=ToricLattice(3).dual()); p
-1-d lattice polytope in 3-d lattice M
sage: p.cardinality()
0
sage: q = Polyhedron(ambient_dim=2); q
The empty polyhedron in ZZ^2
sage: q.cardinality()
0
sage: r = Polyhedron(rays=[(1, 0)]); r
A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex and
↪1 ray
sage: r.cardinality()
+Infinity
```

```
>>> from sage.all import *
>>> p = LatticePolytope([], lattice=ToricLattice(Integer(3)).dual()); p
-1-d lattice polytope in 3-d lattice M
>>> p.cardinality()
0
>>> q = Polyhedron(ambient_dim=Integer(2)); q
The empty polyhedron in ZZ^2
>>> q.cardinality()
0
>>> r = Polyhedron(rays=[(Integer(1), Integer(0))]); r
A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex and
↪1 ray
>>> r.cardinality()
+Infinity
```

cartesian_product(*other*)

Return the Cartesian product.

INPUT:

- `other` – another convex set

OUTPUT: the Cartesian product of `self` and `other`

closure()

Return the topological closure of `self`.

EXAMPLES:

```
sage: from sage.geometry.convex_set import ConvexSet_closed
sage: C = ConvexSet_closed()
sage: C.closure() is C
True
```

```
>>> from sage.all import *
>>> from sage.geometry.convex_set import ConvexSet_closed
>>> C = ConvexSet_closed()
>>> C.closure() is C
True
```

codim()

Return the codimension of `self` in `self.ambient()`.

EXAMPLES:

```
sage: P = Polyhedron(vertices=[(1,2,3)], rays=[(1,0,0)])
sage: P.codimension()
2
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[(Integer(1),Integer(2),Integer(3))],␣
↪rays=[(Integer(1),Integer(0),Integer(0))])
>>> P.codimension()
2
```

An alias is `codim()`:

```
sage: P.codim()
2
```

```
>>> from sage.all import *
>>> P.codim()
2
```

codimension()

Return the codimension of `self` in `self.ambient()`.

EXAMPLES:

```
sage: P = Polyhedron(vertices=[(1,2,3)], rays=[(1,0,0)])
sage: P.codimension()
2
```

```
>>> from sage.all import *
>>> P = Polyhedron(vertices=[(Integer(1),Integer(2),Integer(3))],␣
↪rays=[(Integer(1),Integer(0),Integer(0))])
>>> P.codimension()
2
```

An alias is `codim()`:

```
sage: P.codim()
2
```

```
>>> from sage.all import *
>>> P.codim()
2
```

contains(*point*)

Test whether `self` contains the given `point`.

INPUT:

- `point` – a point or its coordinates

dilation(*scalar*)

Return the dilated (uniformly stretched) set.

INPUT:

- `scalar` – a scalar, not necessarily in `base_ring()`

EXAMPLES:

```
sage: from sage.geometry.convex_set import ConvexSet_compact
sage: class GlorifiedPoint(ConvexSet_compact):
....:     def __init__(self, p):
....:         self._p = p
....:     def ambient_vector_space(self):
....:         return self._p.parent().vector_space()
....:     def linear_transformation(self, linear_transf):
....:         return GlorifiedPoint(linear_transf * self._p)
sage: P = GlorifiedPoint(vector([2, 3]))
sage: P.dilation(10)._p
(20, 30)
```

```
>>> from sage.all import *
>>> from sage.geometry.convex_set import ConvexSet_compact
>>> class GlorifiedPoint(ConvexSet_compact):
...     def __init__(self, p):
...         self._p = p
...     def ambient_vector_space(self):
...         return self._p.parent().vector_space()
...     def linear_transformation(self, linear_transf):
...         return GlorifiedPoint(linear_transf * self._p)
>>> P = GlorifiedPoint(vector([Integer(2), Integer(3)]))
>>> P.dilation(Integer(10))._p
(20, 30)
```

dim()

Return the dimension of `self`.

Subclasses must provide an implementation of this method or of the method `an_affine_basis()`.

dimension()

Return the dimension of `self`.

This is the same as `dim()`.

EXAMPLES:

```
sage: from sage.geometry.convex_set import ConvexSet_base
sage: class ExampleSet(ConvexSet_base):
....:     def dim(self):
....:         return 42
sage: ExampleSet().dimension()
42
```

```
>>> from sage.all import *
>>> from sage.geometry.convex_set import ConvexSet_base
>>> class ExampleSet(ConvexSet_base):
...     def dim(self):
...         return Integer(42)
>>> ExampleSet().dimension()
42
```

interior()

Return the topological interior of `self`.

EXAMPLES:

```
sage: from sage.geometry.convex_set import ConvexSet_open
sage: C = ConvexSet_open()
sage: C.interior() is C
True
```

```
>>> from sage.all import *
>>> from sage.geometry.convex_set import ConvexSet_open
>>> C = ConvexSet_open()
>>> C.interior() is C
True
```

intersection(*other*)

Return the intersection of `self` and `other`.

INPUT:

- `other` – another convex set

OUTPUT: the intersection

is_closed()

Return whether `self` is closed.

The default implementation of this method only knows that the empty set, a singleton set, and the ambient space are closed.

OUTPUT: boolean

EXAMPLES:

```
sage: from sage.geometry.convex_set import ConvexSet_base
sage: class ExampleSet(ConvexSet_base):
....:     def dim(self):
....:         return 0
```

(continues on next page)

(continued from previous page)

```
sage: ExampleSet().is_closed()
True
```

```
>>> from sage.all import *
>>> from sage.geometry.convex_set import ConvexSet_base
>>> class ExampleSet(ConvexSet_base):
...     def dim(self):
...         return Integer(0)
>>> ExampleSet().is_closed()
True
```

is_compact()

Return whether `self` is compact.

The default implementation of this method only knows that a non-closed set cannot be compact, and that the empty set and a singleton set are compact.

OUTPUT: boolean

sage: from sage.geometry.convex_set import ConvexSet_base sage: class ExampleSet(ConvexSet_base): ….: def dim(self): ….: return 0 sage: ExampleSet().is_compact() True

is_empty()

Test whether `self` is the empty set.

OUTPUT: boolean

EXAMPLES:

```
sage: p = LatticePolytope([], lattice=ToricLattice(3).dual()); p
-1-d lattice polytope in 3-d lattice M
sage: p.is_empty()
True
```

```
>>> from sage.all import *
>>> p = LatticePolytope([], lattice=ToricLattice(Integer(3)).dual()); p
-1-d lattice polytope in 3-d lattice M
>>> p.is_empty()
True
```

is_finite()

Test whether `self` is a finite set.

OUTPUT: boolean

EXAMPLES:

```
sage: p = LatticePolytope([], lattice=ToricLattice(3).dual()); p
-1-d lattice polytope in 3-d lattice M
sage: p.is_finite()
True
sage: q = Polyhedron(ambient_dim=2); q
The empty polyhedron in ZZ^2
sage: q.is_finite()
True
```

(continues on next page)

(continued from previous page)

```
sage: r = Polyhedron(rays=[(1, 0)]); r
A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex and␣
↪1 ray
sage: r.is_finite()
False
```

```
>>> from sage.all import *
>>> p = LatticePolytope([], lattice=ToricLattice(Integer(3)).dual()); p
-1-d lattice polytope in 3-d lattice M
>>> p.is_finite()
True
>>> q = Polyhedron(ambient_dim=Integer(2)); q
The empty polyhedron in ZZ^2
>>> q.is_finite()
True
>>> r = Polyhedron(rays=[(Integer(1), Integer(0))]); r
A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex and␣
↪1 ray
>>> r.is_finite()
False
```

is_full_dimensional()

Return whether `self` is full dimensional.

OUTPUT: boolean; whether the polyhedron is not contained in any strict affine subspace

EXAMPLES:

```
sage: c = Cone([(1,0)])
sage: c.is_full_dimensional()
False

sage: polytopes.hypercube(3).is_full_dimensional()
True
sage: Polyhedron(vertices=[(1,2,3)], rays=[(1,0,0)]).is_full_dimensional()
False
```

```
>>> from sage.all import *
>>> c = Cone([(Integer(1),Integer(0))])
>>> c.is_full_dimensional()
False

>>> polytopes.hypercube(Integer(3)).is_full_dimensional()
True
>>> Polyhedron(vertices=[(Integer(1),Integer(2),Integer(3))],␣
↪rays=[(Integer(1),Integer(0),Integer(0))]).is_full_dimensional()
False
```

is_open()

Return whether `self` is open.

The default implementation of this method only knows that the empty set and the ambient space are open.

OUTPUT: boolean

EXAMPLES:

```
sage: from sage.geometry.convex_set import ConvexSet_base
sage: class ExampleSet(ConvexSet_base):
....:     def is_empty(self):
....:         return False
....:     def is_universe(self):
....:         return True
sage: ExampleSet().is_open()
True
```

```
>>> from sage.all import *
>>> from sage.geometry.convex_set import ConvexSet_base
>>> class ExampleSet(ConvexSet_base):
...     def is_empty(self):
...         return False
...     def is_universe(self):
...         return True
>>> ExampleSet().is_open()
True
```

is_relatively_open()

Return whether `self` is relatively open.

The default implementation of this method only knows that open sets are also relatively open, and in addition singletons are relatively open.

OUTPUT: boolean

EXAMPLES:

```
sage: from sage.geometry.convex_set import ConvexSet_base
sage: class ExampleSet(ConvexSet_base):
....:     def is_open(self):
....:         return True
sage: ExampleSet().is_relatively_open()
True
```

```
>>> from sage.all import *
>>> from sage.geometry.convex_set import ConvexSet_base
>>> class ExampleSet(ConvexSet_base):
...     def is_open(self):
...         return True
>>> ExampleSet().is_relatively_open()
True
```

is_universe()

Test whether `self` is the whole ambient space.

OUTPUT: boolean

linear_transformation(*linear_transf*)

Return the linear transformation of `self`.

INPUT:

- `linear_transf` – a matrix

relative_interior()

Return the relative interior of `self`.

EXAMPLES:

```
sage: from sage.geometry.convex_set import ConvexSet_relatively_open
sage: C = ConvexSet_relatively_open()
sage: C.relative_interior() is C
True
```

```
>>> from sage.all import *
>>> from sage.geometry.convex_set import ConvexSet_relatively_open
>>> C = ConvexSet_relatively_open()
>>> C.relative_interior() is C
True
```

representative_point()

Return a "generic" point of `self`.

OUTPUT: a point in the relative interior of `self` as a coordinate vector

EXAMPLES:

```
sage: C = Cone([[1, 2, 0], [2, 1, 0]])
sage: C.representative_point()
(1, 1, 0)
```

```
>>> from sage.all import *
>>> C = Cone([[Integer(1), Integer(2), Integer(0)], [Integer(2), Integer(1),␣
→Integer(0)]])
>>> C.representative_point()
(1, 1, 0)
```

some_elements()

Return a list of some points of `self`.

If `self` is empty, an empty list is returned; no exception will be raised.

The default implementation delegates to `_some_elements_()`.

EXAMPLES:

```
sage: from sage.geometry.convex_set import ConvexSet_compact
sage: class BlueBox(ConvexSet_compact):
....:     def _some_elements_(self):
....:         yield 'blue'
....:         yield 'cyan'
sage: BlueBox().some_elements()
['blue', 'cyan']
```

```
>>> from sage.all import *
>>> from sage.geometry.convex_set import ConvexSet_compact
>>> class BlueBox(ConvexSet_compact):
...     def _some_elements_(self):
...         yield 'blue'
```

(continues on next page)

(continued from previous page)

```
...         yield 'cyan'
>>> BlueBox().some_elements()
['blue', 'cyan']
```

translation(*displacement*)

Return the translation of `self` by a `displacement` vector.

INPUT:

- `displacement` – a displacement vector or a list/tuple of coordinates that determines a displacement vector

class sage.geometry.convex_set.**ConvexSet_closed**

Bases: *ConvexSet_base*

Abstract base class for closed convex sets.

is_closed()

Return whether `self` is closed.

OUTPUT: boolean

EXAMPLES:

```
sage: hcube = polytopes.hypercube(5)
sage: hcube.is_closed()
True
```

```
>>> from sage.all import *
>>> hcube = polytopes.hypercube(Integer(5))
>>> hcube.is_closed()
True
```

is_open()

Return whether `self` is open.

OUTPUT: boolean

EXAMPLES:

```
sage: hcube = polytopes.hypercube(5)
sage: hcube.is_open()
False

sage: zerocube = polytopes.hypercube(0)
sage: zerocube.is_open()
True
```

```
>>> from sage.all import *
>>> hcube = polytopes.hypercube(Integer(5))
>>> hcube.is_open()
False

>>> zerocube = polytopes.hypercube(Integer(0))
>>> zerocube.is_open()
True
```

class sage.geometry.convex_set.**ConvexSet_compact**

Bases: *ConvexSet_closed*

Abstract base class for compact convex sets.

is_compact()

Return whether `self` is compact.

OUTPUT: boolean

EXAMPLES:

```
sage: cross3 = lattice_polytope.cross_polytope(3)
sage: cross3.is_compact()
True
```

```
>>> from sage.all import *
>>> cross3 = lattice_polytope.cross_polytope(Integer(3))
>>> cross3.is_compact()
True
```

is_relatively_open()

Return whether `self` is open.

OUTPUT: boolean

EXAMPLES:

```
sage: hcube = polytopes.hypercube(5)
sage: hcube.is_open()
False

sage: zerocube = polytopes.hypercube(0)
sage: zerocube.is_open()
True
```

```
>>> from sage.all import *
>>> hcube = polytopes.hypercube(Integer(5))
>>> hcube.is_open()
False

>>> zerocube = polytopes.hypercube(Integer(0))
>>> zerocube.is_open()
True
```

is_universe()

Return whether `self` is the whole ambient space.

OUTPUT: boolean

EXAMPLES:

```
sage: cross3 = lattice_polytope.cross_polytope(3)
sage: cross3.is_universe()
False
sage: point0 = LatticePolytope([[]]); point0
```

(continues on next page)

(continued from previous page)

```
0-d reflexive polytope in 0-d lattice M
sage: point0.is_universe()
True
```

```
>>> from sage.all import *
>>> cross3 = lattice_polytope.cross_polytope(Integer(3))
>>> cross3.is_universe()
False
>>> point0 = LatticePolytope([[]]); point0
0-d reflexive polytope in 0-d lattice M
>>> point0.is_universe()
True
```

class sage.geometry.convex_set.**ConvexSet_open**

Bases: *ConvexSet_relatively_open*

Abstract base class for open convex sets.

is_closed()

Return whether `self` is closed.

OUTPUT: boolean

EXAMPLES:

```
sage: from sage.geometry.convex_set import ConvexSet_open
sage: class OpenBall(ConvexSet_open):
....:     def dim(self):
....:         return 3
....:     def is_universe(self):
....:         return False
sage: OpenBall().is_closed()
False
```

```
>>> from sage.all import *
>>> from sage.geometry.convex_set import ConvexSet_open
>>> class OpenBall(ConvexSet_open):
...     def dim(self):
...         return Integer(3)
...     def is_universe(self):
...         return False
>>> OpenBall().is_closed()
False
```

is_open()

Return whether `self` is open.

OUTPUT: boolean

EXAMPLES:

```
sage: from sage.geometry.convex_set import ConvexSet_open
sage: b = ConvexSet_open()
sage: b.is_open()
True
```

```
>>> from sage.all import *
>>> from sage.geometry.convex_set import ConvexSet_open
>>> b = ConvexSet_open()
>>> b.is_open()
True
```

class sage.geometry.convex_set.**ConvexSet_relatively_open**

Bases: *ConvexSet_base*

Abstract base class for relatively open convex sets.

is_open()

Return whether `self` is open.

OUTPUT: boolean

EXAMPLES:

```
sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: ri_segment = segment.relative_interior()
sage: ri_segment.is_open()
False
```

```
>>> from sage.all import *
>>> segment = Polyhedron([[Integer(1), Integer(2)], [Integer(3), Integer(4)]])
>>> ri_segment = segment.relative_interior()
>>> ri_segment.is_open()
False
```

is_relatively_open()

Return whether `self` is relatively open.

OUTPUT: boolean

EXAMPLES:

```
sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: ri_segment = segment.relative_interior()
sage: ri_segment.is_relatively_open()
True
```

```
>>> from sage.all import *
>>> segment = Polyhedron([[Integer(1), Integer(2)], [Integer(3), Integer(4)]])
>>> ri_segment = segment.relative_interior()
>>> ri_segment.is_relatively_open()
True
```

4.3 Linear Expressions

A linear expression is just a linear polynomial in some (fixed) variables (allowing a nonzero constant term). This class only implements linear expressions for others to use.

EXAMPLES:

```
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L.<x,y,z> = LinearExpressionModule(QQ);  L
Module of linear expressions in variables x, y, z over Rational Field
sage: x + 2*y + 3*z + 4
x + 2*y + 3*z + 4
sage: L(4)
0*x + 0*y + 0*z + 4
```

```
>>> from sage.all import *
>>> from sage.geometry.linear_expression import LinearExpressionModule
>>> L = LinearExpressionModule(QQ, names=('x', 'y', 'z',)); (x, y, z,) = L._first_
↪ngens(3);  L
Module of linear expressions in variables x, y, z over Rational Field
>>> x + Integer(2)*y + Integer(3)*z + Integer(4)
x + 2*y + 3*z + 4
>>> L(Integer(4))
0*x + 0*y + 0*z + 4
```

You can also pass coefficients and a constant term to construct linear expressions:

```
sage: L([1, 2, 3], 4)
x + 2*y + 3*z + 4
sage: L([(1, 2, 3), 4])
x + 2*y + 3*z + 4
sage: L([4, 1, 2, 3])   # note: constant is first in single-tuple notation
x + 2*y + 3*z + 4
```

```
>>> from sage.all import *
>>> L([Integer(1), Integer(2), Integer(3)], Integer(4))
x + 2*y + 3*z + 4
>>> L([(Integer(1), Integer(2), Integer(3)), Integer(4)])
x + 2*y + 3*z + 4
>>> L([Integer(4), Integer(1), Integer(2), Integer(3)])   # note: constant is first␣
↪in single-tuple notation
x + 2*y + 3*z + 4
```

The linear expressions are a module over the base ring, so you can add them and multiply them with scalars:

```
sage: m = x + 2*y + 3*z + 4
sage: 2*m
2*x + 4*y + 6*z + 8
sage: m+m
2*x + 4*y + 6*z + 8
sage: m-m
0*x + 0*y + 0*z + 0
```

```
>>> from sage.all import *
>>> m = x + Integer(2)*y + Integer(3)*z + Integer(4)
>>> Integer(2)*m
2*x + 4*y + 6*z + 8
>>> m+m
2*x + 4*y + 6*z + 8
```

(continues on next page)

(continued from previous page)

```
>>> m-m
0*x + 0*y + 0*z + 0
```

class sage.geometry.linear_expression.**LinearExpression**(*parent*, *coefficients*, *constant*, *check=True*)

Bases: ModuleElement

A linear expression.

A linear expression is just a linear polynomial in some (fixed) variables.

EXAMPLES:

```
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L.<x,y,z> = LinearExpressionModule(QQ)
sage: m = L([1, 2, 3], 4); m
x + 2*y + 3*z + 4
sage: m2 = L([(1, 2, 3), 4]); m2
x + 2*y + 3*z + 4
sage: m3 = L([4, 1, 2, 3]); m3   # note: constant is first in single-tuple␣
↪notation
x + 2*y + 3*z + 4
sage: m == m2
True
sage: m2 == m3
True
sage: L.zero()
0*x + 0*y + 0*z + 0
sage: a = L([12, 2/3, -1], -2)
sage: a - m
11*x - 4/3*y - 4*z - 6
sage: LZ.<x,y,z> = LinearExpressionModule(ZZ)
sage: a - LZ([2, -1, 3], 1)
10*x + 5/3*y - 4*z - 3
```

```
>>> from sage.all import *
>>> from sage.geometry.linear_expression import LinearExpressionModule
>>> L = LinearExpressionModule(QQ, names=('x', 'y', 'z',)); (x, y, z,) = L._first_
↪ngens(3)
>>> m = L([Integer(1), Integer(2), Integer(3)], Integer(4)); m
x + 2*y + 3*z + 4
>>> m2 = L([(Integer(1), Integer(2), Integer(3)), Integer(4)]); m2
x + 2*y + 3*z + 4
>>> m3 = L([Integer(4), Integer(1), Integer(2), Integer(3)]); m3   # note:␣
↪constant is first in single-tuple notation
x + 2*y + 3*z + 4
>>> m == m2
True
>>> m2 == m3
True
>>> L.zero()
0*x + 0*y + 0*z + 0
>>> a = L([Integer(12), Integer(2)/Integer(3), -Integer(1)], -Integer(2))
>>> a - m
```

(continues on next page)

(continued from previous page)

```
11*x - 4/3*y - 4*z - 6
>>> LZ = LinearExpressionModule(ZZ, names=('x', 'y', 'z',)); (x, y, z,) = LZ._
↪first_ngens(3)
>>> a - LZ([Integer(2), -Integer(1), Integer(3)], Integer(1))
10*x + 5/3*y - 4*z - 3
```

A()

Return the coefficient vector.

OUTPUT: the coefficient vector of the linear expression

EXAMPLES:

```
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L.<x,y,z> = LinearExpressionModule(QQ)
sage: linear = L([1, 2, 3], 4);  linear
x + 2*y + 3*z + 4
sage: linear.A()
(1, 2, 3)
sage: linear.b()
4
```

```
>>> from sage.all import *
>>> from sage.geometry.linear_expression import LinearExpressionModule
>>> L = LinearExpressionModule(QQ, names=('x', 'y', 'z',)); (x, y, z,) = L._
↪first_ngens(3)
>>> linear = L([Integer(1), Integer(2), Integer(3)], Integer(4));  linear
x + 2*y + 3*z + 4
>>> linear.A()
(1, 2, 3)
>>> linear.b()
4
```

b()

Return the constant term.

OUTPUT: the constant term of the linear expression

EXAMPLES:

```
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L.<x,y,z> = LinearExpressionModule(QQ)
sage: linear = L([1, 2, 3], 4);  linear
x + 2*y + 3*z + 4
sage: linear.A()
(1, 2, 3)
sage: linear.b()
4
```

```
>>> from sage.all import *
>>> from sage.geometry.linear_expression import LinearExpressionModule
>>> L = LinearExpressionModule(QQ, names=('x', 'y', 'z',)); (x, y, z,) = L._
↪first_ngens(3)
```

(continues on next page)

(continued from previous page)

```
>>> linear = L([Integer(1), Integer(2), Integer(3)], Integer(4));  linear
x + 2*y + 3*z + 4
>>> linear.A()
(1, 2, 3)
>>> linear.b()
4
```

change_ring(*base_ring*)

Change the base ring of this linear expression.

INPUT:

- `base_ring` – a ring; the new base ring

OUTPUT: a new linear expression over the new base ring

EXAMPLES:

```
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L.<x,y,z> = LinearExpressionModule(QQ)
sage: a = x + 2*y + 3*z + 4;  a
x + 2*y + 3*z + 4
sage: a.change_ring(RDF)
1.0*x + 2.0*y + 3.0*z + 4.0
```

```
>>> from sage.all import *
>>> from sage.geometry.linear_expression import LinearExpressionModule
>>> L = LinearExpressionModule(QQ, names=('x', 'y', 'z',)); (x, y, z,) = L._
↪first_ngens(3)
>>> a = x + Integer(2)*y + Integer(3)*z + Integer(4);  a
x + 2*y + 3*z + 4
>>> a.change_ring(RDF)
1.0*x + 2.0*y + 3.0*z + 4.0
```

coefficients()

Return all coefficients.

OUTPUT:

The constant (as first entry) and coefficients of the linear terms (as subsequent entries) in a list.

EXAMPLES:

```
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L.<x,y,z> = LinearExpressionModule(QQ)
sage: linear = L([1, 2, 3], 4);  linear
x + 2*y + 3*z + 4
sage: linear.coefficients()
[4, 1, 2, 3]
```

```
>>> from sage.all import *
>>> from sage.geometry.linear_expression import LinearExpressionModule
>>> L = LinearExpressionModule(QQ, names=('x', 'y', 'z',)); (x, y, z,) = L._
↪first_ngens(3)
>>> linear = L([Integer(1), Integer(2), Integer(3)], Integer(4));  linear
```

(continues on next page)

(continued from previous page)

```
x + 2*y + 3*z + 4
>>> linear.coefficients()
[4, 1, 2, 3]
```

constant_term()

Return the constant term.

OUTPUT: the constant term of the linear expression

EXAMPLES:

```
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L.<x,y,z> = LinearExpressionModule(QQ)
sage: linear = L([1, 2, 3], 4);  linear
x + 2*y + 3*z + 4
sage: linear.A()
(1, 2, 3)
sage: linear.b()
4
```

```
>>> from sage.all import *
>>> from sage.geometry.linear_expression import LinearExpressionModule
>>> L = LinearExpressionModule(QQ, names=('x', 'y', 'z',)); (x, y, z,) = L._
↪first_ngens(3)
>>> linear = L([Integer(1), Integer(2), Integer(3)], Integer(4));  linear
x + 2*y + 3*z + 4
>>> linear.A()
(1, 2, 3)
>>> linear.b()
4
```

dense_coefficient_list()

Return all coefficients.

OUTPUT:

The constant (as first entry) and coefficients of the linear terms (as subsequent entries) in a list.

EXAMPLES:

```
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L.<x,y,z> = LinearExpressionModule(QQ)
sage: linear = L([1, 2, 3], 4);  linear
x + 2*y + 3*z + 4
sage: linear.coefficients()
[4, 1, 2, 3]
```

```
>>> from sage.all import *
>>> from sage.geometry.linear_expression import LinearExpressionModule
>>> L = LinearExpressionModule(QQ, names=('x', 'y', 'z',)); (x, y, z,) = L._
↪first_ngens(3)
>>> linear = L([Integer(1), Integer(2), Integer(3)], Integer(4));  linear
x + 2*y + 3*z + 4
```

(continues on next page)

(continued from previous page)

```
>>> linear.coefficients()
[4, 1, 2, 3]
```

evaluate(*point*)

Evaluate the linear expression.

INPUT:

- `point` – list/tuple/iterable of coordinates; the coordinates of a point

OUTPUT: the linear expression $Ax + b$ evaluated at the point x

EXAMPLES:

```
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L.<x,y> = LinearExpressionModule(QQ)
sage: ex = 2*x + 3* y + 4
sage: ex.evaluate([1,1])
9
sage: ex([1,1])     # syntactic sugar
9
sage: ex([pi, e])                                                          #␣
↪needs sage.symbolic
2*pi + 3*e + 4
```

```
>>> from sage.all import *
>>> from sage.geometry.linear_expression import LinearExpressionModule
>>> L = LinearExpressionModule(QQ, names=('x', 'y',)); (x, y,) = L._first_
↪ngens(2)
>>> ex = Integer(2)*x + Integer(3)* y + Integer(4)
>>> ex.evaluate([Integer(1),Integer(1)])
9
>>> ex([Integer(1),Integer(1)])     # syntactic sugar
9
>>> ex([pi, e])                                                          #␣
↪needs sage.symbolic
2*pi + 3*e + 4
```

monomial_coefficients(*copy=True*)

Return a dictionary whose keys are indices of basis elements in the support of `self` and whose values are the corresponding coefficients.

INPUT:

- `copy` – ignored

EXAMPLES:

```
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L.<x,y,z> = LinearExpressionModule(QQ)
sage: linear = L([1, 2, 3], 4)
sage: sorted(linear.monomial_coefficients().items(), key=lambda x: str(x[0]))
[(0, 1), (1, 2), (2, 3), ('b', 4)]
```

```
>>> from sage.all import *
>>> from sage.geometry.linear_expression import LinearExpressionModule
>>> L = LinearExpressionModule(QQ, names=('x', 'y', 'z',)); (x, y, z,) = L._
↪first_ngens(3)
>>> linear = L([Integer(1), Integer(2), Integer(3)], Integer(4))
>>> sorted(linear.monomial_coefficients().items(), key=lambda x:␣
↪str(x[Integer(0)]))
[(0, 1), (1, 2), (2, 3), ('b', 4)]
```

class sage.geometry.linear_expression.**LinearExpressionModule**(*base_ring*, *names=()*)

Bases: Parent, UniqueRepresentation

The module of linear expressions.

This is the module of linear polynomials which is the parent for linear expressions.

EXAMPLES:

```
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L = LinearExpressionModule(QQ, ('x', 'y', 'z'))
sage: L
Module of linear expressions in variables x, y, z over Rational Field
sage: L.an_element()
x + 0*y + 0*z + 0
```

```
>>> from sage.all import *
>>> from sage.geometry.linear_expression import LinearExpressionModule
>>> L = LinearExpressionModule(QQ, ('x', 'y', 'z'))
>>> L
Module of linear expressions in variables x, y, z over Rational Field
>>> L.an_element()
x + 0*y + 0*z + 0
```

Element

alias of *LinearExpression*

ambient_module()

Return the ambient module.

See also

ambient_vector_space()

OUTPUT:

The domain of the linear expressions as a free module over the base ring.

EXAMPLES:

```
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L = LinearExpressionModule(QQ, ('x', 'y', 'z'))
sage: L.ambient_module()
Vector space of dimension 3 over Rational Field
sage: M = LinearExpressionModule(ZZ, ('r', 's'))
```

(continues on next page)

(continued from previous page)

```
sage: M.ambient_module()
Ambient free module of rank 2 over the principal ideal domain Integer Ring
sage: M.ambient_vector_space()
Vector space of dimension 2 over Rational Field
```

```
>>> from sage.all import *
>>> from sage.geometry.linear_expression import LinearExpressionModule
>>> L = LinearExpressionModule(QQ, ('x', 'y', 'z'))
>>> L.ambient_module()
Vector space of dimension 3 over Rational Field
>>> M = LinearExpressionModule(ZZ, ('r', 's'))
>>> M.ambient_module()
Ambient free module of rank 2 over the principal ideal domain Integer Ring
>>> M.ambient_vector_space()
Vector space of dimension 2 over Rational Field
```

ambient_vector_space()

Return the ambient vector space.

See also

ambient_module()

OUTPUT:

The vector space (over the fraction field of the base ring) where the linear expressions live.

EXAMPLES:

```
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L = LinearExpressionModule(QQ, ('x', 'y', 'z'))
sage: L.ambient_vector_space()
Vector space of dimension 3 over Rational Field
sage: M = LinearExpressionModule(ZZ, ('r', 's'))
sage: M.ambient_module()
Ambient free module of rank 2 over the principal ideal domain Integer Ring
sage: M.ambient_vector_space()
Vector space of dimension 2 over Rational Field
```

```
>>> from sage.all import *
>>> from sage.geometry.linear_expression import LinearExpressionModule
>>> L = LinearExpressionModule(QQ, ('x', 'y', 'z'))
>>> L.ambient_vector_space()
Vector space of dimension 3 over Rational Field
>>> M = LinearExpressionModule(ZZ, ('r', 's'))
>>> M.ambient_module()
Ambient free module of rank 2 over the principal ideal domain Integer Ring
>>> M.ambient_vector_space()
Vector space of dimension 2 over Rational Field
```

basis()

Return a basis of `self`.

EXAMPLES:

```
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L = LinearExpressionModule(QQ, ('x', 'y', 'z'))
sage: list(L.basis())
[x + 0*y + 0*z + 0,
 0*x + y + 0*z + 0,
 0*x + 0*y + z + 0,
 0*x + 0*y + 0*z + 1]
```

```
>>> from sage.all import *
>>> from sage.geometry.linear_expression import LinearExpressionModule
>>> L = LinearExpressionModule(QQ, ('x', 'y', 'z'))
>>> list(L.basis())
[x + 0*y + 0*z + 0,
 0*x + y + 0*z + 0,
 0*x + 0*y + z + 0,
 0*x + 0*y + 0*z + 1]
```

change_ring(*base_ring*)

Return a new module with a changed base ring.

INPUT:

- `base_ring` – a ring; the new base ring

OUTPUT: a new linear expression over the new base ring

EXAMPLES:

```
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: M.<y> = LinearExpressionModule(ZZ)
sage: L = M.change_ring(QQ);  L
Module of linear expressions in variable y over Rational Field
```

```
>>> from sage.all import *
>>> from sage.geometry.linear_expression import LinearExpressionModule
>>> M = LinearExpressionModule(ZZ, names=('y',)); (y,) = M._first_ngens(1)
>>> L = M.change_ring(QQ);  L
Module of linear expressions in variable y over Rational Field
```

gen(i)

Return the i-th generator.

INPUT:

- `i` – integer

OUTPUT: a linear expression

EXAMPLES:

```
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L = LinearExpressionModule(QQ, ('x', 'y', 'z'))
sage: L.gen(0)
x + 0*y + 0*z + 0
```

```
>>> from sage.all import *
>>> from sage.geometry.linear_expression import LinearExpressionModule
>>> L = LinearExpressionModule(QQ, ('x', 'y', 'z'))
>>> L.gen(Integer(0))
x + 0*y + 0*z + 0
```

gens()

Return the generators of `self`.

OUTPUT: a tuple of linear expressions, one for each linear variable

EXAMPLES:

```
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L = LinearExpressionModule(QQ, ('x', 'y', 'z'))
sage: L.gens()
(x + 0*y + 0*z + 0, 0*x + y + 0*z + 0, 0*x + 0*y + z + 0)
```

```
>>> from sage.all import *
>>> from sage.geometry.linear_expression import LinearExpressionModule
>>> L = LinearExpressionModule(QQ, ('x', 'y', 'z'))
>>> L.gens()
(x + 0*y + 0*z + 0, 0*x + y + 0*z + 0, 0*x + 0*y + z + 0)
```

ngens()

Return the number of linear variables.

OUTPUT: integer

EXAMPLES:

```
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L = LinearExpressionModule(QQ, ('x', 'y', 'z'))
sage: L.ngens()
3
```

```
>>> from sage.all import *
>>> from sage.geometry.linear_expression import LinearExpressionModule
>>> L = LinearExpressionModule(QQ, ('x', 'y', 'z'))
>>> L.ngens()
3
```

random_element()

Return a random element.

EXAMPLES:

```
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L.<x,y,z> = LinearExpressionModule(QQ)
sage: L.random_element() in L
True
```

```
>>> from sage.all import *
>>> from sage.geometry.linear_expression import LinearExpressionModule
```

(continues on next page)

(continued from previous page)

```
>>> L = LinearExpressionModule(QQ, names=('x', 'y', 'z',)); (x, y, z,) = L._
↪first_ngens(3)
>>> L.random_element() in L
True
```

4.4 Newton Polygons

This module implements finite Newton polygons and infinite Newton polygons having a finite number of slopes (and hence a last infinite slope).

class sage.geometry.newton_polygon.**NewtonPolygon_element**(*polyhedron*, *parent*)

Bases: `Element`

Class for infinite Newton polygons with last slope.

last_slope()

Return the last (infinite) slope of this Newton polygon if it is infinite and `+Infinity` otherwise.

EXAMPLES:

```
sage: from sage.geometry.newton_polygon import NewtonPolygon
sage: NP1 = NewtonPolygon([ (0,0), (1,1), (2,8), (3,5) ], last_slope=3)
sage: NP1.last_slope()
3

sage: NP2 = NewtonPolygon([ (0,0), (1,1), (2,5) ])
sage: NP2.last_slope()
+Infinity
```

```
>>> from sage.all import *
>>> from sage.geometry.newton_polygon import NewtonPolygon
>>> NP1 = NewtonPolygon([ (Integer(0),Integer(0)), (Integer(1),Integer(1)),␣
↪(Integer(2),Integer(8)), (Integer(3),Integer(5)) ], last_slope=Integer(3))
>>> NP1.last_slope()
3

>>> NP2 = NewtonPolygon([ (Integer(0),Integer(0)), (Integer(1),Integer(1)),␣
↪(Integer(2),Integer(5)) ])
>>> NP2.last_slope()
+Infinity
```

We check that the last slope of a sum (resp. a product) is the minimum of the last slopes of the summands (resp. the factors):

```
sage: (NP1 + NP2).last_slope()
3
sage: (NP1 * NP2).last_slope()
3
```

```
>>> from sage.all import *
>>> (NP1 + NP2).last_slope()
3
```

(continues on next page)

(continued from previous page)

```
>>> (NP1 * NP2).last_slope()
3
```

plot(***kwargs*)

Plot this Newton polygon.

> **Note**
>
> All usual rendering options (color, thickness, etc.) are available.

EXAMPLES:

```
sage: from sage.geometry.newton_polygon import NewtonPolygon
sage: NP = NewtonPolygon([ (0,0), (1,1), (2,6) ])
sage: polygon = NP.plot()                                                    #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> from sage.geometry.newton_polygon import NewtonPolygon
>>> NP = NewtonPolygon([ (Integer(0),Integer(0)), (Integer(1),Integer(1)),␣
↪(Integer(2),Integer(6)) ])
>>> polygon = NP.plot()                                                     #␣
↪needs sage.plot
```

reverse(*degree=None*)

Return the symmetric of `self`.

INPUT:

- `degree` – integer (default: the top right abscissa of this Newton polygon)

OUTPUT:

The image this Newton polygon under the symmetry '(x,y) mapsto (degree-x, y)`.

EXAMPLES:

```
sage: from sage.geometry.newton_polygon import NewtonPolygon
sage: NP = NewtonPolygon([ (0,0), (1,1), (2,5) ])
sage: NP2 = NP.reverse(); NP2
Finite Newton polygon with 3 vertices: (0, 5), (1, 1), (2, 0)
```

```
>>> from sage.all import *
>>> from sage.geometry.newton_polygon import NewtonPolygon
>>> NP = NewtonPolygon([ (Integer(0),Integer(0)), (Integer(1),Integer(1)),␣
↪(Integer(2),Integer(5)) ])
>>> NP2 = NP.reverse(); NP2
Finite Newton polygon with 3 vertices: (0, 5), (1, 1), (2, 0)
```

We check that the slopes of the symmetric Newton polygon are the opposites of the slopes of the original Newton polygon:

```
sage: NP.slopes()
[1, 4]
sage: NP2.slopes()
[-4, -1]
```

```
>>> from sage.all import *
>>> NP.slopes()
[1, 4]
>>> NP2.slopes()
[-4, -1]
```

slopes(*repetition=True*)

Return the slopes of this Newton polygon.

INPUT:

- `repetition` – boolean (default: `True`)

OUTPUT:

The consecutive slopes (not including the last slope if the polygon is infinity) of this Newton polygon.

If `repetition` is True, each slope is repeated a number of times equal to its length. Otherwise, it appears only one time.

EXAMPLES:

```
sage: from sage.geometry.newton_polygon import NewtonPolygon
sage: NP = NewtonPolygon([ (0,0), (1,1), (3,6) ]); NP
Finite Newton polygon with 3 vertices: (0, 0), (1, 1), (3, 6)

sage: NP.slopes()
[1, 5/2, 5/2]

sage: NP.slopes(repetition=False)
[1, 5/2]
```

```
>>> from sage.all import *
>>> from sage.geometry.newton_polygon import NewtonPolygon
>>> NP = NewtonPolygon([ (Integer(0),Integer(0)), (Integer(1),Integer(1)),␣
↪(Integer(3),Integer(6)) ]); NP
Finite Newton polygon with 3 vertices: (0, 0), (1, 1), (3, 6)

>>> NP.slopes()
[1, 5/2, 5/2]

>>> NP.slopes(repetition=False)
[1, 5/2]
```

vertices(*copy=True*)

Return the list of vertices of this Newton polygon.

INPUT:

- `copy` – boolean (default: `True`)

OUTPUT: the list of vertices of this Newton polygon (or a copy of it if `copy` is set to True)

EXAMPLES:

```
sage: from sage.geometry.newton_polygon import NewtonPolygon
sage: NP = NewtonPolygon([ (0,0), (1,1), (2,5) ]); NP
Finite Newton polygon with 3 vertices: (0, 0), (1, 1), (2, 5)

sage: v = NP.vertices(); v
[(0, 0), (1, 1), (2, 5)]
```

```
>>> from sage.all import *
>>> from sage.geometry.newton_polygon import NewtonPolygon
>>> NP = NewtonPolygon([ (Integer(0),Integer(0)), (Integer(1),Integer(1)),␣
↪(Integer(2),Integer(5)) ]); NP
Finite Newton polygon with 3 vertices: (0, 0), (1, 1), (2, 5)

>>> v = NP.vertices(); v
[(0, 0), (1, 1), (2, 5)]
```

class sage.geometry.newton_polygon.**ParentNewtonPolygon**

Bases: `Parent`, `UniqueRepresentation`

Construct a Newton polygon.

INPUT:

- `arg` – list/tuple/iterable of vertices or of slopes. Currently, slopes must be rational numbers
- `sort_slopes` – boolean (default: `True`); whether slopes must be first sorted
- `last_slope` – rational or infinity (default: `Infinity`); the last slope of the Newton polygon

OUTPUT: the corresponding Newton polygon

Note

By convention, a Newton polygon always contains the point at infinity $(0, \infty)$. These polygons are attached to polynomials or series over discrete valuation rings (e.g. padics).

EXAMPLES:

We specify here a Newton polygon by its vertices:

```
sage: from sage.geometry.newton_polygon import NewtonPolygon
sage: NewtonPolygon([ (0,0), (1,1), (3,5) ])
Finite Newton polygon with 3 vertices: (0, 0), (1, 1), (3, 5)
```

```
>>> from sage.all import *
>>> from sage.geometry.newton_polygon import NewtonPolygon
>>> NewtonPolygon([ (Integer(0),Integer(0)), (Integer(1),Integer(1)), (Integer(3),
↪Integer(5)) ])
Finite Newton polygon with 3 vertices: (0, 0), (1, 1), (3, 5)
```

We note that the convex hull of the vertices is automatically computed:

```
sage: NewtonPolygon([ (0,0), (1,1), (2,8), (3,5) ])
Finite Newton polygon with 3 vertices: (0, 0), (1, 1), (3, 5)
```

```
>>> from sage.all import *
>>> NewtonPolygon([ (Integer(0),Integer(0)), (Integer(1),Integer(1)), (Integer(2),
↪Integer(8)), (Integer(3),Integer(5)) ])
Finite Newton polygon with 3 vertices: (0, 0), (1, 1), (3, 5)
```

Note that the value `+Infinity` is allowed as the second coordinate of a vertex:

```
sage: NewtonPolygon([ (0,0), (1,Infinity), (2,8), (3,5) ])
Finite Newton polygon with 2 vertices: (0, 0), (3, 5)
```

```
>>> from sage.all import *
>>> NewtonPolygon([ (Integer(0),Integer(0)), (Integer(1),Infinity), (Integer(2),
↪Integer(8)), (Integer(3),Integer(5)) ])
Finite Newton polygon with 2 vertices: (0, 0), (3, 5)
```

If last_slope is set, the returned Newton polygon is infinite and ends with an infinite line having the specified slope:

```
sage: NewtonPolygon([ (0,0), (1,1), (2,8), (3,5) ], last_slope=3)
Infinite Newton polygon with 3 vertices: (0, 0), (1, 1), (3, 5) ending by an␣
↪infinite line of slope 3
```

```
>>> from sage.all import *
>>> NewtonPolygon([ (Integer(0),Integer(0)), (Integer(1),Integer(1)), (Integer(2),
↪Integer(8)), (Integer(3),Integer(5)) ], last_slope=Integer(3))
Infinite Newton polygon with 3 vertices: (0, 0), (1, 1), (3, 5) ending by an␣
↪infinite line of slope 3
```

Specifying a last slope may discard some vertices:

```
sage: NewtonPolygon([ (0,0), (1,1), (2,8), (3,5) ], last_slope=3/2)
Infinite Newton polygon with 2 vertices: (0, 0), (1, 1) ending by an infinite␣
↪line of slope 3/2
```

```
>>> from sage.all import *
>>> NewtonPolygon([ (Integer(0),Integer(0)), (Integer(1),Integer(1)), (Integer(2),
↪Integer(8)), (Integer(3),Integer(5)) ], last_slope=Integer(3)/Integer(2))
Infinite Newton polygon with 2 vertices: (0, 0), (1, 1) ending by an infinite␣
↪line of slope 3/2
```

Next, we define a Newton polygon by its slopes:

```
sage: NP = NewtonPolygon([0, 1/2, 1/2, 2/3, 2/3, 2/3, 1, 1])
sage: NP
Finite Newton polygon with 5 vertices: (0, 0), (1, 0), (3, 1), (6, 3), (8, 5)
sage: NP.slopes()
[0, 1/2, 1/2, 2/3, 2/3, 2/3, 1, 1]
```

```
>>> from sage.all import *
>>> NP = NewtonPolygon([Integer(0), Integer(1)/Integer(2), Integer(1)/Integer(2),␣
↪Integer(2)/Integer(3), Integer(2)/Integer(3), Integer(2)/Integer(3), Integer(1),
↪ Integer(1)])
>>> NP
```

(continues on next page)

(continued from previous page)

```
Finite Newton polygon with 5 vertices: (0, 0), (1, 0), (3, 1), (6, 3), (8, 5)
>>> NP.slopes()
[0, 1/2, 1/2, 2/3, 2/3, 2/3, 1, 1]
```

By default, slopes are automatically sorted:

```
sage: NP2 = NewtonPolygon([0, 1, 1/2, 2/3, 1/2, 2/3, 1, 2/3])
sage: NP2
Finite Newton polygon with 5 vertices: (0, 0), (1, 0), (3, 1), (6, 3), (8, 5)
sage: NP == NP2
True
```

```
>>> from sage.all import *
>>> NP2 = NewtonPolygon([Integer(0), Integer(1), Integer(1)/Integer(2),␣
↪Integer(2)/Integer(3), Integer(1)/Integer(2), Integer(2)/Integer(3), Integer(1),
↪ Integer(2)/Integer(3)])
>>> NP2
Finite Newton polygon with 5 vertices: (0, 0), (1, 0), (3, 1), (6, 3), (8, 5)
>>> NP == NP2
True
```

except if the contrary is explicitly mentioned:

```
sage: NewtonPolygon([0, 1, 1/2, 2/3, 1/2, 2/3, 1, 2/3], sort_slopes=False)
Finite Newton polygon with 4 vertices: (0, 0), (1, 0), (6, 10/3), (8, 5)
```

```
>>> from sage.all import *
>>> NewtonPolygon([Integer(0), Integer(1), Integer(1)/Integer(2), Integer(2)/
↪Integer(3), Integer(1)/Integer(2), Integer(2)/Integer(3), Integer(1),␣
↪Integer(2)/Integer(3)], sort_slopes=False)
Finite Newton polygon with 4 vertices: (0, 0), (1, 0), (6, 10/3), (8, 5)
```

Slopes greater that or equal last_slope (if specified) are discarded:

```
sage: NP = NewtonPolygon([0, 1/2, 1/2, 2/3, 2/3, 2/3, 1, 1], last_slope=2/3)
sage: NP
Infinite Newton polygon with 3 vertices: (0, 0), (1, 0), (3, 1) ending by an␣
↪infinite line of slope 2/3
sage: NP.slopes()
[0, 1/2, 1/2]
```

```
>>> from sage.all import *
>>> NP = NewtonPolygon([Integer(0), Integer(1)/Integer(2), Integer(1)/Integer(2),␣
↪Integer(2)/Integer(3), Integer(2)/Integer(3), Integer(2)/Integer(3), Integer(1),
↪ Integer(1)], last_slope=Integer(2)/Integer(3))
>>> NP
Infinite Newton polygon with 3 vertices: (0, 0), (1, 0), (3, 1) ending by an␣
↪infinite line of slope 2/3
>>> NP.slopes()
[0, 1/2, 1/2]
```

Be careful, do not confuse Newton polygons provided by this class with Newton polytopes. Compare:

```
sage: NP = NewtonPolygon([ (0,0), (1,45), (3,6) ]); NP
Finite Newton polygon with 2 vertices: (0, 0), (3, 6)

sage: x, y = polygen(QQ,'x, y')
sage: p = 1 + x*y**45 + x**3*y**6
sage: p.newton_polytope()
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 3 vertices
sage: p.newton_polytope().vertices()
(A vertex at (0, 0), A vertex at (1, 45), A vertex at (3, 6))
```

```
>>> from sage.all import *
>>> NP = NewtonPolygon([ (Integer(0),Integer(0)), (Integer(1),Integer(45)),␣
↪(Integer(3),Integer(6)) ]); NP
Finite Newton polygon with 2 vertices: (0, 0), (3, 6)

>>> x, y = polygen(QQ,'x, y')
>>> p = Integer(1) + x*y**Integer(45) + x**Integer(3)*y**Integer(6)
>>> p.newton_polytope()
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 3 vertices
>>> p.newton_polytope().vertices()
(A vertex at (0, 0), A vertex at (1, 45), A vertex at (3, 6))
```

Element

alias of *NewtonPolygon_element*

4.5 Relative Interiors of Polyhedra and Cones

class sage.geometry.relative_interior.**RelativeInterior**(*polyhedron*)

Bases: *ConvexSet_relatively_open*

The relative interior of a polyhedron or cone.

This class should not be used directly. Use methods `relative_interior()`, `interior()`, *relative_interior()*, *interior()* instead.

EXAMPLES:

```
sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: segment.relative_interior()
Relative interior of
 a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: octant = Cone([(1,0,0), (0,1,0), (0,0,1)])
sage: octant.relative_interior()
Relative interior of 3-d cone in 3-d lattice N
```

```
>>> from sage.all import *
>>> segment = Polyhedron([[Integer(1), Integer(2)], [Integer(3), Integer(4)]])
>>> segment.relative_interior()
Relative interior of
 a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
>>> octant = Cone([(Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0)), (Integer(0),Integer(0),Integer(1))])
```

(continues on next page)

(continued from previous page)

```
>>> octant.relative_interior()
Relative interior of 3-d cone in 3-d lattice N
```

ambient()

Return the ambient convex set or space.

EXAMPLES:

```
sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: ri_segment = segment.relative_interior(); ri_segment
Relative interior of
 a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: ri_segment.ambient()
Vector space of dimension 2 over Rational Field
```

```
>>> from sage.all import *
>>> segment = Polyhedron([[Integer(1), Integer(2)], [Integer(3), Integer(4)]])
>>> ri_segment = segment.relative_interior(); ri_segment
Relative interior of
 a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
>>> ri_segment.ambient()
Vector space of dimension 2 over Rational Field
```

ambient_dim()

Return the dimension of the ambient space.

EXAMPLES:

```
sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: segment.ambient_dim()
2
sage: ri_segment = segment.relative_interior(); ri_segment
Relative interior of
 a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: ri_segment.ambient_dim()
2
```

```
>>> from sage.all import *
>>> segment = Polyhedron([[Integer(1), Integer(2)], [Integer(3), Integer(4)]])
>>> segment.ambient_dim()
2
>>> ri_segment = segment.relative_interior(); ri_segment
Relative interior of
 a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
>>> ri_segment.ambient_dim()
2
```

ambient_vector_space(*base_field=None*)

Return the ambient vector space.

EXAMPLES:

```
sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: ri_segment = segment.relative_interior(); ri_segment
Relative interior of
 a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: ri_segment.ambient_vector_space()
Vector space of dimension 2 over Rational Field
```

```
>>> from sage.all import *
>>> segment = Polyhedron([[Integer(1), Integer(2)], [Integer(3), Integer(4)]])
>>> ri_segment = segment.relative_interior(); ri_segment
Relative interior of
 a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
>>> ri_segment.ambient_vector_space()
Vector space of dimension 2 over Rational Field
```

an_affine_basis()

Return points that form an affine basis for the affine hull.

The points are guaranteed to lie in the topological closure of `self`.

EXAMPLES:

```
sage: segment = Polyhedron([[1, 0], [0, 1]])
sage: segment.relative_interior().an_affine_basis()
[A vertex at (1, 0), A vertex at (0, 1)]
```

```
>>> from sage.all import *
>>> segment = Polyhedron([[Integer(1), Integer(0)], [Integer(0), Integer(1)]])
>>> segment.relative_interior().an_affine_basis()
[A vertex at (1, 0), A vertex at (0, 1)]
```

closure()

Return the topological closure of `self`.

EXAMPLES:

```
sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: ri_segment = segment.relative_interior(); ri_segment
Relative interior of
 a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: ri_segment.closure() is segment
True
```

```
>>> from sage.all import *
>>> segment = Polyhedron([[Integer(1), Integer(2)], [Integer(3), Integer(4)]])
>>> ri_segment = segment.relative_interior(); ri_segment
Relative interior of
 a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
>>> ri_segment.closure() is segment
True
```

dilation(*scalar*)

Return the dilated (uniformly stretched) set.

INPUT:

- `scalar` – a scalar

EXAMPLES:

```
sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: ri_segment = segment.relative_interior(); ri_segment
Relative interior of a
 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: A = ri_segment.dilation(2); A
Relative interior of a
 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: A.closure().vertices()
(A vertex at (2, 4), A vertex at (6, 8))
sage: B = ri_segment.dilation(-1/3); B
Relative interior of a
 1-dimensional polyhedron in QQ^2 defined as the convex hull of 2 vertices
sage: B.closure().vertices()
(A vertex at (-1, -4/3), A vertex at (-1/3, -2/3))
sage: C = ri_segment.dilation(0); C
A 0-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex
sage: C.vertices()
(A vertex at (0, 0),)
```

```
>>> from sage.all import *
>>> segment = Polyhedron([[Integer(1), Integer(2)], [Integer(3), Integer(4)]])
>>> ri_segment = segment.relative_interior(); ri_segment
Relative interior of a
 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
>>> A = ri_segment.dilation(Integer(2)); A
Relative interior of a
 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
>>> A.closure().vertices()
(A vertex at (2, 4), A vertex at (6, 8))
>>> B = ri_segment.dilation(-Integer(1)/Integer(3)); B
Relative interior of a
 1-dimensional polyhedron in QQ^2 defined as the convex hull of 2 vertices
>>> B.closure().vertices()
(A vertex at (-1, -4/3), A vertex at (-1/3, -2/3))
>>> C = ri_segment.dilation(Integer(0)); C
A 0-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex
>>> C.vertices()
(A vertex at (0, 0),)
```

dim()

Return the dimension of `self`.

EXAMPLES:

```
sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: segment.dim()
1
sage: ri_segment = segment.relative_interior(); ri_segment
Relative interior of
 a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
```

(continues on next page)

(continued from previous page)

```
sage: ri_segment.dim()
1
```

```
>>> from sage.all import *
>>> segment = Polyhedron([[Integer(1), Integer(2)], [Integer(3), Integer(4)]])
>>> segment.dim()
1
>>> ri_segment = segment.relative_interior(); ri_segment
Relative interior of
 a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
>>> ri_segment.dim()
1
```

interior()

Return the interior of `self`.

EXAMPLES:

```
sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: ri_segment = segment.relative_interior(); ri_segment
Relative interior of
 a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: ri_segment.interior()
The empty polyhedron in ZZ^2

sage: octant = Cone([(1,0,0), (0,1,0), (0,0,1)])
sage: ri_octant = octant.relative_interior(); ri_octant
Relative interior of 3-d cone in 3-d lattice N
sage: ri_octant.interior() is ri_octant
True
```

```
>>> from sage.all import *
>>> segment = Polyhedron([[Integer(1), Integer(2)], [Integer(3), Integer(4)]])
>>> ri_segment = segment.relative_interior(); ri_segment
Relative interior of
 a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
>>> ri_segment.interior()
The empty polyhedron in ZZ^2

>>> octant = Cone([(Integer(1),Integer(0),Integer(0)), (Integer(0),Integer(1),
↪Integer(0)), (Integer(0),Integer(0),Integer(1))])
>>> ri_octant = octant.relative_interior(); ri_octant
Relative interior of 3-d cone in 3-d lattice N
>>> ri_octant.interior() is ri_octant
True
```

is_closed()

Return whether `self` is closed.

OUTPUT: boolean

EXAMPLES:

```
sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: ri_segment = segment.relative_interior(); ri_segment
Relative interior of a 1-dimensional polyhedron in ZZ^2 defined as the convex␣
↪hull of 2 vertices
sage: ri_segment.is_closed()
False
```

```
>>> from sage.all import *
>>> segment = Polyhedron([[Integer(1), Integer(2)], [Integer(3), Integer(4)]])
>>> ri_segment = segment.relative_interior(); ri_segment
Relative interior of a 1-dimensional polyhedron in ZZ^2 defined as the convex␣
↪hull of 2 vertices
>>> ri_segment.is_closed()
False
```

is_universe()

Return whether `self` is the whole ambient space.

OUTPUT: boolean

EXAMPLES:

```
sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: ri_segment = segment.relative_interior(); ri_segment
Relative interior of
 a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: ri_segment.is_universe()
False
```

```
>>> from sage.all import *
>>> segment = Polyhedron([[Integer(1), Integer(2)], [Integer(3), Integer(4)]])
>>> ri_segment = segment.relative_interior(); ri_segment
Relative interior of
 a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
>>> ri_segment.is_universe()
False
```

linear_transformation(*linear_transf*, ***kwds*)

Return the linear transformation of `self`.

By [Roc1970], Theorem 6.6, the linear transformation of a relative interior is the relative interior of the linear transformation.

INPUT:

- `linear_transf` – a matrix
- `**kwds` – passed to the *linear_transformation()* method of the closure of `self`

EXAMPLES:

```
sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: ri_segment = segment.relative_interior(); ri_segment
Relative interior of a
 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
```

(continues on next page)

(continued from previous page)

```
sage: T = matrix([[1, 1]])
sage: A = ri_segment.linear_transformation(T); A
Relative interior of a
 1-dimensional polyhedron in ZZ^1 defined as the convex hull of 2 vertices
sage: A.closure().vertices()
(A vertex at (3), A vertex at (7))
```

```
>>> from sage.all import *
>>> segment = Polyhedron([[Integer(1), Integer(2)], [Integer(3), Integer(4)]])
>>> ri_segment = segment.relative_interior(); ri_segment
Relative interior of a
 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
>>> T = matrix([[Integer(1), Integer(1)]])
>>> A = ri_segment.linear_transformation(T); A
Relative interior of a
 1-dimensional polyhedron in ZZ^1 defined as the convex hull of 2 vertices
>>> A.closure().vertices()
(A vertex at (3), A vertex at (7))
```

relative_interior()

Return the relative interior of `self`.

As `self` is already relatively open, this method just returns `self`.

EXAMPLES:

```
sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: ri_segment = segment.relative_interior(); ri_segment
Relative interior of
 a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: ri_segment.relative_interior() is ri_segment
True
```

```
>>> from sage.all import *
>>> segment = Polyhedron([[Integer(1), Integer(2)], [Integer(3), Integer(4)]])
>>> ri_segment = segment.relative_interior(); ri_segment
Relative interior of
 a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
>>> ri_segment.relative_interior() is ri_segment
True
```

representative_point()

Return a "generic" point of `self`.

OUTPUT:

A point in `self` (thus, in the relative interior of `self`) as a coordinate vector.

EXAMPLES:

```
sage: C = Cone([[1, 2, 0], [2, 1, 0]])
sage: C.relative_interior().representative_point()
(1, 1, 0)
```

```
>>> from sage.all import *
>>> C = Cone([[Integer(1), Integer(2), Integer(0)], [Integer(2), Integer(1),
↪Integer(0)]])
>>> C.relative_interior().representative_point()
(1, 1, 0)
```

translation(*displacement*)

Return the translation of `self` by a `displacement` vector.

INPUT:

- `displacement` – a displacement vector or a list/tuple of coordinates that determines a displacement vector

EXAMPLES:

```
sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: ri_segment = segment.relative_interior(); ri_segment
Relative interior of a
 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: t = vector([100, 100])
sage: ri_segment.translation(t)
Relative interior of a
 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: ri_segment.closure().vertices()
(A vertex at (1, 2), A vertex at (3, 4))
```

```
>>> from sage.all import *
>>> segment = Polyhedron([[Integer(1), Integer(2)], [Integer(3), Integer(4)]])
>>> ri_segment = segment.relative_interior(); ri_segment
Relative interior of a
 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
>>> t = vector([Integer(100), Integer(100)])
>>> ri_segment.translation(t)
Relative interior of a
 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
>>> ri_segment.closure().vertices()
(A vertex at (1, 2), A vertex at (3, 4))
```

4.6 Ribbon Graphs

This file implements objects called *ribbon graphs*. These are graphs together with a cyclic ordering of the darts adjacent to each vertex. This data allows us to unambiguously "thicken" the ribbon graph to an orientable surface with boundary. Also, every orientable surface with non-empty boundary is the thickening of a ribbon graph.

AUTHORS:

- Pablo Portilla (2016)

class sage.geometry.ribbon_graph.**RibbonGraph**(*sigma*, *rho*)

Bases: `SageObject`, `UniqueRepresentation`

A ribbon graph codified as two elements of a certain permutation group.

A comprehensive introduction on the topic can be found in the beginning of [GGD2011] Chapter 4. More concretely, we will use a variation of what is called in the reference "The permutation representation pair of a dessin".

Note that in that book, ribbon graphs are called "dessins d'enfant". For the sake on completeness we reproduce an adapted version of that introduction here.

Brief introduction

Let Σ be an orientable surface with non-empty boundary and let Γ be the topological realization of a graph that is embedded in Σ in such a way that the graph is a strong deformation retract of the surface.

Let $v(\Gamma)$ be the set of vertices of Γ, suppose that these are white vertices. Now we mark black vertices in an interior point of each edge. In this way we get a bipartite graph where all the black vertices have valency 2 and there is no restriction on the valency of the white vertices. We call the edges of this new graph *darts* (sometimes they are also called *half edges* of the original graph). Observe that each edge of the original graph is formed by two darts.

Given a white vertex $v \in v(\Gamma)$, let $d(v)$ be the set of darts adjacent to v. Let $D(\Gamma)$ be the set of all the darts of Γ and suppose that we enumerate the set $D(\Gamma)$ and that it has n elements.

With the orientation of the surface and the embedding of the graph in the surface we can produce two permutations:

- A permutation that we denote by σ. This permutation is a product of as many cycles as white vertices (that is vertices in Γ). For each vertex consider a small topological circle around it in Σ. This circle intersects each adjacent dart once. The circle has an orientation induced by the orientation on Σ and so defines a cycle that sends the number associated to one dart to the number associated to the next dart in the positive orientation of the circle.
- A permutation that we denote by ρ. This permutation is a product of as many 2-cycles as edges has Γ. It just tells which two darts belong to the same edge.

Abstract definition

Consider a graph Γ (not a priori embedded in any surface). Now we can again consider one vertex in the interior of each edge splitting each edge in two darts. We label the darts with numbers.

We say that a ribbon structure on Γ is a set of two permutations (σ, ρ). Where σ is formed by as many disjoint cycles as vertices had Γ. And each cycle is a cyclic ordering of the darts adjacent to a vertex. The permutation ρ just tell us which two darts belong to the same edge.

For any two such permutations there is a way of "thickening" the graph to a surface with boundary in such a way that the surface retracts (by a strong deformation retract) to the graph and hence the graph is embedded in the surface in a such a way that we could recover σ and ρ.

INPUT:

- `sigma` – a permutation a product of disjoint cycles of any length; singletons (vertices of valency 1) need not be specified
- `rho` – a permutation which is a product of disjoint 2-cycles

Alternatively, one can pass in 2 integers and this will construct a ribbon graph with genus `sigma` and `rho` boundary components. See `make_ribbon()`.

One can also construct the bipartite graph modeling the corresponding Brieskorn-Pham singularity by passing 2 integers and the keyword `bipartite=True`. See `bipartite_ribbon_graph()`.

EXAMPLES:

Consider the ribbon graph consisting of just 1 edge and 2 vertices of valency 1:

```
sage: s0 = PermutationGroupElement('(1)(2)')
sage: r0 = PermutationGroupElement('(1,2)')
sage: R0 = RibbonGraph(s0,r0); R0
Ribbon graph of genus 0 and 1 boundary components
```

```
>>> from sage.all import *
>>> s0 = PermutationGroupElement('(1)(2)')
>>> r0 = PermutationGroupElement('(1,2)')
>>> R0 = RibbonGraph(s0,r0); R0
Ribbon graph of genus 0 and 1 boundary components
```

Consider a graph that has 2 vertices of valency 3 (and hence 3 edges). That is represented by the following two permutations:

```
sage: s1 = PermutationGroupElement('(1,3,5)(2,4,6)')
sage: r1 = PermutationGroupElement('(1,2)(3,4)(5,6)')
sage: R1 = RibbonGraph(s1, r1); R1
Ribbon graph of genus 1 and 1 boundary components
```

```
>>> from sage.all import *
>>> s1 = PermutationGroupElement('(1,3,5)(2,4,6)')
>>> r1 = PermutationGroupElement('(1,2)(3,4)(5,6)')
>>> R1 = RibbonGraph(s1, r1); R1
Ribbon graph of genus 1 and 1 boundary components
```

By drawing the picture in a piece of paper, one can see that its thickening has only 1 boundary component. Since the thickening is homotopically equivalent to the graph and the graph has Euler characteristic -1, we find that the thickening has genus 1:

```
sage: R1.number_boundaries()
1
sage: R1.genus()
1
```

```
>>> from sage.all import *
>>> R1.number_boundaries()
1
>>> R1.genus()
1
```

The following example corresponds to the complete bipartite graph of type $(2, 3)$, where we have added one more edge $(8, 15)$ that ends at a vertex of valency 1. Observe that it is not necessary to specify the vertex (15) of valency 1 when we define sigma:

```
sage: s2 = PermutationGroupElement('(1,3,5,8)(2,4,6)')
sage: r2 = PermutationGroupElement('(1,2)(3,4)(5,6)(8,15)')
sage: R2 = RibbonGraph(s2, r2); R1
Ribbon graph of genus 1 and 1 boundary components
sage: R2.sigma()
(1,3,5,8)(2,4,6)
```

```
>>> from sage.all import *
>>> s2 = PermutationGroupElement('(1,3,5,8)(2,4,6)')
>>> r2 = PermutationGroupElement('(1,2)(3,4)(5,6)(8,15)')
>>> R2 = RibbonGraph(s2, r2); R1
Ribbon graph of genus 1 and 1 boundary components
>>> R2.sigma()
(1,3,5,8)(2,4,6)
```

This example is constructed by taking the bipartite graph of type $(3, 3)$:

```
sage: s3 = PermutationGroupElement('(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,15)(16,
↪17,18)')
sage: r3 = PermutationGroupElement('(1,16)(2,13)(3,10)(4,17)(5,14)(6,11)(7,18)(8,
↪15)(9,12)')
sage: R3 = RibbonGraph(s3, r3); R3
Ribbon graph of genus 1 and 3 boundary components
```

```
>>> from sage.all import *
>>> s3 = PermutationGroupElement('(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,15)(16,17,
↪18)')
>>> r3 = PermutationGroupElement('(1,16)(2,13)(3,10)(4,17)(5,14)(6,11)(7,18)(8,
↪15)(9,12)')
>>> R3 = RibbonGraph(s3, r3); R3
Ribbon graph of genus 1 and 3 boundary components
```

The labeling of the darts can omit some numbers:

```
sage: s4 = PermutationGroupElement('(3,5,10,12)')
sage: r4 = PermutationGroupElement('(3,10)(5,12)')
sage: R4 = RibbonGraph(s4,r4); R4
Ribbon graph of genus 1 and 1 boundary components
```

```
>>> from sage.all import *
>>> s4 = PermutationGroupElement('(3,5,10,12)')
>>> r4 = PermutationGroupElement('(3,10)(5,12)')
>>> R4 = RibbonGraph(s4,r4); R4
Ribbon graph of genus 1 and 1 boundary components
```

The next example is the complete bipartite graph of type $(3, 3)$, where we have added an edge that ends at a vertex of valency 1:

```
sage: s5 = PermutationGroupElement('(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,15)(16,
↪17,18,19)')
sage: r5 = PermutationGroupElement('(1,16)(2,13)(3,10)(4,17)(5,14)(6,11)(7,18)(8,
↪15)(9,12)(19,20)')
sage: R5 = RibbonGraph(s5,r5); R5
Ribbon graph of genus 1 and 3 boundary components
sage: C = R5.contract_edge(9); C
Ribbon graph of genus 1 and 3 boundary components
sage: C.sigma()
(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,15)(16,17,18)
sage: C.rho()
(1,16)(2,13)(3,10)(4,17)(5,14)(6,11)(7,18)(8,15)(9,12)
sage: S = R5.reduced(); S
Ribbon graph of genus 1 and 3 boundary components
sage: S.sigma()
(5,6,8,9,14,15,11,12)
sage: S.rho()
(5,14)(6,11)(8,15)(9,12)
sage: R5.boundary()
[[1, 16, 17, 4, 5, 14, 15, 8, 9, 12, 10, 3],
```

(continues on next page)

(continued from previous page)

```
 [2, 13, 14, 5, 6, 11, 12, 9, 7, 18, 19, 20, 20, 19, 16, 1],
 [3, 10, 11, 6, 4, 17, 18, 7, 8, 15, 13, 2]]
sage: S.boundary()
[[5, 14, 15, 8, 9, 12], [6, 11, 12, 9, 14, 5], [8, 15, 11, 6]]
sage: R5.homology_basis()
[[[5, 14], [13, 2], [1, 16], [17, 4]],
 [[6, 11], [10, 3], [1, 16], [17, 4]],
 [[8, 15], [13, 2], [1, 16], [18, 7]],
 [[9, 12], [10, 3], [1, 16], [18, 7]]]
sage: S.homology_basis()
[[[5, 14]], [[6, 11]], [[8, 15]], [[9, 12]]]
```

```
>>> from sage.all import *
>>> s5 = PermutationGroupElement('(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,15)(16,17,
↪18,19)')
>>> r5 = PermutationGroupElement('(1,16)(2,13)(3,10)(4,17)(5,14)(6,11)(7,18)(8,
↪15)(9,12)(19,20)')
>>> R5 = RibbonGraph(s5,r5); R5
Ribbon graph of genus 1 and 3 boundary components
>>> C = R5.contract_edge(Integer(9)); C
Ribbon graph of genus 1 and 3 boundary components
>>> C.sigma()
(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,15)(16,17,18)
>>> C.rho()
(1,16)(2,13)(3,10)(4,17)(5,14)(6,11)(7,18)(8,15)(9,12)
>>> S = R5.reduced(); S
Ribbon graph of genus 1 and 3 boundary components
>>> S.sigma()
(5,6,8,9,14,15,11,12)
>>> S.rho()
(5,14)(6,11)(8,15)(9,12)
>>> R5.boundary()
[[1, 16, 17, 4, 5, 14, 15, 8, 9, 12, 10, 3],
 [2, 13, 14, 5, 6, 11, 12, 9, 7, 18, 19, 20, 20, 19, 16, 1],
 [3, 10, 11, 6, 4, 17, 18, 7, 8, 15, 13, 2]]
>>> S.boundary()
[[5, 14, 15, 8, 9, 12], [6, 11, 12, 9, 14, 5], [8, 15, 11, 6]]
>>> R5.homology_basis()
[[[5, 14], [13, 2], [1, 16], [17, 4]],
 [[6, 11], [10, 3], [1, 16], [17, 4]],
 [[8, 15], [13, 2], [1, 16], [18, 7]],
 [[9, 12], [10, 3], [1, 16], [18, 7]]]
>>> S.homology_basis()
[[[5, 14]], [[6, 11]], [[8, 15]], [[9, 12]]]
```

We construct a ribbon graph corresponding to a genus 0 surface with 5 boundary components:

```
sage: R = RibbonGraph(0, 5); R
Ribbon graph of genus 0 and 5 boundary components
sage: R.sigma()
(1,9,7,5,3)(2,4,6,8,10)
sage: R.rho()
```

(continues on next page)

(continued from previous page)

```
(1,2)(3,4)(5,6)(7,8)(9,10)
```

```
>>> from sage.all import *
>>> R = RibbonGraph(Integer(0), Integer(5)); R
Ribbon graph of genus 0 and 5 boundary components
>>> R.sigma()
(1,9,7,5,3)(2,4,6,8,10)
>>> R.rho()
(1,2)(3,4)(5,6)(7,8)(9,10)
```

We construct the Brieskorn-Pham singularity of type $(2,3)$:

```
sage: B23 = RibbonGraph(2, 3, bipartite=True); B23
Ribbon graph of genus 1 and 1 boundary components
sage: B23.sigma()
(1,2,3)(4,5,6)(7,8)(9,10)(11,12)
sage: B23.rho()
(1,8)(2,10)(3,12)(4,7)(5,9)(6,11)
```

```
>>> from sage.all import *
>>> B23 = RibbonGraph(Integer(2), Integer(3), bipartite=True); B23
Ribbon graph of genus 1 and 1 boundary components
>>> B23.sigma()
(1,2,3)(4,5,6)(7,8)(9,10)(11,12)
>>> B23.rho()
(1,8)(2,10)(3,12)(4,7)(5,9)(6,11)
```

boundary()

Return the labeled boundaries of `self`.

If you cut the thickening of the graph along the graph. you get a collection of cylinders (recall that the graph was a strong deformation retract of the thickening). In each cylinder one of the boundary components has a labelling of its edges induced by the labelling of the darts.

OUTPUT:

A list of lists. The number of inner lists is the number of boundary components of the surface. Each list in the list consists of an ordered tuple of numbers, each number comes from the number assigned to the corresponding dart before cutting.

EXAMPLES:

We start with a ribbon graph whose thickening has one boundary component. We compute its labeled boundary, then reduce it and compute the labeled boundary of the reduced ribbon graph:

```
sage: s1 = PermutationGroupElement('(1,3,5)(2,4,6)')
sage: r1 = PermutationGroupElement('(1,2)(3,4)(5,6)')
sage: R1 = RibbonGraph(s1,r1); R1
Ribbon graph of genus 1 and 1 boundary components
sage: R1.boundary()
[[1, 2, 4, 3, 5, 6, 2, 1, 3, 4, 6, 5]]
sage: H1 = R1.reduced(); H1
Ribbon graph of genus 1 and 1 boundary components
sage: H1.sigma()
```

(continues on next page)

(continued from previous page)

```
(3,5,4,6)
sage: H1.rho()
(3,4)(5,6)
sage: H1.boundary()
[[3, 4, 6, 5, 4, 3, 5, 6]]
```

```
>>> from sage.all import *
>>> s1 = PermutationGroupElement('(1,3,5)(2,4,6)')
>>> r1 = PermutationGroupElement('(1,2)(3,4)(5,6)')
>>> R1 = RibbonGraph(s1,r1); R1
Ribbon graph of genus 1 and 1 boundary components
>>> R1.boundary()
[[1, 2, 4, 3, 5, 6, 2, 1, 3, 4, 6, 5]]
>>> H1 = R1.reduced(); H1
Ribbon graph of genus 1 and 1 boundary components
>>> H1.sigma()
(3,5,4,6)
>>> H1.rho()
(3,4)(5,6)
>>> H1.boundary()
[[3, 4, 6, 5, 4, 3, 5, 6]]
```

We now consider a ribbon graph whose thickening has 3 boundary components. Also observe that in one of the labeled boundary components, a numbers appears twice in a row. That is because the ribbon graph has a vertex of valency 1:

```
sage: s2=PermutationGroupElement('(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,
↪15)(16,17,18,19)')
sage: r2=PermutationGroupElement('(1,16)(2,13)(3,10)(4,17)(5,14)(6,11)(7,
↪18)(8,15)(9,12)(19,20)')
sage: R2 = RibbonGraph(s2,r2)
sage: R2.number_boundaries()
3
sage: R2.boundary()
[[1, 16, 17, 4, 5, 14, 15, 8, 9, 12, 10, 3],
 [2, 13, 14, 5, 6, 11, 12, 9, 7, 18, 19, 20, 20, 19, 16, 1],
 [3, 10, 11, 6, 4, 17, 18, 7, 8, 15, 13, 2]]
```

```
>>> from sage.all import *
>>> s2=PermutationGroupElement('(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,15)(16,
↪17,18,19)')
>>> r2=PermutationGroupElement('(1,16)(2,13)(3,10)(4,17)(5,14)(6,11)(7,18)(8,
↪15)(9,12)(19,20)')
>>> R2 = RibbonGraph(s2,r2)
>>> R2.number_boundaries()
3
>>> R2.boundary()
[[1, 16, 17, 4, 5, 14, 15, 8, 9, 12, 10, 3],
 [2, 13, 14, 5, 6, 11, 12, 9, 7, 18, 19, 20, 20, 19, 16, 1],
 [3, 10, 11, 6, 4, 17, 18, 7, 8, 15, 13, 2]]
```

contract_edge(*k*)

Return the ribbon graph resulting from the contraction of the `k`-th edge in `self`.

For a ribbon graph (σ, ρ), we contract the edge corresponding to the k-th transposition of ρ.

INPUT:

- `k` – nonnegative integer; the position in ρ of the transposition that is going to be contracted

OUTPUT: a ribbon graph resulting from the contraction of that edge

EXAMPLES:

We start again with the one-holed torus ribbon graph:

```
sage: s1 = PermutationGroupElement('(1,3,5)(2,4,6)')
sage: r1 = PermutationGroupElement('(1,2)(3,4)(5,6)')
sage: R1 = RibbonGraph(s1,r1); R1
Ribbon graph of genus 1 and 1 boundary components
sage: S1 = R1.contract_edge(1); S1
Ribbon graph of genus 1 and 1 boundary components
sage: S1.sigma()
(1,6,2,5)
sage: S1.rho()
(1,2)(5,6)
```

```
>>> from sage.all import *
>>> s1 = PermutationGroupElement('(1,3,5)(2,4,6)')
>>> r1 = PermutationGroupElement('(1,2)(3,4)(5,6)')
>>> R1 = RibbonGraph(s1,r1); R1
Ribbon graph of genus 1 and 1 boundary components
>>> S1 = R1.contract_edge(Integer(1)); S1
Ribbon graph of genus 1 and 1 boundary components
>>> S1.sigma()
(1,6,2,5)
>>> S1.rho()
(1,2)(5,6)
```

However, this ribbon graphs is formed only by loops and hence it cannot be longer reduced, we get an error if we try to contract a loop:

```
sage: S1.contract_edge(1)
Traceback (most recent call last):
...
ValueError: the edge is a loop and cannot be contracted
```

```
>>> from sage.all import *
>>> S1.contract_edge(Integer(1))
Traceback (most recent call last):
...
ValueError: the edge is a loop and cannot be contracted
```

In this example, we consider a graph that has one edge `(19,20)` such that one of its ends is a vertex of valency 1. This is the vertex `(20)` that is not specified when defining σ. We contract precisely this edge and get a ribbon graph with no vertices of valency 1:

```
sage: s2 = PermutationGroupElement('(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,
↪15)(16,17,18,19)')
sage: r2 = PermutationGroupElement('(1,16)(2,13)(3,10)(4,17)(5,14)(6,11)(7,
↪18)(8,15)(9,12)(19,20)')
sage: R2 = RibbonGraph(s2,r2); R2
Ribbon graph of genus 1 and 3 boundary components
sage: R2.sigma()
(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,15)(16,17,18,19)
sage: R2c = R2.contract_edge(9); R2; R2c.sigma(); R2c.rho()
Ribbon graph of genus 1 and 3 boundary components
(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,15)(16,17,18)
(1,16)(2,13)(3,10)(4,17)(5,14)(6,11)(7,18)(8,15)(9,12)
```

```
>>> from sage.all import *
>>> s2 = PermutationGroupElement('(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,
↪15)(16,17,18,19)')
>>> r2 = PermutationGroupElement('(1,16)(2,13)(3,10)(4,17)(5,14)(6,11)(7,
↪18)(8,15)(9,12)(19,20)')
>>> R2 = RibbonGraph(s2,r2); R2
Ribbon graph of genus 1 and 3 boundary components
>>> R2.sigma()
(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,15)(16,17,18,19)
>>> R2c = R2.contract_edge(Integer(9)); R2; R2c.sigma(); R2c.rho()
Ribbon graph of genus 1 and 3 boundary components
(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,15)(16,17,18)
(1,16)(2,13)(3,10)(4,17)(5,14)(6,11)(7,18)(8,15)(9,12)
```

extrude_edge(*vertex*, *dart1*, *dart2*)

Return a ribbon graph resulting from extruding an edge from a vertex, pulling from it, all darts from `dart1` to `dart2` including both.

INPUT:

- `vertex` – the position of the vertex in the permutation σ, which must have valency at least 2
- `dart1` – the position of the first in the cycle corresponding to `vertex`
- `dart2` – the position of the second dart in the cycle corresponding to `vertex`

OUTPUT:

A ribbon graph resulting from extruding a new edge that pulls from `vertex` a new vertex that is, now, adjacent to all the darts from `dart1``to ``dart2` (not including `dart2`) in the cyclic ordering given by the cycle corresponding to `vertex`. Note that `dart1` may be equal to `dart2` allowing thus to extrude a contractible edge from a vertex.

EXAMPLES:

We try several possibilities in the same graph:

```
sage: s1 = PermutationGroupElement('(1,3,5)(2,4,6)')
sage: r1 = PermutationGroupElement('(1,2)(3,4)(5,6)')
sage: R1 = RibbonGraph(s1,r1); R1
Ribbon graph of genus 1 and 1 boundary components
sage: E1 = R1.extrude_edge(1,1,2); E1
Ribbon graph of genus 1 and 1 boundary components
```

(continues on next page)

(continued from previous page)

```
sage: E1.sigma()
(1,3,5)(2,8,6)(4,7)
sage: E1.rho()
(1,2)(3,4)(5,6)(7,8)
sage: E2 = R1.extrude_edge(1,1,3); E2
Ribbon graph of genus 1 and 1 boundary components
sage: E2.sigma()
(1,3,5)(2,8)(4,6,7)
sage: E2.rho()
(1,2)(3,4)(5,6)(7,8)
```

```
>>> from sage.all import *
>>> s1 = PermutationGroupElement('(1,3,5)(2,4,6)')
>>> r1 = PermutationGroupElement('(1,2)(3,4)(5,6)')
>>> R1 = RibbonGraph(s1,r1); R1
Ribbon graph of genus 1 and 1 boundary components
>>> E1 = R1.extrude_edge(Integer(1),Integer(1),Integer(2)); E1
Ribbon graph of genus 1 and 1 boundary components
>>> E1.sigma()
(1,3,5)(2,8,6)(4,7)
>>> E1.rho()
(1,2)(3,4)(5,6)(7,8)
>>> E2 = R1.extrude_edge(Integer(1),Integer(1),Integer(3)); E2
Ribbon graph of genus 1 and 1 boundary components
>>> E2.sigma()
(1,3,5)(2,8)(4,6,7)
>>> E2.rho()
(1,2)(3,4)(5,6)(7,8)
```

We can also extrude a contractible edge from a vertex. This new edge will end at a vertex of valency 1:

```
sage: E1p = R1.extrude_edge(0,0,0); E1p
Ribbon graph of genus 1 and 1 boundary components
sage: E1p.sigma()
(1,3,5,8)(2,4,6)
sage: E1p.rho()
(1,2)(3,4)(5,6)(7,8)
```

```
>>> from sage.all import *
>>> E1p = R1.extrude_edge(Integer(0),Integer(0),Integer(0)); E1p
Ribbon graph of genus 1 and 1 boundary components
>>> E1p.sigma()
(1,3,5,8)(2,4,6)
>>> E1p.rho()
(1,2)(3,4)(5,6)(7,8)
```

In the following example we first extrude one edge from a vertex of valency 3 generating a new vertex of valency 2. Then we extrude a new edge from this vertex of valency 2:

```
sage: s1 = PermutationGroupElement('(1,3,5)(2,4,6)')
sage: r1 = PermutationGroupElement('(1,2)(3,4)(5,6)')
sage: R1 = RibbonGraph(s1,r1); R1
```

(continues on next page)

(continued from previous page)

```
Ribbon graph of genus 1 and 1 boundary components
sage: E1 = R1.extrude_edge(0,0,1); E1
Ribbon graph of genus 1 and 1 boundary components
sage: E1.sigma()
(1,7)(2,4,6)(3,5,8)
sage: E1.rho()
(1,2)(3,4)(5,6)(7,8)
sage: F1 = E1.extrude_edge(0,0,1); F1
Ribbon graph of genus 1 and 1 boundary components
sage: F1.sigma()
(1,9)(2,4,6)(3,5,8)(7,10)
sage: F1.rho()
(1,2)(3,4)(5,6)(7,8)(9,10)
```

```
>>> from sage.all import *
>>> s1 = PermutationGroupElement('(1,3,5)(2,4,6)')
>>> r1 = PermutationGroupElement('(1,2)(3,4)(5,6)')
>>> R1 = RibbonGraph(s1,r1); R1
Ribbon graph of genus 1 and 1 boundary components
>>> E1 = R1.extrude_edge(Integer(0),Integer(0),Integer(1)); E1
Ribbon graph of genus 1 and 1 boundary components
>>> E1.sigma()
(1,7)(2,4,6)(3,5,8)
>>> E1.rho()
(1,2)(3,4)(5,6)(7,8)
>>> F1 = E1.extrude_edge(Integer(0),Integer(0),Integer(1)); F1
Ribbon graph of genus 1 and 1 boundary components
>>> F1.sigma()
(1,9)(2,4,6)(3,5,8)(7,10)
>>> F1.rho()
(1,2)(3,4)(5,6)(7,8)(9,10)
```

genus()

Return the genus of the thickening of `self`.

OUTPUT:

- `g` – nonnegative integer representing the genus of the thickening of the ribbon graph

EXAMPLES:

```
sage: s1 = PermutationGroupElement('(1,3,5)(2,4,6)')
sage: r1 = PermutationGroupElement('(1,2)(3,4)(5,6)')
sage: R1 = RibbonGraph(s1,r1)
sage: R1.genus()
1

sage: s3=PermutationGroupElement('(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,15,
↪16)(17,18,19,20)(21,22,23,24)')
sage: r3=PermutationGroupElement('(1,21)(2,17)(3,13)(4,22)(7,23)(5,18)(6,
↪14)(8,19)(9,15)(10,24)(11,20)(12,16)')
sage: R3 = RibbonGraph(s3,r3); R3.genus()
3
```

```
>>> from sage.all import *
>>> s1 = PermutationGroupElement('(1,3,5)(2,4,6)')
>>> r1 = PermutationGroupElement('(1,2)(3,4)(5,6)')
>>> R1 = RibbonGraph(s1,r1)
>>> R1.genus()
1

>>> s3=PermutationGroupElement('(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,15,
↪16)(17,18,19,20)(21,22,23,24)')
>>> r3=PermutationGroupElement('(1,21)(2,17)(3,13)(4,22)(7,23)(5,18)(6,14)(8,
↪19)(9,15)(10,24)(11,20)(12,16)')
>>> R3 = RibbonGraph(s3,r3); R3.genus()
3
```

homology_basis()

Return an oriented basis of the first homology group of the graph.

OUTPUT:

- A 2-dimensional array of ordered edges in the graph (given by pairs). The length of the first dimension is μ. Each row corresponds to an element of the basis and is a circle contained in the graph.

EXAMPLES:

```
sage: R = RibbonGraph(0,6); R
Ribbon graph of genus 0 and 6 boundary components
sage: R.mu()
5
sage: R.homology_basis()
[[[3, 4], [2, 1]],
 [[5, 6], [2, 1]],
 [[7, 8], [2, 1]],
 [[9, 10], [2, 1]],
 [[11, 12], [2, 1]]]

sage: R = RibbonGraph(1,1); R
Ribbon graph of genus 1 and 1 boundary components
sage: R.mu()
2
sage: R.homology_basis()
[[[2, 5], [4, 1]], [[3, 6], [4, 1]]]
sage: H = R.reduced(); H
Ribbon graph of genus 1 and 1 boundary components
sage: H.sigma()
(2,3,5,6)
sage: H.rho()
(2,5)(3,6)
sage: H.homology_basis()
[[[2, 5]], [[3, 6]]]

sage: s3 = PermutationGroupElement('(1,2,3,4,5,6,7,8,9,10,11,27,25,23)(12,24,
↪26,28,13,14,15,16,17,18,19,20,21,22)')
sage: r3 = PermutationGroupElement('(1,12)(2,13)(3,14)(4,15)(5,16)(6,17)(7,
↪18)(8,19)(9,20)(10,21)(11,22)(23,24)(25,26)(27,28)')
```

(continues on next page)

(continued from previous page)

```
sage: R3 = RibbonGraph(s3,r3); R3
Ribbon graph of genus 5 and 4 boundary components
sage: R3.mu()
13
sage: R3.homology_basis()
[[[2, 13], [12, 1]],
 [[3, 14], [12, 1]],
 [[4, 15], [12, 1]],
 [[5, 16], [12, 1]],
 [[6, 17], [12, 1]],
 [[7, 18], [12, 1]],
 [[8, 19], [12, 1]],
 [[9, 20], [12, 1]],
 [[10, 21], [12, 1]],
 [[11, 22], [12, 1]],
 [[23, 24], [12, 1]],
 [[25, 26], [12, 1]],
 [[27, 28], [12, 1]]]
sage: H3 = R3.reduced(); H3
Ribbon graph of genus 5 and 4 boundary components
sage: H3.sigma()
(2,3,4,5,6,7,8,9,10,11,27,25,23,24,26,28,13,14,15,16,17,18,19,20,21,22)
sage: H3.rho()
(2,13)(3,14)(4,15)(5,16)(6,17)(7,18)(8,19)(9,20)(10,21)(11,22)(23,24)(25,
↪26)(27,28)
sage: H3.homology_basis()
[[[2, 13]],
 [[3, 14]],
 [[4, 15]],
 [[5, 16]],
 [[6, 17]],
 [[7, 18]],
 [[8, 19]],
 [[9, 20]],
 [[10, 21]],
 [[11, 22]],
 [[23, 24]],
 [[25, 26]],
 [[27, 28]]]
```

```
>>> from sage.all import *
>>> R = RibbonGraph(Integer(0),Integer(6)); R
Ribbon graph of genus 0 and 6 boundary components
>>> R.mu()
5
>>> R.homology_basis()
[[[3, 4], [2, 1]],
 [[5, 6], [2, 1]],
 [[7, 8], [2, 1]],
 [[9, 10], [2, 1]],
 [[11, 12], [2, 1]]]
```

(continues on next page)

(continued from previous page)

```
>>> R = RibbonGraph(Integer(1),Integer(1)); R
Ribbon graph of genus 1 and 1 boundary components
>>> R.mu()
2
>>> R.homology_basis()
[[[2, 5], [4, 1]], [[3, 6], [4, 1]]]
>>> H = R.reduced(); H
Ribbon graph of genus 1 and 1 boundary components
>>> H.sigma()
(2,3,5,6)
>>> H.rho()
(2,5)(3,6)
>>> H.homology_basis()
[[[2, 5]], [[3, 6]]]

>>> s3 = PermutationGroupElement('(1,2,3,4,5,6,7,8,9,10,11,27,25,23)(12,24,26,
↪28,13,14,15,16,17,18,19,20,21,22)')
>>> r3 = PermutationGroupElement('(1,12)(2,13)(3,14)(4,15)(5,16)(6,17)(7,
↪18)(8,19)(9,20)(10,21)(11,22)(23,24)(25,26)(27,28)')
>>> R3 = RibbonGraph(s3,r3); R3
Ribbon graph of genus 5 and 4 boundary components
>>> R3.mu()
13
>>> R3.homology_basis()
[[[2, 13], [12, 1]],
 [[3, 14], [12, 1]],
 [[4, 15], [12, 1]],
 [[5, 16], [12, 1]],
 [[6, 17], [12, 1]],
 [[7, 18], [12, 1]],
 [[8, 19], [12, 1]],
 [[9, 20], [12, 1]],
 [[10, 21], [12, 1]],
 [[11, 22], [12, 1]],
 [[23, 24], [12, 1]],
 [[25, 26], [12, 1]],
 [[27, 28], [12, 1]]]
>>> H3 = R3.reduced(); H3
Ribbon graph of genus 5 and 4 boundary components
>>> H3.sigma()
(2,3,4,5,6,7,8,9,10,11,27,25,23,24,26,28,13,14,15,16,17,18,19,20,21,22)
>>> H3.rho()
(2,13)(3,14)(4,15)(5,16)(6,17)(7,18)(8,19)(9,20)(10,21)(11,22)(23,24)(25,
↪26)(27,28)
>>> H3.homology_basis()
[[[2, 13]],
 [[3, 14]],
 [[4, 15]],
 [[5, 16]],
 [[6, 17]],
 [[7, 18]],
 [[8, 19]],
```

(continues on next page)

(continued from previous page)

```
 [[9, 20]],
 [[10, 21]],
 [[11, 22]],
 [[23, 24]],
 [[25, 26]],
 [[27, 28]]]
```

make_generic()

Return a ribbon graph equivalent to `self` but where every vertex has valency 3.

OUTPUT:

- a ribbon graph that is equivalent to `self` but is generic in the sense that all vertices have valency 3

EXAMPLES:

```
sage: R = RibbonGraph(1,3); R
Ribbon graph of genus 1 and 3 boundary components
sage: R.sigma()
(1,2,3,9,7)(4,8,10,5,6)
sage: R.rho()
(1,4)(2,5)(3,6)(7,8)(9,10)
sage: G = R.make_generic(); G
Ribbon graph of genus 1 and 3 boundary components
sage: G.sigma()
(2,3,11)(5,6,13)(7,8,15)(9,16,17)(10,14,19)(12,18,21)(20,22)
sage: G.rho()
(2,5)(3,6)(7,8)(9,10)(11,12)(13,14)(15,16)(17,18)(19,20)(21,22)
sage: R.genus() == G.genus() and R.number_boundaries() == G.number_
↪boundaries()
True

sage: R = RibbonGraph(5,4); R
Ribbon graph of genus 5 and 4 boundary components
sage: R.sigma()
(1,2,3,4,5,6,7,8,9,10,11,27,25,23)(12,24,26,28,13,14,15,16,17,18,19,20,21,22)
sage: R.rho()
(1,12)(2,13)(3,14)(4,15)(5,16)(6,17)(7,18)(8,19)(9,20)(10,21)(11,22)(23,
↪24)(25,26)(27,28)
sage: G = R.reduced(); G
Ribbon graph of genus 5 and 4 boundary components
sage: G.sigma()
(2,3,4,5,6,7,8,9,10,11,27,25,23,24,26,28,13,14,15,16,17,18,19,20,21,22)
sage: G.rho()
(2,13)(3,14)(4,15)(5,16)(6,17)(7,18)(8,19)(9,20)(10,21)(11,22)(23,24)(25,
↪26)(27,28)
sage: G.genus() == R.genus() and G.number_boundaries() == R.number_
↪boundaries()
True

sage: R = RibbonGraph(0,6); R
Ribbon graph of genus 0 and 6 boundary components
sage: R.sigma()
```

(continues on next page)

(continued from previous page)

```
(1,11,9,7,5,3)(2,4,6,8,10,12)
sage: R.rho()
(1,2)(3,4)(5,6)(7,8)(9,10)(11,12)
sage: G = R.reduced(); G
Ribbon graph of genus 0 and 6 boundary components
sage: G.sigma()
(3,4,6,8,10,12,11,9,7,5)
sage: G.rho()
(3,4)(5,6)(7,8)(9,10)(11,12)
sage: G.genus() == R.genus() and G.number_boundaries() == R.number_
↪boundaries()
True
```

```
>>> from sage.all import *
>>> R = RibbonGraph(Integer(1),Integer(3)); R
Ribbon graph of genus 1 and 3 boundary components
>>> R.sigma()
(1,2,3,9,7)(4,8,10,5,6)
>>> R.rho()
(1,4)(2,5)(3,6)(7,8)(9,10)
>>> G = R.make_generic(); G
Ribbon graph of genus 1 and 3 boundary components
>>> G.sigma()
(2,3,11)(5,6,13)(7,8,15)(9,16,17)(10,14,19)(12,18,21)(20,22)
>>> G.rho()
(2,5)(3,6)(7,8)(9,10)(11,12)(13,14)(15,16)(17,18)(19,20)(21,22)
>>> R.genus() == G.genus() and R.number_boundaries() == G.number_boundaries()
True

>>> R = RibbonGraph(Integer(5),Integer(4)); R
Ribbon graph of genus 5 and 4 boundary components
>>> R.sigma()
(1,2,3,4,5,6,7,8,9,10,11,27,25,23)(12,24,26,28,13,14,15,16,17,18,19,20,21,22)
>>> R.rho()
(1,12)(2,13)(3,14)(4,15)(5,16)(6,17)(7,18)(8,19)(9,20)(10,21)(11,22)(23,
↪24)(25,26)(27,28)
>>> G = R.reduced(); G
Ribbon graph of genus 5 and 4 boundary components
>>> G.sigma()
(2,3,4,5,6,7,8,9,10,11,27,25,23,24,26,28,13,14,15,16,17,18,19,20,21,22)
>>> G.rho()
(2,13)(3,14)(4,15)(5,16)(6,17)(7,18)(8,19)(9,20)(10,21)(11,22)(23,24)(25,
↪26)(27,28)
>>> G.genus() == R.genus() and G.number_boundaries() == R.number_boundaries()
True

>>> R = RibbonGraph(Integer(0),Integer(6)); R
Ribbon graph of genus 0 and 6 boundary components
>>> R.sigma()
(1,11,9,7,5,3)(2,4,6,8,10,12)
>>> R.rho()
(1,2)(3,4)(5,6)(7,8)(9,10)(11,12)
```

(continues on next page)

(continued from previous page)

```
>>> G = R.reduced(); G
Ribbon graph of genus 0 and 6 boundary components
>>> G.sigma()
(3,4,6,8,10,12,11,9,7,5)
>>> G.rho()
(3,4)(5,6)(7,8)(9,10)(11,12)
>>> G.genus() == R.genus() and G.number_boundaries() == R.number_boundaries()
True
```

mu()

Return the rank of the first homology group of the thickening of the ribbon graph.

EXAMPLES:

```
sage: s1 = PermutationGroupElement('(1,3,5)(2,4,6)')
sage: r1 = PermutationGroupElement('(1,2)(3,4)(5,6)')
sage: R1 = RibbonGraph(s1,r1);R1
Ribbon graph of genus 1 and 1 boundary components
sage: R1.mu()
2
```

```
>>> from sage.all import *
>>> s1 = PermutationGroupElement('(1,3,5)(2,4,6)')
>>> r1 = PermutationGroupElement('(1,2)(3,4)(5,6)')
>>> R1 = RibbonGraph(s1,r1);R1
Ribbon graph of genus 1 and 1 boundary components
>>> R1.mu()
2
```

normalize()

Return an equivalent graph such that the enumeration of its darts exhausts all numbers from 1 to the number of darts.

OUTPUT:

- a ribbon graph equivalent to `self` such that the enumeration of its darts exhausts all numbers from 1 to the number of darts.

EXAMPLES:

```
sage: s0 = PermutationGroupElement('(1,22,3,4,5,6,7,15)(8,16,9,10,11,12,13,14)
↪')
sage: r0 = PermutationGroupElement('(1,8)(22,9)(3,10)(4,11)(5,12)(6,13)(7,
↪14)(15,16)')
sage: R0 = RibbonGraph(s0,r0); R0
Ribbon graph of genus 3 and 2 boundary components
sage: RN0 = R0.normalize(); RN0; RN0.sigma(); RN0.rho()
Ribbon graph of genus 3 and 2 boundary components
(1,16,2,3,4,5,6,14)(7,15,8,9,10,11,12,13)
(1,7)(2,9)(3,10)(4,11)(5,12)(6,13)(8,16)(14,15)

sage: s1 = PermutationGroupElement('(5,10,12)(30,34,78)')
sage: r1 = PermutationGroupElement('(5,30)(10,34)(12,78)')
sage: R1 = RibbonGraph(s1,r1); R1
```

(continues on next page)

(continued from previous page)

```
Ribbon graph of genus 1 and 1 boundary components
sage: RN1 = R1.normalize(); RN1; RN1.sigma(); RN1.rho()
Ribbon graph of genus 1 and 1 boundary components
(1,2,3)(4,5,6)
(1,4)(2,5)(3,6)
```

```
>>> from sage.all import *
>>> s0 = PermutationGroupElement('(1,22,3,4,5,6,7,15)(8,16,9,10,11,12,13,14)')
>>> r0 = PermutationGroupElement('(1,8)(22,9)(3,10)(4,11)(5,12)(6,13)(7,
→14)(15,16)')
>>> R0 = RibbonGraph(s0,r0); R0
Ribbon graph of genus 3 and 2 boundary components
>>> RN0 = R0.normalize(); RN0; RN0.sigma(); RN0.rho()
Ribbon graph of genus 3 and 2 boundary components
(1,16,2,3,4,5,6,14)(7,15,8,9,10,11,12,13)
(1,7)(2,9)(3,10)(4,11)(5,12)(6,13)(8,16)(14,15)

>>> s1 = PermutationGroupElement('(5,10,12)(30,34,78)')
>>> r1 = PermutationGroupElement('(5,30)(10,34)(12,78)')
>>> R1 = RibbonGraph(s1,r1); R1
Ribbon graph of genus 1 and 1 boundary components
>>> RN1 = R1.normalize(); RN1; RN1.sigma(); RN1.rho()
Ribbon graph of genus 1 and 1 boundary components
(1,2,3)(4,5,6)
(1,4)(2,5)(3,6)
```

number_boundaries()

Return number of boundary components of the thickening of the ribbon graph.

EXAMPLES:

The first example is the ribbon graph corresponding to the torus with one hole:

```
sage: s1 = PermutationGroupElement('(1,3,5)(2,4,6)')
sage: r1 = PermutationGroupElement('(1,2)(3,4)(5,6)')
sage: R1 = RibbonGraph(s1,r1)
sage: R1.number_boundaries()
1
```

```
>>> from sage.all import *
>>> s1 = PermutationGroupElement('(1,3,5)(2,4,6)')
>>> r1 = PermutationGroupElement('(1,2)(3,4)(5,6)')
>>> R1 = RibbonGraph(s1,r1)
>>> R1.number_boundaries()
1
```

This example is constructed by taking the bipartite graph of type $(3,3)$:

```
sage: s2 = PermutationGroupElement('(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,
→15)(16,17,18)')
sage: r2 = PermutationGroupElement('(1,16)(2,13)(3,10)(4,17)(5,14)(6,11)(7,
→18)(8,15)(9,12)')
sage: R2 = RibbonGraph(s2,r2)
```

(continues on next page)

(continued from previous page)

```
sage: R2.number_boundaries()
3
```

```
>>> from sage.all import *
>>> s2 = PermutationGroupElement('(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,
↪15)(16,17,18)')
>>> r2 = PermutationGroupElement('(1,16)(2,13)(3,10)(4,17)(5,14)(6,11)(7,
↪18)(8,15)(9,12)')
>>> R2 = RibbonGraph(s2,r2)
>>> R2.number_boundaries()
3
```

reduced()

Return a ribbon graph with 1 vertex and μ edges (where μ is the first betti number of the graph).

OUTPUT:

- a ribbon graph whose σ permutation has only 1 non-singleton cycle and whose ρ permutation is a product of μ disjoint 2-cycles

EXAMPLES:

```
sage: s1 = PermutationGroupElement('(1,3,5)(2,4,6)')
sage: r1 = PermutationGroupElement('(1,2)(3,4)(5,6)')
sage: R1 = RibbonGraph(s1,r1); R1
Ribbon graph of genus 1 and 1 boundary components
sage: G1 = R1.reduced(); G1
Ribbon graph of genus 1 and 1 boundary components
sage: G1.sigma()
(3,5,4,6)
sage: G1.rho()
(3,4)(5,6)

sage: s2 = PermutationGroupElement('(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,
↪15)(16,17,18,19)')
sage: r2 = PermutationGroupElement('(1,16)(2,13)(3,10)(4,17)(5,14)(6,11)(7,
↪18)(8,15)(9,12)(19,20)')
sage: R2 = RibbonGraph(s2,r2); R2
Ribbon graph of genus 1 and 3 boundary components
sage: G2 = R2.reduced(); G2
Ribbon graph of genus 1 and 3 boundary components
sage: G2.sigma()
(5,6,8,9,14,15,11,12)
sage: G2.rho()
(5,14)(6,11)(8,15)(9,12)

sage: s3 = PermutationGroupElement('(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,15,
↪16)(17,18,19,20)(21,22,23,24)')
sage: r3 = PermutationGroupElement('(1,21)(2,17)(3,13)(4,22)(7,23)(5,18)(6,
↪14)(8,19)(9,15)(10,24)(11,20)(12,16)')
sage: R3 = RibbonGraph(s3,r3); R3
Ribbon graph of genus 3 and 1 boundary components
sage: G3 = R3.reduced(); G3
```

(continues on next page)

(continued from previous page)

```
Ribbon graph of genus 3 and 1 boundary components
sage: G3.sigma()
(5,6,8,9,11,12,18,19,20,14,15,16)
sage: G3.rho()
(5,18)(6,14)(8,19)(9,15)(11,20)(12,16)
```

```
>>> from sage.all import *
>>> s1 = PermutationGroupElement('(1,3,5)(2,4,6)')
>>> r1 = PermutationGroupElement('(1,2)(3,4)(5,6)')
>>> R1 = RibbonGraph(s1,r1); R1
Ribbon graph of genus 1 and 1 boundary components
>>> G1 = R1.reduced(); G1
Ribbon graph of genus 1 and 1 boundary components
>>> G1.sigma()
(3,5,4,6)
>>> G1.rho()
(3,4)(5,6)

>>> s2 = PermutationGroupElement('(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,
↪15)(16,17,18,19)')
>>> r2 = PermutationGroupElement('(1,16)(2,13)(3,10)(4,17)(5,14)(6,11)(7,
↪18)(8,15)(9,12)(19,20)')
>>> R2 = RibbonGraph(s2,r2); R2
Ribbon graph of genus 1 and 3 boundary components
>>> G2 = R2.reduced(); G2
Ribbon graph of genus 1 and 3 boundary components
>>> G2.sigma()
(5,6,8,9,14,15,11,12)
>>> G2.rho()
(5,14)(6,11)(8,15)(9,12)

>>> s3 = PermutationGroupElement('(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,15,
↪16)(17,18,19,20)(21,22,23,24)')
>>> r3 = PermutationGroupElement('(1,21)(2,17)(3,13)(4,22)(7,23)(5,18)(6,
↪14)(8,19)(9,15)(10,24)(11,20)(12,16)')
>>> R3 = RibbonGraph(s3,r3); R3
Ribbon graph of genus 3 and 1 boundary components
>>> G3 = R3.reduced(); G3
Ribbon graph of genus 3 and 1 boundary components
>>> G3.sigma()
(5,6,8,9,11,12,18,19,20,14,15,16)
>>> G3.rho()
(5,18)(6,14)(8,19)(9,15)(11,20)(12,16)
```

rho()

Return the permutation ρ of `self`.

EXAMPLES:

```
sage: s1 = PermutationGroupElement('(1,3,5,8)(2,4,6)')
sage: r1 = PermutationGroupElement('(1,2)(3,4)(5,6)(8,15)')
sage: R = RibbonGraph(s1, r1)
```

(continues on next page)

(continued from previous page)

```
sage: R.rho()
(1,2)(3,4)(5,6)(8,15)
```

```
>>> from sage.all import *
>>> s1 = PermutationGroupElement('(1,3,5,8)(2,4,6)')
>>> r1 = PermutationGroupElement('(1,2)(3,4)(5,6)(8,15)')
>>> R = RibbonGraph(s1, r1)
>>> R.rho()
(1,2)(3,4)(5,6)(8,15)
```

sigma()

Return the permutation σ of `self`.

EXAMPLES:

```
sage: s1 = PermutationGroupElement('(1,3,5,8)(2,4,6)')
sage: r1 = PermutationGroupElement('(1,2)(3,4)(5,6)(8,15)')
sage: R = RibbonGraph(s1, r1)
sage: R.sigma()
(1,3,5,8)(2,4,6)
```

```
>>> from sage.all import *
>>> s1 = PermutationGroupElement('(1,3,5,8)(2,4,6)')
>>> r1 = PermutationGroupElement('(1,2)(3,4)(5,6)(8,15)')
>>> R = RibbonGraph(s1, r1)
>>> R.sigma()
(1,3,5,8)(2,4,6)
```

sage.geometry.ribbon_graph.**bipartite_ribbon_graph**(p, q)

Return the bipartite graph modeling the corresponding Brieskorn-Pham singularity.

Take two parallel lines in the plane, and consider p points in one of them and q points in the other. Join with a line each point from the first set with every point with the second set. The resulting is a planar projection of the complete bipartite graph of type (p, q). If you consider the cyclic ordering at each vertex induced by the positive orientation of the plane, the result is a ribbon graph whose associated orientable surface with boundary is homeomorphic to the Milnor fiber of the Brieskorn-Pham singularity $x^p + y^q$. It satisfies that it has $\gcd(p, q)$ number of boundary components and genus $(pq - p - q - \gcd(p, q) - 2)/2$.

INPUT:

- `p` – positive integer
- `q` – positive integer

EXAMPLES:

```
sage: B23 = RibbonGraph(2,3,bipartite=True); B23; B23.sigma(); B23.rho()
Ribbon graph of genus 1 and 1 boundary components
(1,2,3)(4,5,6)(7,8)(9,10)(11,12)
(1,8)(2,10)(3,12)(4,7)(5,9)(6,11)

sage: B32 = RibbonGraph(3,2,bipartite=True); B32; B32.sigma(); B32.rho()
Ribbon graph of genus 1 and 1 boundary components
(1,2)(3,4)(5,6)(7,8,9)(10,11,12)
```

(continues on next page)

(continued from previous page)

```
(1,9)(2,12)(3,8)(4,11)(5,7)(6,10)

sage: B33 = RibbonGraph(3,3,bipartite=True); B33; B33.sigma(); B33.rho()
Ribbon graph of genus 1 and 3 boundary components
(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,15)(16,17,18)
(1,12)(2,15)(3,18)(4,11)(5,14)(6,17)(7,10)(8,13)(9,16)

sage: B24 = RibbonGraph(2,4,bipartite=True); B24; B24.sigma(); B24.rho()
Ribbon graph of genus 1 and 2 boundary components
(1,2,3,4)(5,6,7,8)(9,10)(11,12)(13,14)(15,16)
(1,10)(2,12)(3,14)(4,16)(5,9)(6,11)(7,13)(8,15)

sage: B47 = RibbonGraph(4,7, bipartite=True); B47; B47.sigma(); B47.rho()
Ribbon graph of genus 9 and 1 boundary components
(1,2,3,4,5,6,7)(8,9,10,11,12,13,14)(15,16,17,18,19,20,21)(22,23,24,25,26,27,
→28)(29,30,31,32)(33,34,35,36)(37,38,39,40)(41,42,43,44)(45,46,47,48)(49,50,51,
→52)(53,54,55,56)
(1,32)(2,36)(3,40)(4,44)(5,48)(6,52)(7,56)(8,31)(9,35)(10,39)(11,43)(12,47)(13,
→51)(14,55)(15,30)(16,34)(17,38)(18,42)(19,46)(20,50)(21,54)(22,29)(23,33)(24,
→37)(25,41)(26,45)(27,49)(28,53)
```

```
>>> from sage.all import *
>>> B23 = RibbonGraph(Integer(2),Integer(3),bipartite=True); B23; B23.sigma();
→B23.rho()
Ribbon graph of genus 1 and 1 boundary components
(1,2,3)(4,5,6)(7,8)(9,10)(11,12)
(1,8)(2,10)(3,12)(4,7)(5,9)(6,11)

>>> B32 = RibbonGraph(Integer(3),Integer(2),bipartite=True); B32; B32.sigma();
→B32.rho()
Ribbon graph of genus 1 and 1 boundary components
(1,2)(3,4)(5,6)(7,8,9)(10,11,12)
(1,9)(2,12)(3,8)(4,11)(5,7)(6,10)

>>> B33 = RibbonGraph(Integer(3),Integer(3),bipartite=True); B33; B33.sigma();
→B33.rho()
Ribbon graph of genus 1 and 3 boundary components
(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,15)(16,17,18)
(1,12)(2,15)(3,18)(4,11)(5,14)(6,17)(7,10)(8,13)(9,16)

>>> B24 = RibbonGraph(Integer(2),Integer(4),bipartite=True); B24; B24.sigma();
→B24.rho()
Ribbon graph of genus 1 and 2 boundary components
(1,2,3,4)(5,6,7,8)(9,10)(11,12)(13,14)(15,16)
(1,10)(2,12)(3,14)(4,16)(5,9)(6,11)(7,13)(8,15)

>>> B47 = RibbonGraph(Integer(4),Integer(7), bipartite=True); B47; B47.sigma();
→B47.rho()
Ribbon graph of genus 9 and 1 boundary components
(1,2,3,4,5,6,7)(8,9,10,11,12,13,14)(15,16,17,18,19,20,21)(22,23,24,25,26,27,
→28)(29,30,31,32)(33,34,35,36)(37,38,39,40)(41,42,43,44)(45,46,47,48)(49,50,51,
→52)(53,54,55,56)
```

(continues on next page)

(continued from previous page)

```
(1,32)(2,36)(3,40)(4,44)(5,48)(6,52)(7,56)(8,31)(9,35)(10,39)(11,43)(12,47)(13,
↪51)(14,55)(15,30)(16,34)(17,38)(18,42)(19,46)(20,50)(21,54)(22,29)(23,33)(24,
↪37)(25,41)(26,45)(27,49)(28,53)
```

sage.geometry.ribbon_graph.**make_ribbon**(g, r)

Return a ribbon graph whose thickening has genus g and r boundary components.

INPUT:

- g – nonnegative integer representing the genus of the thickening
- r – positive integer representing the number of boundary components of the thickening

OUTPUT:

- a ribbon graph that has 2 vertices (two non-trivial cycles in its sigma permutation) of valency $2g + r$ and it has $2g + r$ edges (and hence $4g + 2r$ darts)

EXAMPLES:

```
sage: from sage.geometry.ribbon_graph import make_ribbon
sage: R = make_ribbon(0,1); R
Ribbon graph of genus 0 and 1 boundary components
sage: R.sigma()
()
sage: R.rho()
(1,2)

sage: R = make_ribbon(0,5); R
Ribbon graph of genus 0 and 5 boundary components
sage: R.sigma()
(1,9,7,5,3)(2,4,6,8,10)
sage: R.rho()
(1,2)(3,4)(5,6)(7,8)(9,10)

sage: R = make_ribbon(1,1); R
Ribbon graph of genus 1 and 1 boundary components
sage: R.sigma()
(1,2,3)(4,5,6)
sage: R.rho()
(1,4)(2,5)(3,6)

sage: R = make_ribbon(7,3); R
Ribbon graph of genus 7 and 3 boundary components
sage: R.sigma()
(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,33,31)(16,32,34,17,18,19,20,21,22,23,24,25,
↪26,27,28,29,30)
sage: R.rho()
(1,16)(2,17)(3,18)(4,19)(5,20)(6,21)(7,22)(8,23)(9,24)(10,25)(11,26)(12,27)(13,
↪28)(14,29)(15,30)(31,32)(33,34)
```

```
>>> from sage.all import *
>>> from sage.geometry.ribbon_graph import make_ribbon
>>> R = make_ribbon(Integer(0),Integer(1)); R
Ribbon graph of genus 0 and 1 boundary components
```

(continues on next page)

(continued from previous page)

```
>>> R.sigma()
()
>>> R.rho()
(1,2)

>>> R = make_ribbon(Integer(0),Integer(5)); R
Ribbon graph of genus 0 and 5 boundary components
>>> R.sigma()
(1,9,7,5,3)(2,4,6,8,10)
>>> R.rho()
(1,2)(3,4)(5,6)(7,8)(9,10)

>>> R = make_ribbon(Integer(1),Integer(1)); R
Ribbon graph of genus 1 and 1 boundary components
>>> R.sigma()
(1,2,3)(4,5,6)
>>> R.rho()
(1,4)(2,5)(3,6)

>>> R = make_ribbon(Integer(7),Integer(3)); R
Ribbon graph of genus 7 and 3 boundary components
>>> R.sigma()
(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,33,31)(16,32,34,17,18,19,20,21,22,23,24,25,
→26,27,28,29,30)
>>> R.rho()
(1,16)(2,17)(3,18)(4,19)(5,20)(6,21)(7,22)(8,23)(9,24)(10,25)(11,26)(12,27)(13,
→28)(14,29)(15,30)(31,32)(33,34)
```

4.7 Pseudolines

This module gathers everything that has to do with pseudolines, and for a start a `PseudolineArrangement` class that can be used to describe an arrangement of pseudolines in several different ways, and to translate one description into another, as well as to display *Wiring diagrams* via the `show` method.

In the following, we try to stick to the terminology given in [Fe1997], which can be checked in case of doubt. And please fix this module's documentation afterwards :-)

Definition

A *pseudoline* can not be defined by itself, though it can be thought of as a x-monotone curve in the plane. A *set* of pseudolines, however, represents a set of such curves that pairwise intersect exactly once (and hence mimic the behaviour of straight lines in general position). We also assume that those pseudolines are in general position, that is that no three of them cross at the same point.

The present class is made to deal with a combinatorial encoding of a pseudolines arrangement, that is the ordering in which a pseudoline l_i of an arrangement $l_0, ..., l_{n-1}$ crosses the $n-1$ other lines.

Warning

It is assumed through all the methods that the given lines are numbered according to their y-coordinate on the vertical line $x = -\infty$. For instance, it is not possible that the first transposition be `(0,2)` (or equivalently that the first line l_0 crosses is l_2 and conversely), because one of them would have to cross l_1 first.

4.7.1 Encodings

Permutations

An arrangement of pseudolines can be described by a sequence of n lists of length $n-1$, where the i list is a permutation of $\{0, ..., n-1\} \backslash i$ representing the ordering in which the i th pseudoline meets the other ones.

```
sage: from sage.geometry.pseudolines import PseudolineArrangement
sage: permutations = [[3, 2, 1], [3, 2, 0], [3, 1, 0], [2, 1, 0]]
sage: p = PseudolineArrangement(permutations)
sage: p
Arrangement of pseudolines of size 4
sage: p.show()                                                              #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> from sage.geometry.pseudolines import PseudolineArrangement
>>> permutations = [[Integer(3), Integer(2), Integer(1)], [Integer(3), Integer(2),␣
↪Integer(0)], [Integer(3), Integer(1), Integer(0)], [Integer(2), Integer(1),␣
↪Integer(0)]]
>>> p = PseudolineArrangement(permutations)
>>> p
Arrangement of pseudolines of size 4
>>> p.show()                                                              #␣
↪needs sage.plot
```

Sequence of transpositions

An arrangement of pseudolines can also be described as a sequence of $\binom{n}{2}$ transpositions (permutations of two elements). In this sequence, the transposition $(2, 3)$ appears before $(8, 2)$ iif l_2 crosses l_3 before it crosses l_8. This encoding is easy to obtain by reading the wiring diagram from left to right (see the *show* method).

```
sage: from sage.geometry.pseudolines import PseudolineArrangement
sage: transpositions = [(3, 2), (3, 1), (0, 3), (2, 1), (0, 2), (0, 1)]
sage: p = PseudolineArrangement(transpositions)
sage: p
Arrangement of pseudolines of size 4
sage: p.show()                                                              #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> from sage.geometry.pseudolines import PseudolineArrangement
>>> transpositions = [(Integer(3), Integer(2)), (Integer(3), Integer(1)), (Integer(0),
↪ Integer(3)), (Integer(2), Integer(1)), (Integer(0), Integer(2)), (Integer(0),␣
↪Integer(1))]
>>> p = PseudolineArrangement(transpositions)
>>> p
Arrangement of pseudolines of size 4
>>> p.show()                                                              #␣
↪needs sage.plot
```

Note that this ordering is not necessarily unique.

Felsner's Matrix

Felser gave an encoding of an arrangement of pseudolines that takes n^2 bits instead of the $n^2 \log(n)$ bits required by the two previous encodings.

Instead of storing the permutation `[3, 2, 1]` to remember that line l_0 crosses l_3 then l_2 then l_1, it is sufficient to remember the positions for which each line l_i meets a line l_j with $j < i$. As l_0 – the first of the lines – can only meet pseudolines with higher index, we can store `[0, 0, 0]` instead of `[3, 2, 1]` stored previously. For l_1's permutation `[3, 2, 0]` we only need to remember that l_1 first crosses 2 pseudolines of higher index, and then a pseudoline with smaller index, which yields the bit vector `[0, 0, 1]`. Hence we can transform the list of permutations above into a list of n bit vectors of length $n-1$, that is

$$\begin{array}{ccc} 3 & 2 & 1 \\ 3 & 2 & 0 \\ 3 & 1 & 0 \\ 2 & 1 & 0 \end{array} \Rightarrow \begin{array}{ccc} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{array}$$

In order to go back from Felsner's matrix to an encoding by a sequence of transpositions, it is sufficient to look for occurrences of $\begin{array}{c} 0 \\ 1 \end{array}$ in the first column of the matrix, as it corresponds in the wiring diagram to a line going up while the line immediately above it goes down – those two lines cross. Each time such a pattern is found it yields a new transposition, and the matrix can be updated so that this pattern disappears. A more detailed description of this algorithm is given in [Fe1997].

```
sage: from sage.geometry.pseudolines import PseudolineArrangement
sage: felsner_matrix = [[0, 0, 0], [0, 0, 1], [0, 1, 1], [1, 1, 1]]
sage: p = PseudolineArrangement(felsner_matrix)
sage: p
Arrangement of pseudolines of size 4
```

```
>>> from sage.all import *
>>> from sage.geometry.pseudolines import PseudolineArrangement
>>> felsner_matrix = [[Integer(0), Integer(0), Integer(0)], [Integer(0), Integer(0),␣
↪Integer(1)], [Integer(0), Integer(1), Integer(1)], [Integer(1), Integer(1),␣
↪Integer(1)]]
>>> p = PseudolineArrangement(felsner_matrix)
>>> p
Arrangement of pseudolines of size 4
```

4.7.2 Example

Let us define in the plane several lines l_i of equation $y = ax + b$ by picking a coefficient a and b for each of them. We make sure that no two of them are parallel by making sure all of the a chosen are different, and we avoid a common crossing of three lines by adding a random noise to b:

```
sage: n = 20
sage: l = sorted(zip(Subsets(20*n, n).random_element(),
....:                [randint(0, 20*n) + random() for i in range(n)]))
sage: print(l[:5])                          # not tested                      #␣
↪needs sage.combinat
[(96, 278.0130613051349), (74, 332.92512282478714), (13, 155.65820951249867),
 (209, 34.753946221755307), (147, 193.51376457741441)]
```

```
>>> from sage.all import *
>>> n = Integer(20)
>>> l = sorted(zip(Subsets(Integer(20)*n, n).random_element(),
...                [randint(Integer(0), Integer(20)*n) + random() for i in range(n)]))
```

(continues on next page)

(continued from previous page)

```
>>> print(l[:Integer(5)])                           # not tested                    ␣
↪     # needs sage.combinat
[(96, 278.0130613051349), (74, 332.92512282478714), (13, 155.65820951249867),
 (209, 34.753946221755307), (147, 193.51376457741441)]
```

We can now compute for each i the order in which line i meets the other lines:

```
sage: permutations = [[0..i-1] + [i+1..n-1] for i in range(n)]
sage: def a(x): return l[x][0]
sage: def b(x): return l[x][1]
sage: for i, perm in enumerate(permutations):
....:     perm.sort(key=lambda j: (b(j)-b(i))/(a(i)-a(j)))
```

```
>>> from sage.all import *
>>> permutations = [(ellipsis_range(Integer(0),Ellipsis,i-Integer(1))) + (ellipsis_
↪range(i+Integer(1),Ellipsis,n-Integer(1))) for i in range(n)]
>>> def a(x): return l[x][Integer(0)]
>>> def b(x): return l[x][Integer(1)]
>>> for i, perm in enumerate(permutations):
...     perm.sort(key=lambda j: (b(j)-b(i))/(a(i)-a(j)))
```

And finally build the line arrangement:

```
sage: from sage.geometry.pseudolines import PseudolineArrangement
sage: p = PseudolineArrangement(permutations)
sage: print(p)
Arrangement of pseudolines of size 20
sage: p.show(figsize=[20,8])                                                        #␣
↪needs sage.combinat sage.plot
```

```
>>> from sage.all import *
>>> from sage.geometry.pseudolines import PseudolineArrangement
>>> p = PseudolineArrangement(permutations)
>>> print(p)
Arrangement of pseudolines of size 20
>>> p.show(figsize=[Integer(20),Integer(8)])                                        ␣
↪             # needs sage.combinat sage.plot
```

Author

Nathann Cohen

4.7.3 Methods

class sage.geometry.pseudolines.**PseudolineArrangement**(*seq*, *encoding='auto'*)

Bases: `object`

Create an arrangement of pseudolines.

INPUT:

- `seq` – a sequence describing the line arrangement. It can be:
  - A list of n permutations of size $n-1$.

  - A list of $\binom{n}{2}$ transpositions
  - A Felsner matrix, given as a sequence of n binary vectors of length $n-1$.

- `encoding` – information on how the data should be interpreted, and can assume any value among 'transpositions', 'permutations', 'Felsner' or 'auto'. In the latter case, the type will be guessed (default behaviour).

Note

- The pseudolines are assumed to be integers $0, \ldots, n-1$.
- For more information on the different encodings, see the *pseudolines module*'s documentation.

felsner_matrix()

Return a Felsner matrix describing the arrangement.

See the *pseudolines module*'s documentation for more information on this encoding.

EXAMPLES:

```
sage: from sage.geometry.pseudolines import PseudolineArrangement
sage: permutations = [[3, 2, 1], [3, 2, 0], [3, 1, 0], [2, 1, 0]]
sage: p = PseudolineArrangement(permutations)
sage: p.felsner_matrix()
[[0, 0, 0], [0, 0, 1], [0, 1, 1], [1, 1, 1]]
```

```
>>> from sage.all import *
>>> from sage.geometry.pseudolines import PseudolineArrangement
>>> permutations = [[Integer(3), Integer(2), Integer(1)], [Integer(3),␣
↪Integer(2), Integer(0)], [Integer(3), Integer(1), Integer(0)], [Integer(2),␣
↪Integer(1), Integer(0)]]
>>> p = PseudolineArrangement(permutations)
>>> p.felsner_matrix()
[[0, 0, 0], [0, 0, 1], [0, 1, 1], [1, 1, 1]]
```

permutations()

Return the arrangements as n permutations of size $n-1$.

See the *pseudolines module*'s documentation for more information on this encoding.

EXAMPLES:

```
sage: from sage.geometry.pseudolines import PseudolineArrangement
sage: permutations = [[3, 2, 1], [3, 2, 0], [3, 1, 0], [2, 1, 0]]
sage: p = PseudolineArrangement(permutations)
sage: p.permutations()
[[3, 2, 1], [3, 2, 0], [3, 1, 0], [2, 1, 0]]
```

```
>>> from sage.all import *
>>> from sage.geometry.pseudolines import PseudolineArrangement
>>> permutations = [[Integer(3), Integer(2), Integer(1)], [Integer(3),␣
↪Integer(2), Integer(0)], [Integer(3), Integer(1), Integer(0)], [Integer(2),␣
↪Integer(1), Integer(0)]]
>>> p = PseudolineArrangement(permutations)
```

(continues on next page)

(continued from previous page)

```
>>> p.permutations()
[[3, 2, 1], [3, 2, 0], [3, 1, 0], [2, 1, 0]]
```

show(***args*)

Displays the pseudoline arrangement as a wiring diagram.

INPUT:

- `**args` – any arguments to be forwarded to the `show` method. In particular, to tune the dimensions, use the `figsize` argument (example below).

EXAMPLES:

```
sage: from sage.geometry.pseudolines import PseudolineArrangement
sage: permutations = [[3, 2, 1], [3, 2, 0], [3, 1, 0], [2, 1, 0]]
sage: p = PseudolineArrangement(permutations)
sage: p.show(figsize=[7,5])                                          #␣
↪needs sage.plot
```

```
>>> from sage.all import *
>>> from sage.geometry.pseudolines import PseudolineArrangement
>>> permutations = [[Integer(3), Integer(2), Integer(1)], [Integer(3),␣
↪Integer(2), Integer(0)], [Integer(3), Integer(1), Integer(0)], [Integer(2),␣
↪Integer(1), Integer(0)]]
>>> p = PseudolineArrangement(permutations)
>>> p.show(figsize=[Integer(7),Integer(5)])                                  ␣
↪              # needs sage.plot
```

transpositions()

Return the arrangement as $\binom{n}{2}$ transpositions.

See the *pseudolines module*'s documentation for more information on this encoding.

EXAMPLES:

```
sage: from sage.geometry.pseudolines import PseudolineArrangement
sage: permutations = [[3, 2, 1], [3, 2, 0], [3, 1, 0], [2, 1, 0]]
sage: p1 = PseudolineArrangement(permutations)
sage: transpositions = [(3, 2), (3, 1), (0, 3), (2, 1), (0, 2), (0, 1)]
sage: p2 = PseudolineArrangement(transpositions)
sage: p1 == p2
True
sage: p1.transpositions()
[(3, 2), (3, 1), (0, 3), (2, 1), (0, 2), (0, 1)]
sage: p2.transpositions()
[(3, 2), (3, 1), (0, 3), (2, 1), (0, 2), (0, 1)]
```

```
>>> from sage.all import *
>>> from sage.geometry.pseudolines import PseudolineArrangement
>>> permutations = [[Integer(3), Integer(2), Integer(1)], [Integer(3),␣
↪Integer(2), Integer(0)], [Integer(3), Integer(1), Integer(0)], [Integer(2),␣
↪Integer(1), Integer(0)]]
>>> p1 = PseudolineArrangement(permutations)
>>> transpositions = [(Integer(3), Integer(2)), (Integer(3), Integer(1)),␣
```

(continues on next page)

(continued from previous page)

```
↪(Integer(0), Integer(3)), (Integer(2), Integer(1)), (Integer(0),␣
↪Integer(2)), (Integer(0), Integer(1))]
>>> p2 = PseudolineArrangement(transpositions)
>>> p1 == p2
True
>>> p1.transpositions()
[(3, 2), (3, 1), (0, 3), (2, 1), (0, 2), (0, 1)]
>>> p2.transpositions()
[(3, 2), (3, 1), (0, 3), (2, 1), (0, 2), (0, 1)]
```

4.8 Voronoi diagram

This module provides the class `VoronoiDiagram` for computing the Voronoi diagram of a finite list of points in $\mathbf{R}^d$.

class `sage.geometry.voronoi_diagram.`**`VoronoiDiagram`**(*points*)

Bases: `SageObject`

Base class for the Voronoi diagram.

Compute the Voronoi diagram of a list of points.

INPUT:

- `points` – list of points; any valid input for the `PointConfiguration` will do

OUTPUT: an instance of the VoronoiDiagram class

EXAMPLES:

Get the Voronoi diagram for some points in $\mathbf{R}^3$:

```
sage: V = VoronoiDiagram([[1, 3, .3], [2, -2, 1], [-1, 2, -.1]]); V
The Voronoi diagram of 3 points of dimension 3 in the Real Double Field

sage: VoronoiDiagram([])
The empty Voronoi diagram.
```

```
>>> from sage.all import *
>>> V = VoronoiDiagram([[Integer(1), Integer(3), RealNumber('.3')], [Integer(2), -
↪Integer(2), Integer(1)], [-Integer(1), Integer(2), -RealNumber('.1')]]); V
The Voronoi diagram of 3 points of dimension 3 in the Real Double Field

>>> VoronoiDiagram([])
The empty Voronoi diagram.
```

Get the Voronoi diagram of a regular pentagon in `AA^2`. All cells meet at the origin:

```
sage: DV = VoronoiDiagram([[AA(c) for c in v]                              #␣
↪needs sage.rings.number_field
....:                      for v in polytopes.regular_polygon(5).vertices_
↪list()]); DV
The Voronoi diagram of 5 points of dimension 2 in the Algebraic Real Field
sage: all(P.contains([0, 0]) for P in DV.regions().values())              #␣
↪needs sage.rings.number_field
```

(continues on next page)

(continued from previous page)

```
True
sage: any(P.interior_contains([0, 0]) for P in DV.regions().values())          #␣
↪needs sage.rings.number_field
False
```

```
>>> from sage.all import *
>>> DV = VoronoiDiagram([[AA(c) for c in v]                                   #␣
↪needs sage.rings.number_field
...                       for v in polytopes.regular_polygon(Integer(5)).vertices_
↪list()]); DV
The Voronoi diagram of 5 points of dimension 2 in the Algebraic Real Field
>>> all(P.contains([Integer(0), Integer(0)]) for P in DV.regions().values())      ␣
↪                # needs sage.rings.number_field
True
>>> any(P.interior_contains([Integer(0), Integer(0)]) for P in DV.regions().
↪values())            # needs sage.rings.number_field
False
```

If the vertices are not converted to AA before, the method throws an error:

```
sage: polytopes.dodecahedron().vertices_list()[0][0].parent()                   #␣
↪needs sage.groups sage.rings.number_field
Number Field in sqrt5 with defining polynomial x^2 - 5 with sqrt5 = 2.
↪236067977499790?
sage: VoronoiDiagram(polytopes.dodecahedron().vertices_list())                  #␣
↪needs sage.groups sage.rings.number_field
Traceback (most recent call last):
...
NotImplementedError: Base ring of the Voronoi diagram must be
one of QQ, RDF, AA.
```

```
>>> from sage.all import *
>>> polytopes.dodecahedron().vertices_list()[Integer(0)][Integer(0)].parent()    ␣
↪                # needs sage.groups sage.rings.number_field
Number Field in sqrt5 with defining polynomial x^2 - 5 with sqrt5 = 2.
↪236067977499790?
>>> VoronoiDiagram(polytopes.dodecahedron().vertices_list())                  #␣
↪needs sage.groups sage.rings.number_field
Traceback (most recent call last):
...
NotImplementedError: Base ring of the Voronoi diagram must be
one of QQ, RDF, AA.
```

ALGORITHM:

We use hyperplanes tangent to the paraboloid one dimension higher to get a convex polyhedron and then project back to one dimension lower.

Todo

- The dual construction: Delaunay triangulation

- improve 2d-plotting
- implement 3d-plotting
- more general constructions, like Voroi diagrams with weights (power diagrams)

REFERENCES:

- [Mat2002] Ch.5.7, p.118.

AUTHORS:

- Moritz Firsching (2012-09-21)

ambient_dim()

Return the ambient dimension of the points.

EXAMPLES:

```
sage: V = VoronoiDiagram([[.5, 3], [2, 5], [4, 5], [4, -1]])
sage: V.ambient_dim()
2
sage: V = VoronoiDiagram([[1, 2, 3, 4, 5, 6]]); V.ambient_dim()
6
```

```
>>> from sage.all import *
>>> V = VoronoiDiagram([[RealNumber('.5'), Integer(3)], [Integer(2),␣
↪Integer(5)], [Integer(4), Integer(5)], [Integer(4), -Integer(1)]])
>>> V.ambient_dim()
2
>>> V = VoronoiDiagram([[Integer(1), Integer(2), Integer(3), Integer(4),␣
↪Integer(5), Integer(6)]]); V.ambient_dim()
6
```

base_ring()

Return the base_ring of the regions of the Voronoi diagram.

EXAMPLES:

```
sage: V = VoronoiDiagram([[1, 3, 1], [2, -2, 1], [-1, 2, 1/2]]); V.base_ring()
Rational Field
sage: V = VoronoiDiagram([[1, 3.14], [2, -2/3], [-1, 22]]); V.base_ring()
Real Double Field
sage: V = VoronoiDiagram([[1, 3], [2, 4]]); V.base_ring()
Rational Field
```

```
>>> from sage.all import *
>>> V = VoronoiDiagram([[Integer(1), Integer(3), Integer(1)], [Integer(2), -
↪Integer(2), Integer(1)], [-Integer(1), Integer(2), Integer(1)/Integer(2)]]);
↪ V.base_ring()
Rational Field
>>> V = VoronoiDiagram([[Integer(1), RealNumber('3.14')], [Integer(2), -
↪Integer(2)/Integer(3)], [-Integer(1), Integer(22)]]); V.base_ring()
Real Double Field
>>> V = VoronoiDiagram([[Integer(1), Integer(3)], [Integer(2), Integer(4)]]);␣
```

(continues on next page)

(continued from previous page)

```
↪V.base_ring()
Rational Field
```

plot(*cell_colors=None*, ***kwds*)

Return a graphical representation for 2-dimensional Voronoi diagrams.

INPUT:

- `cell_colors` – (default: `None`) provide the colors for the cells, either as dictionary. Randomly colored cells are provided with `None`.
- `**kwds` – optional keyword parameters, passed on as arguments for plot()

OUTPUT: a graphics object

EXAMPLES:

```
sage: # needs sage.plot
sage: P = [[0.671, 0.650], [0.258, 0.767], [0.562, 0.406],
....:      [0.254, 0.709], [0.493, 0.879]]
sage: V = VoronoiDiagram(P); S=V.plot()
sage: show(S, xmin=0, xmax=1, ymin=0, ymax=1, aspect_ratio=1, axes=false)
sage: S = V.plot(cell_colors={0: 'red', 1: 'blue', 2: 'green',
....:                         3: 'white', 4: 'yellow'})
sage: show(S, xmin=0, xmax=1, ymin=0, ymax=1, aspect_ratio=1, axes=false)
sage: S = V.plot(cell_colors=['red', 'blue', 'red', 'white', 'white'])
sage: show(S, xmin=0, xmax=1, ymin=0, ymax=1, aspect_ratio=1, axes=false)
sage: S = V.plot(cell_colors='something else')
Traceback (most recent call last):
...
AssertionError: 'cell_colors' must be a list or a dictionary
```

```
>>> from sage.all import *
>>> # needs sage.plot
>>> P = [[RealNumber('0.671'), RealNumber('0.650')], [RealNumber('0.258'),
↪RealNumber('0.767')], [RealNumber('0.562'), RealNumber('0.406')],
...      [RealNumber('0.254'), RealNumber('0.709')], [RealNumber('0.493'),
↪RealNumber('0.879')]]
>>> V = VoronoiDiagram(P); S=V.plot()
>>> show(S, xmin=Integer(0), xmax=Integer(1), ymin=Integer(0),
↪ymax=Integer(1), aspect_ratio=Integer(1), axes=false)
>>> S = V.plot(cell_colors={Integer(0): 'red', Integer(1): 'blue',
↪Integer(2): 'green',
...                         Integer(3): 'white', Integer(4): 'yellow'})
>>> show(S, xmin=Integer(0), xmax=Integer(1), ymin=Integer(0),
↪ymax=Integer(1), aspect_ratio=Integer(1), axes=false)
>>> S = V.plot(cell_colors=['red', 'blue', 'red', 'white', 'white'])
>>> show(S, xmin=Integer(0), xmax=Integer(1), ymin=Integer(0),
↪ymax=Integer(1), aspect_ratio=Integer(1), axes=false)
>>> S = V.plot(cell_colors='something else')
Traceback (most recent call last):
...
AssertionError: 'cell_colors' must be a list or a dictionary
```

Trying to plot a Voronoi diagram of dimension other than 2 gives an error:

```
sage: VoronoiDiagram([[1, 2, 3], [6, 5, 4]]).plot()                              #␣
↪needs sage.plot
Traceback (most recent call last):
...
NotImplementedError: Plotting of 3-dimensional Voronoi diagrams not
implemented
```

```
>>> from sage.all import *
>>> VoronoiDiagram([[Integer(1), Integer(2), Integer(3)], [Integer(6),␣
↪Integer(5), Integer(4)]]).plot()                        # needs sage.plot
Traceback (most recent call last):
...
NotImplementedError: Plotting of 3-dimensional Voronoi diagrams not
implemented
```

points()

Return the input points (as a PointConfiguration).

EXAMPLES:

```
sage: V = VoronoiDiagram([[.5, 3], [2, 5], [4, 5], [4, -1]]); V.points()
A point configuration in affine 2-space over Real Field
with 53 bits of precision consisting of 4 points.
The triangulations of this point configuration are
assumed to be connected, not necessarily fine,
not necessarily regular.
```

```
>>> from sage.all import *
>>> V = VoronoiDiagram([[RealNumber('.5'), Integer(3)], [Integer(2),␣
↪Integer(5)], [Integer(4), Integer(5)], [Integer(4), -Integer(1)]]); V.
↪points()
A point configuration in affine 2-space over Real Field
with 53 bits of precision consisting of 4 points.
The triangulations of this point configuration are
assumed to be connected, not necessarily fine,
not necessarily regular.
```

regions()

Return the Voronoi regions of the Voronoi diagram as a dictionary of polyhedra.

EXAMPLES:

```
sage: V = VoronoiDiagram([[1, 3, .3], [2, -2, 1], [-1, 2, -.1]])
sage: P = V.points()
sage: V.regions() == {P[0]: Polyhedron(base_ring=RDF, lines=[(-RDF(0.375),␣
↪RDF(0.13888888890000001), RDF(1.5277777779999999))],
....:                                                     rays=[(RDF(9), -RDF(1),␣
↪-RDF(20)), (RDF(4.5), RDF(1), -RDF(25))],
....:                                                     vertices=[(-RDF(1.
↪1074999999999999), RDF(1.149444444), RDF(9.0138888890000004))]),
....:                     P[1]: Polyhedron(base_ring=RDF, lines=[(-RDF(0.375),␣
↪RDF(0.13888888890000001), RDF(1.5277777779999999))],
....:                                                     rays=[(RDF(9), -RDF(1),␣
```

(continues on next page)

(continued from previous page)

```
↪-RDF(20)), (-RDF(2.25), -RDF(1), RDF(2.5))],
....:                                                    vertices=[(-RDF(1.
↪1074999999999999), RDF(1.149444444), RDF(9.0138888890000004))]),
....:                    P[2]: Polyhedron(base_ring=RDF, lines=[(-RDF(0.375),␣
↪RDF(0.13888888890000001), RDF(1.5277777779999999))],
....:                                                   rays=[(RDF(4.5), RDF(1),
↪ -RDF(25)), (-RDF(2.25), -RDF(1), RDF(2.5))],
....:                                                   vertices=[(-RDF(1.
↪1074999999999999), RDF(1.149444444), RDF(9.0138888890000004))])}
True
```

```
>>> from sage.all import *
>>> V = VoronoiDiagram([[Integer(1), Integer(3), RealNumber('.3')],␣
↪[Integer(2), -Integer(2), Integer(1)], [-Integer(1), Integer(2), -
↪RealNumber('.1')]])
>>> P = V.points()
>>> V.regions() == {P[Integer(0)]: Polyhedron(base_ring=RDF, lines=[(-
↪RDF(RealNumber('0.375')), RDF(RealNumber('0.13888888890000001')),␣
↪RDF(RealNumber('1.5277777779999999')))],
...                                                   rays=[(RDF(Integer(9)), -
↪RDF(Integer(1)), -RDF(Integer(20))), (RDF(RealNumber('4.5')),␣
↪RDF(Integer(1)), -RDF(Integer(25)))],
...                                                   vertices=[(-
↪RDF(RealNumber('1.1074999999999999')), RDF(RealNumber('1.149444444')),␣
↪RDF(RealNumber('9.0138888890000004')))]),
...                    P[Integer(1)]: Polyhedron(base_ring=RDF, lines=[(-
↪RDF(RealNumber('0.375')), RDF(RealNumber('0.13888888890000001')),␣
↪RDF(RealNumber('1.5277777779999999')))],
...                                                   rays=[(RDF(Integer(9)), -
↪RDF(Integer(1)), -RDF(Integer(20))), (-RDF(RealNumber('2.25')), -
↪RDF(Integer(1)), RDF(RealNumber('2.5')))],
...                                                    vertices=[(-
↪RDF(RealNumber('1.1074999999999999')), RDF(RealNumber('1.149444444')),␣
↪RDF(RealNumber('9.0138888890000004')))]),
...                    P[Integer(2)]: Polyhedron(base_ring=RDF, lines=[(-
↪RDF(RealNumber('0.375')), RDF(RealNumber('0.13888888890000001')),␣
↪RDF(RealNumber('1.5277777779999999')))],
...                                                   rays=[(RDF(RealNumber('4.5
↪')), RDF(Integer(1)), -RDF(Integer(25))), (-RDF(RealNumber('2.25')), -
↪RDF(Integer(1)), RDF(RealNumber('2.5')))],
...                                                   vertices=[(-
↪RDF(RealNumber('1.1074999999999999')), RDF(RealNumber('1.149444444')),␣
↪RDF(RealNumber('9.0138888890000004')))])}
True
```

CHAPTER

FIVE

HELPER FUNCTIONS

5.1 Find isomorphisms between fans

exception sage.geometry.fan_isomorphism.**FanNotIsomorphicError**

Bases: Exception

Exception to return if there is no fan isomorphism

sage.geometry.fan_isomorphism.**fan_2d_cyclically_ordered_rays**(*fan*)

Return the rays of a 2-dimensional fan in cyclic order.

INPUT:

- fan – a 2-dimensional fan

OUTPUT:

A *PointCollection* containing the rays in one particular cyclic order.

EXAMPLES:

```
sage: rays = ((1, 1), (-1, -1), (-1, 1), (1, -1))
sage: cones = [(0,2), (2,1), (1,3), (3,0)]
sage: fan = Fan(cones, rays)
sage: fan.rays()
N( 1,  1),
N(-1, -1),
N(-1,  1),
N( 1, -1)
in 2-d lattice N
sage: from sage.geometry.fan_isomorphism import fan_2d_cyclically_ordered_rays
sage: fan_2d_cyclically_ordered_rays(fan)
N(-1, -1),
N(-1,  1),
N( 1,  1),
N( 1, -1)
in 2-d lattice N
```

```
>>> from sage.all import *
>>> rays = ((Integer(1), Integer(1)), (-Integer(1), -Integer(1)), (-Integer(1),␣
↪Integer(1)), (Integer(1), -Integer(1)))
>>> cones = [(Integer(0),Integer(2)), (Integer(2),Integer(1)), (Integer(1),
↪Integer(3)), (Integer(3),Integer(0))]
>>> fan = Fan(cones, rays)
```

(continues on next page)

(continued from previous page)

```
>>> fan.rays()
N( 1,  1),
N(-1, -1),
N(-1,  1),
N( 1, -1)
in 2-d lattice N
>>> from sage.geometry.fan_isomorphism import fan_2d_cyclically_ordered_rays
>>> fan_2d_cyclically_ordered_rays(fan)
N(-1, -1),
N(-1,  1),
N( 1,  1),
N( 1, -1)
in 2-d lattice N
```

sage.geometry.fan_isomorphism.**fan_2d_echelon_form**(*fan*)

Return echelon form of a cyclically ordered ray matrix.

INPUT:

- fan – a fan

OUTPUT:

A matrix. The echelon form of the rays in one particular cyclic order.

EXAMPLES:

```
sage: fan = toric_varieties.P2().fan()                                    #␣
↪needs palp sage.graphs
sage: from sage.geometry.fan_isomorphism import fan_2d_echelon_form
sage: fan_2d_echelon_form(fan)                                            #␣
↪needs palp sage.graphs
[ 1  0 -1]
[ 0  1 -1]
```

```
>>> from sage.all import *
>>> fan = toric_varieties.P2().fan()                                      #␣
↪needs palp sage.graphs
>>> from sage.geometry.fan_isomorphism import fan_2d_echelon_form
>>> fan_2d_echelon_form(fan)                                              #␣
↪needs palp sage.graphs
[ 1  0 -1]
[ 0  1 -1]
```

sage.geometry.fan_isomorphism.**fan_2d_echelon_forms**(*fan*)

Return echelon forms of all cyclically ordered ray matrices.

Note that the echelon form of the ordered ray matrices are unique up to different cyclic orderings.

INPUT:

- fan – a fan

OUTPUT:

A set of matrices. The set of all echelon forms for all different cyclic orderings.

EXAMPLES:

```
sage: fan = toric_varieties.P2().fan()                                    #␣
↪needs palp sage.graphs
sage: from sage.geometry.fan_isomorphism import fan_2d_echelon_forms
sage: fan_2d_echelon_forms(fan)                                           #␣
↪needs palp sage.graphs
frozenset({[ 1  0 -1]
           [ 0  1 -1]})

sage: fan = toric_varieties.dP7().fan()                                   #␣
↪needs palp sage.graphs
sage: sorted(fan_2d_echelon_forms(fan))                                   #␣
↪needs palp sage.graphs
[
[ 1  0 -1 -1  0]  [ 1  0 -1 -1  0]  [ 1  0 -1 -1  1]  [ 1  0 -1  0  1]
[ 0  1  0 -1 -1], [ 0  1  1  0 -1], [ 0  1  1  0 -1], [ 0  1  0 -1 -1],

[ 1  0 -1  0  1]
[ 0  1  1 -1 -1]
]
```

```
>>> from sage.all import *
>>> fan = toric_varieties.P2().fan()                                      #␣
↪needs palp sage.graphs
>>> from sage.geometry.fan_isomorphism import fan_2d_echelon_forms
>>> fan_2d_echelon_forms(fan)                                             #␣
↪needs palp sage.graphs
frozenset({[ 1  0 -1]
           [ 0  1 -1]})

>>> fan = toric_varieties.dP7().fan()                                     #␣
↪needs palp sage.graphs
>>> sorted(fan_2d_echelon_forms(fan))                                     #␣
↪needs palp sage.graphs
[
[ 1  0 -1 -1  0]  [ 1  0 -1 -1  0]  [ 1  0 -1 -1  1]  [ 1  0 -1  0  1]
[ 0  1  0 -1 -1], [ 0  1  1  0 -1], [ 0  1  1  0 -1], [ 0  1  0 -1 -1],
<BLANKLINE>
[ 1  0 -1  0  1]
[ 0  1  1 -1 -1]
]
```

sage.geometry.fan_isomorphism.**fan_isomorphic_necessary_conditions**(*fan1*, *fan2*)

Check necessary (but not sufficient) conditions for the fans to be isomorphic.

INPUT:

- `fan1`, `fan2` – two fans

OUTPUT: boolean; `False` if the two fans cannot be isomorphic. `True` if the two fans may be isomorphic.

EXAMPLES:

```
sage: fan1 = toric_varieties.P2().fan()                                   #␣
↪needs palp sage.graphs
```

(continues on next page)

(continued from previous page)

```
sage: fan2 = toric_varieties.dP8().fan()                                      #␣
↪needs palp sage.graphs
sage: from sage.geometry.fan_isomorphism import fan_isomorphic_necessary_
↪conditions
sage: fan_isomorphic_necessary_conditions(fan1, fan2)                         #␣
↪needs palp sage.graphs
False
```

```
>>> from sage.all import *
>>> fan1 = toric_varieties.P2().fan()                                         #␣
↪needs palp sage.graphs
>>> fan2 = toric_varieties.dP8().fan()                                        #␣
↪needs palp sage.graphs
>>> from sage.geometry.fan_isomorphism import fan_isomorphic_necessary_conditions
>>> fan_isomorphic_necessary_conditions(fan1, fan2)                           #␣
↪needs palp sage.graphs
False
```

sage.geometry.fan_isomorphism.**fan_isomorphism_generator**(*fan1*, *fan2*)

Iterate over the isomorphisms from `fan1` to `fan2`.

ALGORITHM:

The `sage.geometry.fan.Fan.vertex_graph()` of the two fans is compared. For each graph isomorphism, we attempt to lift it to an actual isomorphism of fans.

INPUT:

- `fan1`, `fan2` – two fans

OUTPUT:

Yields the fan isomorphisms as matrices acting from the right on rays.

EXAMPLES:

```
sage: fan = toric_varieties.P2().fan()                                        #␣
↪needs palp sage.graphs
sage: from sage.geometry.fan_isomorphism import fan_isomorphism_generator
sage: sorted(fan_isomorphism_generator(fan, fan))                             #␣
↪needs palp sage.graphs
[
[-1 -1]  [-1 -1]  [ 0  1]  [0 1]  [ 1  0]  [1 0]
[ 0  1], [ 1  0], [-1 -1], [1 0], [-1 -1], [0 1]
]
sage: m1 = matrix([(1, 0), (0, -5), (-3, 4)])
sage: m2 = matrix([(3, 0), (1, 0), (-2, 1)])
sage: m1.elementary_divisors() == m2.elementary_divisors() == [1,1,0]
True
sage: fan1 = Fan([Cone([m1*vector([23, 14]), m1*vector([   3,100])]),
....:             Cone([m1*vector([-1,-14]), m1*vector([-100, -5])])])
sage: fan2 = Fan([Cone([m2*vector([23, 14]), m2*vector([   3,100])]),
....:             Cone([m2*vector([-1,-14]), m2*vector([-100, -5])])])
sage: sorted(fan_isomorphism_generator(fan1, fan2))                           #␣
↪needs sage.graphs
```

(continues on next page)

(continued from previous page)

```
[
[-12   1 -5]
[ -4   0 -1]
[ -5   0 -1]
]

sage: m0 = identity_matrix(ZZ, 2)
sage: m1 = matrix([(1, 0), (0, -5), (-3, 4)])
sage: m2 = matrix([(3, 0), (1, 0), (-2, 1)])
sage: m1.elementary_divisors() == m2.elementary_divisors() == [1,1,0]
True
sage: fan0 = Fan([Cone([m0*vector([1,0]), m0*vector([1,1])]),
....:             Cone([m0*vector([1,1]), m0*vector([0,1])])])
sage: fan1 = Fan([Cone([m1*vector([1,0]), m1*vector([1,1])]),
....:             Cone([m1*vector([1,1]), m1*vector([0,1])])])
sage: fan2 = Fan([Cone([m2*vector([1,0]), m2*vector([1,1])]),
....:             Cone([m2*vector([1,1]), m2*vector([0,1])])])
sage: sorted(fan_isomorphism_generator(fan0, fan0))                         #␣
↪needs sage.graphs
[
[0 1]  [1 0]
[1 0], [0 1]
]
sage: sorted(fan_isomorphism_generator(fan1, fan1))                         #␣
↪needs sage.graphs
[
[ -3 -20  28]  [1 0 0]
[ -1  -4   7]  [0 1 0]
[ -1  -5   8], [0 0 1]
]
sage: sorted(fan_isomorphism_generator(fan1, fan2))                         #␣
↪needs sage.graphs
[
[-24  -3   7]  [-12   1  -5]
[ -7  -1   2]  [ -4   0  -1]
[ -8  -1   2], [ -5   0  -1]
]
sage: sorted(fan_isomorphism_generator(fan2, fan1))                         #␣
↪needs sage.graphs
[
[  0   1  -1]  [ 0  1 -1]
[  1 -13   8]  [ 2 -8  1]
[  0  -5   4], [ 1  0 -3]
]
```

```
>>> from sage.all import *
>>> fan = toric_varieties.P2().fan()                                       #␣
↪needs palp sage.graphs
>>> from sage.geometry.fan_isomorphism import fan_isomorphism_generator
>>> sorted(fan_isomorphism_generator(fan, fan))                            #␣
↪needs palp sage.graphs
[
```

(continues on next page)

(continued from previous page)

```
[-1 -1]  [-1 -1]  [ 0  1]  [0 1]  [ 1  0]  [1 0]
[ 0  1], [ 1  0], [-1 -1], [1 0], [-1 -1], [0 1]
]
>>> m1 = matrix([(Integer(1), Integer(0)), (Integer(0), -Integer(5)), (-
↪Integer(3), Integer(4))])
>>> m2 = matrix([(Integer(3), Integer(0)), (Integer(1), Integer(0)), (-Integer(2),
↪ Integer(1))])
>>> m1.elementary_divisors() == m2.elementary_divisors() == [Integer(1),
↪Integer(1),Integer(0)]
True
>>> fan1 = Fan([Cone([m1*vector([Integer(23), Integer(14)]), m1*vector([  ␣
↪Integer(3),Integer(100)])]),
...             Cone([m1*vector([-Integer(1),-Integer(14)]), m1*vector([-
↪Integer(100), -Integer(5)])])])
>>> fan2 = Fan([Cone([m2*vector([Integer(23), Integer(14)]), m2*vector([  ␣
↪Integer(3),Integer(100)])]),
...             Cone([m2*vector([-Integer(1),-Integer(14)]), m2*vector([-
↪Integer(100), -Integer(5)])])])
>>> sorted(fan_isomorphism_generator(fan1, fan2))                          #␣
↪needs sage.graphs
[
[-12  1 -5]
[ -4  0 -1]
[ -5  0 -1]
]

>>> m0 = identity_matrix(ZZ, Integer(2))
>>> m1 = matrix([(Integer(1), Integer(0)), (Integer(0), -Integer(5)), (-
↪Integer(3), Integer(4))])
>>> m2 = matrix([(Integer(3), Integer(0)), (Integer(1), Integer(0)), (-Integer(2),
↪ Integer(1))])
>>> m1.elementary_divisors() == m2.elementary_divisors() == [Integer(1),
↪Integer(1),Integer(0)]
True
>>> fan0 = Fan([Cone([m0*vector([Integer(1),Integer(0)]), m0*vector([Integer(1),
↪Integer(1)])]),
...             Cone([m0*vector([Integer(1),Integer(1)]), m0*vector([Integer(0),
↪Integer(1)])])])
>>> fan1 = Fan([Cone([m1*vector([Integer(1),Integer(0)]), m1*vector([Integer(1),
↪Integer(1)])]),
...             Cone([m1*vector([Integer(1),Integer(1)]), m1*vector([Integer(0),
↪Integer(1)])])])
>>> fan2 = Fan([Cone([m2*vector([Integer(1),Integer(0)]), m2*vector([Integer(1),
↪Integer(1)])]),
...             Cone([m2*vector([Integer(1),Integer(1)]), m2*vector([Integer(0),
↪Integer(1)])])])
>>> sorted(fan_isomorphism_generator(fan0, fan0))                          #␣
↪needs sage.graphs
[
[0 1]  [1 0]
[1 0], [0 1]
]
```

(continues on next page)

(continued from previous page)

```
>>> sorted(fan_isomorphism_generator(fan1, fan1))                           #␣
↪needs sage.graphs
[
[ -3 -20  28]  [1 0 0]
[ -1  -4   7]  [0 1 0]
[ -1  -5   8], [0 0 1]
]
>>> sorted(fan_isomorphism_generator(fan1, fan2))                           #␣
↪needs sage.graphs
[
[-24  -3   7]  [-12   1  -5]
[ -7  -1   2]  [ -4   0  -1]
[ -8  -1   2], [ -5   0  -1]
]
>>> sorted(fan_isomorphism_generator(fan2, fan1))                           #␣
↪needs sage.graphs
[
[  0   1  -1]  [ 0  1 -1]
[  1 -13   8]  [ 2 -8  1]
[  0  -5   4], [ 1  0 -3]
]
```

sage.geometry.fan_isomorphism.**find_isomorphism**(*fan1*, *fan2*, *check=False*)

Find an isomorphism of the two fans.

INPUT:

- fan1, fan2 – two fans
- check – boolean (default: False); passed to the fan morphism constructor, see *FanMorphism()*

OUTPUT:

A fan isomorphism. If the fans are not isomorphic, a *FanNotIsomorphicError* is raised.

EXAMPLES:

```
sage: rays = ((1, 1), (0, 1), (-1, -1), (3, 1))
sage: cones = [(0,1), (1,2), (2,3), (3,0)]
sage: fan1 = Fan(cones, rays)

sage: m = matrix([[-2,3],[1,-1]])
sage: m.det() == -1
True
sage: fan2 = Fan(cones, [vector(r)*m for r in rays])

sage: from sage.geometry.fan_isomorphism import find_isomorphism
sage: find_isomorphism(fan1, fan2, check=True)                              #␣
↪needs sage.graphs
Fan morphism defined by the matrix
[-2  3]
[ 1 -1]
Domain fan: Rational polyhedral fan in 2-d lattice N
Codomain fan: Rational polyhedral fan in 2-d lattice N
```

(continues on next page)

(continued from previous page)

```
sage: find_isomorphism(fan1, toric_varieties.P2().fan())                    #␣
↪needs palp sage.graphs
Traceback (most recent call last):
...
FanNotIsomorphicError

sage: fan1 = Fan(cones=[[1,3,4,5],[0,1,2,3],[2,3,4],[0,1,5]],
....:            rays=[(-1,-1,0),(-1,-1,3),(-1,1,-1),(-1,3,-1),(0,2,-1),(1,-1,1)])
sage: fan2 = Fan(cones=[[0,2,3,5],[0,1,4,5],[0,1,2],[3,4,5]],
....:            rays=[(-1,-1,-1),(-1,-1,0),(-1,1,-1),(0,2,-1),(1,-1,1),(3,-1,-
↪1)])
sage: fan1.is_isomorphic(fan2)                                              #␣
↪needs sage.graphs
True
```

```
>>> from sage.all import *
>>> rays = ((Integer(1), Integer(1)), (Integer(0), Integer(1)), (-Integer(1), -
↪Integer(1)), (Integer(3), Integer(1)))
>>> cones = [(Integer(0),Integer(1)), (Integer(1),Integer(2)), (Integer(2),
↪Integer(3)), (Integer(3),Integer(0))]
>>> fan1 = Fan(cones, rays)

>>> m = matrix([[-Integer(2),Integer(3)],[Integer(1),-Integer(1)]])
>>> m.det() == -Integer(1)
True
>>> fan2 = Fan(cones, [vector(r)*m for r in rays])

>>> from sage.geometry.fan_isomorphism import find_isomorphism
>>> find_isomorphism(fan1, fan2, check=True)                               #␣
↪needs sage.graphs
Fan morphism defined by the matrix
[-2  3]
[ 1 -1]
Domain fan: Rational polyhedral fan in 2-d lattice N
Codomain fan: Rational polyhedral fan in 2-d lattice N

>>> find_isomorphism(fan1, toric_varieties.P2().fan())                     #␣
↪needs palp sage.graphs
Traceback (most recent call last):
...
FanNotIsomorphicError

>>> fan1 = Fan(cones=[[Integer(1),Integer(3),Integer(4),Integer(5)],[Integer(0),
↪Integer(1),Integer(2),Integer(3)],[Integer(2),Integer(3),Integer(4)],
↪[Integer(0),Integer(1),Integer(5)]],
...            rays=[(-Integer(1),-Integer(1),Integer(0)),(-Integer(1),-
↪Integer(1),Integer(3)),(-Integer(1),Integer(1),-Integer(1)),(-Integer(1),
↪Integer(3),-Integer(1)),(Integer(0),Integer(2),-Integer(1)),(Integer(1),-
↪Integer(1),Integer(1))])
>>> fan2 = Fan(cones=[[Integer(0),Integer(2),Integer(3),Integer(5)],[Integer(0),
↪Integer(1),Integer(4),Integer(5)],[Integer(0),Integer(1),Integer(2)],
↪[Integer(3),Integer(4),Integer(5)]],
```

(continues on next page)

(continued from previous page)

```
...             rays=[(-Integer(1),-Integer(1),-Integer(1)),(-Integer(1),-
↪Integer(1),Integer(0)),(-Integer(1),Integer(1),-Integer(1)),(Integer(0),
↪Integer(2),-Integer(1)),(Integer(1),-Integer(1),Integer(1)),(Integer(3),-
↪Integer(1),-Integer(1))])
>>> fan1.is_isomorphic(fan2)                                            #␣
↪needs sage.graphs
True
```

5.2 Construction of finite atomic and coatomic lattices from incidences

This module provides the function `lattice_from_incidences()` for computing finite atomic and coatomic lattices in the sense of partially ordered sets where any two elements have meet and joint. For example, the face lattice of a polyhedron.

`sage.geometry.hasse_diagram.`**`lattice_from_incidences`**(*atom_to_coatoms*, *coatom_to_atoms*, *face_constructor=None*, *required_atoms=None*, *key=None*, ***kwds*)

Compute an atomic and coatomic lattice from the incidence between atoms and coatoms.

INPUT:

- `atom_to_coatoms` – list; `atom_to_coatom[i]` should list all coatoms over the `i`-th atom
- `coatom_to_atoms` – list; `coatom_to_atom[i]` should list all atoms under the `i`-th coatom
- `face_constructor` – function or class taking as the first two arguments sorted `tuple` of integers and any keyword arguments. It will be called to construct a face over atoms passed as the first argument and under coatoms passed as the second argument. Default implementation will just return these two tuples as a tuple;
- `required_atoms` – list of atoms (default: `None`); each non-empty "face" requires at least one of the specified atoms present. Used to ensure that each face has a vertex.
- `key` – any hashable value (default: `None`); it is passed down to `FinitePoset`
- all other keyword arguments will be passed to `face_constructor` on each call

OUTPUT:

- `finite poset` with elements constructed by `face_constructor`.

> **Note**
>
> In addition to the specified partial order, finite posets in Sage have internal total linear order of elements which extends the partial one. This function will try to make this internal order to start with the bottom and atoms in the order corresponding to `atom_to_coatoms` and to finish with coatoms in the order corresponding to `coatom_to_atoms` and the top. This may not be possible if atoms and coatoms are the same, in which case the preference is given to the first list.

ALGORITHM:

The detailed description of the used algorithm is given in [KP2002].

The code of this function follows the pseudo-code description in the section 2.5 of the paper, although it is mostly based on frozen sets instead of sorted lists - this makes the implementation easier and should not cost a big performance penalty. (If one wants to make this function faster, it should be probably written in Cython.)

While the title of the paper mentions only polytopes, the algorithm (and the implementation provided here) is applicable to any atomic and coatomic lattice if both incidences are given, see Section 3.4.

In particular, this function can be used for strictly convex cones and complete fans.

REFERENCES: [KP2002]

AUTHORS:

- Andrey Novoseltsev (2010-05-13) with thanks to Marshall Hampton for the reference.

EXAMPLES:

Let us construct the lattice of subsets of {0, 1, 2}. Our atoms are {0}, {1}, and {2}, while our coatoms are {0,1}, {0,2}, and {1,2}. Then incidences are

```
sage: atom_to_coatoms = [(0,1), (0,2), (1,2)]
sage: coatom_to_atoms = [(0,1), (0,2), (1,2)]
```

```
>>> from sage.all import *
>>> atom_to_coatoms = [(Integer(0),Integer(1)), (Integer(0),Integer(2)),␣
↪(Integer(1),Integer(2))]
>>> coatom_to_atoms = [(Integer(0),Integer(1)), (Integer(0),Integer(2)),␣
↪(Integer(1),Integer(2))]
```

and we can compute the lattice as

```
sage: from sage.geometry.cone import lattice_from_incidences
sage: L = lattice_from_incidences(atom_to_coatoms, coatom_to_atoms); L         #␣
↪needs sage.graphs
Finite lattice containing 8 elements with distinguished linear extension
sage: for level in L.level_sets(): print(level)                                #␣
↪needs sage.graphs
[((), (0, 1, 2))]
[((0,), (0, 1)), ((1,), (0, 2)), ((2,), (1, 2))]
[((0, 1), (0,)), ((0, 2), (1,)), ((1, 2), (2,))]
[((0, 1, 2), ())]
```

```
>>> from sage.all import *
>>> from sage.geometry.cone import lattice_from_incidences
>>> L = lattice_from_incidences(atom_to_coatoms, coatom_to_atoms); L         #␣
↪needs sage.graphs
Finite lattice containing 8 elements with distinguished linear extension
>>> for level in L.level_sets(): print(level)                                #␣
↪needs sage.graphs
[((), (0, 1, 2))]
[((0,), (0, 1)), ((1,), (0, 2)), ((2,), (1, 2))]
[((0, 1), (0,)), ((0, 2), (1,)), ((1, 2), (2,))]
[((0, 1, 2), ())]
```

For more involved examples see the *source code* of `sage.geometry.cone.ConvexRationalPolyhedralCone.face_lattice()` and `sage.geometry.fan.RationalPolyhedralFan._compute_cone_lattice()`.

5.3 Cython helper methods to compute integral points in polyhedra

class sage.geometry.integral_points.**InequalityCollection**

Bases: object

A collection of inequalities.

INPUT:

- polyhedron – a polyhedron defining the inequalities
- permutation – list; a 0-based permutation of the coordinates Will be used to permute the coordinates of the inequality
- box_min, box_max – the (not permuted) minimal and maximal coordinates of the bounding box; used for bounds checking

EXAMPLES:

```
sage: from sage.geometry.integral_points import InequalityCollection
sage: P_QQ = Polyhedron(identity_matrix(3).columns() + [(-2, -1,-1)], base_
↪ring=QQ)
sage: ieq = InequalityCollection(P_QQ, [0,1,2], [0]*3,[1]*3); ieq
The collection of inequalities
integer: (3, -2, -2) x + 2 >= 0
integer: (-1, 4, -1) x + 1 >= 0
integer: (-1, -1, 4) x + 1 >= 0
integer: (-1, -1, -1) x + 1 >= 0

sage: P_RR = Polyhedron(identity_matrix(2).columns() + [(-2.7, -1)], base_
↪ring=RDF)
sage: InequalityCollection(P_RR, [0,1], [0]*2, [1]*2)
The collection of inequalities
integer: (-1, -1) x + 1 >= 0
generic: (-1.0, 3.7) x + 1.0 >= 0
generic: (1.0, -1.35) x + 1.35 >= 0

sage: line = Polyhedron(eqns=[(2,3,7)])
sage: InequalityCollection(line, [0,1], [0]*2, [1]*2 )
The collection of inequalities
integer: (3, 7) x + 2 >= 0
integer: (-3, -7) x + -2 >= 0
```

```
>>> from sage.all import *
>>> from sage.geometry.integral_points import InequalityCollection
>>> P_QQ = Polyhedron(identity_matrix(Integer(3)).columns() + [(-Integer(2), -
↪Integer(1),-Integer(1))], base_ring=QQ)
>>> ieq = InequalityCollection(P_QQ, [Integer(0),Integer(1),Integer(2)],␣
↪[Integer(0)]*Integer(3),[Integer(1)]*Integer(3)); ieq
The collection of inequalities
integer: (3, -2, -2) x + 2 >= 0
integer: (-1, 4, -1) x + 1 >= 0
integer: (-1, -1, 4) x + 1 >= 0
integer: (-1, -1, -1) x + 1 >= 0

>>> P_RR = Polyhedron(identity_matrix(Integer(2)).columns() + [(-RealNumber('2.7
```

(continues on next page)

(continued from previous page)

```
↪'), -Integer(1))], base_ring=RDF)
>>> InequalityCollection(P_RR, [Integer(0),Integer(1)], [Integer(0)]*Integer(2),␣
↪[Integer(1)]*Integer(2))
The collection of inequalities
integer: (-1, -1) x + 1 >= 0
generic: (-1.0, 3.7) x + 1.0 >= 0
generic: (1.0, -1.35) x + 1.35 >= 0

>>> line = Polyhedron(eqns=[(Integer(2),Integer(3),Integer(7))])
>>> InequalityCollection(line, [Integer(0),Integer(1)], [Integer(0)]*Integer(2),␣
↪[Integer(1)]*Integer(2) )
The collection of inequalities
integer: (3, 7) x + 2 >= 0
integer: (-3, -7) x + -2 >= 0
```

are_satisfied(*inner_loop_variable*)

Return whether all inequalities are satisfied.

You must call *prepare_inner_loop()* before calling this method.

INPUT:

- `inner_loop_variable` – integer; the 0th coordinate of the lattice point

OUTPUT: boolean; whether the lattice point is in the polyhedron

EXAMPLES:

```
sage: from sage.geometry.integral_points import InequalityCollection
sage: line = Polyhedron(eqns=[(2,3,7)])
sage: ieq = InequalityCollection(line, [0,1], [0]*2, [1]*2 )
sage: ieq.prepare_next_to_inner_loop([3,4])
sage: ieq.prepare_inner_loop([3,4])
sage: ieq.are_satisfied(3)
False
```

```
>>> from sage.all import *
>>> from sage.geometry.integral_points import InequalityCollection
>>> line = Polyhedron(eqns=[(Integer(2),Integer(3),Integer(7))])
>>> ieq = InequalityCollection(line, [Integer(0),Integer(1)],␣
↪[Integer(0)]*Integer(2), [Integer(1)]*Integer(2) )
>>> ieq.prepare_next_to_inner_loop([Integer(3),Integer(4)])
>>> ieq.prepare_inner_loop([Integer(3),Integer(4)])
>>> ieq.are_satisfied(Integer(3))
False
```

prepare_inner_loop(*p*)

Peel off the inner loop.

In the inner loop of *rectangular_box_points()*, we have to repeatedly evaluate $Ax + b \geq 0$. To speed up computation, we pre-evaluate

$$c = Ax - A_0 x_0 + b = b + \sum_{i=1} A_i x_i$$

and only test $A_0 x_0 + c \geq 0$ in the inner loop.

You must call *prepare_next_to_inner_loop()* before calling this method.

INPUT:

- p – the coordinates of the point to loop over. Only the p[1:] entries are used

EXAMPLES:

```
sage: from sage.geometry.integral_points import InequalityCollection, print_
↪cache
sage: P = Polyhedron(ieqs=[(2,3,7,11)])
sage: ieq = InequalityCollection(P, [0,1,2], [0]*3,[1]*3); ieq
The collection of inequalities
integer: (3, 7, 11) x + 2 >= 0
sage: ieq.prepare_next_to_inner_loop([2,1,3])
sage: ieq.prepare_inner_loop([2,1,3])
sage: print_cache(ieq)
Cached inner loop: 3 * x_0 + 42 >= 0
Cached next-to-inner loop: 3 * x_0 + 7 * x_1 + 35 >= 0
```

```
>>> from sage.all import *
>>> from sage.geometry.integral_points import InequalityCollection, print_
↪cache
>>> P = Polyhedron(ieqs=[(Integer(2),Integer(3),Integer(7),Integer(11))])
>>> ieq = InequalityCollection(P, [Integer(0),Integer(1),Integer(2)],␣
↪[Integer(0)]*Integer(3),[Integer(1)]*Integer(3)); ieq
The collection of inequalities
integer: (3, 7, 11) x + 2 >= 0
>>> ieq.prepare_next_to_inner_loop([Integer(2),Integer(1),Integer(3)])
>>> ieq.prepare_inner_loop([Integer(2),Integer(1),Integer(3)])
>>> print_cache(ieq)
Cached inner loop: 3 * x_0 + 42 >= 0
Cached next-to-inner loop: 3 * x_0 + 7 * x_1 + 35 >= 0
```

prepare_next_to_inner_loop(*p*)

Peel off the next-to-inner loop.

In the next-to-inner loop of *rectangular_box_points()*, we have to repeatedly evaluate $Ax - A_0x_0 + b$. To speed up computation, we pre-evaluate

$$c = b + \sum_{i=2} A_i x_i$$

and only compute $Ax - A_0x_0 + b = A_1x_1 + c \geq 0$ in the next-to-inner loop.

INPUT:

- p – the point coordinates. Only p[2:] coordinates are potentially used by this method

EXAMPLES:

```
sage: from sage.geometry.integral_points import InequalityCollection, print_
↪cache
sage: P = Polyhedron(ieqs=[(2,3,7,11)])
sage: ieq = InequalityCollection(P, [0,1,2], [0]*3,[1]*3); ieq
The collection of inequalities
integer: (3, 7, 11) x + 2 >= 0
```

(continues on next page)

(continued from previous page)

```
sage: ieq.prepare_next_to_inner_loop([2,1,3])
sage: ieq.prepare_inner_loop([2,1,3])
sage: print_cache(ieq)
Cached inner loop: 3 * x_0 + 42 >= 0
Cached next-to-inner loop: 3 * x_0 + 7 * x_1 + 35 >= 0
```

```
>>> from sage.all import *
>>> from sage.geometry.integral_points import InequalityCollection, print_
↪cache
>>> P = Polyhedron(ieqs=[(Integer(2),Integer(3),Integer(7),Integer(11))])
>>> ieq = InequalityCollection(P, [Integer(0),Integer(1),Integer(2)],␣
↪[Integer(0)]*Integer(3),[Integer(1)]*Integer(3)); ieq
The collection of inequalities
integer: (3, 7, 11) x + 2 >= 0
>>> ieq.prepare_next_to_inner_loop([Integer(2),Integer(1),Integer(3)])
>>> ieq.prepare_inner_loop([Integer(2),Integer(1),Integer(3)])
>>> print_cache(ieq)
Cached inner loop: 3 * x_0 + 42 >= 0
Cached next-to-inner loop: 3 * x_0 + 7 * x_1 + 35 >= 0
```

satisfied_as_equalities(*inner_loop_variable*)

Return the inequalities (by their index) that are satisfied as equalities.

INPUT:

- `inner_loop_variable` – integer; the 0th coordinate of the lattice point

OUTPUT:

A set of integers in ascending order. Each integer is the index of a H-representation object of the polyhedron (either a inequality or an equation).

EXAMPLES:

```
sage: from sage.geometry.integral_points import InequalityCollection
sage: quadrant = Polyhedron(rays=[(1,0), (0,1)])
sage: ieqs = InequalityCollection(quadrant, [0,1], [-1]*2, [1]*2 )
sage: ieqs.prepare_next_to_inner_loop([-1,0])
sage: ieqs.prepare_inner_loop([-1,0])
sage: ieqs.satisfied_as_equalities(-1)
frozenset({1})
sage: ieqs.satisfied_as_equalities(0)
frozenset({0, 1})
sage: ieqs.satisfied_as_equalities(1)
frozenset({1})
```

```
>>> from sage.all import *
>>> from sage.geometry.integral_points import InequalityCollection
>>> quadrant = Polyhedron(rays=[(Integer(1),Integer(0)), (Integer(0),
↪Integer(1))])
>>> ieqs = InequalityCollection(quadrant, [Integer(0),Integer(1)], [-
↪Integer(1)]*Integer(2), [Integer(1)]*Integer(2) )
>>> ieqs.prepare_next_to_inner_loop([-Integer(1),Integer(0)])
>>> ieqs.prepare_inner_loop([-Integer(1),Integer(0)])
```

(continues on next page)

(continued from previous page)

```
>>> ieqs.satisfied_as_equalities(-Integer(1))
frozenset({1})
>>> ieqs.satisfied_as_equalities(Integer(0))
frozenset({0, 1})
>>> ieqs.satisfied_as_equalities(Integer(1))
frozenset({1})
```

swap_ineq_to_front(*i*)

Swap the i-th entry of the list to the front of the list of inequalities.

INPUT:

- i – integer; the *Inequality_int* to swap to the beginning of the list of integral inequalities

EXAMPLES:

```
sage: from sage.geometry.integral_points import InequalityCollection
sage: P_QQ = Polyhedron(identity_matrix(3).columns() + [(-2, -1,-1)], base_
↪ring=QQ)
sage: iec = InequalityCollection(P_QQ, [0,1,2], [0]*3,[1]*3)
sage: iec
The collection of inequalities
integer: (3, -2, -2) x + 2 >= 0
integer: (-1, 4, -1) x + 1 >= 0
integer: (-1, -1, 4) x + 1 >= 0
integer: (-1, -1, -1) x + 1 >= 0
sage: iec.swap_ineq_to_front(3)
sage: iec
The collection of inequalities
integer: (-1, -1, -1) x + 1 >= 0
integer: (3, -2, -2) x + 2 >= 0
integer: (-1, 4, -1) x + 1 >= 0
integer: (-1, -1, 4) x + 1 >= 0
```

```
>>> from sage.all import *
>>> from sage.geometry.integral_points import InequalityCollection
>>> P_QQ = Polyhedron(identity_matrix(Integer(3)).columns() + [(-Integer(2), -
↪Integer(1),-Integer(1))], base_ring=QQ)
>>> iec = InequalityCollection(P_QQ, [Integer(0),Integer(1),Integer(2)],␣
↪[Integer(0)]*Integer(3),[Integer(1)]*Integer(3))
>>> iec
The collection of inequalities
integer: (3, -2, -2) x + 2 >= 0
integer: (-1, 4, -1) x + 1 >= 0
integer: (-1, -1, 4) x + 1 >= 0
integer: (-1, -1, -1) x + 1 >= 0
>>> iec.swap_ineq_to_front(Integer(3))
>>> iec
The collection of inequalities
integer: (-1, -1, -1) x + 1 >= 0
integer: (3, -2, -2) x + 2 >= 0
integer: (-1, 4, -1) x + 1 >= 0
integer: (-1, -1, 4) x + 1 >= 0
```

class sage.geometry.integral_points.**Inequality_generic**

Bases: object

An inequality whose coefficients are arbitrary Python/Sage objects

INPUT:

- A – list of coefficients
- b – element

OUTPUT: inequality $Ax + b \geq 0$

EXAMPLES:

```
sage: from sage.geometry.integral_points import Inequality_generic
sage: Inequality_generic([2 * pi, sqrt(3), 7/2], -5.5)                          #␣
↪needs sage.symbolic
generic: (2*pi, sqrt(3), 7/2) x + -5.50000000000000 >= 0
```

```
>>> from sage.all import *
>>> from sage.geometry.integral_points import Inequality_generic
>>> Inequality_generic([Integer(2) * pi, sqrt(Integer(3)), Integer(7)/Integer(2)],
↪ -RealNumber('5.5'))                             # needs sage.symbolic
generic: (2*pi, sqrt(3), 7/2) x + -5.50000000000000 >= 0
```

class sage.geometry.integral_points.**Inequality_int**

Bases: object

Fast version of inequality in the case that all coefficients fit into machine ints.

INPUT:

- A – list of integers
- b – integer
- max_abs_coordinates – the maximum of the coordinates that one wants to evaluate the coordinates on; used for overflow checking

OUTPUT:

Inequality $Ax + b \geq 0$. A OverflowError is raised if a machine integer is not long enough to hold the results. A ValueError is raised if some of the input is not integral.

EXAMPLES:

```
sage: from sage.geometry.integral_points import Inequality_int
sage: Inequality_int([2,3,7], -5, [10]*3)
integer: (2, 3, 7) x + -5 >= 0

sage: Inequality_int([1]*21, -5, [10]*21)
Traceback (most recent call last):
...
OverflowError: Dimension limit exceeded.

sage: Inequality_int([2,3/2,7], -5, [10]*3)
Traceback (most recent call last):
...
ValueError: Not integral.
```

(continues on next page)

(continued from previous page)

```
sage: Inequality_int([2,3,7], -5.2, [10]*3)
Traceback (most recent call last):
...
ValueError: Not integral.

sage: Inequality_int([2,3,7], -5*10^50, [10]*3)  # actual error message can␣
↪differ between 32 and 64 bit
Traceback (most recent call last):
...
OverflowError: ...
```

```
>>> from sage.all import *
>>> from sage.geometry.integral_points import Inequality_int
>>> Inequality_int([Integer(2),Integer(3),Integer(7)], -Integer(5),␣
↪[Integer(10)]*Integer(3))
integer: (2, 3, 7) x + -5 >= 0

>>> Inequality_int([Integer(1)]*Integer(21), -Integer(5),␣
↪[Integer(10)]*Integer(21))
Traceback (most recent call last):
...
OverflowError: Dimension limit exceeded.

>>> Inequality_int([Integer(2),Integer(3)/Integer(2),Integer(7)], -Integer(5),␣
↪[Integer(10)]*Integer(3))
Traceback (most recent call last):
...
ValueError: Not integral.

>>> Inequality_int([Integer(2),Integer(3),Integer(7)], -RealNumber('5.2'),␣
↪[Integer(10)]*Integer(3))
Traceback (most recent call last):
...
ValueError: Not integral.

>>> Inequality_int([Integer(2),Integer(3),Integer(7)], -
↪Integer(5)*Integer(10)**Integer(50), [Integer(10)]*Integer(3))  # actual error␣
↪message can differ between 32 and 64 bit
Traceback (most recent call last):
...
OverflowError: ...
```

`sage.geometry.integral_points.`**`loop_over_parallelotope_points`**(*e*, *d*, *VDinv*, *R*, *lattice*, *A=None*, *b=None*)

The inner loop of `parallelotope_points()`.

INPUT:

See `parallelotope_points()` for `e`, `d`, `VDinv`, `R`, `lattice`.

- `A`, `b` – either both `None` or a vector and number. If present, only the parallelotope points satisfying $Ax \leq b$ are returned.

OUTPUT:

The points of the half-open parallelotope as a tuple of lattice points.

EXAMPLES:

```
sage: e = [3]
sage: d = prod(e)
sage: VDinv = matrix(ZZ, [[1]])
sage: R = column_matrix(ZZ,[3,3,3])
sage: lattice = ZZ^3
sage: from sage.geometry.integral_points import loop_over_parallelotope_points
sage: loop_over_parallelotope_points(e, d, VDinv, R, lattice)
((0, 0, 0), (1, 1, 1), (2, 2, 2))

sage: A = vector(ZZ, [1,0,0])
sage: b = 1
sage: loop_over_parallelotope_points(e, d, VDinv, R, lattice, A, b)
((0, 0, 0), (1, 1, 1))
```

```
>>> from sage.all import *
>>> e = [Integer(3)]
>>> d = prod(e)
>>> VDinv = matrix(ZZ, [[Integer(1)]])
>>> R = column_matrix(ZZ,[Integer(3),Integer(3),Integer(3)])
>>> lattice = ZZ**Integer(3)
>>> from sage.geometry.integral_points import loop_over_parallelotope_points
>>> loop_over_parallelotope_points(e, d, VDinv, R, lattice)
((0, 0, 0), (1, 1, 1), (2, 2, 2))

>>> A = vector(ZZ, [Integer(1),Integer(0),Integer(0)])
>>> b = Integer(1)
>>> loop_over_parallelotope_points(e, d, VDinv, R, lattice, A, b)
((0, 0, 0), (1, 1, 1))
```

`sage.geometry.integral_points.`**`parallelotope_points`**`(`*spanning_points*, *lattice*`)`

Return integral points in the parallelotope starting at the origin and spanned by the `spanning_points`.

See `semigroup_generators()` for a description of the algorithm.

INPUT:

- `spanning_points` – a non-empty list of linearly independent rays (**Z**-vectors or *toric lattice* elements), not necessarily primitive lattice points.

OUTPUT:

The tuple of all lattice points in the half-open parallelotope spanned by the rays r_i,

$$par(\{r_i\}) = \sum_{0\leq a_i<1} a_i r_i$$

By half-open parallelotope, we mean that the points in the facets not meeting the origin are omitted.

EXAMPLES:

Note how the points on the outward-facing factes are omitted:

```
sage: from sage.geometry.integral_points import parallelotope_points
sage: rays = list(map(vector, [(2,0), (0,2)]))
sage: parallelotope_points(rays, ZZ^2)
((0, 0), (0, 1), (1, 0), (1, 1))
```

```
>>> from sage.all import *
>>> from sage.geometry.integral_points import parallelotope_points
>>> rays = list(map(vector, [(Integer(2),Integer(0)), (Integer(0),Integer(2))]))
>>> parallelotope_points(rays, ZZ**Integer(2))
((0, 0), (0, 1), (1, 0), (1, 1))
```

The rays can also be toric lattice points:

```
sage: rays = list(map(ToricLattice(2), [(2,0), (0,2)]))
sage: parallelotope_points(rays, ToricLattice(2))
(N(0, 0), N(0, 1), N(1, 0), N(1, 1))
```

```
>>> from sage.all import *
>>> rays = list(map(ToricLattice(Integer(2)), [(Integer(2),Integer(0)),
↪(Integer(0),Integer(2))]))
>>> parallelotope_points(rays, ToricLattice(Integer(2)))
(N(0, 0), N(0, 1), N(1, 0), N(1, 1))
```

A non-smooth cone:

```
sage: c = Cone([ (1,0), (1,2) ])
sage: parallelotope_points(c.rays(), c.lattice())
(N(0, 0), N(1, 1))
```

```
>>> from sage.all import *
>>> c = Cone([ (Integer(1),Integer(0)), (Integer(1),Integer(2)) ])
>>> parallelotope_points(c.rays(), c.lattice())
(N(0, 0), N(1, 1))
```

A `ValueError` is raised if the `spanning_points` are not linearly independent:

```
sage: rays = list(map(ToricLattice(2), [(1,1)]*2))
sage: parallelotope_points(rays, ToricLattice(2))
Traceback (most recent call last):
...
ValueError: The spanning points are not linearly independent!
```

```
>>> from sage.all import *
>>> rays = list(map(ToricLattice(Integer(2)), [(Integer(1),
↪Integer(1))]*Integer(2)))
>>> parallelotope_points(rays, ToricLattice(Integer(2)))
Traceback (most recent call last):
...
ValueError: The spanning points are not linearly independent!
```

`sage.geometry.integral_points.`**`print_cache`**`(`*inequality_collection*`)`

Print the cached values in *`Inequality_int`* (for debugging/doctesting only).

EXAMPLES:

```
sage: from sage.geometry.integral_points import InequalityCollection, print_cache
sage: P = Polyhedron(ieqs=[(2,3,7)])
sage: ieq = InequalityCollection(P, [0,1], [0]*2,[1]*2); ieq
The collection of inequalities
integer: (3, 7) x + 2 >= 0
sage: ieq.prepare_next_to_inner_loop([3,5])
sage: ieq.prepare_inner_loop([3,5])
sage: print_cache(ieq)
Cached inner loop: 3 * x_0 + 37 >= 0
Cached next-to-inner loop: 3 * x_0 + 7 * x_1 + 2 >= 0
```

```
>>> from sage.all import *
>>> from sage.geometry.integral_points import InequalityCollection, print_cache
>>> P = Polyhedron(ieqs=[(Integer(2),Integer(3),Integer(7))])
>>> ieq = InequalityCollection(P, [Integer(0),Integer(1)],␣
↪[Integer(0)]*Integer(2),[Integer(1)]*Integer(2)); ieq
The collection of inequalities
integer: (3, 7) x + 2 >= 0
>>> ieq.prepare_next_to_inner_loop([Integer(3),Integer(5)])
>>> ieq.prepare_inner_loop([Integer(3),Integer(5)])
>>> print_cache(ieq)
Cached inner loop: 3 * x_0 + 37 >= 0
Cached next-to-inner loop: 3 * x_0 + 7 * x_1 + 2 >= 0
```

sage.geometry.integral_points.**ray_matrix_normal_form**(*R*)

Compute the Smith normal form of the ray matrix for `parallelotope_points()`.

INPUT:

- `R` – $\mathbf{Z}$-matrix whose columns are the rays spanning the parallelotope

OUTPUT: a tuple containing `e`, `d`, and `VDinv`

EXAMPLES:

```
sage: from sage.geometry.integral_points import ray_matrix_normal_form
sage: R = column_matrix(ZZ,[3,3,3])
sage: ray_matrix_normal_form(R)
([3], 3, [1])
```

```
>>> from sage.all import *
>>> from sage.geometry.integral_points import ray_matrix_normal_form
>>> R = column_matrix(ZZ,[Integer(3),Integer(3),Integer(3)])
>>> ray_matrix_normal_form(R)
([3], 3, [1])
```

sage.geometry.integral_points.**rectangular_box_points**(*box_min*, *box_max*, *polyhedron=None*, *count_only=False*, *return_saturated=False*)

Return the integral points in the lattice bounding box that are also contained in the given polyhedron.

INPUT:

- `box_min` – list of integers; the minimal value for each coordinate of the rectangular bounding box
- `box_max` – list of integers; the maximal value for each coordinate of the rectangular bounding box

- `polyhedron` – a *Polyhedron_base*, a PPL `C_Polyhedron`, or `None` (default)
- `count_only` – boolean (default: `False`); whether to return only the total number of vertices, and not their coordinates. Enabling this option speeds up the enumeration. Cannot be combined with the `return_saturated` option.
- `return_saturated` – boolean (default: `False`); whether to also return which inequalities are saturated for each point of the polyhedron. Enabling this slows down the enumeration. Cannot be combined with the `count_only` option.

OUTPUT:

By default, this function returns a tuple containing the integral points of the rectangular box spanned by `box_min` and `box_max` and that lie inside the `polyhedron`. For sufficiently large bounding boxes, this are all integral points of the polyhedron.

If no polyhedron is specified, all integral points of the rectangular box are returned.

If `count_only` is specified, only the total number (an integer) of found lattice points is returned.

If `return_saturated` is enabled, then for each integral point a pair `(point, Hrep)` is returned where `point` is the point and `Hrep` is the set of indices of the H-representation objects that are saturated at the point.

ALGORITHM:

This function implements the naive algorithm towards counting integral points. Given min and max of vertex coordinates, it iterates over all points in the bounding box and checks whether they lie in the polyhedron. The following optimizations are implemented:

- Cython: Use machine integers and optimizing C/C++ compiler where possible, arbitrary precision integers where necessary. Bounds checking, no compile time limits.
- Unwind inner loop (and next-to-inner loop):

$$Ax \leq b \quad \Leftrightarrow \quad a_1 x_1 \;\leq\; b - \sum_{i=2}^{d} a_i x_i$$

  so we only have to evaluate $a_1 * x_1$ in the inner loop.
- Coordinates are permuted to make the longest box edge the inner loop. The inner loop is optimized to run very fast, so its best to do as much work as possible there.
- Continuously reorder inequalities and test the most restrictive inequalities first.
- Use convexity and only find first and last allowed point in the inner loop. The points in-between must be points of the polyhedron, too.

EXAMPLES:

```
sage: from sage.geometry.integral_points import rectangular_box_points
sage: rectangular_box_points([0,0,0],[1,2,3])
((0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 0, 3),
 (0, 1, 0), (0, 1, 1), (0, 1, 2), (0, 1, 3),
 (0, 2, 0), (0, 2, 1), (0, 2, 2), (0, 2, 3),
 (1, 0, 0), (1, 0, 1), (1, 0, 2), (1, 0, 3),
 (1, 1, 0), (1, 1, 1), (1, 1, 2), (1, 1, 3),
 (1, 2, 0), (1, 2, 1), (1, 2, 2), (1, 2, 3))

sage: from sage.geometry.integral_points import rectangular_box_points
sage: rectangular_box_points([0,0,0],[1,2,3], count_only=True)
24
```

(continues on next page)

(continued from previous page)

```
sage: cell24 = polytopes.twenty_four_cell()
sage: rectangular_box_points([-1]*4, [1]*4, cell24)
((-1, 0, 0, 0), (0, -1, 0, 0), (0, 0, -1, 0), (0, 0, 0, -1),
 (0, 0, 0, 0),
 (0, 0, 0, 1), (0, 0, 1, 0), (0, 1, 0, 0), (1, 0, 0, 0))
sage: d = 3
sage: dilated_cell24 = d*cell24
sage: len( rectangular_box_points([-d]*4, [d]*4, dilated_cell24) )
305

sage: d = 6
sage: dilated_cell24 = d*cell24
sage: len( rectangular_box_points([-d]*4, [d]*4, dilated_cell24) )
3625

sage: rectangular_box_points([-d]*4, [d]*4, dilated_cell24, count_only=True)
3625

sage: polytope = Polyhedron([(-4,-3,-2,-1),(3,1,1,1),(1,2,1,1),(1,1,3,0),(1,3,2,
↪4)])
sage: pts = rectangular_box_points([-4]*4, [4]*4, polytope); pts
((-4, -3, -2, -1), (-1, 0, 0, 1), (0, 1, 1, 1), (1, 1, 1, 1), (1, 1, 3, 0),
 (1, 2, 1, 1), (1, 2, 2, 2), (1, 3, 2, 4), (2, 1, 1, 1), (3, 1, 1, 1))
sage: all(polytope.contains(p) for p in pts)
True

sage: set(map(tuple,pts)) == \
....: set([(-4,-3,-2,-1),(3,1,1,1),(1,2,1,1),(1,1,3,0),(1,3,2,4),
....:      (0,1,1,1),(1,2,2,2),(-1,0,0,1),(1,1,1,1),(2,1,1,1)])   # computed with␣
↪PALP
True
```

```
>>> from sage.all import *
>>> from sage.geometry.integral_points import rectangular_box_points
>>> rectangular_box_points([Integer(0),Integer(0),Integer(0)],[Integer(1),
↪Integer(2),Integer(3)])
((0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 0, 3),
 (0, 1, 0), (0, 1, 1), (0, 1, 2), (0, 1, 3),
 (0, 2, 0), (0, 2, 1), (0, 2, 2), (0, 2, 3),
 (1, 0, 0), (1, 0, 1), (1, 0, 2), (1, 0, 3),
 (1, 1, 0), (1, 1, 1), (1, 1, 2), (1, 1, 3),
 (1, 2, 0), (1, 2, 1), (1, 2, 2), (1, 2, 3))

>>> from sage.geometry.integral_points import rectangular_box_points
>>> rectangular_box_points([Integer(0),Integer(0),Integer(0)],[Integer(1),
↪Integer(2),Integer(3)], count_only=True)
24

>>> cell24 = polytopes.twenty_four_cell()
>>> rectangular_box_points([-Integer(1)]*Integer(4), [Integer(1)]*Integer(4),␣
↪cell24)
```

(continues on next page)

(continued from previous page)

```
((-1, 0, 0, 0), (0, -1, 0, 0), (0, 0, -1, 0), (0, 0, 0, -1),
 (0, 0, 0, 0),
 (0, 0, 0, 1), (0, 0, 1, 0), (0, 1, 0, 0), (1, 0, 0, 0))
>>> d = Integer(3)
>>> dilated_cell24 = d*cell24
>>> len( rectangular_box_points([-d]*Integer(4), [d]*Integer(4), dilated_cell24) )
305

>>> d = Integer(6)
>>> dilated_cell24 = d*cell24
>>> len( rectangular_box_points([-d]*Integer(4), [d]*Integer(4), dilated_cell24) )
3625

>>> rectangular_box_points([-d]*Integer(4), [d]*Integer(4), dilated_cell24, count_
↪only=True)
3625

>>> polytope = Polyhedron([(-Integer(4),-Integer(3),-Integer(2),-Integer(1)),
↪(Integer(3),Integer(1),Integer(1),Integer(1)),(Integer(1),Integer(2),Integer(1),
↪Integer(1)),(Integer(1),Integer(1),Integer(3),Integer(0)),(Integer(1),
↪Integer(3),Integer(2),Integer(4))])
>>> pts = rectangular_box_points([-Integer(4)]*Integer(4),␣
↪[Integer(4)]*Integer(4), polytope); pts
((-4, -3, -2, -1), (-1, 0, 0, 1), (0, 1, 1, 1), (1, 1, 1, 1), (1, 1, 3, 0),
 (1, 2, 1, 1), (1, 2, 2, 2), (1, 3, 2, 4), (2, 1, 1, 1), (3, 1, 1, 1))
>>> all(polytope.contains(p) for p in pts)
True

>>> set(map(tuple,pts)) == set([(-Integer(4),-Integer(3),-Integer(2),-Integer(1)),
↪(Integer(3),Integer(1),Integer(1),Integer(1)),(Integer(1),Integer(2),Integer(1),
↪Integer(1)),(Integer(1),Integer(1),Integer(3),Integer(0)),(Integer(1),
↪Integer(3),Integer(2),Integer(4)),
...       (Integer(0),Integer(1),Integer(1),Integer(1)),(Integer(1),Integer(2),
↪Integer(2),Integer(2)),(-Integer(1),Integer(0),Integer(0),Integer(1)),
↪(Integer(1),Integer(1),Integer(1),Integer(1)),(Integer(2),Integer(1),Integer(1),
↪Integer(1))])    # computed with PALP
True
```

Long ints and non-integral polyhedra are explicitly allowed:

```
sage: polytope = Polyhedron([[1], [10*pi.n()]], base_ring=RDF)                    #␣
↪needs sage.symbolic
sage: len(rectangular_box_points([-100], [100], polytope))                        #␣
↪needs sage.symbolic
31

sage: halfplane = Polyhedron(ieqs=[(-1,1,0)])
sage: rectangular_box_points([0,-1+10^50], [0,1+10^50])
((0, 99999999999999999999999999999999999999999999999999),
 (0, 100000000000000000000000000000000000000000000000000),
 (0, 100000000000000000000000000000000000000000000000001))
sage: len( rectangular_box_points([0,-100+10^50], [1,100+10^50], halfplane) )
```

(continues on next page)

(continued from previous page)

```
201
```

```
>>> from sage.all import *
>>> polytope = Polyhedron([[Integer(1)], [Integer(10)*pi.n()]], base_ring=RDF)   ␣
↪                # needs sage.symbolic
>>> len(rectangular_box_points([-Integer(100)], [Integer(100)], polytope))       ␣
↪                # needs sage.symbolic
31

>>> halfplane = Polyhedron(ieqs=[(-Integer(1),Integer(1),Integer(0))])
>>> rectangular_box_points([Integer(0),-Integer(1)+Integer(10)**Integer(50)],␣
↪[Integer(0),Integer(1)+Integer(10)**Integer(50)])
((0, 99999999999999999999999999999999999999999999999999),
 (0, 100000000000000000000000000000000000000000000000000),
 (0, 100000000000000000000000000000000000000000000000001))
>>> len( rectangular_box_points([Integer(0),-
↪Integer(100)+Integer(10)**Integer(50)], [Integer(1),
↪Integer(100)+Integer(10)**Integer(50)], halfplane) )
201
```

Using a PPL polyhedron:

```
sage: # needs pplpy
sage: from ppl import Variable, Generator_System, C_Polyhedron, point
sage: gs = Generator_System()
sage: x = Variable(0); y = Variable(1); z = Variable(2)
sage: gs.insert(point(0*x + 1*y + 0*z))
sage: gs.insert(point(0*x + 1*y + 3*z))
sage: gs.insert(point(3*x + 1*y + 0*z))
sage: gs.insert(point(3*x + 1*y + 3*z))
sage: poly = C_Polyhedron(gs)
sage: rectangular_box_points([0]*3, [3]*3, poly)
((0, 1, 0), (0, 1, 1), (0, 1, 2), (0, 1, 3), (1, 1, 0), (1, 1, 1), (1, 1, 2), (1,␣
↪1, 3),
 (2, 1, 0), (2, 1, 1), (2, 1, 2), (2, 1, 3), (3, 1, 0), (3, 1, 1), (3, 1, 2), (3,␣
↪1, 3))
```

```
>>> from sage.all import *
>>> # needs pplpy
>>> from ppl import Variable, Generator_System, C_Polyhedron, point
>>> gs = Generator_System()
>>> x = Variable(Integer(0)); y = Variable(Integer(1)); z = Variable(Integer(2))
>>> gs.insert(point(Integer(0)*x + Integer(1)*y + Integer(0)*z))
>>> gs.insert(point(Integer(0)*x + Integer(1)*y + Integer(3)*z))
>>> gs.insert(point(Integer(3)*x + Integer(1)*y + Integer(0)*z))
>>> gs.insert(point(Integer(3)*x + Integer(1)*y + Integer(3)*z))
>>> poly = C_Polyhedron(gs)
>>> rectangular_box_points([Integer(0)]*Integer(3), [Integer(3)]*Integer(3), poly)
((0, 1, 0), (0, 1, 1), (0, 1, 2), (0, 1, 3), (1, 1, 0), (1, 1, 1), (1, 1, 2), (1,␣
↪1, 3),
 (2, 1, 0), (2, 1, 1), (2, 1, 2), (2, 1, 3), (3, 1, 0), (3, 1, 1), (3, 1, 2), (3,␣
↪1, 3))
```

Optionally, return the information about the saturated inequalities as well:

```
sage: cube = polytopes.cube()
sage: cube.Hrepresentation(0)
An inequality (-1, 0, 0) x + 1 >= 0
sage: cube.Hrepresentation(1)
An inequality (0, -1, 0) x + 1 >= 0
sage: cube.Hrepresentation(2)
An inequality (0, 0, -1) x + 1 >= 0
sage: rectangular_box_points([0]*3, [1]*3, cube, return_saturated=True)
(((0, 0, 0), frozenset()),
 ((0, 0, 1), frozenset({2})),
 ((0, 1, 0), frozenset({1})),
 ((0, 1, 1), frozenset({1, 2})),
 ((1, 0, 0), frozenset({0})),
 ((1, 0, 1), frozenset({0, 2})),
 ((1, 1, 0), frozenset({0, 1})),
 ((1, 1, 1), frozenset({0, 1, 2})))
```

```
>>> from sage.all import *
>>> cube = polytopes.cube()
>>> cube.Hrepresentation(Integer(0))
An inequality (-1, 0, 0) x + 1 >= 0
>>> cube.Hrepresentation(Integer(1))
An inequality (0, -1, 0) x + 1 >= 0
>>> cube.Hrepresentation(Integer(2))
An inequality (0, 0, -1) x + 1 >= 0
>>> rectangular_box_points([Integer(0)]*Integer(3), [Integer(1)]*Integer(3), cube,
↪ return_saturated=True)
(((0, 0, 0), frozenset()),
 ((0, 0, 1), frozenset({2})),
 ((0, 1, 0), frozenset({1})),
 ((0, 1, 1), frozenset({1, 2})),
 ((1, 0, 0), frozenset({0})),
 ((1, 0, 1), frozenset({0, 2})),
 ((1, 1, 0), frozenset({0, 1})),
 ((1, 1, 1), frozenset({0, 1, 2})))
```

sage.geometry.integral_points.**simplex_points**(*vertices*)

Return the integral points in a lattice simplex.

INPUT:

- `vertices` – an iterable of integer coordinate vectors. The indices of vertices that span the simplex under consideration.

OUTPUT:

A tuple containing the integral point coordinates as $\mathbf{Z}$-vectors.

EXAMPLES:

```
sage: from sage.geometry.integral_points import simplex_points
sage: simplex_points([(1,2,3), (2,3,7), (-2,-3,-11)])
((-2, -3, -11), (0, 0, -2), (1, 2, 3), (2, 3, 7))
```

```
>>> from sage.all import *
>>> from sage.geometry.integral_points import simplex_points
>>> simplex_points([(Integer(1),Integer(2),Integer(3)), (Integer(2),Integer(3),
↪Integer(7)), (-Integer(2),-Integer(3),-Integer(11))])
((-2, -3, -11), (0, 0, -2), (1, 2, 3), (2, 3, 7))
```

The simplex need not be full-dimensional:

```
sage: simplex = Polyhedron([(1,2,3,5), (2,3,7,5), (-2,-3,-11,5)])
sage: simplex_points(simplex.Vrepresentation())
((2, 3, 7, 5), (0, 0, -2, 5), (-2, -3, -11, 5), (1, 2, 3, 5))

sage: simplex_points([(2,3,7)])
((2, 3, 7),)
```

```
>>> from sage.all import *
>>> simplex = Polyhedron([(Integer(1),Integer(2),Integer(3),Integer(5)),␣
↪(Integer(2),Integer(3),Integer(7),Integer(5)), (-Integer(2),-Integer(3),-
↪Integer(11),Integer(5))])
>>> simplex_points(simplex.Vrepresentation())
((2, 3, 7, 5), (0, 0, -2, 5), (-2, -3, -11, 5), (1, 2, 3, 5))

>>> simplex_points([(Integer(2),Integer(3),Integer(7))])
((2, 3, 7),)
```

5.4 Helper Functions For Freeness Of Hyperplane Arrangements

This contains the algorithms to check for freeness of a hyperplane arrangement. See `sage.geometry.hyperplane_arrangement.HyperplaneArrangementElement.is_free()` for details.

> **Note**
>
> This could be extended to a freeness check for more general modules over a polynomial ring.

`sage.geometry.hyperplane_arrangement.check_freeness.`**`construct_free_chain`**`(A)`

Construct the free chain for the hyperplanes `A`.

ALGORITHM:

We follow Algorithm 6.5 in [BC2012].

INPUT:

- `A` – a hyperplane arrangement

EXAMPLES:

```
sage: from sage.geometry.hyperplane_arrangement.check_freeness import construct_
↪free_chain
sage: H.<x,y,z> = HyperplaneArrangements(QQ)
sage: A = H(z, y+z, x+y+z)
sage: construct_free_chain(A)
[
```

(continues on next page)

(continued from previous page)

```
[1 0 0]  [ 1  0  0]  [     0      1      0]
[0 1 0]  [ 0  z -1]  [y + z      0     -1]
[0 0 z], [ 0  y  1], [     x      0      1]
]
```

```
>>> from sage.all import *
>>> from sage.geometry.hyperplane_arrangement.check_freeness import construct_
↪free_chain
>>> H = HyperplaneArrangements(QQ, names=('x', 'y', 'z',)); (x, y, z,) = H._first_
↪ngens(3)
>>> A = H(z, y+z, x+y+z)
>>> construct_free_chain(A)
[
[1 0 0]  [ 1  0  0]  [     0      1      0]
[0 1 0]  [ 0  z -1]  [y + z      0     -1]
[0 0 z], [ 0  y  1], [     x      0      1]
]
```

`sage.geometry.hyperplane_arrangement.check_freeness.`**`less_generators`**`(X)`

Reduce the generator matrix of the module defined by `X`.

This is Algorithm 6.4 in [BC2012] and relies on the row syzygies of the matrix `X`.

EXAMPLES:

```
sage: from sage.geometry.hyperplane_arrangement.check_freeness import less_
↪generators
sage: R.<x,y,z> = QQ[]
sage: m = matrix([[1, 0, 0], [0, z, -1], [0, 0, 0], [0, y, 1]])
sage: less_generators(m)
[ 1  0  0]
[ 0  z -1]
[ 0  y  1]
```

```
>>> from sage.all import *
>>> from sage.geometry.hyperplane_arrangement.check_freeness import less_
↪generators
>>> R = QQ['x, y, z']; (x, y, z,) = R._first_ngens(3)
>>> m = matrix([[Integer(1), Integer(0), Integer(0)], [Integer(0), z, -
↪Integer(1)], [Integer(0), Integer(0), Integer(0)], [Integer(0), y, Integer(1)]])
>>> less_generators(m)
[ 1  0  0]
[ 0  z -1]
[ 0  y  1]
```

CHAPTER SIX

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

g

r

INDEX

A

B

D

E

F

G

H

I

M

N

O

P

Q

R

S

T

U

V

W

Z