CONTENTS

1 Hyperplane arrangements 3
2 Polyhedral computations 53
3 Triangulations 431
4 Miscellaneous 465
5 Helper functions 495
6 Indices and Tables 511
Bibliography 513
Python Module Index 515
Index 517
Sage includes classes for hyperplane arrangements, polyhedra, toric varieties (including polyhedral cones and fans), triangulations and some other helper classes and functions.
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 $x + 3y - 5z = 7$. 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.normal()
(3, 2, -5)
sage: 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)
```

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
```

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

The default base field is $\mathbb{Q}$, 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>

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>

Notation (iv): using the library of arrangements:

sage: hyperplane_arrangements.braid(4)
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())
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

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

New arrangements from old:

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

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)

(continues on next page)
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
```

A hyperplane arrangement is *essential* if the normals to its hyperplane 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: a = hyperplane_arrangements.braid(4); a
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
```

The connected components of the complement of the hyperplanes of an arrangement in $\mathbb{R}^n$ 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 $\mathbb{Q}^2$ defined as the convex hull of 6 vertices,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 3 vertices,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 3 vertices,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 3 vertices and 1→ray,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 3 vertices,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 3 vertices and 1→ray,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 3 vertices and 1→ray,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 3 vertices and 1→ray,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 3 vertices and 1→ray,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 3 vertices and 1→ray,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 3 vertices,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 3 vertices and 1→ray,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 3 vertices and 1→ray,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 3 vertices,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 3 vertices and 1→ray,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 3 vertices and 1→ray,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 3 vertices and 1→ray,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 3 vertices and 1→ray,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 3 vertices and 1→ray,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 3 vertices and 1→ray,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 3 vertices and 1→ray,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 3 vertices and 1→ray,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 3 vertices and 1→ray,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 3 vertices and 1→ray,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 3 vertices and 1→ray,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 3 vertices and 1→ray,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 3 vertices and 1→ray,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 3 vertices and 1→ray,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 3 vertices and 1→ray,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 1 vertex and 2 rays,
 A 2-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 3 vertices and 1→ray,)``

(continues on next page)
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 1 vertex and 2 rays)

\begin{verbatim}
sage: b.bounded_regions()
(A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 6 vertices,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 3 vertices,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 3 vertices,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 3 vertices,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 3 vertices,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 3 vertices)
sage: b.n_bounded_regions()
7
sage: a.unbounded_regions()
(A 3-dimensional polyhedron in \( \mathbb{Q}^3 \) defined as the convex hull of 1 vertex, 2 rays, 1
→ line,
A 3-dimensional polyhedron in \( \mathbb{Q}^3 \) defined as the convex hull of 3 vertices, 1 ray,
→ 1 line,
A 3-dimensional polyhedron in \( \mathbb{Q}^3 \) defined as the convex hull of 1 vertex, 2 rays, 1
→ line,
A 3-dimensional polyhedron in \( \mathbb{Q}^3 \) defined as the convex hull of 3 vertices, 1 ray,
→ 1 line,
A 3-dimensional polyhedron in \( \mathbb{Q}^3 \) defined as the convex hull of 1 vertex, 2 rays, 1
→ line,
A 3-dimensional polyhedron in \( \mathbb{Q}^3 \) defined as the convex hull of 3 vertices, 1 ray,
→ 1 line,
A 3-dimensional polyhedron in \( \mathbb{Q}^3 \) defined as the convex hull of 1 vertex, 2 rays, 1
→ line,
A 3-dimensional polyhedron in \( \mathbb{Q}^3 \) defined as the convex hull of 3 vertices, 1 ray,
→ 1 line,
A 3-dimensional polyhedron in \( \mathbb{Q}^3 \) defined as the convex hull of 1 vertex, 2 rays, 1
→ line,
A 3-dimensional polyhedron in \( \mathbb{Q}^3 \) defined as the convex hull of 3 vertices, 1 ray,
→ 1 line,
A 3-dimensional polyhedron in \( \mathbb{Q}^3 \) defined as the convex hull of 1 vertex, 2 rays, 1
→ line,
A 3-dimensional polyhedron in \( \mathbb{Q}^3 \) defined as the convex hull of 3 vertices, 1 ray,
→ 1 line,
A 3-dimensional polyhedron in \( \mathbb{Q}^3 \) defined as the convex hull of 1 vertex, 2 rays, 1
→ line)
\end{verbatim}

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

\begin{verbatim}
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
\end{verbatim}

\textbf{Note:} \textit{bounded region} really mean \textit{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):
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 the sum is \( P \) is the intersection poset() of the arrangement and \( \mu \) is the Möbius function of \( P \):

```
sage: a = hyperplane_arrangements.semiorder(5)
sage: a.characteristic_polynomial()                   # long time (about a second on Core_i7)
x^5 - 20*x^4 + 180*x^3 - 790*x^2 + 1380*x
sage: a.poincare_polynomial()                        # long time
1380*x^4 + 790*x^3 + 180*x^2 + 20*x + 1
sage: a.n_regions()                                  # long time
2371
sage: charpoly = a.characteristic_polynomial()      # long time
sage: charpoly(-1)                                   # long time
-2371
sage: a.n_bounded_regions()                         # long time
751
sage: charpoly(1)                                   # long time
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)                          True
sage: b = a.change_ring(GF(5))                         
sage: pa = a.intersection_poset()                     
sage: pb = b.intersection_poset()                      
sage: pa.is_isomorphic(pb)                            True
sage: a.face_vector()                                  (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()                            
[ 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]
```

(continues on next page)
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)

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.

```python
class sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement:
    # Implementation details...

Bases: sage.structure.element.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:

```python
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: A.union(B)  # Arrangement of 8 hyperplanes of dimension 2 and rank 2
Arrangement of 8 hyperplanes of dimension 2 and rank 2
```

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

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

```python
sage: A.add_hyperplane(x+y-1)  # alias
```

(continues on next page)
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:

\begin{verbatim}
sage: A = hyperplane_arrangements.semiorder(3)
sage: A.bounded_regions()
(A 3-dimensional polyhedron in \(\mathbb{Q}^3\) defined as the convex hull of 3 vertices and 1 line,
A 3-dimensional polyhedron in \(\mathbb{Q}^3\) defined as the convex hull of 3 vertices and 1 line,
A 3-dimensional polyhedron in \(\mathbb{Q}^3\) defined as the convex hull of 3 vertices and 1 line,
A 3-dimensional polyhedron in \(\mathbb{Q}^3\) defined as the convex hull of 6 vertices and 1 line,
A 3-dimensional polyhedron in \(\mathbb{Q}^3\) defined as the convex hull of 3 vertices and 1 line,
A 3-dimensional polyhedron in \(\mathbb{Q}^3\) defined as the convex hull of 3 vertices and 1 line,
A 3-dimensional polyhedron in \(\mathbb{Q}^3\) 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
\end{verbatim}

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>
```

### characteristic_polynomial()\

Returns the characteristic polynomial of the hyperplane arrangement.

**OUTPUT:**

The characteristic polynomial in \(\mathbb{Q}[x]\).

**EXAMPLES:**

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

### closed_faces() (labelled=True)

Returns the closed faces of the hyperplane arrangement \(\mathcal{A}\) (provided that \(\mathcal{A}\) 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 \((v_1, v_2, \ldots, v_N)\) 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 \(\text{self.hyperplanes()}\)).

**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: 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,
   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,
   1 ray,
   (1, -1)),
   (0, -1),
   A 1-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex,
   1 ray,
   (-1, -1)),
   (1, 0),
   A 1-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex,
   1 ray,
   (0, 0)),
   (1, 1),
   A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex,
   and 2 rays,
A 2-dimensional polyhedron in \(\mathbb{Q}^2\) defined as the convex hull of 1 vertex and 2 rays,
\((-1, 0)),
\((-1, 0)),
A 1-dimensional polyhedron in \(\mathbb{Q}^2\) defined as the convex hull of 1 vertex and 1 ray,
\((0, -2)),
\((-1, 1)),
A 2-dimensional polyhedron in \(\mathbb{Q}^2\) defined as the convex hull of 1 vertex and 2 rays,
\((1, -2)),
\((-1, -1)),
A 2-dimensional polyhedron in \(\mathbb{Q}^2\) defined as the convex hull of 1 vertex and 2 rays,
\((-1, -2))

\sage{a = hyperplane_arrangements.braid(3)}
\sage{a.hyperplanes()}
\sage{[(v, F, F.representative_point()) for v, F in a.closed_faces()]

(continues on next page)
A 2-dimensional polyhedron in \(\mathbb{Q}^3\) defined as the convex hull of 1 vertex,\(\rightarrow\)1 ray, 1 line,
\((0, 0, 1)),
\((-1, 1, 0),\)
A 2-dimensional polyhedron in \(\mathbb{Q}^3\) defined as the convex hull of 1 vertex,\(\rightarrow\)1 ray, 1 line,
\((1, 0, 1)),
\((-1, 1, 1),\)
A 3-dimensional polyhedron in \(\mathbb{Q}^3\) defined as the convex hull of 1 vertex,\(\rightarrow\)2 rays, 1 line,
\((0, -2, -1)),
\((-1, 1, -1),\)
A 3-dimensional polyhedron in \(\mathbb{Q}^3\) defined as the convex hull of 1 vertex,\(\rightarrow\)2 rays, 1 line,
\((1, 0, 2)),
\((-1, -1, -1),\)
A 3-dimensional polyhedron in \(\mathbb{Q}^3\) defined as the convex hull of 1 vertex,\(\rightarrow\)2 rays, 1 line,
\((-1, 0, 1))\]

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

```python
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)
True
sage: a = hyperplane_arrangements.Shi(3)
sage: test_number(a)  # long time
True
```

cone (\texttt{variable}=\texttt{r})

Return the cone over the hyperplane arrangement.

INPUT:

• \texttt{variable} – string; the name of the additional variable

OUTPUT:

A new hyperplane arrangement. 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 \([0, 1, 0, \ldots, 0]\).

**Warning:** While there is an almost-one-to-one correspondence between the hyperplanes of \(\text{self}\) and those of \(\text{self}.\)\(\text{cone}()\), there is no guarantee that the order in which they appear in \(\text{self}.\)\(\text{hyperplanes}()\) will match the order in which their counterparts in \(\text{self}.\)\(\text{cone}()\) will appear in \(\text{self}.\)\(\text{cone}().\)\(\text{hyperplanes}()\)!

**EXAMPLES:**

```python
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,
 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)

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
-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)
```

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)
```

(continues on previous page)
Arrangement \langle y - 1 | y + 1 | x - y | x + y \rangle

sage: h = H([0,1,0], [0,1,1])
sage: A.deletion(h)
Arrangement \langle y - 1 | y + 1 | x - y \rangle

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: W = WeylGroup(['A',2], prefix='s')
sage: A = W.long_element().inversion_arrangement()
sage: A.derivation_module_free_chain()
[(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: W = WeylGroup(['A',3], prefix='s')
sage: A = W.long_element().inversion_arrangement()
sage: for M in A.derivation_module_free_chain():
    print("%s
" % M)
[ 1 0 0]

### dimension()

Return the ambient space dimension of the arrangement.

**OUTPUT:**

An integer.

**EXAMPLES:**

```sage
erg = HyperplaneArrangements(QQ)
g.dimension()  # dimension of the arrangement
```

### 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:**

An integer. The number of hyperplanes separating the two regions.

**EXAMPLES:**

```sage
c = hyperplane_arrangements.coordinate(2)
r = c.region_containing_point([-1, -1])
s = c.region_containing_point([1, 1])
c.distance_between_regions(r, s)  # number of hyperplanes separating r and s
```

---

The matrix representation of the hyperplane arrangements and the region dimensions are demonstrated, alongside the computational examples using SageMath, which highlight the use of the `dimension()` and `distance_between_regions()` functions.
**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)
c.distance_enumerator(c.region_containing_point([1,1,1]))
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 M"obius function values $\mu(x, y)$ over all $x, y$ in the intersection poset with rank($x$) = $i$ and 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
A = hyperplane_arrangements.Shi(3)
A.doubly_indexed_whitney_number(0, 2)
9
A.whitney_number(2)
9
A.doubly_indexed_whitney_number(1, 2)
-15
```

**REFERENCES:**

- [GZ1983]
\begin{verbatim}
sage: a = hyperplane_arrangements.braid(3)
sage: a.is_essential()
False
sage: a.essentialization()
Arrangement <t1 - t2 | t1 + 2\times t2 | 2\times 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: 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
\end{verbatim}

\textbf{face_product} (\(F, G, \text{normalize}\text{=}True\))

Return the product \(FG\) in the face semigroup of \(\text{self}\), where \(F\) and \(G\) are two closed faces of \(\text{self}\).

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 \(\text{self}\) (see \texttt{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 \(\varepsilon\).

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 \(x\cdot y\cdot x = x\cdot y\)). 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 \(\text{self}\) (as polyhedra)
- \text{normalize} – Boolean (default: True); if True, then this method returns the precise instance of \(FG\) in the list returned by \texttt{self.closed_faces()}, rather than creating a new instance

EXAMPLES:

\begin{verbatim}
sage: a = hyperplane_arrangements.braid(3)
sage: a.hyperplanes()
(Hyperplane 0\times t0 + t1 - t2 + 0,
 Hyperplane t0 - t1 + 0\times t2 + 0,
 Hyperplane t0 + 0\times t1 - t2 + 0)
sage: faces = {F0: F1 for F0, F1 in a.closed_faces()}
sage: xGyEz = faces[(0, 1, 1)] # closed face x \geq y = z
\end{verbatim}
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

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: \(\mathbb{Q}\)), 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: 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
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
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
sage: A = hyperplane_arrangements.Shi(3)
sage: A.face_vector()
(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:
A boolean.

EXAMPLES:
```
sage: a = hyperplane_arrangements.semiorder(3)
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
```

**hyperplanes()**

Return the number of hyperplanes in the arrangement.

**OUTPUT:**

An integer.

**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)
```

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

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

**intersection_poset()**

Return the intersection poset of the hyperplane arrangement.

**OUTPUT:**

The poset of non-empty intersections of hyperplanes.

**EXAMPLES:**

```
sage: a = hyperplane_arrangements.coordinate(2)
sage: a.intersection_poset()
Finite poset containing 4 elements
sage: A = hyperplane_arrangements.semiorder(3)
sage: A.intersection_poset()
Finite poset containing 19 elements
```

**is_central()**

Test whether the intersection of all the hyperplanes is nonempty.

**OUTPUT:**

A boolean whether the hyperplane arrangement is such that the intersection of all the hyperplanes in the arrangement is nonempty.

**EXAMPLES:**

```
sage: a = hyperplane_arrangements.braid(2)
sage: a.is_central()
True
```
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:  
A boolean indicating whether the hyperplane arrangement is essential.  

EXAMPLES:  

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

```python  
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, 3*x+4*y, 3*x+5*z, 5*x+4*y, 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  
```

is_free(algorithm='singular')  
Return if self is free.  
A hyperplane arrangement $A$ is free if the module of derivations $\text{Der}(A)$ 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:  

sage  
Check that the minimal free resolution has length at most 2 by using Singular.
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 \( A_3 \):

```python
sage: W = WeylGroup(['A',3], prefix='s')
sage: for x in W:
    ...:     A = x.inversion_arrangement()
    ...:     assert A.matroid().is_chordal() == A.is_free()
```

**is_linear()**

Test whether all hyperplanes pass through the origin.

**OUTPUT:**

A boolean. Whether all the hyperplanes pass through the origin.

**EXAMPLES:**

```python
sage: a = hyperplane_arrangements.semiorder(3)
sage: a.is_linear()
False
sage: b = hyperplane_arrangements.braid(3)
sage: b.is_linear()
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:**

A boolean. Whether the hyperplane hyperplane separate the given regions.

**EXAMPLES:**

```python
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)
```

(continues on next page)
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:

```python
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: 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:

```python
sage: f = sum([list(M.flats(i)) for i in range(M.rank()+1)], [])
sage: PF = Poset([f, lambda x,y: x < y])
sage: PF.is_isomorphic(A.intersection_poset())
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 \( \dim W_k = \text{rank } A \).

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]:

```python
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
```

n_bounder_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

**n_hyperplanes()**
Return the number of hyperplanes in the arrangement.

OUTPUT:
An integer.

EXAMPLES:

```plaintext
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
```

**n_regions()**
The number of regions of the hyperplane arrangement.

OUTPUT:
An integer.

EXAMPLES:

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

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

```plaintext
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
```

**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()
Graphics object consisting of 3 graphics primitives

poincare_polynomial()

Return the Poincaré polynomial of the hyperplane arrangement.

OUTPUT:
The Poincaré polynomial in \( Q[x] \).

EXAMPLES:

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

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:
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 \(\mathbb{R}^n\).

EXAMPLES:

```
sage: a = hyperplane_arrangements.braid(2)
sage: a.regions()
(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
```

restriction(hyperplane)

Return the restriction to a hyperplane.

INPUT:

- hyperplane – a hyperplane of the hyperplane arrangement

OUTPUT:

The restriction of the hyperplane arrangement to the given hyperplane.

EXAMPLES:
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: 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

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)

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: 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 \( \mathbb{Q}^2 \) defined as the convex hull of 3 vertices and 1 ray,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 3 vertices and 1 ray,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 1 vertex and 2 rays,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 3 vertices and 1 ray,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 1 vertex and 2 rays,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 3 vertices and 1 ray,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 1 vertex and 2 rays,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 3 vertices and 1 ray,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 1 vertex and 2 rays,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 3 vertices and 1 ray,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 1 vertex and 2 rays,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 3 vertices and 1 ray,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 1 vertex and 2 rays,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 3 vertices and 1 ray,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 1 vertex and 2 rays,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 3 vertices and 1 ray,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 1 vertex and 2 rays,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 3 vertices and 1 ray,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 1 vertex and 2 rays,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 3 vertices and 1 ray,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 1 vertex and 2 rays,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 3 vertices and 1 ray,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 1 vertex and 2 rays,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 3 vertices and 1 ray,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 1 vertex and 2 rays,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 3 vertices and 1 ray,
A 2-dimensional polyhedron in \( \mathbb{Q}^2 \) 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: A.union(B)
Arrangement of 8 hyperplanes of dimension 2 and rank 2
```

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
```
varchenko_matrix(names='h')
Return the Varchenko matrix of the arrangement.

Let $H_1, \ldots, H_s$ and $R_1, \ldots, R_t$ denote the hyperplanes and regions, respectively, of the arrangement. Let $S = \mathbb{Q}[h_1, \ldots, h_s]$, 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:

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

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:

```python
cube = hyperplane_arrangements.Cube(3)
sage: v = cube.vertices(); v
((0, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 0))
```

```python
cube.vertices(ambient=QQ)
sage: v = cube.vertices(ambient=QQ); v
((0, 0, 0), (0, 1, 1), (1, 0, 1), (1, 1, 0))
```
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:

```python
sage: A = hyperplane_arrangements.Shi(3)
sage: A.whitney_data()
\[
\begin{pmatrix}
  1 & -6 & 9 \\
  0 & 6 & -15 \\
  0 & 0 & 6
\end{pmatrix},
\begin{pmatrix}
  1 & 6 & 6 \\
  0 & 6 & 15 \\
  0 & 0 & 6
\end{pmatrix}
\]
```

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 \(rank(x) = k\).

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:

```python
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)
```

(continues on next page)
sage: p = A.intersection_poset()
sage: r = p.rank_function()
sage: len([i for i in p if r(i) == 1])
6

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

Bases: sage.structure.parent.Parent, sage.structure.unique_representation.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:

```python
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>
```

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:

```python
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
```

base_ring()

Return the base ring.

OUTPUT:

The base ring of the hyperplane arrangement.

EXAMPLES:

```python
sage: L.<x,y> = HyperplaneArrangements(QQ)
sage: 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
```

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
```

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)
```

ngens()

Return the number of linear variables.

OUTPUT:

An integer.

EXAMPLES:
1.2 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.

```python
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
```

**Catalan** \((n, K=\text{Rational Field}, \text{names}=\text{None})\)

Return the Catalan arrangement.

**INPUT:**
- \(n\) – integer
- \(K\) – field (default: \(\mathbb{Q}\))
- \(\text{names}\) – tuple of strings or \(\text{None}\) (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:**

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

**G_Shi** \((G, K=\text{Rational Field}, \text{names}=\text{None})\)

Return the Shi hyperplane arrangement of a graph \(G\).

**INPUT:**
- \(G\) – graph
- \(K\) – field (default: \(\mathbb{Q}\))
- \(\text{names}\) – tuple of strings or \(\text{None}\) (default); the variable names for the ambient space

**OUTPUT:**
The Shi hyperplane arrangement of the given graph \(G\).

**EXAMPLES:**

```python
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
```

(continues on next page)
sage: a = hyperplane_arrangements.G_Shi(graphs.WheelGraph(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
sage: G = graphs.CompleteGraph(5)
sage: hyperplane_arrangements.G_semiorder(G)
Arrangement of 20 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
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
sage: b = hyperplane_arrangements.Shi(3)
sage: b.characteristic_polynomial()
x^3 - 6*x^2 + 9*x
```

**REFERENCES:**

- [AR2012]

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

Return the Shi arrangement.

**INPUT:**
The Shi arrangement is the set of \( n(n-1) \) hyperplanes: \( \{x_i - x_j = 0 \mid 1 \leq i \leq j \leq n \} \).

**EXAMPLES:**

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

### bigraphical (\( G, A=None, K=\text{Rational Field}, \text{names}=\text{None} \))

Return a bigraphical hyperplane arrangement.

**INPUT:**

- \( G \) – graph
- \( A \) – list, matrix, dictionary (default: None gives semiorder), or the string ‘generic’
- \( K \) – field (default: \( \mathbb{Q} \))
- \( \text{names} \) – 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: G = graphs.CycleGraph(4)
sage: G.edges()
[(0, 1, None), (0, 3, None), (1, 2, None), (2, 3, None)]
sage: G.edges(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
```

**REFERENCES:**

- [HP2016]
The hyperplane arrangement consisting of the \( n(n - 1)/2 \) hyperplanes \( \{x_i - x_j = 0 : 1 \leq i \leq j \leq n\} \).

**EXAMPLES:**

```python
sage: hyperplane_arrangements.braid(4)
Arrangement of 6 hyperplanes of dimension 4 and rank 3
```

**coordinate** \((n, K=\text{Rational Field}, \text{names}=\text{None})\)

Return the coordinate hyperplane arrangement.

**INPUT:**

- \( n \) – integer
- \( K \) – field (default: \( \mathbb{Q} \))
- \( \text{names} \) – tuple of strings or \( \text{None} \) (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:**

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

**graphical** \((G, K=\text{Rational Field}, \text{names}=\text{None})\)

Return the graphical hyperplane arrangement of a graph \( G \).

**INPUT:**

- \( G \) – graph
- \( K \) – field (default: \( \mathbb{Q} \))
- \( \text{names} \) – tuple of strings or \( \text{None} \) (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:**

```python
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
```

**linial** \((n, K=\text{Rational Field}, \text{names}=\text{None})\)

Return the linial hyperplane arrangement.

**INPUT:**

- \( n \) – integer
- \( K \) – field (default: \( \mathbb{Q} \))
- \( \text{names} \) – tuple of strings or \( \text{None} \) (default); the variable names for the ambient space
The linial hyperplane arrangement is the set of hyperplanes \( \{ x_i - x_j = 1 : 1 \leq i < j \leq n \} \).

**EXAMPLES:**

```python
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
```

**semiorder** \((n, \mathbb{K} = \text{Rational Field}, \text{names=\text{None}})\)

Return the semiorder arrangement.

**INPUT:**

- \(n\) – integer
- \(\mathbb{K}\) – field (default: \(\mathbb{Q}\))
- \(\text{names}\) – 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 \(\{ x_i - x_j = -1, 1 \leq i < j \leq n\} \).

**EXAMPLES:**

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

**sage.geometry.hyperplane_arrangement.library.make_parent** \((\text{base}_\text{ring}, \text{dimension}, \text{names=\text{None}})\)

Construct the parent for the hyperplane arrangements.

For internal use only.

**INPUT:**

- \(\text{base}_\text{ring}\) – a ring
- \(\text{dimension}\) – integer
- \(\text{names}\) – None (default) or a list/tuple/iterable of strings

**OUTPUT:**

A new \text{HyperplaneArrangements} instance.

**EXAMPLES:**

```python
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
```

### 1.3 Hyperplanes
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]
```

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([-7, 3, 2, -5], -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
```

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,)
```

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
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

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'

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

  _bases: sage.geometry.linear_expression.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

dimension()
   Return the ambient space dimension.

   OUTPUT:
   An integer.
EXAMPLES:

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

```python
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
```

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

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:

```python
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
```

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

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

dimension()

The dimension of the hyperplane.
OUTPUT:
An integer.

EXAMPLES:

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

```python
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
```

**linear_part**( )
The linear part of the affine space.

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

EXAMPLES:

```python
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]
```

**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
(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)

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)

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)
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

plot(**kwargs)

Plot the hyperplane.

OUTPUT:

A graphics object.

EXAMPLES:

sage: L.<x, y> = HyperplaneArrangements(QQ)
sage: (x+y-2).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: 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)

polyhedron()

Return the hyperplane as a polyhedron.

OUTPUT:

A `Polyhedron()` instance.

EXAMPLES:
**primitive** *(signed=True)*

Return hyperplane defined by primitive equation.

**INPUT:**

- **signed** – boolean (optional, 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:**

```python
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()  # default
False
sage: (4*x + 8).primitive()  # default
Hyperplane x + 0*y + 2
sage: (4*x - y - 8).primitive(signed=True)
Hyperplane 4*x - y - 8
sage: (4*x - y - 8).primitive(signed=False)
Hyperplane -4*x + y + 8
```

**to_symmetric_space()**

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

**EXAMPLES:**

```python
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
```
1.4 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
```

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

Bases: sage.structure.sage_object.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:
\[
\begin{align*}
    p &= (1, 0, 0, 0) \\
    W &= \text{Vector space of dimension 4 over Rational Field}
\end{align*}
\]

**dimension()**
Return the dimension of the affine space.

**OUTPUT:**
An 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
```

**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:
\[
    p = (1, 1, 0) \\
    W = \text{Vector space of degree 3 and dimension 1 over Rational Field}
\]
Basis matrix:
\[
\begin{bmatrix}
0 & 1 & 0
\end{bmatrix}
\]
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:
\[
    p = (7, 8, 0) \\
    W = \text{Vector space of degree 3 and dimension 0 over Rational Field}
\]
Basis matrix:
\[
[]
\]
sage: D = AffineSubspace([1,2,3], VectorSpace(GF(5),3))
sage: E = AffineSubspace([3,4,5], VectorSpace(GF(5),3))
sage: D.intersection(E)
```

(continues on next page)
Affine space \( p + W \) where:
\[ p = (3, 4, 0) \]
\[ W = \text{Vector space of dimension 3 over Finite Field of size 5} \]

**linear_part()**

Return the linear part of the affine space.

**OUTPUT:**

A vector subspace of the ambient space.

**EXAMPLES:**

```sage
define A = AffineSubspace([2,3,1], matrix(QQ, [[1,2,3]]).right_kernel()) A.linear_part() Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
\[
\begin{bmatrix}
1 & 0 & -1/3 \\
0 & 1 & -2/3 \\
\end{bmatrix}
\]
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
define A = AffineSubspace([2,3,1], VectorSpace(QQ,3)) A.point() (2, 3, 1)```

## 1.5 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-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, giving the sizes of points in a zero-
dimensional arrangement (default: 0).

- **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, hyperplane_˓→→legend=False)
Graphics3d Object

sage: c = H3([(1,0,0),0], [(0,0,1),5])
sage: c.plot(ranges=10)
Graphics3d Object

sage: c.plot(ranges=[[9.5,10], [-3,3]])
Graphics3d Object

sage: c.plot(ranges=[[9.5,10], [-3,3]], [[-6,6], [-5,5]])
Graphics3d Object

sage: H2.<s,t> = HyperplaneArrangements(QQ)
sage: h = H2([(1,1),0], [(1,-1),0], [(0,1),2])
sage: h.plot(ranges=20)
Graphics object consisting of 3 graphics primitives

sage: h.plot(ranges=[-1, 10])
Graphics object consisting of 3 graphics primitives

sage: h.plot(ranges=[[-1, 1], [-5, 5], [-1, 10]])
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)
Graphics3d Object

sage: opts['hyperplane_labels'] = 'short'
sage: a.plot(**opts)
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'])
Graphics object consisting of 3 graphics primitives
sage: pts.plot(point_sizes=[50,100,200], hyperplane_colors='blue')
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)
Graphics3d Object
sage: a.plot(hyperplane_labels=True, label_colors=['red', 'green', 'black'])
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')
Graphics object consisting of 6 graphics primitives
sage: b = hyperplane_arrangements.semiorder(4)
sage: c = b.essentialization()
sage: legend_3d_3d(c, list(colors.values())[:12], length='long')
Graphics object consisting of 12 graphics primitives
sage: legend_3d_3d(c, list(colors.values())[:12], length='short')
Graphics object consisting of 12 graphics primitives
sage: p = legend_3d(c, list(colors.values())[:12], length='short')
sage: p.set_legend_options(ncol=4)
sage: type(p)
<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.

Chapter 1. Hyperplane arrangements
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:
```python
sage: B = hyperplane_arrangements.semiorder(4)
sage: B.plot()
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]]`). (The ranges are centered around `hyperplane.point()`.)

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

(continues on next page)
```
sage: b = 3*x + 4*y + 5
sage: b.plot()
Graphics object consisting of 2 graphics primitives
sage: b.plot(ranges=(1,5),label_offset=(2,-1))
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)
Graphics object consisting of 2 graphics primitives
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
```
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

The following constructions are available

- `Birkhoff_polytope()`
- `associahedron()`
- `buckyball()`
- `cross_polytope()`
- `cube()`
- `cuboctahedron()`
- `cyclic_polytope()`
- `dodecahedron()`
- `flow_polytope()`
- `Gosset_3_21()`
- `grand_antiprism()`
- `great_rhombicuboctahedron()`
- `hypercube()`
- `hypersimplex()`
- `icosahedron()`
- `icosidodecahedron()`
- `Kirkman_icosahedron()`
- `octahedron()`
- `parallelotope()`
- `pentakis_dodecahedron()`
- `permutahedron()`
- `regular_polygon()`
- `rhombic_dodecahedron()`
- `rhombicosidodecahedron()`
- `simplex()`
- `six_hundred_cell()`

Continued on next page
Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>small_rhombicuboctahedron()</td>
</tr>
<tr>
<td>snub_cube()</td>
</tr>
<tr>
<td>snub_dodecahedron()</td>
</tr>
<tr>
<td>tetrahedron()</td>
</tr>
<tr>
<td>truncated_cube()</td>
</tr>
<tr>
<td>truncated_dodecahedron()</td>
</tr>
<tr>
<td>truncated_icosidodecahedron()</td>
</tr>
<tr>
<td>truncated_tetrahedron()</td>
</tr>
<tr>
<td>truncated_octahedron()</td>
</tr>
<tr>
<td>twenty_four_cell()</td>
</tr>
</tbody>
</table>

**class** `sage.geometry.polyhedron.library.Polytopes`  
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 – a 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:**

```python
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
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)
```

**Gosset_3_21**(backend=None)  
Return the Gosset 321 polytope.

The Gosset 321 polytope is a uniform 7-polytope. It has 56 vertices, and 702 facets: 126 311 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:**

```python
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)
```

**Kirkman_icosahedron (backend=None)**

Return the Kirkman icosahedron.

The Kirkman icosahedron is a 3-polytope with integer coordinates: $\pm(9, \pm6, \pm6)$, $\pm(12, \pm4, 0)$, $(0, \pm12, \pm8)$, $(\pm6, 0, \pm12)$. See [Fe2012] for more information.

**INPUT:**

- **backend** – the backend to use to create the polytope.

**EXAMPLES:**

```python
sage: ki = polytopes.Kirkman_icosahedron()
sage: ki.f_vector()
(1, 20, 38, 20, 1)
sage: ki.volume()
6528
sage: vertices = ki.vertices()
sage: edges = [edge[0],edge[1]] for edge in ki.bounded_edges()
sage: edge_lengths = [norm(edge[0]-edge[1]) for edge in edges]
sage: union(edge_lengths)
[7, 8, 9, 11, 12, 14, 16]
```

**static associahedron (cartan_type)**

Construct an associahedron.

The generalized associahedron is a polytopal complex with vertices in one-to-one correspondence with clusters in the cluster complex, and with edges between two vertices if and only if the associated two clusters intersect in codimension 1.

The associahedron of type $A_n$ is one way to realize the classical associahedron as defined in the Wikipedia article Associahedron.

A polytopal realization of the associahedron can be found in [CFZ2002]. The implementation is based on [CFZ2002], Theorem 1.5, Remark 1.6, and Corollary 1.9.

**EXAMPLES:**

```python
sage: Asso = polytopes.associahedron(['A',2]); Asso
Generalized associahedron of type ['A', 2] with 5 vertices
sage: sorted(Asso.Hrepresentation(), key=repr)
[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,
 An inequality (1, 1) x + 1 >= 0]
sage: Asso.Vrepresentation()
```

(continues on next page)
(A vertex at (1, -1), A vertex at (1, 1), A vertex at (-1, 1), A vertex at (-1, 0), A vertex at (0, -1))

```
sage: polytopes.associahedron(['B',2])
Generalized associahedron of type ['B', 2] with 6 vertices
```

The two pictures of [CFZ2002] can be recovered with:

```
sage: Asso = polytopes.associahedron(['A',3]); Asso
Generalized associahedron of type ['A', 3] with 14 vertices
sage: Asso.plot()   # long time
Graphics3d Object
```

```
sage: Asso = polytopes.associahedron(['B',3]); Asso
Generalized associahedron of type ['B', 3] with 20 vertices
sage: Asso.plot()   # long time
Graphics3d Object
```

```
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 \( \mathbb{Q}[\phi] \) where \( \phi \) 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 - 6secs
sage: bb.f_vector()               # long time
(1, 60, 90, 32, 1)
sage: bb.base_ring()             # long time
Number Field in sqrt5 with defining polynomial x^2 - 5
```

A much faster implementation using floating point approximations:

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

Its faces are 5 regular pentagons and 6 regular hexagons:

```
sage: sum(1 for f in bb.faces(2) if len(f.vertices()) == 5)
12
```
cross_polytope \((\text{dim}, \text{backend}=\text{None})\)

Return a cross-polytope in dimension \(\text{dim}\).

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:

- \(\text{dim}\) – integer. The dimension of the cross-polytope.
- \(\text{backend}\) – the backend to use to create the polytope.

EXAMPLES:

```python
sage: four_cross = polytopes.cross_polytope(4)
sage: four_cross.f_vector()
(1, 8, 24, 32, 16, 1)
sage: four_cross.is_simple()
False
```

cube \((\text{backend}=\text{None})\)

Return the cube.

The cube is the Platonic solid that is obtained as the convex hull of the points \((\pm 1, \pm 1, \pm 1)\). It generalizes into several dimension into hypercubes.

See also:

hypercube()

INPUT:

- \(\text{backend}\) – the backend to use to create the polytope.

EXAMPLES:

```python
sage: c = polytopes.cube()
sage: c
A 3-dimensional polyhedron in \(\mathbb{Z}^3\) defined as the convex hull of 8 vertices
sage: c.f_vector()
(1, 8, 12, 6, 1)
sage: c.volume()
8
sage: c.plot()
Graphics3d Object
```

cuboctahedron \((\text{backend}=\text{None})\)

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:

- \(\text{backend}\) – the backend to use to create the polytope.
See also:

rhombic_dodecahedron()

EXAMPLES:

```python
sage: co = polytopes.cuboctahedron()
sage: co.f_vector()
(1, 12, 24, 14, 1)
```

Its faces are 8 triangles and 6 squares:

```python
sage: sum(1 for f in co.faces(2) if len(f.vertices()) == 3)
8
sage: sum(1 for f in co.faces(2) if len(f.vertices()) == 4)
6
```

Some more computation:

```python
sage: co.volume()
20/3
sage: 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 \((t, t^2, \ldots, t^\dim)\) with \( t \in \{0, 1, \ldots, n - 1\} \). 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.
- \( \text{base\_ring} \) – either \( \mathbb{Q} \) (default) or \( \mathbb{R}DF \).
- \( \text{backend} \) – the backend to use to create the polytope.

EXAMPLES:

```python
sage: c = polytopes.cyclic_polytope(4,10)
sage: 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:

- \( \text{exact} \) – (boolean, default \( \text{True} \)) If \( \text{False} \) use an approximate ring for the coordinates.
- \( \text{base\_ring} \) – (optional) the ring in which the coordinates will belong to. Note that this ring must contain \( \sqrt{5} \). If it is not provided and \( \text{exact} = \text{True} \) it will be the number field \( \mathbb{Q} \left[ \sqrt{5} \right] \) and if \( \text{exact} = \text{False} \) it will be the real double field.
- \( \text{backend} \) – the backend to use to create the polytope.

EXAMPLES:
Here is an error with a field that does not contain $\sqrt{5}$:

```python
sage: polytopes.dodecahedron(base_ring=QQ)
Traceback (most recent call last):
  ...TypeError: unable to convert 1/4*sqrt(5) + 1/4 to a rational
```

**static flow_polytope** *(edges=None, ends=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 $R^m$, 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()`. 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 [PitSta].

**INPUT:**

- `edges` – (optional, default: `self.edges()`) a list or tuple of all edges of `self` (each only once). 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` – (optional, default: `(self.sources(), self.sinks())`) a pair $(S, T)$ of an iterable $S$ and an iterable $T$.

**Note:** Flow polytopes can also be built through the `polytopes` object.
```python
sage: polytopes.flow_polytope(digraphs.Path(5))
A 0-dimensional polyhedron in QQ^4 defined as the convex hull of 1 vertex
```

**EXAMPLES:**

A commutative square:

```python
sage: G = DiGraph({1: [2, 3], 2: [4], 3: [4]})
sage: fl = G.flow_polytope(); fl
A 1-dimensional polyhedron in QQ^4 defined as the convex hull of 2 vertices
sage: fl.vertices()
(A vertex at (0, 1, 0, 1), A vertex at (1, 0, 1, 0))
```

Using a different order for the edges of the graph:

```python
sage: fl = G.flow_polytope(edges=G.edges(key=lambda x: x[0] - x[1])); fl
A 1-dimensional polyhedron in QQ^4 defined as the convex hull of 2 vertices
sage: fl.vertices()
(A vertex at (0, 1, 1, 0), A vertex at (1, 0, 0, 1))
```

A tournament on 4 vertices:

```python
sage: H = digraphs.TransitiveTournament(4)
sage: fl = H.flow_polytope(); fl
A 3-dimensional polyhedron in QQ^6 defined as the convex hull of 4 vertices
sage: fl.vertices()
(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:

```python
sage: fl = H.flow_polytope(edges=[(0, 1, None), (1, 2, None),
 ..... (2, 3, None), (0, 3, None)]);
A 1-dimensional polyhedron in QQ^4 defined as the convex hull of 2 vertices
sage: fl.vertices()
(A vertex at (0, 0, 0, 1), A vertex at (1, 1, 1, 0))
```

Using a different choice of sources and sinks:

```python
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, 0, 1, 0), A vertex at (0, 0, 0, 1, 0, 0))
```

(continues on next page)
A digraph with one source and two sinks:

```python
sage: Y = DiGraph({1: [2], 2: [3, 4]})
sage: Y.flow_polytope()
```

The empty polyhedron in \( \mathbb{Q}^3 \)

A digraph with one vertex and no edge:

```python
sage: Z = DiGraph({1: []})
sage: Z.flow_polytope()
```

A 0-dimensional polyhedron in \( \mathbb{Q}^0 \) defined as the convex hull of 1 vertex

REFERENCES:

grand_antiprism (exact=True, backend=None)

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:

```python
sage: gap = polytopes.grand_antiprism()  # not tested - very long time
sage: gap
A 4-dimensional polyhedron in (Number Field in sqrt5 with defining polynomial \( x^2 - 5 \))^4 defined as the convex hull of 100 vertices
```
Computation with approximated coordinates is much faster:

```
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
```

**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 the number field \( \mathbb{Q}[\phi] \) where \( \phi \) 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 ~ 3sec
sage: gr.f_vector()  # long time
(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)
```

Its faces are 4 squares, 8 regular hexagons and 6 regular octagons:

```
sage: sum(1 for f in gr.faces(2) if len(f.vertices()) == 4)
12
sage: sum(1 for f in gr.faces(2) if len(f.vertices()) == 6)
8
sage: sum(1 for f in gr.faces(2) if len(f.vertices()) == 8)
6
```

**hypercube** *(dim, backend=None)*

Return a hypercube in the given dimension.

The \( d \) dimensional hypercube is the convex hull of the points \((\pm1, \pm1, \ldots, \pm1)\) in \( \mathbb{R}^d \). For more information see the Wikipedia article Hypercube.

**INPUT:**

- `dim` – integer. The dimension of the cube.
- `backend` – the backend to use to create the polytope.
EXAMPLES:

```
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
```

**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 \( \mathbb{R}^{d-1} \) (with floating point coordinates) then set project=True in the options.

See also:

```python
calculus()
```

INPUT:

- **dim** – the dimension
- **n** – the numbers \( (1, \ldots, n) \) are permuted
- **project** – (boolean, default False) if True, the polytope is (isometrically) projected to a vector space of dimension \( d-1 \). 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: 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
2/3*t^3 + 2*t^2 + 7/3*t + 1
```

```
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)
```

**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 \( \mathbb{Q}[\sqrt{5}] \) 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()
sage: ico.f_vector()
(1, 12, 30, 20, 1)
sage: ico.volume()
5/12*sqrt5 + 5/4
```

Its non exact version:

```
sage: ico = polytopes.icosahedron(exact=False)
sage: ico.base_ring()
Real Double Field
sage: ico.volume() # known bug (trac 18214)
2.181694990...
```

A version using \( \mathbb{A}_{\text{AA}} \subset \text{sage.rings.qqbar.AlgebraicRealField} \):

```
sage: ico = polytopes.icosahedron(base_ring=AA) # long time
sage: ico.base_ring()
# long time
Algebraic Real Field
sage: ico.volume() # long time
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)
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()
sage: id.f_vector()
(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 $\mathbb{Q}[\phi]$ where $\phi$ 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:**

```python
sage: id = polytopes.icosidodecahedron()  # long time - 6secs
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
```

A much faster implementation using floating point approximations:

```python
sage: id = polytopes.icosidodecahedron(exact=False)
sage: id.f_vector()  
(1, 30, 60, 32, 1)
sage: id.base_ring()  
Real Double Field
```

Its faces are 20 triangles and 12 regular pentagons:

```python
sage: sum(1 for f in id.faces(2) if len(f.vertices()) == 3)  
20
sage: sum(1 for f in id.faces(2) if len(f.vertices()) == 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:**

```python
sage: co = polytopes.octahedron()  
sage: co.f_vector()  
(1, 6, 12, 8, 1)
sage: co.volume()  
4/3
```

Some more computation:

```python
sage: co.ehrhart_polynomial()  
4/3*t^3 + 2*t^2 + 8/3*t + 1
```

**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** – a 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')  
sage: sqrt2 = K.gen()  
sage: polytopes.parallelotope([ (1,sqrt2), (1,-1) ])  
A 2-dimensional polyhedron in (Number Field in sqrt2 with defining polynomial x^2 - 2)^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 faces 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 the number field \( \mathbb{Q}[\phi] \) where \( \phi \) 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 - ~10 sec
sage: pd.n_vertices()  
32  
sage: pd.n_inequalities()  
60  
```

A much faster implementation is obtained when setting **exact=False**:

```
sage: pd = polytopes.pentakis_dodecahedron(exact=False)  
sage: pd.n_vertices()  
32  
sage: pd.n_inequalities()  
60  
```

The 60 are triangles:

```
sage: all(len(f.vertices()) == 3 for f in pd.faces(2))  
True
```

---

**permutahedron** *(n, project=False, backend=None)*

Return the standard permutahedron of \((1,\ldots,n)\).
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 \( \text{dim}-1 \). 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:**

```python
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()
Graphics3d Object
sage: perm4.graph().is_isomorphic(graphs.BubbleSortGraph(4))
True
```

See also:
- BubbleSortGraph()
sage: v = octagon.volume()
sage: v
2.828427124746190
sage: v == 2*QQbar(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

rhombic_dodecahedron(backend=None)
Return the rhombic dodecahedron.

The rhombic dodecahedron is a 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)

Its faces are 12 quadrilaterals (not all identical):

sage: sum(1 for f in rd.faces(2) if len(f.vertices()) == 4)
12

Some more computations:

sage: p = rd.ehrhart_polynomial()  # optional - latte_int
sage: p
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]

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 informa-
tion, 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 the number field \(\mathbb{Q}[\phi]\) where \(\phi\) 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:**

```python
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
```

A much faster implementation using floating point approximations:

```python
sage: rid = polytopes.rhombicosidodecahedron(exact=False)
sage: rid.f_vector()
(1, 60, 120, 62, 1)
sage: rid.base_ring()
Real Double Field
```

Its faces are 20 triangles, 30 squares and 12 pentagons:

```python
sage: sum(1 for f in rid.faces(2) if len(f.vertices()) == 3)
20
sage: sum(1 for f in rid.faces(2) if len(f.vertices()) == 4)
30
sage: sum(1 for f in rid.faces(2) if len(f.vertices()) == 5)
12
```

**simplex** *(dim=3, project=False, base_ring=None, backend=None)*

Return the `dim` dimensional simplex.

The `d`-simplex is the convex hull in \(\mathbb{R}^{d+1}\) 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](https://en.wikipedia.org/wiki/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 \(d-1\). 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 \(\mathbb{Z}\). Otherwise, it defaults to `RDF`.

• **backend** – the backend to use to create the polytope.

**See also:**

**tetrahedron()**

**EXAMPLES:**

```python
sage: s5 = polytopes.simplex(5)
sage: s5
A 5-dimensional polyhedron in \(\mathbb{Z}^6\) defined as the convex hull of 6 vertices
sage: s5.f_vector()
```

(continues on next page)
sage: s6 = polytopes.simplex(6, project=True)
sage: s6.volume()  # abs tol 1e-10
0.00367465459870082
sage: sqrt(7.) / factorial(6)
0.00367465459870082

Computation in algebraic reals:

sage: s3 = polytopes.simplex(3, project=True, base_ring=AA)
sage: s3.volume() == sqrt(3+1) / factorial(3)
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()
sage: p600
A 4-dimensional polyhedron in RDF^4 defined as the convex hull of 120 vertices
sage: p600.f_vector()  # long time ~2sec
(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)  # not tested - very long
sage: len(list(p600.bounded_edges()))  # not tested - very long
120

70 Chapter 2. Polyhedral computations
**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](https://en.wikipedia.org/wiki/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 $\mathbb{Q}[\phi]$ where $\phi$ 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:**

```python
sage: sr = polytopes.small_rhombicuboctahedron()
sage: sr.f_vector()
(1, 24, 48, 26, 1)
sage: sr.volume()
80/3*sqrt2 + 32
```

The faces are 8 equilateral triangles and 18 squares:

```python
sage: sum(1 for f in sr.faces(2) if len(f.vertices()) == 3)
8
sage: sum(1 for f in sr.faces(2) if len(f.vertices()) == 4)
18
```

Its non exact version:

```python
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)
```

**snub_cube** *(backend=None)*

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](https://en.wikipedia.org/wiki/Snub_cube).

It uses the real double field for the coordinates.

**INPUT:**

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

**EXAMPLES:**

```python
sage: sc = polytopes.snub_cube()
sage: sc.f_vector()
(1, 24, 60, 38, 1)
```

**snub_dodecahedron** *(base_ring=None, backend=None)*

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:

Unfortunately, no polyhedra backend supports the construction of the snub dodecahedron at the moment:

```python
sage: sd = polytopes.snub_dodecahedron()
Traceback (most recent call last):
  ...  
ValueError: *Error: Numerical inconsistency is found. Use the GMP exact_arithmetic.
```

Its faces are 80 triangles and 12 pentagons:

```python
sage: sum(1 for f in sd.faces(2) if len(f.vertices()) == 3) # not tested
80
sage: sum(1 for f in sd.faces(2) if len(f.vertices()) == 5) # not tested
12
```

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:

```python
sage: co = polytopes.tetrahedron()
sage: co.f_vector()
(1, 4, 6, 4, 1)
```

Its faces are 4 triangles:

```python
sage: sum(1 for f in co.faces(2) if len(f.vertices()) == 3) # not tested
4
```

Some more computation:

```python
sage: co.volume()
1/3
```
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 the number field \(\mathbb{Q}[\sqrt{2}]\) 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()
sage: co.f_vector()
(1, 24, 36, 14, 1)
```

Its faces are 8 triangles and 6 octagons:

```
sage: sum(1 for f in co.faces(2) if len(f.vertices()) == 3)
8
sage: sum(1 for f in co.faces(2) if len(f.vertices()) == 8)
6
```

Some more computation:

```
sage: co.volume()
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 the number field \(\mathbb{Q}[\phi]\) where \(\phi\) 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()
sage: td.f_vector()
(1, 60, 90, 32, 1)
```

(continues on next page)
Its faces are 20 triangles and 12 regular decagons:

```python
sage: sum(1 for f in td.faces(2) if len(f.vertices()) == 3)
sage: sum(1 for f in td.faces(2) if len(f.vertices()) == 10)
20
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:

```python
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):
...
KeyError: ...
sage: 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 \(\mathbb{Q}[\phi]\) where \(\phi\) 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:**

```python
sage: ti = polytopes.truncated_icosidodecahedron()  # long time
tsage: ti.f_vector()  # long time
(1, 120, 180, 62, 1)
tsage: ti.base_ring()  # long time
Number Field in sqrt5 with defining polynomial x^2 - 5
```

The implementation using floating point approximations is much faster:

```python
sage: ti = polytopes.truncated_icosidodecahedron(exact=False)  # random
tsage: ti.f_vector()
sage: ti.base_ring()
(1, 120, 180, 62, 1)
Real Double Field
```

Its faces are 30 squares, 20 hexagons and 12 decagons:

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)

Its faces are 6 squares and 8 hexagons:

sage: sum(1 for f in co.faces(2) if len(f.vertices()) == 4)
6
sage: sum(1 for f in co.faces(2) if len(f.vertices()) == 6)
8

Some more computation:

sage: co.volume()
32
sage: co.ehrhart_polynomial()  # optional - latte_int
32*t^3 + 18*t^2 + 6*t + 1

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)

Its faces are 4 triangles and 4 hexagons:
Some more computation:

```python
sage: co.volume()
184/3
sage: 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:**

```python
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
```

**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 – a list of vectors of same dimension
- backend – the backend to use to create the polytope.

**EXAMPLES:**

```python
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')
sage: sqrt2 = K.gen()
sage: polytopes.parallelotope([(1,sqrt2), (1,-1)])
A 2-dimensional polyhedron in (Number Field in sqrt2 with defining polynomial x^2 - 2)^2 defined as the convex hull of 4 vertices

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:

```python
sage: from sage.geometry.polyhedron.library import project_points
sage: project_points([2,-1,3,2])
[(2.1213203435596424, -2.041241452319315, -0.577350269189626)]
sage: project_points([1,2,3],[3,3,5])
[(-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 $v$:

```python
sage: project_points([1,2])
[(-0.7071067811865475)]
sage: project_points([1,2,3])
[(-0.7071067811865475, -1.2247448713915892)]
sage: project_points([1,2,3,4])
[(-0.7071067811865475, -1.2247448713915892, -1.7320508075688776)]
sage: project_points([1,2,3,4,0])
[(-0.7071067811865475, -1.2247448713915892, -1.7320508075688776, 2.23606797749979)]
```

Check that it is (almost) an isometry:

```python
sage: V = list(map(vector, IntegerVectors(n=5,length=3)))
sage: P = project_points(*V, base_ring=AA)
sage: for i in range(len(V)):
    for j in range(len(V)):
        assert abs((V[i]-V[j]).norm() - (P[i]-P[j]).norm()) < 0.00001
```

Example with exact computation:

```python
sage: V = [ vector(v) for v in IntegerVectors(n=4,length=2) ]
sage: P = project_points(*V, base_ring=AA)
sage: for i in range(len(V)):
    for j in range(len(V)):
        assert (V[i]-V[j]).norm() == (P[i]-P[j]).norm()
```

2.1. Polyhedra
sage.geometry.polyhedron.library.zero_sum_projection(d, base_ring=Real Double Field)

Return a matrix corresponding to the projection on the orthogonal of $(1, 1, \ldots, 1)$ in dimension $d$.

The projection maps the orthonormal basis $(1, -1, 0, \ldots, 0)/\sqrt{2}$, $(1, 1, -1, 0, \ldots, 0)/\sqrt{3}$, idots, $(1, 1, \ldots, 1, -1)/\sqrt{d}$ to the canonical basis in $\mathbb{R}^{d-1}$.

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]
```

Exact computation in $AA < sage.rings.qqbar.AlgebraicRealField >$:

```
sage: zero_sum_projection(3, base_ring=AA)
[ 0.7071067811865475? -0.7071067811865475? 0.0]
[ 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

$$P = \text{conv}\{v_1, \ldots, v_k\} + \sum_{i=1}^{m} R_+ r_i + \sum_{j=1}^{n} R \ell_j$$

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 non-zero 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 non-zero vector, and two lines are equivalent if and only if the vectors are the same up to rescaling with a non-zero (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))
```

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))
```

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: strip.faces(1)
(<0,1>, <0,2>)
sage: for face in strip.faces(1):
    ....:     print("{} = {}".format(face, face.as_polyhedron().Vrepresentation()))
<0,1> = (A line in the direction (0, 1), A vertex at (-1, 0))
<0,2> = (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: v = next(trunc_quadr.vertex_generator())  # the first vertex in the internal...
....:     enumeration
sage: v
A vertex at (0, 1)
```

(continues on next page)
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() )
<type 'sage.modules.vector_integer_dense.Vector_integer_dense'>
sage: v.polyhedron()
A 2-dimensional polyhedron in ZZ^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)]

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)

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/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)
sage: triangle.Hrepresentation()
(An inequality (-1, -0.5773502691896258?) x + 1 >= 0,
 An inequality (1, 0) x + 0 >= 0,
 An inequality (0, 1) 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:
Even faster than all algebraic real numbers (the field $\mathbb{AA}$) is to take the smallest extension field. For the equilateral triangle, that would be:

```python
sage: K.<sqrt3> = NumberField(x^2 - 3, embedding=AA(3)**(1/2))
sage: Polyhedron([(0,0), (1,0), (1/2, sqrt3/2)])
A 2-dimensional polyhedron in (Number Field in sqrt3 with defining polynomial x^2 - 3)^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 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:

```python
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 = [1.123456789012345, 2.123456789012345])
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*:

```python
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
```

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_incidence, 2010
• Volker Braun: rewrite to use PPL instead of cddlib, 2011
• Volker Braun: Add support for arbitrary subfields of the reals

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 – list of point. 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.
• 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 \(7x_1 + 3x_2 + 4x_3 = 1\).
• base_ring – a sub-field of the reals implemented in Sage. The field over which the polyhedron will be defined. For \( \mathbb{Q} \) 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.

• ambient_dim – integer. The ambient space dimension. Usually can be figured out automatically from the H/Vrepresentation dimensions.

• backend – string or None (default). The backend to use. Valid choices are
  - 'cdd': use cdd (backend_cdd) with \( \mathbb{Q} \) or \( \mathbb{R} \) coefficients depending on base_ring.
  - 'normaliz': use normaliz (backend_normaliz) with \( \mathbb{Z} \) or \( \mathbb{Q} \) coefficients depending on base_ring.
  - 'polymake': use polymake (backend_polymake) with \( \mathbb{Q}, \mathbb{R} \) or QuadraticField coefficients depending on base_ring.
  - 'ppl': use ppl (backend_ppl) with \( \mathbb{Z} \) or \( \mathbb{Q} \) 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.

OUTPUT:
The polyhedron defined by the input data.

EXAMPLES:

Construct some polyhedra:

\[
\begin{align*}
\text{sage: } & \text{square_from_vertices = Polyhedron(}\text{vertices} = \left[[1, 1], [1, -1], [-1, 1], [-1, -1]\right]) \\
\text{sage: } & \text{square_from_ieqs = Polyhedron(}\text{ieqs} = \left[[1, 0, 1], [1, 1, 0], [1, 0, -1], [1, -1, 0]\right]) \\
\text{sage: } & \text{list(square_from_ieqs.}\text{vertex_generator()}) \\
& \text{[A vertex at (1, -1),} \\
& \text{A vertex at (1, 1),} \\
& \text{A vertex at (-1, 1),} \\
& \text{A vertex at (-1, -1)]} \\
\text{sage: } & \text{list(square_from_vertices.}\text{inequality_generator()}) \\
& \text{[An inequality (1, 0) x + 1 >= 0,} \\
& \text{An inequality (0, 1) x + 1 >= 0,} \\
& \text{An inequality (-1, 0) x + 1 >= 0,} \\
& \text{An inequality (0, -1) x + 1 >= 0]} \\
\text{sage: } & \text{p = Polyhedron(}\text{vertices} = \left[[1.1, 2.2], [3.3, 4.4]\right], \text{base_ring=RDF}) \\
\text{sage: } & \text{p.n_}\text{inequalities()}} \text{2}
\end{align*}
\]

The same polyhedron given in two ways:

\[
\begin{align*}
\text{sage: } & \text{p = Polyhedron(}\text{ieqs} = \left[[0,1,0,0],[0,0,1,0]\right]) \\
\text{sage: } & \text{p.Vrepresentation()}
& \text{(A line in the direction (0, 0, 1),} \\
& \text{A ray in the direction (1, 0, 0),} \\
& \text{A ray in the direction (0, 1, 0),}
\end{align*}
\]

(continues on next page)
A vertex at \((0, 0, 0)\)

```python
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)
```

Finally, a more complicated example. Take \(\mathbb{R}^6_{\geq 0}\) with coordinates \(a, b, \ldots, f\) and

- The inequality \(e + b \geq c + d\)
- The inequality \(e + c \geq b + d\)
- The equation \(a + b + c + d + e + f = 31\)

```python
sage: positive_coords = Polyhedron(ieqs=[
....: [0, 1, 0, 0, 0, 0],
....: [0, 0, 1, 0, 0, 0],
....: [0, 0, 0, 1, 0, 0],
....: [0, 0, 0, 0, 1, 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, 31, 0, 0),
 A vertex at (0, 0, 31/2, 31/2, 0, 0),
 A vertex at (0, 0, 31/2, 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:

```python
sage: R0.<r0> = QQ[]
sage: R1.<r1> = NumberField(r0^2-5, embedding=AA(5)**(1/2))
sage: grat = (1+r1)/2
sage: v = [[0, 1, grat], [0, 1, -grat], [0, -1, grat], [0, -1, -grat]]
sage: pp = Permutation((1, 2, 3))
sage: icosah = Polyhedron([[pp^2].action(w) for w in v]  
....: + [pp.action(w) for w in v] + v, base_ring=R1)
sage: len(icosah.faces(2))
20
```

When the input contains elements of a Number Field, they require an embedding:

```python
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
```

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` *(base_ring, ambient_dim, backend=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.

• `backend` – string. The name of the backend for computations. There are several backends implemented:
  - `backend="ppl"` uses the Parma Polyhedral 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:

```python
sage: from sage.geometry.polyhedron.parent import Polyhedra
sage: Polyhedra(AA, 3)
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'>
```

CDD does not support integer polytopes directly:

```python
sage: Polyhedra(ZZ, 3, backend='cdd')
Polyhedra in QQ^3
```

### 2.1. Polyhedra

class `sage.geometry.polyhedron.parent.Polyhedra_QQ_cdd` *(base_ring, ambient_dim, backend)*

Bases: `sage.geometry.polyhedron.parent.Polyhedra_base`

Element

alias of `sage.geometry.polyhedron.backend_cdd.Polyhedron_QQ_cdd`
class sage.geometry.polyhedron.parent.Polyhedra_QQ_normaliz(base_ring, ambient_dim, backend)

Bases: sage.geometry.polyhedron.parent.Polyhedra_base

Element
alias of sage.geometry.polyhedron.backend_normaliz.Polyhedron_QQ_normaliz

class sage.geometry.polyhedron.parent.Polyhedra_QQ_ppl(base_ring, ambient_dim, backend)

Bases: sage.geometry.polyhedron.parent.Polyhedra_base

Element
alias of sage.geometry.polyhedron.backend_ppl.Polyhedron_QQ_ppl

class sage.geometry.polyhedron.parent.Polyhedra_RDF_cdd(base_ring, ambient_dim, backend)

Bases: sage.geometry.polyhedron.parent.Polyhedra_base

Element
alias of sage.geometry.polyhedron.backend_cdd.Polyhedron_RDF_cdd

class sage.geometry.polyhedron.parent.Polyhedra_ZZ_normaliz(base_ring, ambient_dim, backend)

Bases: sage.geometry.polyhedron.parent.Polyhedra_base

Element
alias of sage.geometry.polyhedron.backend_normaliz.Polyhedron_ZZ_normaliz

class sage.geometry.polyhedron.parent.Polyhedra_ZZ_ppl(base_ring, ambient_dim, backend)

Bases: sage.geometry.polyhedron.parent.Polyhedra_base

Element
alias of sage.geometry.polyhedron.backend_ppl.Polyhedron_ZZ_ppl

class sage.geometry.polyhedron.parent.Polyhedra_base(base_ring, ambient_dim, backend)

Bases: sage.structure.unique_representation.UniqueRepresentation, sage.structure.parent.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:

```python
def from sage.geometry.polyhedron.parent import Polyhedra
def Polyhedra(ZZ, 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 $\text{ambient\_dim()} + 1$.

EXAMPLES:

```sage
def Hrepresentation_space():
    from sage.geometry.polyhedron.parent import Polyhedra
    Polyhedra(QQ, 3).Hrepresentation_space()
```

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 $\text{ambient\_dim()}$.

EXAMPLES:

```sage
def Vrepresentation_space():
    from sage.geometry.polyhedron.parent import Polyhedra
    Polyhedra(QQ, 4).Vrepresentation_space()
```

ambient_dim()
Return the dimension of the ambient space.

EXAMPLES:

```sage
def ambient_dim():
    from sage.geometry.polyhedron.parent import Polyhedra
    Polyhedra(QQ, 3).ambient_dim()
```

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 $\text{ambient\_dim()}$.

EXAMPLES:

```sage
def ambient_space():
    from sage.geometry.polyhedron.parent import Polyhedra
    Polyhedra(QQ, 4).ambient_space()
```

an_element()
Returns 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

backend()
Return the backend.

EXAMPLES:

sage: from sage.geometry.polyhedron.parent import Polyhedra
sage: Polyhedra(QQ, 3).backend()
'ppl'

base_extend(base_ring, backend=None)
Return the base extended parent.

INPUT:

* base_ring, backend – see Polyhedron().

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

change_ring(base_ring, backend=None)
Return the parent with the new base ring.

INPUT:

* base_ring, backend – see Polyhedron().

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)
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 4 vertices

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

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)])
p = p.parent().recycle(p)
```

**some_elements()**

Returns a list of some elements of the semigroup.

EXAMPLES:

```sage:
from sage.geometry.polyhedron.parent import Polyhedra
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]
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]
```

**universe()**

Return the entire ambient space as polyhedron.

EXAMPLES:

```sage:
from sage.geometry.polyhedron.parent import Polyhedra
P = Polyhedra(QQ, 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
p = Polyhedra(QQ, 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: sage.geometry.polyhedron.parent.Polyhedra_base

Element

- alias of sage.geometry.polyhedron.backend_field.Polyhedron_field

**class** sage.geometry.polyhedron.parent.Polyhedra_polymake(base_ring, ambient_dim, backend)

Bases: sage.geometry.polyhedron.parent.Polyhedra_base
Element
alias of sage.geometry.polyhedron.backend_polymake.Polyhedron_polymake

2.1.4 H(yperplane) and V(ertex) representation objects for polyhedra
class sage.geometry.polyhedron.representation.Equation(polyhedron_parent)
Bases: sage.geometry.polyhedron.representation.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)
Tests 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
```

interior_contains(Vobj)
Tests whether the interior of the halfspace (excluding its boundary) defined by the inequality contains the given vertex/ray/line.
NOTE:
Returns 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
```
is_equation()
Tests if this object is an equation. By construction, it must be.
type()
Returns the type (equation/inequality/vertex/ray/line) as an integer.
OUTPUT:
Integer. One of PolyhedronRepresentation.INEQUALITY, .EQUATION, .VERTEX, .RAY, or .LINE.
EXAMPLES:
```python
sage: p = Polyhedron(vertices = [[0,0,0],[1,1,0],[1,2,0]])
sage: repr_obj = next(p.equation_generator())
sage: repr_obj.type()
i
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
```

class sage.geometry.polyhedron.representation.Hrepresentation(polyhedron_parent)

Bases: sage.geometry.polyhedron.representation.PolyhedronRepresentation

The internal base class for H-representation objects of a polyhedron. Inherits from PolyhedronRepresentation.

A()

Returns the coefficient vector \( A \) in \( A \vec{x} + b \).

EXAMPLES:

```python
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)
```

adjacent()

Alias for neighbors().

b()

Returns the constant \( b \) in \( A \vec{x} + b \).

EXAMPLES:

```python
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
```

eval(Vobj)

Evaluates the left hand side \( A \vec{x} + b \) on the given vertex/ray/line.

NOTES:

- 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:

```python
sage: triangle = Polyhedron(vertices=[[1,0],[0,1],[-1,-1]])
sage: ineq = next(triangle.inequality_generator())
```

(continues on next page)
An inequality $(2, -1) x + 1 \geq 0$

```
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]
```

If you pass a vector, it is assumed to be the coordinate vector of a point:

```
sage: ineq.eval( vector(ZZ, [3,2]) )
5
```

**incident()**

Returns 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 $(2, -1) x + 1 \geq 0$

```

```
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 $(1, 0, 1) x + 0 \geq 0$

```

```
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)]
```

**is_H()**

Returns 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
```

**is_equation()**

Returns 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
```

**is_incident (Vobj)**

Returns whether the incidence matrix element $(Vobj, self) == 1$

**EXAMPLES:**
is_inequality()  
Returns True if the object is an inequality of the H-representation.

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_inequality()
True
```

neighbors()  
Iterate over the adjacent facets (i.e. inequalities/equations)

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]
```

repr_pretty(**kwds)  
Return a pretty representation of this equality/inequality.

INPUT:

- prefix – a string
- indices – a tuple or other iterable
- latex – a boolean

OUTPUT:

A 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():
....:     print(h.repr_pretty())
     x0 + x1 - x2 == 1
     x0 >= 0
     2*x0 + x1 >= -1
```

class sage.geometry.polyhedron.representation.Inequality(polyhedron_parent)  
Bases: sage.geometry.polyhedron.representation.Hrepresentation

A linear inequality (supporting hyperplane) of the polyhedron. Inherits from Hrepresentation.
contains ($Vobj$)
Tests whether the halfspace (including its boundary) defined by the inequality contains the given vertex/ray/line.

EXAMPLES:

```python
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, False, False, False, True, True, True, False]
```

interior_contains ($Vobj$)
Tests whether the interior of the halfspace (excluding its boundary) defined by the inequality contains the given vertex/ray/line.

EXAMPLES:

```python
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)
sage: [i1.interior_contains(q) for q in p2.vertex_generator()]
[True, False, False, False, True, True, True, False]
```

If you pass a vector, it is assumed to be the coordinate vector of a point:

```python
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]
```

is_inequality ()
Returns True since this is, by construction, an inequality.

EXAMPLES:

```python
sage: p = Polyhedron(vertices = [[0,0,0],[1,1,0],[1,2,0]])
sage: a = next(p.inequality_generator())
sage: a.is_inequality()
True
```

type ()
Returns the type (equation/inequality/vertex/ray/line) as an integer.

OUTPUT:
Integer. One of `PolyhedronRepresentation.INEQUALITY`, `EQUATION`, `VERTEX`, `RAY`, or `LINE`.

EXAMPLES:

```python
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
```

(continues on next page)
class sage.geometry.polyhedron.representation.Line(polyhedron_parent)
   
   A line (Minkowski summand \( \cong \mathbb{R} \)) of the polyhedron. Inherits from Vrepresentation.

   **evaluated_on** (Hobj)
   Returns \( A \vec{f} \)

   **EXAMPLES:**

   ```sage
   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
   ```

   **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
   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)
   ```

   **is_line**()
   Tests if the object is a line. By construction it must be.

   **type**()
   Returns the type (equation/inequality/vertex/ray/line) as an integer.

   **OUTPUT:**
   Integer. One of PolyhedronRepresentation.INEQUALITY, .EQUATION, .VERTEX, .RAY, or .LINE.

   **EXAMPLES:**

   ```sage
   sage: p = Polyhedron(ieqs = [[1, 0, 0, 1],[1,1,0,0]])
   sage: repr_obj = next(p.line_generator())
   sage: repr_obj.type()
   ```

2.1. Polyhedra

95
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

```python
class sage.geometry.polyhedron.representation.PolyhedronRepresentation
Bases: sage.structure.sage_object.SageObject

The internal base class for all representation objects of Polyhedron (vertices/rays/lines and in-
equalities/equations)

**Note:** You should not (and cannot) instantiate it yourself. You can only obtain them from a Polyhedron() class.
```

**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:**

```python
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
```

**index** *(*)

Returns an arbitrary but fixed number according to the internal storage order.

**NOTES:**

H-representation and V-representation objects are enumerated independently. That is, amongst all ver-
tices/rays/lines there will be one with `index()==0`, and amongs all inequalities/equations there will be
one with `index()==0`, unless the polyhedron is empty or spans the whole space.

**EXAMPLES:**

```python
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
```
polyhedron()
    Returns the underlying polyhedron.

vector(base_ring=None)
    Returns 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 \( \text{ambient_dim()} \).
    For a H-representation object, a vector of length \( \text{ambient_dim()} + 1 \).

    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())
    <type '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)

class sage.geometry.polyhedron.representation.Ray(polyhedron_parent)
    Bases: sage.geometry.polyhedron.representation.Vrepresentation

    A ray of the polyhedron. Inherits from Vrepresentation.

    evaluated_on(Hobj)
        Returns \( \vec{A}r \)

    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

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)

is_ray()
Tests 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
```

type()
Returns 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
True
sage: repr_obj.type() == repr_obj.LINE
False
```

class sage.geometry.polyhedron.representation.Vertex(polyhedron_parent)
Bases: sage.geometry.polyhedron.representation.Vrepresentation

A vertex of the polyhedron. Inherits from Vrepresentation.

evaluated_on (Hobj)
Returns $A\vec{x} + b$

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 (0, 0, -1) x + 1 >= 0
sage: v.evaluated_on(h)
2
```
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)
```

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]
```

is_vertex()

Tests 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
```

type()

Returns 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
```

(continues on next page)
class sage.geometry.polyhedron.representation.Vrepresentation(polyhedron_parent)

Bases: sage.geometry.polyhedron.representation.PolyhedronRepresentation

The base class for V-representation objects of a polyhedron. Inherits from PolyhedronRepresentation.

adjacent()

Alias for neighbors().

incident()

Returns 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)]
```

is_V()

Returns 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
```

is_incident(Hobj)

Returns whether the incidence matrix element (self,Hobj) == 1

EXAMPLES:

```
sage: p = polytopes.hypercube(3)
sage: h1 = next(p.inequality_generator())
sage: h1
An inequality (0, 0, -1) x + 1 >= 0
sage: v1 = next(p.vertex_generator())
sage: v1
A vertex at (-1, -1, -1)
sage: v1.is_incident(h1)
False
```
is_line()  
Returns 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:

```python
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
```

is_ray()  
Returns 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:

```python
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
```

is_vertex()  
Returns 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:

```python
sage: p = Polyhedron(vertices = [[0,0],[1,0],[0,3],[1,4]])
sage: v = next(p.vertex_generator())
sage: v.is_vertex()  
True
```

neighbors()  
Returns a generator for the adjacent vertices/rays/lines.

EXAMPLES:

```python
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)
```
Return a pretty representation of equation/inequality represented by the coefficients.

**INPUT:**
- **coefficients** – a tuple or other iterable
- **type** – either 0 (PolyhedronRepresentation.INEQUALITY) or 1 (PolyhedronRepresentation.EQUATION)
- **prefix** – a string
- **indices** – a tuple or other iterable
- **latex** – a boolean
- **split** – a 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: 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
```

### 2.1.5 Functions for plotting polyhedra

**class** `sage.geometry.polyhedron.plot.Projection(polyhedron, proj=<function projection_func_identity>)`

**Bases:** `sage.structure.sage_object.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)))
7
```

**coord_indices_of(v_list)**
Convert list of coordinate vectors to the corresponding list of internal indices.

**EXAMPLES:**

```
sage: from sage.geometry.polyhedron.plot import Projection
sage: p = polytopes.hypercube(3)
```

```python
sage: p = polytopes.hypercube(3)
sage: proj = p.projection()
sage: proj.coord_indices_of([vector((1,1,1)), vector((1,-1,1))])
[7, 5]
```

**coordinates_of** *(coord_index_list)*

Given a list of indices, return the projected coordinates.

**EXAMPLES:**

```python
sage: p = polytopes.simplex(4, project=True).projection()
sage: p.coordinates_of([1])
[[0.7071067812, 0.4082482905, 0.2886751346, 0.2236067977]]
```

**identity** *

Return the identity projection of the polyhedron.

**EXAMPLES:**

```python
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
```

**render_0d** *(point_opts={}, line_opts={}, polygon_opts=*)

Return 0d rendering of the projection of a polyhedron into 2-dimensional ambient space.

**INPUT:**

See `plot()`.

**OUTPUT:**

A 2-d graphics object.

**EXAMPLES:**

```python
sage: print(Polyhedron([]).projection().render_0d().description())
Point set defined by 1 point(s): [(0.0, 0.0)]
```

**render_1d** *(point_opts={}, line_opts={}, polygon_opts=*)

Return 1d rendering of the projection of a polyhedron into 2-dimensional ambient space.

**INPUT:**

See `plot()`.

**OUTPUT:**

A 2-d graphics object.

**EXAMPLES:**

```python
sage: Polyhedron([(0,), (1,)]).projection().render_1d()
Graphics object consisting of 2 graphics primitives
```

**render_2d** *(point_opts={}, line_opts={}, polygon_opts=*)

Return 2d rendering of the projection of a polyhedron into 2-dimensional ambient space.

2.1. Polyhedra 103
EXAMPLES:

```python
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()
Graphics object consisting of 17 graphics primitives
```

```
render_3d(point_opts={}, line_opts={}, polygon_opts={})

Return 3d rendering of a polyhedron projected into 3-dimensional ambient space.

EXAMPLES:

```python
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() + p3.projection().plot()
Graphics3d Object
```

It correctly handles various degenerate cases:

```python
sage: Polyhedron(lines=[[1,0,0],[0,1,0],[0,0,1]]).plot()  # whole
    → space
    Graphics3d Object
sage: Polyhedron(verts=[[1,1,1]], rays=[[1,0,0]],
                lines=[[0,1,0],[0,0,1]]).plot()  # half
    → space
    Graphics3d Object
sage: Polyhedron(verts=[[1,1,1]],
                lines=[[0,1,0],[0,0,1]]).plot()  # R^2 in R^3
    → R^3
    Graphics3d Object
sage: Polyhedron(rays=[[0,1,0]], lines=[[1,0,0]]).plot()  # quadrant
    → wedge in R^2
    Graphics3d Object
sage: Polyhedron(rays=[[0,1,0]], lines=[[1,0,0]]).plot()  # upper
    → half plane in R^3
    Graphics3d Object
sage: Polyhedron(lines=[[1,0,0]]).plot()  # R^1 in R^2
    → 2
    Graphics3d Object
sage: Polyhedron(rays=[[0,1,0]]).plot()  # Half-
    → line in R^3
    Graphics3d Object
sage: Polyhedron(verts=[[1,1,1]]).plot()  # point in
    → R^3
    Graphics3d Object
```

The origin is not included, if it is not in the polyhedron (trac ticket #23555):

```python
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
```

(continues on next page)
**render_fill_2d(**kwds)**
Return the filled interior (a polygon) of a polyhedron in 2d.

**EXAMPLES:**
```python
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()
sage: filled_poly.axes_width()
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:**
```python
sage: outline = polytopes.hypercube(1).projection().render_line_1d()
sage: outline._objects[0]
Line defined by 2 points
```

**render_outline_2d(**kwds)**
Return the outline (edges) of a polyhedron in 2d.

**EXAMPLES:**
```python
sage: penta = polytopes.regular_polygon(5)
sage: outline = penta.projection().render_outline_2d()
sage: outline._objects[0]
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:**
```python
sage: cube1 = polytopes.hypercube(1)
sage: proj = cube1.projection()
sage: points = proj.render_points_1d()
sage: points._objects
[Point set defined by 2 point(s)]
```
**render_points_2d(**kwds)**

Return the points of a polyhedron in 2d.

**EXAMPLES:**

```python
sage: hex = polytopes.regular_polygon(6)
sage: proj = hex.projection()
sage: hex_points = proj.render_points_2d()
sage: hex_points._objects
[Point set defined by 6 point(s)]
```

**render_solid_3d(**kwds)**

Return solid 3d rendering of a 3d polytope.

**EXAMPLES:**

```python
sage: p = polytopes.hypercube(3).projection()
sage: p_solid = p.render_solid_3d(opacity = .7)
sage: type(p_solid)
<type 'sage.plot.plot3d.index_face_set.IndexFaceSet'>
```

**render_vertices_3d(**kwds)**

Return the 3d rendering of the vertices.

**EXAMPLES:**

```python
sage: p = polytopes.cross_polytope(3)
sage: proj = p.projection()
sage: verts = proj.render_vertices_3d()
sage: verts.bounding_box()
((-1.0, -1.0, -1.0), (1.0, 1.0, 1.0))
```

**render_wireframe_3d(**kwds)**

Return the 3d wireframe rendering.

**EXAMPLES:**

```python
sage: cube = polytopes.hypercube(3)
sage: cube_proj = cube.projection()
sage: wire = cube_proj.render_wireframe_3d()

FCylinder base -1.0 1.0 -1.0 apex -1.0 -1.0 -1.0 rad 0.005 texture...
```

**schlegel (projection_direction=None, height=1.1)**

Return the Schlegel projection.

- The polyhedron is translated such that its `center()` is at the origin.
- The vertices are then normalized to the unit sphere.
- The normalized points are stereographically projected from a point slightly outside of the sphere.

**INPUT:**

- `projection_direction` – coordinate list/tuple/iterable or `None` (default). The direction of the Schlegel projection. For a full-dimensional polyhedron, the default is the first facet normal; otherwise, the vector consisting of the first n primes is chosen.
- `height` – float (default: 1.1). How far outside of the unit sphere the focal point is.

**EXAMPLES:**
sage: cube4 = polytopes.hypercube(4)
sage: from sage.geometry.polyhedron.plot import Projection
sage: Projection(cube4).schlegel([1,0,0,0])
The projection of a polyhedron into 3 dimensions
sage: _.plot()
Graphics3d Object

**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 $(1,0,\ldots,0)$

**EXAMPLES:**

```python
sage: from sage.geometry.polyhedron.plot import Projection
sage: proj = Projection(polytopes.buckyball())
#long time
sage: proj.stereographic([5,2,3]).plot()
#long time
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
```

**tikz** *(view=[0,0,1], angle=0, scale=2, edge_color='blue!95!black', facet_color='blue!95!black', opacity=0.8, vertex_color='green', axis=False)*

Return a string `tikz_pic` consisting of a tikz picture of `self` according to a projection `view` and an angle `angle` obtained via Jmol through the current state property.

**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: 2) specifying the scaling of the tikz picture.
- **edge_color** - string (default: ‘blue!95!black’) representing colors which tikz recognize.
- **facet_color** - string (default: ‘blue!95!black’) representing colors which tikz recognize.
- **vertex_color** - string (default: ‘green’) representing colors which tikz recognize.
- **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:**

- LatexExpr – containing the TikZ picture.

**Note:** The inputs `view` and `angle` can 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:**

```python
sage: P1 = polytopes.small_rhombicuboctahedron()
sage: Image1 = P1.projection().tikz([1,3,5], 175, scale=4)
sage: type(Image1)
<class 'sage.misc.latex.LatexExpr'>
sage: print('
'.join(Image1.splitlines()[:4]))
\begin{tikzpicture}%
[x={(-0.939161cm, 0.244762cm)},
 y={ (0.097442cm, -0.482887cm)},
 z={ (0.329367cm, 0.840780cm)},
sage: _ = open('polytope-tikz1.tex', 'w').write(Image1)  # not tested
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)
sage: type(Image2)
<class 'sage.misc.latex.LatexExpr'>
sage: print('
'.join(Image2.splitlines()[:4]))
\begin{tikzpicture}[
scale=3.000000,
back/.style={loosely dotted, thin},
edge/.style={color=blue!95!black, thick},
sage: _ = open('polytope-tikz2.tex', 'w').write(Image2)  # not tested
sage: P3 = Polyhedron( vertices=[[-1, -1, 2],[-1, 2, -1],[2, -1, -1]])
sage: 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, vertex_color='yellow', axis=True)
sage: print('
'.join(Image3.splitlines()[:4]))
\begin{tikzpicture}%
[x={(0.658184cm, -0.242192cm)},
 y={(-0.096240cm, 0.912008cm)},
 z={(-0.746680cm, -0.331036cm)},
sage: _ = open('polytope-tikz3.tex', 'w').write(Image3)  # not tested
sage: P=Polyhedron( vertices=[[1,1,0,0],[1,2,0,0],[2,1,0,0],[0,0,1,0],[0,0,0,1]])
sage: P
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 5 vertices
sage: P.projection().tikz()
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.
```python
sage: P=Polyhedron(vertices=[[1,1,0,0],[1,2,0,0],[2,1,0,0],[0,0,1,0],[0,0,0,1]])
sage: P
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 5 vertices
```

```python
sage: P.projection().tikz()
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.

```python
class sage.geometry.polyhedron.plot.ProjectionFuncSchlegel(projection_direction, height=1.1, center=0)
The Schlegel projection from the given input point.

EXAMPLES:

```python
sage: from sage.geometry.polyhedron.plot import ProjectionFuncSchlegel
sage: proj = ProjectionFuncSchlegel([2,2,2])
sage: proj(vector([1.1,1.1,1.11]))[0]
0.0302...
```

```python
class sage.geometry.polyhedron.plot.ProjectionFuncStereographic(projection_point)
The stereographic (or perspective) projection.

EXAMPLES:

```python
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[0]
(-0.0486511..., 0.0859565...)
```

```python
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:

```python
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),
...]
```

(continues on next page)
A vertex at (-1, 0),
A vertex at (0, 1)

Rays are allowed, too:

\begin{verbatim}
    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 1 0]
    [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()
    [0 1 0 0 0]
    [1 0 1 0 0]
    [0 1 0 1 0]
    [0 0 0 1 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)]
\end{verbatim}

:sage: sage.geometry.polyhedron.plot.projection_func_identity(x)
The identity projection.

**EXAMPLES:**

\begin{verbatim}
    sage: from sage.geometry.polyhedron.plot import projection_func_identity
    sage: projection_func_identity((1,2,3))
    [1, 2, 3]
\end{verbatim}
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:

```python
sage: P = polytopes.cross_polytope(3)
sage: P.faces(3)
(0,1,2,3,4,5),
sage: P.faces(2)
(0,1,2), (0,1,3), (0,2,4), (0,3,4), (1,3,4), (1,2,5)
sage: P.faces(1)
(0,1), (0,2), (1,2), (0,3), (1,3), (0,4), (2,4), (3,4), (2,5), (3,5), (4,5), (1,5)
```

or `face_lattice()` to get the whole face lattice as a poset:

```python
sage: P.face_lattice()
Finite lattice containing 28 elements with distinguished linear extension
```

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()`

```python
sage: face = P.faces(1)[3]; face
<0,3>
```

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:

```python
sage: face.as_polyhedron()
A 1-dimensional polyhedron in ZZ^3 defined as the convex hull of 2 vertices
```

```python
class sage.geometry.polyhedron.face.PolyhedronFace(polyhedron, V_indices, H_indices)
    Bases: sage.structure.sage_object.SageObject

    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
<0,1,2>
sage: face.dim()
2
sage: face.ambient_Hrepresentation()
(An inequality (1, 1, 1) x + 1 >= 0,)

ambient_Hrepresentation (index=None)
Return the H-representation objects of the ambient polytope defining the face.

INPUT:

• index – optional. Either an 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():
    ...:     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 (0, 1) x + 1 >= 0, An inequality (-1, 0) 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)
()

ambient_Vrepresentation (index=None)
Return the V-representation objects of the ambient polytope defining the face.

INPUT:

• index – optional. Either an 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:

```python
sage: square = polytopes.hypercube(2)
sage: for fl in square.face_lattice():
    ....:     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))
```

`ambient_dim()`
Return the dimension of the containing polyhedron.

EXAMPLES:

```python
sage: P = Polyhedron(vertices = [[1,0,0,0],[0,1,0,0]])
sage: face = P.faces(1)[0]
sage: face.ambient_dim()
4
```

`as_polyhedron()`
Return the face as an independent polyhedron.

OUTPUT:
A polyhedron.

EXAMPLES:

```python
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]
sage: face
<0,3,4>
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
```

`dim()`
Return the dimension of the face.

OUTPUT:
Integer.

EXAMPLES:

```python
sage: fl = polytopes.dodecahedron().face_lattice()
sage: [ x.dim() for x in fl ]
[-1, 0, 0, 0, 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, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```
line_generator()

Return a generator for the lines of the face.

EXAMPLES:

```python
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)
```

lines()

Return all lines of the face.

OUTPUT:

A tuple of lines.

EXAMPLES:

```python
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),)
```

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:

```python
sage: p = polytopes.cross_polytope(4)
sage: face = p.face_lattice()[10]
sage: face
<0,2>
sage: face.ambient_Hrepresentation()
(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()
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:

```python
sage: p = polytopes.cross_polytope(4)
sage: face = p.face_lattice()[10]
sage: face
<0,2>
```

(continues on next page)
sage: face.ambient_Vrepresentation()
(A vertex at (-1, 0, 0, 0), A vertex at (0, 0, -1, 0))
sage: face.n_ambient_Vrepresentation()
2

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.faces(2)[3]
sage: face
<0,3,4>
sage: 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)[0]
sage: 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)[0]
sage: face.rays()
(A ray in the direction (1, 0, 0), A ray in the direction (0, 1, 0))

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.faces(1)[0]
sage: for v in face.vertex_generator(): print(v)
A vertex at (0, 1)
A vertex at (1, 0)
sage: type(face.vertex_generator())
<... 'generator'>

vertices()
Return all vertices of the face.

OUTPUT:
A tuple of vertices.

EXAMPLES:

\[
\begin{align*}
\text{sage: } & \text{triangle} = \text{Polyhedron(\text{vertices}=\{(1,0), (0,1), (1,1)\})} \\
\text{sage: } & \text{face} = \text{triangle.faces(1)[0]} \\
\text{sage: } & \text{face.vertices()} \\
& (\text{A vertex at (0, 1), A vertex at (1, 0)})
\end{align*}
\]

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:

\[
\begin{align*}
\text{sage: from sage.geometry.polyhedron.cdd_file_format import cdd_Hrepresentation} \\
\text{sage: print(cdd_Hrepresentation('rational', None, \{[0,1]\}) } \\
'H-representation\nlinearity 1 1\nbegin\n 1 2 rational\n 0 1\nend'
\end{align*}
\]

`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:

\[
\begin{align*}
\text{sage: from sage.geometry.polyhedron.cdd_file_format import cdd_Vrepresentation} \\
\text{sage: print(cdd_Vrepresentation('rational', \{(0,0), (1,0), (1,1)\}) } \\
V-representation\nlinearity 1 1\nbegin\n 3 3 rational\n 0 0 1\n 0 1 0\n 1 0 0\nend
\end{align*}
\]
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 \( \mathbb{R}^n \). The dimension \( n \) of a polytope is the smallest \( n \) such that the polytope can be embedded in \( \mathbb{R}^n \).

A *lattice polytope* is a polytope whose vertices all have integer coordinates. If \( L \) is a lattice polytope, the dual polytope of \( L \) is

\[
\{ y \in \mathbb{Z}^n : x \cdot y \geq -1 \text{ all } x \in L \}
\]

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 (2007-01-15): 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
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` – an 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:**

```python
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
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
```

We draw a pretty picture of the polytope in 3-dimensional space:

```python
sage: p.plot3d().show()
```

Now we add an extra point, which is in the interior of the polytope...

```python
sage: points.append((0,0,0))
sage: p = LatticePolytope(points)
sage: p.nvertices()
6
```
You can suppress vertex computation for speed but this can lead to mistakes:

```python
sage: p = LatticePolytope(points, compute_vertices=False)
...
sage: p.nvertices()
7
```

Given points must be in the lattice:

```python
sage: LatticePolytope([[1/2], [3/2]])
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:

```python
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
```

An empty lattice polytope can be considered as well:

```python
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()
((-1-d lattice polytope in 3-d lattice M,),)
```

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() ()
sage: face = o.faces(1)[0]
sage: face.adjacent() (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 \(a \cdot P + b\), 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:

```python
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()
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
```

While you can use rational transformation, the result must be integer:
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: 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

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()  
()  

But each of its other faces is contained in one or more facets:

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
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]
sage: face.ambient_ordered_point_indices()
(5, 8, 4, 9, 10, 11, 6, 12, 7)
sage: cube.points(face.ambient_ordered_point_indices())
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]
sage: face.ambient_point_indices()
(4, 5, 6, 7, 8, 9, 10, 11, 12)
sage: cube.points(face.ambient_point_indices()) == face.points()
True
```

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]
sage: face.ambient_vertex_indices()
(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()
N( 1, 1),
N( 1, -1),
N(-1, -1),
N(-1, 1),
N(-1, 0),
N( 0, -1),
N( 0, 1),
N( 1, 0)
in 2-d lattice N
```

```
sage: square.boundary_point_indices()
(0, 1, 2, 3, 4, 5, 7, 8)
```

For an edge the boundary is formed by the end points:

```
sage: face = square.edges()[0]
sage: face.points()
N(-1, -1),
N(-1, 1),
N(-1, 0)
in 2-d lattice N
```

```
sage: face.boundary_point_indices()
(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()
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]
sage: face.boundary_points()
N(-1, -1),
N(-1, 1),
N(-1, 0)
```

(continues on next page)
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:

```python
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
```

dim()

Return the dimension of this polytope.

EXAMPLES:

We create a 3-dimensional octahedron and check its dimension:

```python
sage: o = lattice_polytope.cross_polytope(3)
sage: o.dim()
3
```

Now we create a 2-dimensional diamond in a 3-dimensional space:

```python
sage: p = LatticePolytope([(1,0,0), (0,1,0), (-1,0,0), (0,-1,0)])
sage: p.dim()
2
sage: 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:

```python
sage: o = lattice_polytope.cross_polytope(3)
sage: o.distances()

[2 0 0 0 2 2 1]
[2 2 0 0 2 2 1]
```

(continues on next page)
Distances from facets to the point (1,2,3):

```
sage: o.distances([1,2,3])
(-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)])
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()
[2 2 0 0 1]
[2 0 2 0 1]
[0 0 2 2 1]
[0 2 0 2 1]
[0 2 2 0 1]
sage: p.distances((1/2, 3, 0))
(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))
(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: 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
```
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, $\mathbb{Z}^n$ is returned, where $n$ is the dimension of self.

EXAMPLES:

```sage
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
```

edges()

Return edges (faces of dimension 1) of self.

OUTPUT:

* tuple of lattice polytopes.

EXAMPLES:

```sage
sage: o = lattice_polytope.cross_polytope(3)
sage: o.edges()
(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())
12
```

edges_lp(*args, **kwds)

Deprecated: Use `edges()` instead. See trac ticket #22122 for details.

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
sage: square = LatticePolytope([(0,0), (1,0), (1,1), (0,1)])
sage: L = square.face_lattice()
sage: L
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)
[-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]

For a particular face you can look at its actual vertices...

sage: face = L.level_sets()[1][0]
sage: face.vertices()
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()
(0,)
sage: square.vertex(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]
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]
sage: D = L.hasse_diagram()
sage: D.neighbors(face)
[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: 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,
Note that if \( p \) is a face of \( \text{superp} \), then the face lattice of \( p \) consists of (appropriate) faces of \( \text{superp} \):

```python
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
```

**faces** \((\text{dim=\text{None}, codim=\text{None}})\)

Return faces of \( \text{self} \) of specified (co)dimension.

**INPUT:**

- \( \text{dim} \) – integer, dimension of the requested faces;
- \( \text{codim} \) – 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 \( \text{dim} \) or \( \text{codim} \) is given, the output will be a tuple of lattice polytopes;
- if neither \( \text{dim} \) nor \( \text{codim} \) 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:

```python
sage: square = LatticePolytope([(0,0), (1,0), (1,1), (0,1)])
sage: square.faces()
((-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),
  (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 dimension one (i.e., edges):

```python
sage: square.faces(dim=1)
(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):

```python
sage: square.faces(codim=1) is square.faces(dim=1)
True
```

Let’s pick a particular face:

```python
sage: face = square.faces(dim=1)[0]
```

Now you can look at the actual vertices of this face...

```python
sage: face.vertices()
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:

```python
sage: face.ambient_vertex_indices()
(0, 3)
sage: square.vertices(face.ambient_vertex_indices())
M(0, 0),
M(0, 1)
in 2-d lattice M
```

```python
faces_lp(*args, **kwds)
```

Deprecated: Use `faces()` instead. See trac ticket #22122 for details.

```python
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()`, `faces()`.

EXAMPLES:
sage: o = lattice_polytope.cross_polytope(3)
sage: o.facet_constant(0)
1
sage: o.facet_constant(0) == o.facet_constants()[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:**

- an integer vector

**See also:**

*facet_constant(), facet_normal(), facetNormals(), facets().*

**EXAMPLES:**

For reflexive polytopes all constants are 1:

```python
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)
```

Here is an example of a 3-dimensional polytope in a 4-dimensional space with 3 facets containing the origin:

```python
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)
```

**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:

```python
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
```

**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:

```python
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
```

Here is an example of a 3-dimensional polytope in a 4-dimensional space:

```python
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()
```

(continues on next page)
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):

\[
\begin{bmatrix}
0 & 6 & 0 & 0 \\
0 & 0 & 2 & 0 \\
3 & 0 & 0 & 0 \\
0 & 0 & 0 & 6
\end{bmatrix}
\]

---

**facet_of()**

Return elements of the ambient face lattice having `self` as a facet.

**OUTPUT:**

- tuple of lattice polytopes.

**EXAMPLES:**

```
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
```

**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()
(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())
8
```

**facets_lp(*args, **kwds)**

Deprecated: Use `facets()` instead. See trac ticket #22122 for details.

**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:

```python
d = lattice_polytope.cross_polytope(2)
d.index()  # 3
```

Note that polytopes with the same index are not necessarily the same:

```python
d.vertices()  # M(1, 0), M(0, 1), M(-1, 0), M(0, -1) in 2-d lattice M
d.flip().vertices()  # M(1, 0), M(0, 1), M(0, -1), M(-1, 0) in 2-d lattice M
```

But they are in the same $GL(Z^n)$ orbit and have the same normal form:

```python
d.normal_form()  # M(1, 0), M(0, 1), M(0, -1), M(-1, 0) in 2-d lattice M
d.flip().normal_form()  # 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:

```python
square = lattice_polytope.cross_polytope(2).polar()
square.points()  # N(1, 1),
```
N( 1, -1),
N(-1, -1),
N(-1, 1),
N(-1, 0),
N( 0, -1),
N( 0, 0),
N( 0, 1),
N( 1, 0)
in 2-d lattice N
sage: square.interior_point_indices()
(6,)

Its edges also have a single interior point each:

sage: face = square.edges()[0]
sage: face.interior_point_indices()
(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()
N(0, 0)
in 2-d lattice N

Its edges also have a single interior point each:

sage: face = square.edges()[0]
sage: face.interior_points()
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

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

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

**lattice()**

Return the ambient lattice of self.

OUTPUT:

- a lattice.

EXAMPLES:

```python
sage: lattice_polytope.cross_polytope(3).lattice()
3-d lattice M
```  

**lattice_dim()**

Return the dimension of the ambient lattice of self.

OUTPUT:

- integer.

EXAMPLES:

```python
sage: p = LatticePolytope([(1,0)])
sage: p.lattice_dim()
2
sage: p.dim()
0
```  

**linearly_independent_vertices()**

Return a maximal set of linearly independent vertices.

OUTPUT:

A tuple of vertex indices.

EXAMPLES:

```python
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,)
```  

**nef_partitions(keep_symmetric=False, keep_products=True, keep_projections=True, hodge_numbers=False)**

Return 2-part nef-partitions of self.
INPUT:

- **keep_symmetric** – (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** – (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** – (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** – (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:

```python
sage: p = lattice_polytope.cross_polytope(4)
sage: p.nef_partitions()
\[
Nef-partition \{0, 1, 4, 5\} U \{2, 3, 6, 7\} (direct product),
Nef-partition \{0, 1, 2, 4\} U \{3, 5, 6, 7\},
Nef-partition \{0, 1, 2, 4, 5\} U \{3, 6, 7\},
Nef-partition \{0, 1, 2, 4, 5, 6\} U \{3, 7\} (direct product),
Nef-partition \{0, 1, 2, 3\} U \{4, 5, 6, 7\},
Nef-partition \{0, 1, 2, 3, 4\} U \{5, 6, 7\},
Nef-partition \{0, 1, 2, 3, 4, 5\} U \{6, 7\},
Nef-partition \{0, 1, 2, 3, 4, 5, 6\} U \{7\} (projection)
\]
```

Now we omit projections:

```python
sage: p.nef_partitions(keep_projections=False)
\[
Nef-partition \{0, 1, 4, 5\} U \{2, 3, 6, 7\} (direct product),
Nef-partition \{0, 1, 2, 4\} U \{3, 5, 6, 7\},
Nef-partition \{0, 1, 2, 4, 5\} U \{3, 6, 7\},
Nef-partition \{0, 1, 2, 4, 5, 6\} U \{3, 7\} (direct product),
Nef-partition \{0, 1, 2, 3\} U \{4, 5, 6, 7\},
Nef-partition \{0, 1, 2, 3, 4\} U \{5, 6, 7\},
Nef-partition \{0, 1, 2, 3, 4, 5\} U \{6, 7\}
\]
```

Currently Hodge numbers cannot be computed for a given nef-partition:

```python
sage: p.nef_partitions()[1].hodge_numbers()
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)
[
    Nef-partition \{0, 1, 4, 5\} U \{2, 3, 6, 7\} (direct product),
    Nef-partition \{0, 1, 2, 4\} U \{3, 5, 6, 7\},
    Nef-partition \{0, 1, 2, 4, 5\} U \{3, 6, 7\},
    Nef-partition \{0, 1, 2, 4, 5, 6\} U \{3, 7\} (direct product),
    Nef-partition \{0, 1, 2, 3\} U \{4, 5, 6, 7\},
    Nef-partition \{0, 1, 2, 3, 4\} U \{5, 6, 7\},
    Nef-partition \{0, 1, 2, 3, 4, 5\} U \{6, 7\},
    Nef-partition \{0, 1, 2, 3, 4, 5, 6\} U \{7\} (projection)
]

Now it is possible to get Hodge numbers:

sage: p.nef_partitions(hodge_numbers=True)[1].hodge_numbers()
(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()
(20,)

We illustrate removal of symmetric partitions on a diamond:

sage: p = lattice_polytope.cross_polytope(2)
sage: p.nef_partitions()
[
    Nef-partition \{0, 2\} U \{1, 3\} (direct product),
    Nef-partition \{0, 1\} U \{2, 3\},
    Nef-partition \{0, 1, 2\} U \{3\} (projection)
]
sage: p.nef_partitions(keep_symmetric=True)
[
    Nef-partition \{0, 1, 3\} U \{2\} (projection),
    Nef-partition \{0, 2, 3\} U \{1\} (projection),
    Nef-partition \{0, 3\} U \{1, 2\},
    Nef-partition \{1, 2, 3\} U \{0\} (projection),
    Nef-partition \{1, 3\} U \{0, 2\} (direct product),
    Nef-partition \{2, 3\} U \{0, 1\},
    Nef-partition \{0, 1, 2\} U \{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()
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 - a 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")
sage: s
# output contains random time
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
```

The number of facets of an interval is 2:

```
sage: LatticePolytope(([1],[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
```

**normal_form(algorithm='palp', permutation=False)**

Return the normal form of vertices of self.

Two full-dimensional lattice polytopes are in the same $GL(\mathbb{Z})$-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”) The algorithm which is used to compute the normal form. Options are:
  - “palp” – Run external PALP code, usually the fastest option.
  - “palp_native” – The original PALP algorithm implemented in sage. Currently considerably slower than PALP.
  - “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—(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()
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()
3
sage: d
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
```

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
```

We can perform the same examples using other algorithms:

```
sage: o = lattice_polytope.cross_polytope(2)
sage: o.normal_form(algorithm="palp_native")
M( 1, 0),
M( 0, 1),
M( 0, -1),
M(-1, 0)
```
in 2-d lattice M

```python
sage: o = lattice_polytope.cross_polytope(2)
sage: o.normal_form(algorithm="palp_modified")
M( 1, 0),
M( 0, 1),
M( 0, -1),
M(-1, 0)
in 2-d lattice M
```

**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:

```python
sage: o = lattice_polytope.cross_polytope(3)
sage: o.npoints()
7
sage: cube = o.polar()
sage: cube.npoints()
27
```

**nvertices()**

Return the number of vertices of this polytope.

**EXAMPLES:** The number of vertices of the 3-dimensional octahedron and its polar cube:

```python
sage: o = lattice_polytope.cross_polytope(3)
sage: o.nvertices()
6
sage: cube = o.polar()
sage: 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:**

```python
sage: p = lattice_polytope.cross_polytope(2)
sage: p.origin()
4
sage: p.point(p.origin())
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
```

Now we make sure that the origin of non-full-dimensional polytopes can be identified correctly (trac ticket #10661):

2.2. Lattice polyhedra
```
sage: LatticePolytope([(1,0,0), (-1,0,0)]).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
```

**plot3d**

```
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), 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:(1,0,0))
- `facet_colors` - (default:None) if specified, must be a list of colors for each facet separately, used instead of `facet_color`
- `show_edges` - (default:True) whether to draw edges as lines
- `edge_thickness` - (default:3)
- `edge_color` - (default:(0.5,0.5,0.5))
- `show_vertices` - (default:True) whether to draw vertices as balls
- `vertex_size` - (default:10)
- `vertex_color` - (default:(1,0,0))
- `show_points` - (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()
Graphics3d Object
```

Plot without facets and points, shown without the frame:

```
sage: c.plot3d(show_facets=false,show_points=false).show(frame=False)
```

Plot with facets of different colors:

```
sage: c.plot3d(facet_colors=rainbow(c.nfacets(), 'rgbtuple'))
```

It is also possible to plot lower dimensional polytopes in 3D (let’s also change labels of vertices):

```
sage: lattice_polytope.cross_polytope(2).plot3d(vlabels=['A', 'B', 'C', 'D'])
```

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),
M( 0, 0, -1)
in 3-d lattice M
sage: o.point(1)
M(0, 1, 0)
The only other point in the octahedron is the origin:

```
sage: o.point(6)
M(0, 0, 0)
```

points(*args, **kwds)

Return all lattice points of self.

INPUT:

• any arguments given will be passed on to the returned object.
• a point collection.

EXAMPLES:

Lattice points of the octahedron and its polar cube:

```python
sage: o = lattice_polytope.cross_polytope(3)
sage: o.points()
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()
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, 0),
N( 0, -1, -1),
N( 0, 0, -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:

```python
sage: p = LatticePolytope([(1,0,0), (0,1,0), (-1,0,0), (0,-1,0)])
sage: p.points()
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)
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
```

def 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()
sage: 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
```

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!
```

def poly_x(keys, reduce_dimension=False)
    Run poly.x with given keys on vertices of this polytope.

    **INPUT:**
    
    - keys - a string of options passed to poly.x. The key “f” is added automatically.
    
    - reduce_dimension - (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"))
8 3 Vertices of P-dual <-> Equations of P
-1 -1 1
1 -1 1
-1 1 1
1 1 1
-1 -1 -1
```

(continues on next page)
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")  # possibly different output depending on your system
Traceback (most recent call last):
  ... 
ValueError: Error executing 'poly.x -fe' for the given polytope!
Output:
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")
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))
4 2 Equations of P
-1 1 0
1 1 2
-1 -1 0
1 -1 2
```

### polyhedron()
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
```

### 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()
```

### skeleton()
Return the graph of the one-skeleton of this polytope.
EXAMPLES:

```python
sage: d = lattice_polytope.cross_polytope(2)
sage: g = d.skeleton()
sage: g
Graph on 4 vertices
sage: g.edges()
[(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:

```python
sage: o = lattice_polytope.cross_polytope(3)
sage: c = o.polar()
sage: c.skeleton_points()
[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:

```python
sage: c.skeleton_points(k=1)
[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:

```python
sage: c.skeleton_points(k=0)
[0, 1, 2, 3, 4, 5, 6, 7]
```

2-skeleton lists all points except for the origin (point #17):

```python
sage: c.skeleton_points(k=2)
[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:

```python
sage: c.skeleton_points(k=3)
[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:

```python
sage: c.skeleton_points(k=100)
[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:

```python
sage: o = lattice_polytope.cross_polytope(3)
sage: o.skeleton_show([1,2,4])
```
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
```

**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()
[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)
```

**vertex_facet_pairing_matrix**

Return the vertex facet pairing matrix \( P \cdot M \).

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 2 2]
[2 2 0 0 2]
[2 2 0 0 0]
[2 0 2 0 2]
[0 0 2 2 0]
[0 0 2 2 2]
[0 2 0 2 0]
[0 2 2 2 0]
[0 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)
in 3-d lattice N
```

```
class sage.geometry.lattice_polytope.NefPartition(data, Delta_polar, check=True)
Bases: sage.structure.sage_object.SageObject, _abcoll.Hashable

Create a nef-partition.

INPUT:

- `data` – a 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 $\Delta \subset M_{\mathbb{R}}$ be a reflexive polytope with polar $\Delta^\circ \subset N_{\mathbb{R}}$. Let $X_\Delta$ be the toric variety associated to the normal fan of $\Delta$. A **nef-partition** is a decomposition of the vertex set $V$ of $\Delta^\circ$ into a disjoint union $V = V_0 \sqcup V_1 \sqcup \cdots \sqcup V_{k-1}$ 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 $\Delta^\circ$). Equivalently, let $\nabla_i \subset N_{\mathbb{R}}$ be the convex hull of vertices from $V_i$ and the origin. These polytopes form a nef-partition if their Minkowski sum $\nabla \subset N_{\mathbb{R}}$ is a reflexive polytope.

The **dual nef-partition** is formed by polytopes $\Delta_i \subset M_{\mathbb{R}}$ of $E_i$, which give a decomposition of the vertex set of $\nabla^\circ \subset M_{\mathbb{R}}$ and their Minkowski sum is $\Delta$, i.e. the polar duality of reflexive polytopes switches convex hull
and Minkowski sum for dual nef-partitions:
\[
\Delta^o = \text{Conv} (\nabla_0, \nabla_1, \ldots, \nabla_{k-1}),
\]
\[
\nabla = \nabla_0 + \nabla_1 + \cdots + \nabla_{k-1},
\]
\[
\Delta = \Delta_0 + \Delta_1 + \cdots + \Delta_{k-1},
\]
\[
\nabla^o = \text{Conv} (\Delta_0, \Delta_1, \ldots, \Delta_{k-1}).
\]

One can also interpret the duality of nef-partitions as the duality of the associated cones. Below \(\mathcal{M} = M \times \mathbb{Z}^k\) and \(\mathcal{N} = N \times \mathbb{Z}^k\) are dual lattices.

The Cayley polytope \(P \subset \mathcal{M}_R\) of a nef-partition is given by \(P = \text{Conv}(\Delta_0 \times e_0, \Delta_1 \times e_1, \ldots, \Delta_{k-1} \times e_{k-1})\), where \(\{e_i\}_{i=0}^{k-1}\) is the standard basis of \(\mathbb{Z}^k\). The dual Cayley polytope \(P^* \subset \mathcal{N}_R\) is the Cayley polytope of the dual nef-partition.

The Cayley cone \(C \subset \mathcal{M}_R\) of a nef-partition is the cone spanned by its Cayley polytope. The dual Cayley cone \(C^\vee \subset \mathcal{M}_R\) 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 0-th and 1-st vertices belonging to the 0-th part (recall that numeration in Sage starts with 0), the 2-nd and 5-th vertices belonging to the 1-st part, and 3-rd and 4-th vertices belonging to the 2-nd part:

```python
sage: o = lattice_polytope.cross_polytope(3)
sage: np = NefPartition([0,0,1,2,2,1], o)
sage: np
Nef-partition {0, 1} U {2, 5} U {3, 4}
```

The octahedron plays the role of \(\Delta^o\) in the above description:

```python
sage: np.Delta_polar() is o
True
```

The dual nef-partition (corresponding to the “mirror complete intersection”) gives decomposition of the vertex set of \(\nabla^o\):

```python
sage: np.dual()
Nef-partition {0, 1, 2} U {3, 4} U {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),
N( 1,  0, 0),
N( 0,  1, 0),
N( 1,  1, 0)
in 3-d lattice N
```

Of course, \(\nabla^o\) is \(\Delta^o\) from the point of view of the dual nef-partition:
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):

```python
sage: o.nef_partitions()
[  
  Nef-partition {0, 1, 3} U {2, 4, 5},
  Nef-partition {0, 1, 3, 4} U {2, 5} (direct product),
  Nef-partition {0, 1, 2} U {3, 4, 5},
  Nef-partition {0, 1, 2, 3} U {4, 5},
  Nef-partition {0, 1, 2, 3, 4} U {5} (projection)
]
```

**Delta** *(i=None)*

Return the polytope $\Delta$ or $\Delta_i$ corresponding to self.

**INPUT:**

- i – an integer. If not given, $\Delta$ will be returned.

**OUTPUT:**

- a lattice polytope.

See `nef-partition` class documentation for definitions and notation.

**EXAMPLES:**

```python
sage: o = lattice_polytope.cross_polytope(3)
sage: np = o.nef_partitions()[0]
sage: np
Nef-partition {0, 1, 3} U {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) 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
```
Delta_polar()
Return the polytope $\Delta^o$ corresponding to self.

OUTPUT:

• a lattice polytope.

See nef-partition class documentation for definitions and notation.

EXAMPLES:

```python
sage: o = lattice_polytope.cross_polytope(3)
sage: np = o.nef_partitions()[0]
sage: np
Nef-partition {0, 1, 3} U {2, 4, 5}
sage: np.Delta_polar() is o
True
```

Deltas()
Return the polytopes $\Delta_i$ corresponding to self.

OUTPUT:

• a tuple of lattice polytopes.

See nef-partition class documentation for definitions and notation.

EXAMPLES:

```python
sage: o = lattice_polytope.cross_polytope(3)
sage: np = o.nef_partitions()[0]
sage: np
Nef-partition {0, 1, 3} U {2, 4, 5}
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: [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, -1, 0),
N( 1, 0, 0),
N(-1, 0, 0),
N( 0, 1, -1),
N( 0, 1, 1),
```

(continues on next page)
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 \( \nabla \) will look like \( \{0, 1, 2\} \cup \{3, 4, 5, 6\} \cup \{7, 8\} \).

EXAMPLES:

```sage
sage: o = lattice_polytope.cross_polytope(3)
sage: np = o.nef_partitions()[0]
sage: np
Nef-partition {0, 1, 3} U {2, 4, 5}
sage: np.dual()
Nef-partition {0, 1, 2, 3} U {4, 5, 6, 7}
sage: np.dual().Delta() is np.nabla()
True
sage: np.dual().nabla(0) is np.Delta(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
sage: p = lattice_polytope.cross_polytope(5)
sage: np = p.nef_partitions()[0]  # long time (4s on sage.math, 2011)
sage: np.hodge_numbers()  # long time
Traceback (most recent call last):
... NotImplementedError: use nef_partitions(hodge_numbers=True)!
sage: np = p.nef_partitions(hodge_numbers=True)[0]  # long time (13s on sage.math, 2011)
sage: np.hodge_numbers()  # long time
(19, 19)
```

nabla(i=None)
Return the polytope \( \nabla \) or \( \nabla_i \) corresponding to self.

INPUT:
• i – an integer. If not given, ∇ will be returned.

OUTPUT:

• a lattice polytope.

See `nef-partition` class documentation for definitions and notation.

EXAMPLES:

```python
sage: o = lattice_polytope.cross_polytope(3)
sage: np = o.nef_partitions()[0]
sage: np
Nef-partition \{0, 1, 3\} U \{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
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
```

`nabla_polar()`

Return the polytope $\nabla^\circ$ corresponding to `self`.

OUTPUT:

• a lattice polytope.

See `nef-partition` class documentation for definitions and notation.

EXAMPLES:

```python
sage: o = lattice_polytope.cross_polytope(3)
sage: np = o.nef_partitions()[0]
sage: np
Nef-partition \{0, 1, 3\} U \{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),
(continues on next page)
```
N(0, 0, -1)
in 3-d lattice N
\texttt{sage: np.nabla_polar()} is \texttt{np.dual().Delta_polar()}
True

\texttt{nablas()}
Return the polytopes $\nabla_i$ corresponding to \texttt{self}.

**OUTPUT:**

- a tuple of \texttt{lattice polytopes}.

See \texttt{nef-partition} class documentation for definitions and notation.

**EXAMPLES:**

\texttt{sage: o = lattice_polytope.cross_polytope(3)}
\texttt{sage: np = o.nef_partitions()[0]}
\texttt{sage: np}
Nef-partition \{0, 1, 3\} U \{2, 4, 5\}
\texttt{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
\texttt{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]

\texttt{nparts()}
Return the number of parts in \texttt{self}.

**OUTPUT:**

- an integer.

**EXAMPLES:**

\texttt{sage: o = lattice_polytope.cross_polytope(3)}
\texttt{sage: np = o.nef_partitions()[0]}
\texttt{sage: np}
Nef-partition \{0, 1, 3\} U \{2, 4, 5\}
\texttt{sage: np.nparts()}
2

\texttt{part}(i, all_points=False)
Return the \texttt{i}-th part of \texttt{self}.

**INPUT:**

- \texttt{i} – an integer
• \texttt{all_points} – (default: False) whether to list all lattice points or just vertices

\textbf{OUTPUT:}

• a tuple of integers, indices of vertices (or all lattice points) of $\Delta^\circ$ belonging to $V_i$.

See \texttt{nef-partition} class documentation for definitions and notation.

\textbf{EXAMPLES:}

```python
sage: o = lattice_polytope.cross_polytope(3)
sage: np = o.nef_partitions()[0]
sage: np
Nef-partition \{0, 1, 3\} U \{2, 4, 5\}
sage: np.part(0)
(0, 1, 3)
sage: np.part(0, all_points=True)
(0, 1, 3)
sage: np.dual().part(0)
(0, 1, 2, 3)
sage: np.dual().part(0, all_points=True)
(0, 1, 2, 3, 8)
```

\texttt{part_of}(\textit{i})

Return the index of the part containing the \textit{i}-th vertex.

\textbf{INPUT:}

• \textit{i} – an integer.

\textbf{OUTPUT:}

• an integer \textit{j} such that the \textit{i}-th vertex of $\Delta^\circ$ belongs to $V_j$.

See \texttt{nef-partition} class documentation for definitions and notation.

\textbf{EXAMPLES:}

```python
sage: o = lattice_polytope.cross_polytope(3)
sage: np = o.nef_partitions()[0]
sage: np
Nef-partition \{0, 1, 3\} U \{2, 4, 5\}
sage: np.part_of(3)
0
sage: np.part_of(2)
1
```

\texttt{part_of_point}(\textit{i})

Return the index of the part containing the \textit{i}-th point.

\textbf{INPUT:}

• \textit{i} – an integer.

\textbf{OUTPUT:}

• an integer \textit{j} such that the \textit{i}-th point of $\Delta^\circ$ belongs to $\nabla_j$.

\textbf{Note:} Since a nef-partition induces a partition on the set of boundary lattice points of $\Delta^\circ$, the value of \textit{j} is well-defined for all \textit{i} but the one that corresponds to the origin, in which case this method will raise a \texttt{ValueError} exception. (The origin always belongs to all $\nabla_j$.)
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]
sage: np
Nef-partition {1, 2, 5, 7, 8} U {0, 3, 4, 6}
sage: p.nvertices()
9
sage: p.npoints()
15
```

We see that the polytope has 6 more points in addition to vertices. One of them is the origin:

```
sage: p.origin()
14
sage: np.part_of_point(14)
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())
....:   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())
....:   if p.origin() != n and np.part_of_point(n) == 1]
[0, 3, 4, 6, 10, 12]
```

`parts(all_points=False)`

Return all parts of `self`.

INPUT:

- `all_points` – (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 $\Delta^\circ$ belonging to $V_i$

See `nef-partition` class documentation for definitions and notation.

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: np = o.nef_partitions()[0]
sage: np
Nef-partition {0, 1, 3} U {2, 4, 5}
sage: np.parts()
((0, 1, 3), (2, 4, 5))
sage: np.parts(all_points=True)
((0, 1, 3), (2, 4, 5))
sage: np.dual().parts()
```

(continues on next page)
((0, 1, 2, 3), (4, 5, 6, 7))
sage: np.dual().parts(all_points=True)
((0, 1, 2, 3, 8), (4, 5, 6, 7, 10))

sage.geometry.lattice_polytope.PPL_point(*args, **kwds)
Construct a point.

**INPUT:**
- expression – a `Linear_expression` or something convertible to it (`Variable` or integer).
- divisor – an integer.

**OUTPUT:**
A new `Generator` representing the point.

Raises a `ValueError` if `divisor==0`.

**Examples:**
```python
>>> from ppl import Generator, Variable
>>> y = Variable(1)
>>> Generator.point(2*y+7, 3)
point(0/3, 2/3)
>>> Generator.point(y+7, 3)
point(0/3, 1/3)
>>> Generator.point(7, 3)
point()
>>> Generator.point(0, 0)
Traceback (most recent call last):
...
ValueError: PPL::point(e, d):
d == 0.
```

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”:
```python
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
```
There are 16 reflexive polygons and numeration starts with 0:

```sage
sage: ReflexivePolytope(2,16)
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
sage: ReflexivePolytope(4,16)
Traceback (most recent call last):
...  
NotImplementedError: only 2- and 3-dimensional reflexive polytopes are available!
```

```
sage: len(ReflexivePolytopes(2))
16
```

It is not possible to load 4-dimensional polytopes in this way:

```sage
sage: ReflexivePolytopes(4)
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.

**Parameters** `dim` (2 or 3) – dimension of required reflexive polytopes

**Return type** list of lattice polytopes

**EXAMPLES:**

There are 16 reflexive polygons:

```sage
sage: len(ReflexivePolytopes(2))
16
```

### `sage.geometry.lattice_polytope.SetOfAllLatticePolytopesClass`

**Bases:** `sage.structure.parent.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:** 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
sage: o = lattice_polytope.cross_polytope(3)
sage: lattice_polytope.all_cached_data([o])
```

### `sage.geometry.lattice_polytope.all_facet_equations(polytopes)`

Compute polar polytopes for all reflexive and equations of facets for all non-reflexive polytopes.

2.2. Lattice polyhedra 159
all_facet_equations and all_polars are synonyms.

This function does it MUCH faster than member functions of LatticePolytope during the first run. So it is recommended to use this function if you work with big sets of data.

INPUT: 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])
sage: o.polar()
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 function does it MUCH faster than member functions of LatticePolytope during the first run. So it is recommended to use this function 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: 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])
sage: o.nef_partitions()
[     Nef-partition {0, 1, 3} U {2, 4, 5},
     Nef-partition {0, 1, 3, 4} U {2, 5} (direct product),
     Nef-partition {0, 1, 2} U {3, 4, 5},
     Nef-partition {0, 1, 2, 3} U {4, 5},
     Nef-partition {0, 1, 2, 3, 4} U {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])
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 function does it MUCH faster than member functions of LatticePolytope during the first run. So it is recommended to use this function if you work with big sets of data.

INPUT: 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])
sage: o.points()
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: 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])
sage: o.polar()
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 - a list that can be converted into vectors of the same dimension over ZZ.

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)]

sage.geometry.lattice_polytope.cross_polytope(dim)
Return a cross-polytope of the given dimension.

INPUT:

- dim – an 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

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)
False
sage: p = LatticePolytope([(1,0), (0,1), (-1,-1)])
sage: p
2-d reflexive polytope #0 in 2-d lattice M
sage: 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 is_NefPartition
sage: is_NefPartition(1)
False
sage: o = lattice_polytope.cross_polytope(3)
sage: np = o.nef_partitions()[0]
sage: np
Nef-partition {0, 1, 3} U {2, 4, 5}
sage: is_NefPartition(np)
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:

```python
sage: lattice_polytope.minkowski_sum([[1,0],[-1,0]],[[0,1],[0,-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 non-negative integer coefficients

EXAMPLES: This is a 3-dimensional reflexive polytope:

```python
sage: p = LatticePolytope([[1,0,0], [0,1,0], 
....: (-1,-1,0), (0,0,1), (-1,0,-1)])
sage: p.points()

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:

```python
sage: p.points().matrix().kernel().echelonized_basis_matrix()

[ 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 non-negative integers:

```python
sage: lattice_polytope.positive_integer_relations(p.points().column_matrix())

[1 0 1 1 0]
[1 1 0 0 0]
[0 0 0 0 1]
```

**sage.geometry.lattice_polytope.read_all_polytopes(file_name)**

Read all polytopes from the given file.

INPUT:
• `file_name` – a 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:

```python
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])
```

```python
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
```

```python
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]
```

```python
sage: 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` – (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:

```python
sage: lattice_polytope.read_palp_matrix("2 3 comment 
  1 2 3 
  4 5 6")
[1 2 3]
[4 5 6]
sage: lattice_polytope.read_palp_matrix("3 2 Will be transposed 
  1 2  
  3 4 
  5 6")
```

(continues on next page)
sage.geometry.lattice_polytope.set_palp_dimension \( (d) \)
Set the dimension for PALP calls to \( d \).

**INPUT:**

- \( d \) – an 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")
Traceback (most recent call last):
  ... ValueError: Error executing 'poly.x -fv' for the given polytope!
```

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")
'7 14 Vertices of P...'
```

Let's go back to default settings:

```
sage: 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: 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:
    ... (continues on next page)
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)
- \( comment \) – a 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:**

```python
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 -1 0 0
0 0 1 0 0 -1
sage: lattice_polytope.write_palp_matrix(o.vertices(), format="%d")
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

    Bases: sage.structure.sage_object.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 \( A \times x + b \) with \( A =
\begin{bmatrix}
  1 & 2 \\
  2 & 3 \\
-1 & 2
\end{bmatrix}
\) \b = \( (1, 2, 3) \)
sage: M._A
\begin{bmatrix}
  1 & 2 \\
  2 & 3 \\
-1 & 2
\end{bmatrix}
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 \( \mathbb{Z}^3 \) with 3 vertices
sage: _.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 \( A \times x + b \) with \( A =
\begin{bmatrix}
  1 & 2 \\
  2 & 3 \\
-1 & 2
\end{bmatrix}
\) \b = \( (1, 2, 3) \)
sage: 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:
domain_dim()

Return the dimension of the domain lattice

EXAMPLES:

```python
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
```

exception sage.geometry.polyhedron.lattice_euclidean_group_element.LatticePolytopeError

Bases: exceptions.Exception

Base class for errors from lattice polytopes

exception sage.geometry.polyhedron.lattice_euclidean_group_element.LatticePolytopeNoEmbeddingError

Bases: sage.geometry.polyhedron.lattice_euclidean_group_element.LatticePolytopeError

Raised when no embedding of the desired kind can be found.

exception sage.geometry.polyhedron.lattice_euclidean_group_element.LatticePolytopesNotIsomorphicError

Bases: sage.geometry.polyhedron.lattice_euclidean_group_element.LatticePolytopeError

Raised when two lattice polytopes are not isomorphic.

### 2.2.3 Access the PALP database(s) of reflexive lattice polytopes

EXAMPLES:

```python
sage: from sage.geometry.polyhedron.palp_database import PALPreader
sage: for lp in PALPreader(2):
....:     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)
```

(continues on next page)
4 \( \frac{t^2 + 6t + 1}{-t^3 + 3t^2 - 3t + 1} \)
4 \( \frac{t^2 + 6t + 1}{-t^3 + 3t^2 - 3t + 1} \)
4 \( \frac{t^2 + 4t + 1}{-t^3 + 3t^2 - 3t + 1} \)
4 \( \frac{t^2 + 3t + 1}{-t^3 + 3t^2 - 3t + 1} \)
5 \( \frac{t^2 + 3t + 1}{-t^3 + 3t^2 - 3t + 1} \)
5 \( \frac{t^2 + 5t + 1}{-t^3 + 3t^2 - 3t + 1} \)
5 \( \frac{t^2 + 4t + 1}{-t^3 + 3t^2 - 3t + 1} \)
6 \( \frac{t^2 + 4t + 1}{-t^3 + 3t^2 - 3t + 1} \)

**class** `sage.geometry.polyhedron.palp_database.PALPreader(dim, data_basename=None, output='Polyhedron')`

**Bases:** `sage.structure.sage_object.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:**

```python
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, 9), (4, 7), (5, 8), (5, 6), (5, 7), (6, 7)]
```

```python
sage: next(iter(PALPreader(2, output='list'))) [[1, 0], [0, 1], [-1, -1]]
sage: type(_)
<... 'list'>
```

```python
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'>
```

```python
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(_)
<type 'sage.geometry.point_collection.PointCollection'>
```

---

**2.2. Lattice polyhedra**
class sage.geometry.polyhedron.palp_database.Reflexive4dHodge(h11, h21, data_basename=None, **kwds)

Bases: sage.geometry.polyhedron.palp_database.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 – Integers. 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))

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: sage.geometry.polyhedron.ppl_lattice_polytope.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
(continues on next page)
The following polygons are isomorphic over \( \mathbb{Q} \), but not as lattice polytopes:

```python
sage: L1 = LatticePolytope_PPL((1, 0, 3), (0, 1, 0), (0, 0, 1))
sage: L2 = LatticePolytope_PPL((1, 0), (0, 1), (0, 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
```

### is_isomorphic (polytope)

Test if self and polytope are isomorphic.

**INPUT:**

- polytope – a lattice polytope.

**OUTPUT:**

Boolean.

**EXAMPLES:**

```python
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
```

### 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:**

```python
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.ordered_vertices()
((0, 0), (0, 1), (1, 0), (1, 1))
sage: square.ordered_vertices()
((0, 0), (1, 0), (1, 1), (0, 1))
```

### plot ()

Plot the lattice polygon.

2.2. Lattice polyhedra
OUTPUT:

A graphics object.

EXAMPLES:

```python
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()
Graphics object consisting of 6 graphics primitives
sage: LatticePolytope_PPL([0], [1]).plot()
Graphics object consisting of 3 graphics primitives
sage: LatticePolytope_PPL([0]).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:

```python
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)
```

`sage.geometry.polyhedron.ppl_lattice_polygon.polar_P1xP1_polytope()`

The polar of the $P^1 \times P^1$ polytope

EXAMPLES:

```python
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), (2, 0), (0, 2), (2, 2))
```

`sage.geometry.polyhedron.ppl_lattice_polygon.polar_P2_112_polytope()`

The polar of the $P^2[1,1,2]$ polytope

EXAMPLES:

```python
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
```

(continues on next page)
sage: _.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))

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

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

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
The lattice sub-polygons of the polar $P^2[1,1,2]$ polytope

OUTPUT:
A tuple of lattice polytopes.

EXAMPLES:

```python
sage: from sage.geometry.polyhedron.ppl_lattice_polygon import subpolygons_of_polar_P2_112
sage: 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 $\mathbb{Z}$-vectors. In the following, we always use `generator` to refer the PPL generator objects and `vertex` (or integral point) for the corresponding $\mathbb{Z}$-vector.

**EXAMPLES:**

```python
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))
```

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()`

```python
sage: F = next(P.fibration_generator(2))
sage: F.vertices()
((1, 0), (-1, 0), (0, 1), (0, -1))
```

Finally, we can compute automorphisms and identify fibrations that only differ by a lattice automorphism:

```python
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)
(frozenset({(0, -1), (0, 1), (0, 0)}), frozenset({(-1, -1), (1, 1)}))
```

**AUTHORS:**

174 Chapter 2. Polyhedral computations
sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope_PPL(*args)
Construct a new instance of the PPL-based lattice polytope class.

EXAMPLES:

```python
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,
    Variable
sage: p = point(Linear_Expression([2,3],0)); p
point(2/1, 3/1)
sage: LatticePolytope_PPL(p)
A 0-dimensional lattice polytope in ZZ^2 with 1 vertex
sage: P = C_Polyhedron(Generator_System(p)); P
A 0-dimensional polyhedron in QQ^2 defined as the convex hull of 1 point
sage: LatticePolytope_PPL(P)
A 0-dimensional lattice polytope in ZZ^2 with 1 vertex
```

A TypeError is raised if the arguments do not specify a lattice polytope:

```python
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_PPL
sage: LatticePolytope_PPL((0,0),(1/2,1))
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,
    Variable
sage: p = point(Linear_Expression([2,3],0), 5); p
point(2/5, 3/5)
sage: LatticePolytope_PPL(p)
Traceback (most recent call last):
... TypeError: generator is not a lattice polytope generator
sage: P = C_Polyhedron(Generator_System(p)); P
A 0-dimensional polyhedron in QQ^2 defined as the convex hull of 1 point
sage: LatticePolytope_PPL(P)
Traceback (most recent call last):
... TypeError: polyhedron has non-integral generators
```

class sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope_PPL_class
Bases: ppl.polyhedron.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

sage: LatticePolytope_PPL((0,0),(1,0),(0,1))
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

affine_space()
Return the affine space spanned by the polytope.

OUTPUT:
The free module \( \mathbb{Z}^n \), 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]

ambient_space()
Return the ambient space.

OUTPUT:
The free module \( \mathbb{Z}^d \), 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
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:

```python
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)
```

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:

```python
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]
```

Note that the basis choice in `base_projection()` for the quotient is usually different:

```python
sage: proj = poly.base_projection(fiber)
sage: proj_matrix = poly.base_projection_matrix(fiber)
sage: [ proj(p) for p in poly.integral_points() ]
[(-1, -1), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (0, 0), (1, 0), (0, 1)]
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)]
```

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 Z-vectors.
EXAMPLES:

```python
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),)
```

`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:

```python
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))
```

`contains(point_coordinates)`

Test whether point is contained in the polytope.

INPUT:

- `point_coordinates` -- a list/tuple/iterable of rational numbers. The coordinates of the point.

OUTPUT:

Boolean.

EXAMPLES:

```python
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
```

`contains_origin()`

Test whether the polytope contains the origin.

OUTPUT:

Boolean.

EXAMPLES:
**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()**,  **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 an a random but fixed choice will be returned.

**EXAMPLES:**

```python
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),(2,0,0,3))
sage: polygon.embed_in_reflexive_polytope()
The map A*x+b with A=
[ 1  1]
[ 0  1]
[-1 -1]
[ 0  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),
```

(continues on next page)
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:

```python
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]
```

has_IP_property()
Whether the lattice polytope has the IP property.

That is, the polytope is full-dimensional and the origin is an interior point not on the boundary.

OUTPUT:

Boolean.

EXAMPLES:

```python
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
```

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:
The polyhedron need not be full-dimensional:

```python
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))

sage: point = LatticePolytope_PPL((2, 3, 7))
sage: point.integral_points()
((2, 3, 7),)

sage: empty = LatticePolytope_PPL()
sage: 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:

```python
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)
sage: len(simplex.integral_points())
49
```

Finally, the 3-d reflexive polytope number 4078:

```python
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()     # PALP
sage: for p in pts1: p.set_immutable()

sage: set(pts1) == set(pts2)
True

sage: timeit('Polyhedron(v).integral_points()')  # random output
sage: timeit('LatticePolytope(v).points()')      # random output
sage: timeit('LatticePolytope_PPL(*v).integral_points()')  # random output
```

```
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:
```python
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))
```

**is_bounded()**

Return whether the lattice polytope is compact.

**OUTPUT:**

Always True, since polytopes are by definition compact.

**EXAMPLES:**

```python
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import *
    →LatticePolytope_PPL
sage: LatticePolytope_PPL((0,0),(1,0),(0,1)).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:**

```python
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
```

**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:**

```python
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import *
    →LatticePolytope_PPL
sage: LatticePolytope_PPL((0,0),(1,0),(0,1,0)).is_simplex()
True
```

**lattice_automorphism_group**(points=None, point_labels=None)

The integral subgroup of the restricted automorphism group.

**INPUT:**

```python
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
```
• points – A 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 – A 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()
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)); G1
Permutation Group with generators [(), (2,3), (1,4), (1,4)(2,3)]
sage: G1.cardinality()
4
sage: G2 = Z3square.restricted_automorphism_group(vertex_labels=(1,2,3,4))
sage: G2 == PermutationGroup([[2,3], [(1,2),(3,4)], [(1,4)]])
True
sage: G2.cardinality()
8
sage: points = Z3square.integral_points(); points
((0, 0), (1, 1), (1, 2), (2, 1), (2, 2), (3, 3))
sage: Z3square.lattice_automorphism_group(points, point_labels=(1,2,3,4,5,6))
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 trac ticket #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))
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
```

n_vertices()
Return the number of vertices.
OUTPUT:

An 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()
sage: 3
```

pointsets_mod_automorphism(pointsets)

Return pointsets modulo the automorphisms of self.

INPUT:

- polytopes a 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)
(frozenset({(0, -1), (0, 1)}), frozenset({(-1, -1), (1, 1)}))
sage: cell24 = LatticePolytope_PPL(....: (1,0,0,0),(0,1,0,0),(0,0,1,0),(0,0,0,1),
....: (1,-1,-1,1),(-1,0,0,1),(-1,0,0,0),(-1,0,-1,0),(1,0,-1,0),
....: (0,1,1,-1),(-1,1,1,0),(-1,1,0,0),(-1,0,1,0),(-1,0,-1,1),
....: (0,0,0,-1),(0,0,0,1),)
sage: fibers = [ f.vertices() for f in cell24.fibration_generator(2) ]
sage: cell24.pointsets_mod_automorphism(fibers) # long time
(frozenset({(-1, 0, 1, 0), (0, -1, -1, 1), (0, 1, 1, -1), (1, 0, -1, 0)},
        frozenset({(-1, -1, 0), (0, 0, 0, 1), (-1, 0, 0, 0), (-1, 0, 0, -1),
                    (0, 0, 0, 1), (0, 0, 0, -1)},
        frozenset({(-1, 0, 1, 0), (0, -1, -1, 1), (0, 1, 1, -1), (1, 0, -1, 0)},
        frozenset({(-1, 0, 0, 1), (0, 0, 0, 1), (0, 0, 0, -1),
                    (1, 0, 0, -1), (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 $\mathcal{E}(d) = GL(d, \mathbb{R}) \ltimes \mathbb{R}^d$ 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 – a 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:

```python
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.restricted_automorphism_group(vertex_labels=(1,2,3,4)) == 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)]),
\rightarrow{(0,0), (3,3))}
True
sage: set(G.domain()) == set(Z3square.vertices())
True
sage: set(map(tuple,G.orbit(Z3square.vertices()[0]))) == set((0, 0), (1, 2),
\rightarrow{(0, 0), (1, 2), (3, 3)})
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),(0,1,0,-1),(1,0,0,-1),(-1,1,1,-1),
....: (0,0,0,1),(-1,0,0,1),(-1,0,1,0),(-1,0,1,0),(-1,0,0,1),(-1,0,0,1))
sage: 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:

```python
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_PPL
sage: P = LatticePolytope_PPL((1,0,0,0), (0,1,0,0), (0,0,1,0), (-1,-1,-1,1),
....: (0,0,0,1),(-1,0,0,1),(-1,0,1,0),(-1,0,1,0),(-1,0,0,1),(-1,0,0,1))
```

```python
vertices()
Return the vertices as a tuple of `Z`-vectors.

OUTPUT:
A tuple of `Z`-vectors. Each entry is the coordinate vector of an integral points of the lattice polytope.

EXAMPLES:

```python
```

2.2. Lattice polyhedra 185
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 \(\mathbb{Z}\)-vectors, as in \(\text{vertices()}\).

EXAMPLES:

```python
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: 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)}
```

\[\text{vertices_saturating(constraint)}\]

Construct a line.

INPUT:

• expression – a Linear Expression or something convertible to it (Variable or integer).

OUTPUT:

A new Generator representing the line.

Raises a ValueError if the homogeneous part of \`expression\ represents the origin of the vector space.

Examples:

```python
>>> from ppl import Generator, Variable
>>> y = Variable(1)
>>> Generator.line(2*y)
line(0, 1)
>>> Generator.line(y)
line(0, 1)
>>> Generator.line(1)
Traceback (most recent call last):
  ...
ValueError: PPL::line(e):
e == 0, but the origin cannot be a line.
```

Construct a point.
INPUT:

• expression – a `Linear_Expression` or something convertible to it (`Variable` or integer).
• divisor – an integer.

OUTPUT:

A new `Generator` representing the point.

Raises a `ValueError` if `divisor==0`.

Examples:

```python
>>> from ppl import Generator, Variable

>>> y = Variable(1)

>>> Generator.point(2*y+7, 3)
point(0/3, 2/3)

>>> Generator.point(y+7, 3)
point(0/3, 1/3)

>>> Generator.point(7, 3)
point()

>>> Generator.point(0, 0)
Traceback (most recent call last):
...
ValueError: PPL::point(e, d):
  d == 0.
```

sage.geometry.polyhedron.ppl_lattice_polytope.ray(*args,**kwds)

Construct a ray.

INPUT:

• expression – a `Linear_Expression` or something convertible to it (`Variable` or integer).

OUTPUT:

A new `Generator` representing the ray.

Raises a `ValueError` if the homogeneous part of `expression` represents the origin of the vector space.

Examples:

```python
>>> from ppl import Generator, Variable

>>> y = Variable(1)

>>> Generator.ray(2*y)
ray(0, 1)

>>> Generator.ray(y)
ray(0, 1)

>>> Generator.ray(1)
Traceback (most recent call last):
...
ValueError: PPL::ray(e):
e == 0, but the origin cannot be a ray.
```
2.3 Toric geometry

2.3.1 Toric lattices

This module was designed as a part of the framework for toric varieties (variety,fano_variety).

All toric lattices are isomorphic to \( \mathbb{Z}^n \) for some \( n \), but will prevent you from doing “wrong” operations with objects from different lattices.

AUTHORS:

• 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
```

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
```

If fact, the above lattice is exactly the same as before as an object in memory:

```
sage: 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)
```

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
```

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)
```

Dual lattices can act on each other:
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)
```

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
```

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!
```

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
```

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)
```
Warning: While integer vectors (elements of $\mathbb{Z}^n$) 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)
```

Instead, use syntax like

```
sage: N(1,2,3) + N(1,2,3)
N(2, 4, 6)
```

class sage.geometry.toric_lattice.ToricLatticeFactory

Bases: sage.structure.factory.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
```

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*
```
However, we can give an explicit name for it too:

```python
sage: L3 = ToricLattice(3, "N", "M")
sage: L3
3-d lattice N
sage: L3.dual()
3-d lattice M
```

If you want, you may also give explicit LaTeX names:

```python
sage: L4 = ToricLattice(3, "N", "M", r"\mathbb{N}", r"\mathbb{M}")
sage: latex(L4)
\mathbb{N}
sage: latex(L4.dual())
\mathbb{M}
```

While all four lattices above are called “N”, only two of them are equal (and are actually the same):

```python
sage: L1 == L2
False
sage: L1 == L3
True
sage: L1 is L3
True
sage: 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

```
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

```
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

```
class sage.geometry.toric_lattice.ToricLattice_ambient (rank, name, dual_name, latex_name, latex_dual_name)
```

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 `sage.geometry.toric_lattice_element.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
```

**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
```

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
```

**plot(**`**options**`**)**

Plot `self`.

**INPUT:**

* any options for toric plots (see `toric_plotter.options`), none are mandatory.
• a plot.

EXAMPLES:

```
sage: N = ToricLattice(3)
sage: N.plot()
Graphics3d Object
```

```
class sage.geometry.toric_lattice.ToricLattice_generic(base_ring, rank, degree, sparse=False, coordinate_ring=None)

Bases: sage.modules.free_module.FreeModule_generic_pid

Abstract base class for toric lattices.

Element

alias of sage.geometry.toric_lattice_element.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 \( \mathbb{Z} \)-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*, \(K \oplus L, K^* \oplus L^*\))
```

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, \(N \oplus N, M \oplus M\))
```

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]
[0 1 0 0 0]
```

(continues on next page)
intersection\(\text{(other)}\)

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)>
```

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
```

quotient\(\text{(sub, check=True, positive_point=None, positive_dual_point=None)}\)

Return the quotient of self by the given sublattice sub.

INPUT:

- sub – sublattice of self;
- check – (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!

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)>
```

Attempting to quotient one lattice by a sublattice of another will result in a ValueError:
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)>
```

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
```

### 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: `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 \texttt{span\_of\_basis()}, \texttt{submodule()}, and \texttt{submodule\_with\_basis()}.

EXAMPLES:

```python
sage: N = ToricLattice(3)
sage: Ns = N.submodule([N.gen(0)])
sage: Ns.span([N.gen(1)])
Sublattice \langle N(0, 1, 0) \rangle
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.
```

\textbf{\texttt{span\_of\_basis}} (basis, base\_ring=\texttt{Integer Ring}, *args, **kwds)

Return the submodule with the given \texttt{basis}.

\textbf{INPUT:}

- basis – list of elements of the ambient vector space of \texttt{self}.
- base\_ring – (default: \texttt{Z}) base ring for the generated module.

\textbf{OUTPUT:}

- submodule spanned by \texttt{basis}.

\textbf{Note:} The output need not be a submodule of \texttt{self}, nor even of the ambient space. It must, however, be contained in the ambient vector space.

See also \texttt{span()}, \texttt{submodule()}, and \texttt{submodule\_with\_basis()}.

EXAMPLES:

```python
sage: N = ToricLattice(3)
sage: Ns = N.span_of_basis([(1,2,3)])
sage: Ns.span_of_basis([(2,4,0)])
Sublattice \langle N(2, 4, 0) \rangle
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:
\begin{bmatrix}
1/5 & 2/5 & 0 \\
1/7 & 1/7 & 0
\end{bmatrix}
Of course the input basis vectors must be linearly independent:

```python
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.
```

\textbf{class} \texttt{sage.geometry.toric_lattice.ToricLattice\_quotient}(V, W, check=True, positive\_point=None, positive\_dual\_point=None)

\textbf{Bases:} \texttt{sage.modules.fg_pid.fgp_module.FGP\_Module\_class}

Construct the quotient of a toric lattice \texttt{V} by its sublattice \texttt{W}.

\textbf{INPUT:}

- \texttt{V} – ambient toric lattice;
• \( W \) – sublattice of \( V \);
• \texttt{check} – (default: \texttt{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:

• \texttt{positive_point} – a lattice point of \texttt{self} not in the sublattice \texttt{sub} (that is, not zero in the quotient lattice). The quotient generator will be in the same direction as \texttt{positive_point}.

• \texttt{positive_dual_point} – a dual lattice point. The quotient generator will be chosen such that its lift has a positive product with \texttt{positive_dual_point}. Note: if \texttt{positive_dual_point} is not zero on the sublattice \texttt{sub}, then the notion of positivity will depend on the choice of lift!

**OUTPUT:**

• quotient of \( V \) by \( W \).

**EXAMPLES:**

The intended way to get objects of this class is to use \texttt{quotient()} method of toric lattices:

```sage
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[0, 0, 1],)
```

Here, \texttt{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
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[0, 0, -1],)
```

or:

```sage
sage: M = N.dual()
sage: Q = N.quotient(sublattice, positive_dual_point=M(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[0, 0, -1],)
```

**Element**

alias of \texttt{ToricLattice_quotient_element}

**base_extend** \((R)\)

Return the base change of \texttt{self} to the ring \texttt{R}.

**INPUT:**

• \( R \) – either \texttt{Z} or \texttt{Q}.

**OUTPUT:**

• \texttt{self} if \( R = \texttt{Z} \), quotient of the base extension of the ambient lattice by the base extension of the sublattice if \( R = \texttt{Q} \).

**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]

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)

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)])
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
dual ()

Return the lattice dual to self.

OUTPUT:

• a toric lattice quotient.

EXAMPLES:

```python
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)>
```

gens ()

Return the generators of the quotient.

OUTPUT:

A tuple of ToricLattice_quotient_element generating the quotient.

EXAMPLES:

```python
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],)
```

is_torsion_free ()

Check if self is torsion-free.

OUTPUT:

• True is self has no torsion and False otherwise.

EXAMPLES:

```python
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
False
sage: Ns = N.submodule([N(1,4,0)])
sage: Q = N/Ns
sage: Q.is_torsion_free()   # True
True
```

rank ()

Return the rank of self.

OUTPUT:

Integer. The dimension of the free part of the quotient.

EXAMPLES:

```python
sage: N = ToricLattice(3)
sage: Ns = N.submodule([N(2,4,0), N(9,12,0)])
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

class sage.geometry.toric_lattice.ToricLattice_quotient_element(parent, x, check=True)

Bases: sage.modules.fg_pid.fgp_element.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()

class sage.geometry.toric_lattice.ToricLattice_sublattice(ambient, gens, check=True, already_echelonized=False)

Bases: sage.geometry.toric_lattice.ToricLattice_sublattice_with_basis, sage.modules.free_module.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.
• 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)]
```

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)]
```

```
class sage.geometry.toric_lattice.ToricLattice_sublattice_with_basis
(ambient, basis, check=True, echelonize=False, echelonized_basis=None, already_echelonized=False)
```

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)]

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)]

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)>

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()
Graphics3d Object

Now we plot both the ambient lattice and its sublattice:

sage: N.plot() + sublattice.plot(point_color="red")
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:

```python
sage: from sage.geometry.toric_lattice import is_ToricLattice
sage: is_ToricLattice(1)
False
sage: N = ToricLattice(3)
3-d lattice N
sage: 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:

```python
sage: from sage.geometry.toric_lattice import is_ToricLatticeQuotient
sage: is_ToricLatticeQuotient(1)
False
sage: N = ToricLattice(3)
3-d lattice N
sage: is_ToricLatticeQuotient(N)
False
sage: Q = N / N.submodule([(1,2,3), (3,2,1)])
Quotient with torsion of 3-d lattice N by Sublattice <N(1, 2, 3), N(0, 4, 8)>
sage: is_ToricLatticeQuotient(Q)
True
```

### 2.3.2 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 (2010-06-17): substantial improvement during review by Volker Braun.
• Volker Braun (2010-06-21): various spanned/quotient/dual lattice computations added.
EXAMPLES:

Use \texttt{Cone()} to construct cones:

\begin{verbatim}
    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)])
\end{verbatim}

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 \texttt{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 \texttt{check=False} to force \texttt{Cone()} to use exactly the directions that you have specified and \texttt{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 \texttt{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:

\begin{verbatim}
    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)
    N(1, 0, 0)
    in 3-d lattice N
    sage: rays[3]
    N(1, 0, 0)
    sage: halfspace.ray(3)
    N(1, 0, 0)
\end{verbatim}

The method \texttt{rays()} returns a \texttt{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 \texttt{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
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:

```python
sage: positive_xy.dim()
2
sage: positive_xy.lattice_dim()
3
```

You also may be interested in this dimension:

```python
sage: dim(positive_xy.linear_subspace())
0
sage: dim(halfspace.linear_subspace())
2
```

Or, perhaps, all you care about is whether it is zero or not:

```python
sage: positive_xy.is_strictly_convex()
True
sage: halfspace.is_strictly_convex()
False
```

You can also perform these checks:

```python
sage: positive_xy.is_simplicial()
True
sage: four_rays.is_simplicial()
False
sage: positive_xy.is_smooth()
True
```

You can work with subcones that form faces of other cones:

```python
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
```

If you need to know inclusion relations between faces, you can use:

```python
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]
```

(continues on next page)
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]

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

And of course you are always welcome to suggest new features that should be added to cones!

REFERENCES:

• [Fu1993]

sage.geometry.cone.Cone(rays, lattice=None, check=True, normalize=True)
Construct a (not necessarily strictly) convex rational polyhedral cone.

INPUT:

• rays – a 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, \( \mathbb{Z}^n \), 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

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
```

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
```

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'
```

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
```

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!
```

---

2.3. Toric geometry
Of course, you can also provide `lattice` in other cases:

```python
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
```

Or you can construct cones from rays of a particular lattice:

```python
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
```

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:

```python
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
```

The cone can also be specified by a `Polyhedron_base`:

```python
sage: p = plane.polyhedron()
sage: Cone(p)
2-d cone in 2-d lattice N
sage: Cone(p) == plane
True
```

```python
class sage.geometry.cone.ConvexRationalPolyhedralCone(rays=None, lattice=None, ambient=None, ambient_ray_indices=None, PPL=None)
```

Bases: `sage.geometry.cone.IntegralRayCollection`, `_abcoll.Container`

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, \(\mathbb{Z}^n\), 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 \(C \subset \mathbb{R}^d\), the Hilbert basis of \(C\) is the set of all irreducible elements in the semigroup \(C \cap \mathbb{Z}^d\). It is the unique minimal generating set over \(\mathbb{Z}\) for the integral points \(C \cap \mathbb{Z}^d\).

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 \(C \cap \mathbb{Z}^d\). 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 TOP-COM, you don’t have to use it:

```
sage: 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

Two more complicated example from GAP/toric:

sage: Cone([[1,0],[3,4]]).dual().Hilbert_basis()
M(0, 1),
M(4, -3),
M(3, -2),
M(2, -1),
M(1, 0)
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),
M(15, -63, 25, 9),
M( 2, -3, 0, 1),
M( 1, -4, 1, 1),
M(-1, 3, 0, 0),
M( 4, -4, 0, 1),
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( 0, 1, 0, 0),
M( 8, -6, 0, 1),
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(-3, 14, 0, -2),
M(-1, 7, 0, -1),
M( 1, 0, 0, 0)
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

Not full-dimensional cones are ok, too (see trac ticket #11312):

```
sage: Cone([(1,1,0), (-1,1,0)]).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=None, verbose=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 Linear Program (LP) solver to be used. If set to None, the default one is used. For more information on LP 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.

**OUTPUT:**

A 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)
```

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)
```

(continues on next page)
in 2-d lattice N
\begin{verbatim}
sage: cone.Hilbert_coefficients( N(1,1) )
(0, 0, 1)
\end{verbatim}

The cone need not be strictly convex:
\begin{verbatim}
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(1, 1, 0)
in 3-d lattice N
sage: cone.Hilbert_coefficients( N(1,1,3) )
(0, 0, 3, 0, 1)
\end{verbatim}

**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:**


**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:

```python
sage: octant = Cone([(1,0,0), (0,1,0), (0,0,1)])
sage: octant.adjacent()
()  # Empty list
sage: one_face = octant.faces(1)[0]
sage: len(one_face.adjacent())
2
sage: one_face.adjacent()[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:

```python
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())
1
```

The second generating cone is adjacent to this one. Now we create the same fan, but embedded into the 3-dimensional space:

```python
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())
1
```

The result is as before, since we still have:

```python
sage: fan.dim()
2
```

Now we add another cone to make the fan 3-dimensional:

```python
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())
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:

```python
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: 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
sage: 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()
(1,)

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

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))
True
sage: c.contains(-1/2, sqrt(3))
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:**


**EXAMPLES:**
Cross-positive operators on the nonnegative orthant are negations of Z-matrices; that is, matrices whose off-diagonal elements are nonnegative:

```
sage: K = Cone([(1,0),(0,1)])
sage: K.cross_positive_operators_gens()
[ [0 1] [1 0] [-1 0] [0 0] [0 0] [0 0] [0 0] [0 0] [0 0]
[0 0] [1 0] [0 0] [0 0] [0 0] [0 1] [0 0] [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
```

The trivial cone in a trivial space has no cross-positive operators:

```
sage: K = Cone([], ToricLattice(0))
sage: 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 0] [0 0] [0 1] [-1 0] [1 0] [0 -1] ]
```

A non-obvious application is to find the cross-positive operators on the right half-plane [OrlitzkyPosZ]:

```
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 1] [-1 0] ]
```

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() )
....:
for l in K.lyapunov_like_basis() :
  ....:
for c in K.cross_positive_operators_gens() :
```

(continues on next page)
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 \(\text{rays}()\).

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 \(\text{rays}()\).
- \(s\) is one of the \(\text{rays}()\) of this cone’s \(\text{dual}()\).

REFERENCES:
- [Or2017]

EXAMPLES:
Pairs of standard basis elements form a discrete complementarity set for the nonnegative orthant:

```
sage: K = Cone([(1,0),(0,1)])
sage: 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)))
```

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()
True
sage: K.discrete_complementarity_set()
()
```

Likewise for trivial cones, whose duals are the entire space:

```
```
sage: L = ToricLattice(0)
sage: K = Cone([], ToricLattice(0))
sage: K.discrete_complementarity_set()
()

**dual()**

Return the dual cone of self.

**OUTPUT:**

- cone.

**EXAMPLES:**

```sage
cone = Cone([(1,0), (-1,3)])
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)])
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)
```

(continues on next page)
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

embed (cone)
Return the cone equivalent to the given one, but sitting in \texttt{self} as a face.

You may need to use this method before calling methods of \texttt{cone} that depend on the ambient structure, such as \texttt{ambient_ray_indices()} or \texttt{facet_of()}. The cone returned by this method will have \texttt{self} as ambient. If \texttt{cone} does not represent a valid cone of \texttt{self}, \texttt{ValueError} exception is raised.

\textbf{Note:} This method is very quick if \texttt{self} is already the ambient structure of \texttt{cone}, so you can use without extra checks and performance hit even if \texttt{cone} is likely to sit in \texttt{self} but in principle may not.

\textbf{INPUT:}

- \texttt{cone} – a \texttt{cone}.

\textbf{OUTPUT:}

- a \texttt{cone}, equivalent to \texttt{cone} but sitting inside \texttt{self}.

\textbf{EXAMPLES:}

Let’s take a 3-d cone on 4 rays:

\begin{verbatim}
sage: c = Cone([(1,0,1), (0,1,1), (-1,0,1), (0,-1,1)])
\end{verbatim}

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:

\begin{verbatim}
sage: ray = Cone([(0,-1,1)])
sage: ray.ambient_ray_indices()
(0,)
sage: ray.adjacent()
()
sage: ray.ambient()
1-d cone in 3-d lattice N
\end{verbatim}

If we want to operate with this ray as a face of the cone, we need to embed it first:

\begin{verbatim}
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()
\end{verbatim}
(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`
```

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]))
```
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`!

```sage```
f 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()
sage: L
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)
[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```
```
```
```
```
```
```
```
```
```
An alternative to extracting faces from the face lattice is to use \texttt{faces()} method:

\begin{verbatim}
sage: face is quadrant.faces(dim=1)[0]
True
\end{verbatim}

The advantage of working with the face lattice directly is that you can (relatively easily) get faces that are related to the given one:

\begin{verbatim}
sage: face = L.level_sets()[1][0]
sage: D = L.hasse_diagram()
sage: D.neighbors(face)
[2-d cone in 2-d lattice N, 
0-d face of 2-d cone in 2-d lattice N]
\end{verbatim}

However, you can achieve some of this functionality using \texttt{facets()}, \texttt{facet_of()}, and \texttt{adjacent()} methods:

\begin{verbatim}
sage: face = quadrant.faces(1)[0]
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
\end{verbatim}

Note that if \texttt{cone} is a face of \texttt{supercone}, then the face lattice of \texttt{cone} consists of (appropriate) faces of \texttt{supercone}:

\begin{verbatim}
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().top()
3-d face of 4-d cone in 4-d lattice N
sage: cone.face_lattice().top() == cone
True
\end{verbatim}

\texttt{faces}(\texttt{dim=None, codim=None})

Return faces of \texttt{self} of specified (co)dimension.

INPUT:
• \( \text{dim} \) – integer, dimension of the requested faces;
• \( \text{codim} \) – 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 \( \text{dim} \) or \( \text{codim} \) is given, the output will be a tuple of cones;
• if neither \( \text{dim} \) nor \( \text{codim} \) 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 \code{face_lattice()} or \code{adjacent()}, \code{facet_of()}, and \code{facets()}.

**EXAMPLES:**

Let’s take a look at the faces of the first quadrant:

```plaintext
sage: quadrant = Cone([(1,0), (0,1)])
sage: quadrant.faces()
((0-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)
(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]
```

Now you can look at the actual rays of this face...

```plaintext
sage: face.rays()
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:

```plaintext
sage: face.ambient_ray_indices()
(0,)
sage: quadrant.ray(0)
N(1, 0)
```

Note that it is OK to ask for faces of too small or high dimension:

```plaintext
sage: quadrant.faces(-1)
()
sage: quadrant.faces(3)
()
```

In the case of non-strictly convex cones even faces of small non-negative dimension may be missing:

```plaintext
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,),)

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 \( \mathbb{Z}^n \)) 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

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]

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

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
```

```
facet_of()
Return cones of the ambient face lattice having self as a facet.

OUTPUT:
- tuple of cones.

EXAMPLES:
```
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
```

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])
sage: fan.generating_cone(0).facet_of()
()
```

Subcones of generating cones work as before:

```
sage: one_cone = fan(1)[0]
sage: len(one_cone.facet_of())
2
```
facets()
Return facets (faces of codimension 1) of self.

OUTPUT:
• tuple of cones.

EXAMPLES:

```python
sage: quadrant = Cone([(1,0), (0,1)])
sage: quadrant.facets()
(1-d face of 2-d cone in 2-d lattice N,
1-d face of 2-d cone in 2-d lattice N)
```

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:

```python
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
```

intersection(other)
Compute the intersection of two cones.

INPUT:
• other - cone.

OUTPUT:
• cone.

Raises ValueError if the ambient space dimensions are not compatible.

EXAMPLES:

```python
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
```

It is OK to intersect cones living in sublattices of the same ambient lattice:
```python
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)>
```

But you cannot intersect cones from incompatible lattices without explicit conversion:

```python
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
```

### 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 $C_1 == C_2$.
2. They describe the same sets of points. This is tested by $C_1.is_equivalent(C_2)$.
3. They are in the same orbit of $GL(n, \mathbb{Z})$ (and, therefore, correspond to isomorphic affine toric varieties). This is tested by $C_1.is_isomorphic(C_2)$.

**EXAMPLES:**

```python
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
```
**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:**

```python
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
```

Being a face means more than just saturating a facet inequality:

```python
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
```

**is_full_space** *
Check if this cone is equal to its ambient vector space.

**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:

```python
sage: K = Cone([(1,0)])
sage: K.is_full_space()
False
```

Neither is the nonnegative orthant:

```python
sage: K = Cone([(1,0),(0,1)])
sage: K.is_full_space()
False
```

The right half-space contains a vector subspace, but it is still not equal to the entire space:

```python
sage: K = Cone([(1,0),(-1,0),(0,1)])
sage: 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

is_isomorphic(other)
Check if self is in the same $GL(n, \mathbb{Z})$-orbit as other.

INPUT:
• other - cone.

OUTPUT:
• True if self and other are in the same $GL(n, \mathbb{Z})$-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 $C_1 == C_2$.
2. They describe the same sets of points. This is tested by $C_1.is_equivalent(C_2)$.
3. They are in the same orbit of $GL(n, \mathbb{Z})$ (and, therefore, correspond to isomorphic affine toric varieties). This is tested by $C_1.is_isomorphic(C_2)$.

EXAMPLES:

sage: cone1 = Cone([(1,0), (0, 3)])
sage: m = matrix(ZZ, [(1, -5), (-1, 4)])  # a $GL(2, \mathbb{Z})$-matrix
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

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 = Cone([(1,0), (0,1)])
sage: quadrant.is_proper()
True
sage: octant = Cone([(1,0,0), (0,1,0), (0,0,1)])
sage: 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:

```python
sage: quadrant = Cone([(1,0,0), (0,1,0)])
sage: quadrant.is_proper()
False
```

Likewise, a half-space contains at least one line, so it is not proper:

```python
sage: halfspace = Cone([(1,0),(0,1),(-1,0)])
sage: halfspace.is_proper()
False
```

### 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:**

```python
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
```

### 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:**

```python
sage: cone1 = Cone([(1,0), (0, 1)])
sage: cone2 = Cone([(1,0), (-1, 3)])
sage: cone1.is_smooth()
True
sage: cone2.is_smooth()
False
```

The following cones are the same up to a $SL(2, \mathbb{Z})$ coordinate transformation:

```python
sage: Cone([(1,0,0), (2,1,-1)]).is_smooth()
True
sage: Cone([(1,0,0), (2,1,1)]).is_smooth()
True
sage: Cone([(1,0,0), (2,1,2)]).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.

OUTPUT:
True if this cone is solid, and False otherwise.

See also:

is_proper()

EXAMPLES:
The nonnegative orthant is always solid:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: quadrant.is_solid()
True
sage: octant = Cone([(1,0,0), (0,1,0), (0,0,1)])
sage: 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
```

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
```

is_trivial()
Checks if the cone has no rays.

OUTPUT:

• True if the cone has no rays, False otherwise.

EXAMPLES:

```
sage: c0 = Cone([], lattice=ToricLattice(3))
sage: c0.is_trivial()
True
```

(continues on next page)
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
c0 = Cone([(1,0,0), (0,1,0), (0,0,1)])
c0.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
c0 = Cone([(1,0,0), (-1,0,0), (0,1,0), (0,0,1)])
c0.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
c0 = Cone([(1,0), (-1,0), (0,1), (0,-1)])
c0.is_full_space() True
c0.lineality() 2
c0.lattice_dim() 2
```

Per the definition, the lineality of the trivial cone in a trivial space is zero:

```sage
c0 = Cone([], lattice=ToricLattice(0))
c0.lineality() 0
```

linear_subspace()

Return the largest linear subspace contained inside of self.

OUTPUT:

• subspace of the ambient space of self.

EXAMPLES:

```sage
c0 = Cone([(1,0), (0,1), (-1,0)])
c0.linear_subspace()
Vector space of degree 2 and dimension 1 over Rational Field
```

(continues on next page)
Basis matrix:
\[
\begin{bmatrix}
1 & 0 \\
\end{bmatrix}
\]

**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:**

```python
sage: halfplane = Cone([(1,0), (0,1), (-1,0)])
sage: halfplane.lines()
N(1, 0)
in 2-d lattice N
sage: fullplane = Cone([(1,0), (0,1), (-1,-1)])
sage: 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:

```python
sage: K = Cone([(0,0)])
sage: M = MatrixSpace(K.lattice().base_field(), K.lattice_dim())
sage: list(M.basis()) == K.lyapunov_like_basis()
True
```

And by duality, every transformation is Lyapunov-like on the ambient space:

```python
sage: K = Cone([(1,0), (-1,0), (0,1), (0,-1)])
sage: K.is_full_space()
True
```
However, in a trivial space, there are no non-trivial linear maps, so there can be no Lyapunov-like basis:

```python
sage: L = ToricLattice(0)
sage: K = Cone([], lattice=L)
sage: K.lyapunov_like_basis()
[]
```

The Lyapunov-like transformations on the nonnegative orthant are diagonal matrices:

```python
sage: K = Cone([(1,)])
sage: K.lyapunov_like_basis()
[[1]]

sage: K = Cone([(1,0),(0,1)])
sage: K.lyapunov_like_basis()
[ 1 0
 [0 1]

sage: K = Cone([(1,0,0),(0,1,0),(0,0,1)])
sage: K.lyapunov_like_basis()
[ 1 0 0
 [0 0 0
 [0 0 0
 [0 0 0], [0 1 0], [0 0 1]
]
```

Only the identity matrix is Lyapunov-like on the pyramids defined by the one- and infinity-norms [RNPA2011]:

```python
sage: l31 = Cone([(1,0,1), (0,-1,1), (-1,0,1), (0,1,1)])
sage: l31.lyapunov_like_basis()
[ 1 0 0

sage: l3infty = Cone([(0,1,1), (1,0,1), (0,-1,1), (-1,0,1)])
sage: l3infty.lyapunov_like_basis()
[ 1 0 0
```

```
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:

A 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 \( \text{is_proper()} \), 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 \( \text{lyapunov_like_basis()} \) 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 = Cone([[(1,)]])
sage: positives.lyapunov_rank()
1
sage: quadrant = Cone([[(1,0), (0,1)]])
sage: quadrant.lyapunov_rank()
2
sage: octant = Cone([[(1,0,0), (0,1,0), (0,0,1)]])
sage: 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
```

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
```

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()
(continues on next page)
A subspace of dimension $m$ in an $n$-dimensional ambient space has Lyapunov rank $n^2 - m(n - m)$ [Or2017]:

```python
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])
sage: K.lyapunov_rank()
19
sage: K.lattice_dim()**2 - K.dim()*K.codim()
19
```

Lyapunov rank is additive on a product of proper cones [RNPA2011]:

```python
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
```

Two linearly-isomorphic cones have the same Lyapunov rank [RNPA2011]. A cone linearly-isomorphic to the nonnegative octant will have Lyapunov rank 3:

```python
sage: K = Cone([(1,2,3), (-1,1,0), (1,0,6)])
sage: K.lyapunov_rank()
3
```

Lyapunov rank is invariant under `dual()` [RNPA2011]:

```python
sage: K = Cone([(2,2,4), (-1,9,0), (2,0,6)])
sage: K.lyapunov_rank() == K.dual().lyapunov_rank()
True
```

**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 $\sigma$. Then, in the notation of [Fu1993], this method returns the sublattice $M(\sigma) \overset{\text{def}}{=} \sigma^\perp \cap M \subset M$

**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:**
```python
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)>
```

```python
plot(**options)
Plot self.

INPUT:
- any options for toric plots (see toric_plotter.options), none are mandatory.

OUTPUT:
- a plot.

EXAMPLES:
```python
sage: quadrant = Cone([(1,0), (0,1)])
sage: quadrant.plot()
Graphics object consisting of 9 graphics primitives
```

```python
polyhedron()
Return the polyhedron associated to self.
Mathematically this polyhedron is the same as self.

OUTPUT:
- Polyhedron_base.

EXAMPLES:
```python
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
```

Here is an example of a trivial cone (see trac ticket #10237):
```python
sage: origin = Cone([])
sage: origin.polyhedron()
A 0-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex
```

```python
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:**


**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] [1 0] [0 1]]
```

The trivial cone in a trivial space has no positive operators:

```
sage: K = Cone([], ToricLattice(0))
sage: K.positive_operators_gens()[]
```

Every operator is positive on the trivial cone:
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]]
```

A non-obvious application is to find the positive operators on the right half-plane \([OrlitzkyPosZ]\):

```
sage: K = Cone([(1,0),(0,1),(0,-1)])
sage: K.positive_operators_gens()
[
[1 0], [-1 0], [0 1], [ 0 -1], [0 0], [ 0 0], [0 0], [ 0 0], [ 0 0], [1 0], [-1 0], [0 1], [ 0 -1]]
```

random_element (\textit{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 \textit{ring} which defaults to \(\mathbb{Z}\). When \textit{ring} is not \(\mathbb{Z}\), the random element returned will be a vector. Only the rings \(\mathbb{Z}\) and \(\mathbb{Q}\) are currently supported.

INPUT:

- \textit{ring} – (default: \(\mathbb{Z}\)) the ring from which the random generator weights are chosen; either \(\mathbb{Z}\) or \(\mathbb{Q}\).

OUTPUT:

Either a lattice element or vector contained in both this cone and its ambient vector space. If \textit{ring} is \(\mathbb{Z}\), a lattice element is returned; otherwise a vector is returned. If \textit{ring} is neither \(\mathbb{Z}\) nor \(\mathbb{Q}\), then a \texttt{NotImplementedError} is raised.
EXAMPLES:

The trivial element () is always returned in a trivial space:

```
sage: set_random_seed()
sage: K = Cone((), ToricLattice(0))
sage: K.random_element()
N()
sage: K.random_element(ring=QQ)
()```

A random element of the trivial cone in a nontrivial space is zero:

```
sage: set_random_seed()
sage: K = Cone([(0,0,0)])
sage: K.random_element()
N(0, 0, 0)
sage: K.random_element(ring=QQ)
(0, 0, 0)```

A random element of the nonnegative orthant should have all components nonnegative:

```
sage: set_random_seed()
sage: K = Cone([(1,0,0),(0,1,0),(0,0,1)])
sage: all( x >= 0 for x in K.random_element() )
True
sage: all( x >= 0 for x in K.random_element(ring=QQ) )
True```

If ring is not ZZ or QQ, an error is raised:

```
sage: set_random_seed()
sage: K = Cone([(1,0),(0,1)])
sage: K.random_element(ring=RR)
Traceback (most recent call last):
  ...imotoError: ring must be either ZZ or QQ.
```

\noindent relative\_interior\_contains(*args) \noindent 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 \texttt{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))
```
relative_orthogonal_quotient (supercone)
The quotient of the dual spanned lattice by the dual of the supercone’s spanned lattice.

In the notation of [Fu1993], if supercone = \( \rho > \sigma = \text{self} \) is a cone that contains \( \sigma \) as a face, then \( M(\rho) = \text{supercone.orthogonal_sublattice()} \) is a saturated sublattice of \( M(\sigma) = \text{self.orthogonal_sublattice()} \). 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 \( M_{\rho} \), then
• \( M_{\rho}.\text{cover()} == \text{self.orthogonal_sublattice()}, \text{and} \)
• \( M_{\rho}.\text{relations()} == \text{supercone.orthogonal_sublattice()} \).

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 \( \sigma \).

EXAMPLES:

```python
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],)
```

Different codimension:

```python
sage: rho = Cone([(1,-1,1,3),[-1,-1,1,3]])
sage: sigma = rho.facets()[0]
sage: sigma.orthogonal_sublattice()
```
Sublattice \(<M(1, 0, 2, -1), M(0, 1, 1, 0), M(0, 0, 3, -1)>

\begin{verbatim}
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],)
\end{verbatim}

Sign choice in the codimension one case:

\begin{verbatim}
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],)
\end{verbatim}

relative_quotient (subcone)

The quotient of the spanned lattice by the lattice spanned by a subcone.

In the notation of [Fu1993], let \(N\) be the ambient lattice and \(N_\sigma\) the sublattice spanned by the given cone \(\sigma\). If \(\rho < \sigma\) is a subcone, then \(N_\rho = \rho.\text{sublattice()}\) is a saturated sublattice of \(N_\sigma = \text{self.\text{sublattice()}}\). 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 \(N_\rho\), generate \(N_\sigma\).

OUTPUT:

- toric lattice quotient.

Note:

- The quotient \(N_\sigma/N_\rho\) of spanned sublattices has no torsion since the sublattice \(N_\rho\) is saturated.
- In the codimension one case, the generator of \(N_\sigma/N_\rho\) is chosen to be in the same direction as the image \(\sigma/N_\rho\).

**EXAMPLES:**

\begin{verbatim}
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(1, 0, 0, 0), N(1, 1, 0, 0)>
sage: rho.sublattice()
Sublattice \(<N(-1, -1, 1, 3), N(0, -1, 0, 0)>
sage: sigma.relative_orthogonal_quotient(rho)
1-d lattice, quotient
of Sublattice \(<N(-1, -1, 1, 3), N(1, 0, 0, 0), N(1, 1, 0, 0)>
by Sublattice \(<N(1, 0, -1, 3), N(0, 1, 0, 0)>
sage: sigma.relative_orthogonal_quotient(rho).gens()
(N[1, 1, 0, 0],)
\end{verbatim}

More complicated example:

\begin{verbatim}
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()
\end{verbatim}
Sublattice <N(-1, 1, 1), N(1, 2, 3), N(0, 1, 1)>

```
sage: N_rho = rho.sublattice()
sage: N_rho
```

Sublattice <N(1, -1, 1), N(1, 2, 3)>

```
sage: sigma.relative_quotient(rho).gens()
(N[0, 1, 1],)
sage: N = rho.lattice()
sage: 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)
```

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 TOP-COM, you don’t have to use it:

```
sage: 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
```

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)]
```
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()
print(len( cone.semigroup_generators() ))
2806
```

The cone need not be strictly convex:

```sage
halfplane = Cone([(1,0),(2,1),(-1,0)])
print(halfplane.semigroup_generators())
(N(0, 1, 0), N(1, 0, 0), N(-1, 0))
line = Cone([(1,1,1),(-1,-1,-1)])
print(line.semigroup_generators())
(N(1, 1, 1), N(-1, -1, -1))
wedge = Cone([(1,0,0), (1,2,0), (0,0,1), (0,0,-1)])
print(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 trac ticket #11312):

```sage
cone3d = 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
```

Neither full-dimensional nor simplicial:

```sage
A = matrix([[1, 3, 0], [-1, 0, 1], (1, -1, 2), (15, -2, 0)])
print(A.elementary_divisors())
[1, 1, 1, 0]
cone3d = Cone([(3,0,-1), (1,-1,0), (0,1,0), (0,0,1)])
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)]
print(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

$$
\text{par}(x_1, \ldots, x_d) = \mathbb{Z}^n \cap \{ q_1 x_1 + \cdots + q_d x_d : 0 \leq q_i < 1 \}
$$

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(1, 0),
N(0, 1)
```

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
```

The solid restriction of the trivial cone lives in a trivial space:

```
sage: K = Cone([], ToricLattice(0))
sage: 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
```

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()
(continues on next page)```
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

The quotient of the trivial cone is trivial:

sage: K = Cone([], ToricLattice(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

**sublattice** (*args, **kwds*)

The sublattice spanned by the cone.

Let \( \sigma \) be the given cone and \( N = self.lattice() \) the ambient lattice. Then, in the notation of [Fu1993], this method returns the sublattice

\[
N_\sigma \overset{\text{def}}{=} \text{span}(N \cap \sigma)
\]

**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 \( \mathbb{Q} \)-basis, you may want to try the \( \text{basis()} \) method on the result of \( \text{rays()} \).
- 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 \( N_\sigma \), 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 = Cone([(1, 1, 1), (1, -1, 1), (-1, -1, 1), (-1, 1, 1)])
sublattice <N(-1, -1, 1), N(1, 0, 0), N(1, 1, 0)>
sage: matrix( cone.sublattice().gens() ).det()
1
Another example:

```python
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(1, 2, 3), N(4, -5, 1)>
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
```

%sublattice_complement(*args, **kwds)

A complement of the sublattice spanned by the cone.

In other words, `sublattice()` and `sublattice_complement()` together form a \( \mathbb{Z} \)-basis for the ambient `lattice()`.

In the notation of [Fu1993], let \( \sigma \) be the given cone and \( N = \text{self.lattice()} \) the ambient lattice. Then this method returns

\[
N(\sigma) \overset{\text{def}}{=} N/N_{\sigma}
\]

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:**

```python
sage: C2_Z2 = Cone([(1,0),(1,2)])  # C^2/Z_2
sage: c1, c2 = C2_Z2.facets()
sage: c2.sublattice()
Sublattice <N(0, 1)>
sage: c2.sublattice_complement()
Sublattice <N(0, 1)>
```

A more complicated example:

```python
sage: c = Cone([(1,2,3), (4,-5,1)])
sage: c.sublattice()
Sublattice <N(1, 2, 3), N(4, -5, 1)>
sage: c.sublattice_complement()
Sublattice <N(0, -6, -5)>
sage: m = matrix( c.sublattice().gens() + c.sublattice_complement().gens() )
sage: m
[ 1  2  3]
[ 4 -5  1]
[ 0 -6 -5]
```

(continues on next page)
sage: 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:

```python
sage: C2_Z2 = Cone([(1,0),(1,2)])  # C^2/Z_2
sage: c1, c2 = C2_Z2.facets()
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)
```

class sage.geometry.cone.IntegralRayCollection(rays, lattice)

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, \( \mathbb{Z}^n \), 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.

cartesian_product(*other, lattice=None)

Return the Cartesian product of self with other.
• 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:**

```python
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
```

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:**
A 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:

```python
sage: K = Cone([(1,0,0), (0,1,0), (0,0,1)])
sage: 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:

```python
sage: K = Cone([(1,0,0), (0,1,0)])
sage: K.codim()
1
```

If our cone is all of \(\mathbb{R}^2\), then its codimension is zero:

```python
sage: K = Cone([(1,0), (-1,0), (0,1), (0,-1)])
sage: K.is_full_space()
True
sage: 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 = Cone([], lattice=ToricLattice(0))
sage: K.lattice_dim()
0
sage: K.codim()
0

sage: K = Cone([(0,0)])
sage: K.lattice_dim()
1
sage: K.codim()
1

sage: K = Cone([(0,0)])
sage: K.lattice_dim()
2
sage: 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

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, \( \mathbb{Z}^n \) 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

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()
Ambient free module of rank 3
over the principal ideal domain Integer Ring

**lattice_dim()**

Return the dimension of the ambient lattice of self.

OUTPUT:

• integer.

EXAMPLES:

```
sage: c = Cone([(1,0)])
sage: c.lattice_dim()
2
sage: 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
```

**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()
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)
```

**rays (**\texttt{args}**)  
Return (some of the) rays of \texttt{self}.**

**INPUT:**

- \texttt{ray_list} – a list of integers, the indices of the requested rays. If not specified, all rays of \texttt{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
```

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
```

**\texttt{span(**base\_ring=\texttt{None}**)**  
Return the span of \texttt{self}.**

**INPUT:**

- `base_ring` – (default: from lattice) the base ring to use for the generated module.

**OUTPUT:**

A module spanned by the generators of \texttt{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)>
```

The span of the nonnegative orthant is the entire ambient lattice:
By specifying a `base_ring`, we can obtain a vector space:

```python
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]
```

### sage.geometry.cone.PPL_point(*args, **kwds)

Construct a point.

**INPUT:**

- expression – a `Linear_Expression` or something convertible to it (`Variable` or integer).
- divisor – an integer.

**OUTPUT:**

A new `Generator` representing the point.

Raises a `ValueError` if `divisor==0`.

**Examples:**

```python
>>> from ppl import Generator, Variable
>>> y = Variable(1)
>>> Generator.point(2*y+7, 3)
point(0/3, 2/3)
>>> Generator.point(y+7, 3)
point(0/3, 1/3)
>>> Generator.point(7, 3)
point()
>>> Generator.point(0, 0)
Traceback (most recent call last):
  ...
ValueError: PPL::point(e, d):
  d == 0.
```

### sage.geometry.cone.PPL_ray(*args, **kwds)

Construct a ray.

**INPUT:**

- expression – a `Linear_Expression` or something convertible to it (`Variable` or integer).

**OUTPUT:**

A new `Generator` representing the ray.

Raises a `ValueError` if the homogeneous part of `expression` represents the origin of the vector space.

**Examples:**
>>> from ppl import Generator, Variable
>>> y = Variable(1)
>>> Generator.ray(2*y)
ray(0, 1)
>>> Generator.ray(y)
ray(0, 1)
>>> Generator.ray(1)
Traceback (most recent call last):
...
ValueError: PPL::ray(e):
e == 0, but the origin cannot be a ray.

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 \(GL(2, \mathbb{Z})\) equivalence. See Proposition 10.1.1 of [CLS] for the definition. We return the unique \((d, k)\) with minimal \(k\), see Proposition 10.1.3 of [CLS].

EXAMPLES:

```python
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)
(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)
```

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 \(v = r \cdot u\), where \(u\) is the primitive integral vector in the direction of \(v\).

EXAMPLES:

```python
sage: from sage.geometry.cone import integral_length
sage: integral_length([1, 2, 4])
```

(continues on next page)
1 sage: integral_length([2, 2, 4])
2 sage: integral_length([2/3, 2, 4])

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)
False
sage: quadrant = Cone([(1,0), (0,1)])
2-d cone in 2-d lattice N
sage: 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, \mathbb{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)
[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)
[]

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: set_random_seed()
sage: random_cone(max_ambient_dim=0, max_rays=0)
0-d cone in 0-d lattice N
```

We can predict the ambient dimension when `min_ambient_dim` == `max_ambient_dim`:

```
sage: set_random_seed()
sage: K = random_cone(min_ambient_dim=4, max_ambient_dim=4)
sage: K.lattice_dim()
4
```

Likewise for the number of rays when `min_rays` == `max_rays`:
If we specify a lattice, then the returned cone will live in it:

```python
sage: set_random_seed()
sage: L = ToricLattice(5, "L")
sage: K = random_cone(lattice=L)
sage: K.lattice() is L
True
```

We can also request a strictly convex cone:

```python
sage: set_random_seed()
sage: K = random_cone(max_ambient_dim=8, max_rays=10,
.....: strictly_convex=True)
sage: K.is_strictly_convex()
True
```

Or one that isn’t strictly convex:

```python
sage: set_random_seed()
sage: K = random_cone(min_ambient_dim=5, min_rays=2,
.....: strictly_convex=False)
sage: K.is_strictly_convex()
False
```

An example with all parameters set:

```python
sage: set_random_seed()
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
sage: K.is_strictly_convex()
False
sage: K.is_solid()
True
```

### 2.3.3 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:**

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
```

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
```

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))
```

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)
```

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)
in 3-d lattice M
M(-1, 0, 0),
```

(continues on next page)
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

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)

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
for cone in fan1.cones(2):
    print(cone.ambient_ray_indices())
```

(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (0, 4),
(2, 4), (3, 4), (1, 5), (3, 5), (4, 5), (0, 5)

In fact, you do not have to type `.cones`:

```sage
for cone in fan1(2):
    print(cone.ambient_ray_indices())
```

(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (0, 4),
(2, 4), (3, 4), (1, 5), (3, 5), (4, 5), (0, 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()`:

2.3. Toric geometry
You can check how “good” a fan is:

```
sage: fan1.is_complete()
True
sage: fan1.is_simplicial()
True
sage: 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
```

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
```

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: 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
```

Let’s see how “different” are `fan2` and `fan4`:
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!)

```
sage: fan2.ngenerating_cones()
sage: fan2.nrays()
sage: fan4.ngenerating_cones()
sage: fan4.nrays()
```

\texttt{Cone\_of\_fan} \texttt{(ambient, ambient\_ray\_indices)}

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” \texttt{cones}, such cones know their relation to the fan.

**INPUT:**

- \texttt{ambient} – fan whose cone is constructed;
- \texttt{ambient\_ray\_indices} – increasing list or tuple of integers, indices of rays of \texttt{ambient} generating this cone.

**OUTPUT:**

- cone of \texttt{ambient}.

**EXAMPLES:**

The intended way to get objects of this class is the following:

```
sage: fan = toric_varieties.P1xP1().fan()
sage: cone = fan.generating_cone(0)
sage: 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,)
```

**star\_generator\_indices**

Return indices of generating cones of the “ambient fan” containing \texttt{self}.

**OUTPUT:**

- increasing \texttt{tuple} of integers.

**EXAMPLES:**

2.3. Toric geometry
sage: P1xP1 = toric_varieties.P1xP1()
sage: cone = P1xP1.fan().generating_cone(0)
sage: cone.star_generator_indices()
(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()
sage: cone = P1xP1.fan().generating_cone(0)
sage: cone.star_generators()
(2-d cone of Rational polyhedral fan in 2-d lattice N,)
```

````
sage.geometry.fan.Fan(polytope, lattice=None)

Construct the face fan of the given rational polytope.

INPUT:

• polytope – a polytope over \( \mathbb{Q} \) or a lattice polytope. A (not necessarily full-dimensional) polytope containing the origin in its relative interior.
• lattice – ToricLattice, \( \mathbb{Z}^n \), 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 = Fan(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),
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)

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 $\text{check=False}$ option to skip consistency checks.

**INPUT:**

- **cones** – list of either $\text{Cone}$ objects or lists of integers interpreted as indices of generating rays in $\text{rays}$. These must be only **maximal** cones of the fan, unless $\text{discard_faces=True}$ option is specified;
- **rays** – list of rays given as list or vectors convertible to the rational extension of $\text{lattice}$. If $\text{cones}$ are given by $\text{Cone}$ objects $\text{rays}$ 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 $\text{cones}$. 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** – $\text{ToricLattice}$, $\mathbb{Z}^n$, 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). If you know for sure that the input is correct, you may significantly decrease construction time using $\text{check=False}$ option;
- **normalize** – you can further speed up construction using $\text{normalize=False}$ option. In this case $\text{cones}$ must be a list of **sorted** tuples and $\text{rays}$ must be immutable primitive vectors in $\text{lattice}$. 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 $\text{is_complete=True}$ option to speed up some computations. You may also pass $\text{is_complete=False}$ 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 $\text{virtual_rays()}$;
- **discard_faces** – by default, the fan constructor expects the list of **maximal** cones. If you provide “extra” ones and leave $\text{check=True}$ (default), an exception will be raised. If you provide “extra” cones and set $\text{check=False}$, 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 $\text{discard_faces=True}$ option (it works both for $\text{check=True}$ and $\text{check=False}$).

**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 $\text{Fan2d()}$ 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([[0,0], [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.
```

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
```

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
```

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
```

That may seem wrong, but it is not:

```
sage: 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
```

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'
```

264 Chapter 2. Polyhedral computations
Yet another way is to use functions `FaceFan()` and `NormalFan()` to construct fans from lattice polytopes.

We have not yet used `lattice` argument, since it was determined automatically:

```
sage: P2.lattice()
2-d lattice N
sage: 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
```

`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`, `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
```

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
```

(continues on next page)
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

sage.geometry.fan.NormalFan (polytope, lattice=None)
Construct the normal fan of the given rational polytope.

INPUT:
• polytope – a full-dimensional polytope over \( \mathbb{Q} \) or:
  sage.geometry.lattice_polytope.LatticePolytopeClass >.
• lattice – ToricLattice, \( \mathbb{Z}^n \), 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:

```python
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(-1, -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),
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
```

```python
cuboctahed = polytopes.cuboctahedron()
sage: NormalFan(cuboctahed)
Rational polyhedral fan in 3-d lattice N
```

class sage.geometry.fan.RationalPolyhedralFan (cones, rays, lattice, is_complete=None, virtual_rays=None)

Chapter 2. Polyhedral computations

266
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`, `ZZ`, 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:**

```python
sage: fan = toric_varieties.P1xP1().fan()
sage: fan.Gale_transform()
[ 1 1 0 0 -2]
[ 0 0 1 1 -2]
sage: _.base_ring()
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:**
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:

```python
sage: K = ToricLattice(1, 'K')
sage: fan1 = Fan([[0],[1]], [[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
```

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:

```python
sage: F0 = Fan2d([0,1,0,0,0,0])
sage: F0.common_refinement(F0) == F0
True
```

A more complex example with complete fans:

```python
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()
```
sage: FF.ngenerating_cones()

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
```

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
```

`complex (base_ring=Integer Ring, extended=False)`

Return the chain complex of the fan.

To a $d$-dimensional fan $\Sigma$, one can canonically associate a chain complex $K^*$

$$0 \longrightarrow \mathbb{Z}^{\Sigma(d)} \longrightarrow \mathbb{Z}^{\Sigma(d-1)} \longrightarrow \cdots \longrightarrow \mathbb{Z}^{\Sigma(0)} \longrightarrow 0$$

where the leftmost non-zero entry is in degree 0 and the rightmost entry in degree $d$. See [Klyachko], eq. (3.2). This complex computes the homology of $|\Sigma| \subset N_R$ with arbitrary support,

$$H_i(K) = H_{d-i}(|\Sigma|, \mathbb{Z})_{\text{non-cpct}}$$

For a complete fan, this is just the non-compactly supported homology of $\mathbb{R}^d$. In this case, $H_0(K) = \mathbb{Z}$ and 0 in all non-zero degrees.

For a complete fan, there is an extended chain complex

$$0 \longrightarrow \mathbb{Z} \longrightarrow \mathbb{Z}^{\Sigma(d)} \longrightarrow \mathbb{Z}^{\Sigma(d-1)} \longrightarrow \cdots \longrightarrow \mathbb{Z}^{\Sigma(0)} \longrightarrow 0$$

where we take the first $\mathbb{Z}$ 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 $\mathbb{Z}$-term at degree -1 or not.
- `base_ring` – A ring (default: ZZ). The ring to use instead of $\mathbb{Z}$.

OUTPUT:

The complex associated to the fan as a ChainComplex. Raises a ValueError if the extended complex is requested for a non-complete fan.

EXAMPLES:

```
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
```
Homology computations are much faster over \( \mathbb{Q} \) if you do not care about the torsion coefficients:

```python
sage: toric_varieties.P2_123().fan().complex(extended=True, base_ring=QQ)
Chain complex with at most 4 nonzero terms over Rational Field
sage: _.homology()
{-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:

```python
sage: fan = Fan([Cone([(1,0)])])
sage: 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:

```python
sage: K1 = Fan([Cone([(-1,0,0)]), Cone([(1,0,0)])]).complex()
sage: K2 = Fan([Cone([(-1,-1,0)]), Cone([(1,0,0)])]).complex()
sage: K1 == K2
True
```

Things get more complicated for non-complete fans:

```python
sage: fan = Fan([Cone([(1,1,1)]), Cone([(0,1,0), (1,0,0)])])
sage: fan.complex().homology()
{0: 0, 1: 0, 2: Z x Z, 3: 0}
sage: fan = Fan([Cone([(1,0,0), (0,1,0)]), Cone([(-1,0,0), (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}
```

`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).
• 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([(-1,0), (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
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(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()  # empty cone
0-d cone of Rational polyhedral fan in 2-d lattice N
sage: f.cone_containing((0,0))  # empty cone
0-d cone of Rational polyhedral fan in 2-d lattice N
sage: f.cone_containing((-1,1))  # empty cone
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)]
```

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 \texttt{cones()}. \par

**EXAMPLES:**

Cone lattices can be computed for arbitrary fans:

\begin{verbatim}
sage: cone1 = Cone([(1,0), (0,1)])
sage: cone2 = Cone([(-1,0)])
sage: fan = Fan([cone1, cone2])
sage: fan.rays()
N( 0, 1),
N( 1, 0),
N(-1, 0)
in 2-d lattice N
sage: for cone in fan: print(cone.ambient_ray_indices())
(0, 1)
(2,)
sage: L = fan.cone_lattice()
sage: L
Finite poset containing 6 elements with distinguished linear extension
\end{verbatim}

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:

\begin{verbatim}
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
\end{verbatim}

For example, you can do

\begin{verbatim}
sage: for l in L.level_sets()[:-1]:
  ....: print([f.ambient_ray_indices() for f in l])
[()]
[(0,), (1,), (2,)]
[(0, 1)]
\end{verbatim}

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:

\begin{verbatim}
sage: fan = toric_varieties.P1xP1().fan()
sage: L = fan.cone_lattice()
sage: for l in L.level_sets()[:-1]:
  ....: print([f.ambient_ray_indices() for f in l])
[()]
[(0,), (1,), (2,), (3,)]
[(0, 2), (1, 2), (0, 3), (1, 3)]
\end{verbatim}

Let’s also consider the cone lattice of a fan generated by a single cone:

\begin{verbatim}
sage: fan = Fan([cone1])
sage: L = fan.cone_lattice()
sage: L
 Finite poset containing 5 elements with distinguished linear extension
\end{verbatim}

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=\text{None}, \text{codim}=\text{None})*

Return the specified cones of \text{self}.

**INPUT:**

- \text{dim} – dimension of the requested cones;
- \text{codim} – codimension of the requested cones.

**Note:** You can specify at most one input parameter.

**OUTPUT:**

- \text{tuple} of cones of \text{self} of the specified (co)dimension, if either \text{dim} or \text{codim} is given. Otherwise \text{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(0, 1)
N(1, 0)
N(-1, 0)
sage: fan.cones(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!
```

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)
()
```

**contains** *(\text{cone})*

Check if a given \text{cone} is equivalent to a cone of the fan.

**INPUT:**

- \text{cone} – anything.

**OUTPUT:**

```
• False if \( \text{cone} \) is not a cone or if \( \text{cone} \) 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 \( \text{support\_contains()} \).

**EXAMPLES:**

We first construct a simple fan:

```sage
cone1 = Cone([(0,-1), (1,0)])
cone2 = Cone([(1,0), (0,1)])
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 (\( \text{check=False} \) 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
f.contains(Cone([(0,1), (1,0)], check=False))
True
```

Now we check that a non-generating cone is in our fan:

```sage
f.contains(Cone([(1,0)]))
True
Cone([(1,0)]) 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
f.contains(Cone([(1,0), (-0,1)]))
True
```

A point is not a cone:

```sage
n = f.lattice()(1,1); n
N(1, 1)
f.contains(n)
False
```

**embed** (\( \text{cone} \))

Return the cone equivalent to the given one, but sitting in \( \text{self} \).

You may need to use this method before calling methods of \( \text{cone} \) that depend on the ambient structure, such as \( \text{ambient\_ray\_indices()} \) or \( \text{facet\_of()} \). The cone returned by this method will have \( \text{self} \) as ambient. If \( \text{cone} \) does not represent a valid cone of \( \text{self} \), \( \text{ValueError} \) exception is raised.

**Note:** This method is very quick if \( \text{self} \) is already the ambient structure of \( \text{cone} \), so you can use without extra checks and performance hit even if \( \text{cone} \) is likely to sit in \( \text{self} \) 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)])])
```

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
```

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,)
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
```

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!
```
generating_cone \( n \)
Return the \( n \)-th generating cone of \( \text{self} \).

INPUT:
• \( n \) – integer, the index of a generating cone.

OUTPUT:
• \text{cone of fan}.

EXAMPLES:

```
sage: fan = toric_varieties.P1xP1().fan()
sage: fan.generating_cone(0)
2-d cone of Rational polyhedral fan in 2-d lattice \( \mathbb{Z} \)
```

generating_cones()
Return generating cones of \( \text{self} \).

OUTPUT:
• \text{tuple of cones of fan}.

EXAMPLES:

```
sage: fan = toric_varieties.P1xP1().fan()
sage: fan.generating_cones()
(2-d cone of Rational polyhedral fan in 2-d lattice \( \mathbb{Z} \),
 2-d cone of Rational polyhedral fan in 2-d lattice \( \mathbb{Z} \),
 2-d cone of Rational polyhedral fan in 2-d lattice \( \mathbb{Z} \),
 2-d cone of Rational polyhedral fan in 2-d lattice \( \mathbb{Z} \))
sage: cone1 = Cone([(1,0), (0,1)])
sage: cone2 = Cone([-1,0])
sage: fan = Fan([cone1, cone2])
sage: fan.generating_cones()
(2-d cone of Rational polyhedral fan in 2-d lattice \( \mathbb{Z} \),
 1-d cone of Rational polyhedral fan in 2-d lattice \( \mathbb{Z} \))
```

is_complete()
Check if \( \text{self} \) is complete.

A rational polyhedral fan is complete if its cones fill the whole space.

OUTPUT:
• True if \( \text{self} \) is complete and False otherwise.

EXAMPLES:

```
sage: fan = toric_varieties.P1xP1().fan()
sage: fan.is_complete()
True
sage: cone1 = Cone([(1,0), (0,1)])
sage: cone2 = Cone([-1,0])
sage: fan = Fan([cone1, cone2])
sage: fan.is_complete()
False
```

is_equivalent \( \text{other} \)
Check if \( \text{self} \) is “mathematically” the same as \( \text{other} \).

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 $GL(n, \mathbb{Z})$ (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=[[0,1], (0,1), (-1,-1)],
...: check=False)
sage: fan2 = Fan(cones=[(2,1), (0,2)],
...: rays=[[0,1], (-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
```

`is_isomorphic(other)`

Check if self is in the same $GL(n, \mathbb{Z})$-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 $GL(n, \mathbb{Z})$ (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 $GL(n, \mathbb{Z})$-orbit, False otherwise.
See also:

If you want to obtain the actual fan isomorphism, use \texttt{isomorphism}().

EXAMPLES:

Here we pick an $SL(2, \mathbb{Z})$ matrix \texttt{m} and then verify that the image fan is isomorphic:

\begin{verbatim}
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

These fans are “mirrors” of each other:

\begin{verbatim}
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
\end{verbatim}

\texttt{is_simplicial}()

Check if \texttt{self} is simplicial.

A rational polyhedral fan is \texttt{simplicial} if all of its cones are, i.e. primitive vectors along generating rays of every cone form a part of a \textit{rational} basis of the ambient space.

OUTPUT:

- True if \texttt{self} is simplicial and False otherwise.

EXAMPLES:

\begin{verbatim}
sage: fan = toric_varieties.P1xP1().fan()
sage: fan.is_simplicial()
True
sage: conel = Cone([(1,0), (0,1)])
sage: cone2 = Cone([(-1,0)])
sage: fan = Fan([conel, cone2])
sage: fan.is_simplicial()
True
\end{verbatim}

In fact, any fan in a two-dimensional ambient space is simplicial. This is no longer the case in dimension three:
```python
sage: fan = NormalFan(lattice_polytope.cross_polytope(3))
sage: fan.is_simplicial()
False
sage: fan.generating_cone(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:**

```python
sage: fan = toric_varieties.P1xP1().fan()
sage: fan.is_smooth()
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
```

**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())
Traceback (most recent call last):
... FanNotIsomorphicError

linear_equivalence_ideal Rings)
Return the ideal generated by linear relations.

INPUT:

- A polynomial ring in self.nrays() variables.

OUTPUT:

Returns 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)])
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

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:
ngenerating_cones()  
Return the number of generating cones of self.  
  OUTPUT:  
  • integer.  
  EXAMPLES:

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:
(1, 1-d cone of Rational polyhedral fan in 3-d lattice N)

```python
sage: fan.oriented_boundary(bdry[0][1])
```

-0-d cone of Rational polyhedral fan in 3-d lattice N

```python
sage: fan.oriented_boundary(bdry[1][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()`

```python
sage: fan.oriented_boundary(fan)
```

-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

```python
sage: [cone.rays().basis().matrix().det() for cone in fan.generating_cones()]
```

[-1, 1, -1, 1]

A non-full dimensional fan:

```python
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

```python
sage: fan.oriented_boundary(fan)
```

1-d cone of Rational polyhedral fan in 2-d lattice N

```python
plot(**options)
```

Plot `self`.

**INPUT:**

- any options for toric plots (see `toric_plotter.options`), none are mandatory.

**OUTPUT:**

- a plot.

**EXAMPLES:**

```python
sage: fan = toric_varieties.dP6().fan()
sage: fan.plot()
```

Graphics object consisting of 31 graphics primitives

```python
primitive_collections()
```

Return the primitive collections.

**OUTPUT:**

Returns the subsets \( \{i_1, \ldots, i_k\} \subset \{1, \ldots, n\} \) such that

- The points \( \{p_{i_1}, \ldots, p_{i_k}\} \) 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 \( \{i_1, \ldots, i_k\} \) of each primitive collection with the monomials \( x_{i_1} \cdots x_{i_k} \) one generates the Stanley-Reisner ideal in \( \mathbb{Z}[x_1, \ldots] \).

**REFERENCES:**
EXAMPLES:

```python
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})]
```

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:**

```python
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
```

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:**

```python
sage: Fan([Cone([(1,0)]), Cone([(0,1)])]).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 $GL(2,\mathbb{Z})$ transformation, see `classify_cone_2d()`.

**EXAMPLES:**

```python
sage: dP8 = toric_varieties.dP8()
sage: g = dP8.fan().vertex_graph()
sage: g
Graph on 4 vertices
sage: set(dP8.fan(1)) == set(g.vertices())
True
sage: g.edge_labels()  # all edge labels the same since every cone is smooth
[(1, 0), (1, 0), (1, 0), (1, 0)]
```

**virtual_rays** *(\*args)*

Return (some of the) virtual rays of `self`.

Let $N$ be the $D$-dimensional `lattice()` of a $d$-dimensional fan $\Sigma$ in $N_{\mathbb{R}}$. Then the corresponding toric variety is of the form $X \times (\mathbb{C}^*)^{D-d}$. The actual `rays()` of $\Sigma$ 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 $N_{\mathbb{Q}}$ “missed” by the actual rays. (In general addition of virtual rays is not sufficient to span $N$ over $\mathbb{Z}$.)

**Note:** You may use a particular choice of virtual rays by passing optional argument `virtual_rays` to the `Fan()` constructor.

**INPUT:**
• `ray_list` – a 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:**

```python
sage: f = Fan([Cone([(1,0,1,0), (0,1,1,0)])])
sage: f.virtual_rays()
N(0, 0, 0, 1),
N(0, 0, 1, 0)
in 4-d lattice N
sage: f.rays()
N(1, 0, 1, 0),
N(0, 1, 1, 0)
in 4-d lattice N
sage: f.virtual_rays([0])
N(0, 0, 0, 1)
in 4-d lattice N
```

You can also give virtual ray indices directly, without packing them into a list:

```python
sage: f.virtual_rays(0)
N(0, 0, 0, 1)
in 4-d lattice N
```

Make sure that trac ticket #16344 is fixed and one can compute the virtual rays of fans in non-saturated lattices:

```python
sage: N = ToricLattice(1)
sage: B = N.submodule([2]).basis()
sage: f = Fan([Cone([B[0]])])
sage: len(f.virtual_rays())
```

```
sage.geometry.fan.discard_faces(cones)
```

Return the cones of the given list which are not faces of each other.

**INPUT:**

• `cones` – a list of `cones`.

**OUTPUT:**

• a list of `cones`, sorted by dimension in decreasing order.

**EXAMPLES:**

Consider all cones of a fan:

```python
sage: Sigma = toric_varieties.P2().fan()
sage: cones = flatten(Sigma.cones())
sage: len(cones)
7
```

Most of them are not necessary to generate this fan:
sage: from sage.geometry.fan import discard_faces
sage: len(discard_faces(cones))
3
sage: Sigma.ngenerating_cones()
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)
False
sage: fan = toric_varieties.P2().fan()
sage: fan
Rational polyhedral fan in 2-d lattice N
sage: is_Fan(fan)
True

2.3.4 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]
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!

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:

```python
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
```

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).

```python
class sage.geometry.fan_morphism.FanMorphism(morphism, domain_fan, codomain=None, subdivide=False, check=True, verbose=False)
```

Create a fan morphism.

Let $\Sigma_1$ and $\Sigma_2$ be two fans in lattices $N_1$ and $N_2$ respectively. Let $\phi$ be a morphism (i.e. a linear map) from $N_1$ to $N_2$. We say that $\phi$ is compatible with $\Sigma_1$ and $\Sigma_2$ if every cone $\sigma_1 \in \Sigma_1$ is mapped by $\phi$ into a single cone $\sigma_2 \in \Sigma_2$, i.e. $\phi(\sigma_1) \subset \sigma_2$ ($\sigma_2$ may be different for different $\sigma_1$).

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;

2.3. Toric geometry

287
• 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 – (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 – (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 – (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:

```python
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
```

Note, that since \(\phi\) is compatible with these fans, the returned fan is exactly the same object as the initial domain_fan.

```python
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!
sage: fm = FanMorphism(phi, normal, face, subdivide=True)
sage: fm.domain_fan() is normal
False
sage: 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:

```python
sage: fm = FanMorphism(phi, face)
sage: fm.codomain_fan()
```

(continues on next page)
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):
  ... 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!
```

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)
sage: fm.domain_fan().is_equivalent(F2)
True
```

The codomain fan takes the following inputs:

```
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.
```

2.3. Toric geometry 289
OUTPUT:

• **rational polyhedral fan** if no parameters were given, **tuple of cones** otherwise.

EXAMPLES:

```python
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
```

**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:

```python
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
```

**factor**()

Factor self into injective * birational * surjective morphisms.

OUTPUT:

• a triple of **FanMorphism** \((\phi_s, \phi_b, \phi_i)\), such that \(\phi_s\) is surjective, \(\phi_b\) is birational, \(\phi_i\) 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 \(\Sigma \rightarrow \Sigma'\), then we have factorization into

\[
\Sigma \rightarrow \Sigma_s \rightarrow \Sigma_i \rightarrow \Sigma.
\]

Note:

• \(\Sigma_s\) is the finest fan with the smallest support that is compatible with self: any fan morphism from \(\Sigma\) given by the same map of lattices as self factors through \(\Sigma_s\).
• \(\Sigma_i\) is the coarsest fan of the largest support that is compatible with self: any fan morphism into \(\Sigma'\) given by the same map of lattices as self factors though \(\Sigma_i\).
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
sage: A2 = toric_varieties.A2()
sage: P3 = toric_varieties.P(3)
sage: m = matrix([(2,0,0), (1,1,0)])
sage: phi = A2.hom(m, P3)
sage: phi.as_polynomial_map()
Scheme morphism:
    From: 2-d affine toric variety
    To: 3-d CPR-Fano toric variety covered by 4 affine patches
    Defn: Defined on coordinates by sending [x : y] to
           [x^2*y : y : 1 : 1]
```

Now we will work with the underlying fan morphism:

```sage
sage: phi = phi.fan_morphism()
sage: 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
```

Double cover (surjective):

```sage
sage: A2.fan().rays()
N(1, 0),
N(0, 1)
in 2-d lattice N
sage: phi_s
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()
N(1, 0, 0),
N(1, 1, 0)
in Sublattice <N(1, 0, 0), N(0, 1, 0)>
```

Blowup chart (birational):

```sage
sage: phi_b
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 \(<N(1, 0, 0), N(0, 1, 0)>\)
\[
sage: \text{phi}\_b.\text{codomain_fan()}.\text{rays()}
\]
\[
N(1, 0, 0),
N(0, 1, 0),
N(-1, -1, 0)
\]
in Sublattice \(<N(1, 0, 0), N(0, 1, 0)>\)

Coordinate plane inclusion (injective):

\[
sage: \text{phi}\_i
\]
Fan morphism defined by the matrix
\[
\begin{bmatrix}
1 & 0 & 0 \\
0 & 1 & 0
\end{bmatrix}
\]
Domain fan: Rational polyhedral fan in Sublattice \(<N(1, 0, 0), N(0, 1, 0)>\)
Codomain fan: Rational polyhedral fan in 3-d lattice \(N\)
\[
sage: \text{phi}.\text{codomain_fan()}.\text{rays()}
\]
\[
N(1, 0, 0),
N(0, 1, 0),
N(0, 0, 1),
N(-1, -1, -1)
\]
in 3-d lattice \(N\)

\text{image_cone} (cone)

Return the cone of the codomain fan containing the image of \(cone\).

INPUT:

- \(cone\) – a \textit{cone} equivalent to a cone of the \textit{domain\_fan()} of \textit{self}.

OUTPUT:

- a \textit{cone} of the \textit{codomain\_fan()} of \textit{self}.

EXAMPLES:

\[
sage: \text{quadrant} = \text{Cone}([[1,0], [0,1]])
\]
\[
sage: \text{quadrant} = \text{Fan}([[\text{quadrant}]])
\]
\[
sage: \text{quadrant}\_bl = \text{quadrant}.\text{subdivide}([[1,1]])
\]
\[
sage: \text{fm} = \text{FanMorphism(\text{identity_matrix}(2), quadrant}\_\text{bl, quadrant)}
\]
\[
sage: \text{fm}.\text{image_cone(\text{Cone}([[1,0]]))}
\]
1-d cone of Rational polyhedral fan in 2-d lattice \(N\)
\[
sage: \text{fm}.\text{image_cone(\text{Cone}([[1,1]]))}
\]
2-d cone of Rational polyhedral fan in 2-d lattice \(N\)

\text{index} (cone=\text{None})

Return the index of \textit{self} as a map between lattices.

INPUT:

- \(cone\) – (default: \text{None}) a \textit{cone} of the \textit{codomain\_fan()} of \textit{self}.

OUTPUT:

- an integer, \text{infinity}, or \text{None}.

If no cone was specified, this function computes the index of the image of \textit{self} in the codomain. If a cone \(\sigma\) was given, the index of \textit{self} over \(\sigma\) is computed in the sense of Definition 2.1.7 of [HLY2002]: if \(\sigma'\) is any cone of the \textit{domain\_fan()} of \textit{self} whose relative interior is mapped to the relative interior of \(\sigma\), 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 \( \sigma' \) 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:

```python
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
```

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:

```python
sage: Sigma_p.rays()
N( 1),
N(-1)
in 1-d lattice N
sage: Sigma.rays()
N( 1, 1),
N( 0, 1),
N(-1, -1),
N( 1, 0)
in 2-d lattice N
sage: xi.factor()[0].domain_fan().rays()
N( 1, 0),
N(-1, 0)
in Sublattice <N(1, 0)>
```

We see that one of the rays of the fan of \( \mathbb{P}1 \) is mapped to a ray, while the other one to the interior of some 2-d cone. Both rays correspond to single points on \( \mathbb{P}1 \), 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:

```python
sage: [xi.index(cone) for cone in Sigma(1)]
[None, None, None, +Infinity]
sage: [xi.index(cone) for cone in Sigma(2)]
[None, 1, None, None]
```

```

.. _is_birational:

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: 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(), psi.is_bundle(), xi.is_bundle()
(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 $\Sigma_0$ be the kernel fan of $\phi$. Then $\phi$ is a bundle (or splitting) if there is a subfan $\hat{\Sigma}$ of $\Sigma$ such that the following two conditions are satisfied:

1. Cones of $\Sigma$ are precisely the cones of the form $\sigma_0 + \hat{\sigma}$, where $\sigma_0 \in \Sigma_0$ and $\hat{\sigma} \in \hat{\Sigma}$.
2. Cones of $\hat{\Sigma}$ are in bijection with cones of $\Sigma'$ induced by $\phi$ and $\phi$ maps lattice points in every cone $\hat{\sigma} \in \hat{\Sigma}$ bijectively onto lattice points in $\phi(\hat{\sigma})$.

If a fan morphism $\phi : \Sigma \to \Sigma'$ is a bundle, then $X_\Sigma$ is a fiber bundle over $X_{\Sigma'}$ with fibers $X_{\Sigma_0,N_0}$, where $N_0$ is the kernel lattice of $\phi$. 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: 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
sage: psi.index()
2
sage: xi.is_fibration()
False
sage: 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 \( \phi \) is dominant if it is surjective as a map of vector spaces. That is, \( \phi_R : N_R \to N'_R \) 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()
sage: phi = FanMorphism(matrix([[1]]), A1.fan(), P1.fan())
sage: phi.is_dominant()
True
sage: 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 \( \sigma' \) the linear map of vector spaces \( \phi_R \) induces a bijection between \( \sigma \) and \( \sigma' \), and, in addition, \( \phi \) is dominant (that is, \( \phi_R : N_R \to N'_R \) 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: 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_dominant()  # already covered
True
sage: phi.is_fibration()
True
sage: phi.index()
1
sage: psi.is_bundle()
False
```
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.

\begin{Verbatim}
\textbf{sage}: psi.is_fibration()
\textbf{True}
\textbf{sage}: psi.index()
\textbf{2}
\textbf{sage}: xi.is_fibration()
\textbf{False}
\textbf{sage}: xi.index()
\textbf{+Infinity}
\end{Verbatim}

\texttt{is_injective()} \\
Check if \texttt{self} is injective.

\textbf{OUTPUT:}

\begin{itemize}
  \item \texttt{True} if \texttt{self} is injective, \texttt{False} otherwise.
\end{itemize}

Let $\phi : \Sigma \to \Sigma'$ be a fan morphism such that the underlying lattice morphism $\phi : N \to N'$ bijectively maps $N$ to a \textit{saturated} sublattice of $N'$. Let $\psi : \Sigma \to \Sigma'_0$ be the restriction of $\phi$ to the image. Then $\phi$ is \textit{injective} if the map between cones corresponding to $\psi$ (injectively) maps each cone of $\Sigma$ 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.

\textbf{See also:}

\texttt{factor()}.

\textbf{EXAMPLES:}

Consider the fan of the affine plane:

\begin{Verbatim}
\textbf{sage}: A2 = toric_varieties.A(2).fan()
\end{Verbatim}

We will map several fans consisting of a single ray into the interior of the 2-cone:

\begin{Verbatim}
\textbf{sage}: Sigma = Fan([Cone([1,1])])
\textbf{sage}: m = identity_matrix(2)
\textbf{sage}: FanMorphism(m, Sigma, A2).is_injective()
\textbf{False}
\end{Verbatim}

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.

\begin{Verbatim}
\textbf{sage}: Sigma = Fan([Cone([1,1])])
\textbf{sage}: m = matrix(1, 2, [1,1])
\textbf{sage}: FanMorphism(m, Sigma, A2).is_injective()
\textbf{True}
\end{Verbatim}

While the fans in this example are close to the previous one, here the ray corresponds to a 0-dimensional orbit.

\begin{Verbatim}
\textbf{sage}: Sigma = Fan([Cone([1,1])])
\textbf{sage}: m = matrix(1, 2, [2,2])
\textbf{sage}: FanMorphism(m, Sigma, A2).is_injective()
\textbf{False}
\end{Verbatim}
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()
sage: m = identity_matrix(2)
sage: FanMorphism(m, A2, P2).is_injective()
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 $\sigma$ is mapped to the relative interior of $\sigma'$ and, in addition, $\phi_{\mathbb{R}}: N_{\mathbb{R}} \to N'_{\mathbb{R}}$ 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
```

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
```

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
```

**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:

```python
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)>),)
```

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:

```python
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)
```

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:
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 $\sigma$ is a primitive cone corresponding to $\sigma'$ if there is no proper face $\tau$ of $\sigma$ 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
sage: Sigma = toric_varieties.dP6().fan()
sage: Sigma.rays()
N( 0, 1),
N(-1, 0),
N(-1, -1),
N( 0, -1),
N( 1, 0),
N( 1, 1)
in 2-d lattice N
sage: Sigma_p = toric_varieties.P1().fan()
sage: phi = FanMorphism(matrix([[1], [-1]]), Sigma, Sigma_p)
```

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
sage: len(phi.preimage_cones(Cone({(1,)})))
5
```

Yet only rays are primitive:

```sage
sage: phi.primitive_preimage_cones(Cone({(1,)}))
(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()
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()
False
sage: phi.index()
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,)
(1-d cone of Rational polyhedral fan in 2-d lattice N,)
(2-d cone of Rational polyhedral fan in 2-d lattice N,)
```

### 2.3.5 Point collections

This module was designed as a part of framework for toric varieties (`variety`, `fano_variety`).

**AUTHORS:**
- 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)
```

(continues on next page)
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
```

• 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)})
```

• 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
```

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 $\Delta$ is a reflexive polytope. Then the same point collection can be linked as

1. vertices of $\Delta$;
2. facet normals of its polar $\Delta^\circ$;
3. ray generators of the face fan of $\Delta$;
4. ray generators of the normal fan of $\Delta$.

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.

```python
class sage.geometry.point_collection.PointCollection
    Bases: sage.structure.sage_object.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`, 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. Note that None is not the default value - you always must give this argument explicitly, even if it is None.

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 \( \textbf{R} \)-basis for the vector space spanned by the points of self.

EXAMPLES:
```python
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
```
Calling this method twice will always return exactly the same point collection:
```python
sage: c.basis().basis() is c.basis()
True
```

cardinality()
Return the number of points in self.

OUTPUT:
• an integer.

EXAMPLES:
```python
sage: c = Cone([(0,0,1), (1,0,1), (0,1,1), (1,1,1)]).rays()
sage: 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:
```python
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, 1, 1, 1, 1, 1),
```
N+N(1, 1, 1, 1, 1, 1)
in 6-d lattice N+N

\textbf{column_matrix()}

Return a matrix whose columns are points of \texttt{self}.

\textbf{OUTPUT:}

\begin{itemize}
  \item a matrix.
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}
 sage: c = Cone([(0,0,1), (1,0,1), (0,1,1), (1,1,1)]).rays()
sage: c.column_matrix()
\end{verbatim}

\begin{verbatim}
\begin{bmatrix}
  0 & 1 & 0 & 1 \\
  0 & 0 & 1 & 1 \\
  1 & 1 & 1 & 1 \\
\end{bmatrix}
\end{verbatim}

\textbf{dim()}

Return the dimension of the space spanned by points of \texttt{self}.

\textbf{Note:} You can use either \texttt{dim()} or \texttt{dimension()}.  

\textbf{OUTPUT:}

\begin{itemize}
  \item an integer.
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}
 sage: c = Cone([(0,0,1), (1,1,1)]).rays()
sage: c.dimension()
2
 sage: c.dim()
2
\end{verbatim}

\textbf{dimension()}

Return the dimension of the space spanned by points of \texttt{self}.

\textbf{Note:} You can use either \texttt{dim()} or \texttt{dimension()}.  

\textbf{OUTPUT:}

\begin{itemize}
  \item an integer.
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}
 sage: c = Cone([(0,0,1), (1,1,1)]).rays()
sage: c.dimension()
2
 sage: c.dim()
2
\end{verbatim}

\textbf{dual_module()}

Return the dual of the ambient module of \texttt{self}.

\textbf{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
```

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
```

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)
```

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]
```

module()

Return the ambient module of self.

OUTPUT:
• a module.

EXAMPLES:

```python
sage: c = Cone([(0,0,1), (1,0,1), (0,1,1), (1,1,1)]).rays()
sage: c.module()
3-d lattice N
```

The function `output_format` allows you to change the format of output for point collections.

**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:**

```python
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
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")
```

(continues on next page)
Note that the last two outputs are identical, separators are only inserted in the \LaTeX\ mode:

\begin{verbatim}
\texttt{sage: latex(c)}
\begin{array}{r|r|r|r}
0 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 \\
1 & 1 & 1 & 1 \\
\end{array}_{N}
\end{verbatim}

Since this is a static method, you can call it for the class directly:

\begin{verbatim}
\texttt{sage: from sage.geometry.point_collection import PointCollection}
\texttt{sage: PointCollection.output_format("default")}
\texttt{sage: c}
\end{verbatim}

\begin{verbatim}
\texttt{N(0, 0, 1),}
\texttt{N(1, 0, 1),}
\texttt{N(0, 1, 1),}
\texttt{N(1, 1, 1)}
in 3-d lattice \texttt{N}
\end{verbatim}

\begin{verbatim}
set()
\end{verbatim}

Return points of \texttt{self} as a \texttt{frozenset}.

OUTPUT:

\begin{itemize}
\item a \texttt{frozenset}.
\end{itemize}

EXAMPLES:

\begin{verbatim}
\texttt{sage: c = Cone([\langle 0,0,1 \rangle, \langle 1,0,1 \rangle, \langle 0,1,1 \rangle, \langle 1,1,1 \rangle]).rays()}
\texttt{sage: c.set()}
frozenset({\langle 0, 0, 1 \rangle, \langle 0, 1, 1 \rangle, \langle 1, 0, 1 \rangle, \langle 1, 1, 1 \rangle})
\end{verbatim}

\begin{verbatim}
write_for_palp(f)
\end{verbatim}

Write \texttt{self} into an open file \texttt{f} in PALP format.

INPUT:

\begin{itemize}
\item \texttt{f} – a file opened for writing.
\end{itemize}

EXAMPLES:

\begin{verbatim}
\texttt{sage: o = lattice_polytope.cross_polytope(3)}
\texttt{sage: from six import StringIO}
\texttt{sage: f = StringIO()}
\texttt{sage: o.vertices().write_for_palp(f)}
\texttt{sage: print (f.getvalue())}
6 3
1 0 0
0 1 0
0 0 1
-1 0 0
\end{verbatim}

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 is_PointCollection
sage: is_PointCollection(1)
False
sage: c = Cone([(0,0,1), (1,0,1), (0,1,1), (1,1,1)])
sage: is_PointCollection(c.rays())
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 – (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\n3 transposed\n1 2\n3 4\n5 6"
sage: print(data)
3 2 regular
1 2
3 4
5 6
```

(continues on next page)
2 3 transposed
1 2 3
4 5 6

sage: from six 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),
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

2.3.6 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()
sage: fan.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()
Graphics object consisting of 19 graphics primitives
sage: toric_plotter.reset_options()
sage: toric_plotter.options("show_rays")
True
sage: fan.plot()
Graphics object consisting of 31 graphics primitives

class sage.geometry.toric_plotter.ToricPlotter(all_options, dimension, generators=None)

Bases: sage.structure.sage_object.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** – an 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()
sage: fan.plot()
```

Graphics object consisting of 31 graphics primitives

```
sage: print(fan.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: 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

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())
```
All options are exposed as attributes of toric plotters and can be modified after constructions, however you will have to manually call :meth:`adjust_options()` and :meth:`set_rays()` again if you decide to change the plotting mode and/or cut-off bounds. Otherwise plots maybe invalid.

.. _adjust_options:

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 :class:`ToricPlotter` for a detailed example.

OUTPUT:

  • none.

.. _include_points:

include_points(points, force=False)  
Try to include points into the bounding box of self.

INPUT:

  • points – a 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:

.. code-block:: sage

    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...
    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...

.. _plot_generators:

plot_generators()  
Plot ray generators.

Ray generators must be specified during construction or using :meth:`set_rays()` before calling this method.

OUTPUT:

  • a plot.

EXAMPLES:

.. code-block:: sage

    sage: from sage.geometry.toric_plotter import ToricPlotter
    sage: tp = ToricPlotter(dict(), 2, [(3,4)])
    sage: tp.plot_generators()
    Graphics object consisting of 1 graphics primitive

.. _plot_labels:

plot_labels(labels, positions)  
Plot labels at specified positions.

INPUT:
• labels – a string or a list of strings;
• positions – a list of points.

OUTPUT:
• a plot.

EXAMPLES:

```python
sage: from sage.geometry.toric_plotter import ToricPlotter
sage: tp = ToricPlotter(dict(), 2)

sage: tp.plot_labels("u", [(1.5,0)])
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:

```python
sage: from sage.geometry.toric_plotter import ToricPlotter
sage: tp = ToricPlotter(dict(), 2)

sage: tp.adjust_options()

sage: tp.plot_lattice()
Graphics object consisting of 1 graphics primitive
```

`plot_points(points)`
Plot given points.

INPUT:
• points – a list of points.

OUTPUT:
• a plot.

EXAMPLES:

```python
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)])
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
generate: tp = ToricPlotter(dict(), 2, [(3,4)])
sage: tp.plot_ray_labels()
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
generate: tp = ToricPlotter(dict(), 2, [(3,4)])
sage: tp.plot_rays()
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` – a list of 2-d cones.

**OUTPUT:**

- a plot.

**EXAMPLES:**

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: from sage.geometry.toric_plotter import ToricPlotter
generate: tp = ToricPlotter(dict(), 2, quadrant.rays())
sage: tp.plot_walls([quadrant])
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])
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` - a list of primitive non-zero ray generators.

**OUTPUT:**

- none.
EXAMPLES:

```python
sage: from sage.geometry.toric_plotter import ToricPlotter
sage: tp = ToricPlotter(dict(), 2)
sage: tp.adjust_options()
sage: tp.plot_rays()
Traceback (most recent call last):
  ...  
AttributeError: 'ToricPlotter' object has no attribute 'rays'
sage: tp.set_rays([(0,1)])
sage: tp.plot_rays()
Graphics object consisting of 2 graphics primitives
```

`.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 - an integer.

OUTPUT:

- a 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:

```python
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):
  ...
ValueError: expected 4 colors, got 3!
```

`.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 - an integer;
- math_mode – boolean, if True, will produce LaTeX expressions for labels;
- index_set – a list of integers (default: range(n)) that will be used as subscripts for labels.
• a list of \( n \) labels.

If \( \text{label} \) was a list of \( n \) entries, it is returned without changes. If \( \text{label} \) is \( \text{None} \), a list of \( n \) \( \text{None} \)'s is returned. If \( \text{label} \) is a string, a list of strings of the form “\( \text{label}_i \)” is returned, where \( i \) ranges over \( \text{index_set} \). (If \( \text{math_mode} = \text{False} \), the form “\( \text{label}_i \)” is used instead.) If \( n=1 \), there is no subscript added, unless \( \text{index_set} \) was specified explicitly.

EXAMPLES:

```python
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)
['$u$']
```

**INPUT:**

• None;

OR:

• \( \text{option} \) – a 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 \( \text{option} \) argument was given, the current value of \( \text{option} \);

• 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` – an integer, the size of lattice points;
- `ray_thickness` – an integer, the thickness of rays;
- `generator_thickness` – an integer, the thickness of generators;
- `font_size` – an integer, the size of font used for labels;
- `ray_label` – a string or a list of strings used for ray labels; use None to hide labels;
- `wall_label` – a string or a 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
```
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")
```

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")
```

```python
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

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

```python
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` – a dictionary of options that should be passed to lower level plotting functions.

**OUTPUT:**

* a plot.
EXAMPLES:

```python
sage: from sage.geometry.toric_plotter import sector
sage: sector((1,0), (0,1))
Graphics object consisting of 1 graphics primitive
sage: sector((3,2,1), (1,2,3))
Graphics3d Object
```

2.3.7 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.
- 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:

```python
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]]
```

REFERENCES:

- Anders N. Jensen; Gfan, a software system for Groebner fans; http://home.math.au.dk/jensen/software/gfan/gfan.html

```python
class sage.rings.polynomial.groebner_fan.GroebnerFan(I, is_groebner_basis=False, symmetry=None, verbose=False)
```

Bases: sage.structure.sage_object.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

2.3. Toric geometry 317
• `is_groebner_basis` - bool (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:**

```python
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
```

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:

```python
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'))
```

`buchberger()`

Return a lexicographic reduced Groebner basis for the ideal.

**EXAMPLES:**

```python
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]
```

`characteristic()`

Return the characteristic of the base ring.

**EXAMPLES:**

```python
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
```

`dimension_of_homogeneity_space()`

Return the dimension of the homogeneity space.

**EXAMPLES:**

```python
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
```

318  Chapter 2. Polyhedral computations
gfan (cmd='bases', I=None, format=True)
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.

EXAMPLES:

```python
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]
{{
y^9-1-y+3*y^3-3*y^6,
x+1-y^3}
,
{x^3-y,
y^3-1-x}
,
{x^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:

```python
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()
```

ideal()
Return the ideal the was used to define this Groebner fan.

EXAMPLES:

```python
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
```

interactive(*args, **kwds)
See the documentation for self[0].interactive(). This does not work with the notebook.

EXAMPLES:

```python
sage: 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:

```python
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
```

minimal_total_degree_of_a_groebner_basis()
Return the minimal total degree of any Groebner basis.

EXAMPLES:

```python
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
```
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
```
**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
˓
→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]]
```

**render** *(file=None, larger=False, shift=0, rgbcolor=(0, 0, 0), polyfill=<function max_degree>, 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** - bool (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()
```

2.3. Toric geometry 321
```python
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)
```

```python
sage: R.<x,y> = PolynomialRing(QQ, 2)
Sage: R.ideal([y^3 - x^2, y^2 - 13*x]).groebner_fan().render()
Traceback (most recent call last):
  ...\nNotImplementedError
```

**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:**

```python
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()
```

```python
sage: P.<a,b,c> = PolynomialRing(QQ, 3, order="lex")
Sage: sage.rings.ideal.Katsura(P, 3).groebner_fan().render3d()
Traceback (most recent call last):
  ...\nNotImplementedError
```

**ring()**

Return the multivariate polynomial ring.

**EXAMPLES:**

```python
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
```

**tropical_basis** *(check=True, verbose=False)*

Return a tropical basis for the tropical curve associated to this ideal.

**INPUT:**

- **check** - bool (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:**

```python
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]
```
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) - a list of variables to be considered as parameters
- symmetry_generators (optional) - generators of the symmetry group

OUTPUT: a TropicalPrevariety object

EXAMPLES:

```python
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]]
```

```python
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]]
```

```python
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]]
```

```python
sage: GF = I.groebner_fan()
sage: TI = GF.tropical_intersection(parameters=[y])
sage: TI.rays()
[[-1, 0, 0]]
```

weight_vectors()

Return the weight vectors corresponding to the reduced Groebner bases.

EXAMPLES:

```python
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)]
```

```python
sage: r4.<x,y,z,w> = PolynomialRing(QQ,4)
sage: g4 = r4.ideal([x^3+y,y^3-z,z^2-x,w]).groebner_fan()
sage: len(g4.weight_vectors())
23
```
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:**

```python
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]
```

**cone()**
The cone associated with the initial form system.

**EXAMPLES:**

```python
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]
```

**initial_forms()**
The initial forms (polynomials).

**EXAMPLES:**

```python
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()[0]
[y^2 - 1, y^2 - 2, y^2 - 3]
```

**internal_ray()**
A ray internal to the cone associated with the initial form system.

**EXAMPLES:**

```python
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)
```

**rays()**
The rays of the cone associated with the initial form system.

**EXAMPLES:**

```python
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]]

class sage.rings.polynomial.groebner_fan.PolyhedralCone(gfan_polyhedral_cone, ring=Rational Field)

Bases: sage.structure.sage_object.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]]

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

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

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]]

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.

2.3. Toric geometry 325
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
```

`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]
```

**class** `sage.rings.polynomial.groebner_fan.PolyhedralFan(gfan_polyhedral_fan, parameter_indices=[])`

Bases: `sage.structure.sage_object.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> = 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()
(continues on next page)
```

**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[0].polymahedralfan()
sage: 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 = l+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()
(continues on next page)
```
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]]}

**dim()**

Return the dimension of the Groebner fan.

**EXAMPLES:**

```sage
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
```

**f_vector()**

The f-vector of the fan.

**EXAMPLES:**

```sage
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]
```

**is_simplicial()**

Whether the fan is simplicial or not.

**EXAMPLES:**

```sage
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.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
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
```

**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
2.3. Toric geometry
```
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]]}

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]]

to_RationalPolyhedralFan()
Converts 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()

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]

class sage.rings.polynomial.groebner_fan.ReducedGroebnerBasis(groebner_fan, gens, gfan_gens)

Bases: sage.structure.sage_object.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:

```python
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]
```

`groebner_cone (restrict=False)`

Return defining inequalities for the full-dimensional Groebner cone associated to this marked minimal reduced Groebner basis.

**INPUT:**

- `restrict` - bool (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:**

```python
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: G = R.ideal([y^3 - x^2, y^2 - 13*x]).groebner_fan()
sage: G[1].groebner_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]]
```

`ideal ()`

Return the ideal generated by this basis.

**EXAMPLES:**

```python
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
```

`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:**

```python
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               *
*****************************************************
....
```

**class** `sage.rings.polynomial.groebner_fan.TropicalPrevariety (gfan_polyhedral_fan, polynomial_system, poly_ring, parameters={})`

2.3. Toric geometry

329
Bases: `sage.rings.polynomial.groebner_fan.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` - a list of polynomials
- `poly_ring` - the polynomial ring of the list of polynomials
- `parameters` (optional) - a list of variables to be considered as parameters

EXAMPLES:

```python
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]
```

`initial_form_systems()`

Return a list of systems of initial forms for each cone in the tropical prevariety.

EXAMPLES:

```python
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\}]
```

`sage.rings.polynomial.groebner_fan.ideal_to_gfan_format(input_ring, polys)`

Return the ideal in gfan’s notation.

EXAMPLES:

```python
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,z^2*x-y}'
```

`sage.rings.polynomial.groebner_fan.max_degree(list_of_polys)`

Compute the maximum degree of a list of polynomials

EXAMPLES:

```python
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
```
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:

```python
sage: from sage.rings.polynomial.groebner_fan import prefix_check
sage: prefix_check(['z1','z1z1'])
False
sage: prefix_check(['z1','zz1'])
True
```

sage.rings.polynomial.groebner_fan.ring_to_gfan_format(input_ring)
Converts a ring to gfan's format.

EXAMPLES:

```python
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]'```

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:

```python
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)]```

## 2.4 Base classes for polyhedra

### 2.4.1 Base class for polyhedra

class sage.geometry.polyhedron.base.Polyhedron_base(parent, Vrep, Hrep, **kwds)
Bases: sage.structure.element.Element

Base class for Polyhedron objects

**INPUT:**

- `parent` – the parent, an instance of `Polyhedra`.
- `Vrep` – a list `[vertices, rays, lines]` or None. The V-representation of the polyhedron. If None, the polyhedron is determined by the H-representation.
- `Hrep` – a list `[ieqs, eqns]` or None. The H-representation of the polyhedron. If None, the polyhedron is determined by the V-representation.

Only one of `Vrep` or `Hrep` can be different from `None`. 

**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 (0, 0, -1) 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)
sage: p.Hrepresentation(0)
An inequality (0, 0, -1) x + 1 >= 0
sage: p.Hrepresentation(0) == p.Hrepresentation()[0]
True
```

**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
```

**Hrepresentation_str(separator='\n', latex=False, style='>=', align=None, **kwds)**

Return a human-readable string representation of the Hrepresentation of this polyhedron.

**INPUT:**

- `separator` — a string. Default is `"\n"`.
- `latex` — a boolean. Default is `False`.
- `style` — either "positive" (making all coefficients positive) or "<=" or ">=". Default is ">=".
- `align` — a 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` — a string
- `indices` — a tuple or other iterable
OUTPUT:
A string.

EXAMPLES:

```python
sage: P = polytopes.permutahedron(3)
sage: print(P.Hrepresentation_str())
x0 + x1 + x2 == 6
-x1 - x2 >= -5
-x2 >= -3
-x1 >= -3
x1 >= 1
x1 + x2 >= 3
x2 >= 1

sage: print(P.Hrepresentation_str(style='<='))
-x0 - x1 - x2 == -6
x1 + x2 <= 5
x2 <= 3
x1 <= 3
-x1 <= -1
-x1 - x2 <= -3
-x2 <= -1

sage: print(P.Hrepresentation_str(style='positive'))
x0 + x1 + x2 == 6
5 >= x1 + x2
3 >= x2
3 >= x1
x1 >= 1
x1 + x2 >= 3
x2 >= 1

sage: print(P.Hrepresentation_str(latex=True))
\begin{array}{rcl}
x_{0} + x_{1} + x_{2} & = & 6 \\
-x_{1} - x_{2} & \geq & -5 \\
-x_{2} & \geq & -3 \\
-x_{1} & \geq & -3 \\
x_{1} & \geq & 1 \\
x_{1} + x_{2} & \geq & 3 \\
x_{2} & \geq & 1
\end{array}

sage: print(P.Hrepresentation_str(align=False))
x0 + x1 + x2 == 6
-x1 - x2 >= -5
-x2 >= -3
-x1 >= -3
x1 >= 1
x1 + x2 >= 3
x2 >= 1

sage: c = polytopes.cube()
sage: c.Hrepresentation_str(separator=', ', style='positive')
'1 >= x2, 1 >= x1, 1 >= x0, x0 + 1 >= 0, x2 + 1 >= 0, x1 + 1 >= 0'
```

Minkowski_difference (other)

2.4. Base classes for polyhedra
Minkowski sum

`Minkowski_sum(other)`

Deprecated: Use `minkowski_difference()` instead. See trac ticket #23685 for details.

Vrep_generator()

Returns an iterator over the objects of the V-representation (vertices, rays, and lines).

EXAMPLES:

```python
sage: p = polytopes.cyclic_polytope(3, 4)
sage: vg = p.Vrep_generator()
sage: next(vg)
A vertex at (0, 0, 0)
sage: 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:

```python
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
```

Vrepresentation_space()

Return the ambient vector space.

OUTPUT:

A free module over the base ring of dimension `ambient_dim()`.

EXAMPLES:

```python
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
```

adjacency_matrix()

Return the binary matrix of vertex adjacencies.

EXAMPLES:
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:

```python
sage: half_plane = Polyhedron(ieqs=[(0,1,0)])
sage: half_plane.Vrepresentation()
(A line in the direction (0, 1), A ray in the direction (1, 0), A vertex at \(\rightarrow(0, 0)\))
```

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:

```python
sage: half_plane.Hrepresentation()
(An inequality \((1, 0) x + 0 >= 0,\))
```

In one dimension higher, that is for a half-space in 3 dimensions, there is no one-dimensional face. Hence nothing is adjacent:

```python
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]
```

**EXAMPLES:**

In a bounded polygon, every vertex has precisely two adjacent ones:

```python
sage: P = Polyhedron(vertices=[(0, 1), (1, 0), (3, 0), (4, 1)])
sage: for v in P.Vrep_generator():
....:     print("{} {} A vertex at {}\)\".format(P.adjacency_matrix().row(v.index()), v))
(0, 1, 0, 1) A vertex at (0, 1)
```

(continued from previous page)

(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 =
....:
sage: for
....:
(0, 1, 0,
(1, 0, 1,
(0, 1, 0,
(0, 0, 1,
(1, 0, 0,

Polyhedron(vertices=[(0, 1), (1, 0), (3, 0), (4, 1)],
rays=[(0,1)])
v in P.Vrep_generator():
print("{} {}".format(P.adjacency_matrix().row(v.index()), v))
0, 1) A ray in the direction (0, 1)
0, 0) A vertex at (0, 1)
1, 0) A vertex at (1, 0)
0, 1) A vertex at (3, 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 =
....:
sage: for
....:
(0, 1, 0,
(1, 0, 1,
(0, 1, 0,
(0, 0, 0,
(0, 0, 1,

Polyhedron(vertices=[(0, 1), (1, 0), (3, 0), (4, 1)],
rays=[(0,1), (1,1)])
v in P.Vrep_generator():
print("{} {}".format(P.adjacency_matrix().row(v.index()), v))
0, 0) A ray in the direction (0, 1)
0, 0) A vertex at (0, 1)
0, 1) A vertex at (1, 0)
0, 1) A ray in the direction (1, 1)
1, 0) A vertex at (3, 0)

affine_hull(as_affine_map=False, orthogonal=False, orthonormal=False, extend=False)
Return the affine hull.
Each polyhedron is contained in some smallest affine subspace (possibly the entire ambient space). The
affine hull is the same polyhedron but thought of as a full-dimensional polyhedron in this subspace. We
provide a projection of the ambient space of the polyhedron to Euclidian space of dimension of the
polyhedron. Then the image of the polyhedron under this projection (or, depending on the parameter
as_affine_map, the projection itself) is returned.
INPUT:
• as_affine_map (boolean, default = False) – If False, return a polyhedron. If True, return the
affine transformation, that 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.
• 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 neccessary square roots, the extend parameter needs to be set to True.
• extend (boolean, default = False) – if True, allow base ring to be extended if neccessary. This
becomes relevant when requiering an orthonormal transformation.
OUTPUT:
A full-dimensional polyhedron or a linear transformation, depending on the parameter as_affine_map.

336

Chapter 2. Polyhedral computations


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()
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().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(); 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(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(orthonormal=True)
Traceback (most recent call last):
  ...:
ValueError: The base ring needs to be extended; try with "extend=True"
```

```sage
A = L.affine_hull(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 (1.414213562373095?))
```

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(); 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(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),
 A vertex at (1, 1/2, 4/3))
```

```sage
A = S.affine_hull(orthonormal=True, extend=True); A
A 3-dimensional polyhedron in AA^3 defined as the convex hull of 4 vertices
```

(continues on next page)
More examples with the orthonormal parameter:

```python
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(orthonormal=True, extend=True).→volume() for F in P.affine_hull().faces(1))) == {1, sqrt(AA(2))}
True
sage: set((F.as_polyhedron().affine_hull(orthonormal=True, extend=True).→volume() for F in P.affine_hull(orthonormal=True, extend=True).faces(1)→)) == {sqrt(AA(2))}
True
```

```python
sage: D = polytopes.dodecahedron()
sage: F = D.faces(2)[0].as_polyhedron()
sage: F.affine_hull(orthogonal=True)
A 2-dimensional polyhedron in (Number Field in sqrt5 with defining polynomial
→x^2 - 5)^2 defined as the convex hull of 5 vertices
sage: F.affine_hull(orthonormal=True, extend=True)
A 2-dimensional polyhedron in AA^2 defined as the convex hull of 5 vertices
```

```python
sage: K.<sqrt2> = QuadraticField(2)
sage: P = Polyhedron([2*[K.zero()],2*[sqrt2]])
sage: K.<sqrt3> = QuadraticField(3)
sage: P = Polyhedron([2*[K.zero()],2*[sqrt3]])
sage: P.vertices()
(A vertex at (0, 0), A vertex at (sqrt2, sqrt2))
```

```python
sage: A = P.affine_hull(orthonormal=True); A
A 1-dimensional polyhedron in (Number Field in sqrt2 with defining polynomial
→x^2 - 2)^1 defined as the convex hull of 2 vertices
sage: A.vertices()
(A vertex at (0), A vertex at (2))
```

```python
sage: sqrt(6).n()
2.44948974278318
```

The affine hull is combinatorially equivalent to the input:

```python
sage: P.is_combinatorially_isomorphic(P.affine_hull())
True
```
sage: P.is_combinatorially_isomorphic(P.affine_hull(orthogonal=True))
True
sage: P.is_combinatorially_isomorphic(P.affine_hull(orthonormal=True, extend=True))
True

The orthonormal=True parameter preserves volumes; it provides an isometric copy of the polyhedron:

```python
sage: Pentagon = polytopes.dodecahedron().faces(2)[0].as_polyhedron()
sage: P = Pentagon.affine_hull(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
```

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:

```python
sage: Pentagon = polytopes.dodecahedron().faces(2)[0].as_polyhedron()
sage: Pnormal = Pentagon.affine_hull(orthonormal=True, extend=True)
sage: Pgonal = Pentagon.affine_hull(orthogonal=True)
sage: A, b = Pentagon.affine_hull(orthogonal=True, as_affine_map=True)
sage: Adet = (A.matrix().transpose()*A.matrix()).det()
sage: Pnormal.volume()
1.53406271079097?
sage: Pgonal.volume()/sqrt(Adet)
-80*(55*sqrt(5) - 123)/sqrt(-6368*sqrt(5) + 14240)
sage: Pgonal.volume()/sqrt(Adet).n(digits=20)
1.5340627107909646651
sage: AA(Pgonal.volume()^2) == (Pnormal.volume()^2)*AA(Adet)
True
```

An other example with as_affine_map=True:

```python
sage: P = polytopes.permutahedron(4)
sage: A, b = P.affine_hull(orthonormal=True, as_affine_map=True, extend=True)
sage: Q = P.affine_hull(orthonormal=True, extend=True)
sage: Q.center()
(0.7071067811865475?, 1.224744871391589?, 1.732050807568878?)
sage: A(Q.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 \( \mathbb{Q}^2 \) 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(orthogonal=True)
Traceback (most recent call last):
...
NotImplementedError: "orthogonal=True" and "orthonormal=True" work only for compact polyhedra
sage: P.affine_hull(orthonormal=True)
Traceback (most recent call last):
...
NotImplementedError: "orthogonal=True" and "orthonormal=True" work only for compact polyhedra

Setting `as_affine_map` to `True` only works in combination with `orthogonal` or `orthonormal` set to `True`:

sage: S = polytopes.simplex()
sage: S.affine_hull(as_affine_map=True)
Traceback (most recent call last):
...
NotImplementedError: "as_affine_map=True" only works with "orthogonal=True" and "orthonormal=True"

If the polyhedron is full-dimensional, it is returned:

sage: polytopes.cube().affine_hull()
A 3-dimensional polyhedron in \( \mathbb{Z}^3 \) defined as the convex hull of 8 vertices
sage: polytopes.cube().affine_hull(as_affine_map=True)
(Vector space morphism represented by the matrix:
  \[
  \begin{bmatrix}
  1 & 0 & 0 \\
  0 & 1 & 0 \\
  0 & 0 & 1 \\
  \end{bmatrix}
  \]
Domain: Vector space of dimension 3 over Rational Field
Codomain: Vector space of dimension 3 over Rational Field, \((0, 0, 0)\)

`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

`ambient_space()`

Return the ambient vector space.

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

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()
sage: D.backend()
'field'
sage: P = Polyhedron([[1.23]])
sage: P.backend()
'cdd'
```

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:

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:
```python
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)
sage: P.barycentric_subdivision()
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 8 vertices
```

### base_extend

**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), use the same defaulting behavior as described there; it is not attempted to keep the same backend.

**OUTPUT:**

The same polyhedron, but over a larger base ring and possibly with a changed backend.

**EXAMPLES:**

```python
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
```

### base_ring

**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:**

```python
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_ring() == ZZ
True
```
```python
sage: triangle = Polyhedron(vertices = [[1,0],[0,1],[1,1]])
sage: triangle.base_ring() == ZZ
True
```

**bipyramid()**

Return a polyhedron that is a bipyramid over the original.

**EXAMPLES:**

```python
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()]
sage: 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:

```python
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]
```

**bounded_edges()**

Return the bounded edges (excluding rays and lines).

**OUTPUT:**

A generator for pairs of vertices, one pair per edge.

**EXAMPLES:**

```python
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)]
```

**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 \((\text{box}_\text{min}, \text{box}_\text{max})\) where \(\text{box}_\text{min}\) are the coordinates of a point bounding the coordinates of the polytope from below and \(\text{box}_\text{max}\) 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))
sage: polytopes.buckyball(exact=False).bounding_box()
((-0.8090169944, -0.8090169944, -0.8090169944), (0.8090169944, 0.8090169944, 0.8090169944))
```

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: a 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
end
```

```
sage: triangle = Polyhedron(vertices = [[1,0],[0,1],[1,1]],base_ring=AA)
sage: triangle.base_ring()
Algebraic Real Field
sage: triangle.cdd_Hrepresentation()  
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: a string

EXAMPLES:

```
sage: q = Polyhedron(vertices = [[1,1],[0,0],[1,0],[0,1]])
sage: print(q.cdd_Vrepresentation())
V-representation
begin
4 3 rational
1 0 0
```

(continues on next page)
center()

Return the average of the vertices.

See also:

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)
```

change_ring(base_ring=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), use the same defaulting behavior as described there; it is not attempted to keep the same backend.

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...

sage: P.change_ring(ZZ)
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices...

sage: P.change_ring(ZZ) == P
True

sage: P = Polyhedron(vertices=[(-1.3,0), (0,2.3)], base_ring=RDF); P.
(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

sage: P.change_ring(ZZ)
Traceback (most recent call last):
...
TypeError: cannot change the base ring to the Integer Ring (cannot coerce -1.3 into the Integer Ring)
```

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}

```
sage: P = Polyhedron([[2/3,0],[6666666666666666/10^16,0]], base_ring=AA); P
A 1-dimensional polyhedron in AA^2 defined as the convex hull of 2 vertices
sage: P.change_ring(RDF)
A 0-dimensional polyhedron in RDF^2 defined as the convex hull of 1 vertex
sage: P == P.change_ring(RDF)
False
```

\textbf{combinatorial\_automorphism\_group (vertex\_graph\_only=\texttt{False})}

Computes the combinatorial automorphism group.

If \texttt{vertex\_graph\_only} is \texttt{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.

\textbf{INPUT:}

- \texttt{vertex\_graph\_only} – boolean (default: \texttt{False}); whether to return the automorphism group of the vertex edges graph or of the lattice.

\textbf{OUTPUT:}

A \texttt{PermutationGroup} that is isomorphic to the combinatorial automorphism group is returned.

- \texttt{if vertex\_graph\_only is True}: The automorphism group of the vertex-edge graph of the polyhedron
- \texttt{if vertex\_graph\_only is False} (default): The automorphism group of the vertex-facet graph of the polyhedron, see \texttt{vertex\_facet\_graph()}. This group is isomorphic to the automorphism group of the face lattice of the polyhedron.

\textbf{NOTE:}

Depending on \texttt{vertex\_graph\_only}, this method returns groups that are not necessarily isomorphic, see the examples below.

\textbf{See also:}

\texttt{is\_combinatorially\_isomorphic()}, \texttt{graph()}, \texttt{vertex\_facet\_graph()}

\textbf{EXAMPLES:}
Permutations can only exchange vertices with vertices, rays with rays, and lines with lines:

```python
sage: P = Polyhedron(vertices=[(1,0,0), (1,1,0)], rays=[(1,0,0)], lines=[(0,0,1)])
```

```python
sage: P.combinatorial_automorphism_group(vertex_graph_only=True)
Permutation Group with generators [(A vertex at (1,0,0),A vertex at (1,1,0))]
```

This shows an example of two polytopes whose vertex-edge graphs are isomorphic, but their face_lattices are not isomorphic:

```python
sage: Q=Polyhedron([[-123984206864/2768850730773, -101701330976/922950243591, -64154618668/2768850730773, -2748446474675/2768850730773], ....: [-11083969050/98314591817, -4717557075/98314591817, -32618537490/98314591817], ....: [-9690950/554883199, -73651220/554883199, 1823050/554883199, -549885101/554883199], [-5174928/72012097, 7436288/72012097, 5436288/72012097, -37977984/72012097], ....: -60721345/72012097], ....: [-19184/902877, 26136/300959, -21472/902877, 899005/902877], [5351152/1167061933, 88410344/1167061933, 621795064/1167061933, 982203941/1167061933], ....: [4674489456/83665171433, -4026061312/83665171433, 28596876672/83665171433, -83665171433, -83665171433], [85794884940/98972360190089, -1091022223200/98972360190089, 2974263671400/98972360190089, -9832046346111/98972360190089])
```

```python
sage: C = polytopes.cyclic_polytope(4,8)
```

```python
sage: C.is_combinatorially_isomorphic(Q)
False
```

The automorphism group of the face lattice is isomorphic to the combinatorial automorphism group:

```python
sage: CG = C.face_lattice().hasse_diagram().automorphism_group()
```

```python
sage: C.combinatorial_automorphism_group().is_isomorphic(CG)
True
```

```python
sage: QG = Q.face_lattice().hasse_diagram().automorphism_group()
```

```python
sage: Q.combinatorial_automorphism_group().is_isomorphic(QG)
True
```

Test whether the polyhedron contains the given point.

See also:

- interior_contains()
- relative_interior_contains()
• point – coordinates of a point (an iterable).

OUTPUT:
Boolean.

EXAMPLES:

```python
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
```

As a shorthand, one may use the usual in operator:

```python
sage: P.center() in P
True
sage: [-1,-1] in P
False
```

The point need not have coordinates in the same field as the polyhedron:

```python
sage: ray = Polyhedron(vertices=[(0,0)], rays=[(1,0)], base_ring=QQ)
sage: ray.contains([sqrt(2)/3,0])  # irrational coordinates are ok
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
```

The empty polyhedron needs extra care, see trac ticket #10238:

```python
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
```

**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(verts = verts)
sage: simplex_union = a_simplex.convex_hull(another_simplex)
sage: simplex_union.n_vertices()
7

dilation(scalar)

Return the dilated (uniformly stretched) polyhedron.

INPUT:

• scalar – A scalar, not necessarily in \texttt{base\_ring}.

OUTPUT:

The polyhedron dilated by that scalar, possibly coerced to a bigger base ring.

EXAMPLES:

sage: p = Polyhedron(verts = [[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)
sage: p.dilation(2) == p * 2
True

dim()

Return the dimension of the polyhedron.

OUTPUT:

-1 if the polyhedron is empty, otherwise a non-negative integer.

EXAMPLES:

sage: simplex = Polyhedron(verts = [[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

The empty set is a special case (trac ticket #12193):

sage: P1=Polyhedron(verts=[[1,0,0],[0,1,0],[0,0,1]])
sage: P2=Polyhedron(verts=[[2,0,0],[0,2,0],[0,0,2]])
sage: P12 = P1.intersection(P2)
sage: P12
The empty polyhedron in \texttt{ZZ^3}
sage: P12.dim()
-1

dimension()

Return the dimension of the polyhedron.

OUTPUT:
-1 if the polyhedron is empty, otherwise a non-negative integer.

EXAMPLES:

```python
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
```

The empty set is a special case (trac ticket #12193):

```python
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
```

**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:

```python
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))
```

See also:

join() subdirect_sum()

equation_generator()

Return a generator for the linear equations satisfied by the polyhedron.

EXAMPLES:

```python
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
```

equations()

Return all linear constraints of the polyhedron.
OUTPUT:

A tuple of equations.

EXAMPLES:

```python
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,)
```

```python
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.
```

```python
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]]
```

```python
f_vector()
Return the f-vector.

OUTPUT:

Returns a vector whose i-th entry is the number of i-dimensional faces of the polytope.

EXAMPLES:

```python
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)
```

```python
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:

```python
sage: T = polytopes.cuboctahedron()
sage: 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

The polytope has to have rational coordinates:

sage: S = polytopes.dodecahedron()
sage: S.face_fan()
Traceback (most recent call last):
...
NotImplementedError: face fan handles only polytopes over the rationals

REFERENCES:
For more information, see Chapter 7 of [Zie2007].

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:
For a full-dimensional polytope, the basic algorithm is described in lattice_from_incidences(). There are three generalizations of [KP2002] necessary to deal with more general polytopes, corresponding to the extra H/V-representation objects:

- Lines are removed before calling lattice_from_incidences(), and then added back to each face V-representation except for the “empty face”.
- Equations are removed before calling lattice_from_incidences(), and then added back to each face H-representation.
- Rays: Consider the half line as an example. The V-representation objects are a point and a ray, which we can think of as a point at infinity. However, the point at infinity has no inequality associated to it, so there is only one H-representation object altogether. The face lattice does not contain the “face
at infinity”. This means that in `lattice_from_incidences()`, one needs to drop faces with V-representations that have no matching H-representation. In addition, one needs to ensure that every non-empty face contains at least one vertex.

**EXAMPLES:**

```python
sage: square = polytopes.hypercube(2)
sage: square.face_lattice()
Finite lattice containing 10 elements with distinguished linear extension
sage: list(_)
[<>, <0>, <1>, <2>, <3>, <0,1>, <0,2>, <2,3>, <1,3>, <0,1,2,3>]
sage: poset_element = _[6]
sage: a_face = poset_element
sage: a_face
<0,2>
sage: a_face.dim()
1
sage: set(a_face.ambient_Vrepresentation()) == set([square.
  Vrepresentation(0), 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,)
```

A more complicated example:

```python
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]
```

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:

```python
sage: line = Polyhedron(vertices=[[0,]], lines=[[1,]])
sage: [ fl.dim() for fl in line.face_lattice() ]
[-1, 1]
```

**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:**

```python
sage: pentagon = polytopes.regular_polygon(5)
sage: f = pentagon.faces()[0]
sage: fsplit_pentagon = pentagon.face_split(f)
sage: 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:**

```python
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: vertex_trunc1.faces(2)
(<0,1,2,3>,<2,3,4,5>,<1,2,5,6>,<0,1,6,7,8>,<4,5,6,7,9>,<7,8,9>,<0,3,4,8,9>)
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, 1),
 A vertex at (-1, -1, -1/3),
 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: vertex_trunc2.faces(2)
(<0,1,2,3>,<2,3,4,5>,
 (continues on next page)
```

(continues on next page)
\begin{verbatim}
<1,2,5,6>,
<0,1,6,7,8>,
<4,5,6,7,9>,
<7,8,9>,
<0,3,4,8,9>)
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),
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)[0])
sage: edge_trunc.f_vector()
(1, 10, 15, 7, 1)
sage: edge_trunc.faces(2)
(<0,1,2,3>,
<1,2,4,5>,
<4,5,6,7>,
<0,1,5,6,8>,
<2,3,4,7,9>,
<6,7,8,9>,
<0,3,8,9>)
sage: face_trunc = Cube.face_truncation(Cube.faces(2)[0])
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())
True
\end{verbatim}

\textbf{faces (face\_dimension)}

Return the faces of given dimension

\textbf{INPUT:}

\begin{itemize}
  \item face\_dimension – integer. The dimension of the faces whose representation will be returned.
\end{itemize}
OUTPUT:

A tuple of PolyhedronFace. See face for details. The order random but fixed.

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: p.faces(3)
(<0,1,2,3,4,5,6,7>, <0,1,2,3,8,9,10,11>, <0,1,4,5,8,9,12,13>,
 <0,2,4,6,8,10,12,14>, <2,3,6,7,10,11,14,15>, <8,9,10,11,12,13,14,15>,
 <4,5,6,7,12,13,14,15>, <1,3,5,7,9,11,13,15>)
sage: face = p.faces(3)[0]
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))
```

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()]
[0, 1, 2, 3, 4, 5, 6, 7]
sage: 
[ ([get_idx(_)
    for _ in face.ambient_Vrepresentation()],
    ....:   [get_idx(_)
    for _ in face.ambient_Hrepresentation()])
    ....:   for face in p.faces(3) ]

```

```
facet_adjacency_matrix()
```

Return the adjacency matrix for the facets and hyperplanes.

```
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]
```

```
gale_transform()
```

Return the Gale transform of a polytope as described in the reference below.
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:

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

REFERENCES:


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 self.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 self.integral_points_count().

ALGORITHM:

The function computes each of the components of the requested point in turn. To compute x_i, the ith com-
ponent, 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:

```python
sage: P = Polyhedron(vertices=[(-1,-1),(1,0),(1,1),(0,1)])
sage: P.get_integral_point(1)
(0, 0)
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: ...
```

```python
sage: Q = Polyhedron([(1,3), (2, 7), (9, 77)])
sage: Q.get_integral_point(1) for i in range(Q.integral_points_count())]
[(-1, -1), (0, 0), (0, 1), (1, 0), (1, 1)]
sage: P.get_integral_point(5)
Traceback (most recent call last):
  ... IndexError: ...
```

```python
sage: Q = Polyhedron([(1,3), (2, 7), (9, 77)])
sage: Q.get_integral_point(1) for i in range(Q.integral_points_count())]
[(-1, -1), (0, 0), (0, 1), (1, 0), (1, 1)]
sage: P.get_integral_point(5)
Traceback (most recent call last):
  ... IndexError: ...
```

(continues on next page)
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: ...

graph()

Return a graph in which the vertices correspond to vertices of the polyhedron, and edges to edges.

EXAMPLES:

sage: g3 = polytopes.hypercube(3).vertex_graph(); g3
Graph on 8 vertices
sage: g3.automorphism_group().cardinality()
48
sage: s4 = polytopes.simplex(4).vertex_graph(); s4
Graph on 5 vertices
sage: s4.is_eulerian()
True

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 polytope.

EXAMPLES:

sage: p = polytopes.hypercube(2)
sage: p.hyperplane_arrangement()
Arrangement <t0 + 1 | -t1 + 1 | t1 + 1 | t0 + 1>

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()
inequalities()  
Return all inequalities.

OUTPUT:
A tuple of inequalities.

EXAMPLES:

```python
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)
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
```

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:

```python
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]
(-6, 0, 1, 1, 1)
```


[[0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]]

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

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

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))
Here is a simplex where the naive algorithm of running over all points in a rectangular bounding box no longer works fast enough:

```python
sage: v = [(1,0,-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
```

A case where rounding in the right direction goes a long way:

```python
sage: P = 1/10*polytopes.hypercube(14)
sage: 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:

```python
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() # PALP
sage: for p in pts1: p.set_immutable()
```

```python
sage: set(pts1) == set(pts2)
True
```

```python
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
```

**integral_points_count(****kwds**)

Return the number of integral points in the polyhedron.

This generic version of this method simply calls integral_points.

EXAMPLES:

```python
sage: P = polytopes.cube()
sage: P.integral_points_count()
27
```

We shrink the polyhedron a little bit:
 Sage: Q = P*(8/9)
 Sage: 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

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: ...

**integrate** (polynomial, **kwds)

Return the integral of a polynomial over a polytope.

**INPUT:**

- P – Polyhedron.
- polynomial – A multivariate polynomial or a valid LattE description string for polynomials.
- **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

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
Integral over a non full-dimensional polytope:

```python
sage: x, y = polygens(QQ, 'x, y')
sage: P = Polyhedron( vertices=[[0,0],[1,1]])
sage: P.integrate(x*y)  # optional - latte_int
Traceback (most recent call last):
  ...  
NotImplementedError: The polytope must be full-dimensional.
```

**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:

True or False.

EXAMPLES:

```python
sage: P = Polyhedron( vertices=[[0,0],[1,1],[1,-1]])
sage: P.contains( [1,0] )
True
sage: P.interior_contains( [1,0] )
False
```

If the polyhedron is of strictly smaller dimension than the ambient space, its interior is empty:

```python
sage: P = Polyhedron( vertices=[[0,1],[0,-1]])
sage: P.contains( [0,0] )
True
sage: P.interior_contains( [0,0] )
False
```

The empty polyhedron needs extra care, see trac ticket #10238:

```python
sage: empty = Polyhedron(); empty
The empty polyhedron in ZZ^0
sage: empty.interior_contains([])
False
```

**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

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

The intersection of two $\mathbb{Z}$-polyhedra is not necessarily a $\mathbb{Z}$-polyhedron:

sage: cube.intersection(oct*2)
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 12 vertices

sage: cube & oct*2
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 12 vertices

is_Minkowski_summand(*args, **kwds)
Deprecated: Use is_minkowski_summand() instead. See trac ticket #23685 for details.

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 faces
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

All the faces of the 3-dimensional permutahedron are either combinatorially isomorphic to a square or a
hexagon:
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]
```

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],
    \([-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, 24/89, -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
```

False

```python
sage: T.f_vector(), S.f_vector()
((1, 10, 15, 7, 1), (1, 10, 15, 7, 1))
```

```python
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'
```

```python
sage: G = Graph()
sage: C.is_combinatorially_isomorphic(G)
Traceback (most recent call last):
  ...?
AssertionError: input 'other' must be a polyhedron
```

```python
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
```

### `is_compact()`

Test for boundedness of the polytope.

**EXAMPLES:**

```python
sage: p = polytopes.icosahedron()
sage: p.is_compact()
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
```

### `is_empty()`

Test whether the polyhedron is the empty polyhedron

**OUTPUT:**

Boolean.

**EXAMPLES:**

```python
sage: P = Polyhedron(vertices=[[1,0,0],[0,1,0],[0,0,1]]);
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=());
The empty polyhedron in ZZ^0
sage: Q.is_empty(), Q.is_universe()
(True, False)

sage: R = Polyhedron(lines=[(1,0),(0,1)]);
R
```

(continues on next page)
A 2-dimensional polyhedron in \( \mathbb{Z}^2 \) defined as the convex hull of 1 vertex and 2 lines.

```python
sage: R.is_empty(), R.is_universe()
(False, True)
```

### is_full_dimensional()

Return whether the polyhedron is full dimensional.

**OUTPUT:**

Boolean. Whether the polyhedron is not contained in any strict affine subspace.

**EXAMPLES:**

```python
sage: polytopes.hypercube(3).is_full_dimensional()
True
sage: Polyhedron(vertices=[(1,2,3)], rays=[(1,0,0)]).is_full_dimensional()
False
```

### 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` – (default: False) boolean; 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:**

```python
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])
```

(continues on next page)
The method is not implemented for non-full-dimensional polytope or unbounded polyhedra:

```python
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, polyhedron 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, polyhedron.
```

### 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:**

```python
sage: polytopes.cross_polytope(3).is_lattice_polytope()
True
sage: polytopes.regular_polygon(5).is_lattice_polytope()
False
```

### is_minkowski_summand(Y)

Test whether Y is a Minkowski summand.

See :meth:`minkowski_sum()`.

**OUTPUT:**

Boolean. Whether there exists another polyhedron Z such that self can be written as Y ⊕ Z.

**EXAMPLES:**

```python
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)
**is_neighborly** *(k=None)*

Return whether the polyhedron is neighborly.

If the input `k` is provided then return whether the polyhedron is `k`-neighborly

See Wikipedia article Neighborly_polytope

**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:**

```python
sage: cube = polytopes.hypercube(3)
sage: cube.is_neighborly()
True
sage: cube = polytopes.hypercube(4)
sage: cube.is_neighborly()
False
```

Cyclic polytopes are neighborly:

```python
sage: all([polytopes.cyclic_polytope(i, i + 1 + j).is_neighborly() for i in range(5) for j in range(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:

```python
sage: testpolys = [polytopes.cube(), polytopes.cyclic_polytope(6, 9), polytopes.simplex(6)]
sage: [(P.neighborliness()>=floor(P.dim()/2)) == P.is_neighborly() for P in testpolys]
[True, True, True]
```

**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

**is_simplex()**
Return whether the polyhedron is a simplex.

EXAMPLES:

```sage
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
```

**is_simplicial()**
Tests if the polytope is simplicial

A polytope is simplicial if every facet is a simplex.

See Wikipedia article Simplicial_polytope

EXAMPLES:

```sage
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
```

The method is not implemented for unbounded polyhedra:

```sage
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.
```

**is_universe()**
Test whether the polyhedron is the whole ambient space

OUTPUT:

Boolean.

EXAMPLES:
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
```

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
```

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 \(\text{envelope=}\text{False}\), a \text{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 #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
```

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); lp
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
```

**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)
```

**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),)

**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:**

```python
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
```

**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

\[ X \ominus Y = (X^c \oplus Y)^c = \cap_{y \in Y} (X - y) \]

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:**

```python
sage: X = polytopes.hypercube(3)
sage: Y = Polyhedron( facets=[[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
```

The polyhedra need not be full-dimensional:
Minus sign is really an alias for `minkowski_difference()`

```python
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 16 vertices

```python
sage: four_cube.minkowski_difference(four_simplex) == four_cube - four_simplex
True
```

Coercion of the base ring works:

```python
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
```

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:**

```python
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

```python
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

```python
sage: four_cube.minkowski_sum(four_simplex) == four_cube + four_simplex
True
```

```python
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)
```

(continues on next page)
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

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

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

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

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
**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
```

**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
```

**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
```

**n_vertices()**

Return the number of vertices. The representation will always be minimal.

**EXAMPLES:**

```
sage: p = Polyhedron(vertices = [[1,0],[0,1],[1,1]], rays=[[1,1]])
sage: p.n_vertices()
2
```

**neighborliness()**

Returns the largest \( k \), such that the polyhedron is \( k \)-neighborly.

In case of the \( d \)-dimensional simplex, it returns \( d + 1 \).

See [Wikipedia article Neighborly_polytope](https://en.wikipedia.org/wiki/Nearliness)

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()
```

(continues on next page)
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]

normal_fan()
Return the normal fan of a compact full-dimensional rational polyhedron.

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

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], [AA(sqrt(2))), 0], [0, AA(sqrt(2))]])
sage: R.normal_fan()
Traceback (most recent call last):
...
NotImplementedError: normal fan handles only polytopes over the rationals

REFERENCES:
For more information, see Chapter 7 of [Zie2007].

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:
\begin{verbatim}
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: 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)
\end{verbatim}

It works with a polyhedral face as well:
\begin{verbatim}
sage: vv = cube.faces(0)[0]
sage: ops_cube2 = cube.one_point_suspension(vv)
sage: ops_cube == ops_cube2
True
\end{verbatim}

See also:
face_split()
plot(point=None, line=None, polygon=None, wireframe='blue', fill='green', projection_direction=None, **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)
• **projection_direction** – coordinate list/tuple/iterable or None (default). The direction to use for the `schlegel_projection()` of the polytope. If not specified, no projection is used in dimensions $< 4$ and parallel projection is used in dimension 4.

• **kwds** – optional keyword parameters that are passed to all graphics objects.

OUTPUT:
A (multipart) graphics object.

EXAMPLES:

```python
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)
```

By default, the wireframe is rendered in blue and the fill in green:

```python
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
```

Draw the lines in red and nothing else:

```python
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
```

Draw points in red, no lines, and a blue polygon:

```python
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
```

If we instead use the **fill** and **wireframe** options, the coloring depends on the dimension of the object:
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

polar()

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.

EXAMPLES:

sage: p = Polyhedron(vertices = [[0,0,1],[0,1,0],[1,0,0],[0,0,0],[1,1,1]], base_ring=QQ)
sage: 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

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

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)
The Cartesian product is the product in the semiring of polyhedra:

```python
sage: P1.product(P2)
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 4 vertices
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
```

### projection()

Return a projection object.

**See also:**

`schlegel_projection()` for a more interesting projection.

**OUTPUT:**

The identity projection. This is useful for plotting polyhedra.

**EXAMPLES:**

```python
sage: p = polytopes.hypercube(3)
sage: proj = p.projection()
sage: proj
The projection of a polyhedron into 3 dimensions
```

### pyramid()

Returns a polyhedron that is a pyramid over the original.

**EXAMPLES:**

```python
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)
```

### 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
```

**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
```

**random_integral_point(**kwds)**

Return an integral point in this polyhedron chosen uniformly at random.

**INPUT:**

- `**kwds` — optional keyword parameters that are passed to `self.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()  # random, optional - latte_int
True
sage: P.random_integral_point(explicit Enumeration_threshold=0, triangulation=\"cddlib\")  # random, optional - latte_int
(1, 1)
sage: P.random_integral_point(explicit Enumeration_threshold=0, triangulation=\"cddlib\", foo=7)  # optional - latte_int
Traceback (most recent call last):
  ...RuntimeError: ...
sage: Q = Polyhedron(vertices=[(2, 1/3)], rays=[(1, 2)])
Traceback (most recent call last):
  ...ValueError: ...
sage: R = Polyhedron(vertices=[(1/2, 0), (1, 1/2), (0, 1/2)])
Traceback (most recent call last):
```

(continues on next page)
... EmptySetError: ...

**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)]
```

**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))
```

**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:**
A 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() == [list(r) for r in p.ray_generator()]
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:**
True or False.
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
```

The empty polyhedron needs extra care, see trac ticket #10238:

```
sage: empty = Polyhedron(); empty
The empty polyhedron in ZZ^0
sage: empty.relative_interior_contains([])
False
```

```
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)
sage: type(p_solid)
<type '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()
sage: p_wireframe._objects
[Line defined by 2 points, Line defined by 2 points, Line defined by 2 points]
```

```
repr_pretty_Hrepresentation(*args, **kwds)
Deprecated: Use Hrepresentation_str() instead. See trac ticket #24837 for details.
```

```
representative_point()
Return a “generic” point.

See also:
center().

OUTPUT:

A point as a coordinate vector. The point is chosen to be interior as far as 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)
```
(continues on next page)
restricted_automorphism_group (output='abstract')

Return the restricted automorphism group.

First, let the linear automorphism group be the subgroup of the affine group \( AGL(d, \mathbb{R}) = GL(d, \mathbb{R}) \times \mathbb{R}^d \) preserving the \( d \)-dimensional polyhedron. The affine group acts in the usual way \( \bar{x} \mapsto A\bar{x} + b \) 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

\[
Q = \{(x, y) \mid x \geq 0, \ y \geq 0\} \subset \mathbb{Q}^2
\]

Then the linear automorphism group is

\[
\text{Aut}(Q) = \left\{ \begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}, \begin{pmatrix} 0 & c \\ d & 0 \end{pmatrix} : a, b, c, d \in \mathbb{Q}_{>0} \right\} \subset GL(2, \mathbb{Q}) \subset E(d)
\]

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

\[
\text{Aut}(Q) = \left\{ \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \right\} \cong \mathbb{Z}_2
\]

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 \( \text{self}.\text{Vrepresentation}(0) \) and \( \text{self}.\text{Vrepresentation}(1) \).
  - "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:

2.4. Base classes for polyhedra 385
• [BSS2009]

EXAMPLES:

A cross-polytope example:

```python
sage: P = polytopes.cross_polytope(3)
sage: P.restricted_automorphism_group() == PermutationGroup([(3,4), (2,5), (1,2), (5,6), (1,6)])
True
sage: P.restricted_automorphism_group(output="permutation") == PermutationGroup([(2,3), (1,2), (3,4), (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]]]
We test groups for equality in a fool-proof way; they can have different generators, etc:

```python
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
``` 

24-cell example:

```python
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
``` 

Here is the quadrant example mentioned in the beginning:

```python
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)]
Also, the polyhedron need not be full-dimensional:

```python
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")
sage: G
```

(continues on next page)
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: 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()
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)]
```

The `output="matrixlist"` can be used over fields without a complete implementation of matrix groups:

```
sage: P = polytopes.dodecahedron(); P
A 3-dimensional polyhedron in (Number Field in sqrt5 with defining polynomial $x^2 - 5$) defined as the convex hull of 20 vertices
sage: G = P.restricted_automorphism_group(output="matrixlist")
sage: 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()
Permutation Group with generators [(2,3), (1,2)]
sage: len(P.restricted_automorphism_group(output="matrixlist"))
6
```
schlegel_projection (projection_dir=None, height=1.1)

Return the Schlegel projection.

- The polyhedron is translated such that its center() is at the origin.
- The vertices are then normalized to the unit sphere
- The normalized points are stereographically projected from a point slightly outside of the sphere.

INPUT:

- projection_direction – coordinate list/tuple/iterable or None (default). The direction of the Schlegel projection. For a full-dimensional polyhedron, the default is the first facet normal; Otherwise, the vector consisting of the first n primes is chosen.
- height – float (default: 1.1). How far outside of the unit sphere the focal point is.

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])
4

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')

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 integer. 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.
A Polyhedron object

```python
sage: cube = polytopes.cube()
sage: square_face = cube.faces(2)[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: Can not 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: hexaprism = polytopes.regular_polygon(6).prism()
sage: hexaprism.f_vector()
(1, 12, 18, 8, 1)
sage: square_face = hexaprism.faces(2)[0]
sage: stacked_hexaprism = hexaprism.stack(square_face)
sage: stacked_hexaprism.f_vector()
(1, 13, 22, 11, 1)

sage: hexaprism.stack(square_face,position=4)
Traceback (most recent call last):
...
ValueError: The chosen position is too large.
```

It is possible to stack on unbounded faces:

```python
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 (0, 2),
 A vertex at (1, 0),
 A ray in the direction (1, 1))
sage: Q.stack(E[1],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))
sage: Q.stack(E[2],1/2).Vrepresentation()
(A vertex at (0, 0),
 A vertex at (1, 0),
 A ray in the direction (1, 1),
 A vertex at (2, 0))
```
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))
```

See also:
join() direct_sum()

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 – (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]
```
Floating-point linear program:

```python
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
```

Irrational algebraic linear program over an embedded number field:

```python
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
```

Same example with floating point:

```python
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
```

Same example with a specific floating point solver:

```python
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
```

Irrational algebraic linear program over AA:

```python
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?
```

**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:**

```python
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
```

**triangulate (engine='auto', connected=True, fine=False, regular=None, star=None)**

Returns a triangulation of the polytope.

**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’), 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:

```python
sage: cube = polytopes.hypercube(3)
sage: triangulation = cube.triangulate(engine='internal') # to make doctest independent of TOPCOM
sage: triangulation
(<0,1,2,7>, <0,1,4,7>, <0,2,4,7>, <1,2,3,7>, <1,4,5,7>, <2,4,6,7>)
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
```

### 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:

```python
sage: cube = polytopes.hypercube(3)
sage: trunc_cube = cube.truncation()
sage: trunc_cube.n_vertices()
```
24
sage: trunc_cube.n_inequalities()
14

vertex_adjacency_matrix()
Return the binary matrix of vertex adjacencies.

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]

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 \((1, 0) x + 0 >= 0,)\)
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]

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:

In one dimension higher, that is for a half-space in 3 dimensions, there is no one-dimensional face. Hence nothing is adjacent:
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)
```

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, 1) A ray in the direction (0, 1)
(1, 0, 1, 0) A vertex at (0, 1)
(0, 1, 0, 1) A vertex at (1, 0)
(0, 0, 1, 0) A vertex at (3, 0)
(1, 0, 0, 1) 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, 1) A ray in the direction (0, 1)
(1, 0, 1, 0) A vertex at (0, 1)
(0, 0, 0, 1) A ray in the direction (1, 1)
(0, 0, 1, 0) A vertex at (3, 0)
(0, 0, 1, 1) A vertex at (4, 1)
```

`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: \( f(v) = v \cdot \text{dot}_\text{product}(f) \).
- `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:

```python
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
```

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 an 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:

```python
sage: P = polytopes.cube()
sage: G = P.vertex_facet_graph(); G
Digraph on 14 vertices
sage: G.vertices(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 (1, 0, 0) x + 1 >= 0]
sage: G.automorphism_group().is_isomorphic(P.face_lattice().hasse_diagram().automorphism_group())
```

(continues on next page)
(continued from previous page)

```
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)
False
sage: G.reverse_edges(G.edges())
sage: G.is_isomorphic(H)
True
```

**vertex_generator()**

Return a generator for the vertices of the polyhedron.

**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)]
```

**vertex_graph()**

Return a graph in which the vertices correspond to vertices of the polyhedron, and edges to edges.

**EXAMPLES:**

```
sage: g3 = polytopes.hypercube(3).vertex_graph(); g3
Graph on 8 vertices
sage: g3.automorphism_group().cardinality()
48
sage: s4 = polytopes.simplex(4).vertex_graph(); s4
Graph on 5 vertices
sage: s4.is_eulerian()
True
```

**vertices()**

Return all vertices of the polyhedron.

**OUTPUT:**

A tuple of vertices.

**EXAMPLES:**
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.

EXAMPLES:

```
sage: triangle = Polyhedron(vertices=[[0,1],[1,0],[1,1]])
sage: triangle.vertices_list()
[[0, 1], [1, 0], [1, 1]]
sage: a_simplex = Polyhedron(i eqs = [
    ....: [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
```

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.

EXAMPLES:

```
sage: triangle = Polyhedron(vertices=[[0,1],[1,0],[1,1]])
sage: triangle.vertices_matrix()
[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
```
volume\( (\text{measure}='\text{ambient}', \text{engine}='\text{auto}', **\text{kwds}) \)

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
- 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).
- **kwds -- keyword arguments that are passed to the triangulation engine.

OUTPUT:

The volume of the polytope.

EXAMPLES:

\[
\begin{align*}
sage: & \text{polytopes.hypercube(3).volume()} \\
     &= 8 \\
sage: & (\text{polytopes.hypercube(3)}\times2).\text{volume()} \\
     &= 64 \\
sage: & \text{polytopes.twenty\_four\_cell().volume()} \\
     &= 2
\end{align*}
\]

Volume of the same polytopes, using the optional package lrslib (which requires a rational polytope). For mysterious historical reasons, Sage casts lrs’s exact answer to a float:

\[
\begin{align*}
sage: & \text{I3 = polytopes.hypercube(3)} \\
sage: & \text{I3.volume(engine='lrs')} \quad \#\text{optional - lrslib} \\
     &= 8.0 \\
sage: & \text{C24 = polytopes.twenty\_four\_cell()} \\
sage: & \text{C24.volume(engine='lrs')} \quad \#\text{optional - lrslib} \\
     &= 2.0
\end{align*}
\]

If the base ring is exact, the answer is exact:

\[
\begin{align*}
sage: & \text{P5 = polytopes.regular\_polygon(5)} \\
sage: & \text{P5.volume()} \\
     &= 2.377641290737884? \\
\end{align*}
\]

\[
\begin{align*}
sage: & \text{polytopes.icosahedron().volume()} \\
     &= 5/12*\text{sqrt5} + 5/4 \\
sage: & \text{numerical\_approx(\_)} \quad \#\text{abs tol 1e9} \\
     &= 2.18169499062491
\end{align*}
\]

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')
sqrt(2)
sage: P.volume(measure='induced_rational')  # optional -- latte_int
1
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)[2].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
sage: Dexact = polytopes.dodecahedron()
sage: v = Dexact.faces(2)[0].as_polyhedron().volume(measure='induced', engine='internal'); v
-80*(55*sqrt(5) - 123)/sqrt(-6368*sqrt(5) + 14240)
sage: RDF(v)  # abs tol 1e-9
1.53406271079044
sage: Dinexact = polytopes.dodecahedron(exact=False)
sage: w = Dinexact.faces(2)[0].as_polyhedron().volume(measure='induced', engine='internal'); RDF(w)  # abs tol 1e-9
1.534062710738235
sage: [polytopes.simplex(d).volume(measure='induced') for d in range(1,5)] == [sqrt(d+1)/factorial(d) for d in range(1,5)]
True
sage: I = Polyhedron([[-3, 0], [0, 9]])
sage: I.volume(measure='induced')
3*sqrt(10)
sage: I.volume(measure='induced_rational')  # optional -- latte_int
1/2
sage: T = Polyhedron([[3, 0, 0], [0, 4, 0], [0, 0, 5]])
sage: T.volume(measure='induced')
1/2*sqrt(769)
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
```

The volume of a full-dimensional unbounded polyhedron is infinity:
sage: P = Polyhedron(vertices = [[1, 0], [0, 1]], rays = [[1, 1]])

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

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

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
tmp_filename
sage: filename = tmp_filename(ext='.ext')
sage: 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
tmp_filename
sage: filename = tmp_filename(ext='.ext')
sage: polytopes.cube().write_cdd_Vrepresentation(filename)

sage.geometry.polyhedron.base.is_Polyhedron(X)
Test whether X is a Polyhedron.

INPUT:

• X – anything.

OUTPUT:

Boolean.
EXAMPLES:

```python
sage: p = polytopes.hypercube(2)
sage: from sage.geometry.polyhedron.base import is_Polyhedron
sage: is_Polyhedron(p)
True
sage: is_Polyhedron(123456)
False
```

2.4.2 Base class for polyhedra over \( \mathbb{Q} \)

```python
class sage.geometry.polyhedron.base_QQ.Polyhedron_QQ(parent, Vrep, Hrep, **kwds)
Bases: sage.geometry.polyhedron.base.Polyhedron_base
```

Base class for Polyhedra over \( \mathbb{Q} \)

```python
integral_points_count(verbos=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; `False` by default) – whether to display verbose output.
- `use_Hrepresentation` - (boolean; `False` by default) – whether to send the H or V representation to LattE
- `preprocess` - (boolean; `True` by default) – 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:

```python
sage: P = polytopes.cube()
sage: P.integral_points_count()
27
sage: P.integral_points_count(explicit_enumeration_threshold=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.

```python
sage: (10000000000*P).integral_points_count(verbos=True) # optional - →latte_int This is LattE integrale... ... Total time:... 80000000120000000060000000001
```

We shrink the polyhedron a little bit:

```python
sage: Q = P*(8/9)
sage: Q.integral_points_count()
1
sage: Q.integral_points_count(explicit_enumeration_threshold=0) # optional - →latte_int
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: ...
```

"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
33
```

### 2.4.3 Base class for polyhedra over \( Z \)

```python
class sage.geometry.polyhedron.base_ZZ.Polyhedron_ZZ (parent, Vrep, Hrep, **kwds)
Bases: sage.geometry.polyhedron.base_QQ.Polyhedron_QQ
```

Base class for Polyhedra over \( Z \)

**Minkowski_decompositions**

*Deprecated:* Use `minkowski_decompositions()` instead. See trac ticket #23685 for details.

**ehrhart_polynomial**

(\texttt{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 \( \mathbb{R}^d \) and define \( L(P, t) = \#(tP \cap \mathbb{Z}^d) \). Then E. Ehrhart proved in 1962 that \( L \) coincides with a rational polynomial of degree \( d \) for integer \( t \). \( L \) is called the \textit{Ehrhart polynomial} of \( P \).

For more information see the Wikipedia article \texttt{Ehrhart_polynomial}.

**INPUT:**

- \texttt{verbose} - (boolean, default to \texttt{False}) if True, print the whole output of the \texttt{LattE} command.

The following options are passed to the \texttt{LattE} command, for details you should consult the \texttt{LattE} documentation:

- \texttt{dual} - (boolean) triangulate and signed-decompose in the dual space

- \texttt{irrational_primal} - (boolean) triangulate in the dual space, signed-decompose in the primal space using irrationalization.

- \texttt{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 ‘Irs’ 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

**Note:** Any additional argument is forwarded to LattE’s executable `count`. All occurrences of ‘_’ will be replaced with a ‘-‘.

**ALGORITHM:**
This method calls the program `count` from LattE integrale, a program for lattice point enumeration (see https://www.math.ucdavis.edu/~latte/).

**See also:**
`latte` the interface to LattE integrale

**EXAMPLES:**

```python
sage: P = Polyhedron(verts=[(0,0,0),(3,3,3),(-3,2,1),(1,-1,-2)])
sage: p = P.ehrhart_polynomial()  # optional - latte_int
sage: p  # optional - latte_int
7/2*t^3 + 2*t^2 - 1/2*t + 1
sage: p(1)                         # optional - latte_int
6
sage: len(P.integral_points())   # optional - latte_int
6
sage: p(2)                        # optional - latte_int
36
sage: len((2*P).integral_points()) # optional - latte_int
36
```

The unit hypercubes:

```python
sage: from itertools import product
sage: def hypercube(d):
....:     return Polyhedron(verts=list(product([0,1],repeat=d)))
sage: hypercube(3).ehrhart_polynomial()  # optional - latte_int
t^3 + 3*t^2 + 3*t + 1
sage: hypercube(4).ehrhart_polynomial()  # optional - latte_int
t^4 + 4*t^3 + 6*t^2 + 4*t + 1
sage: hypercube(5).ehrhart_polynomial()  # optional - latte_int
t^5 + 5*t^4 + 10*t^3 + 10*t^2 + 5*t + 1
sage: hypercube(6).ehrhart_polynomial()  # optional - latte_int
t^6 + 6*t^5 + 15*t^4 + 20*t^3 + 15*t^2 + 6*t + 1
```

An empty polyhedron:
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)
sage: list( P.fibration_generator(2) )
[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 \(\mathbb{Z}\)-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
```

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:

```python
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
```

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:

```python
sage: polytopes.cross_polytope(3).is_lattice_polytope()
True
sage: polytopes.regular_polygon(5).is_lattice_polytope()
False
```

**is_reflexive()**

EXAMPLES:

```python
sage: p = Polyhedron(vertices=[(1,0,0),(0,1,0),(0,0,1),(-1,-1,-1)], base_ring=ZZ)
sage: p.is_reflexive()
True
```

**minkowski_decompositions()**

Return all Minkowski sums that add up to the polyhedron.

**OUTPUT:**

A tuple consisting of pairs \((X, Y)\) of \(\mathbb{Z}\)-polyhedra that add up to \(\text{self}\). All pairs up to exchange of the summands are returned, that is, \((Y, X)\) is not included if \((X, Y)\) already is.

EXAMPLES:

```python
sage: square = Polyhedron(vertices=[(0,0),(1,0),(0,1),(1,1)])
sage: square.minkowski_decompositions()
((A 0-dimensional polyhedron in \(\mathbb{Z}^2\) defined as the convex hull of 1 vertex, 
  A 2-dimensional polyhedron in \(\mathbb{Z}^2\) defined as the convex hull of 4 
  \text{vertices}), 
  (A 1-dimensional polyhedron in \(\mathbb{Z}^2\) defined as the convex hull of 2 vertices, 
  A 1-dimensional polyhedron in \(\mathbb{Z}^2\) defined as the convex hull of 2 
  \text{vertices}))

Example from http://cgi.di.uoa.gr/~amantzaf/geo/

```python
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 \(\mathbb{Z}^2\) defined as the convex hull of 1 vertex, 
  A 2-dimensional polyhedron in \(\mathbb{Z}^2\) defined as the convex hull of 2 
  \text{vertices}, 
  A 1-dimensional polyhedron in \(\mathbb{Z}^2\) defined as the convex hull of 2 vertices, 
  A 1-dimensional polyhedron in \(\mathbb{Z}^2\) defined as the convex hull of 2 
  \text{vertices}))
```
A 2-dimensional polyhedron in \( \mathbb{Z}^2 \) defined as the convex hull of 7 vertices,
(A 2-dimensional polyhedron in \( \mathbb{Z}^2 \) defined as the convex hull of 4 vertices,
A 2-dimensional polyhedron in \( \mathbb{Z}^2 \) defined as the convex hull of 4 vertices,
(A 1-dimensional polyhedron in \( \mathbb{Z}^2 \) defined as the convex hull of 2 vertices,
A 2-dimensional polyhedron in \( \mathbb{Z}^2 \) defined as the convex hull of 7 vertices),
(A 2-dimensional polyhedron in \( \mathbb{Z}^2 \) defined as the convex hull of 5 vertices,
A 2-dimensional polyhedron in \( \mathbb{Z}^2 \) defined as the convex hull of 4 vertices,
(A 1-dimensional polyhedron in \( \mathbb{Z}^2 \) defined as the convex hull of 2 vertices,
A 2-dimensional polyhedron in \( \mathbb{Z}^2 \) defined as the convex hull of 7 vertices),
(A 2-dimensional polyhedron in \( \mathbb{Z}^2 \) defined as the convex hull of 5 vertices,
A 2-dimensional polyhedron in \( \mathbb{Z}^2 \) defined as the convex hull of 4 vertices,
(A 1-dimensional polyhedron in \( \mathbb{Z}^2 \) defined as the convex hull of 2 vertices,
A 2-dimensional polyhedron in \( \mathbb{Z}^2 \) defined as the convex hull of 7 vertices),
(A 1-dimensional polyhedron in \( \mathbb{Z}^2 \) defined as the convex hull of 2 vertices,
A 2-dimensional polyhedron in \( \mathbb{Z}^2 \) defined as the convex hull of 6 vertices)

```python
sage: [ len(square.dilation(i).minkowski_decompositions())
....:     for i in range(6) ]
[1, 2, 5, 8, 13, 18]
sage: [ ceiling((i^2+2*i-1)/2)+1 for i in range(10) ]
[1, 2, 5, 8, 13, 18, 25, 32, 41, 50]
```

**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:**

```python
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 \( \mathbb{Z}^3 \) 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()
```

(continues on next page)
2.4.4 Base class for polyhedra over RDF.

class sage.geometry.polyhedron.base_RDF.Polyhedron_RDF(parent, Vrep, Hrep, **kwds)
Bases: sage.geometry.polyhedron.base.Polyhedron_base

Base class for polyhedra over RDF.

2.5 Backends for Polyhedra

2.5.1 The cdd backend for polyhedral computations

class sage.geometry.polyhedron.backend_cdd.Polyhedron_QQ_cdd(parent, Vrep, Hrep, **kwds)
Bases: sage.geometry.polyhedron.backend_cdd.Polyhedron_cdd, sage.geometry.polyhedron.base_QQ.Polyhedron_QQ

Polyhedra over QQ with cdd

INPUT:

• parent – the parent, an instance of Polyhedra.
• Vrep – a list [vertices, rays, lines] or None.
• Hrep – a 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
sage: Polyhedron_QQ_cdd(parent, [[(1,0),(0,1),(0,0)], [], []], None, \n←verbose=False)
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices

class sage.geometry.polyhedron.backend_cdd.Polyhedron_RDF_cdd(ambient_dim, Vrep, Hrep, **kwds)
Bases: sage.geometry.polyhedron.backend_cdd.Polyhedron_cdd, sage.geometry.polyhedron.base_RDF.Polyhedron_RDF

Polyhedra over RDF with cdd

INPUT:

• ambient_dim – integer. The dimension of the ambient space.
• Vrep – a list [vertices, rays, lines] or None.
• Hrep – a 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 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

class sage.geometry.polyhedron.backend_cdd.Polyhedron_cdd(parent, Vrep, Hrep, **kwds)

Bases: sage.geometry.polyhedron.base.Polyhedron_base
Base class for the cdd backend.

2.5.2 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:

```python
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)
```

class sage.geometry.polyhedron.backend_field.Polyhedron_field(parent, Vrep, Hrep, Vrep_minimal=None, Hrep_minimal=None, **kwds)

Bases: sage.geometry.polyhedron.base.Polyhedron_base
Polyhedra over all fields supported by Sage

INPUT:

- `Vrep` – a list [vertices, rays, lines] or None.
- `Hrep` – a list [ieqs, eqns] or None.

EXAMPLES:

```python
sage: p = Polyhedron(vertices=[(0,0),(AA(2).sqrt(),0),(0,AA(3).sqrt())], 
   ....: rays=[(1,1)], lines=[], backend='field', base_ring=AA)
```

2.5.3 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

```python
class sage.geometry.polyhedron.backend_normaliz.Polyhedron_QQ_normaliz:
    Bases: sage.geometry.polyhedron.backend_normaliz.Polyhedron_normaliz, sage.geometry.polyhedron.base_QQ.Polyhedron_QQ
    Polyhedra over $\mathbb{Q}$ with normaliz.
    INPUT:
    - `Vrep` - a list [vertices, rays, lines] or None
    - `Hrep` - a 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(skip='_test_pickling')
```

```python
class sage.geometry.polyhedron.backend_normaliz.Polyhedron_ZZ_normaliz:
    Bases: sage.geometry.polyhedron.backend_normaliz.Polyhedron_normaliz, sage.geometry.polyhedron.base_ZZ.Polyhedron_ZZ
    Polyhedra over $\mathbb{Z}$ with normaliz.
    INPUT:
    - `Vrep` - a list [vertices, rays, lines] or None
    - `Hrep` - a 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(skip='_test_pickling')
```

2.5. Backends for Polyhedra
class sage.geometry.polyhedron.backend_normaliz.Polyhedron_normaliz(parent, Vrep, Hrep, normaliz_cone=None, **kwds)

Bases: sage.geometry.polyhedron.base.Polyhedron_base

Polyhedra with normaliz

INPUT:

• parent – Polyhedra the parent
• Vrep – a list [vertices, rays, lines] or None; the V-representation of the polyhedron; if None, the polyhedron is determined by the H-representation
• Hrep – a 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)],
                   # optional - pynormaliz
                   ....: lines=[], backend='normaliz')

sage: TestSuite(p).run(skip='_test_pickling')

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

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 - pynormaliz
                   ....: backend='normaliz')

sage: P.n_inequalities()

1

(An equation (-1, 0) x + 1 == 0,)

The empty polyhedron:
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”:

```python
sage: P = Polyhedron(ieqs=[[1, 0, 2], [3, 0, -2], [3, 2, -2]], backend='normaliz')
    # optional - pynormaliz
    ....: backend='normaliz')
sage: PI = P.integral_hull()
    # optional - pynormaliz
sage: P.plot(color='yellow') + PI.plot(color='green')
    # optional - pynormaliz
Graphics object consisting of 10 graphics primitives
sage: PI.Vrepresentation()
    # optional - pynormaliz
(A vertex at (-1, 0), A vertex at (0, 1), A ray in the direction (1, 0))
```

Nonpointed case:

```python
sage: P = Polyhedron(ieqs=[[1/2, 1/3]], backend='normaliz')
    ......: lines=[[-1, 1]], backend='normaliz')
sage: PI = P.integral_hull()
    # optional - pynormaliz
sage: PI.Vrepresentation()
    # optional - pynormaliz
(A vertex at (1, 0),
 A ray in the direction (1, 0),
 A line in the direction (-1, 1))
```

Empty polyhedron:

```python
sage: P = Polyhedron(backend='normaliz')
    # optional - pynormaliz
sage: PI = P.integral_hull()
    # optional - pynormaliz
sage: PI.Vrepresentation()
    # optional - pynormaliz
()
```

integral_points (threshold=10000)

Return the integral points in the polyhedron.
Uses either the naïve 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 \texttt{ValueError} is raised.

EXAMPLES:

```python
sage: Polyhedron(vertices=[(-1,-1), (1,0), (1,1), (0,1)], # optional - pynormaliz
    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)], # optional - pynormaliz
    backend='normaliz')
sage: simplex.integral_points() # optional - pynormaliz
((-2, -3, -11), (0, 0, -2), (1, 2, 3), (2, 3, 7))
sage: point = Polyhedron([2,3,7], # optional - pynormaliz
    backend='normaliz')
sage: point.integral_points() # optional - pynormaliz
((2, 3, 7),)
sage: empty = Polyhedron(backend='normaliz')
sage: empty.integral_points() # optional - pynormaliz
()
```

The polyhedron need not be full-dimensional:

```python
sage: simplex = Polyhedron([[1,2,3,5), (2,3,7,5), (-2,-3,-11,5)], # optional - pynormaliz
    backend='normaliz')
sage: simplex.integral_points() # optional - pynormaliz
((-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() # optional - pynormaliz
((2, 3, 7),)
sage: empty = Polyhedron(backend='normaliz')
sage: empty.integral_points() # optional - pynormaliz
()
```

Here is a simplex where the naïve algorithm of running over all points in a rectangular bounding box no longer works fast enough:

```python
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 # optional - pynormaliz
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 5 vertices
sage: len(simplex.integral_points()) # optional - pynormaliz
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):
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')  # optional - pynormaliz
sage: all(P.contains(p) for p in pts1)  # optional - pynormaliz
True
sage: pts2 = LatticePolytope(v).points()  # PALP
sage: set(pts1) == set(pts2)  # optional - pynormaliz
True
```

2.5.4 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: sage.geometry.polyhedron.backend_polymake.Polyhedron_polytope, sage.
geometry.polyhedron.base_QQ.Polyhedron_QQ

Polyhedra over Q with polymake.

INPUT:

• Vrep—a list [vertices, rays, lines] or None
• `Hrep` – a list `[ieqs, eqns]` or `None`

**EXAMPLES:**

```python
sage: p = Polyhedron(vertices=[(0,0),(1,0),(0,1)],
   ....: backend='polymake', base_ring=QQ)
```

```python
sage: TestSuite(p).run(skip='_test_pickling')
```

```python
class sage.geometry.polyhedron.backend_polymake.Polyhedron_ZZ_polymake(
    parent, Vrep, Hrep, polymake_polytope=None, **kwds)
```

Polyhedra over \( \mathbb{Z} \) with polymake.

**INPUT:**

• `Vrep` – a list `[vertices, rays, lines]` or `None`

• `Hrep` – a list `[ieqs, eqns]` or `None`

**EXAMPLES:**

```python
sage: p = Polyhedron(vertices=[(0,0),(1,0),(0,1)],
   ....: backend='polymake', base_ring=ZZ)
```

```python
sage: TestSuite(p).run(skip='_test_pickling')
```

```python
class sage.geometry.polyhedron.backend_polymake.Polyhedron_polymake(
    parent, Vrep, Hrep, polymake_polytope=None, **kwds)
```

Polyhedra with polymake

**INPUT:**

• `parent` – `Polyhedra` the parent

• `Vrep` – a list `[vertices, rays, lines]` or `None`; the V-representation of the polyhedron; if `None`, the polyhedron is determined by the H-representation

• `Hrep` – a 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:**
Two ways to get the full space:

```
sage: Polyhedron(eqns=[[0, 0, 0]], backend='polymake')  # optional - polymake
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 - polymake
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='polymake')
....: backend='polymake')
sage: P.n_inequalities()  # optional - polymake
1
sage: P.equations()  # optional - polymake
(An equation (x, y) - 1 == 0,)
```

The empty polyhedron:

```
sage: P=Polyhedron(ieqs=[[-2, 1, 1], [-3, -1, -1], [-4, 1, -2]], backend='polymake')
....: backend='polymake')
sage: P  # optional - polymake
The empty polyhedron in QQ^2
sage: P.Vrepresentation()  # optional - polymake
()
```

Quadratic fields work:

```
sage: V = polytopes.dodecahedron().vertices_list()
sage: Polyhedron(backend='polymake')  # optional - polymake
A 3-dimensional polyhedron in (Number Field in sqrt5 with defining polynomial x^2 - 5)^3 defined as the convex hull of 20 vertices
```
2.5.5 The PPL (Parma Polyhedra Library) backend for polyhedral computations

```python
class sage.geometry.polyhedron.backend_ppl.Polyhedron_QQ_ppl(parent, Vrep, Hrep, **kwds):
    Bases: sage.geometry.polyhedron.backend_ppl.Polyhedron_ppl, sage.geometry.polyhedron.base.QQ.Polyhedron_QQ

Polyhedra over $\mathbb{Q}$ with ppl

INPUT:

• Vrep -- a list [vertices, rays, lines] or None.
• Hrep -- a 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(skip='_test_pickling')
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
OUTPUT:
A new Generator representing the line.

Raises a ValueError if the homogeneous part of `expression` represents the origin of the vector space.

Examples:

```python
>>> from ppl import Generator, Variable
>>> y = Variable(1)
>>> Generator.line(2*y)
line(0, 1)
>>> Generator.line(y)
line(0, 1)
>>> Generator.line(1)
Traceback (most recent call last):
... ValueError: PPL::line(e):
e == 0, but the origin cannot be a line.
```

```
sage.geometry.polyhedron.backend_ppl.point(*args, **kwds)
Construct a point.

INPUT:
- `expression` - a Linear_Expression or something convertible to it (Variable or integer).
- `divisor` - an integer.

OUTPUT:
A new Generator representing the point.

 Raises a ValueError if `divisor==0`.

Examples:

```python
>>> from ppl import Generator, Variable
>>> y = Variable(1)
>>> Generator.point(2*y+7, 3)
point(0/3, 2/3)
>>> Generator.point(y+7, 3)
point(0/3, 1/3)
>>> Generator.point(7, 3)
point()
>>> Generator.point(0, 0)
Traceback (most recent call last):
... ValueError: PPL::point(e, d):
d == 0.
```
```

```
sage.geometry.polyhedron.backend_ppl.ray(*args, **kwds)
Construct a ray.

INPUT:
- `expression` - a Linear_Expression or something convertible to it (Variable or integer).

OUTPUT:
A new Generator representing the ray.
```

2.5. Backends for Polyhedra
Raises a `ValueError` if the homogeneous part of `expression` represents the origin of the vector space.

Examples:

```python
>>> from ppl import Generator, Variable
>>> y = Variable(1)
>>> Generator.ray(2*y)
ray(0, 1)
>>> Generator.ray(y)
ray(0, 1)
>>> Generator.ray(1)
Traceback (most recent call last):
...  
ValueError: PPL::ray(e):
  e == 0, but the origin cannot be a ray.
```

### 2.5.6 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 back-end 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:**

```python
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

\[
\begin{bmatrix}
 1 & 0 & 1 \\
 0 & 1 & 1 \\
-1 & -1 & 1 \\
\end{bmatrix}
\]
\[
\begin{bmatrix}
 2/3 & -1/3 & -1/3 \\
-1/3 & 2/3 & -1/3 \\
 1/3 & 1/3 & 1/3 \\
\end{bmatrix}
\]
```

The implementation works over any exact field that is embedded in \( \mathbb{R} \), for example:

```python
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),
                   (-AA(3).sqrt(),-AA(2).sqrt(),1)])
sage: alg = StandardAlgorithm(A)
sage: alg.run().R
[(-0.4177376677004119?, 0.5822623322995881?, 0.4177376677004119?),
 (-0.2411809548974793?, -0.2411809548974793?, 0.2411809548974793?),
 (0.07665629029830300?, 0.07665629029830300?, 0.2411809548974793?),
 (0.5822623322995881?, -0.4177376677004119?, 0.4177376677004119?)]
```

```python
class sage.geometry.polyhedron.double_description.DoubleDescriptionPair(
    problem, A_rows, R_cols)
```

Base class for a double description pair \((A, R)\)
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:


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)])
```

`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)])
```

2.5. Backends for Polyhedra
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)
```

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 0]
[0 1 0]
A = [1 0 0], R = [1 0 -1]
[0 0 1]
[0 0 1]
```

double_description_pair() 
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
[ 1 1]
[-1 1]
A = [ 1/2 -1/2], R = [ 1/2 1/2]
sage: DD.first_coordinate_plane()
double description pair (A, R) defined by
[ 1 1]
[-1 0]
[-1 0]
[ 1 0]
```

420 Chapter 2. Polyhedral computations
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:**

```python
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]
```

is_extremal(ray)

Test whether the ray is extremal.

**EXAMPLES:**

```python
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
```

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:**

```python
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: 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
```

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 $\mathbb{Q}$.

**EXAMPLES:**

2.5. Backends for Polyhedra 421
sage: from sage.geometry.polyhedron.double_description import \n....:   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)])
sage: 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)}

class sage.geometry.polyhedron.double_description.Problem(A)

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 ≥ 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))

A_matrix()

Return the defining matrix A.

OUTPUT:
Matrix whose rows are the inequalities.

EXAMPLES:

```python
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]
```

**base_ring()**
Return the base field.

**OUTPUT:**
A field.

**EXAMPLES:**

```python
sage: A = matrix(AA, [(1, 1), (-1, 1)])
sage: from sage.geometry.polyhedron.double_description import Problem.double_description
sage: Problem(A).base_ring()
Algebraic Real Field
```

**dim()**
Return the ambient space dimension.

**OUTPUT:**
Integer. The ambient space dimension of the cone.

**EXAMPLES:**

```python
sage: A = matrix(QQ, [(1, 1), (-1, 1)])
sage: from sage.geometry.polyhedron.double_description import Problem.double_description
sage: 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 :class:`DoubleDescriptionPair`.

**OUTPUT:**
A pair consisting of a :class:`DoubleDescriptionPair` instance and the tuple of remaining unused inequalities.

**EXAMPLES:**

```python
sage: A = matrix([(-1, 1), (-1, 2), (1/2, -1/2), (1/2, 2)])
sage: from sage.geometry.polyhedron.double_description import Problem.double_description
sage: DD, remaining = Problem(A).initial_pair()
sage: DD.verify()
sage: remaining
[(1/2, -1/2), (1/2, 2)]
```
pair_class
alias of DoubleDescriptionPair

class sage.geometry.polyhedron.double_description.StandardAlgorithm(A)
   Bases: sage.geometry.polyhedron.double_description.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)]

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
\[
\begin{bmatrix}
  0 & 1 & 0 \\
  1 & 0 & 0 \\
  0 & -1 & 1 \\
-1 & 0 & 1
\end{bmatrix}, \begin{bmatrix} 0 & 0 & 1 & 1 \\
  1 & 0 & 1 & 0 \\
  1 & 1 & 1 & 1
\end{bmatrix}
\]

class sage.geometry.polyhedron.double_description.StandardDoubleDescriptionPair(problem, A_rows, R_cols)
   Bases: sage.geometry.polyhedron.double_description.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

\[
\begin{pmatrix}
-1 & 1 & 0 \\
-1 & 2 & 1 \\
1/2 & -1/2 & -1 \\
1 & 0 & 0
\end{pmatrix}
\]

\[
\begin{pmatrix}
1 & 1 & 0 & 0 \\
1 & 1 & 1 & 1 \\
0 & -1 & -1/2 & -2 \\
0 & 3 & -1/3
\end{pmatrix}
\]

```
sage: from sage.geometry.polyhedron.double_description import random_inequalities
sage: P = random_inequalities(5, 10)
sage: P.run().verify()
```

### 2.5.7 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 \((1, * \ldots, *)\). 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.

```
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]
```

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]
```

(continues on next page)
class sage.geometry.polyhedron.double_description_inhomogeneous.Hrep2Vrep(
    base_ring, dim, inequalities, equations)

Bases:
    sage.geometry.polyhedron.double_description_inhomogeneous.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:

```python
sage: from sage.geometry.polyhedron.double_description_inhomogeneous import *

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)], [])
[-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| 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, [], [])  # empty
[]
```
**verify** *(inequalities, equations)*

Compare result to PPL if the base ring is \(\mathbb{Q}\).

This method is for debugging purposes and compares the computation with another backend if available.

**INPUT:**

- inequalities, equations – see \texttt{Hrep2Vrep}.

**EXAMPLES:**

```python
sage: from sage.geometry.polyhedron.double_description_inhomogeneous import *
   ...: Hrep2Vrep
sage: H = Hrep2Vrep(QQ, 1, [(1,2)], [])
```

```python
sage: H.verify([(1,2)], [])
```

---

**class** \texttt{sage.geometry.polyhedron.double_description_inhomogeneous.PivotedInequalities}(base\_ring, \(\text{dim}\))

**Bases:** \texttt{sage.structure.sage_object.SageObject}

Base class for inequalities that may contain linear subspaces

**INPUT:**

- base\_ring – a field.
- \(\text{dim}\) – integer. The ambient space dimension.

**EXAMPLES:**

```python
sage: from sage.geometry.polyhedron.double_description_inhomogeneous import *
   ...: 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]
```

---

**class** \texttt{sage.geometry.polyhedron.double_description_inhomogeneous.Vrep2Hrep}(base\_ring, \(\text{dim}\), \text{vertices}, \text{rays}, \text{lines})

**Bases:** \texttt{sage.geometry.polyhedron.double_description_inhomogeneous.PivotedInequalities}

Convert V-representation to a minimal H-representation.

**INPUT:**

- base\_ring – a field.
- \(\text{dim}\) – integer. The ambient space dimension.
- \text{vertices} – list of vertices. Each vertex is given as list of \(\text{dim}\) coordinates.
- \text{rays} – list of rays. Each ray is given as list of \(\text{dim}\) 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]

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]

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)])
```

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)
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. It is available as an optional package for Sage, and you can install it with the shell command

```
sage -i topcom
```

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
```

A 2-dimensional point configuration:

```
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.
```

A triangulation of it:

```
sage: t = p.triangulate()  # a single triangulation
sage: t
(<1,3,4>, <2,3,4>)
```

(continues on next page)
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()
sage: 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.
```
A 3-dimensional point configuration:

```python
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)
Graphics3d Object
```

The standard example of a non-regular triangulation (requires TOPCOM):

```python
sage: PointConfiguration.set_engine('topcom')  # optional - topcom
sage: p = PointConfiguration([[-1,-5/9],[0,10/9],[1,-5/9],[-2,-10/9],[0,20/9],[2,-10/9]])
sage: regular = p.restrict_to_regular_triangulations(True).triangulations_list()  # optional - topcom
sage: nonregular = p.restrict_to_regular_triangulations(False).triangulations_list()  # optional - topcom
sage: len(regular)  # optional - topcom
16
sage: len(nonregular)  # optional - topcom
2
sage: nonregular[0].plot(aspect_ratio=1, axes=False)  # optional - topcom
Graphics object consisting of 25 graphics primitives
```

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:

```python
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
```

3.1. Triangulations of a point configuration
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: 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=True,
                                           fine=False,
                                           regular=None,
                                           star=False,
                                           defined_affine=False)

Bases:  sage.structure.unique_representation.UniqueRepresentation, sage.
        geometry.triangulation.base.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:

```python
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: p.triangulate()  # a single triangulation
(<1,3,4>, <2,3,4>)
```

**Element**

alias of `sage.geometry.triangulation.element.Triangulation`

**Gale_transform** *(points=None)*

Return the Gale transform of self.

**INPUT:**

- points – a 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.

**OUTPUT:**

A matrix over `base_ring()`.

**EXAMPLES:**

```python
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]
```

**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:**
The 3d analog:

```python
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>)),
```

A 2d flip on the base of the pyramid over a square:

```python
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>)),
```

`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

\[ \{0, 1, \ldots, n - 1\} = C_+ \cup C_0 \cup C_- \]

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 \( \vec{p}_i = (p_1, \ldots, p_k, 1) \) 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:**

```python
sage: p = PointConfiguration([(0,0),(+1,0),(-1,0),(0,+1),(0,-1)])
sage: 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: list( p.circuits_support() )
[(0, 3, 4), (0, 1, 2), (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 \texttt{Point} or None (default). A specific point to start with when picking the simplex vertices.
- **point_order** – a list or tuple of (some or all) \texttt{Point}s or None (default).

OUTPUT:

A tuple of points that span a simplex of dimension \texttt{dim()}. If \texttt{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 \texttt{point_order} is specified, the simplex is greedily constructed by considering the points in this order. The \texttt{large} option and \texttt{initial_point} is ignored in this case. The \texttt{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, 0), 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))
sage: # lower-dimensional example:
sage: pc.contained_simplex(point_order = [pc[0],pc[3],pc[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 convex hull in ZZ^2 defined as the convex hull of 4 vertices

distance (x, y)
Returns 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:**

```python
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),(1,1,1),(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]
```

**distance_FS** $(x, y)$

Returns 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-Study 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:**

```python
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]
```

**distance_affine** $(x, y)$

Returns 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:**

```python
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]
```
**exclude_points** *(point_idx_list)*

Return a new point configuration with the given points removed.

**INPUT:**

- point_idx_list – a list of integers. The indices of points to exclude.

**OUTPUT:**

A new *PointConfiguration* with the given points removed.

**EXAMPLES:**

```python
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)]
sage: p.exclude_points( p.face_interior(codim=1) ).points()
(P(-1, 0), P(0, 0), P(1, -1), P(1, 1))
```

**face_codimension** *(point)*

Return the smallest $d \in \mathbb{Z}$ such that point is contained in the interior of a codimension-$d$ face.

**EXAMPLES:**

```python
tsage: triangle = PointConfiguration([[0,0], [1,-1], [1,0], [1,1]])
tsage: triangle.point(2)
P(1, 0)
tsage: triangle.face_codimension(2)
1
tsage: triangle.face_codimension( [1,0] )
1
```

This also works for degenerate cases like the tip of the pyramid over a square (which saturates four inequalities):

```python
tsage: pyramid = PointConfiguration([[1,0,0],[0,1,1],[0,1,-1],[0,-1,-1],[0,-1,˓→1]])
tsage: pyramid.face_codimension(0)
3
```

**face_interior** *(dim=None, codim=None)*

Return points by the codimension of the containing face in the convex hull.

**EXAMPLES:**

```python
tsage: triangle = PointConfiguration([[-1,0], [0,0], [1,-1], [1,0], [1,1]])
tsage: triangle.face_interior()
((1,), (3,), (0, 2, 4))
tsage: triangle.face_interior(dim=0)   # the vertices of the convex hull
(0, 2, 4)
tsage: triangle.face_interior(codim=1)   # interior of facets
(3,)
```

**farthest_point** *(points, among=None)*

Return the point with the most distance from points.

**INPUT:**

- points
- among – an optional subset of the point configuration.
• points – a list of points.
• among – a 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)
```

**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>)
```

**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, 1, 0, 0],
....: [ 0, 0, 0, 0, 0, 0, 0,-1, 1, 0, 0, 0, 0],
....: [ 0, 0, 0, 0, 0, 0,-1, 0, 1, 0, 0, 0, 0],
....: [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
....: [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
....: [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
....: [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
....: [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
....: [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
....: [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
....: [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
....: [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
....: [ 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]))
sage: p = PointConfiguration(U.columns())
sage: triangulation = p.placing_triangulation(); triangulation
(<0,2,3,4,6,12>, <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>,
```
```
\[
\begin{align*}
&<1,3,6,7,11,13>, <1,3,6,11,12,13>, <1,4,6,7,11,13>, <1,4,6,11,12,13>, \\
&<3,4,6,7,11,12>, <3,4,7,11,12,13>, <3,6,7,11,12,13>, <4,6,7,11,12,13>) \\
\text{sage: } &\text{sage: sum}(p\text{.volume}(t) \text{ for } t \text{ in triangulation}) \\
&42 \\
\text{sage: } &\text{sage: p0 = PointConfiguration([[0,0],[+1,0],[-1,0],[0,+1],[0,-1]])} \\
&(<1,2,3>, <1,2,4>) \\
\text{sage: } &\text{sage: p0.pushing_triangulation(point_order=[1,2,0,3,4])} \\
&(<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>) \\
\text{sage: } &\text{sage: # the same triangulation with renumbered points 0->4, 1->0, etc.:} \\
\text{sage: } &\text{sage: p1 = PointConfiguration([(+1,0),(-1,0),(0,+1),(0,-1),(0,0)])} \\
\text{sage: } &\text{sage: p1.pushing_triangulation(point_order=[4,0,1,2,3])} \\
&(<0,2,4>, <0,3,4>, <1,2,4>, <1,3,4>)
\end{align*}
\]

plot (**kwds)

Produce a graphical representation of the point configuration.

EXAMPLES:

\[
\begin{align*}
\text{sage: } &\text{sage: p = PointConfiguration([[0,0],[0,1],[1,0],[1,1],[-1,-1]])} \\
\text{sage: } &\text{sage: p.plot(axes=False)} \\
\text{Graphics object consisting of 5 graphics primitives} \\
\end{align*}
\]

\[
\begin{align*}
\text{sage: } &\text{sage: p0 = PointConfiguration([[0,0],[0,1],[1,0],[1,1],[-1,-1]])} \\
\text{sage: } &\text{sage: p.plot(axes=False)} \\
\text{Graphics object consisting of 5 graphics primitives} \\
\end{align*}
\]

positive_circuits (**negative)**

Returns 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:

\* \textbf{negative} – integer. The indices of points.

OUTPUT:

A tuple of all circuits with \(C_- = \text{negative}\).

EXAMPLES:

\[
\begin{align*}
\text{sage: } &\text{sage: p = PointConfiguration([[(1,0,0),(0,1,0),(0,0,1),(-2,0,-1),(-2,-1,0),(-3,} \\
\text{sage: } &\text{sage: } \rightarrow-1,-1),(1,1,1),(-1,0,0),(0,0,0)])} \\
\text{sage: } &\text{sage: p.positive_circuits(8)} \\
&((0, 7), (0, 1, 4), (0, 2, 3), (0, 5, 6), (0, 1, 2, 5), (0, 3, 4, 6))
\end{align*}
\]
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:**

```python
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>),
 <0,1,4>, <1,2,3>, <1,3,4>)
sage: pc.placing_triangulation(point_order=(3,2,1,4,0))
((<0,1,4>, <1,2,3>, <1,3,4>),
 <0,1,4>, <1,3,4>)
sage: pc.placing_triangulation(point_order=[pc[1],pc[3],pc[4],pc[0]])
((<0,1,4>, <1,3,4>),
 <0,1,4>, <1,3,4>)
```

```python
sage: U=matrix(
    ....: [ 0, 0, 0, 0, 0, 2, 4,-1, 1, 1, 0, 0, 1, 0],
    ....: [ 0, 0, 0, 1, 0, -1, 0, 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, -1, 1, 0, 0, 0, 0, 0],
    ....: [ 0, 0, 0, 1, 0, 0, 0, 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>),
 <0,1,4>, <0,1,4>, <0,2,3>, <0,2,4>)
sage: sum(p.volume(t) for t in triangulation)
42
```

```python
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>),
 <0,1,4>, <0,2,3>, <0,2,4>)
sage: p0.pushing_triangulation(point_order=[0,1,2,3,4])
((<1,2,3>, <1,2,4>),
 <0,1,4>, <0,2,3>, <0,2,4>)
sage: # 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>),
 <0,1,4>, <0,2,3>, <0,2,4>)
```

Restrict to connected triangulations.

**NOTE:**

Finding non-connected triangulations requires the optional TOPCOM package.

**INPUT:**

```python
sage: restrict_to_connected_triangulations(connected=True)
```

• *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:**

```python
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
```  

```python
sage: PointConfiguration.set_engine('topcom')  # optional - 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
```  

```
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:**

```python
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.
```  

```python
sage: len(p.triangulations_list())
4
sage: p_fine = p.restrict_to_fine_triangulations()
```  

```python
sage: len(p_fine.triangulations_list())
4
sage: 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]])
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: PointConfiguration.set_engine('topcom')  # optional - 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')
```

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(star=None)  # 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
```
restricted_automorphism_group()

Return the restricted automorphism group.

First, let the linear automorphism group be the subgroup of the affine group $AGL(d, \mathbb{R}) = GL(d, \mathbb{R}) \ltimes \mathbb{R}^d$ 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 non-negative 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()
sage: G == PermutationGroup([(3,5)], [(2,3),(4,5)], [(2,4)])
True
sage: DihedralGroup(4).is_isomorphic(G)
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()
Permutation Group with generators [(3,5)]
sage: DihedralGroup(1).is_isomorphic(_)
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 3 5 3 5)
[3 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 \((5, 1, 4, 4, 1)\))

```python
sage: poly.Hrepresentation()
(An equation \((0, 0, 1, 2, 1) \cdot x - 13 == 0\),
An equation \((1, 0, 0, 2, 2) \cdot x - 15 == 0\),
An equation \((0, 1, 0, -3, -2) \cdot x + 13 == 0\),
An inequality \((0, 0, 0, -1, -1) \cdot x + 7 \geq 0\),
An inequality \((0, 0, 0, 1, 0) \cdot x - 2 \geq 0\),
An inequality \((0, 0, 0, -2, -1) \cdot x + 11 \geq 0\),
An inequality \((0, 0, 0, 0, 1) \cdot x - 1 \geq 0\),
An inequality \((0, 0, 0, 3, 2) \cdot x - 14 \geq 0\))
```

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’), TOPCOM is used if it is available and internal routines otherwise.

**EXAMPLES:**

```python
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()            # optional - topcom
(<1,3,4>, <2,3,4>)
sage: p.set_engine('topcom')    # optional - topcom
sage: p.triangulate()            # optional - topcom
(<0,1,2>, <0,1,4>, <0,2,4>, <1,2,3>)
sage: p.set_engine('internal')  # optional - topcom
```

```
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:**

```python
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)
```

(continues on next page)
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.
```

**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. 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)]
```

Using TOPCOM yields a different, but equally good, triangulation:

```
sage: p.set_engine('topcom')               # optional - topcom
sage: p.triangulate()                      # optional - topcom
(<0,1,2>, <0,1,4>, <0,2,4>, <1,2,3>)
sage: list( p.triangulate() )             # optional - topcom
[(0, 1, 2), (0, 1, 4), (0, 2, 4), (1, 2, 3)]
```

**triangulations** *(verbose=False)*

Returns 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()
```

(continues on next page)
[(<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: p.set_engine('topcom')  # optional - topcom
sage: iter = p.triangulations()  # optional - topcom
sage: next(iter)  # optional - topcom
(<0,1,2>, <0,1,4>, <0,2,4>, <1,2,3>)
sage: next(iter)  # optional - topcom
(<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>)
sage: next(iter)  # optional - topcom
(<1,2,3>, <1,2,4>)
sage: next(iter)  # optional - topcom
(<1,3,4>, <2,3,4>)
sage: p.set_engine('internal')  # optional - topcom

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')  # optional - topcom
sage: p.triangulations_list()  # optional - topcom
[(<0,1,2>, <1,2,3>), (<0,1,3>, <0,2,3>)]
sage: p.set_engine('internal')  # optional - topcom

volume (simplex=None)

Chapter 3. Triangulations
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: $n!*(\text{volume of } simplex)$.
- Without argument: $n!*(\text{the total volume of the convex hull})$.

**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
```

The square can be triangulated into two minimal simplices, so in the “integral” normalization its volume equals two:

```
sage: p.volume()
2
```

**Note:** We return $n!*(\text{metric volume of the simplex})$ 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.

```python
class sage.geometry.triangulation.base.ConnectedTriangulationsIterator
Bases: sage.structure.sage_object.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:**

```python
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
```

You can reconstruct the triangulation from the compressed output via:

```python
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>)
```

How to use the restrictions:

```python
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)
[]
```

`next()` -> the next value, or raise `StopIteration`

**class** `sage.geometry.triangulation.base.Point`

**Bases:** `sage.structure.sage_object.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()`.
You should not create \texttt{Point} objects manually. The constructor of \texttt{PointConfiguration\_base} takes care of this for you.

**INPUT:**

- \texttt{point\_configuration} – \texttt{PointConfiguration\_base}. The point configuration to which the point belongs.
- \texttt{i} – integer. The index of the point in the point configuration.
- \texttt{projective} – the projective coordinates of the point.
- \texttt{affine} – the affine coordinates of the point.
- \texttt{reduced} – the reduced (with linearities removed) coordinates of the point.

**EXAMPLES:**

```python
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)
```

**affine()**

Return the affine coordinates of the point in the ambient space.

**OUTPUT:**

A tuple containing the coordinates.

**EXAMPLES:**

```python
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)
```

**index()**

Return the index of the point in the point configuration.

**EXAMPLES:**

```python
sage: pc = PointConfiguration([[0, 1], [0, 0], [1, 0]])
sage: p = pc.point(2); p
P(1, 0)
sage: p.index()
2
```

**point\_configuration()**

Return the point configuration to which the point belongs.

**OUTPUT:**
A `PointConfiguration`.

**EXAMPLES:**

```python
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
```

**projective()**

Return the projective coordinates of the point in the ambient space.

**OUTPUT:**

A tuple containing the coordinates.

**EXAMPLES:**

```python
c sage: pc = PointConfiguration([[10, 0, 1], [10, 0, 0], [10, 2, 3]])
c sage: p = pc.point(2); p
P(10, 2, 3)
c sage: p.affine()
(10, 2, 3)
c sage: p.projective()
(10, 2, 3, 1)
c sage: p.reduced_affine()
(2, 2)
c sage: p.reduced_projective()
(2, 2, 1)
c sage: 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:**

```python
c sage: pc = PointConfiguration([[10, 0, 1], [10, 0, 0], [10, 2, 3]])
c sage: p = pc.point(2); p
P(10, 2, 3)
c sage: p.affine()
(10, 2, 3)
c sage: p.projective()
(10, 2, 3, 1)
c sage: p.reduced_affine()
(2, 2)
c sage: p.reduced_projective()
(2, 2, 1)
c sage: 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
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: p.reduced_projective_vector()
(2, 2, 1)
```

**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
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: p.reduced_projective_vector()
(2, 2, 1)
```

**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
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: p.reduced_projective_vector()
(2, 2, 1)
```
class sage.geometry.triangulation.base.PointConfiguration_base
Bases: sage.structure.parent.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:

```python
sage: p = PointConfiguration([[0,0,0]])
sage: p.ambient_dim()
3
sage: p.dim()
0
```

base_ring()
Return the base ring, that is, the ring containing the coordinates of the points.

OUTPUT:
A ring.

EXAMPLES:

```python
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
```

dim()
Return the actual dimension of the point configuration.

See also ambient_dim()

EXAMPLES:

```python
sage: p = PointConfiguration([[0,0,0]])
sage: p.ambient_dim()
3
```
**int_to_simplex(s)**

Reverses the enumeration of possible simplices in `simplex_to_int()`.

The enumeration is compatible with [PUNTOS].

**INPUT:**
- `s` – int. An integer that uniquely specifies a simplex.

**OUTPUT:**
An ordered tuple consisting of the indices of the vertices of the simplex.

**EXAMPLES:**

```python
sage: U = matrix([ 0, 0, 0, 0, 0, 2, 4,-1, 1, 0, 0, 1, 0],
....: [ 0, 0, 0, 1, 0, 0,-1, 0, 0, 0, 0, 0, 0],
....: [ 0, 2, 0, 0, 0, 0,-1, 0, 1, 0, 1, 0, 1],
....: [ 0, 1, 1, 0, 0, 1, 0,-2, 1, 0, 0,-1, 1],
....: [ 0, 0, 0, 0, 1, 0,-1, 0, 0, 0, 0, 0, 0])

sage: pc = PointConfiguration(U.columns())

sage: pc.int_to_simplex(1678)  # (1, 3, 4, 7, 10, 13)
```

**is_affine()**

Whether the configuration is defined by affine points.

**OUTPUT:**
Boolean. If true, the homogeneous coordinates all have 1 as their last entry.

**EXAMPLES:**

```python
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)

sage: p.is_affine()  # False
```

**n_points()**

Return the number of points.

Same as `len(self)`.

**EXAMPLES:**

```python
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
```

(continues on next page)
fine, not necessarily regular.
sage: len(p)
5
sage: 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)

points()
Return a list of the points.

OUTPUT:
Returns 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)

reduced_affine_vector_space()
Return the vector space that contains the affine points.

OUTPUT:
A vector space over the fraction field of \texttt{base\_ring()}. 

\textbf{EXAMPLES:}

\begin{verbatim}
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
\end{verbatim}

\textbf{reduced\_projective\_vector\_space()}

Return the vector space that is spanned by the homogeneous coordinates.

\textbf{OUTPUT:}

A vector space over the fraction field of \texttt{base\_ring()}. 

\textbf{EXAMPLES:}

\begin{verbatim}
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
\end{verbatim}

\textbf{simplex\_to\_int (simplex)}

Returns an integer that uniquely identifies the given simplex.

See also the inverse method \texttt{int\_to\_simplex()}. 

The enumeration is compatible with [PUNTOS].

\textbf{INPUT:}

- simplex – iterable, for example a list. The elements are the vertex indices of the simplex.

\textbf{OUTPUT:}

An integer that uniquely specifies the simplex.

\textbf{EXAMPLES:}

\begin{verbatim}
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,-1, 0, 1, 0, 1, 0, 0, 0, 1],
            [ 0, 1, 1, 0, 1, 0,-2, 1, 0, 0,-1, 1, 1],
            [ 0, 0, 0, 0, 0,-1, 0, 0, 0, 0, 0, 0, 0, 0]
            ])
sage: pc = PointConfiguration(U.columns())
sage: pc.simplex_to_int([1,3,4,7,10,13])
1678
\end{verbatim}
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:

```python
sage: PointConfiguration.set_engine('internal')  # to make doctests independent of TOPCOM
```

Here is a simple example of how to triangulate a point configuration:

```python
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)
Graphics3d Object
```

See `sage.geometry.triangulation.point_configuration` for more details.

```python
class sage.geometry.triangulation.element.Triangulation(triangulation, parent, check=True)
Bases: sage.structure.element.Element
A triangulation of a PointConfiguration.
```

**Warning:** You should never create `Triangulation` objects manually. See `triangulate()` and `triangulations()` to triangulate point configurations.

```python
adjacency_graph()
Returns 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:

```python
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]],
[1,0,-1], [-1,0,0], [0,-1,0], [0,0,-1]])
sage: t = p.triangulate()
sage: t.adjacency_graph()
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')
triangulation
(<0,1,2,7>, <0,1,4,7>, <0,2,4,7>, <1,2,3,7>, <1,4,5,7>, <2,4,6,7>)
triangulation.boundary()
frozenset({(0, 1, 2), 
    (0, 1, 4), 
    (0, 2, 4), 
    (1, 2, 3), 
    (1, 3, 7), 
    (1, 4, 5), 
    (1, 5, 7), 
    (2, 3, 7), 
    (2, 4, 6), 
    (2, 6, 7), 
    (4, 5, 7), 
    (4, 6, 7)})
```

enumerate_simplices()  
Return the enumerated simplices.

OUTPUT:

A tuple of integers that uniquely specifies the triangulation.

EXAMPLES:

```sage
c = 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, 1, 0,-2, 1, 0, 0,-1, 1, 1],  
    [0, 0, 0, 0, 1, 0,-1, 0, 0, 0, 0, 0, 0, 0],  
    [0, 1, 2, 0, 0, 2, 4,-1, 1, 1, 0, 0, 1, 0],  
    [0, 1, 4, 7, 0, 2, 4, 7, 1, 2, 3, 7, 1, 4, 5],  
    [0, 1, 4, 5, 0, 2, 4, 7, 1, 2, 3, 7, 1, 5, 7],  
    [0, 1, 5, 7, 0, 2, 4, 7, 1, 2, 3, 7, 2, 3, 7],  
    [0, 4, 5, 7, 2, 3, 7, 1, 2, 3, 7, 2, 4, 6],  
    [0, 4, 6, 7, 2, 3, 7, 1, 2, 3, 7, 2, 6, 7],  
    [0, 6, 7, 2, 3, 7, 1, 2, 3, 7, 4, 5, 7],  
    [0, 6, 7, 2, 3, 7, 1, 2, 3, 7, 4, 6, 7])
])
triangulation = pc.lexicographic_triangulation() 
triangulation.enumerate_simplices() 
(1678, 1688, 1769, 1779, 1895, 1905, 2112, 2143, 2234, 2360, 2365, 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
Triangulation([1678, 1688, 1769, 1779, 1895, 1905, 2112, 2143, 2234, 2360, 2365, 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)
```
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() )
True
```

Toric diagrams (the $\mathbb{Z}_5$ 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) )
sage: 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
```
**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:**

```python
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)
```

**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:**

```python
sage: triangulation = polytopes.cube().triangulate(engine='internal')
sage: triangulation
(<0,1,2,7>, <0,1,4,7>, <0,2,4,7>, <1,2,3,7>, <1,4,5,7>, <2,4,6,7>)
sage: triangulation.boundary()
frozenset({(0, 1, 2), (0, 1, 4), (0, 2, 4), (1, 2, 3), (1, 3, 7), (1, 4, 5), (1, 5, 7), (2, 3, 7), (2, 4, 6), (2, 6, 7), (4, 5, 7), (4, 6, 7)})
sage: triangulation.interior_facets()
frozenset({(0, 1, 7), (0, 2, 7), (0, 4, 7), (1, 2, 7), (1, 4, 7), (2, 4, 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
sage: triangulation = polytopes.hypercube(2).triangulate(engine='internal')
```

```sage
sage: triangulation
(<0,1,3>, <0,2,3>)
```

```sage
sage: N = triangulation.normal_cone(); N
4-d cone in 4-d lattice
```

```sage
sage: N.rays()
(-1, 0, 0, 0),
( 1, 0, 1, 0),
(-1, 0, -1, 0),
( 1, 0, 0, -1),
(-1, 0, 0, 1),
( 1, 1, 0, 0),
(-1, -1, 0, 0)
in Ambient free module of rank 4
over the principal ideal domain Integer Ring
```

```sage
sage: N.dual().rays()
(-1, 1, 1, -1)
in Ambient free module of rank 4
over the principal ideal domain Integer Ring
```

```sage
plot(**kwds)
```

Produce a graphical representation of the triangulation.

EXAMPLES:

```sage
sage: p = PointConfiguration([[0,0],[0,1],[1,0],[1,1],[-1,-1]])
```

```sage
sage: triangulation = p.triangulate()
```

```sage
sage: triangulation
(<1,3,4>, <2,3,4>)
```

```sage
sage: triangulation.plot(axes=False)
Graphics object consisting of 12 graphics primitives
```

```sage
point_configuration()
```

Returns the point configuration underlying the triangulation.

EXAMPLES:

```sage
sage: pconfig = PointConfiguration([[0,0],[0,1],[1,0]])
```

```sage
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
sage: triangulation = pconfig.triangulate()
```

```sage
sage: triangulation
(<0,1,2>)
```

```sage
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
```

(continues on next page)
fine, not necessarily regular.
\[\text{sage: } \text{pconfig == triangulation.point_configuration() }\]
\[\text{True}\]

\textbf{simplicial\_complex()}

Return a simplicial complex from a triangulation of the point configuration.

\textbf{OUTPUT:}

A \texttt{SimplicialComplex}.

\textbf{EXAMPLES:}

\begin{verbatim}
\text{sage: } p = polytopes.cuboctahedron()
\text{sage: } sc = p.triangulate(engine='internal').simplicial_complex()
\text{sage: } sc
Simplicial complex with 12 vertices and 16 facets
\end{verbatim}

Any convex set is contractable, so its reduced homology groups vanish:

\begin{verbatim}
\text{sage: } sc.homology()
\{0: 0, 1: 0, 2: 0, 3: 0\}
\end{verbatim}

\texttt{sage.geometry.triangulation.element.triangulation\_render\_2d(triangulation, **kwds)}

Return a graphical representation of a 2-d triangulation.

\textbf{INPUT:}

- \texttt{triangulation} – a \texttt{Triangulation}.
- \texttt{**kwds} – keywords that are passed on to the graphics primitives.

\textbf{OUTPUT:}

A 2-d graphics object.

\textbf{EXAMPLES:}

\begin{verbatim}
\text{sage: } points = PointConfiguration([[0,0],[0,1],[1,0],[1,1],[-1,-1]])
\text{sage: } triang = points.triangulate()
\text{sage: } triang.plot(axes=False, aspect_ratio=1)  # indirect doctest
Graphics object consisting of 12 graphics primitives
\end{verbatim}

\texttt{sage.geometry.triangulation.element.triangulation\_render\_3d(triangulation, **kwds)}

Return a graphical representation of a 3-d triangulation.

\textbf{INPUT:}

- \texttt{triangulation} – a \texttt{Triangulation}.
- \texttt{**kwds} – keywords that are passed on to the graphics primitives.

\textbf{OUTPUT:}

A 3-d graphics object.

\textbf{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
Graphics3d Object
4.1 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:**

```python
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
```

You can also pass coefficients and a constant term to construct linear expressions:

```python
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
```

The linear expressions are a module over the base ring, so you can add them and multiply them with scalars:

```python
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
```

```python
class sage.geometry.linear_expression.LinearExpression(parent, coefficients, constant, check=True)

Bases: sage.structure.element.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: m2 == m3
True
sage: L.zero()
0*x + 0*y + 0*z + 0
sage: a = L([12, 2/3, -1], -2)
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

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

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

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:

```python
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L.<x,y,z> = LinearExpressionModule(QQ)
```

```python
sage: a = x + 2*y + 3*z + 4; a
x + 2*y + 3*z + 4
```

```python
sage: 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:

```python
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L.<x,y,z> = LinearExpressionModule(QQ)
```

```python
sage: linear = L([1, 2, 3], 4); linear
x + 2*y + 3*z + 4
```

```python
sage: linear.coefficients()
[4, 1, 2, 3]
```

`constant_term()`

Return the constant term.

OUTPUT:

The constant term of the linear expression.

EXAMPLES:

```python
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L.<x,y,z> = LinearExpressionModule(QQ)
```

```python
sage: linear.A()
(1, 2, 3)
```

```python
sage: 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:

```python
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L.<x,y,z> = LinearExpressionModule(QQ)
```

```python
4.1. Linear Expressions 467
```
**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:**

```python
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])
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:**

```python
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)]
```

---

**class** `sage.geometry.linear_expression.LinearExpressionModule` *(base_ring, names=())*

**Bases:** `sage.structure.parent.Parent`, `sage.structure.unique_representation.UniqueRepresentation`

The module of linear expressions.

This is the module of linear polynomials which is the parent for linear expressions.

**EXAMPLES:**

```python
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L = LinearExpressionModule(QQ, ('x', 'y', 'z'))
sage: 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'))
Vector space of dimension 3 over Rational Field
sage: L.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
```

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'))
Vector space of dimension 3 over Rational Field
sage: L.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'))
[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 \( (\text{base\_ring}) \)
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:
```python
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
```

gen \( (i) \)
Return the \( i \)-th generator.

INPUT:
• \( i \) – integer

OUTPUT:
A linear expression.

EXAMPLES:
```python
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
```

gens ()
Return the generators of self.

OUTPUT:
A tuple of linear expressions, one for each linear variable.

EXAMPLES:
```python
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)
```

ngens ()
Return the number of linear variables.

OUTPUT:
An integer.

EXAMPLES:
```python
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L = LinearExpressionModule(QQ, ('x', 'y', 'z'))
sage: 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()
-1/2*x - 1/95*y + 1/2*z - 12
```

## 4.2 Newton Polygons

This module implements finite Newton polygons and infinite Newton polygons having a finite number of slopes (and hence a last infinite slope).

```
sage.geometry.newton_polygon.NewtonPolygon = Parent for Newton polygons
```

```
class sage.geometry.newton_polygon.NewtonPolygon_element (polyhedron, parent)
Bases: sage.structure.element.Element
```

Class for infinite Newton polygons with last slope.

```
last_slope()
Returns 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
```

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
```

```
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()
```

4.2. Newton Polygons
reverse (degree=None)
Returns the symmetric of self

INPUT:

• degree – an 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)

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]

slopes (repetition=True)
Returns the slopes of this Newton polygon

INPUT:

• repetition – a 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]

vertices (copy=True)
Returns the list of vertices of this Newton polygon

INPUT:

• copy – a boolean (default: True)

OUTPUT:

The list of vertices of this Newton polygon (or a copy of it if copy is set to True)

EXAMPLES:
```python
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)]
```

```python
class sage.geometry.newton_polygon.ParentNewtonPolygon
    Bases: sage.structure.parent.Parent, sage.structure.unique_representation.UniqueRepresentation

Construct a Newton polygon.

**INPUT:**
- `arg` – a list/tuple/iterable of vertices or of slopes. Currently, slopes must be rational numbers.
- `sort_slopes` – boolean (default: True). Specifying 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:

```python
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)
```

We note that the convex hull of the vertices is automatically computed:

```python
sage: NewtonPolygon([(0,0), (1,1), (2,8), (3,5)])
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:

```python
sage: NewtonPolygon([(0,0), (1,Infinity), (2,8), (3,5)])
Finite Newton polygon with 2 vertices: (0, 0), (1, 1) ending by an \(\rightarrow\) infinite line of slope 3/2
```

If `last_slope` is set, the returned Newton polygon is infinite and ends with an infinite line having the specified slope:

```python
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 \(\rightarrow\) infinite line of slope 3
```

Specifying a last slope may discard some vertices:

```python
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 \(\rightarrow\) line of slope 3/2
```

4.2. Newton Polygons
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]
```

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
```

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)
```

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
\rightarrow infinite line of slope 2/3
sage: 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))
```

4.3 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: sage.structure.sage_object.SageObject, sage.structure.unique_representation.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 $\Sigma$ be an orientable surface with non-empty boundary and let $\Gamma$ be the topological realization of a graph that is embedded in $\Sigma$ in such a way that the graph is a strong deformation retract of the surface.

Let $v(\Gamma)$ be the set of vertices of $\Gamma$, 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 $\Gamma$ 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 $\sigma$. This permutation is a product of as many cycles as white vertices (that is vertices in $\Gamma$). For each vertex consider a small topological circle around it in $\Sigma$. This circle intersects each adjacent dart once. The circle has an orientation induced by the orientation on $\Sigma$ 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 $\rho$. This permutation is a product of as many 2-cycles as edges has $\Gamma$. It just tells which two darts belong to the same edge.

**Abstract definition**

Consider a graph $\Gamma$ (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 $\Gamma$ is a set of two permutations $(\sigma, \rho)$. Where $\sigma$ is formed by as many disjoint cycles as vertices had $\Gamma$. And each cycle is a cyclic ordering of the darts adjacent to a vertex. The permutation $\rho$ 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 $\sigma$ and $\rho$.

**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:

```python
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
```

Consider a graph that has 2 vertices of valency 3 (and hence 3 edges). That is represented by the following two permutations:

```python
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
```

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:

```python
sage: R1.number_boundaries()
1
sage: 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 $(15)$ when we define sigma:

```python
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); R2
Ribbon graph of genus 1 and 1 boundary components
```

This example is constructed by taking the bipartite graph of type $(3,3)$:

```python
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
```

The labeling of the darts can omit some numbers:

```python
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
```

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 = 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.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]]]

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()
(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)

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:

4.3. Ribbon Graphs 477
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:

```python
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()
(3,5,4,6)
sage: H1.rho()
(3,4)(5,6)
sage: 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:

```python
sage: s2 = PermutationGroupElement('(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,15)(16,17,18,19)')
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]]
```

**contract_edge** (*k*)

Return the ribbon graph resulting from the contraction of the *k*-th edge in **self**.

For a ribbon graph \((\sigma, \rho)\), we contract the edge corresponding to the *k*-th transposition of \(\rho\).

INPUT:

- *k* – non-negative integer; the position in \(\rho\) 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:

```python
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
```
However, this ribbon graph 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
```

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 \(\sigma\). 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 = 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)
sage: R2c = R2.contract_edge(9); 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 \(\sigma\), 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
```

(continues on next page)
We can also extrude a contractible edge from a vertex. This new edge will end at a vertex of valency 1:

```python
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)
```

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:

```python
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(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)
```

**genus()**

Return the genus of the thickening of `self`.

**OUTPUT:**

- `g` – non-negative integer representing the genus of the thickening of the ribbon graph

**EXAMPLES:**

```python
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
```
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 \( \mu \). Each row corresponds to an element of the basis and is a circle contained in the graph.

EXAMPLES:

```python
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]]]
```

```python
H = R.reduced(); H
Ribbon graph of genus 1 and 1 boundary components
sage: H.sigma()
(2,3,5,6)
```

```python
sage: H.rho()
(2,5) (3,6)
```

```python
sage: H.homology_basis()
[[[2, 5]], [[3, 6]]]
```

```python
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 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]],
```

(continues on next page)
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()
```sage
R = RibbonGraph(0, 6);
R
Ribbon graph of genus 0 and 6 boundary components
sage: R.sigma()
(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
```

\[ (1,2,3,4,5,6,7,8,9,10,11,27,25,23) (12,24,26,28,13,14,15,16,17,18,19,20,21,22) \]

\[ \text{sage}: \ R.\rho() \]
\[ (1,12) (2,13) (3,14) (4,15) (5,16) (6,17) (7,18) (8,19) (9,20) (10,21) (11,22) (23, 24) (25,26) (27,28) \]

\[ \text{sage}: \ G = R.\text{reduced}(); \ G \]
Ribbon graph of genus 5 and 4 boundary components

\[ \text{sage}: \ G.\sigma() \]
\[ (2,3,4,5,6,7,8,9,10,11,27,25,23,24,26,28,13,14,15,16,17,18,19,20,21,22) \]

\[ \text{sage}: \ G.\rho() \]
\[ (2,13) (3,14) (4,15) (5,16) (6,17) (7,18) (8,19) (9,20) (10,21) (11,22) (23,24) (25, 26) (27,28) \]

\[ \text{sage}: \ G.\text{genus}() == R.\text{genus}() \text{ and } G.\text{number_boundaries}() == R.\text{number_boundaries}() \]
True

\[ \text{mu()} \]
Return the rank of the first homology group of the thickening of the ribbon graph.

\[ \text{EXAMPLES:} \]

\[ \text{sage}: \ s1 = \text{PermutationGroupElement}('(1,3,5)(2,4,6)') \]
\[ \text{sage}: \ r1 = \text{PermutationGroupElement}('(1,2)(3,4)(5,6)') \]
\[ \text{sage}: \ R1 = \text{RibbonGraph}(s1, r1); R1 \]
Ribbon graph of genus 1 and 1 boundary components

\[ \text{sage}: \ R1.\text{mu}() \]
2

\[ \text{normalize()} \]
Return an equivalent graph such that the enumeration of its darts exhausts all numbers from 1 to the number of darts.

\[ \text{OUTPUT:} \]

- a ribbon graph equivalent to \text{sage} such that the enumeration of its darts exhausts all numbers from 1 to the number of darts.

\[ \text{EXAMPLES:} \]

\[ \text{sage}: \ s0 = \text{PermutationGroupElement}('(1,22,3,4,5,6,7,15)(8,16,9,10,11,12,13,14) \to') \]
\[ \text{sage}: \ r0 = \text{PermutationGroupElement}('(1,8)(22,9)(3,10)(4,11)(5,12)(6,13)(7, \to14)(15,16)') \]
The first example is the ribbon graph corresponding to the torus with one hole:

```python
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)
```

This example is constructed by taking the bipartite graph of type $(3,3)$:

```python
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)
sage: R2.number_boundaries()
3
```

reduced()
Return a ribbon graph with 1 vertex and $\mu$ edges (where $\mu$ is the first betti number of the graph).

OUTPUT:

- a ribbon graph whose $\sigma$ permutation has only 1 non-singleton cycle and whose $\rho$ permutation is a product of $\mu$ disjoint 2-cycles

EXAMPLES:

```python
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()
```

(continues on next page)
(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 = 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 = RibbonGraph(s3, r3); R3
Ribbon graph of genus 3 and 1 boundary components
sage: G3 = R3.reduced(); G3
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)

rho()
Return the permutation \( \rho \) 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.rho()
(1, 2) (3, 4) (5, 6) (8, 15)

sigma()
Return the permutation \( \sigma \) 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)

.. function:: 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:**

- $p$ – a positive integer
- $q$ – a positive integer

**EXAMPLES:**

```sage
text
```

**sage.geometry.ribbon_graph.make_ribbon** $(g, r)$

Return a ribbon graph whose thickening has genus $g$ and $r$ boundary components.

**INPUT:**

- $g$ – non-negative 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
text
```
4.4 Pseudolines

This module gathers everything that has to do with pseudolines, and for a start a \texttt{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 \textit{Wiring diagrams} via the \texttt{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 :-)

\textbf{Definition}

A \textit{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.

\textbf{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.

4.4.1 Encodings

\textbf{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 \( \{0, ..., n-1\} \setminus \{i\} \) representing the ordering in which the \( i \) th pseudoline meets the other ones.

```sage
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
```

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 \((8, 2)\) if \( l_2 \) 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
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
```

Note that this ordering is not necessarily unique.

Felsner’s Matrix

Felsner 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

\[
\begin{align*}
3 & 2 & 1 & 0 & 0 & 0 \\
3 & 2 & 0 & 0 & 0 & 1 \\
3 & 1 & 0 & \Rightarrow & 0 & 1 & 1 \\
2 & 1 & 0 & 1 & 1 & 1
\end{align*}
\]

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 \( \begin{bmatrix} 0 \\ 1 \end{bmatrix} \) 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
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
```
4.4.2 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 $b$:

```python
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
[(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:

```python
sage: permutations = [[0..i-1]+[i+1..n-1] for i in range(n)]
sage: a = lambda x : l[x][0]
sage: b = lambda x : l[x][1]
sage: 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:

```python
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])
```

Author

Nathann Cohen

4.4.3 Methods

**class sage.geometry.pseudolines.PseudolineArrangement**($seq$, $encoding='auto'$)

Creates 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 $0..(n-1)$.
- For more information on the different encodings, see the **pseudolines module**’s documentation.
felsner_matrix()
Returns a Felsner matrix describing the arrangement.

See the pseudolines module's documentation for more information on this encoding.

EXAMPLES:

```python
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]]
```

permutations()
Returns the arrangements as $n$ permutations of size $n - 1$.

See the pseudolines module's documentation for more information on this encoding.

EXAMPLES:

```python
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]]
```

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:

```python
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])
```

transpositions()
Returns the arrangement as $\binom{n}{2}$ transpositions.

See the pseudolines module's documentation for more information on this encoding.

EXAMPLES:

```python
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)]
```
4.5 Voronoi diagram

This module provides the class `VoronoiDiagram` for computing the Voronoi diagram of a finite list of points in \( \mathbb{R}^d \).

```python
class sage.geometry.voronoi_diagram.VoronoiDiagram(points)
    Bases: sage.structure.sage_object.SageObject

    Base class for the Voronoi diagram.

    Compute the Voronoi diagram of a list of points.

    INPUT:
        • points – a 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 \( \mathbb{R}^3 \):
    
    sage: V = VoronoiDiagram([[1, 3, 2], [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.

    Get the Voronoi diagram of a regular pentagon in \( \mathbb{A}^2 \). All cells meet at the origin:
    
    sage: DV = VoronoiDiagram([[AA(c) for c in v] 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())
    True
    sage: any(P.interior_contains([0, 0]) for P in DV.regions().values())
    False

    If the vertices are not converted to \( \mathbb{A} \) before, the method throws an error:
    
    sage: polytopes.dodecahedron().vertices_list()[0][0].parent()
    Number Field in sqrt5 with defining polynomial x^2 - 5
    sage: VoronoiDiagram(polytopes.dodecahedron().vertices_list())
    Traceback (most recent call last):
    ... Not Implemented Error: 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()
sage: V = VoronoiDiagram([[1, 2, 3, 4, 5, 6]]); V.ambient_dim()
```

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
```

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: 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)
``` (continues on next page)
points()  
Return the input points (as a PointConfiguration).

EXAMPLES:

```python
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.
```

regions()  
Return the Voronoi regions of the Voronoi diagram as a dictionary of polyhedra.

EXAMPLES:

```python
sage: V = VoronoiDiagram([[1, 3, .3], [2, -2, 1], [-1, 2, -.1]])
sage: P = 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(1.074999999999999), RDF(0.1494444444), RDF(9.0138888890000004)), ....: P[1]: Polyhedron(base_ring=RDF, lines=[(-RDF(0.375), RDF(1.074999999999999), RDF(0.1494444444), RDF(9.0138888890000004))], ....: rays=[(RDF(9), -RDF(1), -RDF(20)), (RDF(4.5), RDF(1), -RDF(25))], ....: vertices=[(-RDF(1.074999999999999), RDF(0.1494444444), RDF(9.0138888890000004))]), ....: P[2]: Polyhedron(base_ring=RDF, lines=[(-RDF(0.375), RDF(1.074999999999999), RDF(0.1494444444), RDF(9.0138888890000004))], ....: rays=[(RDF(4.5), RDF(1), -RDF(25)), (RDF(2.25), -RDF(1), RDF(2.5))], ....: vertices=[(-RDF(1.074999999999999), RDF(0.1494444444), RDF(9.0138888890000004))])
True
```
5.1 Find isomorphisms between fans.

**exception** `sage.geometry.fan_isomorphism.FanNotIsomorphicError`

Bases: `exceptions.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:**

```python
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
```

`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:

```python
sage: fan = toric_varieties.P2().fan()
sage: from sage.geometry.fan_isomorphism import fan_2d_echelon_form
sage: fan_2d_echelon_form(fan)
[ 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:

```python
sage: fan = toric_varieties.P2().fan()
sage: from sage.geometry.fan_isomorphism import fan_2d_echelon_forms
sage: fan_2d_echelon_forms(fan)
\{\begin{bmatrix} 1 & 0 & -1 \\ 0 & 1 & -1 \end{bmatrix}\}\}
sage: fan = toric_varieties.dP7().fan()
sage: sorted(fan_2d_echelon_forms(fan))
\{\begin{bmatrix} 1 & 0 & -1 & -1 & 0 \\ 0 & 1 & 0 & -1 & -1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & -1 & -1 & 0 \\ 0 & 1 & 1 & 0 & -1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & -1 & 0 & 1 \\ 0 & 1 & 1 & 0 & -1 \end{bmatrix}, \begin{bmatrix} 1 & 0 & -1 & 0 & 1 \\ 0 & 1 & 0 & -1 & -1 \end{bmatrix}\}
```

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:

```python
sage: fan1 = toric_varieties.P2().fan()
sage: fan2 = toric_varieties.dP8().fan()
sage: from sage.geometry.fan_isomorphism import fan_isomorphic_necessary_conditions
sage: fan_isomorphic_necessary_conditions(fan1, fan2)
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()
sage: from sage.geometry.fan_isomorphism import fan_isomorphism_generator
sage: sorted(fan_isomorphism_generator(fan, fan))
[([-1 -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: next(fan_isomorphism_generator(fan1, fan2))
[18 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: tuple(fan_isomorphism_generator(fan0, fan0))
(([1 0] [0 1]
  [0 1] [1 0])
sage: tuple(fan_isomorphism_generator(fan1, fan1))
(([-3 -20 28]
  [ 0 1] [ -1 -4 7]
  [0 0 1], [ -1 -5 8])
sage: tuple(fan_isomorphism_generator(fan1, fan2))
((18 1 -5]
 [ 4 0 -1] [ 1 -1 2]
 [ 5 0 -1], [ 2 -1 2]
```

(continues on next page)
sage: tuple(fan_isomorphism_generator(fan2, fan1))
([ 0 -1  1] [ 0 -1  1]
[ 1 -7  2] [ 2 -2 -5]
[ 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:

```python
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)
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: f = find_isomorphism(fan1, toric_varieties.P2().fan())
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),(-1,3,-1),(0,2,-1),(1,-1,1)])
sage: fan1.is_isomorphic(fan2)
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 join. For example, the face lattice of a polyhedron.

```python
sage.geometry.hasse_diagram.lattice_from_incidences(atom_to_coatoms,
coatom_to_atoms,
face_constructor=None,
required_atoms=None,
**kwds)
```

Compute an atomic and coatomic lattice from the incidence between atoms and coatoms.

**INPUT:**

- `atom_to_coatoms` – list, `atom_to_coatoms[i]` should list all coatoms over the i-th atom;
- `coatom_to_atoms` – list, `coatom_to_atoms[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:**

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)]
```

and we can compute the lattice as

```
sage: L = sage.geometry.cone.lattice_from_incidences(
....: atom_to_coatoms, coatom_to_atoms)

sage: for level in L.level_sets(): print(level)

[[((), (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
```

(continues on next page)
generic: (-1.0, 3.7) x + 1.0 >= 0
generic: (1.0, -1.35) x + 1.35 >= 0

```
sage: line = Polyhedron(eqs=[(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
```

**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 0-th 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(eqs=[(2,3,7)])
sage: ieq = InequalityCollection(line, [0,1], [0]*2, [1]*2 ); ieq
The collection of inequalities
integer: (3, 7) x + 2 >= 0
integer: (-3, -7) x + -2 >= 0
sage: ieq.prepare_next_to_inner_loop([3,4])
sage: ieq.prepare_inner_loop([3,4])
sage: ieq.are_satisfied(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

$$c = Ax - A_0 x_0 + b + \sum_{i=1} A_i x_i$$

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)
```

(continues on next page)
Cached inner loop: 3 \times x_0 + 42 \geq 0  
Cached next-to-inner loop: 3 \times x_0 + 7 \times x_1 + 35 \geq 0

\textbf{prepare\_next\_to\_inner\_loop}(p)

Peel off the next-to-inner loop.

In the next-to-inner loop of \texttt{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 \(Ax - A_0x_0 + b = A_1 x_1 + c \geq 0\) in the next-to-inner loop.

\textbf{INPUT:}

\begin{itemize}
  \item \(p\) – the point coordinates. Only \(p[2:]\) coordinates are potentially used by this method.
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}
from sage.geometry.integral_points import InequalityCollection, print_cache

P = Polyhedron(ieqs=[(2,3,7,11)])
ieq = InequalityCollection(P, [0,1,2], [0]*3,[1]*3); ieq

The collection of inequalities
integer: (3, 7, 11) x + 2 \geq 0

ieq.prepare_next_to_inner_loop([2,1,3])
ieq.prepare_inner_loop([2,1,3])
print_cache(ieq)
Cached inner loop: 3 \times x_0 + 42 \geq 0
Cached next-to-inner loop: 3 \times x_0 + 7 \times x_1 + 35 \geq 0
\end{verbatim}

\textbf{satisfied\_as\_equalities}(inner\_loop\_variable)

Return the inequalities (by their index) that are satisfied as equalities.

\textbf{INPUT:}

\begin{itemize}
  \item inner\_loop\_variable – Integer. the 0-th coordinate of the lattice point.
\end{itemize}

\textbf{OUTPUT:}

A set of integers in ascending order. Each integer is the index of a \(H\)-representation object of the polyhedron (either an inequality or an equation).

\textbf{EXAMPLES:}

\begin{verbatim}
from sage.geometry.integral_points import InequalityCollection

quadrant = Polyhedron(rays=[(1,0), (0,1)])
ieqs = InequalityCollection(quadrant, [0,1], [-1]*2, [1]*2 )

ieqs.prepare_next_to_inner_loop([-1,0])
ieqs.prepare_inner_loop([-1,0])
ieqs.satisfied_as_equalities(-1)
frozenset({1})
ieqs.satisfied_as_equalities(0)
frozenset({0, 1})
ieqs.satisfied_as_equalities(1)
frozenset({1})
\end{verbatim}

\textbf{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 \texttt{Inequality\_int} to swap to the beginning of the list of integral inequalities.

EXAMPLES:

```python
sage: from sage.geometry.integral_points import InequalityCollection
generic: (2*pi, sqrt(3), 7/2) x + -5.50000000000000 >= 0
```

```python
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:

```python
sage: from sage.geometry.integral_points import Inequality\_generic
sage: Inequality\_generic([2*pi, sqrt(3), 7/2], -5.5)
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
- \texttt{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 \texttt{OverflowError} is raised if a machine integer is not long enough to hold the results. A \texttt{ValueError} is raised if some of the input is not integral.

EXAMPLES:

```python
sage: from sage.geometry.integral_points import Inequality\_generic
generic: (2*pi, sqrt(3), 7/2) x + -5.50000000000000 >= 0
```
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.
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: ...

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))

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 \texttt{semigroup\_generators()} for a description of the algorithm.

**INPUT:**

- \texttt{spanning\_points} – a non-empty list of linearly independent rays (\texttt{Z}-vectors or \texttt{toric\ lattice} elements), not necessarily primitive lattice points.

**OUTPUT:**

The tuple of all lattice points in the half-open parallelotope spanned by the rays \( r_i \),

\[
\text{par}\{\{r_i\}\} = \sum_{0 \leq a_i < 1} a_i r_i
\]

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 facets are omitted:

```python
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), (1, 0), (0, 1), (1, 1))
```

The rays can also be toric lattice points:

```python
sage: rays = list(map(ToricLattice(2), [(2,0), (0,2)]))
(sage: parallelotope_points(rays, ToricLattice(2))
(N(0, 0), N(1, 0), N(0, 1), N(1, 1))
```

A non-smooth cone:

```python
sage: c = Cone([ (1,0), (1,2) ]) s sage: parallelotope_points(c.rays(), c.lattice())
(N(0, 0), N(1, 1))
```

A \texttt{ValueError} is raised if the \texttt{spanning\_points} are not linearly independent:

```python
sage: rays = list(map(ToricLattice(2), [(1,1)]*2))
(sage: parallelotope_points(rays, ToricLattice(2))
Traceback (most recent call last):
... \texttt{ValueError}: The spanning points are not linearly independent!
```

\texttt{sage.geometry.integral_points.print\_cache(inequality\_collection)}

Print the cached values in \texttt{Inequality\_int} (for debugging/doctesting only).

**EXAMPLES:**

```python
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
```

### 5.3. Cython helper methods to compute integral points in polyhedra.
Compute the Smith normal form of the ray matrix for `parallelotope_points()`.

**INPUT:**

- \( R \) – \( \mathbb{Z} \)-matrix whose columns are the rays spanning the parallelotope.

**OUTPUT:**

A tuple containing \( e, d, \) and \( VDinv \).

**EXAMPLES:**

```python
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], [1]])
```

Return the integral points in the lattice bounding box that are also contained in the given polyhedron.

**INPUT:**

- `box_min` – A list of integers. The minimal value for each coordinate of the rectangular bounding box.
- `box_max` – A 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 \((\text{point}, \ Hrep)\) 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):

\[
Ax \leq b \iff a_1x_1 \leq b - \sum_{i=2}^{d} a_ix_i
\]
so we only have to evaluate $a_1 \cdot x_1$ 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:**

```python
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))
```

```python
sage: from sage.geometry.integral_points import rectangular_box_points
sage: rectangular_box_points([0,0,0],[1,2,3], count_only=True)
24
```

```python
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, 1, 0, 0), (0, 1, 0, 1), (1, 0, 0, 0))
sage: d = 3
sage: dilated_cell24 = d*cell24
sage: len( rectangular_box_points([-d]*4, [d]*4, dilated_cell24) )
305
```

```python
sage: d = 6
sage: dilated_cell24 = d*cell24
sage: len( rectangular_box_points([-d]*4, [d]*4, dilated_cell24) )
3625
```

```python
sage: rectangular_box_points([-d]*4, [d]*4, dilated_cell24, count_only=True)
3625
```

```python
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, 1, 1), (1, 1, 1, 1), (1, 1, 1, 1), (1, 1, 1, 1), (1, 1, 1, 1))
sage: all(polytope.contains(p) for p in pts)
True
```

```python
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
```

Long ints and non-integral polyhedra are explicitly allowed:
Using a PPL polyhedron:

```python
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))
```

Optionally, return the information about the saturated inequalities as well:

```python
sage: cube = polytopes.cube()
sage: cube.Hrepresentation(0)
An inequality (0, 0, -1) x + 1 >= 0
sage: cube.Hrepresentation(1)
An inequality (0, -1, 0) x + 1 >= 0
sage: cube.Hrepresentation(2)
An inequality (-1, 0, 0) x + 1 >= 0
sage: rectangular_box_points([0]*3, [1]*3, cube, return_saturated=True)
(((0, 0, 0), frozenset()), ((0, 0, 1), frozenset(0)), ((0, 1, 0), frozenset(1)), ((0, 1, 1), frozenset(0, 1)), ((1, 0, 0), frozenset(2)), ((1, 0, 1), frozenset(0, 2)), ((1, 1, 0), frozenset(1, 2)), ((1, 1, 1), frozenset(0, 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 \( \mathbb{Z} \)-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))

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),)

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)
[[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]])

(continues on next page)
sage: less_generators(m)
[1 0 0]
[0 z -1]
[0 y 1]
CHAPTER SIX

INDICES AND TABLES

- Index
- Module Index
- Search Page
BIBLIOGRAPHY


sage.geometry.cone, 203
sage.geometry.fan, 257
sage.geometry.fan_isomorphism, 495
sage.geometry.fan_morphism, 286
sage.geometry.hasse_diagram, 499
sage.geometry.hyperplane_arrangement.affine_subspace, 46
sage.geometry.hyperplane_arrangement.arrangement, 3
sage.geometry.hyperplane_arrangement.check_freeness, 509
sage.geometry.hyperplane_arrangement.hyperplane, 38
sage.geometry.hyperplane_arrangement.library, 34
sage.geometry.hyperplane_arrangement.plot, 48
sage.geometry.integral_points, 500
sage.geometry.lattice_polytope, 117
sage.geometry.linear_expression, 465
sage.geometry.newton_polygon, 471
sage.geometry.point_collection, 300
sage.geometry.polyhedron.backend_cdd, 407
sage.geometry.polyhedron.backend_field, 408
sage.geometry.polyhedron.backend_normaliz, 408
sage.geometry.polyhedron.backend_polymake, 413
sage.geometry.polyhedron.backend_ppl, 416
sage.geometry.polyhedron.base, 331
sage.geometry.polyhedron.base_QQ, 401
sage.geometry.polyhedron.base_RDF, 407
sage.geometry.polyhedron.base_ZZ, 402
sage.geometry.polyhedron.cdd_file_format, 116
sage.geometry.polyhedron.constructor, 78
sage.geometry.polyhedron.double_description, 418
sage.geometry.polyhedron.double_description_inhomogeneous, 425
sage.geometry.polyhedron.face, 111
sage.geometry.polyhedron.lattice_euclidean_group_element, 167
sage.geometry.polyhedron.library, 53
sage.geometry.polyhedron.palp_database, 168
sage.geometry.polyhedron.parent, 85
sage.geometry.polyhedron.plot, 102
sage.geometry.polyhedron.ppl_lattice_polygon, 170
sage.geometry.polyhedron.ppl_lattice_polytope, 174
sage.geometry.polyhedron.representation, 90
sage.geometry.pseudolines, 487
sage.geometry.ribbon_graph, 474
sage.geometry.toric_lattice, 188
sage.geometry.toric_plotter, 308
sage.geometry.triangulation.base, 449
sage.geometry.triangulation.element, 458
sage.geometry.triangulation.point_configuration, 431
sage.geometry.voronoi_diagram, 491

r
sage.rings.polynomial.groebner_fan, 317
INDEX

A

A() (sage.geometry.linear_expression.LinearExpression method), 466
A() (sage.geometry.polyhedron.double_description.Problem method), 422
A() (sage.geometry.polyhedron.representation.Hrepresentation method), 91
A_matrix() (sage.geometry.polyhedron.double_description.Problem method), 422
add_hyperplane() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 8
add_inequality() (sage.geometry.polyhedron.double_description.StandardDoubleDescriptionPair method), 424
adjacency_graph() (sage.geometry.triangulation.element.Triangulation method), 458
adjacency_matrix() (sage.geometry.polyhedron.base.Polyhedron_base method), 334
adjacent() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 212
adjacent() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 120
adjacent() (sage.geometry.polyhedron.representation.Hrepresentation method), 91
adjacent() (sage.geometry.polyhedron.representation.Vrepresentation method), 100
adjust_options() (sage.geometry.toric_plotter.ToricPlotter method), 310
affine() (sage.geometry.triangulation.base.Point method), 451
affine_hull() (sage.geometry.polyhedron.base.Polyhedron_base method), 336
affine_lattice_polytope() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope_PPL_class method), 176
affine_space() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope_PPL_class method), 176
affine_transform() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 120
AffineSubspace (class in sage.geometry.hyperplane_arrangement.affine_subspace), 46
all_cached_data() (in module sage.geometry.lattice_polytope), 159
all_facet_equations() (in module sage.geometry.lattice_polytope), 159
all_nef_partitions() (in module sage.geometry.lattice_polytope), 160
all_points() (in module sage.geometry.lattice_polytope), 160
all_polars() (in module sage.geometry.lattice_polytope), 161
ambient() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 213
ambient() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 122
ambient_dim() (sage.geometry.polyhedron.base.Polyhedron_base method), 340
ambient_dim() (sage.geometry.polyhedron.face.PolyhedronFace method), 113
ambient_dim() (sage.geometry.polyhedron.parent.Polyhedra_base method), 87
ambient_dim() (sage.geometry.triangulation.base.PointConfiguration_base method), 454
ambient_dim() (sage.geometry.voronoi_diagram.VoronoiDiagram method), 492
ambient_dim() (sage.rings.polynomial.groebner_fan.PolyhedralCone method), 325
ambient_dim() (sage.rings.polynomial.groebner_fan.PolyhedralFan method), 326
ambient_facet_indices() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 122
ambient_Hrepresentation() (sage.geometry.polyhedron.face.PolyhedronFace method), 112
ambient_module() (sage.geometry.linear_expression.LinearExpressionModule method), 469
<table>
<thead>
<tr>
<th>Method/Class</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bounding_box() (sage.geometry.polyhedron.base.Polyhedron_base method)</td>
<td>343</td>
</tr>
<tr>
<td>bounding_box() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope_PPL_class method)</td>
<td>178</td>
</tr>
<tr>
<td>braid() (sage.geometry.hyperplane_arrangement.library.HyperplaneArrangementLibrary method)</td>
<td>36</td>
</tr>
<tr>
<td>buchberger() (sage.rings.polynomial.groebner_fan.GroebnerFan method)</td>
<td>318</td>
</tr>
<tr>
<td>buckyball() (sage.geometry.polyhedron.library.Polytopes method)</td>
<td>56</td>
</tr>
<tr>
<td>cardinality() (sage.geometry.point_collection.PointCollection method)</td>
<td>302</td>
</tr>
<tr>
<td>cartesian_product() (sage.geometry.cone.ConvexRationalPolyhedralCone method)</td>
<td>214</td>
</tr>
<tr>
<td>cartesian_product() (sage.geometry.cone.IntegralRayCollection method)</td>
<td>247</td>
</tr>
<tr>
<td>cartesian_product() (sage.geometry.fan.RationalPolyhedralFan method)</td>
<td>268</td>
</tr>
<tr>
<td>cartesian_product() (sage.geometry.point_collection.PointCollection method)</td>
<td>302</td>
</tr>
<tr>
<td>Catalan() (sage.geometry.hyperplane_arrangement.library.HyperplaneArrangementLibrary method)</td>
<td>34</td>
</tr>
<tr>
<td>cdd_Hrepresentation() (in module sage.geometry.polyhedron.cdd_file_format)</td>
<td>116</td>
</tr>
<tr>
<td>cdd_Hrepresentation() (sage.geometry.polyhedron.base.Polyhedron_base method)</td>
<td>344</td>
</tr>
<tr>
<td>cdd_Vrepresentation() (in module sage.geometry.polyhedron.cdd_file_format)</td>
<td>116</td>
</tr>
<tr>
<td>cdd_Vrepresentation() (sage.geometry.polyhedron.base.Polyhedron_base method)</td>
<td>344</td>
</tr>
<tr>
<td>center() (sage.geometry.polyhedron.base.Polyhedron_base method)</td>
<td>345</td>
</tr>
<tr>
<td>change_ring() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method)</td>
<td>9</td>
</tr>
<tr>
<td>change_ring() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangements method)</td>
<td>32</td>
</tr>
<tr>
<td>change_ring() (sage.geometry.hyperplane_arrangement.hyperplane.AmbientVectorSpace method)</td>
<td>40</td>
</tr>
<tr>
<td>change_ring() (sage.geometry.linear_expression.LinearExpression method)</td>
<td>466</td>
</tr>
<tr>
<td>change_ring() (sage.geometry.linear_expression.LinearExpressionModule method)</td>
<td>469</td>
</tr>
<tr>
<td>change_ring() (sage.geometry.polyhedron.base.Polyhedron_base method)</td>
<td>345</td>
</tr>
<tr>
<td>change_ring() (sage.geometry.polyhedron.parent.Polyhedra_base method)</td>
<td>88</td>
</tr>
<tr>
<td>characteristic() (sage.rings.polynomial.groebner_fan.GroebnerFan method)</td>
<td>318</td>
</tr>
<tr>
<td>characteristic_polynomial() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method)</td>
<td>10</td>
</tr>
<tr>
<td>circuits() (sage.geometry.triangulation.point_configuration.PointConfiguration method)</td>
<td>436</td>
</tr>
<tr>
<td>circuits_support() (sage.geometry.triangulation.point_configuration.PointConfiguration method)</td>
<td>436</td>
</tr>
<tr>
<td>classify_cone_2d() (cone) (in module sage.geometry.cone)</td>
<td>253</td>
</tr>
<tr>
<td>closed_faces() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method)</td>
<td>10</td>
</tr>
<tr>
<td>codim() (sage.geometry.cone.IntegralRayCollection method)</td>
<td>248</td>
</tr>
<tr>
<td>codomain_dim() (sage.geometry.polyhedron.lattice.euclidean_group_element.LatticeEuclideanGroupElement method)</td>
<td>167</td>
</tr>
<tr>
<td>codomain_fan() (sage.geometry.fan_morphism.FanMorphism method)</td>
<td>289</td>
</tr>
<tr>
<td>coefficients() (sage.geometry.linear_expression.LinearExpression method)</td>
<td>467</td>
</tr>
<tr>
<td>color_list() (in module sage.geometry.toric_plotter)</td>
<td>313</td>
</tr>
<tr>
<td>column_matrix() (sage.geometry.point_collection.PointCollection method)</td>
<td>303</td>
</tr>
<tr>
<td>combinatorial_automorphism_group() (sage.geometry.polyhedron.base.Polyhedron_base method)</td>
<td>346</td>
</tr>
<tr>
<td>common_refinement() (sage.geometry.fan.RationalPolyhedralFan method)</td>
<td>268</td>
</tr>
<tr>
<td>complex() (sage.geometry.fan.RationalPolyhedralFan method)</td>
<td>269</td>
</tr>
<tr>
<td>Cone() (sage.geometry.cone)</td>
<td>206</td>
</tr>
<tr>
<td>cone() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method)</td>
<td>13</td>
</tr>
<tr>
<td>cone() (sage.geometry.polyhedron.double_description.DoubleDescriptionPair method)</td>
<td>419</td>
</tr>
<tr>
<td>cone() (sage.rings.polynomial.groebner_fan.InitialForm method)</td>
<td>324</td>
</tr>
<tr>
<td>cone_containing() (cone) (sage.geometry.fan.RationalPolyhedralFan method)</td>
<td>270</td>
</tr>
<tr>
<td>cone_lattice() (sage.geometry.fan.RationalPolyhedralFan method)</td>
<td>271</td>
</tr>
<tr>
<td>Cone_of_fan (class in sage.geometry.fan)</td>
<td>261</td>
</tr>
<tr>
<td>cones() (sage.geometry.fan.RationalPolyhedralFan method)</td>
<td>273</td>
</tr>
</tbody>
</table>
cones() (sage.rings.polynomial.groebnerFan.PolyhedralFan method), 326
ConnectedTriangulationsIterator (class in sage.geometry.triangulation.base), 449
constant_term() (sage.geometry.linear_expression.LinearExpression method), 467
construct_free_chain() (in module sage.geometry.hyperplane_arrangement.check_freeness), 509
construction() (sage.geometry.toric_lattice.ToricLattice_generic method), 193
contained_simplex() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 437
contains() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 214
contains() (sage.geometry.rationalPolyhedralFan.RationalPolyhedralFan method), 273
contains() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 125
contains() (sage.geometry.polyhedron.base.Polyhedron_base method), 347
contains() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope_PPL_class method), 178
contains() (sage.geometry.polyhedron.representation.Equation method), 90
contains() (sage.geometry.polyhedron.representation.Inequality method), 93
contains_origin() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope_PPL_class method), 178
contract_edge() (sage.geometry.ribbon_graph.RibbonGraph method), 478
convex_hull() (in module sage.geometry.lattice_polytope), 161
convex_hull() (sage.geometry.polyhedron.base.Polyhedron_base method), 348
convex_hull() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 437
ConvexRationalPolyhedralCone (class in sage.geometry.cone), 208
del_index_of() (sage.geometry.polyhedron.plot.Projection method), 102
del_indices_of() (sage.geometry.polyhedron.plot.Projection method), 102
delinate() (sage.geometry.hyperplane_arrangement.library.HyperplaneArrangementLibrary method), 37
delinate_vector() (sage.geometry.toric_lattice.ToricLattice_quotient method), 198
delization() (sage.geometry.polyhedron.plot.Projection method), 103
delization() (sage.geometry.polyhedron.representation.PolyhedronRepresentation method), 96
delitate_key() (sage.geometry.toric_lattice.ToricLatticeFactory method), 191
delitate_object() (sage.geometry.toric_lattice.ToricLatticeFactory method), 191
cross_polytope() (in module sage.geometry.lattice_polytope), 161
cross_polytope() (sage.geometry.polyhedron.library.Polytopes method), 57
cross_positive_operators_gens() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 215
cube() (sage.geometry.polyhedron.library.Polytopes method), 57
cuboctahedron() (sage.geometry.polyhedron.library.Polytopes method), 57
cyclic_polytope() (sage.geometry.polyhedron.library.Polytopes method), 58
cyclic_sort_vertices_2d() (in module sage.geometry.polyhedron.plot), 109

defining_polynomial() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 14
deletion() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 14
Delta() (sage.geometry.lattice_polytope.NefPartition method), 151
Delta_polar() (sage.geometry.lattice_polytope.NefPartition method), 151
Deltas() (sage.geometry.lattice_polytope.NefPartition method), 152
dense_coefficient_list() (sage.geometry.linear_expression.LinearExpression method), 467
derivation_module_basis() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 15
derivation_module_free_chain() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 15
dilation() (sage.geometry.polyhedron.base.Polyhedron_base method), 349
dim() (sage.geometry.cone.IntegralRayCollection method), 249
dim() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 125
Element (sage.geometry.hyperplane_arrangement.hyperplane.AmbientVectorSpace attribute), 40
Element (sage.geometry.linear_expression.LinearExpressionModule attribute), 468
Element (sage.geometry.newton_polygon.ParentNewtonPolygon attribute), 474
Element (sage.geometry.polyhedron.parent.Polyhedra_field attribute), 89
Element (sage.geometry.polyhedron.parent.Polyhedra_polymake attribute), 89
Element (sage.geometry.polyhedron.parent.Polyhedra_QQ_cdd attribute), 85
Element (sage.geometry.polyhedron.parent.Polyhedra_QQ_normaliz attribute), 86
Element (sage.geometry.polyhedron.parent.Polyhedra_QQ_ppl attribute), 86
Element (sage.geometry.polyhedron.parent.Polyhedra_RDF_cdd attribute), 86
Element (sage.geometry.polyhedron.parent.Polyhedra_RDF_ppl attribute), 86
Element (sage.geometry.polyhedron.parent.Polyhedra_ZZ_normaliz attribute), 86
Element (sage.geometry.polyhedron.parent.Polyhedra_ZZ_ppl attribute), 86
Element (sage.geometry.toric_lattice.ToricLattice_ambient attribute), 192
Element (sage.geometry.toric_lattice.ToricLattice_generic attribute), 193
Element (sage.geometry.toric_lattice.ToricLattice_quotient attribute), 197
Element (sage.geometry.triangulation.point_configuration.PointConfiguration attribute), 438
embedding() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope_PPL_class method), 179
empty() (sage.geometry.polyhedron.parent.Polyhedra_base method), 88
enumerate_simplices() (sage.geometry.triangulation.element.Triangulation method), 459
Equation (class in sage.geometry.polyhedron.representation), 90
equation_generator() (sage.geometry.polyhedron.base.Polyhedron_base method), 350
equations() (sage.geometry.polyhedron.base.Polyhedron_base method), 350
equations_list() (sage.geometry.polyhedron.base.Polyhedron_base method), 351
essentialization() (sage.geometry.polyhedron.representation.arrangement.HyperplaneArrangementElement method), 17
eval() (sage.geometry.polyhedron.representation.Hrepresentation method), 91
evaluate() (sage.geometry.linear_expression.LinearExpression method), 468
evaluated_on() (sage.geometry.polyhedron.representation.Line method), 95
evaluated_on() (sage.geometry.polyhedron.representation.Ray method), 97
evaluated_on() (sage.geometry.polyhedron.representation.Vertex method), 98
exclude_points() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 438
extrude_edge() (sage.geometry.ribbon_graph.RibbonGraph method), 479

F

f_vector() (sage.geometry.polyhedron.base.Polyhedron_base method), 351
f_vector() (sage.rings.polynomial.groebner_fan.PolyhedralFan method), 327
face_codimension() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 439
face_fan() (sage.geometry.polyhedron.base.Polyhedron_base method), 351
face_interior() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 439
face_lattice() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 220
face_lattice() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 127
face_lattice() (sage.geometry.polyhedron.base.Polyhedron_base method), 352
face_product() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 18
face_semigroup_algebra() (sage.geometry.polyhedron.representation.arrangement.HyperplaneArrangementElement method), 19
face_split() (sage.geometry.polyhedron.base.Polyhedron_base method), 353
face_truncation() (sage.geometry.polyhedron.base.Polyhedron_base method), 354
face_vector() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 20

522 Index
FaceFan() (in module sage.geometry.fan), 262
faces() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 221
faces() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 129
faces() (sage.geometry.polyhedron.base.Polyhedron_base method), 355
faces_lp() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 130
facet_adjacency_matrix() (sage.geometry.polyhedron.base.Polyhedron_base method), 356
facet_constant() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 130
facet_constants() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 131
facet_normal() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 131
facet_normals() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 223
facet_normals() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 132
facet_of() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 224
facet_of() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 133
facets() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 224
facets() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 133
facets() (sage.rings.polynomial.groebner_fan.PolyhedralCone method), 325
facets_lp() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 133
factor() (sage.geometry.fan_morphism.FanMorphism method), 290
Fan() (in module sage.geometry.fan), 262
fan() (sage.geometry.triangulation.element.Triangulation method), 460
Fan2d() (in module sage.geometry.fan), 265
fan_2d_cyclically_ordered_rays() (in module sage.geometry.fan_isomorphism), 495
fan_2d_echelon_form() (in module sage.geometry.fan_isomorphism), 495
fan_2d_echelon_forms() (in module sage.geometry.fan_isomorphism), 496
fan_isomorphic_necessary_conditions() (in module sage.geometry.fan_isomorphism), 496
fan_isomorphism_generator() (in module sage.geometry.fan_isomorphism), 496
FanMorphism (class in sage.geometry.fan_morphism), 287
FanNotIsomorphicError, 495
farthest_point() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 439
felsner_matrix() (sage.geometry.pseudolines.PseudolineArrangement method), 490
fibration_generator() (sage.geometry.polyhedron.base_ZZ.Polyhedron_ZZ method), 404
fibration_generator() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope_PPL_class method), 180
find_isomorphism() (in module sage.geometry.fan_isomorphism), 498
find_isomorphism() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope_PPL_class method), 170
find_translation() (sage.geometry.polyhedron.base_ZZ.Polyhedron_ZZ method), 404
first_coordinate_plane() (sage.geometry.polyhedron.double_description.DoubleDescriptionPair method), 420
flow_polytope() (sage.geometry.polyhedron.library.Polytopes static method), 59
G
G_semiorder() (sage.geometry.hyperplane_arrangement.library.HyperplaneArrangementLibrary method), 35
G_Shi() (sage.geometry.hyperplane_arrangement.library.HyperplaneArrangementLibrary method), 34
Gale_transform() (sage.geometry.fan.RationalPolyhedralFan method), 267
gale_transform() (sage.geometry.polyhedron.base.Polyhedron_base method), 356
Gale_transform() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 435
gen() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangements method), 33
gen() (sage.geometry.linear_expression.LinearExpressionModule method), 470
generating_cone() (sage.geometry.fan.RationalPolyhedralFan method), 275
generating_cones() (sage.geometry.fan.RationalPolyhedralFan method), 276
gens() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangements method), 33
gens() (sage.geometry.linear_expression.LinearExpressionModule method), 470
Index

gens() (sage.geometry.toric_lattice.ToricLattice_quotient method), 199
genus() (sage.geometry.ribbon_graph.RibbonGraph method), 480
get_integral_point() (sage.geometry.polyhedron.base.Polyhedron_base method), 357
gfan() (sage.rings.polynomial.groebner_fan.GroebnerFan method), 318
gkz_phi() (sage.geometry.triangulation.element.Triangulation method), 460
Gosset_3_21() (sage.geometry.polyhedron.library.Polytopes method), 54
grand_antiprism() (sage.geometry.polyhedron.library.Polytopes method), 61
graph() (sage.geometry.polyhedron.base.Polyhedron_base method), 358
graphical() (sage.geometry.hyperplane_arrangement.library.HyperplaneArrangementLibrary method), 37
great_rhombicuboctahedron() (sage.geometry.polyhedron.library.Polytopes method), 62
groebner_cone() (sage.rings.polynomial.groebner_fan.ReducedGroebnerBasis method), 329
GroebnerFan (class in sage.rings.polynomial.groebner_fan), 317

H

has_good_reduction() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 20
has_IP_property() (sage.geometry.polyhedron.base.ZZ.Polyhedron_ZZ method), 404
Hilbert_basis() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 209
Hilbert_coefficients() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 211
hodge_numbers() (sage.geometry.lattice_polytope.NefPartition method), 153
homogeneity_space() (sage.rings.polynomial.groebner_fan.GroebnerFan method), 319
homogeneous_vector() (sage.geometry.polyhedron.representation.Line method), 95
homogeneous_vector() (sage.geometry.polyhedron.representation.Ray method), 97
homogeneous_vector() (sage.geometry.polyhedron.representation.Vertex method), 98
homology_basis() (sage.geometry.polyhedron.library.Polytopes method), 481
Hrep2Vrep (class in sage.geometry.polyhedron.double_description_inhomogeneous), 426
Hrep_generator() (sage.geometry.polyhedron.base.Polyhedron_base method), 331
Hrepresentation (class in sage.geometry.polyhedron.representation), 91
Hrepresentation() (sage.geometry.polyhedron.base.Polyhedron_base method), 332
Hrepresentation_space() (sage.geometry.polyhedron.base.Polyhedron_base method), 332
Hrepresentation_space() (sage.geometry.polyhedron.parent.Polyhedra_base method), 86
Hrepresentation_str() (sage.geometry.polyhedron.base.Polyhedron_base method), 332
hypercube() (sage.geometry.polyhedron.library.Polytopes method), 62
Hyperplane (class in sage.geometry.hyperplane_arrangement.hyperplane), 41
hyperplane_arrangement() (sage.geometry.polyhedron.library.Polytopes method), 63
HyperplaneArrangementElement (class in sage.geometry.hyperplane_arrangement.arrangement), 8
HyperplaneArrangementLibrary (class in sage.geometry.hyperplane_arrangement.library), 34
HyperplaneArrangements (class in sage.geometry.hyperplane_arrangement.arrangement), 32
hyperplanes() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 21
hypersimplex() (sage.geometry.polyhedron.library.Polytopes method), 63

I

icosahedron() (sage.geometry.polyhedron.library.Polytopes method), 63
icosidodecahedron() (sage.geometry.polyhedron.library.Polytopes method), 64
icosidodecahedron_V2() (sage.geometry.polyhedron.library.Polytopes method), 64
ideal() (sage.rings.polynomial.groebner_fan.GroebnerFan method), 319
ideal() (sage.rings.polynomial.groebner_fan.ReducedGroebnerBasis method), 329
ideal_to_gfan_format() (in module sage.rings.polynomial.groebner_fan), 330
identity() (sage.geometry.polyhedron.plot.Projection method), 103
Index
is_affine() (sage.geometry.triangulation.base.PointConfiguration_base method), 455
is_birational() (sage.geometry.fan_morphism.FanMorphism method), 293
is_bounded() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope_PPL_class method), 182
is_bundle() (sage.geometry.fan_morphism.FanMorphism method), 294
is_central() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 21
is_combinatorially_isomorphic() (sage.geometry.polyhedron.base.Polyhedron_base method), 364
is_compact() (sage.geometry.polyhedron.base.Polyhedron_base method), 366
is_complete() (sage.geometry.fan.RationalPolyhedralFan method), 276
is_Cone() (in module sage.geometry.cone), 254
is_dominant() (sage.geometry.fan_morphism.FanMorphism method), 294
is_empty() (sage.geometry.polyhedron.base.Polyhedron_base method), 366
is_equation() (sage.geometry.polyhedron.representation.Equation method), 90
is_equation() (sage.geometry.polyhedron.representation.Hrepresentation method), 92
is_equivalent() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 226
is_equivalent() (sage.geometry.fan.RationalPolyhedralFan method), 276
is_essential() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 21
is_extremal() (sage.geometry.polyhedron.double_description.DoubleDescriptionPair method), 421
is_face_of() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 226
is_Fan() (in module sage.geometry.fan), 286
is_fibration() (sage.geometry.fan_morphism.FanMorphism method), 295
is_formal() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 22
is_free() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 22
is_full_dimensional() (sage.geometry.polyhedron.base.Polyhedron_base method), 367
is_full_space() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 227
is_H() (sage.geometry.polyhedron.representation.Hrepresentation method), 92
is_incident() (sage.geometry.polyhedron.representation.Hrepresentation method), 92
is_incident() (sage.geometry.polyhedron.representation.Vrepresentation method), 100
is_inequality() (sage.geometry.polyhedron.representation.Hrepresentation method), 93
is_inequality() (sage.geometry.polyhedron.representation.Inequality method), 94
is_injective() (sage.geometry.fan_morphism.FanMorphism method), 296
is_inscribed() (sage.geometry.polyhedron.base.Polyhedron_base method), 367
is_integral() (sage.geometry.polyhedron.representation.Vertex method), 99
is_isomorphic() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 228
is_isomorphic() (sage.geometry.fan.RationalPolyhedralFan method), 277
is_isomorphic() (sage.geometry.polyhedron.ppl_lattice_polygon.LatticePolygon_PPL_class method), 171
is_lattice_polytope() (sage.geometry.polyhedron.base.Polyhedron_base method), 368
is_lattice_polytope() (sage.geometry.polyhedron.base_ZZ.Polyhedron_ZZ method), 405
is_LatticePolytope() (in module sage.geometry.lattice_polytope), 162
is_line() (sage.geometry.polyhedron.representation.Line method), 95
is_line() (sage.geometry.polyhedron.representation.Vrepresentation method), 100
is_linear() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 23
is_Minkowski_summand() (sage.geometry.polyhedron.base.Polyhedron_base method), 364
is_Minkowski_summand() (sage.geometry.polyhedron.base.Polyhedron_base method), 368
is_NefPartition() (in module sage.geometry.lattice_polytope), 162
is_neighborly() (sage.geometry.polyhedron.base.Polyhedron_base method), 369
is_PointCollection() (in module sage.geometry.point_collection), 307
is_Polyhedron() (in module sage.geometry.polyhedron.base), 400
is_proper() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 228
is_ray() (sage.geometry.polyhedron.representation.Ray method), 98
is_ray() (sage.geometry.polyhedron.representation.Vrepresentation method), 101
is_reflexive() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 135
is_reflexive() (sage.geometry.polyhedron.base_ZZ.Polyhedron_ZZ method), 405
is_separating_hyperplane() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 23
is_simple() (sage.geometry.polyhedron.base.Polyhedron_base method), 369
is_simplex() (sage.geometry.polyhedron.base.Polyhedron_base method), 370
is_simplex() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope_PPL_class method), 182
is_simplicial() (sage.geometry.fan.RationalPolyhedralFan method), 278
is_simplicial() (sage.geometry.polyhedron.base.Polyhedron_base method), 370
is_simplicial() (sage.rings.polynomial.groebner_fan.PolyhedralFan method), 327
is_smooth() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 229
is_smooth() (sage.geometry.polyhedron.base.Polyhedron_base method), 370
is_solid() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 229
is_strictly_convex() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 230
is_surjective() (sage.geometry.fan_morphism.FanMorphism method), 297
is_ToricLattice() (in module sage.geometry.toric_lattice), 202
is_ToricLatticeQuotient() (in module sage.geometry.toric_lattice), 203
is_torsion_free() (sage.geometry.toric_lattice.ToricLattice_quotient method), 199
is_trivial() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 230
is_universe() (sage.geometry.polyhedron.base.Polyhedron_base method), 370
is_V() (sage.geometry.polyhedron.representation.Vrepresentation method), 100
is_vertex() (sage.geometry.polyhedron.representation.Vertex method), 99
is_vertex() (sage.geometry.polyhedron.representation.Vrepresentation method), 101
Ish() (sage.geometry.hyperplane_arrangement.library.HyperplaneArrangementLibrary method), 35
isomorphism() (sage.geometry.fan.RationalPolyhedralFan method), 279

J
join() (sage.geometry.polyhedron.base.Polyhedron_base method), 371

K
kernel_fan() (sage.geometry.fan_morphism.FanMorphism method), 297
Kirkman_icosahedron() (sage.geometry.polyhedron.library.Polytopes method), 55

L
label_list() (in module sage.geometry.toric_plotter), 313
last_slope() (sage.geometry.newton_polygon.NewtonPolygon_element method), 471
lattice() (sage.geometry.cone.IntegralRayCollection method), 249
lattice() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 136
lattice_automorphism_group() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope_PPL_class method), 182
lattice_dim() (sage.geometry.cone.IntegralRayCollection method), 250
lattice_dim() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 136
lattice_from_incidences() (in module sage.geometry.hasse_diagram), 499
lattice_polytope() (sage.geometry.polyhedron.base.Polyhedron_base method), 371
LatticeEuclideanGroupElement (class in sage.geometry.polyhedron.lattice_euclidean_group_element), 167
LatticePolygon_PPL_class (class in sage.geometry.polyhedron.lattice_polygon), 170
LatticePolytope() (in module sage.geometry.lattice_polytope), 118
LatticePolytope_PPL() (in module sage.geometry.polyhedron.ppl_lattice_polytope), 175
LatticePolytope_PPL_class (class in sage.geometry.polyhedron.ppl_lattice_polytope), 175
LatticePolytopeClass (class in sage.geometry.lattice_polytope), 119
LatticePolytopeError, 168
LatticePolytopeNoEmbeddingError, 168
LatticePolytopesNotIsomorphicError, 168
legend_3d() (in module sage.geometry.hyperplane_arrangement.plot), 50
less_generators() (in module sage.geometry.hyperplane_arrangement.check_freeness), 509
lexicographic_triangulation() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 440
Line (class in sage.geometry.polyhedron.representation), 95
line() (in module sage.geometry.polyhedron.backend_ppl), 416
line() (in module sage.geometry.polyhedron.ppl_lattice_polytope), 186
line_generator() (sage.geometry.polyhedron.base.Polyhedron_base method), 372
linearity() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 113
linearity_dim() (sage.rings.polynomial.groebner_fan.PolyhedralCone method), 325
linearity_dim() (sage.rings.polynomial.groebner_fan.PolyhedralFan method), 327
linear_equivalence_ideal() (sage.geometry.fan.RationalPolyhedralFan method), 280
linear_part() (sage.geometry.hyperplane_arrangement.affine_subspace.AffineSubspace method), 48
linear_part() (sage.geometry.hyperplane_arrangement.hyperplane.Hyperplane method), 42
linear_part_projection() (sage.geometry.hyperplane_arrangement.hyperplane.Hyperplane method), 42
linear_subspace() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 231
LinearExpression (class in sage.geometry.linear_expression), 465
LinearExpressionModule (class in sage.geometry.linear_expression), 468
linearly_independent_vertices() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 136
lines() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 232
lines() (sage.geometry.polyhedron.base.Polyhedron_base method), 372
lines() (sage.geometry.polyhedron.face.PolyhedronFace method), 114
lines_list() (sage.geometry.polyhedron.base.Polyhedron_base method), 373
linial() (sage.geometry.hyperplane_arrangement.library.HyperplaneArrangementLibrary method), 37
loop_over_parallelepiped_points() (in module sage.geometry.integral_points), 504
lyapunov_like_basis() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 232
lyapunov_rank() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 233

M
make_generic() (sage.geometry.ribbon_graph.RibbonGraph method), 482
make_parent() (in module sage.geometry.hyperplane_arrangement.library), 38
make_ribbon() (in module sage.geometry.ribbon_graph), 486
make_simplicial() (sage.geometry.fan.RationalPolyhedralFan method), 280
matrix() (sage.geometry.point_collection.PointCollection method), 304
matrix_space() (sage.geometry.polyhedron.double_description.DoubleDescriptionPair method), 421
matroid() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 24
max_degree() (in module sage.rings.polynomial.groebner_fan), 330
maximal_cones() (sage.rings.polynomial.groebner_fan.PolyhedralFan method), 327
maximal_total_degree_of_a_graebner_basis() (sage.rings.polynomial.groebner_fan.GroebnerFan method), 319
minimal_generated_number() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 24
minimal_total_degree_of_a_graebner_basis() (sage.rings.polynomial.groebner_fan.GroebnerFan method), 319
Minkowski_decompositions() (sage.geometry.polyhedron.base_ZZ.Polyhedron_ZZ method), 402
minkowski_decompositions() (sage.geometry.polyhedron.base_ZZ.Polyhedron_ZZ method), 405
Index 529
normalize_rays() (in module sage.geometry.cone), 254
nparts() (sage.geometry.lattice_polytope.NefPartition method), 155
npoints() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 141
nrays() (sage.geometry.cone.IntegralRayCollection method), 250
number_boundaries() (sage.geometry.ribbon_graph.RibbonGraph method), 484
number_of_reduced_groebner_bases() (sage.rings.polynomial.groebner_fan.GroebnerFan method), 320
number_of_variables() (sage.rings.polynomial.groebner_fan.GroebnerFan method), 320
nvertices() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 141

O
octahedron() (sage.geometry.polyhedron.library.Polytopes method), 65
one_point_suspension() (sage.geometry.polyhedron.base.Polyhedron_base method), 378
options() (in module sage.geometry.toric_plotter), 314
ordered_vertices() (sage.geometry.polyhedron.ppl_lattice_polygon.LatticePolygon_PPL_class method), 171
oriented_boundary() (sage.geometry.fan.RationalPolyhedralFan method), 281
origin() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 141
orlik_solomon_algebra() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 25
orthogonal_projection() (sage.geometry.hyperplane_arrangement.hyperplane.Hyperplane method), 43
orthogonal_sublattice() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 235
output_format() (sage.geometry.point_collection.PointCollection static method), 305

P
pair_class (sage.geometry.polyhedron.double_description.Problem attribute), 423
pair_class (sage.geometry.polyhedron.double_description.StandardAlgorithm attribute), 424
PALReader (class in sage.geometry.polyhedron.palp_database), 169
parallelotope() (sage.geometry.polyhedron.library.Polytopes method), 65
parallelotope_points() (in module sage.geometry.integral_points), 504
parent() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 142
ParentNewtonPolygon (class in sage.geometry.newton_polygon), 473
part() (sage.geometry.lattice_polytope.NefPartition method), 155
part_of() (sage.geometry.lattice_polytope.NefPartition method), 156
part_of_point() (sage.geometry.lattice_polytope.NefPartition method), 156
parts() (sage.geometry.lattice_polytope.NefPartition method), 157
pentakis_dodecahedron() (sage.geometry.polyhedron.library.Polytopes method), 66
permutahedron() (sage.geometry.polyhedron.library.Polytopes method), 66
permutations() (sage.geometry.pseudolines.PseudolineArrangement method), 490
PivotedInequalities (class in sage.geometry.polyhedron.double_description_inhomogeneous), 427
placing_triangulation() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 440
plot() (in module sage.geometry.hyperplane_arrangement.plot), 50
plot() (sage.geometry.convex_polygon.ConvexPolygon_class method), 236
plot() (sage.geometry.convex_polygon.IntegralRayCollection method), 250
plot() (sage.geometry.fan.RationalPolyhedralFan method), 282
plot() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 25
plot() (sage.geometry.hyperplane_arrangement.hyperplane.Hyperplane method), 44
plot() (sage.geometry.newton_polygon.NewtonPolygon_element method), 471
plot() (sage.geometry.polyhedron_base.Polyhedron_base method), 378
plot() (sage.geometry.polyhedron.ppl_lattice_polygon.LatticePolygon_PPL_class method), 171
plot() (sage.geometry.toric_lattice.ToricLattice_ambient method), 192
plot() (sage.geometry.toric_lattice.ToricLattice_sublattice_with_basis method), 202
plot() (sage.geometry.triangulation.element.Triangulation method), 462
plot() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 441
plot() (sage.geometry.voronoi_diagram.VoronoiDiagram method), 492
plot3d() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 142
plot_generators() (sage.geometry.toric_plotter.ToricPlotter method), 310
plot_hyperplane() (in module sage.geometry.hyperplane_arrangement.plot), 51
plot_labels() (sage.geometry.toric_plotter.ToricPlotter method), 310
plot_lattice() (sage.geometry.toric_plotter.ToricPlotter method), 311
plot_points() (sage.geometry.toric_plotter.ToricPlotter method), 311
plot_ray_labels() (sage.geometry.toric_plotter.ToricPlotter method), 311
plot_rays() (sage.geometry.toric_plotter.ToricPlotter method), 312
plot_walls() (sage.geometry.toric_plotter.ToricPlotter method), 312
poincare_polynomial() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 26
Point (class in sage.geometry.triangulation.base), 450
point() (in module sage.geometry.polyhedron.backend_ppl), 417
point() (in module sage.geometry.polyhedron.ppl_lattice_polytope), 186
point() (sage.geometry.hyperplane_arrangement.affine_subspace.AffineSubspace method), 48
point() (sage.geometry.hyperplane_arrangement.hyperplane.Hyperplane method), 44
point() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 143
point() (sage.geometry.triangulation.base.PointConfiguration_base method), 456
point_configuration() (sage.geometry.triangulation.base.PointConfiguration_base method), 451
point_configuration() (sage.geometry.triangulation.element.Triangulation method), 462
PointCollection (class in sage.geometry.point_collection), 301
PointConfiguration (class in sage.geometry.triangulation.point_configuration), 434
PointConfiguration_base (class in sage.geometry.triangulation.base), 454
points() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 143
points() (sage.geometry.triangulation.base.PointConfiguration_base method), 456
points() (sage.geometry.voronoi_diagram.VoronoiDiagram method), 493
pointsets_mod_automorphism() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope_PPL_class method), 184
polar() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 145
polar() (sage.geometry.polyhedron.base.Polyhedron_base method), 380
polar() (sage.geometry.polyhedron.base_ZZ.Polyhedron_ZZ method), 406
polar_P1xP1_polytope() (in module sage.geometry.polyhedron.ppl_lattice_polygon), 172
polar_P2_112_polytope() (in module sage.geometry.polyhedron.ppl_lattice_polygon), 172
polar_P2_polytope() (in module sage.geometry.polyhedron.ppl_lattice_polygon), 173
poly_x() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 145
Polyhedra() (in module sage.geometry.polyhedron.parent), 85
Polyhedra_base (class in sage.geometry.polyhedron.parent), 86
Polyhedra_field (class in sage.geometry.polyhedron.parent), 89
Polyhedra_pymake (class in sage.geometry.polyhedron.parent), 89
Polyhedra_QQ_cdd (class in sage.geometry.polyhedron.parent), 85
Polyhedra_QQ_normaliz (class in sage.geometry.polyhedron.parent), 85
Polyhedra_QQ_ppl (class in sage.geometry.polyhedron.parent), 86
Polyhedra_RDF_cdd (class in sage.geometry.polyhedron.parent), 86
Polyhedra_ZZ_normaliz (class in sage.geometry.polyhedron.parent), 86
Polyhedra_ZZ_ppl (class in sage.geometry.polyhedron.parent), 86
PolyhedralCone (class in sage.rings.polynomial.groebner_fan), 325
PolyhedralFan (class in sage.rings.polynomial.groebner_fan), 326
polyhedralfan() (sage.rings.polynomial.groebner_fan.GroebnerFan method), 320
Polyhedron() (in module sage.geometry.polyhedron.constructor), 82
polyhedron() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 236
polyhedron() (sage.geometry.hyperplane_arrangement.hyperplane.Hyperplane method), 44
polyhedron() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 146
polyhedron() (sage.geometry.polyhedron.face.PolyhedronFace method), 115
polyhedron() (sage.geometry.polyhedron.representation.PolyhedronRepresentation method), 96
Polyhedron_base (class in sage.geometry.polyhedron.base), 331
Polyhedron_cdd (class in sage.geometry.polyhedron.backend_cdd), 408
Polyhedron_field (class in sage.geometry.polyhedron.backend_field), 408
Polyhedron_normaliz (class in sage.geometry.polyhedron.backend_normaliz), 409
Polyhedron_pokymake (class in sage.geometry.polyhedron.backend_pokymake), 414
Polyhedron_ppl (class in sage.geometry.polyhedron.backend_ppl), 416
Polyhedron_QQ (class in sage.geometry.polyhedron.backend_QQ), 401
Polyhedron_QQ_cdd (class in sage.geometry.polyhedron.backend_cdd), 407
Polyhedron_QQ_normaliz (class in sage.geometry.polyhedron.backend_normaliz), 409
Polyhedron_QQ_pokymake (class in sage.geometry.polyhedron.backend_pokymake), 413
Polyhedron_QQ_ppl (class in sage.geometry.polyhedron.backend_ppl), 416
Polyhedron_RDF (class in sage.geometry.polyhedron.backend_RDF), 407
Polyhedron_RDF_cdd (class in sage.geometry.polyhedron.backend_cdd), 407
Polyhedron_RDF_ZZ (class in sage.geometry.polyhedron.backend_ZZ), 402
Polyhedron_ZZ_normaliz (class in sage.geometry.polyhedron.backend_normaliz), 409
Polyhedron_ZZ_pokymake (class in sage.geometry.polyhedron.backend_pokymake), 414
Polyhedron_ZZ_ppl (class in sage.geometry.polyhedron.backend_ppl), 416
PolyhedronFace (class in sage.geometry.polyhedron.face), 111
PolyhedronRepresentation (class in sage.geometry.polyhedron.representation), 96
Polytopes (class in sage.geometry.polyhedron.library), 54
positive_circuits() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 441
positive_integer_relations() (in module sage.geometry.lattice_polytope), 163
positive_operators_gens() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 236
PPL_point() (in module sage.geometry.cone), 252
PPL_point() (in module sage.geometry.lattice_polytope), 158
PPL_point() (in module sage.geometry.lattice_polytope), 158
prefix_check() (in module sage.rings.polynomial.groebner_fan), 330
preimage_cones() (sage.geometry.fan_morphism.FanMorphism method), 298
preimage_fan() (sage.geometry.fan_morphism.FanMorphism method), 298
prepare_inner_loop() (sage.geometry.integral_points.InequalityCollection method), 501
prepare_next_to_inner_loop() (sage.geometry.integral_points.InequalityCollection method), 502
primitive() (sage.geometry.hyperplane_arrangement.hyperplane.Hyperplane method), 45
primitive_collections() (sage.geometry.fan.RationalPolyhedralFan method), 282
primitive_preimage_cones() (sage.geometry.fan_morphism.FanMorphism method), 299
print_cache() (in module sage.geometry.integral_points), 505
prism() (sage.geometry.polyhedron.base.Polyhedron_base method), 380
Problem (class in sage.geometry.polyhedron.double_description), 422
product() (sage.geometry.polyhedron.base.Polyhedron_base method), 380
project_points() (in module sage.geometry.polyhedron.library), 77
Projection (class in sage.geometry.polyhedron.plot), 102
projection() (sage.geometry.polyhedron.base.Polyhedron_base method), 381
projection_func_identity() (in module sage.geometry.polyhedron.plot), 110
ProjectionFuncSchlegel (class in sage.geometry.polyhedron.plot), 109
ProjectionFuncStereographic (class in sage.geometry.polyhedron.plot), 109
projective() (sage.geometry.triangulation.base.Point method), 452
PseudolineArrangement (class in sage.geometry.pseudolines), 489
pushing_triangulation() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 442
pyramid() (sage.geometry.polyhedron.base.Polyhedron_base method), 381

Q

quotient() (sage.geometry.toric_lattice.ToricLattice_generic method), 194

R

R_by_sign() (sage.geometry.polyhedron.double_description.DoubleDescriptionPair method), 419
radius() (sage.geometry.polyhedron.base.Polyhedron_base method), 381
radius_square() (sage.geometry.polyhedron.base.Polyhedron_base method), 382
random_cone() (in module sage.geometry.cone), 255
random_element() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 238
random_element() (sage.geometry.linear_expression.LinearExpressionModule method), 470
random_inequalities() (in module sage.geometry.polyhedron.double_description), 425
random_integral_point() (sage.geometry.polyhedron.base.Polyhedron_base method), 382
rank() (sage.geometry.toric_lattice.arrangement.HyperplaneArrangementElement method), 26
rank() (sage.geometry.toric_lattice.ToricLattice_quotient method), 199
RationalPolyhedralFan (class in sage.geometry.fan), 266
Ray (class in sage.geometry.polyhedron.representation), 97
ray() (in module sage.geometry.polyhedron.backend_ppl), 417
ray() (in module sage.geometry.polyhedron.ppl_lattice_polytope), 187
ray() (sage.geometry.cone.IntegralRayCollection method), 250
ray_generator() (sage.geometry.polyhedron.base.Polyhedron_base method), 383
ray_generator() (sage.geometry.polyhedron.face.PolyhedronFace method), 115
ray_matrix_normal_form() (in module sage.geometry.integral_points), 505
rays() (sage.geometry.cone.IntegralRayCollection method), 251
rays() (sage.geometry.polyhedron.base.Polyhedron_base method), 383
rays() (sage.geometry.polyhedron.face.PolyhedronFace method), 115
rays() (sage.rings.polynomial.groebner_fan.InitialForm method), 324
rays() (sage.rings.polynomial.groebner_fan.PolyhedralFan method), 328
rays_list() (sage.geometry.polyhedron.base.Polyhedron_base method), 383
read_all_polytopes() (in module sage.geometry.lattice_polytope), 163
read_palp_matrix() (in module sage.geometry.lattice_polytope), 164
read_palp_point_collection() (in module sage.geometry.point_collection), 307
rectangular_box_points() (in module sage.geometry.integral_points), 506
recycle() (sage.geometry.polyhedron.parent.Polyhedra_base method), 88
reduced() (sage.geometry.ribbon_graph.RibbonGraph method), 484
reduced_affine() (sage.geometry.triangulation.base.Point method), 452
reduced_affine_vector() (sage.geometry.triangulation.base.Point method), 452
reduced_affine_vector_space() (sage.geometry.triangulation.base.PointConfiguration_base method), 456
reduced_groebner_bases() (sage.rings.polynomial.groebner_fan.GroebnerFan method), 320
reduced_projective() (sage.geometry.triangulation.base.Point method), 453
reduced_projective_vector() (sage.geometry.triangulation.base.Point method), 453
ReducedGroebnerBasis (class in sage.rings.polynomial.groebner_fan), 328
Reflexive4dHodge (class in sage.geometry.polyhedron.pulp_database), 169
ReflexivePolytope() (in module sage.geometry.lattice_polytope), 158

Index 533
ReflexivePolytopes() (in module sage.geometry.lattice_polytope), 159
region_containing_point() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 26
regions() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 27
regions() (sage.geometry.voronoi_diagram.VoronoiDiagram method), 493
regular_polygon() (sage.geometry.polyhedron.library.Polytopes method), 67
relative_interior_contains() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 239
relative_interior_contains() (sage.geometry.polyhedron.base.Polyhedron_base method), 383
relative_interior_point() (sage.rings.polynomial.groebner_fan.PolyhedralCone method), 326
relative_orthogonal_quotient() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 240
relative_quotient() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 241
relative_star_generators() (sage.geometry.fan.morphism.FanMorphism method), 300
render() (sage.rings.polynomial.groebner_fan.GroebnerFan method), 321
render3d() (sage.rings.polynomial.groebner_fan.GroebnerFan method), 322
render_0d() (sage.geometry.polyhedron.plot.Projection method), 103
render_1d() (sage.geometry.polyhedron.plot.Projection method), 103
render_2d() (sage.geometry.polyhedron.plot.Projection method), 103
render_3d() (sage.geometry.polyhedron.plot.Projection method), 104
render_fill_2d() (sage.geometry.polyhedron.plot.Projection method), 105
render_line_1d() (sage.geometry.polyhedron.plot.Projection method), 105
render outline_2d() (sage.geometry.polyhedron.plot.Projection method), 105
render_points_1d() (sage.geometry.polyhedron.plot.Projection method), 105
render solid() (sage.geometry.polyhedron.base.Polyhedron_base method), 384
render solid_3d() (sage.geometry.polyhedron.plot.Projection method), 106
render vertices_3d() (sage.geometry.polyhedron.plot.Projection method), 106
render_wireframe() (sage.geometry.polyhedron.base.Polyhedron_base method), 384
render_wireframe_3d() (sage.geometry.polyhedron.plot.Projection method), 106
repr_pretty() (in module sage.geometry.polyhedron.representation), 101
repr_pretty() (sage.geometry.polyhedron.representation.Hrepresentation method), 93
repr_pretty_Hrepresentation() (sage.geometry.polyhedron.base.Polyhedron_base method), 384
representative_point() (sage.geometry.polyhedron.base.Polyhedron_base method), 384
reset_options() (in module sage.geometry.toric_plotter), 316
restrict_to_connected_triangulations() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 442
restrict_to_fine_triangulations() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 443
restrict_to_regular_triangulations() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 443
restrict_to_star_triangulations() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 444
restricted_automorphism_group() (sage.geometry.polyhedron.base.Polyhedron_base method), 385
restricted_automorphism_group() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope_PPL_class method), 184
restricted_automorphism_group() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 444
restriction() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 27
reverse() (sage.geometry.newton_polygon.NewtonPolygon_element method), 471
rho() (sage.geometry.ribbon_graph.RibbonGraph method), 485
rhombic_dodecahedron() (sage.geometry.polyhedron.library.Polytopes method), 68
rhombicosidodecahedron() (sage.geometry.polyhedron.library.Polytopes method), 68
RibbonGraph (class in sage.geometry.ribbon_graph), 474
ring() (sage.rings.polynomial.groebner_fan.GroebnerFan method), 322
ring_to_gfan_format() (in module sage.rings.polynomial.groebner_fan), 331
run() (sage.geometry.polyhedron.double_description.StandardAlgorithm method), 424

S
sage.geometry.cone (module), 203
sage.geometry.fan (module), 257
sage.geometry.fan_isomorphism (module), 495
sage.geometry.fan_morphism (module), 286
sage.geometry.hasse_diagram (module), 499
sage.geometry.hyperplane_arrangement.affine_subspace (module), 46
sage.geometry.hyperplane_arrangement.arrangement (module), 3
sage.geometry.hyperplane_arrangement.check_freeness (module), 509
sage.geometry.hyperplane_arrangement.hyperplane (module), 38
sage.geometry.hyperplane_arrangement.library (module), 34
sage.geometry.hyperplane_arrangement.plot (module), 48
sage.geometry.integral_points (module), 500
sage.geometry.lattice_polytope (module), 117
sage.geometry.linear_expression (module), 465
sage.geometry.newton_polygon (module), 471
sage.geometry.point_collection (module), 300
sage.geometry.polyhedron.backend_cdd (module), 407
sage.geometry.polyhedron.backend_field (module), 408
sage.geometry.polyhedron.backend_normaliz (module), 408
sage.geometry.polyhedron.backend_polspace (module), 413
sage.geometry.polyhedron.backend_ppl (module), 416
sage.geometry.polyhedron.base (module), 331
sage.geometry.polyhedron.base_QQ (module), 401
sage.geometry.polyhedron.base_RDF (module), 407
sage.geometry.polyhedron.base_ZZ (module), 402
sage.geometry.polyhedron.cdd_file_format (module), 116
sage.geometry.polyhedron.constructor (module), 78
sage.geometry.polyhedron.double_description (module), 418
sage.geometry.polyhedron.double_description_inhomogeneous (module), 425
sage.geometry.polyhedron.face (module), 111
sage.geometry.polyhedron.lattice_euclidean_group_element (module), 167
sage.geometry.polyhedron.library (module), 53
sage.geometry.polyhedron.palp_database (module), 168
sage.geometry.polyhedron.parent (module), 85
sage.geometry.polyhedron.plot (module), 102
sage.geometry.polyhedron.ppl_lattice_polygon (module), 170
sage.geometry.polyhedron.ppl_lattice_polytope (module), 174
sage.geometry.polyhedron.representation (module), 90
sage.geometry.pseudolines (module), 487
sage.geometry.ribbon_graph (module), 474
sage.geometry.toric_lattice (module), 188
sage.geometry.toric_plotter (module), 308
sage.geometry.triangulation.base (module), 449
sage.geometry.triangulation.element (module), 458
sage.geometry.triangulation.point_configuration (module), 431
sage.geometry.voronoi_diagram (module), 491

Index 535
sage.rings.polynomial.groebner_fan (module), 317
satisfied_as_equalities() (sage.geometry.integral_points.InequalityCollection method), 502
saturation() (sage.geometry.toric_lattice.ToricLattice.Generic method), 195
schlegel() (sage.geometry.polyhedron.plot.Projection method), 106
schlegel_projection() (sage.geometry.polyhedron.base.Polyhedron_base method), 387
secondary_polytope() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 445
sector() (in module sage.geometry.toric_plotter), 316
semigroup_generators() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 242
semiorder() (sage.geometry.hyperplane_arrangement.library.HyperplaneArrangementLibrary method), 38
set() (sage.geometry.point_collection.PointCollection method), 306
set_engine() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 446
set_immutable() (sage.geometry.toric_lattice.ToricLattice_quotient_element method), 200
set_palp_dimension() (in module sage.geometry.lattice_polytope), 165
set_rays() (sage.geometry.toric_plotter.ToricPlotter method), 312
SetOfAllLatticePolytopesClass (class in sage.geometry.lattice_polytope), 159
Shi() (sage.geometry.hyperplane_arrangement.library.HyperplaneArrangementLibrary method), 35
show() (sage.geometry.polyhedron.base.Polyhedron_base method), 388
show() (sage.geometry.pseudolines.PseudolineArrangement method), 490
show3d() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 146
sign_vector() (sage.geometry.ribbon_graph.RibbonGraph method), 485
simplex() (sage.geometry.polyhedron.library.Polytopes method), 69
simplex_points() (in module sage.geometry.integral_points), 508
simplex_to_int() (sage.geometry.triangulation.base.PointConfiguration_base method), 457
simplicial_complex() (sage.geometry.triangulation.element.Triangulation method), 463
six_hundred_cell() (sage.geometry.polyhedron.library.Polytopes method), 70
skeleton() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 146
skeleton_points() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 147
skeleton_show() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 147
skip_palp_matrix() (in module sage.geometry.lattice_polytope), 165
slopes() (sage.geometry.newton_polygon.NewtonPolygon_element method), 472
small_rhombicuboctahedron() (sage.geometry.polyhedron.library.Polytopes method), 70
snub_cube() (sage.geometry.polyhedron.library.Polytopes method), 71
snub_dodecahedron() (sage.geometry.polyhedron.library.Polytopes method), 71
solid_restriction() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 243
some_elements() (sage.geometry.polyhedron.parent.Polyhedra_base method), 89
span() (sage.geometry.cone.IntegralRayCollection method), 251
span() (sage.geometry.toric_lattice.ToricLattice.Generic method), 195
span_of_basis() (sage.geometry.toric_lattice.ToricLattice.Generic method), 196
stack() (sage.geometry.polyhedron.base.Polyhedron_base method), 388
StandardAlgorithm (class in sage.geometry.polyhedron.double_description), 424
StandardDoubleDescriptionPair (class in sage.geometry.polyhedron.double_description), 424
Stanley_Reisner_ideal() (sage.geometry.fan.RationalPolyhedralFan method), 267
star_center() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 446
star_generator_indices() (sage.geometry.fan.Cone_of_fan method), 261
star_generators() (sage.geometry.fan.Cone_of_fan method), 262
stereographic() (sage.geometry.polyhedron.plot.Projection method), 107
strict_quotient() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 244
sub_polytope_generator() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope_PPL_class method), 185
sub_polytopes() (sage.geometry.polyhedron.ppl_lattice_polygon.LatticePolygon_PPL_class method), 172
sub_reflexive_polygons() (in module sage.geometry.polyhedron.ppl_lattice_polygon), 173
subdirect_sum() (sage.geometry.polyhedron.base.Polyhedron_base method), 389
subdivide() (sage.geometry.fan.RationalPolyhedralFan method), 283
sublattice() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 245
sublattice_complement() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 246
sublattice_quotient() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 247
subpolygons_of_polar_P1xP1() (in module sage.geometry.polyhedron.ppl_lattice_polygon), 173
subpolygons_of_polar_P2() (in module sage.geometry.polyhedron.ppl_lattice_polygon), 173
subpolygons_of_polar_P2_112() (in module sage.geometry.polyhedron.ppl_lattice_polygon), 173
support_contains() (sage.geometry.fan.RationalPolyhedralFan method), 283
swap_ineq_to_front() (sage.geometry.integral_points.InequalityCollection method), 502
symmetric_space() (sage.geometry.hyperplane_arrangement.hyperplane.AmbientVectorSpace method), 41
T
tetrahedron() (sage.geometry.polyhedron.library.Polytopes method), 72
tikz() (sage.geometry.polyhedron.plot.Projection method), 107
to_linear_program() (sage.geometry.polyhedron.base.Polyhedron_base method), 390
to_RationalPolyhedralFan() (sage.rings.polynomial.groebner_fan.PolyhedralFan method), 328
to_symmetric_space() (sage.geometry.hyperplane_arrangement.hyperplane.Hyperplane method), 45
toric_variety() (sage.geometry.fan.RationalPolyhedralFan method), 283
ToricLattice_ambient (class in sage.geometry.toric_lattice), 191
ToricLattice_generic (class in sage.geometry.toric_lattice), 193
ToricLattice_quotient (class in sage.geometry.toric_lattice), 196
ToricLattice_quotient_element (class in sage.geometry.toric_lattice), 200
ToricLattice_sublattice (class in sage.geometry.toric_lattice), 200
ToricLattice_sublattice_with_basis (class in sage.geometry.toric_lattice), 201
ToricLatticeFactory (class in sage.geometry.toric_lattice), 190
ToricPlotter (class in sage.geometry.toric_plotter), 308
translation() (sage.geometry.polyhedron.base.Polyhedron_base method), 391
transpositions() (sage.geometry.pseudolines.PseudolineArrangement method), 490
traverse_boundary() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 148
triangulate() (sage.geometry.polyhedron.base.Polyhedron_base method), 391
triangulate() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 447
triangulation() (sage.geometry.triangulation.point_configuration.Triangulation method), 447
triangulation_render_2d() (in module sage.geometry.triangulation.element), 463
triangulation_render_3d() (in module sage.geometry.triangulation.element), 463
triangulations() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 447
tropical_basis() (sage.rings.polynomial.groebner_fan.GroebnerFan method), 322
tropical_intersection() (sage.rings.polynomial.groebner_fan.GroebnerFan method), 322
TropicalPrevariety (class in sage.rings.polynomial.groebner_fan), 329
truncated_cube() (sage.geometry.polyhedron.library.Polytopes method), 73
truncated_dodecahedron() (sage.geometry.polyhedron.library.Polytopes method), 73
truncated_icosidodecahedron() (sage.geometry.polyhedron.library.Polytopes method), 74
truncated_octahedron() (sage.geometry.polyhedron.library.Polytopes method), 75
truncated_tetrahedron() (sage.geometry.polyhedron.library.Polytopes method), 75
truncation() (sage.geometry.polyhedron.base.Polyhedron_base method), 392
twenty_four_cell() (sage.geometry.polyhedron.library.Polytopes method), 76
type() (sage.geometry.polyhedron.representation.Equation method), 90
type() (sage.geometry.polyhedron.representation.Inequality method), 94
type() (sage.geometry.polyhedron.representation.Line method), 95
    type() (sage.geometry.polyhedron.representation.Ray method), 98
    type() (sage.geometry.polyhedron.representation.Vertex method), 99

U
    unbounded_regions() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 28
    union() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 29
    universe() (sage.geometry.polyhedron.parent.Polyhedra_base method), 89

V
    varchenko_matrix() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 29
    vector() (sage.geometry.polyhedron.representation.PolyhedronRepresentation method), 97
    verify() (sage.geometry.polyhedron.double_description.DoubleDescriptionPair method), 421
    verify() (sage.geometry.polyhedron.double_description_inhomogeneous.Hrep2Vrep method), 426
    verify() (sage.geometry.polyhedron.double_description_inhomogeneous.Vrep2Hrep method), 428
    Vertex (class in sage.geometry.polyhedron.representation), 98
    vertex() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 148
    vertex_adjacency_matrix() (sage.geometry.polyhedron.base.Polyhedron_base method), 393
    vertex_generator() (sage.geometry.polyhedron.base.Polyhedron_base method), 396
    vertex_generator() (sage.geometry.polyhedron.face.PolyhedronFace method), 115
    vertex_graph() (sage.geometry.fan.RationalPolyhedralFan method), 284
    vertex_graph() (sage.geometry.polyhedron.base.Polyhedron_base method), 396
    vertices() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 30
    vertices() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 148
    vertices() (sage.geometry.newton_polygon.NewtonPolygon_element method), 472
    vertices() (sage.geometry.polyhedron.base.Polyhedron_base method), 396
    vertices() (sage.geometry.polyhedron.face.PolyhedronFace method), 115
    vertices() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope_PPL_class method), 185
    vertices() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope_PPL_class method), 186
    vertices_matrix() (sage.geometry.polyhedron.base.Polyhedron_base method), 397
    vertices_saturating() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope_PPL_class method), 186
    verts_for_normal() (in module sage.rings.polynomial.groebner_fan), 331
    virtual_rays() (sage.geometry.fan.RationalPolyhedralFan method), 284
    volume() (sage.geometry.polyhedron.base.Polyhedron_base method), 397
    volume() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 448
    VoronoiDiagram (class in sage.geometry.voronoi_diagram), 491
    Vrep2Hrep (class in sage.geometry.polyhedron.double_description_inhomogeneous), 427
    Vrep_generator() (sage.geometry.polyhedron.base.Polyhedron_base method), 334
    Vrepresentation (class in sage.geometry.polyhedron.representation), 100
    Vrepresentation() (sage.geometry.polyhedron.base.Polyhedron_base method), 334
    Vrepresentation_space() (sage.geometry.polyhedron.base.Polyhedron_base method), 334
    Vrepresentation_space() (sage.geometry.polyhedron.parent.Polyhedra_base method), 87

W
    weight_vectors() (sage.rings.polynomial.groebner_fan.GroebnerFan method), 323
whitney_data() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 31
whitney_number() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 31
write_cdd_Hrepresentation() (sage.geometry.polyhedron.base.Polyhedron_base method), 400
write_cdd_Vrepresentation() (sage.geometry.polyhedron.base.Polyhedron_base method), 400
write_for_palp() (sage.geometry.point_collection.PointCollection method), 306
write_palp_matrix() (in module sage.geometry.lattice_polytope), 166

Z
Z_operators_gens() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 212
zero() (sage.geometry.polyhedron.parent.Polyhedra_base method), 89
zero_set() (sage.geometry.polyhedron.double_description.DoubleDescriptionPair method), 422
zero_sum_projection() (in module sage.geometry.polyhedron.library), 77
zonotope() (sage.geometry.polyhedron.library.Polytopes method), 76