<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Hyperplane arrangements</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Polyhedral computations</td>
<td>61</td>
</tr>
<tr>
<td>3</td>
<td>Triangulations</td>
<td>565</td>
</tr>
<tr>
<td>4</td>
<td>Miscellaneous</td>
<td>601</td>
</tr>
<tr>
<td>5</td>
<td>Helper functions</td>
<td>647</td>
</tr>
<tr>
<td>6</td>
<td>Indices and Tables</td>
<td>663</td>
</tr>
<tr>
<td></td>
<td>Python Module Index</td>
<td>665</td>
</tr>
<tr>
<td></td>
<td>Index</td>
<td>667</td>
</tr>
</tbody>
</table>
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 \(3x + 3y - 5z = 7\). To create it in Sage, you first have to create a `HyperplaneArrangements` object to define the variables \(x, y, z\):

```python
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:

```python
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:

```python
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:
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>
```

Number fields are also possible:

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

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:

```python
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)
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
```

1.1.2 Properties of Arrangements

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

```python
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:

```python
sage: a = hyperplane_arrangements.semiorder(3)
sage: b = a.essentialization(); b
Arrangement of 6 hyperplanes of dimension 2 and rank 2
sage: b.n_regions()
19
sage: b.regions()
(A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 6 vertices,
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
(continues on next page)
```
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 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,
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

\[
\text{sage: b.bounded_regions()}
\]
(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,
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

\[
\text{sage: b.n_bounded_regions()}
\]
7

\[
\text{sage: a.unbounded_regions()}
\]
(A 3-dimensional polyhedron in \(\mathbb{Q}^3\) defined as the convex hull of 1 vertex, 2 rays, \(L\)
A 3-dimensional polyhedron in \(\mathbb{Q}^3\) defined as the convex hull of 3 vertices, 1 ray, \(L\)
A 3-dimensional polyhedron in \(\mathbb{Q}^3\) defined as the convex hull of 1 vertex, 2 rays, \(L\)
A 3-dimensional polyhedron in \(\mathbb{Q}^3\) defined as the convex hull of 3 vertices, 1 ray, \(L\)
A 3-dimensional polyhedron in \(\mathbb{Q}^3\) defined as the convex hull of 1 vertex, 2 rays, \(L\)
A 3-dimensional polyhedron in \(\mathbb{Q}^3\) defined as the convex hull of 3 vertices, 1 ray, \(L\)
A 3-dimensional polyhedron in \(\mathbb{Q}^3\) defined as the convex hull of 1 vertex, 2 rays, \(L\)
A 3-dimensional polyhedron in \(\mathbb{Q}^3\) defined as the convex hull of 3 vertices, 1 ray, \(L\)
A 3-dimensional polyhedron in \(\mathbb{Q}^3\) defined as the convex hull of 1 vertex, 2 rays, \(L\)
A 3-dimensional polyhedron in \(\mathbb{Q}^3\) defined as the convex hull of 3 vertices, 1 ray, \(L\)
A 3-dimensional polyhedron in \(\mathbb{Q}^3\) defined as the convex hull of 1 vertex, 2 rays, \(L\)
A 3-dimensional polyhedron in \(\mathbb{Q}^3\) defined as the convex hull of 3 vertices, 1 ray, \(L\)
A 3-dimensional polyhedron in \(\mathbb{Q}^3\) defined as the convex hull of 1 vertex, 2 rays, \(L\)
A 3-dimensional polyhedron in \(\mathbb{Q}^3\) defined as the convex hull of 3 vertices, 1 ray, \(L\)
A 3-dimensional polyhedron in \(\mathbb{Q}^3\) defined as the convex hull of 1 vertex, 2 rays, \(L\)

The distance between regions is defined as the number of hyperplanes separating them. For example:
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.distanceEnumerator(r1)  
# generating function for distances from r1
6*x^3 + 6*x^2 + 6*x + 1

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

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

sage: a = hyperplane_arrangements.braid(3)
sage: p = a.intersectionPoset()
sage: p.isRanked()  
True
sage: p.orderPolytope()
A 5-dimensional polyhedron in ZZ^5 defined as the convex hull of 10 vertices

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

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

where the sum is \( P \) is the intersectionPoset() of the arrangement and \( \mu \) is the Möbius function of \( P \):

sage: a = hyperplane_arrangements.semiorder(5)
sage: a.characteristicPolynomial()  
# long time (about a second on Core i7)
x^5 - 20*x^4 + 180*x^3 - 790*x^2 + 1380*x
sage: a.poincarePolynomial()  
# long time
1380*x^4 + 790*x^3 + 180*x^2 + 20*x + 1
sage: a.nRegions()  
# long time
2371
sage: charpoly = a.characteristicPolynomial()  
# long time
sage: charpoly(-1)  
# long time
-2371
sage: a.nBoundedRegions()  
# long time
751
sage: charpoly(1)  
# long time
751

For finer invariants derived from the intersection poset, see whitneyNumber() and doublyIndexedWhitneyNumber().

Miscellaneous methods (see documentation for an explanation):

sage: a = hyperplane_arrangements.semiorder(3)
sage: a.hasGoodReduction(5)  
True

(continues on next page)
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.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()
\[
\begin{bmatrix}
1 & h2 & h2*h4 & h2*h3 & h2*h3*h4 & h2*h3*h4*h5 \\
h2 & 1 & h4 & h3 & h3*h4 & h3*h4*h5 \\
h2*h4 & h4 & 1 & h3*h4 & h3 & h3*h5 \\
h2*h3 & h3 & h3*h4 & 1 & h4 & h4*h5 \\
h2*h3*h4 & h3*h4 & h3 & h4 & 1 & h5 \\
h2*h3*h4*h5 & h3*h4*h5 & h3*h5 & h4*h5 & h5 & 1 \\
\end{bmatrix}
\]

There are extensive methods for visualizing hyperplane arrangements in low dimensions. See \texttt{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.

class \texttt{sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement}(parent, hyperplanes, check=True, backend=None)

Bases: \texttt{sage.structure.element.Element}

A hyperplane arrangement.

\textbf{Warning:} You should never create \texttt{HyperplaneArrangementElement} instances directly, always use the parent.

\texttt{add_hyperplane}(other)

The union of self with other.

INPUT:
• other – a hyperplane arrangement or something that can be converted into a hyperplane arrangement

OUTPUT:
A new hyperplane arrangement.

EXAMPLES:

```sage
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
sage: A | B  # syntactic sugar
Arrangement of 8 hyperplanes of dimension 2 and rank 2
```

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

```sage
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
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
```

```backen```

Return the backend used for polyhedral objects

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

EXAMPLES:

By default, no backend is specified:

```sage
sage: H = HyperplaneArrangements(QQ)
sage: A = H()
sage: A.backend() None
```

Otherwise, one may specify a polyhedral backend:

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

```bounded_regions```

Return the relatively bounded regions of the arrangement.

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

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

See also:

unbounded_regions()

EXAMPLES:

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

center()

Return the center of the hyperplane arrangement.

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

OUTPUT:

A polyhedron.

EXAMPLES:

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

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

The Shi arrangement in dimension 3 has an empty center:

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

The Braid arrangement in dimension 3 has a center that is neither empty nor full-dimensional:
A 1-dimensional polyhedron in QQ^3 defined as the convex hull of 1 vertex and 1 line.
is a closed face of $\mathcal{A}$, then the sign vector of $G$ is defined to be the vector $(v_1, v_2, \ldots, v_N) \in \{-1, 0, 1\}^N$ where $x$ is any point in the relative interior of $G$ and where, for each $i$, the number $v_i$ is the sign of $u_i(x)$. (This does not depend on the choice of $x$.)

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

**INPUT:**

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

**OUTPUT:**

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

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

**EXAMPLES:**

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

(continues on next page)
sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: a = H(x, y+1)
sage: a.hyperplanes()
(Hyperplane 0*x + y + 1, Hyperplane x + 0*y + 0)
sage: [(v, F, F.representative_point()) for v, F in a.closed_faces()]
[((0, 0), A 0-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex, (0, -1)), ((0, 1), A 1-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and → 1 ray, (1, -1)), ((0, -1), A 1-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and → 1 ray, (-1, -1)), ((1, 0), A 1-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and → 1 ray, (0, 0)), ((1, 1), A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and → 2 rays, (1, 0)), ((1, -1), A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and → 2 rays, (-1, 0)), ((-1, 0), A 1-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and → 1 ray, (0, -2)), ((-1, 1), A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and → 2 rays, (1, -2)), ((-1, -1), A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and → 2 rays, (-1, -2))]
sage: a = hyperplane_arrangements.braid(3)
sage: a.hyperplanes()
(Hyperplane 0*t0 + t1 - t2 + 0, Hyperplane t0 - t1 + 0*t2 + 0, Hyperplane t0 + 0*t1 - t2 + 0)
sage: [(v, F, F.representative_point()) for v, F in a.closed_faces()]
[((0, 0, 0), A 1-dimensional polyhedron in QQ^3 defined as the convex hull of 1 vertex and → 1 line,
Let us check that the number of closed faces with a given dimension computed using `self.closed_faces()` equals the one computed using `face_vector()`:
sage: def test_number(a):
....:     Qx = PolynomialRing(QQ, 'x'); x = Qx.gen()
....:     RHS = Qx.sum(vi * x ** i for i, vi in enumerate(a.face_vector()))
....:     LHS = Qx.sum(x ** F[1].dim() for F in a.closed_faces())
....:     return LHS == RHS
sage: a = hyperplane_arrangements.Catalan(2)
sage: test_number(a)
True
sage: a = hyperplane_arrangements.Shi(3)
sage: test_number(a)  # long time
True

cone(variable='t')
Return the cone over the hyperplane arrangement.

INPUT:

• 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 self and those of self.cone(), there is no guarantee that the order in which they appear in self.hyperplanes() will match the order in which their counterparts in self.cone().hyperplanes() will appear in self.cone().hyperplanes()!

EXAMPLES:

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:

```python
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:

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

derivation_module_basis(algorithm='singular')

Return a basis for the derivation module of \( \text{self} \) if one exists, otherwise return None.

See also:

derivation_module_free_chain(), is_free()

INPUT:

• \( \text{algorithm} \) – (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:**

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

**derivation_module_free_chain()**

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

**See also:**

`is_free()`

**EXAMPLES:**

```python
sage: W = WeylGroup(['A',3], prefix='s')
sage: A = W.long_element().inversion_arrangement()
sage: for M in A.derivation_module_free_chain(): print(M)
[ 1 0 0]
[ 0 1 0]
[ 0 0 a3]
[ 1 0 0]
[ 0 0 1]
[ a2 0]
[ 1 0 0]
[ 0 -1 -1]
[ a2 -a3]
[ 0 1 0]
[ 0 0 1]
[a1 0 0]
[ 1 0 -1]
[a3 -1 0]
[a1 0 a2]
[ 1 0 0]
[a3 -1 -1]
[ 0 a1 -a2 - a3]
```

1.1. Hyperplane Arrangements
dimension()  
Return the ambient space dimension of the arrangement.  

OUTPUT:  
An integer.  

EXAMPLES:

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

distance_between_regions(region1, region2)  
Return the number of hyperplanes separating the two regions.  

INPUT:  
• region1, region2 – regions of the arrangement or representative points of regions  

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

EXAMPLES:

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

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 \( \text{base\_region} \), i.e., the number of hyperplanes separated by \( i \) hyperplanes from \( \text{base\_region} \).  

EXAMPLES:

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

doubly_indexed_whitney_number(i, j, kind=1)  
Return the \( i, j \)-th doubly-indexed Whitney number.  

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

If \( \text{kind} = 2 \), 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
• \(\text{kind}\) – (default: 1) 1 or 2

OUTPUT:
Integer. The \((i,j)\)-th entry of the \(\text{kind}\) Whitney number.

See also:
\texttt{whitney\_number()}, \texttt{whitney\_data()}

EXAMPLES:

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

REFERENCES:
• [GZ1983]

\texttt{essentialization()}

Return the essentialization of the hyperplane arrangement.

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

OUTPUT:
The essentialization as a new hyperplane arrangement.

EXAMPLES:

```python
sage: a = hyperplane_arrangements.braid(3)
sage: a.is_essential()
False
sage: a.essentialization()
Arrangement \(<t1 - t2 | t1 + 2*t2 | 2*t1 + t2>
```

```python
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>
```

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

(continues on next page)
sage: h = hyperplane_arrangements.semiorder(4)
sage: h.essentialization()
Arrangement of 12 hyperplanes of dimension 3 and rank 3

face_product(F, G, normalize=True)

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

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

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 $xyx = xy$). When the arrangement is central, then this semigroup is a monoid. See [Br2000] (Appendix A in particular) for further properties.

INPUT:

* F, G – two faces of self (as polyhedra)
* normalize – Boolean (default: True); if True, then this method returns the precise instance of $FG$ in the list returned by self.closed_faces(), rather than creating a new instance

EXAMPLES:

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

```python
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
```

(continues on next page)
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: 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 hyperplanes in the arrangement as a tuple.
OUTPUT:

An integer.

EXAMPLES:

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

**intersection_poset**(element_label='int')

Return the intersection poset of the hyperplane arrangement.

INPUT:

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

OUTPUT:

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

EXAMPLES:

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

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

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

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

1.1. Hyperplane Arrangements
```python
sage: A = hyperplane_arrangements.semiorder(3)
sage: L = A.intersection_poset(element_label='subset')
sage: [sorted(level, key=sorted) for level in L.level_sets()]
[[{}],
 [\{0\}, \{1\}, \{2\}, \{3\}, \{4\}, \{5\}],
 [\{0, 2\}, \{0, 3\}, \{0, 4\}, \{0, 5\}, \{1, 2\}, \{1, 3\}, \{1, 4\}, \{1, 5\}, \{2, 4\}, \{2, 5\} \rightarrow, \{3, 4\}, \{3, 5\}]]
```

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

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

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

**is_central** *(certificate=False)*

Test whether the intersection of all the hyperplanes is nonempty.

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

**INPUT:**

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

**OUTPUT:**

If **certificate** is `True`, returns a tuple containing:

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

If **certificate** is `False`, returns a boolean.
EXAMPLES:

```sage
sage: a = hyperplane_arrangements.braid(2)
sage: a.is_central()
True
```

The Catalan arrangement in dimension 3 is not central:

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

The empty arrangement in dimension 5 is central:

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

`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:

```sage
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:

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

**singular**

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

**BC**

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

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

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

EXAMPLES:

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

```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
```

(continues on next page)
False
sage: c.is_central()
True

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

Test whether the hyperplane separates the given regions.

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

**OUTPUT:**
A boolean. Whether the 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)
True
sage: r = A.region_containing_point([1,1])
sage: s = A.region_containing_point([-1,1])
sage: A.is_separating_hyperplane(r, s, x)
True
```

**is_simplicial()**

Test whether the arrangement is simplicial.

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

**OUTPUT:**
A boolean whether the hyperplane arrangement is simplicial.

**EXAMPLES:**

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

**matroid()**

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

**EXAMPLES:**

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

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

```sage
f = sum([list(M.flats(i)) for i in range(M.rank()+1)], [])
PF = Poset([f, lambda x,y: x < y])
P.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]:

```sage
P.<x,y,z> = HyperplaneArrangements(QQ)
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)
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)
A.minimal_generated_number()
3
B.minimal_generated_number()
4
```

**n_bounded_regions()**
Return the number of (relatively) bounded regions.

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

**EXAMPLES:**

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

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

**OUTPUT:**
An integer.
EXAMPLES:

```python
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:

```python
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:

```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)
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
```

```

orlik_terao_algebra(base_ring=None, ordering=None)
Return the Orlik-Terao algebra of self.

INPUT:

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

EXAMPLES:

```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)
sage: A.orlik_terao_algebra()
Orlik-Terao algebra of Linear matroid of rank 3 on 7 elements
```

(continues on next page)
represented over the Rational Field over Rational Field

```
sage: A.orlik_terao_algebra(base_ring=QQ['t'])
```

Orlik-Terao algebra of Linear matroid of rank 3 on 7 elements represented over the Rational Field over Univariate Polynomial Ring in t over Rational Field

```
sage: 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

```
sage: 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
```

```
sage: poset_of_regions(B=None, numbered_labels=True)
```

Return the poset of regions for a central hyperplane arrangement.

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

**INPUT:**

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

**OUTPUT:**

A Poset object containing the poset of regions.

**EXAMPLES:**

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

```
sage: A = hyperplane_arrangements.braid(3)
sage: A.poset_of_regions()
```

(continues on next page)
Finite poset containing 6 elements
`sage: A.poset_of_regions(numbered_labels=False)`
Finite poset containing 6 elements
`sage: A = hyperplane_arrangements.braid(4)`
`sage: A.poset_of_regions()`
Finite poset containing 24 elements
`sage: H.<x,y,z> = HyperplaneArrangements(QQ)`
`sage: A = H([[0,1,1,1],[0,1,2,3],[0,1,3,2],[0,2,1,3]])`  
`sage: R = A.regions()`  
`sage: base_region = R[3]`  
`sage: A.poset_of_regions(B=base_region)`
Finite poset containing 14 elements

**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

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

**region_containing_point(p)**
The region in the hyperplane arrangement containing a given point.
The base field must have characteristic zero.

**INPUT:**

* p – point

**OUTPUT:**

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

```python
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: 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
```

Example 6 of [KP2020]:

```python
sage: from itertools import product
sage: def zero_one(d):
    ....:     for x in product([0,1], repeat=d):
    ....:         if any(y for y in x):
(continues on next page)```
sage: K.<x,y> = HyperplaneArrangements(QQ)
sage: A = K(*zero_one(2))
sage: len(A.regions())
6
sage: K.<x,y,z> = HyperplaneArrangements(QQ)
sage: A = K(*zero_one(3))
sage: len(A.regions())
32
sage: K.<x,y,z,w> = HyperplaneArrangements(QQ)
sage: A = K(*zero_one(4))
sage: len(A.regions())
370
sage: K.<x,y,z,w,r> = HyperplaneArrangements(QQ)
sage: A = K(*zero_one(5))
sage: len(A.regions())
# not tested (~25s)
11292

It is possible to specify the backend:

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

**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], [0,1]]); 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 QQ^2 defined as the convex hull of 3 vertices, and 1 ray,
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
```

```
sage: A | B  # syntactic sugar
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:

```sage
a = hyperplane_arrangements.coordinate(3)
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:

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

(continues on next page)
sage: len(chessboard.vertices())
81
sage: chessboard.vertices(exclude_sandwiched=True)
((0, 0), (0, 8), (8, 0), (8, 8))

whitney_data()
Return the Whitney numbers.

See also:
whitney_number(), doubly_indexed_whitney_number()

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

EXAMPLES:

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

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 $\text{rank}(x) = k$.

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

See [GZ1983] for more details.

INPUT:
• $k$ – integer
• $kind$ – 1 or 2 (default: 1)

OUTPUT:
Integer. The $k$-th Whitney number.

See also:
doubly_indexed_whitney_number() whitney_data()

EXAMPLES:

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

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:

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:

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:**

```python
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:**

```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.gen(0)
Hyperplane x + 0*y + 0*z + 0
```

```python
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:

```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
```

### 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
class sage.geometry.hyperplane_arrangement.library.HyperplaneArrangementLibrary
    Bases: object

    The library of hyperplane arrangements.

    Catalan(n, K=Rational Field, names=None)
    Return the Catalan arrangement.

    INPUT:
    • n – integer
    • K – field (default: Q)
    • names – tuple of strings or None (default); the variable names for the ambient space

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

    EXAMPLES:

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

    G_Shi(G, K=Rational Field, names=None)
    Return the Shi hyperplane arrangement of a graph \(G\).

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

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

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

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: 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

1.2. Library of Hyperplane Arrangements
REFERENCES:

• [AR2012]

\textbf{Shi}(data, K=\texttt{Rational Field}, names=None, m=1)

Return the Shi arrangement.

INPUT:

• data – either an integer or a Cartan type (or coercible into; see “CartanType”)
• K – field (default: \texttt{QQ})
• names – tuple of strings or \texttt{None} (default); the variable names for the ambient space
• m – integer (default: 1)

OUTPUT:

• If data is an integer \( n \), return the Shi arrangement in dimension \( n \), i.e. the set of \( n(n-1) \) hyperplanes:
  \[
  \{ x_i - x_j = 0, 1 : 1 \leq i \leq j \leq n \}.
  \]
  This corresponds to the Shi arrangement of Cartan type \( A_{n-1} \).

• If data is a Cartan type, return the Shi arrangement of given type.

• If \( m > 1 \), return the \( m \)-extended Shi arrangement of given type.

The \( m \)-extended Shi arrangement of a given crystallographic Cartan type is defined by the inner product
\[
\langle a, x \rangle = k \quad \text{for} \quad -m < k \leq m \quad \text{and} \quad a \in \Phi^+ \text{ is a positive root of the root system } \Phi.
\]

EXAMPLES:

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

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

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

The characteristic polynomial is pre-computed using the results of [Ath1996]:
sage: hyperplane_arrangements.Shi("A3").characteristic_polynomial()
x^4 - 12*x^3 + 48*x^2 - 64*x
sage: hyperplane_arrangements.Shi("A3",m=2).characteristic_polynomial()
x^4 - 24*x^3 + 192*x^2 - 512*x
sage: hyperplane_arrangements.Shi("C3").characteristic_polynomial()
x^3 - 18*x^2 + 108*x - 216
sage: hyperplane_arrangements.Shi("E6").characteristic_polynomial()
x^8 - 72*x^7 + 2160*x^6 - 34560*x^5 + 311040*x^4 - 1492992*x^3 + 2985984*x^2 + 2985984*x - 168
sage: hyperplane_arrangements.Shi("B4",m=3).characteristic_polynomial()
x^4 - 96*x^3 + 3456*x^2 - 55296*x + 331776

**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:**

```python
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
sage: hyperplane_arrangements.bigraphical(G, 'generic').n_regions()
65
sage: hyperplane_arrangements.bigraphical(G).n_regions()
59
```

**REFERENCES:**

- [HP2016]

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

The braid arrangement.

**INPUT:**

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

---

1.2. Library of Hyperplane Arrangements
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
OUTPUT:
The linial hyperplane arrangement is the set of hyperplanes \( \{ x_i - x_j = 1 : 1 \leq i < j \leq n \} \).

EXAMPLES:

```sage
a = hyperplane_arrangements.linial(4); a
Arrangement of 6 hyperplanes of dimension 4 and rank 3
```

```
sage: a.characteristic_polynomial()
x^4 - 6*x^3 + 15*x^2 - 14*x
```

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

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

EXAMPLES:

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

`sage.geometry.hyperplane_arrangement.library.make_parent` \((\text{base_ring}, \text{dimension}, \text{names}=\text{None})\)

Construct the parent for the hyperplane arrangements.

For internal use only.

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

OUTPUT:
A new `HyperplaneArrangements` instance.

EXAMPLES:

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

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

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

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

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

```sage
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
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
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
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'
```

```python

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
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:

```
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:

```
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)*

Bases: `sage.geometry.linear_expression.LinearExpression`

A hyperplane.

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

INPUT:

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

EXAMPLES:

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

sage: ambient = H.ambient_space()
sage: ambient._element_constructor_(x+y-1)
Hyperplane x + y - 1
```
For technical reasons, we must allow the degenerate cases of an empty space and of a full space:

```sage
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:**

```sage
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:**

```sage
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:**

```sage
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
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
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:**

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

**plot(****kwds**)**

Plot the hyperplane.

**OUTPUT:**

A graphics object.

**EXAMPLES:**

```python
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:**

```python
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
```

(continues on next page)
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:

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

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:

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

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

EXAMPLES:

```sage
sage: L.<x, y> = HyperplaneArrangements(QQ)
sage: h = -1/3*x + 1/2*y
call_method 'to_symmetric_space'
sage: h.to_symmetric_space()
-1/3*x + 1/2*y
call_method 'to_symmetric_space'
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
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
    AffineSubspace
sage: a = AffineSubspace([1,0,0,0], VectorSpace(QQ,4))
sage: a
Affine space p + W where:
  p = (1, 0, 0, 0)
  W = Vector space of dimension 4 over Rational Field

dimension()

Return the dimension of the affine space.

OUTPUT:
An integer.

EXAMPLES:

sage: from sage.geometry.hyperplane_arrangement.affine_subspace import AffineSubspace
    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 = Vector space of degree 3 and dimension 1 over Rational Field
  Basis matrix:
  [0 1 0]

sage: C = AffineSubspace((0,0,1), U)
sage: A.intersection(C)

sage: C = AffineSubspace((7,8,9), U.complement())
sage: A.intersection(C)

Affine space p + W where:
  p = (7, 8, 0)
  W = Vector space of degree 3 and dimension 0 over Rational Field
  Basis matrix:
  []

sage: A.intersection(C).intersection(B)

 sage: D = AffineSubspace([1,2,3], VectorSpace(GF(5),3))
sage: E = AffineSubspace([3,4,5], VectorSpace(GF(5),3))
sage: D.intersection(E)

Affine space p + W where:
  p = (3, 4, 0)
  W = Vector space of dimension 3 over Finite Field of size 5

linear_part()

Return the linear part of the affine space.

OUTPUT:

A vector subspace of the ambient space.

EXAMPLES:

sage: from sage.geometry.hyperplane_arrangement.affine_subspace import AffineSubspace

sage: A = AffineSubspace([2,3,1], matrix(QQ, [[1,2,3]]).right_kernel())
sage: A.linear_part()

Vector space of degree 3 and dimension 2 over Rational Field
  Basis matrix:
  [ 1 0 -1/3]
  [ 0 1 -2/3]

sage: A.linear_part().ambient_vector_space()

point()

Return a point p in the affine space.

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

EXAMPLES:

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

**1.5 Plotting of Hyperplane Arrangements**

**PLOT OPTIONS:**

Besides 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 by which labels are offset from h.point() for each hyperplane h. The format is different for each dimension: if the hyperplanes have dimension 0, the offset can be a single number or a list of numbers, one for each hyperplane; if the hyperplanes have dimension 1, the offset can be a single 2-tuple, or a list of 2-tuples, one for each hyperplane; if the hyperplanes have dimension 2, the offset can be a single 3-tuple or a list of 3-tuples, one for each hyperplane. (Defaults: 0-dim: 0.1, 1-dim: (0,1), 2-dim: (0,0,0.2)).
- **hyperplane_legend** – Boolean, 'short', 'long' (default: 'long'; in 3-d: False). If False, no legend is shown; if True, 'short', or 'long', the legend is shown with the default, long, or short labeling, respectively. (For arrangements of lines or planes, only.)
- **hyperplane_opacities** – A number or list of numbers, one for each hyperplane, between 0 and 1. Only applies to 3d plots.
- **point_sizes** – Number or list of numbers, one for each hyperplane giving the sizes of points in a zero-dimensional arrangement (default: 50).
- **ranges** – Range for the parameters or a list of ranges of parameters, one for each hyperplane, for the parametric plots of the hyperplanes. If a single positive number $r$ is given for ranges, then all parameters run from $-r$ to $r$. Otherwise, for a line in the plane, the range has the form $[a,b]$ (default: $[-3,3]$), and for a plane in 3-space, the range has the form $[[a,b],[c,d]]$ (default: $[[-3,3],[-3,3]]$). The ranges are centered around `hyperplane_arrangement.point()`.

EXAMPLES:

```
sage: H3.<x,y,z> = HyperplaneArrangements(QQ)
sage: A = H3([(1,0,0), 0], [(0,0,1), 5])
sage: A.plot(hyperplane_opacities=0.5, hyperplane_labels=True, hyperplane_legend=False)
Graphics3d Object
sage: c = H3([(1,0,0),0], [(0,0,1),5])
sage: c.plot(ranges=10)
```

(continues on next page)
Graphics3d Object
\[
\text{ sage: } c\text{.plot(ranges=}[[9.5,10], [-3,3]])
\]
Graphics3d Object
\[
\text{ sage: } c\text{.plot(ranges=}[[[9.5,10], [-3,3]], [[-6,6], [-5,5]]])
\]

\[
\text{ sage: } H_{2}\langle s,t\rangle = \text{ HyperplaneArrangements}(\mathbb{Q})
\]
\[
\text{ sage: } h = H_{2}([(1,1),0], [(1,-1),0], [(0,1),2])
\]
\[
\text{ sage: } h\text{.plot(ranges=}[20])
\]
\[
\text{ Graphics object consisting of 3 graphics primitives}
\]
\[
\text{ sage: } h\text{.plot(ranges=}[[-1, 10]])
\]
\[
\text{ Graphics object consisting of 3 graphics primitives}
\]
\[
\text{ sage: } h\text{.plot(ranges=}[[-1, 1], [-5, 5], [-1, 10]])
\]
\[
\text{ Graphics object consisting of 3 graphics primitives}
\]
\[
\text{ sage: } a = \text{ hyperplane_arrangements.coordinate}(3)
\]
\[
\text{ sage: } opts = \{\text{hyperplane_colors}:[\text{'yellow'}, \text{'green'}, \text{'blue'}]\}
\]
\[
\text{ sage: } opts[\text{hyperplane_labels}] = \text{True}
\]
\[
\text{ sage: } opts[\text{label_offsets}] = [(0,2,2), (2,0,2), (2,2,0)]
\]
\[
\text{ sage: } opts[\text{hyperplane_legend}] = \text{False}
\]
\[
\text{ sage: } opts[\text{hyperplane_opacities}] = 0.7
\]
\[
\text{ sage: } a\text{.plot(**opts)}
\]
\[
\text{ Graphics3d Object}
\]
\[
\text{ sage: } opts[\text{hyperplane_labels}] = \text{'short'}
\]
\[
\text{ sage: } a\text{.plot(**opts)}
\]
\[
\text{ Graphics3d Object}
\]
\[
\text{ sage: } H_{.}\langle u\rangle = \text{ HyperplaneArrangements}(\mathbb{Q})
\]
\[
\text{ sage: } pts = H(3u+4, 2u+5, 7u+1)
\]
\[
\text{ sage: } pts\text{.plot(hyperplane_colors=}[\text{'yellow'}, \text{'black'}, \text{'blue'}])
\]
\[
\text{ Graphics object consisting of 3 graphics primitives}
\]
\[
\text{ sage: } pts\text{.plot(point_sizes=}[50,100,200], hyperplane_colors=\text{'blue'})
\]
\[
\text{ Graphics object consisting of 3 graphics primitives}
\]
\[
\text{ sage: } H_{.}\langle x,y,z\rangle = \text{ HyperplaneArrangements}(\mathbb{Q})
\]
\[
\text{ sage: } a = H(x, y+1, y+2)
\]
\[
\text{ sage: } a\text{.plot(hyperplane_labels=}\text{True}, label_colors=\text{'blue'}, label_fontsize=18)
\]
\[
\text{ Graphics3d Object}
\]
\[
\text{ sage: } a\text{.plot(hyperplane_labels=}\text{True}, label_colors=}[\text{'red'}, \text{'green'}, \text{'black'})
\]
\[
\text{ Graphics3d Object}
\]

\[
\text{ sage.geometry.hyperplane_arrangement.plot.legend_3d(hyperplane_arrangement, hyperplane_colors, length)}
\]
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:

```python
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(c, list(colors.values())[:12], length='long')
Graphics object consisting of 12 graphics primitives

sage: legend_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.

**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: $[[-3,3],[-3,3]]$). (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

sage: H2.<x,y> = HyperplaneArrangements(QQ)
sage: b = 3*x + 4*y + 5
sage: b.plot()  # indirect doctest
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()  # indirect doctest
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=[[0,1]], label_offset=(2,2,2), 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. 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`
- `bitruncated_six_hundred_cell`
- `buckyball`
- `cantellated_one_hundred_twenty_cell`
- `cantellated_six_hundred_cell`
- `cantitruncated_one_hundred_twenty_cell`
- `cantitruncated_six_hundred_cell`
- `cross_polytope`
- `cube`
- `cuboctahedron`
- `cyclic_polytope`
- `dodecahedron`
- `flow_polytope`
- `Gosset_3_21`
- `grand_antiprism`
- `great_rhombicuboctahedron`
- `hypercube`
- `hypersimplex`
- `icosahedron`
- `icosidodecahedron`
- `Kirkman_icosahedron`
- `octahedron`
- `omnitruncated_one_hundred_twenty_cell`
- `omnitruncated_six_hundred_cell`
- `one_hundred_twenty_cell`

continues on next page
Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>parallelootope()</td>
</tr>
<tr>
<td>pentakis_dodecahedron()</td>
</tr>
<tr>
<td>permutahedron()</td>
</tr>
<tr>
<td>generalized_permutahedron()</td>
</tr>
<tr>
<td>rectified_one_hundred_twenty_cell()</td>
</tr>
<tr>
<td>rectified_six_hundred_cell()</td>
</tr>
<tr>
<td>regular_polygon()</td>
</tr>
<tr>
<td>rhombic_dodecahedron()</td>
</tr>
<tr>
<td>rhombicosidodecahedron()</td>
</tr>
<tr>
<td>runcinated_one_hundred_twenty_cell()</td>
</tr>
<tr>
<td>runcitruncated_one_hundred_twenty_cell()</td>
</tr>
<tr>
<td>runcitruncated_six_hundred_cell()</td>
</tr>
<tr>
<td>simplex()</td>
</tr>
<tr>
<td>six_hundred_cell()</td>
</tr>
<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>truncated_one_hundred_twenty_cell()</td>
</tr>
<tr>
<td>truncated_six_hundred_cell()</td>
</tr>
<tr>
<td>twenty_four_cell()</td>
</tr>
</tbody>
</table>

class sage.geometry.polyhedron.library.Polytopes

Bases: object

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

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

Return the Birkhoff polytope with \(n\) vertices.

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

**INPUT:**

- n – 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: b4 = polytopes.Birkhoff_polytope(4)

Gosset_3_21(backend=None)
Return the Gosset 3_21 polytope.

The Gosset 3_21 polytope is a uniform 7-polytope. It has 56 vertices, and 702 facets: 126 3_11 and 576 6-simplex. For more information, see the Wikipedia article 3_21_polytope.

INPUT:

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

EXAMPLES:

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

Kirkman_icosahedron(backend=None)
Return the Kirkman icosahedron.

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

INPUT:

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

EXAMPLES:

sage: ki = polytopes.Kirkman_icosahedron()
sage: ki.n_vertices(), ki.ambient_dim(), ki.dim()
(24, 16, 9)
static associahedron(cartan_type, backend='ppl')
    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.

    INPUT:
    • cartan_type – a cartan type according to sage.combinat.root_system.cartan_type.
    CartanTypeFactory
    • backend – string ('ppl'); the backend to use; see sage.geometry.polyhedron.constructor.
    Polyhedron()

    EXAMPLES:

    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()
    (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

    bitruncated_six_hundred_cell(exact=True, backend=None)
    Return the bitruncated 600-cell.

    The bitruncated 600-cell is a 4-dimensional 4-uniform polytope in the $H_4$ family. It has 3600 vertices. For
    more information see Wikipedia article Bitruncated 600-cell.
Warning: The coordinates are exact by default. The computation with inexact coordinates (using the backend "cdd") returns a numerical inconsistency error, and thus cannot be computed.

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

EXAMPLES:

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

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

Return the bucky ball.

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

See also:
`icosahedron()`

INPUT:
- exact – (boolean, default True) If False use an approximate ring for the coordinates.
- base_ring – the ring in which the coordinates will belong to. If it is not provided and exact=True it will be a the number field \( \mathbb{Q}[\phi] \) 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: 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 with sqrt5 = 2.
→236067977499790?
```

A much faster implementation using floating point approximations:

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

Its facets are 5 regular pentagons and 6 regular hexagons:

```python
sage: sum(1 for f in bb.facets() if len(f.vertices()) == 5)
12
sage: sum(1 for f in bb.facets() if len(f.vertices()) == 6)
20
```
\texttt{cantellated\_one\_hundred\_twenty\_cell}(\texttt{exact=True, backend=None})

Return the cantellated 120-cell.

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

\textbf{Warning:} The coordinates are exact by default. The computation with inexact coordinates (using the backend 'cdd') returns a numerical inconsistency error, and thus cannot be computed.

INPUT:
- **exact** - (boolean, default True) if True use exact coordinates instead of floating point approximations.
- **backend** – the backend to use to create the polytope.

EXAMPLES:
\begin{verbatim}
sage: polytopes.cantellated_one_hundred_twenty_cell(backend='normaliz') # not tested - long time
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 3600 vertices
\end{verbatim}

\texttt{cantellated\_six\_hundred\_cell}(\texttt{exact=False, backend=None})

Return the cantellated 600-cell.

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

\textbf{Warning:} The coordinates are inexact by default. The computation with inexact coordinates (using the backend 'cdd') issues a UserWarning on inconsistencies.

INPUT:
- **exact** - (boolean, default False) if True use exact coordinates instead of floating point approximations.
- **backend** – the backend to use to create the polytope.

EXAMPLES:
\begin{verbatim}
sage: polytopes.cantellated_six_hundred_cell() # not tested - very long time
doctest:warning ...
UserWarning: This polyhedron data is numerically complicated; cdd could not convert between the inexact V and H representation without loss of data. The resulting object might show inconsistencies.
A 4-dimensional polyhedron in RDF^4 defined as the convex hull of 3600 vertices
\end{verbatim}

It is possible to use the backend 'normaliz' to get an exact representation:
\begin{verbatim}
sage: polytopes.cantellated_six_hundred_cell(exact=True, backend='normaliz') # not tested - long time
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 3600 vertices
\end{verbatim}

\texttt{cantitruncated\_one\_hundred\_twenty\_cell}(\texttt{exact=True, backend=None})

Return the cantitruncated 120-cell.
The cantitruncated 120-cell is a 4-dimensional 4-uniform polytope in the $H_4$ family. It has 7200 vertices. For more information see Wikipedia article Cantitruncated 120-cell.

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

**INPUT:**
- exact - (boolean, default True) if True use exact coordinates instead of floating point approximations.
- backend – the backend to use to create the polytope.

**EXAMPLES:**

```sage
cpolytopes.cantitruncated_one_hundred_twenty_cell(exact=True, backend='normaliz')
```

A 4-dimensional polyhedron in AA^4 defined as the convex hull of 7200 vertices

`sage: cantitruncated_six_hundred_cell(exact=True, backend='normaliz')`

Return the cantitruncated 600-cell.

The cantitruncated 600-cell is a 4-dimensional 4-uniform polytope in the $H_4$ family. It has 7200 vertices. For more information see Wikipedia article Cantitruncated 600-cell.

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

**INPUT:**
- exact - (boolean, default True) if True use exact coordinates instead of floating point approximations.
- backend – the backend to use to create the polytope.

**EXAMPLES:**

```sage
cpolytopes.cantitruncated_six_hundred_cell(exact=True, backend='normaliz')
```

A 4-dimensional polyhedron in AA^4 defined as the convex hull of 7200 vertices

`sage: cross_polytope(4, backend='normaliz')`

Return a cross-polytope in dimension dim.

A cross-polytope is a higher dimensional generalization of the octahedron. It is the convex hull of the $2d$ points $(\pm 1, 0, \ldots, 0), (0, \pm 1, \ldots, 0), \ldots, (0, 0, \ldots, \pm 1)$. See the Wikipedia article Cross-polytope for more information.

**INPUT:**
- dim – integer. The dimension of the cross-polytope.
- backend – the backend to use to create the polytope.

**EXAMPLES:**

```sage:
sage: cross_polytope(4)
sage: cross_polytope.f_vector()
```

(continues on next page)
sage: four_cross.is_simple()
False

cube(intervals=None, backend=None)

Return the cube.

The cube is the Platonic solid that is obtained as the convex hull of the eight \pm 1 vectors of length 3 (by default). Alternatively, the cube is the product of three intervals from \texttt{intervals}.

\textbf{See also:}

\texttt{hypercube()}

\textbf{INPUT:}

- \texttt{intervals} – list (default=None). It takes the following possible inputs:
  - If the input is \texttt{None} (the default), returns the convex hull of the eight \pm 1 vectors of length three.
  - \texttt{'zero_one'} – (string). Return the 0/1-cube.
  - a list of 3 lists of length 2. The cube will be a product of these three intervals.

- \texttt{backend} – the backend to use to create the polytope.

\textbf{OUTPUT:}

A cube as a polyhedron object.

\textbf{EXAMPLES:}

Return the \pm 1-cube:

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

Return the 0/1-cube:

\begin{verbatim}
sage: cc = polytopes.cube(intervals = 'zero_one')
sage: cc.vertices_list()
[[1, 0, 0],
 [1, 1, 0],
 [1, 1, 1],
 [1, 0, 1],
 [0, 0, 1],
 [0, 0, 0],
 [0, 1, 0],
 [0, 1, 1]]
\end{verbatim}

cuboctahedron(backend=None)

Return the cuboctahedron.
The cuboctahedron is an Archimedean solid with 12 vertices and 14 faces dual to the rhombic dodecahedron. It can be defined as the convex hull of the twelve vertices $(0, \pm1, \pm1), (\pm1, 0, \pm1)$ and $(\pm1, \pm1, 0)$. For more information, see the Wikipedia article Cuboctahedron.

INPUT:

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

See also:

rhombic_dodecahedron()

EXAMPLES:

```sage
c = polytopes.cuboctahedron()
c.f_vector()
(1, 12, 24, 14, 1)
```

Its facets are 8 triangles and 6 squares:

```sage
s = sum(1 for f in c.facets() if len(f.vertices()) == 3)
s
8
```

```sage
s = sum(1 for f in c.facets() if len(f.vertices()) == 4)
s
6
```

Some more computation:

```sage
c = polytopes.cuboctahedron()
c.volume()
20/3
```

```sage
c = polytopes.cuboctahedron()
c.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 $(t, t^2, \ldots, t^\text{dim})$ 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.
- base_ring – either QQ (default) or RDF.
- backend – the backend to use to create the polytope.

EXAMPLES:

```sage
c = polytopes.cyclic_polytope(4, 10)
c.f_vector()
(1, 10, 45, 70, 35, 1)
```

dodecahedron(exact=True, base_ring=None, backend=None)

Return a dodecahedron.

The dodecahedron is the Platonic solid dual to the icosahedron().

INPUT:

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

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

**EXAMPLES:**

```python
sage: d12 = polytopes.dodecahedron()
sage: d12.f_vector()
(1, 20, 30, 12, 1)
sage: d12.volume()
-176*sqrt5 + 400
sage: numerical_approx(_)
6.45203596003699

sage: d12 = polytopes.dodecahedron(exact=False)
sage: d12.base_ring()
Real Double Field
```

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**

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 $\mathbb{R}^m$, 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 [PS2002].

**INPUT:**

• **edges** – list (default: None); a list of edges of self. If not specified, the list of all edges of self is used with the default ordering of self.edges(). 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.
edges. Notice that the edges entered here must be in the precisely same format as outputted by \texttt{self.edges()}: so, if \texttt{self.edges()} outputs an edge in the form \((1, 3, \text{None})\), then \((1, 3)\) will not do!

- \texttt{ends} – (optional, default: \((\text{self.sources()}, \text{self.sinks()}\)) a pair \((S, T)\) of an iterable \(S\) and an iterable \(T\).

- \texttt{backend} – string or \texttt{None} (default); the backend to use; see \texttt{sage.geometry.polyhedron.constructor.Polyhedron()}

\textbf{Note:} Flow polytopes can also be built through the \texttt{polytopes.} \texttt{<tab>} object:

\begin{verbatim}
\texttt{sage: polytopes.flow_polytope(digraphs.Path(5))}
\end{verbatim}

A 0-dimensional polyhedron in \(\mathbb{Q}^4\) defined as the convex hull of 1 vertex

\textbf{EXAMPLES:}

A commutative square:

\begin{verbatim}
\texttt{sage: G = DiGraph({1: [2, 3], 2: [4], 3: [4]})}
\texttt{sage: fl = G.flow_polytope(); fl}
\end{verbatim}

A 1-dimensional polyhedron in \(\mathbb{Q}^4\) defined as the convex hull of 2 vertices

\texttt{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:

\begin{verbatim}
\texttt{sage: fl = G.flow_polytope(edges=G.edges(key=lambda x: x[0] - x[1])); fl}
\end{verbatim}

A 1-dimensional polyhedron in \(\mathbb{Q}^4\) defined as the convex hull of 2 vertices

\texttt{sage: fl.vertices()}

(A vertex at \((0, 1, 1, 0)\), A vertex at \((1, 0, 0, 1)\))

A tournament on 4 vertices:

\begin{verbatim}
\texttt{sage: H = digraphs.TransitiveTournament(4)}
\texttt{sage: fl = H.flow_polytope(); fl}
\end{verbatim}

A 3-dimensional polyhedron in \(\mathbb{Q}^6\) defined as the convex hull of 4 vertices

\texttt{sage: fl.vertices()}

(A vertex at \((0, 0, 1, 0, 0, 0)\), A vertex at \((0, 1, 0, 0, 0, 1)\), A vertex at \((1, 0, 0, 1, 0, 0)\), A vertex at \((1, 0, 0, 1, 1, 0)\))

Restricting to a subset of the edges:

\begin{verbatim}
\texttt{sage: fl = H.flow_polytope(edges=[[0, 1, None], (1, 2, None),}
\texttt{.....: (2, 3, None), (0, 3, None)]])}
\texttt{sage: fl.vertices()}
\end{verbatim}

(A vertex at \((0, 0, 0, 1)\), A vertex at \((1, 1, 1, 0)\))
Using a different choice of sources and sinks:

```
sage: fl1 = H.flow_polytope(ends=(1, [3])); fl1
A 1-dimensional polyhedron in QQ^6 defined as the convex hull
of 2 vertices
sage: fl1.vertices()
(A vertex at (0, 0, 0, 1, 0, 1), A vertex at (0, 0, 0, 0, 1, 0))
sage: fl2 = H.flow_polytope(ends=(0, 1), [3])); fl2
The empty polyhedron in QQ^6
sage: fl3 = H.flow_polytope(ends=(3, [0])); fl3
The empty polyhedron in QQ^6
sage: fl4 = H.flow_polytope(ends=(0, 1), [2, 3])); fl4
A 3-dimensional polyhedron in QQ^6 defined as the convex hull
of 5 vertices
sage: fl4.vertices()
(A vertex at (0, 0, 1, 1, 0, 0),
A vertex at (0, 1, 0, 0, 1, 0),
A vertex at (1, 0, 0, 2, 0, 1),
A vertex at (1, 0, 0, 1, 1, 0),
A vertex at (0, 1, 0, 1, 0, 1))
```

A digraph with one source and two sinks:

```
sage: Y = DiGraph({1: [2], 2: [3, 4]})
sage: Y.flow_polytope()
The empty polyhedron in QQ^3
```

A digraph with one vertex and no edge:

```
sage: Z = DiGraph({1: []})
sage: Z.flow_polytope()
A 0-dimensional polyhedron in QQ^0 defined as the convex hull
of 1 vertex
```

A digraph with multiple edges (trac ticket #28837):

```
sage: G = DiGraph([(0, 1), (0, 1)], multiedges=True)
sage: G
Multi-digraph on 2 vertices
sage: P = G.flow_polytope()
sage: P
A 1-dimensional polyhedron in QQ^2 defined as the convex hull of 2 vertices
sage: P.vertices()
```

generalized_permutahedron(coxeter_type, point=None, exact=True, regular=False, backend=None)

Return the generalized permutahedron of type `coxeter_type` as the convex hull of the orbit of `point` in the fundamental cone.

This generalized permutahedron lies in the vector space used in the geometric representation, that is, in the default case, the dimension of generalized permutahedron equals the dimension of the space.

**INPUT:**

- `coxeter_type` – a Coxeter type; given as a pair `[type,rank]`, where type is a letter and rank is the number of generators.
- `point` – a list (default: `None`); a point given by its coordinates in the weight basis. If `None` is given, the point `(1,1,1,...)` is used.
- `exact` - (boolean, default `True`) if `False` use floating point approximations instead of exact coordinates
- `regular` – boolean (default: `False`); whether to apply a linear transformation making the vertex figures isometric.
- `backend` – backend to use to create the polytope; (default: `None`)

**EXAMPLES:**

```sage
sage: perm_a3 = polytopes.generalized_permutahedron(['A',3]); perm_a3
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 24 vertices
```

You can put the starting point along the hyperplane of the first generator:

```sage
sage: perm_a3_011 = polytopes.generalized_permutahedron(['A',3],[0,1,1]); perm_a3_011
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 12 vertices
sage: perm_a3_110 = polytopes.generalized_permutahedron(['A',3],[1,1,0]); perm_a3_110
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 12 vertices
sage: perm_a3_110.is_combinatorially_isomorphic(perm_a3_011)
True
sage: perm_a3_101 = polytopes.generalized_permutahedron(['A',3],[1,0,1]); perm_a3_101
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 12 vertices
sage: perm_a3_101.is_combinatorially_isomorphic(perm_a3_101)
False
sage: perm_a3_011.f_vector()
(1, 12, 18, 8, 1)
sage: perm_a3_101.f_vector()
(1, 12, 24, 14, 1)
```

The usual output does not necessarily give a polyhedron with isometric vertex figures:

```sage
sage: perm_a2 = polytopes.generalized_permutahedron(['A',2])
sage: perm_a2.vertices()
```

(A vertex at (1, 0), A vertex at (0, 1))
sage: P.lines()
()
Setting regular=True applies a linear transformation to get isometric vertex figures and the result is inscribed. Even though there are traces of small numbers, the internal computations are done using an exact embedded NumberField:

```
sage: perm_a2_reg = polytopes.generalized_permutahedron(['A',2],regular=True)
sage: V = sorted(perm_a2_reg.vertices()); V          # random
[A vertex at (-1, 0),
 A vertex at (-1/2, -0.866025403784439?),
 A vertex at (-1/2, 0.866025403784439?),
 A vertex at (1/2, -0.866025403784439?),
 A vertex at (1/2, 0.866025403784439?),
 A vertex at (1.000000000000000?, 0.?e-18)]
sage: for v in V:
    ....:         for x in v:
    ....:             x.exactify()
sage: V
[A vertex at (-1, 0),
 A vertex at (-1/2, -0.866025403784439?),
 A vertex at (-1/2, 0.866025403784439?),
 A vertex at (1/2, -0.866025403784439?),
 A vertex at (1/2, 0.866025403784439?),
 A vertex at (1, 0)]
sage: perm_a2_reg.is_inscribed()
True
sage: perm_a3_reg = polytopes.generalized_permutahedron(['A',3],regular=True)
˓→# long time
sage: perm_a3_reg.is_inscribed()
˓→# long time
True
```

The same is possible with vertices in RDF:

```
sage: perm_a2_inexact = polytopes.generalized_permutahedron(['A',2],exact=False)
sage: sorted(perm_a2_inexact.vertices())
[A vertex at (-1.0, -1.0),
 A vertex at (-1.0, 0.0),
 A vertex at (0.0, -1.0),
 A vertex at (0.0, 1.0),
 A vertex at (1.0, 0.0),
 A vertex at (1.0, 1.0)]
sage: perm_a2_inexact_reg = polytopes.generalized_permutahedron(['A',2],
˓→exact=False,regular=True)
sage: sorted(perm_a2_inexact_reg.vertices())
[A vertex at (-1.0, 0.0),
```
A vertex at (-0.5, -0.8660254038),
A vertex at (-0.5, 0.8660254038),
A vertex at (0.5, -0.8660254038),
A vertex at (0.5, 0.8660254038),
A vertex at (1.0, 0.0)]

It works also with types with non-rational coordinates:

```
sage: perm_b3 = polytopes.generalized_permutahedron(['B',3]); perm_b3 # long
    time
A 3-dimensional polyhedron in (Number Field in a with defining polynomial x^2 -
    2 with a = 1.414213562373095?)^3 defined as the convex hull of 48 vertices
```

```
sage: perm_b3_reg = polytopes.generalized_permutahedron(['B',3],regular=True); perm_b3_reg # not tested - long time (12sec on 64 bits).
A 3-dimensional polyhedron in AA^3 defined as the convex hull of 48 vertices
```

It is faster with the backend 'normaliz':

```
sage: perm_b3_reg_norm = polytopes.generalized_permutahedron(['B',3],regular=True,backend='normaliz') # optional - pynormaliz
sage: perm_b3_reg_norm
A 3-dimensional polyhedron in AA^3 defined as the convex hull of 48 vertices
```

The backend 'normaliz' allows further faster computation in the non-rational case:

```
sage: perm_h3 = polytopes.generalized_permutahedron(['H',3],backend='normaliz'). # optional - pynormaliz
sage: perm_h3
A 3-dimensional polyhedron in (Number Field in a with defining polynomial x^2 -
    5 with a = 2.236067977499790?)^3 defined as the convex hull of 120 vertices
```

```
sage: perm_f4 = polytopes.generalized_permutahedron(['F',4],backend='normaliz'). # optional - pynormaliz, long time
sage: perm_f4
A 4-dimensional polyhedron in (Number Field in a with defining polynomial x^2 -
    2 with a = 1.414213562373095?)^4 defined as the convex hull of 1152 vertices
```

See also:

- permutahedron()
- permutahedron()

```
grand_antiprism(exact=True, backend=None, verbose=False)
    Return the grand antiprism.
```

The grand antiprism is a 4-dimensional non-Wythoffian uniform polytope. The coordinates were taken from [http://eusebeia.dyndns.org/4d/gap](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
# not tested - very long time
A 4-dimensional polyhedron in (Number Field in sqrt5 with defining polynomial x^2 - 5 with sqrt5 = 2.236067977499790?)^4 defined as the convex hull of 100 vertices

Computation with the backend 'normaliz' is instantaneous:

```python
sage: gap_norm = polytopes.grand_antiprism(backend='normaliz')  # optional - pynormaliz
sage: gap_norm
# optional - pynormaliz
A 4-dimensional polyhedron in (Number Field in sqrt5 with defining polynomial x^2 - 5 with sqrt5 = 2.236067977499790?)^4 defined as the convex hull of 100 vertices
```

Computation with approximated coordinates is also faster, but inexact:

```python
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 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:**
A faster implementation is obtained by setting `exact=False`:

```python
sage: gr = polytopes.great_rhombicuboctahedron(exact=False)
sage: gr.f_vector()
```
```
(1, 48, 72, 26, 1)
```

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

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

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

Return a hypercube of the given dimension.

The `dim`-dimensional hypercube is by default the convex hull of the \(2^{\text{dim}} \pm 1\) vectors of length \(\text{dim}\). Alternatively, it is the product of \(\text{dim}\) line segments given in the intervals. For more information see the wikipedia article [Wikipedia article Hypercube](https://en.wikipedia.org/wiki/Hypercube).

**INPUT:**

- `dim` – integer. The dimension of the hypercube.
- `intervals` – (default = None). It takes the following possible inputs:
  - If `None` (the default), it returns the \(\pm 1\)-cube of dimension `dim`.
  - `'zero_one'` – (string). Return the 0/1-cube.
  - a list of length `dim`. Its elements are pairs of numbers \((a, b)\) with \(a < b\). The cube will be the product of these intervals.
- `backend` – the backend to use to create the polytope.

**EXAMPLES:**

Create the \(\pm 1\)-hypercube of dimension 4:

```python
sage: four_cube = polytopes.hypercube(4)
sage: four_cube.is_simple()
True
```

Return the 0/1-hypercube of dimension 4:
sage: z_cube = polytopes.hypercube(4, intervals = 'zero_one')
sage: z_cube.vertices()[0]
A vertex at (1, 0, 1, 1)
sage: z_cube.is_simple()
True
sage: z_cube.base_ring()
Integer Ring
sage: z_cube.volume()
1
sage: z_cube.ehrhart_polynomial()  # optional - latte_int
```
t^4 + 4*t^3 + 6*t^2 + 4*t + 1
```

Return the 4-dimensional combinatorial cube that is the product of [0,3]^4:

sage: t_cube = polytopes.hypercube(4, intervals = [[0,3]]*4)

Checking that t_cube is three times the previous 0/1-cube:

sage: t_cube == 3 * z_cube
True

**hypersimplex(dim, k, project=False, backend=None)**

Return the hypersimplex in dimension dim and parameter k.

The hypersimplex $\Delta_{d,k}$ is the convex hull of the vertices made of $k$ ones and $d-k$ zeros. It lies in the $d-1$ hyperplane of vectors of sum $k$. If you want a projected version to $\mathbb{R}^{d-1}$ (with floating point coordinates) then set `project=True` in the options.

See also:

**simplex()**

**INPUT:**

- `dim` – the dimension
- `n` – the numbers $(1,...,n)$ are permuted
- `project` – (boolean, default False) if True, the polytope is (isometrically) projected to a vector space of dimension `dim-1`. This operation turns the coordinates into floating point approximations and corresponds to the projection given by the matrix from `zero_sum_projection()`.
- `backend` – the backend to use to create the polytope.

**EXAMPLES:**

sage: 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: TestSuite(h_4_2).run()
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

(continues on next page)
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 $\mathbb{A}A < \text{sage.rings.qqbar.AlgebraicRealField}>$:

```
sage: ico = polytopes.icosahedron(baseリング=AA)  # long time
sage: ico.base_ring()
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:

```python
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_V2()  # 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 with sqrt5 = 2. 
\rightarrow 236067977499790?
```

A much faster implementation using floating point approximations:

```python
sage: id = polytopes.icosidodecahedron_V2(exact=False)
sage: id.f_vector()
(1, 30, 60, 32, 1)
sage: id.base_ring()
Real Double Field
```

Its facets are 20 triangles and 12 regular pentagons:

```python
sage: sum(1 for f in id.facets() if len(f.vertices()) == 3)
20
sage: sum(1 for f in id.facets() if len(f.vertices()) == 5)
12
```

octahedron(backend=None)
Return the octahedron.

The octahedron is a Platonic solid with 6 vertices and 8 faces dual to the cube. It can be defined as the convex hull of the six vertices \((0, 0, \pm 1), (\pm 1, 0, 0)\) and \((0, \pm 1, 0)\). For more information, see the Wikipedia article Octahedron.

INPUT:
• backend – the backend to use to create the polytope.
EXAMPLES:

```
sage: co = polytopes.octahedron()
sage: co.f_vector()
(1, 6, 12, 8, 1)
```

Its facets are 8 triangles:

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

Some more computation:

```
sage: co.volume()
4/3
sage: co.ehrhart_polynomial()  # optional - latte_int
4/3*t^3 + 2*t^2 + 8/3*t + 1
```

`omnitruncated_one_hundred_twenty_cell` *(exact=True, backend=None)*

Return the omnitruncated 120-cell.

The omnitruncated 120-cell is a 4-dimensional 4-uniform polytope in the \( H_4 \) family. It has 14400 vertices. For more information see Wikipedia article Omnitruncated 120-cell.

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

**INPUT:**

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

**EXAMPLES:**

```
sage: polytopes.omnitruncated_one_hundred_twenty_cell(backend='normaliz')  # not tested - very long time ~10min
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 14400 vertices
```

`omnitruncated_six_hundred_cell` *(exact=True, backend=None)*

Return the omnitruncated 120-cell.

The omnitruncated 120-cell is a 4-dimensional 4-uniform polytope in the \( H_4 \) family. It has 14400 vertices. For more information see Wikipedia article Omnitruncated 120-cell.

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

**INPUT:**

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

**EXAMPLES:**

```
```
The 120-cell is a 4-dimensional 4-uniform polytope in the $H_4$ family. It has 600 vertices and 120 facets. For more information see Wikipedia article 120-cell.

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

**INPUT:**

- `exact` – (boolean, default `True`) if `True` use exact coordinates instead of floating point approximations.
- `backend` – the backend to use to create the polytope.
- `construction` – the construction to use (string, default 'coxeter'); the other possibility is 'as_permutahedron'.

**EXAMPLES:**

The classical construction given by Coxeter in [Cox1969] is given by:

```python
sage: polytopes.one_hundred_twenty_cell() # nottested - very long time ~10min
```

A 4-dimensional polyhedron in AA^4 defined as the convex hull of 14400 vertices

One can also use the `normaliz` backend:

```python
sage: P = polytopes.one_hundred_twenty_cell(backend='normaliz'); P
```

A 4-dimensional polyhedron in (Number Field in sqrt5 with defining polynomial $x^2 - 5$ with sqrt5 = 2.236067977499790?)^4 defined as the convex hull of 600 vertices

It is also possible to realize it using the generalized permutahedron of type $H_4$:

```python
sage: P = polytopes.one_hundred_twenty_cell(backend='normaliz',construction='as_permutahedron'); P
```

A 4-dimensional polyhedron in AA^4 defined as the convex hull of 600 vertices

**parallelotope** 

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 $\mathbb{Z}^2$ 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 $\mathbb{Z}^4$ defined as the convex hull of 16 vertices

```sage
K = QuadraticField(2, 'sqrt2')
sqrt2 = K.gen()
P = polytopes.parallelotope([(1,sqrt2), (1,-1)]); P
```
A 2-dimensional polyhedron in $(\mathbb{Q}[\sqrt{2}])^2$ defined as the convex hull of 4 vertices

pentakis_dodecahedron(exact=True, base_ring=None, backend=None)

Return the pentakis dodecahedron.

The pentakis dodecahedron (or icosidodecahedron) is a face-regular, vertex-uniform polytope dual to the truncated icosahedron. It has 60 facets and 32 vertices. See the Wikipedia article Pentakis_dodecahedron for more information.

INPUT:

• **exact** – (boolean, default True) If False use an approximate ring for the coordinates.
• **base_ring** – the ring in which the coordinates will belong to. If it is not provided and exact=True it will be a the number field $\mathbb{Q}[\phi]$ 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.facets())
```
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 \(dim-1\). This operation turns the coordinates into floating point approximations and corresponds to the projection given by the matrix from zero_sum_projection().
- \(backend\) – the backend to use to create the polytope.

EXAMPLES:

```
sage: perm4 = polytopes.permutahedron(4)
sage: perm4
A 3-dimensional polyhedron in ZZ^4 defined as the convex hull of 24 vertices
sage: perm4.is_lattice_polytope()
True
sage: perm4.ehrhart_polynomial()  # optional - latte_int
16*t^3 + 15*t^2 + 6*t + 1
sage: perm4 = polytopes.permutahedron(4, project=True)
sage: perm4
A 3-dimensional polyhedron in RDF^3 defined as the convex hull of 24 vertices
sage: perm4.plot()
Graphics3d Object
sage: perm4.graph().is_isomorphic(graphs.BubbleSortGraph(4))
True
```

As both Hrepresentation an Vrepresentation are known, the permutahedron can be set up with both using the backend field. The following takes very very long time to recompute, e.g. with backend ppl:

```
sage: polytopes.permutahedron(8, backend='field')  # (~1s)
A 7-dimensional polyhedron in QQ^8 defined as the convex hull of 40320 vertices
sage: polytopes.permutahedron(9, backend='field')  # not tested (memory consumption)  # (~5s)
A 8-dimensional polyhedron in QQ^9 defined as the convex hull of 362880 vertices
```

See also:

- BubbleSortGraph()

rectified_one_hundred_twenty_cell\((exact=True, backend=None)\)

Return the rectified 120-cell.

The rectified 120-cell is a 4-dimensional 4-uniform polytope in the \(H_4\) family. It has 1200 vertices. For more information see Wikipedia article Rectified 120-cell.
Warning: The coordinates are exact by default. The computation with inexact coordinates (using the backend 'cdd') returns a numerical inconsistency error, and thus cannot be computed.

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

**EXAMPLES:**

```python
sage: polytopes.rectified_one_hundred_twenty_cell(backend='normaliz') # not tested - long time
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 1200 vertices
```

**rectified_six_hundred_cell**(exact=True, backend=None)
Return the rectified 600-cell.

The rectified 600-cell is a 4-dimensional 4-uniform polytope in the $H_4$ family. It has 720 vertices. For more information see Wikipedia article Rectified 600-cell.

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

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

**EXAMPLES:**

```python
sage: polytopes.rectified_six_hundred_cell(backend='normaliz') # not tested - long time ~14sec
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 720 vertices
```

**regular_polygon**(n, exact=True, base_ring=None, backend=None)
Return a regular polygon with $n$ vertices.

**INPUT:**
- `n` – a positive integer, the number of vertices.
- `exact` – (boolean, default True) if False floating point numbers are used for coordinates.
- `base_ring` – a ring in which the coordinates will lie. It is None by default. If it is not provided and `exact` is True then it will be the field of real algebraic number, if `exact` is False it will be the real double field.
- `backend` – the backend to use to create the polytope.

**EXAMPLES:**

```python
sage: octagon = polytopes.regular_polygon(8)
sage: octagon
A 2-dimensional polyhedron in AA^2 defined as the convex hull of 8 vertices
sage: octagon.n_vertices()
```

(continues on next page)
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 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 facets are 12 quadrilaterals (not all identical):

sage: sum(1 for f in rd.facets() 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 infor-
mation, see the Wikipedia article Rhombicosidodecahedron.

INPUT:
• `exact` – (boolean, default `True`) If `False` use an approximate ring for the coordinates.

• `base_ring` – the ring in which the coordinates will belong to. If it is not provided and `exact=True` it will be a the number field $\mathbb{Q}[\phi]$ 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 with sqrt5 = 2.
˓→236067977499790?
```

A much faster implementation using floating point approximations:

```python
sage: rid = polytopes.rhombicosidodecahedron(exact=False)
```

It's facets are 20 triangles, 30 squares and 12 pentagons:

```python
sage: sum(1 for f in rid.facets() if len(f.vertices()) == 3)  # long time
20
sage: sum(1 for f in rid.facets() if len(f.vertices()) == 4)  # long time
30
sage: sum(1 for f in rid.facets() if len(f.vertices()) == 5)  # long time
12
```

`runcinated_one_hundred_twenty_cell(exact=False, backend=None)`

Return the runcinated 120-cell.

The runcinated 120-cell is a 4-dimensional 4-uniform polytope in the $H_4$ family. It has 2400 vertices. For more information see Wikipedia article Runcinated 120-cell.

**Warning:** The coordinates are inexact by default. The computation with inexact coordinates (using the backend `cdd`) issues a `UserWarning` on inconsistencies.

INPUT:

• `exact` - (boolean, default `False`) if `True` use exact coordinates instead of floating point approxima-

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

EXAMPLES:

```python
sage: polytopes.runcinated_one_hundred_twenty_cell(exact=False) # not tested ˓→very long time
doctest:warning ... UserWarning: This polyhedron data is numerically complicated; cdd could not convert between the inexact
```
V and H representation without loss of data. The resulting object might show inconsistencies.

A 4-dimensional polyhedron in RDF^4 defined as the convex hull of 2400 vertices.

It is possible to use the backend 'normaliz' to get an exact representation:

```sage
polytopes.runcinated_one_hundred_twenty_cell(exact=True, backend='normaliz')  # not tested - very long time
```

A 4-dimensional polyhedron in AA^4 defined as the convex hull of 2400 vertices.

```
Warning: The coordinates are inexact by default. The computation with inexact coordinates (using the backend 'cdd') issues a UserWarning on inconsistencies.
```

**INPUT:**

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

**EXAMPLES:**

```sage
polytopes.runcitruncated_one_hundred_twenty_cell(exact=False)  # not tested - very long time
```

... UserWarning: This polyhedron data is numerically complicated; cdd could not convert between the inexact V and H representation without loss of data. The resulting object might show inconsistencies.

It is possible to use the backend 'normaliz' to get an exact representation:

```sage
polytopes.runcitruncated_one_hundred_twenty_cell(exact=True, backend='normaliz')  # not tested - very long time
```

A 4-dimensional polyhedron in AA^4 defined as the convex hull of 2400 vertices.

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

**INPUT:**

```
runcitruncated_one_hundred_twenty_cell(exact=False, backend=None)
Return the runcitruncated 120-cell.
```

The runcitruncated 120-cell is a 4-dimensional 4-uniform polytope in the $H_4$ family. It has 7200 vertices. For more information see Wikipedia article Runcitruncated 120-cell.

**Warning:** The coordinates are inexact by default. The computation with inexact coordinates (using the backend 'cdd') issues a UserWarning on inconsistencies.

**INPUT:**

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

**EXAMPLES:**

```sage
polytopes.runcitruncated_one_hundred_twenty_cell(exact=False)  # not tested - very long time
```

... UserWarning: This polyhedron data is numerically complicated; cdd could not convert between the inexact V and H representation without loss of data. The resulting object might show inconsistencies.

It is possible to use the backend 'normaliz' to get an exact representation:

```sage
polytopes.runcitruncated_one_hundred_twenty_cell(exact=True, backend='normaliz')  # not tested - very long time
```

A 4-dimensional polyhedron in AA^4 defined as the convex hull of 7200 vertices.

```
runcitruncated_six_hundred_cell(exact=True, backend=None)
Return the runcitruncated 600-cell.
```

The runcitruncated 600-cell is a 4-dimensional 4-uniform polytope in the $H_4$ family. It has 7200 vertices. For more information see Wikipedia article Runcitruncated 600-cell.

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

**INPUT:**
• exact - (boolean, default True) if True use exact coordinates instead of floating point approximations.
• backend – the backend to use to create the polytope.

EXAMPLES:

```
sage: polytopes.runcitruncated_six_hundred_cell(backend='normaliz') # not tested - very long time
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 7200 vertices
```

```
sage: polytopes.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.

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:

```
sage: s5 = polytopes.simplex(5)
sage: s5
A 5-dimensional polyhedron in ZZ^6 defined as the convex hull of 6 vertices
sage: s5.f_vector()
(1, 6, 15, 20, 15, 6, 1)
sage: s5 = polytopes.simplex(5, project=True)
sage: s5
A 5-dimensional polyhedron in RDF^5 defined as the convex hull of 6 vertices
```

Its volume is $\sqrt{d+1}/d!$:

```
sage: s5 = polytopes.simplex(5, project=True)
sage: s5.volume()  # abs tol 1e-10
0.0204124145231931
sage: sqrt(6.) / factorial(5)
0.0204124145231931
sage: s6 = polytopes.simplex(6, project=True)
sage: s6.volume()  # abs tol 1e-10
0.00367465459870082
sage: sqrt(7.) / factorial(6)
0.00367465459870082
```
Computation in algebraic reals:

```python
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:**

```python
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:

```python
sage: p600 = polytopes.six_hundred_cell(exact=True)  # not tested - very long

```

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

Return the (small) rhombicuboctahedron.

The rhombicuboctahedron is an Archimedean solid with 24 vertices and 26 faces. See the Wikipedia article Rhombicuboctahedron for more information.

**INPUT:**

- `exact` – (boolean, default True) If False use an approximate ring for the coordinates.
- `base_ring` – the ring in which the coordinates will belong to. If it is not provided and exact=True it will be a the number field $\mathbb{Q} [\phi]$ 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)
```

(continues on next page)
The faces are 8 equilateral triangles and 18 squares:

```
sage: sum(1 for f in sr.facets() if len(f.vertices()) == 3)
sage: sum(1 for f in sr.facets() if len(f.vertices()) == 4)
```

Its non exact version:

```
sage: sr = polytopes.snub_cube(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**(exact=False, base_ring=None, backend=None, verbose=False)

Return a snub cube.

The snub cube is an Archimedean solid. It has 24 vertices and 38 faces. For more information see the Wikipedia article Snub_cube.

The constant $z$ used in constructing this polytope is the reciprocal of the tribonacci constant, that is, the solution of the equation $x^3 + x^2 + x - 1 = 0$. See Wikipedia article Generalizations_of_Fibonacci_numbers#Tribonacci_numbers.

**INPUT:**

- **exact** – (boolean, default False) if True use exact coordinates instead of floating point approximations
- **base_ring** – the field to use. If None (the default), construct the exact number field needed (if exact is True) or default to RDF (if exact is True).
- **backend** – the backend to use to create the polytope. If None (the default), the backend will be selected automatically.

**EXAMPLES:**

```
sage: sc_inexact = polytopes.snub_cube(exact=False)
sage: sc_inexact
A 3-dimensional polyhedron in RDF^3 defined as the convex hull of 24 vertices
sage: sc_inexact.f_vector()
(1, 24, 60, 38, 1)
sage: sc_exact = polytopes.snub_cube(exact=True)
# long time
sage: sc_exact.f_vector()
# long time
(1, 24, 60, 38, 1)
sage: sorted(sc_exact.vertices())
# long time
[A vertex at (-1, -z, -z^2),
 A vertex at (-1, -z^2, z),
 A vertex at (-1, z^2, -z),
 A vertex at (-1, z, z^2),
 A vertex at (-z, -1, z^2),
(continues on next page)
A vertex at \((-z, -z^2, -1)\),
A vertex at \((-z, z^2, 1)\),
A vertex at \((-z, 1, -z^2)\),
A vertex at \((-z^2, -1, -z)\),
A vertex at \((-z^2, z, -1)\),
A vertex at \((-z^2, z, -1)\),
A vertex at \((-z^2, 1, -z)\),
A vertex at \((z^2, -1, -z^2)\),
A vertex at \((z^2, -z, 1)\),
A vertex at \((z^2, z, 1)\),
A vertex at \((z^2, 1, -z)\),
A vertex at \((z, -1, -z^2)\),
A vertex at \((z, -z^2, 1)\),
A vertex at \((z, z^2, -1)\),
A vertex at \((z, 1, z^2)\),
A vertex at \((1, -z, z^2)\),
A vertex at \((1, -z^2, -z)\),
A vertex at \((1, z^2, z)\),
A vertex at \((1, z, -z^2)\)

```
sage: sc_exact.is_combinatorially_isomorphic(sc_inexact)  # long time
True
```

\texttt{snub\_dodecahedron}(\texttt{base\_ring=\text{None}, backend=\text{None}, verbose=\text{False}})

Return the snub dodecahedron.

The snub dodecahedron is an Archimedean solid. It has 92 faces and 60 vertices. For more information, see the Wikipedia article \texttt{Snub\_dodecahedron}.

\textbf{INPUT:}

- \texttt{base\_ring} – the ring in which the coordinates will belong to. If it is not provided it will be the real double field.

- \texttt{backend} – the backend to use to create the polytope.

\textbf{EXAMPLES:}

Only the backend using the optional \texttt{normaliz} package can construct the snub dodecahedron in reasonable time:

```
sage: sd = polytopes.snub_dodecahedron(base_ring=AA, backend='normaliz')  # optional - pynormaliz, long time
sage: sd.f_vector()  # optional - pynormaliz, long time
(1, 60, 150, 92, 1)
sage: sd.base_ring()  # optional - pynormaliz, long time
Algebraic Real Field
```

Its facets are 80 triangles and 12 pentagons:

```
sage: sum(1 for f in sd.facets() if len(f.vertices()) == 3)  # optional - pynormaliz, long time
80
```
tetrahedron\(\text{(backend=\text{None})}\)

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:

\(\text{simplex()}\)

EXAMPLES:

```python
sage: co = polytopes.tetrahedron()
sage: co.f_vector()  # optional - pynormaliz, long time
(1, 4, 6, 4, 1)
```

Its facets are 4 triangles:

```python
sage: sum(1 for f in co.facets() if len(f.vertices()) == 3)  # optional - pynormaliz, long time
4
```

Some more computation:

```python
sage: co.volume()  # optional - pynormaliz, long time
1/3
sage: co.ehrhart_polynomial()  # optional - latte_int
1/3*t^3 + t^2 + 5/3*t + 1
```

truncated_cube\(\text{(exact=True, base_ring=\text{None}, backend=\text{None})}\)

Return the truncated cube.

The truncated cube is an Archimedean solid with 24 vertices and 14 faces. It can be defined as the convex hull of the 24 vertices \((\pm x, \pm 1, \pm 1), (\pm 1, \pm x, \pm 1), (\pm 1, \pm 1, \pm x)\) where \(x = \sqrt(2) - 1\). For more information, see the Wikipedia article Truncated_cube.

INPUT:

- exact – (boolean, default True) If False use an approximate ring for the coordinates.
- base_ring – the ring in which the coordinates will belong to. If it is not provided and exact=True it will be a the number field \(\mathbb{Q}[\sqrt{2}]\) and if exact=False it will be the real double field.
- backend – the backend to use to create the polytope.

EXAMPLES:

```python
sage: co = polytopes.truncated_cube()
sage: co.f_vector()  # optional - pynormaliz, long time
(1, 24, 36, 14, 1)
```

Its facets are 8 triangles and 6 octagons:
Some more computation:

```python
sage: co.volume()
56/3*sqrt2 - 56/3
```

**truncated_dodecahedron**\((\text{exact}=\text{True}, \text{base\_ring}=\text{None}, \text{backend}=\text{None})\)

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

**INPUT:**

- \texttt{exact} – (boolean, default \texttt{True}) If \texttt{False} use an approximate ring for the coordinates.
- \texttt{base\_ring} – the ring in which the coordinates will belong to. If it is not provided and \texttt{exact=\text{True}} it will be a the number field \(\mathbb{Q}[\phi]\) where \(\phi\) is the golden ratio and if \texttt{exact=\text{False}} it will be the real double field.
- \texttt{backend} – the backend to use to create the polytope.

**EXAMPLES:**

```python
sage: td = polytopes.truncated_dodecahedron()
sage: td.f_vector()
(1, 60, 90, 32, 1)
sage: td.base_ring()
Number Field in sqrt5 with defining polynomial x^2 - 5 with sqrt5 = 2.
˓→236067977499790?
```

Its facets are 20 triangles and 12 regular decagons:

```python
sage: sum(1 for f in td.facets() if len(f.vertices()) == 3)
20
sage: sum(1 for f in td.facets() if len(f.vertices()) == 10)
12
```

The faster implementation using floating point approximations does not fully work unfortunately, see [https://github.com/cddlib/cddlib/pull/7](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()
```

(continues on next page)
truncated_icosidodecahedron\( (\text{exact}=\text{True}, \text{base\_ring}=\text{None}, \text{backend}=\text{None}) \)

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](https://en.wikipedia.org/wiki/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:**

```sage
sage: ti = polytopes.truncated_icosidodecahedron() # long time
sage: ti.f_vector() # long time
(1, 120, 180, 62, 1)

sage: ti.base_ring() # long time
Number Field in sqrt5 with defining polynomial x^2 - 5 with sqrt5 = 2.
    → 236067977499790?
```

The implementation using floating point approximations is much faster:

```sage
sage: ti = polytopes.truncated_icosidodecahedron(exact=False) # random
sage: ti.f_vector()
(1, 120, 180, 62, 1)

sage: ti.base_ring()
Real Double Field
```

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

```sage
sage: sum(1 for f in ti.facets() if len(f.vertices()) == 4)
30

sage: sum(1 for f in ti.facets() if len(f.vertices()) == 6)
20

sage: sum(1 for f in ti.facets() if len(f.vertices()) == 10)
12
```

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

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

This is also known as the permutohedron of dimension 3.

**INPUT:**

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

**EXAMPLES:**
Its facets are 6 squares and 8 hexagons:

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

Some more computation:

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

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

Return the truncated 120-cell.

The truncated 120-cell is a 4-dimensional 4-uniform polytope in the $H_4$ family. It has 2400 vertices. For more information see Wikipedia article Truncated 120-cell.

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

**INPUT:**

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

**EXAMPLES:**

```python
sage: polytopes.truncated_one_hundred_twenty_cell(backend='normaliz') # not tested - long time
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 2400 vertices
```

**truncated_six_hundred_cell** *(exact=False, backend=None)*

Return the truncated 600-cell.

The truncated 600-cell is a 4-dimensional 4-uniform polytope in the $H_4$ family. It has 1440 vertices. For more information see Wikipedia article Truncated 600-cell.

**Warning:** The coordinates are not exact by default. The computation with exact coordinates takes a huge amount of time.

**INPUT:**

- exact - (boolean, default False) if True use exact coordinates instead of floating point approximations
- backend – the backend to use to create the polytope.
EXAMPLES:

```
sage: polytopes.truncated_six_hundred_cell()  # not tested - long time
A 4-dimensional polyhedron in RDF^4 defined as the convex hull of 1440 vertices
```

It is possible to use the backend 'normaliz' to get an exact representation:

```
sage: polytopes.truncated_six_hundred_cell(exact=True, backend='normaliz')  # not tested - long time ~16 sec
A 4-dimensional polyhedron in AA^4 defined as the convex hull of 1440 vertices
```

```
truncated_tetrahedron(backend=None)

Return the truncated tetrahedron.

The truncated tetrahedron is an Archimedean solid with 12 vertices and 8 faces. It can be defined as the convex hull of 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 facets are 4 triangles and 4 hexagons:

```
sage: sum(1 for f in co.facets() if len(f.vertices()) == 3)
4
sage: sum(1 for f in co.facets() if len(f.vertices()) == 6)
4
```

Some more computation:

```
sage: co.volume()
184/3
sage: co.ehrhart_polynomial()  # optional - latte_int
184/3*t^3 + 28*t^2 + 26/3*t + 1
```

```
twenty_four_cell(backend=None)

Return the standard 24-cell polytope.

The 24-cell polyhedron (also called icositetrachoron or octaplex) is a regular polyhedron in 4-dimension. For more information see the Wikipedia article 24-cell.

INPUT:

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

EXAMPLES:

```
sage: p24 = polytopes.twenty_four_cell()
sage: p24.f_vector()
(1, 24, 96, 96, 24, 1)
sage: v = next(p24.vertex_generator())
```

(continues on next page)
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:

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: P = polytopes.parallelotope([ (1,sqrt2), (1,-1) ]); P
A 2-dimensional polyhedron in (Number Field in sqrt2 with defining polynomial x^2 - 2 with sqrt2 = 1.414213562373095?)^2 defined as
the convex hull of 4 vertices

sage.geometry.polyhedron.library.gale_transform_to_polytope(vectors, base_ring=None, backend=None)
Return the polytope associated to the list of vectors forming a Gale transform.
This function is the inverse of gale_transform() up to projective transformation.

INPUT:
• vectors – the vectors of the Gale transform
• base_ring – string (default: None); the base ring to be used for the construction
• backend – string (default: None); the backend to use to create the polytope

Note: The order of the input vectors will not be preserved.
If the center of the (input) vectors is the origin, the function is much faster and might give a nicer representation of the polytope.
If this is not the case, the vectors will be scaled (each by a positive scalar) accordingly to obtain the polytope.

See also:

:func:`~sage.geometry.polyhedron.library.gale_transform_to_primal`.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.library import gale_transform_to_polytope
sage: points = polytopes.octahedron().gale_transform()
sage: points
((0, -1), (-1, 0), (1, 1), (1, 1), (-1, 0), (0, -1))
sage: P = gale_transform_to_polytope(points); P
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 6 vertices
sage: P.vertices()
(A vertex at (-1, 0, 0),
 A vertex at (0, -1, 0),
 A vertex at (0, 0, -1),
 A vertex at (0, 0, 1),
 A vertex at (0, 1, 0),
 A vertex at (1, 0, 0))
```

One can specify the base ring:

```
sage: gale_transform_to_polytope(  
.....:     [(1,1), (-1,-1), (1,0),  
.....:     (-1,0), (1,-1), (-2,1)].vertices()  
(A vertex at (-25, 0, 0),
 A vertex at (-15, 50, -60),
 A vertex at (0, -25, 0),
 A vertex at (0, 0, -25),
 A vertex at (16, -35, 54),
 A vertex at (24, 10, 31))
sage: gale_transform_to_polytope(  
.....:     [(1,1), (-1,-1), (1,0),  
.....:     (-1,0), (1,-1), (-2,1)],
.....:     base_ring=RDF).vertices()  
(A vertex at (-0.64, 1.4, -2.16),
 A vertex at (-0.96, -0.4, -1.24),
 A vertex at (0.6, -2.0, 2.4),
 A vertex at (1.0, 0.0, 0.0),
 A vertex at (0.0, 1.0, 0.0),
 A vertex at (0.0, 0.0, 1.0))
```

One can also specify the backend:

```
sage: gale_transform_to_polytope(  
.....:     [(1,1), (-1,-1), (1,0),  
.....:     (-1,0), (1,-1), (-2,1)],
.....:     backend='field').backend()  
'field'
sage: gale_transform_to_polytope(  
.....:     [(1,1), (-1,-1), (1,0),  
.....:     (-1,0), (1,-1), (-2,1)],
```

(continues on next page)
A gale transform corresponds to a polytope if and only if every oriented (linear) hyperplane has at least two vectors on each side. See Theorem 6.19 of [Zie2007]. If this is not the case, one of two errors is raised.

If there is such a hyperplane with no vector on one side, the vectors are not totally cyclic:

```python
sage: gale_transform_to_polytope([(0,1), (1,1), (1,0), (-1,0)])
Traceback (most recent call last):
  ...:
ValueError: input vectors not totally cyclic
```

If every hyperplane has at least one vector on each side, then the gale transform corresponds to a point configuration. It corresponds to a polytope if and only if this point configuration is convex and if and only if every hyperplane contains at least two vectors of the gale transform on each side.

If this is not the case, an error is raised:

```python
sage: gale_transform_to_polytope([(0,1), (1,1), (1,0), (-1,-1)])
Traceback (most recent call last):
  ...:
ValueError: the gale transform does not correspond to a polytope
```

`sage.geometry.polyhedron.library.gale_transform_to_primal(vectors, base_ring=None, backend=None)`

Return a point configuration dual to a totally cyclic vector configuration.

This is the dehomogenized vector configuration dual to the input. The dual vector configuration is acyclic and can therefore be dehomogenized as the input is totally cyclic.

INPUT:

- `vectors` – the ordered vectors of the Gale transform
- `base_ring` – string (default: `None`); the base ring to be used for the construction
- `backend` – string (default: `None`); the backend to be use to construct a polyhedral, used internally in case the center is not the origin, see `Polyhedron()`

OUTPUT: An ordered point configuration as list of vectors.

**Note:** If the center of the (input) vectors is the origin, the function is much faster and might give a nicer representation of the point configuration.

If this is not the case, the vectors will be scaled (each by a positive scalar) accordingly.

**ALGORITHM:**

Step 1: If the center of the (input) vectors is not the origin, we do an appropriate transformation to make it so.

Step 2: We add a row of ones on top of \( \text{Matrix}(vectors) \). The right kernel of this larger matrix is the dual configuration space, and a basis of this space provides the dual point configuration.

More concretely, the dual vector configuration (inhomogeneous) is obtained by taking a basis of the right kernel of \( \text{Matrix}(vectors) \). If the center of the (input) vectors is the origin, there exists a basis of the right kernel of the form \([[1], [V]]\), where \([1]\) represents a row of ones. Then, \(V\) is a dehomogenization and thus the dual point configuration.
To extend [1] to a basis of \texttt{Matrix(vectors)}, we add a row of ones to \texttt{Matrix(vectors)} and calculate a basis of the right kernel of the obtained matrix.

REFERENCES:

For more information, see Section 6.4 of [Zie2007] or Definition 2.5.1 and Definition 4.1.35 of [DLRS2010].

See also:

\texttt{:func:`sage.geometry.polyhedron.library.gale_transform_to_polytope`}.

EXAMPLES:

```python
sage: from sage.geometry.polyhedron.library import gale_transform_to_primal
sage: points = ((0, -1), (-1, 0), (1, 1), (1, 1), (-1, 0), (0, -1))
sage: gale_transform_to_primal(points)
[(0, 0, 1), (0, 1, 0), (1, 0, 0), (-1, 0, 0), (0, -1, 0), (0, 0, -1)]
```

One can specify the base ring:

```python
sage: gale_transform_to_primal(  
....:   [(1,1), (-1,-1), (1,0),  
....:    (-1,0), (1,-1), (-2,1)])
[[16, -35, 54],  
(24, 10, 31),  
(-15, 50, -60),  
(-25, 0, 0),  
(0, -25, 0),  
(0, 0, -25)]
```

One can also specify the backend to be used internally:

```python
sage: gale_transform_to_primal(  
....:   [(1,1), (-1,-1), (1,0),  
....:    (-1,0), (1,-1), (-2,1)], backend='field')
[[48, -71, 88],  
(84, -28, 99),  
(-77, 154, -132),  
(-55, 0, 0),  
(0, -55, 0),  
(0, 0, -55)]
```
The input vectors should be totally cyclic:

```python
sage: gale_transform_to_primal([[0,1], (1,0), (1,1), (-1,0)])
Traceback (most recent call last):
...
ValueError: input vectors not totally cyclic
```

```python
sage: gale_transform_to_primal(
.....:    [(1,1,0), (-1,-1,0), (1,0,0),
.....:     (-1,0,0), (1,-1,0), (-2,1,0)], backend='field')
Traceback (most recent call last):
...
ValueError: input vectors not totally cyclic
```

`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])  # abs tol 1e-15
[(-0.7071067811865475)]
```

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])  # abs tol 1e-15
[(-0.7071067811865475)]
```

```python
sage: project_points([1,2,3])  # abs tol 1e-15
[(-0.7071067811865475, -1.2247448713915892)]
```

Check that it is (almost) an isometry:

```python
sage: V = list(map(vector, IntegerVectors(n=5, length=3)))
sage: P = project_points(V)
```
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()
```

```python
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 \((1,-1,0,\ldots,0)/\sqrt{2}\), \((1,1,-1,0,\ldots,0)/\sqrt{3}\), \ldots, \((1,1,\ldots,1,-1)/\sqrt{d}\) 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:**

```python
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`:

```python
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

\[
P = \text{conv}\{v_1,\ldots,v_k\} + \sum_{i=1}^{m} \mathbb{R} r_i + \sum_{j=1}^{n} \mathbb{R} \ell_j
\]
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: [f.ambient_V_indices() for f in strip.faces(1)]
[(0, 2), (0, 1)]
sage: for face in strip.faces(1):
....:     print(face.ambient_V_indices())
(0, 0)
```

(continues on next page)
(0, 2)
(0, 1)
sage: for face in strip.faces(1):
    ....: print("{} = "\(\{}\).format(face.ambient_V_indices(), face.as_polyhedron().
    ...Vrepresentation()))
(0, 2) = (A line in the direction (0, 1), A vertex at (1, 0))
(0, 1) = (A line in the direction (0, 1), A vertex at (-1, 0))

EXAMPLES:

sage: trunc_quadr = Polyhedron(vertices=[[1,0],[0,1]], rays=[[1,0],[0,1]])
sage: trunc_quadr
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 2 vertices and 2 rays
sage: v = next(trunc_quadr.vertex_generator())  # the first vertex in the internal enumeration
sage: v
A vertex at (0, 1)
sage: v.vector()
(0, 1)
sage: list(v)
[0, 1]
sage: len(v)
2
sage: v[0] + v[1]
1
sage: v.is_vertex()
True
sage: type(v)
<class 'sage.geometry.polyhedron.representation.Vertex'>
sage: type( v() )
<type 'sage.modules.vector_rational_dense.Vector_rational_dense'>
sage: v.polyhedron()
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 2 vertices and 2 rays
sage: r = next(trunc_quadr.ray_generator())
sage: r
A ray in the direction (0, 1)
sage: r.vector()
(0, 1)
sage: list( v.neighbors() )
[A ray in the direction (0, 1), A vertex at (1, 0)]

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.

2.1. Polyhedra
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)

triangle.Hrepresentation()
```

Without specifying the `base_ring`, the `sqrt(3)` would be a symbolic ring element and, therefore, the polyhedron defined over the symbolic ring. This is currently not supported as SR is not exact:

```sage
Polyhedron([(0,0), (1,0), (1/2, sqrt(3)/2)])
Traceback (most recent call last):
  ... ValueError: no default backend for computations with Symbolic Ring
```

Even faster than all algebraic real numbers (the field `AA`) is to take the smallest extension field. For the equilateral triangle, that would be:

```sage
K.<sqrt3> = NumberField(x^2 - 3, embedding=AA(3)**(1/2))

Polyhedron([(0,0), (1,0), (1/2, sqrt3/2)])
```

A 2-dimensional polyhedron in (Number Field in sqrt3 with defining polynomial $x^2 - 3$ with sqrt3 = 1.732050807568878?)^2 defined as the convex hull of 3 vertices

**Warning:** Be careful when you construct polyhedra with floating point numbers. The only available backend for such computation is cdd which uses machine floating point numbers which have have limited precision. If the input consists of floating point numbers and the `base_ring` is not specified, the base ring is set to be the `RealField` with the precision given by the minimal bit precision of the input. Then, if the obtained minimum is 53 bits of precision, the constructor converts automatically the base ring to `RDF`. Otherwise, it returns an error:

```sage
Polyhedron(vertices = [[1.12345678901234, 2.12345678901234]])
```

A 0-dimensional polyhedron in RDF^2 defined as the convex hull of 1 vertex

```sage
Polyhedron(vertices = [[1.123456789012345, 2.123456789012345]])
```

A 0-dimensional polyhedron in RDF^2 defined as the convex hull of 1 vertex

```sage
Polyhedron(vertices = [[1.123456789012345, 2.123456789012345]])
```

A 0-dimensional polyhedron in RDF^2 defined as the convex hull of 1 vertex

```sage
Traceback (most recent call last):
  ... ValueError: the only allowed inexact ring is 'RDF' with backend 'cdd'
```

The strongly suggested method to input floating point numbers is to specify the `base_ring` to be `RDF`: 

```sage
Polyhedron(vertices = [[1.12345678901234, 2.12345678901234]])
```
```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:

- Arnaud Bergeron: improvements to triangulation and rendering, 2008
- Sebastien Barthelemy: documentation improvements, 2008
- Volker Braun: reifying, 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

```python
sage.geometry.polyhedron.constructor.Polyhedron(vertices=None, rays=None, lines=None, ieqs=None, eqns=None, ambient_dim=None, base_ring=None, minimize=True, verbose=False, backend=None, mutable=False)
```

Construct a polyhedron object.

You may either define it with vertex/ray/line or inequalities/equations data, but not both. Redundant data will automatically be removed (unless `minimize=False`), and the complementary representation will be computed.

**INPUT:**

- **vertices** – list of points. Each point can be specified as any iterable container of `base_ring` elements. If `rays` or `lines` are specified but no `vertices`, the origin is taken to be the single vertex.

- **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 `7x1 + 3x2 + 4x3 ≥ 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 `7x1 + 3x2 + 4x3 = 1`.

- **base_ring** – a sub-field of the reals implemented in Sage. The field over which the polyhedron will be defined. For `QQ` and algebraic extensions, exact arithmetic will be used. For `RDF`, floating point numbers will be used. Floating point arithmetic is faster but might give the wrong result for degenerate input.

- **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 `Q` or `R` coefficients depending on `base_ring`
  - `'normaliz'`: use normaliz (`backend_normaliz`) with `Z` or `Q` coefficients depending on `base_ring`
  - `'polymake'`: use polymake (`backend_polymake`) with `Q`, `R` or `QuadraticField` coefficients depending on `base_ring`
  - `'ppl'`: use ppl (`backend_ppl`) with `Z` or `Q` coefficients depending on `base_ring`
  - `'field'`: use python implementation (`backend_field`) for any field

Some backends support further optional arguments:

- **minimize** – boolean (default: `True`); whether to immediately remove redundant H/V-representation data; currently not used.

- **verbose** – boolean (default: `False`); whether to print verbose output for debugging purposes; only supported by the cdd and normaliz backends

- **mutable** – boolean (default: `False`); whether the polyhedron is mutable

**OUTPUT:**

The polyhedron defined by the input data.

**EXAMPLES:**

Construct some polyhedra:
```python
sage: square_from_vertices = Polyhedron(vertices = [[1, 1], [1, -1], [-1, 1], [-1, -1]])
sage: square_from_ieqs = Polyhedron(ieqs = [[1, 0, 1], [1, 1, 0], [1, 0, -1], [1, -1, 0]])
sage: list(square_from_ieqs.vertex_generator())
[(A vertex at (1, -1),
  A vertex at (1, 1),
  A vertex at (-1, 1),
  A vertex at (-1, -1)]

sage: list(square_from_vertices.inequality_generator())
[An inequality (1, 0) x + 1 >= 0,
 An inequality (0, 1) x + 1 >= 0,
 An inequality (-1, 0) x + 1 >= 0,
 An inequality (0, -1) x + 1 >= 0]

sage: p = Polyhedron(vertices = [[1.1, 2.2], [3.3, 4.4]], base_ring=RDF)
sage: p.n_inequalities()
2

The same polyhedron given in two ways:

```python
sage: p = Polyhedron(ieqs = [[0,1,0,0],[0,0,1,0]])
sage: p.Vrepresentation()
(A line in the direction (0, 0, 1),
 A ray in the direction (1, 0, 0),
 A ray in the direction (0, 1, 0),
 A vertex at (0, 0, 0))

sage: q = Polyhedron(vertices=[[0,0,0]], rays=[[1,0,0],[0,1,0]], lines=[[0,0,1]])
sage: q.Hrepresentation()
(An inequality (1, 0, 0) x + 0 >= 0,
 An inequality (0, 1, 0) x + 0 >= 0)
```

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, 0, 1, 0, 0, 0, 0],
    ....:   [0, 0, 0, 1, 0, 0, 0], [0, 0, 0, 0, 1, 0, 0],
    ....:   [0, 0, 0, 0, 0, 1, 0], [0, 0, 0, 0, 0, 0, 1]])
sage: P = Polyhedron(ieqs=positive_coords.inequalities() + (
    ....:   [0,0,1,-1,-1,1,0], [0,0,-1,1,-1,1,0]), eqns=[[31,1,1,1,1,1,1]])
sage: P
A 5-dimensional polyhedron in QQ^6 defined as the convex hull of 7 vertices
sage: P.dim()
5
sage: P.Vrepresentation()
(A vertex at (31, 0, 0, 0, 0, 0), A vertex at (0, 0, 0, 0, 0, 31),
 A vertex at (0, 0, 0, 31, 0, 0), A vertex at (0, 31, 0, 0, 0, 0),
 A vertex at (31, 31, 0, 0, 0, 0), A vertex at (0, 31, 31, 0, 0, 0),
 A vertex at (0, 31, 0, 31, 0, 0))
```

Regular icosahedron, centered at 0 with edge length 2, with vertices given by the cyclic shifts of $(0, \pm 1, \pm(1 +
\(\sqrt{5})/2\), cf. Wikipedia article Regular icosahedron. It needs a number field:

```
sage: R0.<r0> = QQ[]
sage: R1.<r1> = NumberField(r0^2-5, embedding=AA(5)**(1/2))
sage: gold = (1+r1)/2
sage: v = [[0, 1, gold], [0, 1, -gold], [0, -1, gold], [0, -1, -gold]]
sage: pp = Permutation((1, 2, 3))
sage: icosah = Polyhedron([(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:

```
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
```

Create a mutable polyhedron:

```
sage: P = Polyhedron(vertices=[[0, 1], [1, 0]], mutable=True)
sage: P.is_mutable()
True
sage: hasattr(P, "_Vrepresentation")
False
sage: P.Vrepresentation()
(A vertex at (0, 1), A vertex at (1, 0))
sage: hasattr(P, "_Vrepresentation")
True
```

**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

Construct a suitable parent class for polyhedra

**INPUT:**

- `base_ring` – A ring. Currently there are backends for \(\mathbb{Z}\), \(\mathbb{Q}\), and \(\mathbb{R}\).
- `ambient_dim` – integer. The ambient space dimension.
- `ambient_space` – A free module.
- `backend` – string. The name of the backend for computations. There are several backends implemented:
  - `backend="ppl"` uses the Parma Polyhedra Library
  - `backend="cdd"` uses CDD
  - `backend="normaliz"` uses normaliz
  - `backend="polymake"` uses polymake
  - `backend="field"` a generic Sage implementation

**OUTPUT:**

A parent class for polyhedra over the given base ring if the backend supports it. If not, the parent base ring can be larger (for example, \(\mathbb{Q}\) instead of \(\mathbb{Z}\)). 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: 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'>
sage: Polyhedra(ZZ, 3, backend='cdd')
Polyhedra in QQ^3
```

CDD does not support integer polytopes directly:

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

Using a more general form of the constructor:

```python
sage: V = VectorSpace(QQ, 3)
sage: Polyhedra(V) is Polyhedra(QQ, 3)
True
sage: Polyhedra(V, backend='field') is Polyhedra(QQ, 3, 'field')
True
sage: Polyhedra(backend='field', ambient_space=V) is Polyhedra(QQ, 3, 'field')
True
```

(continues on next page)
sage: M = FreeModule(ZZ, 2)
sage: Polyhedra(M, backend='ppl') is Polyhedra(ZZ, 2, 'ppl')
True

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:

```
sage: from sage.geometry.polyhedron.parent import Polyhedra
c sage: 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: from sage.geometry.polyhedron.parent import Polyhedra
c sage: Polyhedra(ZZ, 2).Hrepresentation_space()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
```

Vrepresentation_space()
Return the ambient vector space.

This is the vector space or module containing the Vrepresentation vectors.

OUTPUT:
A free module over the base ring of dimension \(\text{ambient_dim()}\).

EXAMPLES:

```
sage: from sage.geometry.polyhedron.parent import Polyhedra
c sage: Polyhedra(QQ, 4).Vrepresentation_space()
Vector space of dimension 4 over Rational Field
c sage: Polyhedra(QQ, 4).ambient_space()
Vector space of dimension 4 over Rational Field
```

ambient_dim()
Return the dimension of the ambient space.

EXAMPLES:

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

ambient_space()
Return the ambient vector space.

This is the vector space or module containing the Vrepresentation vectors.

OUTPUT:
A free module over the base ring of dimension \(\text{ambient_dim()}\).

EXAMPLES:

```
sage: from sage.geometry.polyhedron.parent import Polyhedra
c sage: Polyhedra(QQ, 4).Vrepresentation_space()
Vector space of dimension 4 over Rational Field
```

(continues on next page)
sage: Polyhedra(QQ, 4).ambient_space()
Vector space of dimension 4 over Rational Field

**an_element()**

Return a Polyhedron.

**EXAMPLES:**

```python
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:**

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

**base_extend**(base_ring, backend=None, ambient_dim=None)

Return the base extended parent.

**INPUT:**

- base_ring, backend – see Polyhedron().
- ambient_dim – if not None change ambient dimension accordingly.

**EXAMPLES:**

```python
sage: from sage.geometry.polyhedron.parent import Polyhedra
sage: Polyhedra(ZZ,3).base_extend(QQ)
Polyhedra in QQ^3
sage: Polyhedra(ZZ,3).an_element().base_extend(QQ)
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 4 vertices
sage: Polyhedra(QQ, 2).base_extend(ZZ)
Polyhedra in QQ^2
```

**change_ring**(base_ring, backend=None, ambient_dim=None)

Return the parent with the new base ring.

**INPUT:**

- base_ring, backend – see Polyhedron().
- ambient_dim – if not None change ambient dimension accordingly.

**EXAMPLES:**

```python
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
sage: Polyhedra(QQ, 2).change_ring(ZZ)
Polyhedra in QQ^2
```

(continues on next page)
'cdd'
sage: Polyhedra(QQ, 3).change_ring(ZZ, ambient_dim=4)
Polyhedra in ZZ^4
sage: Polyhedra(QQ, 3, backend='cdd').change_ring(QQ, ambient_dim=4).backend()
'cdd'

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

list()
Return the two polyhedra in ambient dimension 0, raise an error otherwise

EXAMPLES:

sage: from sage.geometry.polyhedron.parent import Polyhedra
sage: P = Polyhedra(QQ, 3)
sage: P.cardinality()  # +Infinity
sage: P = Polyhedra(AA, 0)
sage: P.category()  # Category of finite enumerated polyhedral sets over Algebraic Real Field
sage: P.list()  # [The empty polyhedron in AA^0, a 0-dimensional polyhedron in AA^0 defined as the convex hull of 1 vertex]
sage: P.cardinality()  # 2

recycle(polyhedron)
Recycle the H/V-representation objects of a polyhedron.

This speeds up creation of new polyhedra by reusing objects. After recycling a polyhedron object, it is not in a consistent state any more and neither the polyhedron nor its H/V-representation objects may be used any more.

INPUT:

• polyhedron – a polyhedron whose parent is self.

EXAMPLES:

sage: p = Polyhedron([(0,0),(1,0),(0,1)])
sage: p.parent().recycle(p)

some_elements()
Return a list of some elements of the semigroup.

EXAMPLES:
sage: from sage.geometry.polyhedron.parent import Polyhedra
sage: Polyhedra(QQ, 4).some_elements()

[A 3-dimensional polyhedron in \(\mathbb{Q}^4\) defined as the convex hull of 4 vertices,
A 4-dimensional polyhedron in \(\mathbb{Q}^4\) defined as the convex hull of 1 vertex and \(4\) rays,
A 2-dimensional polyhedron in \(\mathbb{Q}^4\) defined as the convex hull of 2 vertices and \(1\) ray,
The empty polyhedron in \(\mathbb{Q}^4\)]

sage: Polyhedra(ZZ, 0).some_elements()

[The empty polyhedron in \(\mathbb{Z}^0\),
A 0-dimensional polyhedron in \(\mathbb{Z}^0\) defined as the convex hull of 1 vertex]

```
universe()

Return the entire ambient space as polyhedron.

EXAMPLES:

sage: from sage.geometry.polyhedron.parent import Polyhedra
sage: P = Polyhedra(QQ, 4)
sage: P.universe()
A 4-dimensional polyhedron in \(\mathbb{Q}^4\) defined as the convex hull of 1 vertex and \(4\) lines
sage: P.universe().is_universe()
True
```

```
zero()

Return the polyhedron consisting of the origin, which is the neutral element for Minkowski addition.

EXAMPLES:

sage: from sage.geometry.polyhedron.parent import Polyhedra
sage: p = Polyhedra(QQ, 4).zero(); p
A 0-dimensional polyhedron in \(\mathbb{Q}^4\) defined as the convex hull of 1 vertex
sage: 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_normaliz(base_ring, ambient_dim, backend)

Bases: sage.geometry.polyhedron.parent.Polyhedra_base

Element

alias of sage.geometry.polyhedron.backend_normaliz.Polyhedron_normaliz

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

sage.geometry.polyhedron.parent.does_backend_handle_base_ring(base_ring, backend)

Return true, if backend can handle base_ring.

EXAMPLES:
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: Return False for any equation.

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[0,0,0],[1,1,0],[1,2,0]])
sage: v = next(p.vertex_generator())
sage: v
A vertex at (0, 0, 0)
sage: a = next(p.equation_generator())
sage: a
An equation (0, 0, 1) x + 0 == 0
sage: a.interior_contains(v)
False
```

is_equation()
Tests if this object is an equation. By construction, it must be.
type()
Return the type (equation/inequality/vertex/ray/line) as an integer.

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

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[0,0,0],[1,1,0],[1,2,0]])
sage: repr_obj = next(p.equation_generator())
sage: repr_obj.type()
1
sage: repr_obj.type() == repr_obj.INEQUALITY
False
sage: repr_obj.type() == repr_obj.EQUATION
True
sage: repr_obj.type() == repr_obj.VERTEX
False
sage: repr_obj.type() == repr_obj.RAY
False
sage: repr_obj.type() == repr_obj.LINE
False
```

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

EXAMPLES:

```
sage: p = Polyhedron(ieqs = [[0,1,0],[0,0,1],[1,-1,0],[1,0,-1]])
sage: pH = p.Hrepresentation(2)
sage: pH.A()
(1, 0)
```

adjacent()
Alias for neighbors().

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

EXAMPLES:

```
sage: p = Polyhedron(ieqs = [[0,1,0],[0,0,1],[1,-1,0],[1,0,-1]])
sage: pH = p.Hrepresentation(2)
sage: pH.b()
0
```

eval(Vobj)
Evaluate the left hand side \( A \vec{x} + b \) 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: [ 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()

Return a generator for the incident H-representation objects, that is, the vertices/rays/lines satisfying the (in)equality.

EXAMPLES:

sage: triangle = Polyhedron(vertices=[[1,0],[0,1],[-1,-1]])
sage: ineq = next(triangle.inequality_generator())
sage: ineq
An inequality (2, -1) x + 1 >= 0

sage: [ v for v in ineq.incident() ]
[A vertex at (-1, -1), A vertex at (0, 1)]

sage: p = Polyhedron(vertices=[[0,0,0],[0,1,0],[0,0,1]], rays=[[1,-1,-1]])
sage: ineq = p.Hrepresentation(2)
sage: ineq
An inequality (1, 0, 1) x + 0 >= 0

sage: [ x for x in ineq.incident() ]
[A vertex at (0, 0, 0),
A vertex at (0, 1, 0),
A ray in the direction (1, -1, -1)]

is_H()

Return True if the object is part of a H-representation (inequality or equation).

EXAMPLES:

sage: p = Polyhedron(ieqs = [[0,1,0],[0,0,1],[1,-1,0],[1,0,-1]])
sage: pH = p.Hrepresentation(0)
sage: pH.is_H()
True

is_equation()

Return True if the object is an equation of the H-representation.

EXAMPLES:

sage: p = Polyhedron(ieqs = [[0,1,0],[0,0,1],[1,-1,0],[1,0,-1]], eqns = [[1,1,-1]])
sage: pH = p.Hrepresentation(0)
sage: pH.is_equation()
True
is_incident(Vobj)
Return whether the incidence matrix element (Vobj,self) == 1

EXAMPLES:
```
sage: p = Polyhedron(ieqs = [[0,0,0,1],[0,0,1,0],[0,1,0,0],
,...: [1,-1,0,0],[1,0,-1,0],[1,0,0,-1]])
sage: pH = p.Hrepresentation(0)
sage: pH.is_incident(p.Vrepresentation(1))
True
sage: pH.is_incident(p.Vrepresentation(5))
False
```

is_inequality()
Return True if the object is an inequality of the H-representation.

EXAMPLES:
```
sage: p = Polyhedron(ieqs = [[0,1,0],[0,0,1],[1,-1,0],[1,0,-1]])
sage: pH = p.Hrepresentation(0)
sage: pH.is_inequality()
True
```

neighbors()
Iterate over the adjacent facets (i.e. inequalities).
Only defined for inequalities.

EXAMPLES:
```
sage: p = Polyhedron(ieqs = [[0,0,0,1],[0,0,1,0],[0,1,0,0],
,...: [1,-1,0,0],[1,0,-1,0],[1,0,0,-1]])
sage: pH = p.Hrepresentation(0)
sage: a = list(pH.neighbors())
sage: a[0]
An inequality (0, -1, 0) x + 1 >= 0
sage: list(a[0])
[1, 0, -1, 0]
```

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():
```
(continues on next page)
....:
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:

sage: p = polytopes.cross_polytope(3)
sage: i1 = next(p.inequality_generator())
sage: [i1.contains(q) for q in p.vertex_generator()]
[True, True, True, True, True, True]
sage: p2 = 3*polytopes.hypercube(3)
sage: [i1.contains(q) for q in p2.vertex_generator()]
[True, True, False, True, False, True, False, False]

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

EXAMPLES:

sage: p = polytopes.cross_polytope(3)
sage: i1 = next(p.inequality_generator())
sage: [i1.interior_contains(q) for q in p.vertex_generator()]
[False, True, True, False, False, True]
sage: p2 = 3*polytopes.hypercube(3)
sage: [i1.interior_contains(q) for q in p2.vertex_generator()]
[True, True, False, True, False, True, False, False]

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

sage: P = Polyhedron(vertices=[[1,1],[1,-1],[-1,1],[-1,-1]])
sage: p = vector(ZZ, [1,0])
sage: [ ieq.interior_contains(p) for ieq in P.inequality_generator() ]
[True, True, False, True]

is_facet_defining_inequality(other)
Check if self defines a facet of other.

INPUT:

• other – a polyhedron

See also:

slack_matrix() incidence_matrix()
```
sage: P = Polyhedron(vertices=[[0,0,0],[0,1,0]], rays=[[1,0,0]])
sage: P.inequalities()
(An inequality (1, 0, 0) x + 0 >= 0, An inequality (0, 1, 0) x + 0 >= 0, An inequality (0, -1, 0) x + 1 >= 0)
sage: Q = Polyhedron(ieqs=[[0,1,0,0]])
sage: Q.inequalities()[0].is_facet_defining_inequality(P)
True
sage: Q = Polyhedron(ieqs=[[0,2,0,3]])
sage: Q.inequalities()[0].is_facet_defining_inequality(P)
True
sage: Q = Polyhedron(ieqs=[[0,AA(2).sqrt(),0,3]])
sage: Q.inequalities()[0].is_facet_defining_inequality(P)
True
sage: Q = Polyhedron(ieqs=[[1,1,0,0]])
sage: Q.inequalities()[0].is_facet_defining_inequality(P)
False
```

```
sage: P = Polyhedron(vertices=[[0,0,0],[0,1,0]], lines=[[1,0,0]])
sage: P.inequalities()
(An inequality (0, 1, 0) x + 0 >= 0, An inequality (0, -1, 0) x + 1 >= 0)
sage: Q = Polyhedron(ieqs=[[0,1,0,0]])
sage: Q.inequalities()[0].is_facet_defining_inequality(P)
False
sage: Q = Polyhedron(ieqs=[[0,-1,0,0]])
sage: Q.inequalities()[0].is_facet_defining_inequality(P)
False
sage: Q = Polyhedron(ieqs=[[0,0,1,3]])
sage: Q.inequalities()[0].is_facet_defining_inequality(P)
True
```

**is_inequality()**

Return True since this is, by construction, an inequality.

**EXAMPLES:**

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

**outer_normal()**

Return the outer normal vector of self.

**OUTPUT:**

The normal vector directed away from the interior of the polyhedron.

**EXAMPLES:**

```
sage: p = Polyhedron(vertices = [[0,0,0],[1,1,0],[1,2,0]])
sage: a = next(p.inequality_generator())
sage: a.outer_normal()
(1, -1, 0)
```
type()
Return the type (equation/inequality/vertex/ray/line) as an integer.

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

EXAMPLES:
```
sage: p = Polyhedron(vertices = [[0,0,0],[1,1,0],[1,2,0]])
sage: repr_obj = next(p.inequality_generator())
sage: repr_obj.type()
0
sage: repr_obj.type() == repr_obj.INEQUALITY
True
sage: repr_obj.type() == repr_obj.EQUATION
False
sage: repr_obj.type() == repr_obj.VERTEX
False
sage: repr_obj.type() == repr_obj.RAY
False
sage: repr_obj.type() == repr_obj.LINE
False
```

```python
class sage.geometry.polyhedron.representation.Line(polyhedron_parent)
Bases: sage.geometry.polyhedron.representation.Vrepresentation
A line (Minkowski summand \( \mathbb{R} \)) of the polyhedron. Inherits from Vrepresentation.

evaluated_on(Hobj)
Return \( A \ell \)

EXAMPLES:
```
sage: p = Polyhedron(ieqs = [[1,0,0,1],[1,1,0,0]])
sage: a = next(p.line_generator())
sage: h = next(p.inequality_generator())
sage: a.evaluated_on(h)
0
```

homogeneous_vector(base_ring=None)
Return homogeneous coordinates for this line.
Since a line is given by a direction, this is the vector with a 0 appended.

INPUT:
- base_ring – the base ring of the vector.

EXAMPLES:
```
sage: P = Polyhedron(vertices=[[2,0]], rays=[[1,0]], lines=[[3,2]])
sage: P.lines()[0].homogeneous_vector()
(3, 2, 0)
sage: P.lines()[0].homogeneous_vector(RDF)
(3.0, 2.0, 0.0)
```

is_line()
Tests if the object is a line. By construction it must be.
type()
Return the type (equation/inequality/vertex/ray/line) as an integer.

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

EXAMPLES:

```
sage: p = Polyhedron(ieqs = [[1, 0, 0, 1],[1,1,0,0]])
sage: repr_obj = next(p.line_generator())
sage: repr_obj.type()
4
sage: repr_obj.type() == repr_obj.INEQUALITY
False
sage: repr_obj.type() == repr_obj.EQUATION
False
sage: repr_obj.type() == repr_obj.VERTEX
False
sage: repr_obj.type() == repr_obj.RAY
False
sage: repr_obj.type() == repr_obj.LINE
True
```

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 inequalities/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:

```
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()
Return an arbitrary but fixed number according to the internal storage order.

Note: H-representation and V-representation objects are enumerated independently. That is, amongst all vertices/rays/lines there will be one with index() == 0, and amongst all inequalities/equations there will be
one with \( \text{index()} = 0 \), 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()**

Return the underlying polyhedron.

**vector**\((\text{base\_ring}=\text{None})\)

Return the vector representation of the H/V-representation object.

**INPUT:**

- \( \text{base\_ring} \) – 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:

```python
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:

```python
sage: v.vector(RDF)
(-1.0, -1.0, 0.0)
sage: vector(RDF, v)
(-1.0, -1.0, 0.0)
```

**class** \( \text{sage.geometry.polyhedron.representation.Ray}(\text{polyhedron\_parent}) \)

Bases: \( \text{sage.geometry.polyhedron.representation.Vrepresentation} \)

A ray of the polyhedron. Inherits from \( V\text{representation} \).

**evaluated\_on\( (\text{Hobj})\)**

Return \( A\vec{r} \)

EXAMPLES:
sage: p = Polyhedron(ieqs = [[0,0,1],[0,1,0],[1,-1,0]])
sage: a = next(p.ray_generator())
sage: h = next(p.inequality_generator())
sage: a.evaluated_on(h)
0

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()

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

OUTPUT:

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

EXAMPLES:

sage: p = Polyhedron(ieqs = [[0,0,1],[0,1,0],[1,-1,0]])
sage: repr_obj = next(p.ray_generator())
sage: repr_obj.type()
3
sage: repr_obj.type() == repr_obj.INEQUALITY
False
sage: repr_obj.type() == repr_obj.EQUATION
False
sage: repr_obj.type() == repr_obj.VERTEX
False
sage: repr_obj.type() == repr_obj.RAY
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)

Return $A\vec{x} + b$

**Examples:**

```python
sage: p = polytopes.hypercube(3)
sage: v = next(p.vertex_generator())
sage: h = next(p.inequality_generator())
sage: v
A vertex at (1, -1, -1)
sage: h
An inequality (1, 0, 0) x + 1 >= 0
sage: v.evaluated_on(h)
0
```

**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:**

```python
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:**

```python
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:**

```python
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()
Return the type (equation/inequality/vertex/ray/line) as an integer.

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

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[0,0,0],[1,1,0],[1,2,0]])
sage: repr_obj = next(p.vertex_generator())
sage: repr_obj.type()
2
sage: repr_obj.type() == repr_obj.INEQUALITY
False
sage: repr_obj.type() == repr_obj.EQUATION
False
sage: repr_obj.type() == repr_obj.VERTEX
True
sage: repr_obj.type() == repr_obj.RAY
False
sage: repr_obj.type() == repr_obj.LINE
False
```

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()
Return a generator for the equations/inequalities that are satisfied on the given vertex/ray/line.

EXAMPLES:

```
sage: triangle = Polyhedron(vertices=[[1,0],[0,1],[-1,-1]])
sage: ineq = next(triangle.inequality_generator())
sage: ineq
An inequality (2, -1) x + 1 >= 0
sage: [ v for v in ineq.incident() ]
[A vertex at (-1, -1), A vertex at (0, 1)]
sage: p = Polyhedron(vertices=[[0,0,0],[0,1,0],[0,0,1]], rays=[[1,-1,-1]])
sage: ineq = p.Hrepresentation(2)
sage: ineq
An inequality (1, 0, 1) x + 0 >= 0
sage: [ x for x in ineq.incident() ]
[A vertex at (0, 0, 0), A vertex at (0, 1, 0), A ray in the direction (1, -1, -1)]
```

is_V()
Return True if the object is part of a V-representation (a vertex, ray, or line).

EXAMPLES:
```python
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)
Return whether the incidence matrix element (self,Hobj) == 1

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

is_line()
Return True if the object is a line of the V-representation. This method is over-ridden by the corresponding method in the derived class Line.

EXAMPLES:
```
```
```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
sage: v1 = next(p.vertex_generator())
sage: v1.is_line()
False
```

is_ray()
Return True if the object is a ray of the V-representation. This method is over-ridden by the corresponding method in the derived class Ray.

EXAMPLES:
```
```
```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
sage: v1 = next(p.vertex_generator())
sage: v1
A vertex at (-1, -1, 0, -1)
sage: v1.is_ray()
False
```

is_vertex()
Return True if the object is a vertex of the V-representation. This method is over-ridden by the corresponding method in the derived class Vertex.

EXAMPLES:
\begin{verbatim}
sage: p = Polyhedron(vertices = [[0,0],[1,0],[0,3],[1,3]])
sage: v = next(p.vertex_generator())
sage: v.is_vertex()
True
sage: p = Polyhedron(ieqs = [[1, 0, 0, 0, 1], [1, 1, 0, 0, 0], [1, 0, 1, 0, 0]])
sage: r1 = next(p.ray_generator())
sage: r1.is_vertex()
False
\end{verbatim}

\textbf{neighbors()}

Return a generator for the adjacent vertices/rays/lines.

\textbf{EXAMPLES:}

\begin{verbatim}
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)
\end{verbatim}

\textbf{sage.geometry.polyhedron.representation.repr_pretty}(coefficients, type, prefix='x', indices=None, latex=False, style='>=' or '<=', split=False)

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

\textbf{INPUT:}

\begin{itemize}
  \item coefficients – a tuple or other iterable
  \item type – either 0 (PolyhedronRepresentation.INEQUALITY) or 1 (PolyhedronRepresentation.EQUATION)
  \item prefix – a string
  \item indices – a tuple or other iterable
  \item latex – a boolean
  \item 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.
  \item style – either "positive" (making all coefficients positive), or "<=" or ">=".
\end{itemize}

\textbf{OUTPUT:}

A string or 3-tuple of strings (depending on split).

\textbf{EXAMPLES:}

\begin{verbatim}
sage: from sage.geometry.polyhedron.representation import repr_pretty
sage: from sage.geometry.polyhedron.representation import PolyhedronRepresentation
sage: print(repr_pretty((0, 1, 0, 0), PolyhedronRepresentation.INEQUALITY))
x0 >= 0
sage: print(repr_pretty((1, 2, 1, 0), PolyhedronRepresentation.INEQUALITY))
2*x0 + x1 >= -1
sage: print(repr_pretty((1, -1, -1, 1), PolyhedronRepresentation.EQUATION))
-x0 - x1 + x2 == -1
\end{verbatim}
2.1.5 Functions for plotting polyhedra

class sage.geometry.polyhedron.plot.Projection(polyhedron, proj=<function projection_func_identity at 0x7f0c56f9d5e0>)

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
sage: p = polytopes.hypercube(3)
sage: proj = p.projection()
sage: proj.coord_index_of(vector((1,1,1)))
2
```

coord_indices_of(v_list)

Convert list of coordinate vectors to the corresponding list of internal indices.

EXAMPLES:

```sage
sage: p = polytopes.hypercube(3)
sage: proj = p.projection()
sage: proj.coord_indices_of([vector((1,1,1)),vector((1,-1,1))])
[2, 3]
```

coordinates_of(coord_index_list)

Given a list of indices, return the projected coordinates.

EXAMPLES:

```sage
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:

```sage
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=None, line_opts=None, polygon_opts=None)

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

INPUT:

See plot().

OUTPUT:
A 2-d graphics object.

**EXAMPLES:**

```python
sage: print(Polyhedron([]).projection().render_0d().description())
```

```python
sage: print(Polyhedron(ieqs=[(1,)]).projection().render_0d().description())
```

*Point set defined by 1 point(s): [(0.0, 0.0)]*

---

**render_1d** (*point_opts=None, line_opts=None, polygon_opts=None*)

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

**INPUT:**

See `plot()`.

**OUTPUT:**

A 2-d graphics object.

**EXAMPLES:**

```python
sage: Polyhedron([0, 0, 0]).projection().render_1d()
```

*Graphics object consisting of 2 graphics primitives*

---

**render_2d** (*point_opts=None, line_opts=None, polygon_opts=None*)

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

**EXAMPLES:**

```python
sage: p1 = Polyhedron(vertices=[[1,1],[1,0],[0,1],[0,0]])
sage: q1 = p1.projection()
sage: p2 = Polyhedron(vertices=[[1,0],[0,1],[0,0]], rays=[[1,0],[0,1]])
sage: p3 = Polyhedron(vertices=[[1,2]])
sage: p4 = Polyhedron(vertices=[[2,0],[0,2],[0,0]], lines=[[1,0],[0,1]])
sage: q1.plot() + q2.plot() + q3.plot() + q4.plot()
```

*Graphics object consisting of 17 graphics primitives*

---

**render_3d** (*point_opts=None, line_opts=None, polygon_opts=None*)

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

**EXAMPLES:**

```python
sage: p1 = Polyhedron(vertices=[[1,1,1],[1,1,0],[1,0,1],[0,1,1]])
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*

```python
sage: Polyhedron(ieqs=[[1,0,0],[0,1,0],[0,0,0,1]]).plot() # half space
```

(continues on next page)
Graphics3d Object

sage: Polyhedron(vertices=[[1,1,1]],
 lines=[[0,1,0],[0,0,1]]).plot()  # R^2 in R^3
Graphics3d Object

sage: Polyhedron(rays=[[0,1,0],[0,0,1]], 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

˓→ Half-line

Graphics3d Object

sage: Polyhedron(rays=[[0,1,0]]).plot()
# point in R^3

˓→ point in R^3

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

sage: Q = Polyhedron([[100],[101]])
sage: P = Q*Q*Q; P
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 8 vertices
sage: p = P.plot()
sage: p.bounding_box()
((100.0, 100.0, 100.0), (101.0, 101.0, 101.0))

render_fill_2d(**kwds)
Retrun the filled interior (a polygon) of a polyhedron in 2d.

EXAMPLES:

sage: cps = [i^3 for i in srange(-2,2,1/5)]
sage: p = Polyhedron(vertices = [[(t^2-1)/(t^2+1),2*t/(t^2+1)] for t in cps])
sage: proj = p.projection()
sage: filled_poly = proj.render_fill_2d()
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:

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:

```
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:

```
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:

```
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:

```
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:

```
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:

sage: cube = polytopes.hypercube(3)
sage: cube_proj = cube.projection()
sage: wire = cube_proj.render_wireframe_3d()
sage: print(wire.tachyon().split('n')[77]) # for testing
FCylinder base 1.0 1.0 -1.0 apex -1.0 1.0 -1.0 rad 0.005 texture...

schlegel(facet=None, position=None)
Return the Schlegel projection.

• The facet is orthonormally transformed into its affine hull.
• The position specifies a point coming out of the barycenter of the facet from which the other vertices will be projected into the facet.

INPUT:

• facet – a PolyhedronFace. The facet into which the Schlegel diagram is created. The default is the first facet.
• position – a positive number. Determines a relative distance from the barycenter of facet. A value close to 0 will place the projection point close to the facet and a large value further away. If the given value is too large, an error is returned. If no position is given, it takes the midpoint of the possible point of views along a line spanned by the barycenter of the facet and a valid point outside the facet.

EXAMPLES:

sage: cube4 = polytopes.hypercube(4)
sage: from sage.geometry.polyhedron.plot import Projection
sage: Projection(cube4).schlegel()
The projection of a polyhedron into 3 dimensions
sage: _.plot()
Graphics3d Object

The 4-cube with a truncated vertex seen into the resulting tetrahedron facet:

sage: tcube4 = cube4.face_truncation(cube4.faces(0)[0])
sage: tcube4.facets()[4]
A 3-dimensional face of a Polyhedron in QQ^4 defined as the convex hull of 4 vertices
sage: into_tetra = Projection(tcube4).schlegel(tcube4.facets()[4])
sage: into_tetra.plot()
Graphics3d Object

Taking a larger value for the position changes the image:

sage: into_tetra_far = Projection(tcube4).schlegel(tcube4.facets()[4], 4)
sage: into_tetra_far.plot()
Graphics3d Object
A value which is too large or negative give a projection point that sees more than one facet resulting in an error:

```
sage: Projection(tcube4).schlegel(tcube4.facets()[4],5)
Traceback (most recent call last):
...  
ValueError: the chosen position is too large

sage: Projection(tcube4).schlegel(tcube4.facets()[4],-1)
Traceback (most recent call last):
...  
ValueError: 'position' should be a positive number
```

**stereographic**

Return the stereographic projection.

**INPUT:**

- `projection_point` - The projection point. This must be distinct from the polyhedron’s vertices. Default is 
  $(1,0,...,0)$

**EXAMPLES:**

```
sage: from sage.geometry.polyhedron.plot import Projection
sage: proj = Projection(polytopes.buckyball()) #long time
sage: proj  

The projection of a polyhedron into 3 dimensions
```

```
sage: proj.stereographic([5,2,3]).plot()  
Graphics object consisting of 123 graphics primitives
```

```
sage: Projection( polytopes.twenty_four_cell() ).stereographic([2,0,0,0])
```

The projection of a polyhedron into 3 dimensions

**tikz**

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

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: 1) 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 by visualizing it using `.show(aspect_ratio=1)`. This will open an interactive view in your default browser, where you can rotate the polytope. Once the desired view angle is found, click on the information icon in the lower right-hand corner and select `Get
Viewpoint. This will copy a string of the form \('[x,y,z].angle' to your local clipboard. Go back to Sage and type 
\texttt{Image = P.projection().tikz([x,y,z],angle)}.

The inputs \texttt{view} and \texttt{angle} can also be obtained from the viewer Jmol:

1) Right click on the image
2) Select `Console`.
3) Select the tab `State`.
4) Scroll to the line `moveto`.

It reads something like:

\texttt{moveto 0.0 \{x y z angle\} Scale}

The \texttt{view} is then \([x,y,z]\) and \texttt{angle} is angle. The following number is the scale.

Jmol performs a rotation of \texttt{angle} degrees along the vector \([x,y,z]\) and show the result from the z-axis.

\textbf{EXAMPLES:}

\begin{verbatim}
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('\n'.join(Image1.splitlines()[:4]))
\begin{tikzpicture}[
x={(-0.939161cm, 0.244762cm)},
y={(0.097442cm, -0.482887cm)},
z={(0.329367cm, 0.840780cm)}],
sage: with open('polytope-tikz1.tex', 'w') as f:
    _ = f.write(Image1)
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: print('\n'.join(Image2.splitlines()[:4]))
\begin{tikzpicture}[
[scale=3.000000, 
back/.style={loosely dotted, thin},
edge/.style={color=blue!95!black, thick},
sage: with open('polytope-tikz2.tex', 'w') as f:
    _ = f.write(Image2)
sage: P3 = Polyhedron(vertices=[[-1, -1, 2], [-1, 2, -1], [2, -1, -1]])
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('\n'.join(Image3.splitlines()[:4]))
\begin{tikzpicture}[
[x={(-0.939161cm, 0.244762cm)},
\end{verbatim}

(continues on next page)
\begin{verbatim}
y={(-0.096240cm, 0.912008cm)},
z={(-0.746680cm, -0.331036cm)},
sage: with open('polytope-tikz3.tex', 'w') as f:  # not tested
    _ = f.write(Image3)
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.
\end{verbatim}

Todo:
Make it possible to draw Schlegel diagram for 4-polytopes.

\begin{verbatim}
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.
\end{verbatim}

Make it possible to draw 3-polytopes living in higher dimension.

\begin{class}
\begin{code}
\begin{verbatim}
class sage.geometry.polyhedron.plot.ProjectionFuncSchlegel(facet, projection_point)
    Bases: object
    The Schlegel projection from the given input point.

    EXAMPLES:

    sage: from sage.geometry.polyhedron.plot import ProjectionFuncSchlegel
    sage: f = polytopes.hypercube(4)
    sage: facet = f.facets()[0]
    sage: proj = ProjectionFuncSchlegel(facet, [0, 1, 0, 0])
    sage: proj([0,0,0,0])
    1.0
\end{verbatim}
\end{code}
\end{class}

\begin{class}
\begin{code}
\begin{verbatim}
class sage.geometry.polyhedron.plot.ProjectionFuncStereographic(projection_point)
    Bases: object
    The stereographic (or perspective) projection onto a codimension-1 linear subspace with respect to a sphere centered at the origin.

    EXAMPLES:

    sage: from sage.geometry.polyhedron.plot import ProjectionFuncStereographic
    sage: c = polytopes.hypercube(3)
    sage: facet = c.facets()[0]
    sage: proj = ProjectionFuncStereographic([1,2,3])
    sage: proj([0,0,0])
    \end{verbatim}
\end{code}
\end{class}

\end{verbatim}
sage: ppoints[5]
(-0.0918273..., -0.036375...)

sage.geometry.polyhedron.plot.cyclic_sort_vertices_2d(Vlist)
  Return the vertices/rays in cyclic order if possible.

**Note:** This works if and only if each vertex/ray is adjacent to exactly two others. For example, any 2-dimensional polyhedron satisfies this.

See `vertex_adjacency_matrix()` for a discussion of “adjacent”.

**EXAMPLES:**

```sage
def cyclic_sort_vertices_2d(Vlist):
    # Implementation details...
```
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()

\[
\begin{bmatrix} 0 & 0 & 1 \\
0 & 0 & 0 \\
1 & 0 & 0 \\
\end{bmatrix}
\]

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)]

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

EXAMPLES:

\[
\begin{bmatrix} 1, 2, 3 \\
\end{bmatrix}
\]

2.1.6 A class to keep information about faces of a polyhedron

This module gives you a tool to work with the faces of a polyhedron and their relative position. First, you need to find the faces. To get the faces in a particular dimension, use the face() method:

sage: P = polytopes.cross_polytope(3)
sage: P.faces(3)
(A 3-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 6 vertices,)

sage: [f.ambient_V_indices() for f in P.facets()]
[(3, 4, 5),
(2, 4, 5),
(1, 3, 5),
(1, 2, 5),
(0, 3, 4),
(0, 2, 4),
(0, 1, 3),
(0, 1, 2)]

sage: [f.ambient_V_indices() for f in P.faces(1)]
[(4, 5),
(3, 5),
(2, 5),
(1, 5),
(3, 4),
(2, 4),
(0, 4),]
or `face_lattice()` to get the whole face lattice as a poset:

```python
sage: P.face_lattice()
Finite lattice containing 28 elements
```

The faces are printed in shorthand notation where each integer is the index of a vertex/ray/line in the same order as the containing Polyhedron’s `Vrepresentation()`

```python
sage: face = P.faces(1)[8]; face
A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2 vertices
sage: face.ambient_V_indices()
(0, 3)
sage: P.Vrepresentation(0)
A vertex at (-1, 0, 0)
sage: P.Vrepresentation(3)
A vertex at (0, 0, 1)
sage: face.vertices()
(A vertex at (-1, 0, 0), A vertex at (0, 0, 1))
```

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
sage: _.equations()
(An equation (0, 1, 0) x + 0 == 0,
An equation (1, 0, -1) x + 1 == 0)
```

```python
class sage.geometry.polyhedron.face.PolyhedronFace(polyhedron, V_indices, H_indices)

Bases: sage.geometry.convex_set.ConvexSet_closed

A face of a polyhedron.

This class is for use in `face_lattice()`.

INPUT:

No checking is performed whether the H/V-representation indices actually determine a face of the polyhedron. You should not manually create `PolyhedronFace` objects unless you know what you are doing.

OUTPUT:

A `PolyhedronFace`.

EXAMPLES:

```python
sage: octahedron = polytopes.cross_polytope(3)
sage: inequality = octahedron.Hrepresentation(2)
sage: face_h = tuple([ inequality ])
```
sage: face_v = tuple( inequality.incident() )
sage: face_h_indices = [ h.index() for h in face_h ]
sage: face_v_indices = [ v.index() for v in face_v ]
sage: from sage.geometry.polyhedron.face import PolyhedronFace
sage: face = PolyhedronFace(octahedron, face_v_indices, face_h_indices)
sage: face
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3 → vertices
sage: face.dim()
2
sage: face.ambient_V_indices()
(0, 1, 2)
sage: face.ambient_Hrepresentation()
(An inequality (1, 1, 1) x + 1 >= 0,)
sage: face.ambient_Vrepresentation()
(A vertex at (-1, 0, 0), A vertex at (0, -1, 0), A vertex at (0, 0, -1))

affine_tangent_cone()

Return the affine tangent cone of self as a polyhedron.

It is equal to the sum of self and the cone of feasible directions at any point of the relative interior of self.

OUTPUT:
A polyhedron.

EXAMPLES:

sage: half_plane_in_space = Polyhedron(ieqs=[(0,1,0,0)], eqns=[(0,0,0,1)])
sage: line = half_plane_in_space.faces(1)[0]; line
A 1-dimensional face of a Polyhedron in QQ^3 defined as the convex hull of 1 vertex and 1 line
sage: T_line = line.affine_tangent_cone()
sage: T_line == half_plane_in_space
True
sage: c = polytopes.cube()
sage: edge = min(c.faces(1))
sage: edge.vertices()
(A vertex at (1, -1, -1), A vertex at (1, 1, -1))
sage: T_edge = edge.affine_tangent_cone()
sage: T_edge.Vrepresentation()
(A line in the direction (0, 1, 0),
A ray in the direction (0, 0, 1),
A vertex at (1, 0, -1),
A ray in the direction (-1, 0, 0))

ambient()

Return the containing polyhedron.

EXAMPLES:

sage: P = polytopes.cross_polytope(3); P
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 6 vertices
sage: face = P.facets()[3]
sage: face
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3→vertices
sage: face.polyhedron()
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 6 vertices

ambient_H_indices()

Return the indices of the H-representation objects of the ambient polyhedron that make up the H-representation of self.

See also ambient_Hrepresentation().

OUTPUT:

Tuple of indices

EXAMPLES:

sage: Q = polytopes.cross_polytope(3)
sage: F = Q.faces(1)
sage: [f.ambient_H_indices() for f in F]
[(4, 5),
 (5, 6),
 (4, 7),
 (6, 7),
 (0, 5),
 (3, 4),
 (0, 3),
 (1, 6),
 (0, 1),
 (2, 7),
 (2, 3),
 (1, 2)]

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)
ambient_V_indices()
Return the indices of the V-representation objects of the ambient polyhedron that make up the V-representation of self.

See also ambient_Vrepresentation().

OUTPUT:
Tuple of indices

EXAMPLES:

```sage
P = polytopes.cube()
sage: F = P.faces(2)
sage: [f.ambient_V_indices() for f in F]
[(0, 3, 4, 5),
 (0, 1, 5, 6),
 (4, 5, 6, 7),
 (2, 3, 4, 7),
 (1, 2, 6, 7),
 (0, 1, 2, 3)]
```

ambient_Vrepresentation(index=None)

Return the V-representation objects of the ambient polytope defining the face.

INPUT:

- index – 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:

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

EXAMPLES:

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

ambient_vector_space(base_field=None)
Return the ambient vector space.

It is the ambient free module of the containing polyhedron tensored with a field.

INPUT:
• base_field – (default: the fraction field of the base ring) a field.

EXAMPLES:

```python
sage: half_plane = Polyhedron(ieqs=[(0,1,0)])
```
```python
sage: line = half_plane.faces(1)[0]; line
A 1-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 1 vertex and 1 line
```
```python
sage: line.ambient_vector_space()
Vector space of dimension 2 over Rational Field
```
```python
sage: line.ambient_vector_space(AA)
Vector space of dimension 2 over Algebraic Real Field
```

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
```
```python
sage: face = P.faces(2)[3]
```
```python
sage: face
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3 vertices
```
```python
sage: face.as_polyhedron()
A 2-dimensional polyhedron in ZZ^3 defined as the convex hull of 3 vertices
```
```python
sage: P.intersection(face.as_polyhedron()) == face.as_polyhedron()
True
```
contains(point)
Test whether the polyhedron contains the given point.

INPUT:
• point – a point or its coordinates

EXAMPLES:

```
sage: half_plane = Polyhedron(ieqs=[(0,1,0)])
sage: line = half_plane.faces(1)[0]; line
A 1-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 1→vertex and 1 line
sage: line.contains([0, 1])
True
```

As a shorthand, one may use the usual in operator:

```
sage: [5, 7] in line
False
```

dim()
Return the dimension of the face.

OUTPUT:
Integer.

EXAMPLES:

```
sage: fl = polytopes.dodecahedron().face_lattice()
sage: sorted([ x.dim() for x in fl ])  
[-1, 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, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3]
```

is_compact()
Return whether self is compact.

OUTPUT:
Boolean.

EXAMPLES:

```
sage: half_plane = Polyhedron(ieqs=[(0,1,0)])
sage: line = half_plane.faces(1)[0]; line
A 1-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 1→vertex and 1 line
sage: line.is_compact()
False
```

is_relatively_open()
Return whether self is relatively open.

OUTPUT:
Boolean.

EXAMPLES:
sage: half_plane = Polyhedron(ieqs=[(0,1,0)])
sage: line = half_plane.faces(1)[0]; line
A 1-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 1 vertex and 1 line
sage: line.is_relatively_open()
True

line_generator()

Return a generator for the lines of the face.

EXAMPLES:

sage: pr = Polyhedron(rays=[[1,0],[-1,0],[0,1]], vertices=[[1,-1]])
sage: face = pr.faces(1)[0]
sage: next(face.line_generator())
A line in the direction (1, 0)

lines()

Return all lines of the face.

OUTPUT:
A tuple of lines.

EXAMPLES:

sage: p = Polyhedron(rays=[[1,0],[-1,0],[0,1],[1,1]], vertices=[[1,-1],[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:

sage: p = polytopes.cross_polytope(4)
sage: face = p.face_lattice()[5]
sage: face
A 1-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 2 vertices
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().

2.1. Polyhedra 147
OUTPUT:
Integer.

EXAMPLES:

```sage
p = polytopes.cross_polytope(4)
f = p.face_lattice()[5]
f
```

```
A 1-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 2 --> vertices
```

```sage
c = f.ambient_Vrepresentation()
c
```

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

```sage
c = f.n_ambient_Vrepresentation()
c
```

```
2
```

**n_lines()**

Return the number of lines of the face.

OUTPUT:
Integer.

EXAMPLES:

```sage
p = Polyhedron(rays = [[1,0],[-1,0],[0,1],[1,1]], vertices = [[-2,-2],[2,3]])
q = p.n_lines()
```

```
1
```

**n_rays()**

Return the number of rays of the face.

OUTPUT:
Integer.

EXAMPLES:

```sage
p = Polyhedron(ieqs = [[0,0,0,1],[0,0,1,0],[1,1,0,0]])
f = p.faces(2)[0]
f.n_rays()
```

```
2
```

**n_vertices()**

Return the number of vertices of the face.

OUTPUT:
Integer.

EXAMPLES:

```sage
c = polytopes.cross_polytope(3)
f = c.faces(2)[0]
f.n_vertices()
```

```
3
```

**normal_cone(direction='outer')**

Return the polyhedral cone consisting of normal vectors to hyperplanes supporting `self`. 
INPUT:

- **direction** – string (default: 'outer'), the direction in which to consider the normals. The other allowed option is 'inner'.

OUTPUT:

A polyhedron.

EXAMPLES:

```python
sage: p = Polyhedron(vertices = [[1,2],[2,1],[-2,2],[-2,-2],[2,-2]])
```

```python
sage: for v in p.face_generator(0):
    ....:     vect = v.vertices()[0].vector()
    ....:     nc = v.normal_cone().rays_list()
    ....:     print("{} has outer normal cone spanned by ".format(vect,nc))
    ....:
(2, 1) has outer normal cone spanned by [[1, 0], [1, 1]]
(1, 2) has outer normal cone spanned by [[0, 1], [1, 1]]
(2, -2) has outer normal cone spanned by [[0, -1], [1, 0]]
(-2, -2) has outer normal cone spanned by [[-1, 0], [0, -1]]
(-2, 2) has outer normal cone spanned by [[-1, 0], [0, 1]]
```

```python
sage: for v in p.face_generator(0):
    ....:     vect = v.vertices()[0].vector()
    ....:     nc = v.normal_cone(direction='inner').rays_list()
    ....:     print("{} has inner normal cone spanned by ".format(vect,nc))
    ....:
(2, 1) has inner normal cone spanned by [[-1, -1], [-1, 0]]
(1, 2) has inner normal cone spanned by [[-1, -1], [0, -1]]
(2, -2) has inner normal cone spanned by [[-1, 0], [0, 1]]
(-2, -2) has inner normal cone spanned by [[0, 1], [1, 0]]
(-2, 2) has inner normal cone spanned by [[0, -1], [1, 0]]
```

The function works for polytopes that are not full-dimensional:

```python
sage: p = polytopes.permutahedron(3)
```

```python
sage: f1 = p.faces(0)[0]
sage: f2 = p.faces(1)[0]
sage: f3 = p.faces(2)[0]
sage: f1.normal_cone()
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 1 vertex, 2 rays, 1 line
```

```python
sage: f2.normal_cone()
A 2-dimensional polyhedron in ZZ^3 defined as the convex hull of 1 vertex, 1 ray, 1 line
```

```python
sage: f3.normal_cone()
A 1-dimensional polyhedron in ZZ^3 defined as the convex hull of 1 vertex and 1 line
```

Normal cones are only defined for non-empty faces:

```python
sage: f0 = p.faces(-1)[0]
sage: f0.normal_cone()
Traceback (most recent call last):
```

(continues on next page)

... Value Error: the empty face does not have a normal cone

polyhedron()
Return the containing polyhedron.

EXAMPLES:
sage: P = polytopes.cross_polytope(3); P
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 6 vertices
sage: face = P.facets()[3]
sage: face
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3 vertices
sage: face.polyhedron()
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 6 vertices

ray_generator()
Return a generator for the rays of the face.

EXAMPLES:
sage: pi = Polyhedron(ieqs = [[1,1,0],[1,0,1]])
sage: face = pi.faces(1)[1]
sage: next(face.ray_generator())
A ray in the direction (1, 0)

rays()
Return the rays of the face.

OUTPUT:
A tuple of rays.

EXAMPLES:
sage: p = Polyhedron(ieqs = [[0,0,0,1],[0,0,1,0],[1,1,0,0]])
sage: face = p.faces(2)[2]
sage: face.rays()
(A ray in the direction (1, 0, 0), A ray in the direction (0, 1, 0))

stacking_locus()
Return the polyhedron containing the points that sees every facet containing self.

OUTPUT:
A polyhedron.

EXAMPLES:
sage: cp = polytopes.cross_polytope(4)
sage: facet = cp.facets()[0]
sage: facet.stacking_locus().vertices()
(A vertex at (1/2, 1/2, 1/2, 1/2),
 A vertex at (1, 0, 0, 0),
 A vertex at (0, 0, 0, 1),
 A vertex at (0, 0, 1, 0),

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

```python
sage: face = cp.faces(2)[0]
sage: face.stacking_locus().vertices()
(A vertex at (0, 1, 0, 0),
 A vertex at (0, 0, 1, 0),
 A vertex at (1, 0, 0, 0),
 A vertex at (1, 1, 1, 0),
 A vertex at (1/2, 1/2, 1/2, 1/2),
 A vertex at (1/2, 1/2, 1/2, -1/2))
```

**vertex_generator()**

Return a generator for the vertices of the face.

**EXAMPLES:**

```python
sage: triangle = Polyhedron(vertices=[[1,0],[0,1],[1,1]])
sage: face = triangle.facets()[0]
sage: for v in face.vertex_generator(): print(v)
A vertex at (1, 0)
A vertex at (1, 1)
sage: type(face.vertex_generator())
<... 'generator'>
```

**vertices()**

Return all vertices of the face.

**OUTPUT:**

A tuple of vertices.

**EXAMPLES:**

```python
sage: triangle = Polyhedron(vertices=[[1,0],[0,1],[1,1]])
sage: face = triangle.faces(1)[2]
sage: face.vertices()
(A vertex at (0, 1), A vertex at (1, 0))
```

```python
sage.geometry.polyhedron.face.combinatorial_face_to_polyhedral_face(polyhedron, combinatorial_face)
```

Convert a combinatorial face to a face of a polyhedron.

**INPUT:**

- polyhedron – a polyhedron containing combinatorial_face
- combinatorial_face – a CombinatorialFace

**OUTPUT:** a PolyhedronFace.

**EXAMPLES:**

```python
sage: from sage.geometry.polyhedron.face import combinatorial_face_to_polyhedral_face
sage: P = polytopes.simplex()
sage: C = P.combinatorial_polyhedron()
sage: it = C.face_iter()
sage: comb_face = next(it)
```
sage: combinatorial_face_to_polyhedral_face(P, comb_face)
A 2-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 3 vertices

2.1.7 Generate cdd .ext / .ine file format

sage.geometry.polyhedron.cdd_file_format.cdd_Hrepresentation(cdd_type, ieqs, eqns, file_output=None)

Return a string containing the H-representation in cddlib’s ine format.

INPUT:

- file_output (string; optional) – a filename to which the representation should be written. If set to None (default), representation is returned as a string.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.cdd_file_format import cdd_Hrepresentation
sage: cdd_Hrepresentation('rational', None, [[0,1]])
'H-representation\nlinearity 1 1\nbegin\n 1 2 rational\n 0 1\nend'
```

sage.geometry.polyhedron.cdd_file_format.cdd_Vrepresentation(cdd_type, vertices, rays, lines, file_output=None)

Return a string containing the V-representation in cddlib’s ext format.

INPUT:

- file_output (string; optional) – a filename to which the representation should be written. If set to None (default), representation is returned as a string.

Note: If there is no vertex given, then the origin will be implicitly added. You cannot write the empty V-representation (which cdd would refuse to process).

EXAMPLES:

```
sage: from sage.geometry.polyhedron.cdd_file_format import cdd_Vrepresentation
sage: print(cdd_Vrepresentation('rational', [[0,0]], [[1,0]], [[0,1]]))
V-representation
linearity 1 1
begin
 3 3 rational
 0 0 1
 0 1 0
 1 0 0
end
```
2.1.8 Formal modules generated by polyhedra

class sage.geometry.polyhedron.modules.formal_polyhedra_module.FormalPolyhedraModule(base_ring, dimension, basis, category)

Bases: sage.combinat.free_module.CombinatorialFreeModule

Class for formal modules generated by polyhedra.

It is formal because it is free – it does not know about linear relations of polyhedra.

A formal polyhedral module is graded by dimension.

INPUT:

• base_ring – base ring of the module; unrelated to the base ring of the polyhedra
• dimension – the ambient dimension of the polyhedra
• basis – the basis

EXAMPLES:

```python
sage: from sage.geometry.polyhedron.modules.formal_polyhedra_module import FormalPolyhedraModule
sage: def closed_interval(a,b): return Polyhedron(vertices=[[a], [b]])

A three-dimensional vector space of polyhedra:

```sage```
I01 = closed_interval(0, 1); I01.rename("conv([0], [1])")
I11 = closed_interval(1, 1); I11.rename("{[1]}")
I12 = closed_interval(1, 2); I12.rename("conv([1], [2])")
basis = [I01, I11, I12]
M = FormalPolyhedraModule(QQ, 1, basis=basis); M
Free module generated by {conv([0], [1]), {[1]}, conv([1], [2])} over Rational Field
```
sage: M.get_order()
[conv([0], [1]), {[1]}, conv([1], [2])]
```sage```
A one-dimensional subspace; bases of subspaces just use the indexing set 0, . . . , 𝑑 − 1, where 𝑑 is the dimension:

```sage```
M_lower = M.submodule([M(I11)]); M_lower
Free module generated by {0} over Rational Field
```
sage: M_lower.print_options(prefix='S')
```
sage: M_lower.is_submodule(M)
True
```
sage: x = M(I01) - 2*M(I11) + M(I12)
```
sage: M_lower.reduce(x)
[conv([0], [1]), conv([1], [2])]
```
sage: M_lower.retract.domain() is M
True
```
sage: y = M_lower.retract(M(I11)); y
S[0]
```
sage: M_lower.lift(y)
[{{1}}]
```
Quotient space; bases of quotient space are families indexed by elements of the ambient space:

```python
sage: M_mod_lower = M.quotient_module(M_lower); M_mod_lower
Free module generated by {conv([0], [1]), conv([1], [2])} over Rational Field
sage: M_mod_lower.print_options(prefix='Q')
sage: M_mod_lower.retract(x)
Q[conv([0], [1])] + Q[conv([1], [2])]
sage: M_mod_lower.retract(M(I01) - 2*M(I11) + M(I12)) == M_mod_lower.
   → retract(M(I01) + M(I12))
True
```

degree_on_basis(m)

The degree of an element of the basis is defined as the dimension of the polyhedron.

INPUT:

- m – an element of the basis (a polyhedron)

EXAMPLES:

```python
sage: from sage.geometry.polyhedron.modules.formal_polyhedra_module import FormalPolyhedraModule
sage: def closed_interval(a, b):
    return Polyhedron(vertices=[[a], [b]])

sage: I01 = closed_interval(0, 1); I01.rename("conv([0], [1])")
sage: I11 = closed_interval(1, 1); I11.rename("{[1]}")
sage: I12 = closed_interval(1, 2); I12.rename("conv([1], [2])")
sage: I02 = closed_interval(0, 2); I02.rename("conv([0], [2])")
sage: M = FormalPolyhedraModule(QQ, 1, basis=[I01, I11, I12, I02])
```

We can extract homogeneous components:

```python
sage: O = M(I01) + M(I11) + M(I12)
sage: O.homogeneous_component(0)

[[[1]]]
sage: O.homogeneous_component(1)
[conv([0], [1])] + [conv([1], [2])]
```

We note that modulo the linear relations of polyhedra, this would only be a filtration, not a grading, as the following example shows:

```python
sage: X = M(I01) + M(I12) - M(I02)
sage: X.degree()
1
sage: Y = M(I11)
sage: Y.degree()
0
```
## 2.2 Lattice polyhedra

### 2.2.1 Lattice and reflexive polytopes

This module provides tools for work with lattice and reflexive polytopes. A **convex polytope** is the convex hull of finitely many points in \( \mathbb{R}^n \). 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

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

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 this 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
- Andrey Novoseltsev (2010-12-15): new version of nef-partitions
sage.geometry.lattice_polytope.LatticePolytope\(\text{data, compute\_vertices=True, n=0, lattice=None}\)

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
class sage.geometry.lattice_polytope.LatticePolytopeClass

Bases: sage.geometry.convex_set.ConvexSetCompact, collections.abc.Hashable

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 \texttt{LatticePolytope()} to construct lattice polytopes.

Lattice polytopes are immutable, but they cache most of the returned values.

INPUT:

The input can be either:

- \texttt{points} – \texttt{PointCollection};
- \texttt{compute_vertices} – boolean.

or (these parameters must be given as keywords):

- \texttt{ambient} – ambient structure, this polytope \textit{must be a face of ambient};
- \texttt{ambient_vertex_indices} – increasing list or tuple of integers, indices of vertices of ambient generating this polytope;
- \texttt{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.

\textbf{adjacent()}

Return faces adjacent to \texttt{self} in the ambient face lattice.

Two distinct faces \( F_1 \) and \( F_2 \) of the same face lattice are \texttt{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:

\begin{verbatim}
\texttt{sage: o = lattice_polytope.cross_polytope(3)}
\texttt{sage: o.adjacent()} ()
\texttt{sage: face = o.faces(1)[0]}
\texttt{sage: face.adjacent()} (1-d face of 3-d reflexive polytope in 3-d lattice \( \mathbb{M} \),
1-d face of 3-d reflexive polytope in 3-d lattice \( \mathbb{M} \),
1-d face of 3-d reflexive polytope in 3-d lattice \( \mathbb{M} \),
1-d face of 3-d reflexive polytope in 3-d lattice \( \mathbb{M} \))
\end{verbatim}

\textbf{affine_transform}(a=1, b=0)

Return \( a \# P + b \), where \( P \) is this lattice polytope.

Note:
1. While \(a\) and \(b\) may be rational, the final result must be a lattice polytope, i.e. all vertices must be integral.

2. If the transform (restricted to this polytope) is bijective, facial structure will be preserved, e.g. the first facet of the image will be spanned by the images of vertices which span the first facet of the original polytope.

INPUT:

- \(a\) - (default: 1) rational scalar or matrix
- \(b\) - (default: 0) rational scalar or vector, scalars are interpreted as vectors with the same components

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(2)
sage: o.vertices()
M( 1, 0),
M( 0, 1),
M(-1, 0),
M( 0, -1)
in 2-d lattice M
sage: o.affine_transform(2).vertices()
M( 2, 0),
M( 0, 2),
M(-2, 0),
M( 0, -2)
in 2-d lattice M
sage: o.affine_transform(1,1).vertices()
M(2, 1),
M(1, 2),
M(0, 1),
M(1, 0)
in 2-d lattice M
sage: o.affine_transform(b=1).vertices()
M(2, 1),
M(1, 2),
M(0, 1),
M(1, 0)
in 2-d lattice M
sage: o.affine_transform(b=(1, 0)).vertices()
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),
```

(continues on next page)
While you can use rational transformation, the result must be integer:

```
sage: o.affine_transform(a)
Traceback (most recent call last):
  ...  
ValueError: points 
   [(1/2, 0), (0, 3/2), (-1/2, 0), (0, -3/2)]
are not in 2-d lattice M!
```

**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_dim()**

Return the dimension of the ambient lattice of self.

An alias is `ambient_dim()`.

**OUTPUT:**

- integer.

**EXAMPLES:**

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

**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.ambient_facet_indices()
(4, 5)
sage: face.ambient_facet_indices()
(4, 5)
sage: face.ambient_facet_indices()
(4, 5)
sage: face.ambient_facet_indices()
(4, 5)
sage: face.ambient_facet_indices()
(4, 5)
sage: face.ambient_facet_indices()
(4, 5)
```

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_point_indices()
(5, 8, 4, 9, 10, 11, 6, 12, 7)
sage: cube.points(face.ambient_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_vector_space(base_field=None)
Return the ambient vector space.

It is the ambient lattice (lattice()) tensored with a field.

INPUT:

\* base_field – (default: the rationals) a field.

EXAMPLES:

sage: p = LatticePolytope([(1,0)])
sage: p.ambient_vector_space()
Vector space of dimension 2 over Rational Field
sage: p.ambient_vector_space(AA)
Vector space of dimension 2 over Algebraic Real Field

ambient_vertex_indices()
Return indices of vertices of the ambient structure generating self.

OUTPUT:

\* increasing tuple of integers.

EXAMPLES:

sage: o = lattice_polytope.cross_polytope(3)
sage: o.ambient_vertex_indices()
(0, 1, 2, 3, 4, 5)
sage: face = o.faces(1)[0]
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),
(continues on next page)
For an edge the boundary is formed by the end points:

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

`contains(*args)`

Check if a given point is contained in `self`.

INPUT:

• an attempt will be made to convert all arguments into a single element of the ambient space of `self`; if it fails, `False` will be returned
OUTPUT:

- True if the given point is contained in self, False otherwise

EXAMPLES:

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

Distances from facets to the point (1,2,3):
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 0 2 1]
[0 0 2 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
sage: e.ambient_facet_indices() == ed.ambient_vertex_indices()
True
```

dual_lattice()
Return the dual of the ambient lattice of self.

OUTPUT:
• a lattice. If possible (that is, if \texttt{lattice()} has a \texttt{dual()} method), the dual lattice is returned. Otherwise, \(\mathbb{Z}^n\) is returned, where \(n\) is the dimension of \texttt{self}.

**EXAMPLES:**

```python
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
```

\textbf{edges()}

Return edges (faces of dimension 1) of \texttt{self}.

**OUTPUT:**

• tuple of \textit{lattice polytopes}.

**EXAMPLES:**

```python
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
```

\textbf{face_lattice()}

Return the face lattice of \texttt{self}.

This lattice will have the empty polytope as the bottom and this polytope itself as the top.

**OUTPUT:**

• \textit{finite poset} of \textit{lattice polytopes}.

**EXAMPLES:**

Let’s take a look at the face lattice of a square:

```python
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:

```python
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]
[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: sorted(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,
  0-d face of 2-d lattice polytope in 2-d lattice M)
sage: face.adjacent()[0].vertices()
M(1, 0)
in 2-d lattice M
```

Note that if p is a face of superp, then the face lattice of p consists of (appropriate) faces of superp:
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(dim=None, codim=None)
Return faces of self of specified (co)dimension.

INPUT:

• dim – integer, dimension of the requested faces;
• codim – integer, codimension of the requested faces.

Note: You can specify at most one parameter. If you don’t give any, then all faces will be returned.

OUTPUT:

• if either dim or codim is given, the output will be a tuple of lattice polytopes;
• if neither dim nor codim is given, the output will be the tuple of tuples as above, giving faces of all existing dimensions. If you care about inclusion relations between faces, consider using face_lattice(), adjacent(), facet_of(), and facets().

EXAMPLES:

Let’s take a look at the faces of a square:

sage: square = LatticePolytope([(0,0), (1,0), (1,1), (0,1)])
sage: square.faces()
((-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),
 (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),
 (2-d lattice polytope in 2-d lattice M),))

Its faces of dimension one (i.e., edges):
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):

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

Let’s pick a particular face:

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

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

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:

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

**facet_constant**(i)

Return the constant in the i-th facet inequality of this polytope.

This is equivalent to facet_constants()[i].

**INPUT:**

- i – integer; the index of the facet

**OUTPUT:**

- integer – the constant in the i-th facet inequality.

**See also:**

facet_constants(), facet_normal(), facet_normals(), facets().

**EXAMPLES:**

sage: o = lattice_polytope.cross_polytope(3)
sage: o.facet_constant(0)
1
sage: o.facet_constant(0) == o.facet_constants()[0]
True

**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:**

- \texttt{facet_constant()}
- \texttt{facet_normal()}
- \texttt{facet_normals()}
- \texttt{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)
```

**\texttt{facet_normal}(i)**

Return the inner normal to the \( i \)-th facet of this polytope.

This is equivalent to \texttt{facet_normals()[i]}.

**Input:**

- \( i \) – integer; the index of the facet

**Output:**

- a vector

**See also:**

- \texttt{facet_constant()}
- \texttt{facet_constants()}
- \texttt{facet_normals()}
- \texttt{facets()}

**Examples:**
facets()  
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.facet_normal(0)
N(1, -1, -1)
sage: o.facet_normal(0) is o.facet_normals()[0]
True
```

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()
N( 0, 3, 0, -1),
(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):

```
sage: matrix([[n * v + c for v in p.vertices()] for n, c in zip(p.facet_normals(), p.facet_constants())])
```

```
[0 6 0 0]
[0 0 2 0]
[3 0 0 0]
[0 0 0 6]
```

### 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
```

### 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
```

### incidence_matrix()

Return the incidence matrix.
Note: The columns correspond to facets/facet normals in the order of `facet_normals()`, the rows correspond to the vertices in the order of `vertices()`.

EXAMPLES:

```python
sage: o = lattice_polytope.cross_polytope(2)
sage: o.incidence_matrix()
[0 0 1 1]
[0 1 1 0]
[1 1 0 0]
[1 0 0 1]
sage: o.faces(1)[0].incidence_matrix()
[1 0]
[0 1]
```

```python
sage: o = lattice_polytope.cross_polytope(4)
sage: o.incidence_matrix().column(3).nonzero_positions()
[3, 4, 5, 6]
sage: o.facets()[3].ambient_vertex_indices()
(3, 4, 5, 6)
```

`index()`

Return the index of this polytope in the internal database of 2- or 3-dimensional reflexive polytopes. Databases are stored in the directory of the package.

Note: The first call to this function for each dimension can take a few seconds while the dictionary of all polytopes is constructed, but after that it is cached and fast.

**Return type** integer

EXAMPLES: We check what is the index of the “diamond” in the database:

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

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

```python
sage: d.vertices()
M( 1, 0),
M( 0, 1),
M(-1, 0),
M( 0, -1)
in 2-d lattice M
sage: lattice_polytope.ReflexivePolytope(2,3).vertices()
M( 1, 0),
M( 0, 1),
M( 0, -1),
M(-1, 0)
in 2-d lattice M
```

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

Return indices of (relative) interior lattice points of this polytope.

OUTPUT:

• increasing tuple of integers.

EXAMPLES:

The origin is the only interior point of this square:

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

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

`lattice_dim()`

Return the dimension of the ambient lattice of `self`.

An alias is `ambient_dim()`.

OUTPUT:

• integer.

EXAMPLES:
```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},
```

(continues on next page)
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:

```
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:

```
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:
The number of facets of an interval is 2:

```
sage: LatticePolytope(([1],[2])).nfacets()
sage: 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()
sage: 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()
sage: M( 1, 0),
M( 0, 1),
M(-1, 0),
M( 0, -1)
in 2-d lattice M
sage: d.normal_form()
sage: M( 1, 0),
(continues on next page)```
The diamond is the 3rd polytope in the internal database:

```sage```
d.index()
3
ds
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)])
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)
o.normal_form(algorithm="palp_native")
M( 1, 0),
M( 0, 1),
M( 0, -1),
M(-1, 0)
in 2-d lattice M
```

```
o = lattice_polytope.cross_polytope(2)
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:
```
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:

```
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:

```
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):

```
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(show_facets=True, facet_opacity=0.5, facet_color=(0, 1, 0), facet_colors=None, show_edges=True, edge_thickness=3, edge_color=(0.5, 0.5, 0.5), show_vertices=True, vertex_size=10, vertex_color=(1, 0, 0), show_points=True, point_size=10, point_color=(0, 0, 1), show_vindices=None, vindex_color=(0, 0, 0), vlabels=None, show_pindices=None, pindex_color=(0, 0, 0), index_shift=1.1)

Return a 3d-plot of this polytope.

Polytopes with ambient dimension 1 and 2 will be plotted along x-axis or in xy-plane respectively. Polytopes of dimension 3 and less with ambient dimension 4 and greater will be plotted in some basis of the spanned space.

By default, everything is shown with more or less pretty combination of size and color parameters.

INPUT: Most of the parameters are self-explanatory:

- show_facets - (default:True)
- facet_opacity - (default:0.5)
- facet_color - (default:(0,1,0))
- facet_colors - (default:None) if specified, must be a list of colors for each facet separately, used instead of facet_color
- show_edges - (default:True) whether to draw edges as lines
- edge_thickness - (default:3)
- edge_color - (default:(0,0,1))
- 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:None) if specified, must be a list of labels for each vertex, default labels are vertex indices
- 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:None) if specified, must be a list of labels for each vertex, default labels are vertex indices
- 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'))
```

Graphics3d Object

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"])
```

Graphics3d Object

**point(i)**

Return the i-th point of this polytope, i.e. the i-th column of the matrix returned by points().

EXAMPLES: First few points are actually vertices:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.vertices()
M( 1, 0, 0),
M( 0, 1, 0),
M( 0, 0, 1),
M(-1, 0, 0),
M( 0, -1, 0),
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)
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
```

**points(*args, **kwards)**

Return all lattice points of self.

INPUT:

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

OUTPUT:

• a point collection.

EXAMPLES:

Lattice points of the octahedron and its polar cube:

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

Lattice points of a 2-dimensional diamond in a 3-dimensional space:

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

```python
sage: p = LatticePolytope([[1]])
sage: p.points()
M(1)
in 1-d lattice M
```

**polar()**

Return the polar polytope, if this polytope is reflexive.

**EXAMPLES:** The polar polytope to the 3-dimensional octahedron:

```python
sage: o = lattice_polytope.cross_polytope(3)
sage: cube = o.polar()
sage: cube
3-d reflexive polytope in 3-d lattice M
```

The polar polytope “remembers” the original one:

```python
sage: cube.polar()
3-d reflexive polytope in 3-d lattice M
sage: cube.polar().polar() is cube
True
```

Only reflexive polytopes have polars:

```python
sage: p = LatticePolytope([(1,0,0), (0,1,0), (0,0,2),
....:    (-1,0,0), (0,-1,0), (0,0,-1)])
sage: p.polar()
Traceback (most recent call last):
...
ValueError: The given polytope is not reflexive!
Polytope: 3-d lattice polytope in 3-d lattice M
```

**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:

```python
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
 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, ...
```

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, ...
```

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, ...
```

`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:

```python
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
```
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),
in 3-d lattice N
```

class sage.geometry.lattice_polytope.NefPartition(data, Delta_polar, check=True)

Bases: sage.structure.sage_object.SageObject, collections.abc.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 \(\Delta\) 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 \(\Delta \subset M_R\) be a reflexive polytope with polar \(\Delta^\circ \subset N_R\). 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

2.2. Lattice polyhedra
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 dual duality of reflexive polytopes switches convex hull and Minkowski sum for dual nef-partitions:

\[
\Delta^\circ = \operatorname{Conv}(\nabla_0, \nabla_1, \ldots, \nabla_{k-1}), \\
\nabla = \nabla_0 + \nabla_1 + \cdots + \nabla_{k-1}, \\
\Delta = \Delta_0 + \Delta_1 + \cdots + \Delta_{k-1}, \\
\nabla^\circ = \operatorname{Conv}(\Delta_0, \Delta_1, \ldots, \Delta_{k-1}).
\]

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

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

The **Cayley cone** $C \subset M_\mathbb{R}$ of a nef-partition is the cone spanned by its Cayley polytope. The dual Cayley cone $C^\vee \subset M_\mathbb{R}$ 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:

```
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 $\Delta^\circ$ in the above description:

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

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

```
sage: np.dual()  
Nef-partition {0, 1} U {3, 4} U {5, 6, 7}  
sage: np.nabla_polar().vertices()  
N(-1, -1, 0),  
N(-1, 0, 0),  
```

(continues on next page)
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, ∇^o is ∆^o from the point of view of the dual nef-partition:

```python
sage: np.dual().Delta_polar() is np.nabla_polar()
True
sage: np.Delta(1).vertices()
N(0, 0, -1),
N(0, 0, 1)
in 3-d lattice N
sage: np.dual().nabla(1).vertices()
N(0, 0, -1),
N(0, 0, 1)
in 3-d lattice N
```

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 = lattice_polytope.cross_polytope(3)
sage: np = o.nef_partitions()
sage: np
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 Δ or Δ_i corresponding to self.

**INPUT:**

- i – an integer. If not given, Δ 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()
```
\( N(1, -1, -1), \)
\( N(1, 1, -1), \)
\( N(1, 1, 1), \)
\( N(1, -1, 1), \)
\( N(-1, -1, 1), \)
\( N(-1, -1, -1), \)
\( N(-1, 1, -1), \)
\( N(-1, 1, 1) \)
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 \( \Delta^0 \) corresponding to \( self \).

**OUTPUT:**

- a **lattice polytope**.

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

**EXAMPLES:**

```
sage: o = lattice_polytope.cross_polytope(3)
sage: np = 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:**

```
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()
\( N(1, -1, -1), \)
\( N(1, 1, -1), \)
\( N(1, 1, 1), \)
\( N(1, -1, 1), \)
\( N(-1, -1, 1), \)
\( N(-1, 1, -1), \)
\( N(-1, 1, 1) \)
```

(continues on next page)
N(-1, 1, -1),
N(-1, 1, 1)
in 3-d lattice N

sage: [Delta_i.vertices() for Delta_i in np.Deltas()]
[N(-1, -1, 0),
 N(-1, 0, 0),
 N( 1, 0, 0),
 N( 1, -1, 0)
in 3-d lattice N,
N(0, 0, -1),
N(0, 1, 1),
N(0, 0, 1),
N(0, 1, -1)
in 3-d lattice N]

sage: np.nabla_polar().vertices()
N(-1, -1, 0),
N( 1, -1, 0),
N( 1, 0, 0),
N(-1, 0, 0),
N( 0, 1, -1),
N( 0, 1, 1),
N( 0, 0, 1),
N( 0, 0, -1)
in 3-d lattice N

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: 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:

```python
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, $\nabla$ 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),
```

(continues on next page)
Return the polytope $\nabla$ 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),
N( 0, 0, -1)
in 3-d lattice N
sage: np.nabla_polar() is np.dual().Delta_polar()
True
```

Return the polytopes $\nabla_i$ 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_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: [nabla_i.vertices() for nabla_i in np.nablas()]
[M(-1, 0, 0),
 M( 1, 0, 0),
 M( 0, 1, 0)
in 3-d lattice M,
M(0, -1, 0),
M(0, 0, -1),
M(0, 0, 1)
in 3-d lattice M]

nparts()

Return the number of parts in self.

OUTPUT:

• an integer.

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.nparts()
2

part(i, all_points=False)

Return the i-th part of self.

INPUT:

• i – an integer

• all_points – (default: False) whether to list all lattice points or just vertices

OUTPUT:

• a tuple of integers, 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.part(0)
(0, 1, 3)
sage: np.part(0, all_points=True)
(0, 1, 3)
sage: np.dual().part(0)
(0, 1, 3)
sage: np.dual().part(0, all_points=True)
(0, 1, 2, 3, 8)
part_of(i)
Return the index of the part containing the i-th vertex.

INPUT:
• i – an integer.

OUTPUT:
• an integer j such that the i-th vertex of Δ° belongs to V_j.

See nef-partition class documentation for definitions and notation.

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: np = 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
```

part_of_point(i)
Return the index of the part containing the i-th point.

INPUT:
• i – an integer.

OUTPUT:
• an integer j such that the i-th point of Δ° belongs to ∇_j.

Note: Since a nef-partition induces a partition on the set of boundary lattice points of Δ°, the value of j is well-defined for all i but the one that corresponds to the origin, in which case this method will raise a ValueError exception. (The origin always belongs to all ∇_j.)

See nef-partition class documentation for definitions and notation.

EXAMPLES:

We consider a relatively complicated reflexive polytope #2252 (easily accessible in Sage as ReflexivePolytope(3, 2252), we create it here explicitly to avoid loading the whole database):

```
sage: p = LatticePolytope([(1,0,0), (0,1,0), (0,0,1), (0,1,-1),
....: (0,-1,1), (-1,1,0), (0,-1,-1), (-1,-1,0), (-1,-1,2)])
sage: np = p.nef_partitions()[0]
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 \texttt{np}:

\begin{verbatim}
sage: [n for n in range(p.npoints())]
[1, 2, 5, 7, 8, 9, 11, 13]
sage: [n for n in range(p.npoints())]
[0, 3, 4, 6, 10, 12]
\end{verbatim}

\texttt{parts(all_points=False)}

Return all parts of \texttt{self}.

**INPUT:**

- \texttt{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 \(\Sigma V_i\)

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

**EXAMPLES:**

\begin{verbatim}
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()
((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))
\end{verbatim}

\texttt{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 \(\text{dim} = 2\) and \(0 \leq n \leq 4318\) for \(\text{dim} = 3\).
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 \texttt{ReflexivePolytopes(dim)[n]} but checks bounds first.
EXAMPLES:

The 3rd 2-dimensional polytope is “the diamond”:

```
sage: ReflexivePolytope(2, 3)
2-d reflexive polytope #3 in 2-d lattice M
sage: lattice_polytope.ReflexivePolytope(2,3).vertices()
M( 1, 0),
M( 0, 1),
M( 0, -1),
M(-1, 0)
in 2-d lattice M
```

There are 16 reflexive polygons and numeration starts with 0:

```
sage: ReflexivePolytope(2,16)
Traceback (most recent call last):
... 
ValueError: there are only 16 reflexive polygons!
```

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

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

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

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

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

```
class sage.geometry.lattice_polytope.SetOfAllLatticePolytopesClass
    Bases: sage.structure.parent.Set_generic
```

2.2. Lattice polyhedra
sage.geometry.lattice_polytope.all_cached_data(polytopes)
Compute all cached data for all given polytopes and their polars.

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. 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: 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.

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:
```python
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 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:

```python
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),
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:

```python
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: 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: 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: 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:**

```python
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), ...
\quad (-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),
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]
```

(continues on next page)
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:

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

```sage
lattice_polytope.positive_integer_relations(cm)
```

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

```sage
lm = ReflexivePolytope(2,1).vertices().column_matrix()
lattice_polytope.positive_integer_relations(lm)
```

```sage
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:

```sage
d = lattice_polytope.cross_polytope(2)
o = lattice_polytope.cross_polytope(3)
result_name = lattice_polytope._palp("poly.x -fe", [d, o])
with open(result_name) as f:
    print(f.read())
```

```
4 2 Vertices of P-dual <-> Equations of P
-1 1
 1 1
-1 -1
 1 -1
8 3 Vertices of P-dual <-> Equations of P
-1 -1 1
 1 -1 1
-1 1 1
 1 1 1
-1 -1 -1
 1 -1 -1
-1 1 -1
 1 1 -1
```

```sage
lattice_polytope.read_all_polytopes(result_name)
```

```sage
os.remove(result_name)
```

```sage
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 \texttt{matrix()}.

**INPUT:**

- \texttt{data} – Either a string containing the filename or the file itself containing the output by PALP.
- \texttt{permutation} – (default: \texttt{False}) If \texttt{True}, try to retrieve the permutation output by PALP. This parameter makes sense only when PALP computed the normal form of a lattice polytope.

**OUTPUT:**

A matrix or a tuple of a matrix and a permutation.

**EXAMPLES:**

```
sage: lattice_polytope.read_palp_matrix("2 3 comment 
\1 2 3 \n4 5 6")
[1 2 3]
[4 5 6]
sage: lattice_polytope.read_palp_matrix("3 2 Will be transposed \n\1 2 \n3 4 \n5 6 ...")
[1 3 5]
[2 4 6]
```

**2.2. Lattice polyhedra**

**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 \texttt{None}.

**OUTPUT:**

- \texttt{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! 
Output:
Please increase POLY_Dmax to at least 7 
```

However, we can work with this polytope by changing PALP dimension to 11:

```
sage: lattice_polytope.set_palp_dimension(11)
sage: p._palp("poly.x -fv") 
'7 14 Vertices of P...'
```
Let’s go back to default settings:

```python
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:

```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])
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
sage: f = open(result_name)
sage: lattice_polytope.skip_palp_matrix(f)
sage: lattice_polytope.read_palp_matrix(f)
[-1  1 -1  1 -1  1 -1  1]
[-1 -1  1  1 -1 -1  1  1]
[ 1  1  1 -1 -1 -1 -1  1]
sage: f.close()
sage: os.remove(result_name)
```

`sage.geometry.lattice_polytope.write_palp_matrix(m, ofile=None, comment="", format=None)`
Write m into ofile in PALP format.

**INPUT:**

- `m` – a matrix over integers or a point collection.
- `ofile` – a file opened for writing (default: stdout)
- `comment` – 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 0 -1 0
 0 0 1 0 0 -1
sage: lattice_polytope.write_palp_matrix(o.vertices(), format="%4d")
3 6
 1 0 0 -1 0 0
 0 1 0 0 -1 0
 0 0 1 0 0 -1
```

### 2.2.2 Lattice Euclidean Group Elements

The classes here are used to return particular isomorphisms of PPL lattice polytopes.

```python
class sage.geometry.polyhedron.lattice_euclidean_group_element.LatticeEuclideanGroupElement(A, b)
```

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:

```python
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_PPL,
    C_Polyhedron
sage: from sage.geometry.polyhedron.lattice_euclidean_group_element import LatticeEuclideanGroupElement

sage: M = LatticeEuclideanGroupElement([[1,2],[2,3],[-1,2]], [1,2,3])
sage: M
The map A*x+b with A=
[[ 1  2]
 [ 2  3]
[-1  2]]
b =
(1, 2, 3)
sage: M._A
[[ 1  2]
 [ 2  3]
[-1  2]]
sage: M._b
(1, 2, 3)
sage: M(vector([0,0]))
(1, 2, 3)
sage: M(LatticePolytope_PPL((0,0),(1,0),(0,1)))
```

(continues on next page)
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:

```
sage: zero_map = LatticeEuclideanGroupElement([[0,0],[0,0],[0,0]], [0,0,0])
sage: zero_map.codomain_dim()
3
```

domain_dim()

Return the dimension of the domain lattice

**EXAMPLES:**

```
sage: from sage.geometry.polyhedron.lattice_euclidean_group_element import LatticeEuclideanGroupElement
sage: M = LatticeEuclideanGroupElement([[1,2],[2,3],[-1,2]], [1,2,3])
sage: M
The map \( A \times x + b \) with \( A = \\
\begin{bmatrix}
1 & 2 \\
2 & 3 \\
-1 & 2 \\
\end{bmatrix} \\
b = \\
(1, 2, 3)
sage: M.domain_dim()
2
```

**exception** sage.geometry.polyhedron.lattice_euclidean_group_element.LatticePolytopeError

Bases: Exception

Base class for errors from lattice polytopes

**exception** sage.geometry.polyhedron.lattice_euclidean_group_element.LatticePolytopeNoEmbeddingError

Bases: 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 + 4*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
4 (t^2 + 2*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
4 (t^2 + 6*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
4 (t^2 + 6*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
4 (t^2 + 2*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
4 (t^2 + 4*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
5 (t^2 + 3*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
5 (t^2 + 5*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
5 (t^2 + 5*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
6 (t^2 + 4*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
6 (t^2 + 4*t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
```

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

Bases: 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)
```

```python
sage: [ (p.n_Vrepresentation(), len(p.integral_points())) for p in polygons ]
```

(continues on next page)

(continued from previous page)

[(3, 4), (3, 10), (3, 5), (3, 9), (3, 7), (4, 6), (4, 8), (4, 9),
(4, 5), (4, 5), (4, 9), (4, 7), (5, 8), (5, 6), (5, 7), (6, 7)]

sage: next(iter(PALPreader(2, output='list')))
[[1, 0], [0, 1], [-1, -1]]

sage: type(_)
<...

sage: next(iter(PALPreader(2, output='Polyhedron')))
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 3 vertices

sage: type(_)
<class 'sage.geometry.polyhedron.parent.Polyhedra_ZZ_ppl_with_category.element_class'

sage: next(iter(PALPreader(2, output='PPL')))
A 2-dimensional lattice polytope in ZZ^2 with 3 vertices

sage: type(_)
<class 'sage.geometry.polyhedron.ppl_lattice_polygon.LatticePolygon_PPL_class'>

sage: next(iter(PALPreader(2, output='PointCollection')))
[ 1, 0],
[ 0, 1],
[-1, -1]
in Ambient free module of rank 2 over the principal ideal domain Integer Ring

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

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
   True
   sage: iso(L1) == L2
   True

   The following polygons are isomorphic over \(\mathbb{Q}\), but not as lattice polytopes:

   sage: L1 = LatticePolytope_PPL((1, 0), (0, 1), (0, 3), (1, 0))
   sage: L2 = LatticePolytope_PPL((0, 0, 2, 1),(0,1,2,0),(2,0,0,3),(2,3,0,0))
   sage: iso = L1.find_isomorphism(L2)
   sage: iso(L1) == L2
   True

   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.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.

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:

```
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: from sage.geometry.polyhedron.ppl_lattice_polygon import polar_P1xP1_polytope
sage: polar_P1xP1_polytope()
A 2-dimensional lattice polytope in ZZ^2 with 4 vertices
sage: _.vertices()
((0, 0), (0, 2), (2, 0), (2, 2))
```

```
sage: from sage.geometry.polyhedron.ppl_lattice_polygon import polar_P2_112_polytope
sage: polar_P2_112_polytope()
A 2-dimensional lattice polytope in ZZ^2 with 3 vertices
sage: _.vertices()
((0, 0), (0, 2), (4, 0))
```

```
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: from sage.geometry.polyhedron.ppl_lattice_polygon import sub_reflexive_polygons
sage: 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:**
```python
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: from sage.geometry.polyhedron.ppl_lattice_polygon import subpolygons_of_polar_P1xP1()
The lattice sub-polygons of the polar $P1 \times P1$ polytope

OUTPUT:
A tuple of lattice polytopes.

EXAMPLES:
```python
sage: from sage.geometry.polyhedron.ppl_lattice_polygon import subpolygons_of_polar_P1xP1()
sage: len(subpolygons_of_polar_P1xP1())
20
```

```
sage: from sage.geometry.polyhedron.ppl_lattice_polygon import subpolygons_of_polar_P2()
The lattice sub-polygons of the polar $P2$ polytope

OUTPUT:
A tuple of lattice polytopes.

EXAMPLES:
```python
sage: from sage.geometry.polyhedron.ppl_lattice_polygon import subpolygons_of_polar_P2()
sage: len(subpolygons_of_polar_P2())
27
```

```
sage: from sage.geometry.polyhedron.ppl_lattice_polygon import subpolygons_of_polar_P2_112()
The lattice sub-polygons of the polar $P2[1,1,2]$ 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 \texttt{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.

\textbf{Note:} For general lattice polyhedra you should use \texttt{Polyhedron()} with base\_ring=\texttt{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 \texttt{Z\_vectors}. In the following, we always use \textit{generator} to refer the PPL generator objects and \textit{vertex} (or integral point) for the corresponding \texttt{Z\_vector}.

\textbf{EXAMPLES:}

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

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 \texttt{fibration\_generator()}.

\begin{verbatim}
sage: F = next(P.fibration\_generator(2))
sage: F.vertices()
((1, 0, 0, 0), (0, 1, 0, 0), (-3, -2, 0, 0))
\end{verbatim}

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

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

\textbf{AUTHORS:}

- Volker Braun: initial version, 2012

\texttt{sage.geometry.polyhedron.ppl\_lattice\_polytope.LatticePolytope\_PPL(*args)}

Construct a new instance of the PPL-based lattice polytope class.

\textbf{EXAMPLES:}

\begin{verbatim}
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
\end{verbatim}
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:

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_PPL
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:
From the Sage geometry library, we can define a 2-dimensional lattice polytope in 4 dimensions and a 2-dimensional lattice polytope in 2 dimensions.

```python
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));
sage: 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();
sage: 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:**

```python
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:**

```python
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
```

2.2. Lattice polyhedra
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))

base_projection_matrix(fiber)
The projection that maps the sub-polytope fiber to a single point.

OUTPUT:
An integer matrix that represents the projection to the base.

See also:
The base_projection() yields equivalent information, and is easier to use. However, just returning the matrix has lower overhead.

EXAMPLES:

sage: proj = poly.base_projection(fiber)
sage: proj_matrix = poly.base_projection_matrix(fiber)

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

sage: for p in poly.integral_points():
   print(proj(p)

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: line = LatticePolytope_PPL((0, 0), (1, 0), (0, 1)).bounding_box()

sage: line
((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:**

```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_origin()
False
```

2.2. Lattice polyhedra
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_PPL
sage: LatticePolytope_PPL((1,2,3), (-1,-2,-3)).contains_origin()
True
sage: LatticePolytope_PPL((1,2,5), (-1,-2,-3)).contains_origin()
False

embed_in_reflexive_polytope(output='hom')
Find an embedding as a sub-polytope of a maximal reflexive polytope.

INPUT:

• hom – string. One of 'hom' (default), 'polytope', or points. How the embedding is returned. See the output section for details.

OUTPUT:

An embedding into a reflexive polytope. Depending on the output option slightly different data is returned.

• If output='hom', a map from a reflexive polytope onto self is returned.

• If output='polytope', a reflexive polytope that contains self (up to a lattice linear transformation) is returned. That is, the domain of the output='hom' map is returned. If the affine span of self is less or equal 2-dimensional, the output is one of the following three possibilities: polar_P2_polytope(), polar_P1xP1_polytope(), or polar_P2_112_polytope().

• If output='points', a dictionary containing the integral points of self as keys and the corresponding integral point of the reflexive polytope as value.

If there is no such embedding, a LatticePolytopeNoEmbeddingError is raised. Even if it exists, the ambient reflexive polytope is usually not uniquely determined and a random but fixed choice will be returned.

EXAMPLES:

sage: from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_PPL
sage: polygon = LatticePolytope_PPL((0,0,2,1),(0,1,2,0),(2,3,0,0),(2,0,0,3))
sage: polygon.embed_in_reflexive_polytope()
The map A*x+b with A=
[[ 1  1]
 [ 0  1]
[-1 -1]
 [ 1  0]]
b =
(-1, 0, 3, 0)
sage: polygon.embed_in_reflexive_polytope('polytope')
A 2-dimensional lattice polytope in ZZ^2 with 3 vertices
sage: polygon.embed_in_reflexive_polytope('points')

(continues on next page)
fibration_generator\(\text{(dim)}\)

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 a 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:
```python
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_
˓→PPL
sage: LatticePolytope_PPL((-1,-1),(1,0),(1,1),(0,1)).integral_points()
((-1, -1), (0, 0), (0, 1), (1, 0), (1, 1))

sage: simplex = LatticePolytope_PPL((1,2,3), (2,3,7), (-2,-3,-11))
sage: simplex.integral_points()
((-2, -3, -11), (0, 0, -2), (1, 2, 3), (2, 3, 7))

The polyhedron need not be full-dimensional:

sage: simplex = LatticePolytope_PPL((1,2,3,5), (2,3,7,5), (-2,-3,-11,5))
sage: simplex.integral_points()
((-2, -3, -11, 5), (0, 0, -2, 5), (1, 2, 3, 5), (2, 3, 7, 5))

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

sage: empty = LatticePolytope_PPL()
sage: empty.integral_points()
()

Here is a simplex where the naive algorithm of running over all points in a rectangular bounding box no
longer works fast enough:

sage: v = [(1,0,7,-1), (-2,-2,4,-3), (-1,-1,-1,4), (2,9,0,-5), (-2,-1,5,1)]

sage: simplex = LatticePolytope_PPL(*v); simplex
A 4-dimensional lattice polytope in ZZ^4 with 5 vertices
sage: len(simplex.integral_points())
49

Finally, the 3-d reflexive polytope number 4078:

sage: v = [(1,0,0), (0,1,0), (0,0,1), (0,0,-1), (0,-2,1),
....:     (-1,2,-1), (-1,2,-2), (-1,1,-2), (-1,-1,2), (-1,-3,2)]

sage: P = LatticePolytope_PPL(*v)
sage: pts1 = P.integral_points() # Sage's own code
sage: pts2 = LatticePolytope(v).points() # PALP
sage: for p in pts1: p.set_immutable()

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

sage: len(Polyhedron(v).integral_points()) # takes about 1 ms
23
sage: len(LatticePolytope(v).points()) # takes about 13 ms
23
sage: len(LatticePolytope_PPL(*v).integral_points()) # takes about 0.5 ms
23

integral_points_not_interior_to_facets()
Return the integral points not interior to facets

OUTPUT:
A tuple whose entries are the coordinate vectors of integral points not interior to facets (codimension one faces) of the lattice polytope.

**EXAMPLES:**

```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),(1,0),(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((0,0,0), (1,0,0), (0,1,0)).is_simplex()
True
```
lattice_automorphism_group\((points=None, point_labels=None)\)

The integral subgroup of the restricted automorphism group.

**INPUT:**

- `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:**

```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.lattice_automorphism_group()
Permutation Group with generators [(1,2), (2,1), (0,0), (3,3)]

sage: G1 = Z3square.lattice_automorphism_group(point_labels=(1,2,3,4)); G1
Permutation Group with generators [(1,2), (2,1), (0,0), (3,3)]

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 [(1,2), (0,1,2,3,4,5,6)]
```

Point labels also work for lattice polytopes that are not full-dimensional, see trac ticket #16669:

```python
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), (0,1,2,3,4,5,6)]
```

n_integral_points()

Return the number of integral points.

**OUTPUT:**

Integer. The number of integral points contained in the lattice polytope.

**EXAMPLES:**
n_vertices()
Return the number of vertices.

OUTPUT:
An integer, the number of vertices.

EXAMPLES:

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

```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: fibers = [ f.vertices() for f in square.fibration_generator(1) ]
sage: square.pointsets_mod_automorphism(fibers)
(frozenset({(-1, -1), (1, 1)}), frozenset({(-1, 0), (1, 0)}))
sage: cell24 = LatticePolytope_PPL((1,0,0,0),(0,1,0,0),(0,0,1,0),(0,0,0,1),(1,-1,-1,1),(0,0,-1,1),
....: (0,-1,0,1),(-1,1,1,0),(-1,1,0,0),(-1,0,1,0),(1,-1,0,0),(-1,0,0,1))
sage: fibers = [f.vertices() for f in cell24.fibration_generator(2)]
sage: cell24.pointsets_mod_automorphism(fibers)  # long time
(frozenset({(-1, -1, 0, 0), (-1, 0, 0, 1), (0, 0, 0, -1), (0, 0, 0, 1), (1, 0, 0, 0)}),
 frozenset({(-1, -1, 1, 1), (-1, 1, 1, 0), (-1, 1, 0, 0), (0, -1, 1, 1), (0, 0, 0, -1)}))
```

restricted_automorphism_group(vertex_labels=None)
Return the restricted automorphism group.
First, let the linear automorphism group be the subgroup of the Euclidean group \(E(d) = GL(d, \mathbb{R}) \times \mathbb{R}^d\) 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))
```

```python
sage: Z3square.restricted_automorphism_group(vertex_labels=(1,2,3,4)) == PermutationGroup([(2,3), (1,2), (3,4)])
True
```

```python
sage: G = Z3square.restricted_automorphism_group()
```

```python
sage: G == PermutationGroup([(1,2), (2,1), (3,3)])
True
```

```python
sage: set(G.domain()) == set(Z3square.vertices())
True
```

```python
sage: set(map(tuple, G.orbit(Z3square.vertices()[0]))) == set([(0, 0), (1, 2), (3, 3), (2, 1)])
True
```

```python
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,-1,0,1),(-1,0,0,1),(1,0,0,-1),(0,1,0,-1),(0,0,1,-1),(-1,1,1,-1),....: (1,-1,-1,0),(0,-1,0,-1),(0,-1,0,0),(1,-1,0,0),(1,0,-1,0),....: (0,1,1,-1),(-1,1,1,0),(-1,1,0,0),(-1,0,1,0),(0,-1,1,1),(0,0,0,-1))
```

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

```python
sage: for p in P.sub_polytope_generator():
    print(p.vertices())
```
vertices()  
Return the vertices as a tuple of \( \mathbb{Z} \)-vectors.

OUTPUT:  
A tuple of \( \mathbb{Z} \)-vectors. Each entry is the coordinate vector of an integral point of the lattice polytope.

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)}
```

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 vertices().

EXAMPLES:

```python
sage: from sage.geometry.polyhedron.ppl_lattice_polytope import LatticePolytope_PPL
sage: p = LatticePolytope_PPL((0,0),(0,1),(1,0))
sage: ieq = next(iter(p.constraints())); ieq  
x0>=0
sage: p.vertices_saturating(ieq)  
((0, 0), (0, 1))
```

sage.geometry.polyhedron.ppl_lattice_polytope.line(cls, expression)  
Generator.line(type cls, expression)

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.
```

```python
sage.geometry.polyhedron.ppl_lattice_polytope.point(cls, expression=0, divisor=1)
Generator.point(type cls, expression=0, divisor=1)
```
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.
```

## 2.3 Polyhedral complexes

### 2.3.1 Finite polyhedral complexes

This module implements the basic structure of finite polyhedral complexes. For more information, see `PolyhedralComplex`.

**AUTHORS:**

- Yuan Zhou (2021-05): initial implementation
## List of PolyhedralComplex methods

### Maximal cells and cells

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>maximal_cells()</code></td>
<td>Return the dictionary of the maximal cells in this polyhedral complex.</td>
</tr>
<tr>
<td><code>maximal_cell_iterator()</code></td>
<td>Return an iterator over maximal cells in this polyhedral complex.</td>
</tr>
<tr>
<td><code>maximal_cells_sorted()</code></td>
<td>Return the sorted list of all maximal cells in this polyhedral complex.</td>
</tr>
<tr>
<td><code>n_maximal_cells()</code></td>
<td>List the maximal cells of dimension $n$ in this polyhedral complex.</td>
</tr>
<tr>
<td><code>_n_maximal_cells_sorted()</code></td>
<td>Return the sorted list of maximal cells of dim $n$ in this complex.</td>
</tr>
<tr>
<td><code>is_maximal_cell()</code></td>
<td>Return True if the given cell is a maximal cell in this complex.</td>
</tr>
<tr>
<td><code>cells()</code></td>
<td>Return the dictionary of the cells in this polyhedral complex.</td>
</tr>
<tr>
<td><code>cell_iterator()</code></td>
<td>Return an iterator over cells in this polyhedral complex.</td>
</tr>
<tr>
<td><code>cells_sorted()</code></td>
<td>Return the sorted list of all cells in this polyhedral complex.</td>
</tr>
<tr>
<td><code>n_cells()</code></td>
<td>List the cells of dimension $n$ in this polyhedral complex.</td>
</tr>
<tr>
<td><code>_n_cells_sorted()</code></td>
<td>Return the sorted list of $n$-cells in this polyhedral complex.</td>
</tr>
<tr>
<td><code>is_cell()</code></td>
<td>Return True if the given cell is in this polyhedral complex.</td>
</tr>
<tr>
<td><code>face_poset()</code></td>
<td>Return the poset of nonempty cells in the polyhedral complex.</td>
</tr>
<tr>
<td><code>relative_boundary_cells()</code></td>
<td>List the maximal cells on the boundary of the polyhedral complex.</td>
</tr>
</tbody>
</table>

### Properties of the polyhedral complex

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>dimension()</code></td>
<td>Return the dimension of the polyhedral complex.</td>
</tr>
<tr>
<td><code>ambient_dimension()</code></td>
<td>Return the ambient dimension of the polyhedral complex.</td>
</tr>
<tr>
<td><code>is_pure()</code></td>
<td>Return True if the polyhedral complex is pure.</td>
</tr>
<tr>
<td><code>is_full_dimensional()</code></td>
<td>Return True if the polyhedral complex is full dimensional.</td>
</tr>
<tr>
<td><code>is_compact()</code></td>
<td>Return True if the polyhedral complex is bounded.</td>
</tr>
<tr>
<td><code>is_connected()</code></td>
<td>Return True if the polyhedral complex is connected.</td>
</tr>
<tr>
<td><code>is_subcomplex()</code></td>
<td>Return True if this complex is a subcomplex of the other.</td>
</tr>
<tr>
<td><code>is_convex()</code></td>
<td>Return True if the polyhedral complex is convex.</td>
</tr>
<tr>
<td><code>is Mutable()</code></td>
<td>Return True if the polyhedral complex is mutable.</td>
</tr>
<tr>
<td><code>isImmutable()</code></td>
<td>Return True if the polyhedral complex is not mutable.</td>
</tr>
<tr>
<td><code>is_simplicial_complex()</code></td>
<td>Return True if the polyhedral complex is a simplicial complex.</td>
</tr>
<tr>
<td><code>is_polyhedral_fan()</code></td>
<td>Return True if the polyhedral complex is a fan.</td>
</tr>
<tr>
<td><code>is_simplicial_fan()</code></td>
<td>Return True if the polyhedral complex is a simplicial fan.</td>
</tr>
</tbody>
</table>

### New polyhedral complexes from old ones

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>connected_component()</code></td>
<td>Return the connected component containing a cell as a subcomplex.</td>
</tr>
<tr>
<td><code>connected_components()</code></td>
<td>Return the connected components of this polyhedral complex.</td>
</tr>
<tr>
<td><code>n_skeleton()</code></td>
<td>Return the $n$-skeleton of this polyhedral complex.</td>
</tr>
<tr>
<td><code>stratify()</code></td>
<td>Return the (pure) subcomplex formed by the maximal cells of dim $n$ in this complex.</td>
</tr>
<tr>
<td><code>boundary_subcomplex()</code></td>
<td>Return the boundary subcomplex of this polyhedral complex.</td>
</tr>
<tr>
<td><code>product()</code></td>
<td>Return the (Cartesian) product of this polyhedral complex with another one.</td>
</tr>
<tr>
<td><code>disjoint_union()</code></td>
<td>Return the disjoint union of this polyhedral complex with another one.</td>
</tr>
<tr>
<td><code>union()</code></td>
<td>Return the union of this polyhedral complex with another one.</td>
</tr>
<tr>
<td><code>join()</code></td>
<td>Return the join of this polyhedral complex with another one.</td>
</tr>
<tr>
<td><code>subdivide()</code></td>
<td>Return a new polyhedral complex (with option make_simplicial) subdividing this one.</td>
</tr>
</tbody>
</table>

### Update polyhedral complex

## 2.3. Polyhedral complexes
**Miscellaneous**

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>set_immutable()</code></td>
<td>Make this polyhedral complex immutable.</td>
</tr>
<tr>
<td><code>add_cell()</code></td>
<td>Add a cell to this polyhedral complex.</td>
</tr>
<tr>
<td><code>remove_cell()</code></td>
<td>Remove a cell from this polyhedral complex.</td>
</tr>
</tbody>
</table>

**Classes and functions**

```python
class sage.geometry.polyhedral_complex.PolyhedralComplex(maximal_cells=None, backend=None, maximality_check=True, face_to_face_check=False, is_mutable=True, is_immutable=False, ambient_dim=None):
```

Bases: `sage.topology.cell_complex.GenericCellComplex`

A polyhedral complex.

A **polyhedral complex** $PC$ is a collection of polyhedra in a certain ambient space $\mathbb{R}^n$ such that the following hold:

- If a polyhedron $P$ is in $PC$, then all the faces of $P$ are in $PC$.
- If polyhedra $P$ and $Q$ are in $PC$, then $P \cap Q$ is either empty or a face of both $P$ and $Q$.

In this context, a “polyhedron” means the geometric realization of a polyhedron. This is in contrast to simplicial complex, whose cells are abstract simplices. The concept of a polyhedral complex generalizes that of a geometric simplicial complex.

**Note:** This class derives from `GenericCellComplex`, and so inherits its methods. Some of those methods are not listed here; see the [Generic Cell Complex](#) page instead.

**INPUT:**

- `maximal_cells` – a list, a tuple, or a dictionary (indexed by dimension) of cells of the Complex. Each cell is of class `Polyhedron` of the same ambient dimension. To set up a `PolyhedralComplex`, it is sufficient to provide the maximal faces. Use keyword argument `partial=True` to set up a partial polyhedral complex, which is a subset of the faces (viewed as relatively open) of a polyhedral complex that is not necessarily closed under taking intersection.
- `maximality_check` – boolean (default: True); if True, then the constructor checks that each given maximal cell is indeed maximal, and ignores those that are not.
- `face_to_face_check` – boolean (default: False); if True, then the constructor checks whether the cells are face-to-face, and it raises a `ValueError` if they are not.
- `is_mutable` and `is_immutable` – boolean (default: True and False respectively); set `is_mutable=False` or `is_immutable=True` to make this polyhedral complex immutable.
- `backend` – string (optional); the name of the backend used for computations on Sage polyhedra; if it is not given, then each cell has its own backend; otherwise it must be one of the following:
- 'ppl' - the Parma Polyhedra Library
- 'cdd' - CDD
- 'normaliz' - normaliz
- 'polymake' - polymake
- 'field' - a generic Sage implementation

• ambient_dim – integer (optional); used to set up an empty complex in the intended ambient space

EXAMPLES:

```python
sage: pc = PolyhedralComplex([  
....: Polyhedron(vertices=[[1/3, 1/3], (0, 0), (1/7, 2/7)]),  
....: Polyhedron(vertices=[[1/7, 2/7], (0, 0), (0, 1/4)])])
sage: [p.Vrepresentation() for p in pc.cells_sorted()]
[(A vertex at (0, 0), A vertex at (0, 1/4), A vertex at (1/7, 2/7)),
 (A vertex at (0, 0), A vertex at (1/3, 1/3), A vertex at (1/7, 2/7)),
 (A vertex at (0, 0), A vertex at (0, 1/4)),
 (A vertex at (0, 0), A vertex at (1/7, 2/7)),
 (A vertex at (0, 0), A vertex at (1/3, 1/3)),
 (A vertex at (0, 1/4), A vertex at (1/7, 2/7)),
 (A vertex at (1/3, 1/3), A vertex at (1/7, 2/7)),
 (A vertex at (0, 0),),
 (A vertex at (0, 1/4),),
 (A vertex at (1/7, 2/7),),
 (A vertex at (1/3, 1/3),)]
sage: pc.plot()
Graphics object consisting of 10 graphics primitives
sage: pc.is_pure()
True
sage: pc.is_full_dimensional()
True
sage: pc.is_compact()
True
sage: pc.boundary_subcomplex()
Polyhedral complex with 4 maximal cells
sage: pc.is_convex()
True
sage: pc.union_as_polyhedron().Hrepresentation()
(An inequality (1, -4) x + 1 >= 0,
 An inequality (-1, 1) x + 0 >= 0,
 An inequality (1, 0) x + 0 >= 0)
sage: pc.face_poset()
Finite poset containing 11 elements
sage: pc.is_connected()
True
sage: pc.connected_component() == pc
True
```

`add_cell(cell)`

Add a cell to this polyhedral complex.

INPUT:

• cell – a polyhedron
This changes the polyhedral complex, by adding a new cell and all of its subfaces.

EXAMPLES:

Set up an empty complex in the intended ambient space, then add a cell:

```
sage: pc = PolyhedralComplex(ambient_dim=2)
sage: pc.add_cell(Polyhedron(vertices=\[(1, 2), (0, 2)\]))
sage: pc
Polyhedral complex with 1 maximal cell
```

If you add a cell which is already present, there is no effect:

```
sage: pc.add_cell(Polyhedron(vertices=\[(1, 2)\]))
sage: pc
Polyhedral complex with 1 maximal cell
sage: pc.dimension()
1
```

Add a cell and check that dimension is correctly updated:

```
sage: pc.add_cell(Polyhedron(vertices=\[(1, 2), (0, 0), (0, 2)\]))
sage: pc.dimension()
2
sage: pc.maximal_cells()
\{2: \{A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 3 vertices\}\}
sage: pc.is_convex()
True
```

Add another cell and check that the properties are correctly updated:

```
sage: pc.add_cell(Polyhedron(vertices=\[(1, 0), (0, 0), (1, 2)\]))
sage: pc
Polyhedral complex with 2 maximal cells
sage: len(pc._cells[1])
5
sage: pc._face_poset
Finite poset containing 11 elements
sage: pc._is_convex
True
sage: pc._polyhedron.vertices_list()
[[0, 0], [0, 2], [1, 1], [1, 2]]
```

Add a ray which makes the complex non convex:

```
sage: pc.add_cell(Polyhedron(rays=\[(1, 0)\]))
sage: pc
Polyhedral complex with 3 maximal cells
sage: len(pc._cells[1])
6
sage: (pc._is_convex is False) and (pc._polyhedron is None)
True
```

alexander_whitney(cell, dim_left)

The decomposition of cell in this complex into left and right factors, suitable for computing cup products.
Todo: Implement `alexander_whitney()` of a polyhedral complex.

**Examples:**

```python
sage: pc = PolyhedralComplex([Polyhedron(vertices=[[0], [1]])])
sage: pc.alexander_whitney(None, 1)
Traceback (most recent call last):
  ...  
NotImplementedError: alexander_whitney is not implemented for polyhedral complex
```

**ambient_dimension()**

The ambient dimension of this cell complex: the ambient dimension of each of its cells.

**Examples:**

```python
sage: pc = PolyhedralComplex([Polyhedron(vertices=[(1, 2, 3)])])
sage: pc.ambient_dimension()
3
sage: empty_pc = PolyhedralComplex([])
sage: empty_pc.ambient_dimension()
-1
sage: pc0 = PolyhedralComplex(ambient_dim=2)
sage: pc0.ambient_dimension()
2
```

**boundary_subcomplex()**

Return the sub-polyhedral complex that is the boundary of `self`.

A point `P` is on the boundary of a set `S` if `P` is in the closure of `S` but not in the interior of `S`.

**Examples:**

```python
sage: p1 = Polyhedron(vertices=[[1, 1], [0, 0], [1, 2]])
sage: p2 = Polyhedron(vertices=[[1, 2], [0, 0], [0, 2]])
sage: p3 = Polyhedron(vertices=[[1, 2], [0, 2]])
sage: bd = PolyhedralComplex([p1, p2]).boundary_subcomplex()
sage: len(bd.n_maximal_cells(2))
0
sage: len(bd.n_maximal_cells(1))
4
sage: pt = PolyhedralComplex([p3])
sage: pt.boundary_subcomplex() == pt
True
```

Test on polyhedral complex which is not pure:

```python
sage: pc_non_pure = PolyhedralComplex([p1, p3])
sage: pc_non_pure.boundary_subcomplex() == pc_non_pure.n_skeleton(1)
True
```

Test with `maximality_check == False`:

```python
sage: pc_invalid = PolyhedralComplex([[p2, p3],
        ...: maximality_check=False)
```

(continues on next page)
Test unbounded cases:

```python
sage: pc1 = PolyhedralComplex([Polyhedron(vertices=[[1,0], [0,1]], rays=[[1,0], [0,1]]))
sage: pc1.boundary_subcomplex() == pc1.n_skeleton(1)
True
sage: pc1b = PolyhedralComplex([Polyhedron(vertices=[[1,0,0], [0,1,0]], rays=[[1,0,0],[0,1,0]]))
sage: pc1b.boundary_subcomplex() == pc1b.n_skeleton(1)
True
sage: pc2 = PolyhedralComplex([Polyhedron(vertices=[[-1,0], [1,0]], lines=[[0,1]]))
sage: pc2.boundary_subcomplex() == pc2.n_skeleton(1)
True
sage: pc3 = PolyhedralComplex([Polyhedron(vertices=[[1,0], [0,1]], rays=[[1,0], [0,1]]), Polyhedron(vertices=[[1,0], [0,-1]], rays=[[1,0], [0,-1]]))
sage: pc3.boundary_subcomplex() == pc3.n_skeleton(1)
False
```

**cell_iterator** *(increasing=True)*

An iterator for the cells in this polyhedral complex.

**INPUT:**

- increasing – (default True) if True, return cells in increasing order of dimension, thus starting with the zero-dimensional cells; otherwise it returns cells in decreasing order of dimension

**Note:** Among the cells of a fixed dimension, there is no sorting.

**EXAMPLES:**

```python
sage: pc = PolyhedralComplex([Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)]), Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)]))
```

**cells** *(subcomplex=None)*

The cells of this polyhedral complex, in the form of a dictionary: the keys are integers, representing dimension, and the value associated to an integer $d$ is the set of $d$-cells.

**INPUT:**

- subcomplex – (optional) if a subcomplex is given then return the cells which are **not** in this subcomplex

**EXAMPLES:**

```python
sage: pc = PolyhedralComplex([Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)]), Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])])
```
....: Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])
sage: list(pc.cells().keys())
[2, 1, 0]

cells_sorted(subcomplex=None)
The sorted list of the cells of this polyhedral complex in non-increasing dimensions.

INPUT:

• subcomplex – (optional) if a subcomplex is given then return the cells which are not in this subcomplex

EXAMPLES:

sage: pc = PolyhedralComplex([ ....: Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)]), ....: Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])])
sage: len(pc.cells_sorted())
11
sage: pc.cells_sorted()[0].Vrepresentation()
(A vertex at (0, 0), A vertex at (0, 2), A vertex at (1, 2))

chain_complex(subcomplex=None, augmented=False, verbose=False, check=True, dimensions=None, base_ring=Integer Ring, cochain=False)
The chain complex associated to this polyhedral complex.

Todo: Implement chain complexes of a polyhedral complex.

EXAMPLES:

sage: pc = PolyhedralComplex([ ....: Polyhedron(vertices=[[0], [1]])])
sage: pc.chain_complex()
Traceback (most recent call last):
  ...
NotImplementedError: chain_complex is not implemented for polyhedral complex

connected_component(cell=None)
Return the connected component of this polyhedral complex containing a given cell.

INPUT:

• cell – (default: self.an_element()) a cell of self

OUTPUT:
The connected component containing cell. If the polyhedral complex is empty or if it does not contain the given cell, raise an error.

EXAMPLES:

sage: t1 = Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)])
sage: t2 = Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])
sage: v1 = Polyhedron(vertices=[(1, 1)])
sage: v2 = Polyhedron(vertices=[(0, 2)])
sage: v3 = Polyhedron(vertices=[(-1, 0)])
sage: o = Polyhedron(vertices=[(0, 0)])
sage: r = Polyhedron(rays=[(1, 0)])
sage: l = Polyhedron(  
vertices=[(-1, 0)],  
lines=[(1, -1)])
sage: pc1 = PolyhedralComplex([t1, t2])
sage: pc1.connected_component() == pc1
True
sage: pc1.connected_component(v1) == pc1
True
sage: pc2 = PolyhedralComplex([t1, v2])
sage: pc2.connected_component(t1) == PolyhedralComplex([t1])
True
sage: pc2.connected_component(o) == PolyhedralComplex([t1])
True
sage: pc2.connected_component(v3)
Traceback (most recent call last):
...  
ValueError: the polyhedral complex does not contain the given cell
sage: pc2.connected_component(r)
Traceback (most recent call last):
...
ValueError: the polyhedral complex does not contain the given cell
sage: pc3 = PolyhedralComplex([t1, t2, r])
sage: pc3.connected_component(v2) == pc3
True
sage: pc4 = PolyhedralComplex([t1, t2, r, l])
sage: pc4.connected_component(o) == pc3
True
sage: pc4.connected_component(v3)
Traceback (most recent call last):
...
ValueError: the polyhedral complex does not contain the given cell
sage: pc5 = PolyhedralComplex([t1, t2, r, l, v3])
sage: pc5.connected_component(v3) == PolyhedralComplex([v3])
True
sage: PolyhedralComplex([]).connected_component()
Traceback (most recent call last):
...
ValueError: the empty polyhedral complex has no connected components

connected_components()

Return the connected components of this polyhedral complex, as list of (sub-)PolyhedralComplexes.

EXAMPLES:

sage: t1 = Polyhedron(  
vertices=[(1, 1), (0, 0), (1, 2)])
sage: t2 = Polyhedron(  
vertices=[(1, 2), (0, 0), (0, 2)])
sage: v1 = Polyhedron(  
vertices=[(1, 1)])
sage: v2 = Polyhedron(  
vertices=[(0, 2)])
sage: v3 = Polyhedron(  
vertices=[(-1, 0)])
sage: o = Polyhedron(  
vertices=[(0, 0)])
sage: r = Polyhedron(  
rays=[(1, 0)])
sage: l = Polyhedron(  
vertices=[(-1, 0)],  
lines=[(1, -1)])
sage: pc1 = PolyhedralComplex([t1, t2])
sage: len(pc1.connected_components())
1
sage: pc2 = PolyhedralComplex([t1, v2])
sage: len(pc2.connected_components())
2
sage: pc3 = PolyhedralComplex([t1, t2, r])
sage: len(pc3.connected_components())
1
sage: pc4 = PolyhedralComplex([t1, t2, r, l])
sage: len(pc4.connected_components())
2
sage: pc5 = PolyhedralComplex([t1, t2, r, l, v3])
sage: len(pc5.connected_components())
3
sage: PolyhedralComplex([]).connected_components()
Traceback (most recent call last):
...
ValueError: the empty polyhedral complex has no connected components

dimension()
The dimension of this cell complex: the maximum dimension of its cells.

EXAMPLES:

sage: pc = PolyhedralComplex([
    ....: Polyhedron(vertices=[[1, 1], (0, 0), (1, 2)]),
    ....: Polyhedron(vertices=[[1, 2], (0, 2)]))
sage: pc.dimension()
2
sage: empty_pc = PolyhedralComplex([])
sage: empty_pc.dimension()
-1

disjoint_union(right)
The disjoint union of this polyhedral complex with another one.

INPUT:

• right – the other polyhedral complex (the right-hand factor)

EXAMPLES:

sage: p1 = Polyhedron(vertices=[(-1, 0), (0, 0), (0, 1)])
sage: p2 = Polyhedron(vertices=[[0, -1], (0, 0), (1, 0)])
sage: p3 = Polyhedron(vertices=[[0, -1], (1, -1), (1, 0)])
sage: pc = PolyhedralComplex([p1]).disjoint_union(PolyhedralComplex([p3]))
sage: set(pc.maximal_cell_iterator()) == set([p1, p3])
True
sage: pc.disjoint_union(PolyhedralComplex([p2]))
Traceback (most recent call last):
...
ValueError: the two complexes are not disjoint

face_poset()
The face poset of this polyhedral complex, the poset of nonempty cells, ordered by inclusion.
EXAMPLES:

```python
sage: pc = PolyhedralComplex([  
....:     Polyhedron(vertices=[(1/3, 1/3), (0, 0), (1, 2)]),  
....:     Polyhedron(vertices=[(1, 2), (0, 0), (0, 1/2)]))
sage: poset = pc.face_poset()
sage: poset
Finite poset containing 11 elements
sage: d = {i: i.vertices_matrix() for i in poset}
sage: poset.plot(element_labels=d)
Graphics object consisting of 28 graphics primitives
```

For a nonbounded polyhedral complex:

```python
sage: pc = PolyhedralComplex([  
....:     Polyhedron(vertices=[(1/3, 1/3), (0, 0), (1, 2)]),  
....:     Polyhedron(vertices=[(1, 2), (0, 0), (0, 1/2)]),  
....:     Polyhedron(vertices=[(-1/2, -1/2)], lines=[[(1, -1)]],  
....:     Polyhedron(rays=[(1, 0)])]
sage: poset = pc.face_poset()
sage: poset
Finite poset containing 13 elements
sage: d = {i: '
'.join([str(v) + '
' for v in i.Vrepresentation()]) for i in poset}
sage: poset.plot(element_labels=d)
```

```python
sage: pc = PolyhedralComplex([  
....:     Polyhedron(rays=[(1,0),(0,1)]),  
....:     Polyhedron(rays=[(-1,0),(0,1)]),  
....:     Polyhedron(rays=[(-1,0),(0,-1)]),  
....:     Polyhedron(rays=[(1,0),(0,-1)])]
sage: pc.face_poset()
Finite poset containing 9 elements
```

```python
graph()
The 1-skeleton of this polyhedral complex, as a graph.
The vertices of the graph are of type `vector`. Raises a `NotImplementedError` if the polyhedral complex is unbounded.

**Warning:** This may give the wrong answer if the polyhedral complex was constructed with `maximality_check` set to `False`.

EXAMPLES:

```python
sage: pc = PolyhedralComplex([  
....:     Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)]),  
....:     Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)]))
sage: g = pc.graph(); g
Graph on 4 vertices
sage: g.vertices()
[(0, 0), (0, 2), (1, 1), (1, 2)]
sage: g.edges(labels=False)
[((0, 0), (0, 2)), ((0, 0), (1, 1)), ((0, 0), (1, 2)), ((0, 2), (1, 2)), ((1, 0), (1, 2))]
```

(continues on next page)
is_cell(c)
Return whether the given cell c is a cell of self.

 EXAMPLES:

```
sage: p1 = Polyhedron(vertices=[[1, 1], [0, 0], [1, 2]])
sage: p2 = Polyhedron(vertices=[[1, 2], [0, 0], [0, 2]])
sage: p3 = Polyhedron(vertices=[[1, 2], [0, 2]])
sage: pc = PolyhedralComplex([p1, p2])
sage: pc.is_cell(p3)
True
sage: pc.is_cell(Polyhedron(verti
```
.. code::

    sage: pc2.is_connected()
    False

    sage: pc3 = PolyhedralComplex(
        ....: Polyhedron(vertices=[(1/3, 1/3), (0, 0), (1, 2)]),
        ....: Polyhedron(vertices=[(-1/2, -1/2)], lines=[[1, -1]]),
        ....: Polyhedron(rays=[[1, 0]])
    sage: pc3.is_connected()
    False

    sage: pc4 = PolyhedralComplex(
        ....: Polyhedron(vertices=[(1/3, 1/3), (0, 0), (1, 2)]),
        ....: Polyhedron(rays=[[1, 0]])
    sage: pc4.is_connected()
    True


.. attribute:: is_convex()

    Return whether the set of points in :attr:`self` is a convex set.

    When :attr:`self` is convex, the union of its cells is a Polyhedron.

    .. seealso::

        :meth:`union_as_polyhedron`

    EXAMPLES:

    .. code::

        sage: p1 = Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)])
        sage: p2 = Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])
        sage: p3 = Polyhedron(vertices=[(0, 0), (1, 1), (2, 0)])
        sage: p4 = Polyhedron(vertices=[(2, 2)])
        sage: PolyhedralComplex([p1, p2]).is_convex()
        True
        sage: PolyhedralComplex([p1, p3]).is_convex()
        False
        sage: PolyhedralComplex([p1, p4]).is_convex()
        False

    Test unbounded cases:

    .. code::

        sage: pc1 = PolyhedralComplex([  
            ....: Polyhedron(vertices=[[1,0], [0,1]], rays=[[1,0], [0,1]])  
        ]
        sage: pc1.is_convex()
        True

        sage: pc2 = PolyhedralComplex([  
            ....: Polyhedron(vertices=[[-1,0], [1,0]], lines=[[0,1]])  
        ]
        sage: pc2.is_convex()
        True

        sage: pc3 = PolyhedralComplex([  
            ....: Polyhedron(vertices=[[1,0], [0,1]], rays=[[1,0], [0,1]]),  
            ....: Polyhedron(vertices=[[1,0], [0,-1]], rays=[[1,0], [0,-1]])  
        ]
        sage: pc3.is_convex()
        False

(continues on next page)
sage: pc4 = PolyhedralComplex([Polyhedron(rays=[[1,0], [-1,1]]),
....:                           Polyhedron(rays=[[1,0], [-1,-1]]))
sage: pc4.is_convex()
False

The whole 3d space minus the first orthant is not convex:

sage: pc5 = PolyhedralComplex([Polyhedron(rays=[[1,0,0], [0,1,0], [0,0,-1]]),
....:                           Polyhedron(rays=[[1,0,0], [0,-1,0], [0,0,-1]]),
....:                           Polyhedron(rays=[[1,0,0], [0,-1,0], [0,0,1]]),
....:                           Polyhedron(rays=[[1,0,0], [0,1,0], [0,0,-1]]),
....:                           Polyhedron(rays=[[1,0,0], [0,1,0], [0,0,1]]))
sage: pc5.is_convex()
False

Test some non-full-dimensional examples:

sage: l = PolyhedralComplex([Polyhedron(vertices=[(1, 2), (0, 2)])])
sage: l.is_convex()
True
sage: pc1b = PolyhedralComplex([Polyhedron(
....:                            vertices=[[1,0,0], [0,1,0]],
....:                            rays=[[1,0,0],[0,1,0]])])
sage: pc1b.is_convex()
True
sage: pc4b = PolyhedralComplex([Polyhedron(rays=[[1,0,0], [-1,1,0]]),
....:                            Polyhedron(rays=[[1,0,0], [-1,-1,0]]))

is_full_dimensional()
Return whether this polyhedral complex is full-dimensional: its dimension is equal to its ambient dimension.

EXAMPLES:

sage: p1 = Polyhedron(vertices=[[1, 1], (0, 0), (1, 2)])
 sage: p2 = Polyhedron(vertices=[[1, 2], (0, 0), (0, 2)])
 sage: p3 = Polyhedron(vertices=[[1, 2], (0, 2)])
 sage: pc = PolyhedralComplex([p1, p2, p3])
 sage: pc.is_full_dimensional()
 True
 sage: PolyhedralComplex([p3]).is_full_dimensional()
 False

is_immutable()
Return whether self is immutable.

EXAMPLES:
### is_maximal_cell(c)

Return whether the given cell $c$ is a maximal cell of `self`.

**Warning:** This may give the wrong answer if the polyhedral complex was constructed with `maximality_check` set to `False`.

#### EXAMPLES:

```python
sage: p1 = Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)])
sage: p2 = Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])
sage: p3 = Polyhedron(vertices=[(1, 2), (0, 2)])
sage: pc = PolyhedralComplex([p1, p2, p3])
sage: pc.is_maximal_cell(p1)
True
sage: pc.is_maximal_cell(p3)
False
```

Wrong answer due to `maximality_check=False`:

```python
sage: pc_invalid = PolyhedralComplex([p1, p2, p3],
                                             maximality_check=False)
sage: pc_invalid.is_maximal_cell(p3)
True
```

### is_mutable()

Return whether `self` is mutable.

#### EXAMPLES:

```python
sage: pc1 = PolyhedralComplex([Polyhedron(vertices=[[0], [1]])])
sage: pc1.is_mutable()
True
sage: pc2 = PolyhedralComplex([Polyhedron(vertices=[[0], [1]])],
                                             is_mutable=False)
sage: pc2.is_mutable()
False
sage: pc1 == pc2
True
sage: pc3 = PolyhedralComplex([Polyhedron(vertices=[[0], [1]])],
                                             is_immutable=True)
(continues on next page)```
(continued from previous page)

```python
sage: pc3.is_mutable()
False
sage: pc2 == pc3
True
```

### `is_polyhedral_fan()`
Test if this polyhedral complex is a polyhedral fan.

A polyhedral complex is a **fan** if all of its (maximal) cells are cones.

**EXAMPLES:**

```python
sage: p1 = Polyhedron(vertices=[(0, 0), (1, 1), (1, 2)])
sage: p2 = Polyhedron(rays=[(1, 0)])
sage: PolyhedralComplex([p1]).is_polyhedral_fan()
False
sage: PolyhedralComplex([p2]).is_polyhedral_fan()
True
sage: halfplane = Polyhedron(rays=[(1, 0), (-1, 0), (0, 1)])
sage: PolyhedralComplex([halfplane]).is_polyhedral_fan()
True
```

### `is_pure()`
Test if this polyhedral complex is pure.

A polyhedral complex is pure if and only if all of its maximal cells have the same dimension.

**Warning:** This may give the wrong answer if the polyhedral complex was constructed with `maximality_check` set to `False`.

**EXAMPLES:**

```python
sage: p1 = Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)])
sage: p2 = Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])
sage: p3 = Polyhedron(vertices=[(1, 2), (0, 2)])
sage: pc = PolyhedralComplex([p1, p2, p3])
sage: pc.is_pure()
True
```

Wrong answer due to `maximality_check=False`:

```python
sage: pc_invalid = PolyhedralComplex([p1, p2, p3],
.....:    maximality_check=False)
sage: pc_invalid.is_pure()
False
```

### `is_simplicial_complex()`
Test if this polyhedral complex is a simplicial complex.

A polyhedral complex is **simplicial** if all of its (maximal) cells are simplices, i.e., every cell is a bounded polytope with $d + 1$ vertices, where $d$ is the dimension of the polytope.

**EXAMPLES:**
is_simplicial_fan()  
Test if this polyhedral complex is a simplicial fan.  
A polyhedral complex is a simplicial fan if all of its (maximal) cells are simplicial cones, i.e., every cell is a pointed cone (with vertex being the origin) generated by \( d \) linearly independent rays, where \( d \) is the dimension of the cone.

EXAMPLES:

```python
sage: p1 = Polyhedron(vertices=[(0, 0), (1, 1), (1, 2)])
sage: p2 = Polyhedron(rays=[(1, 0)])
sage: PolyhedralComplex([p1]).is_simplicial_fan()  # False
sage: PolyhedralComplex([p2]).is_simplicial_fan()  # True
sage: halfplane = Polyhedron(rays=[(1, 0), (-1, 0), (0, 1)])

# False
sage: PolyhedralComplex([halfplane]).is_simplicial_fan()
```

is_subcomplex(other)  
Return whether self is a subcomplex of other.

INPUT:

- other – a polyhedral complex

Each maximal cell of self must be a cell of other for this to be True.

EXAMPLES:

```python
sage: p1 = Polyhedron(vertices=[(1/3, 1/3), (0, 0), (1, 2)])
sage: p2 = Polyhedron(vertices=[(1, 2), (0, 0), (0, 1/2)])
sage: p3 = Polyhedron(vertices=[(0, 0), (1, 0)])
sage: pc = PolyhedralComplex([p1, Polyhedron(vertices=[(1, 0)])])
sage: pc.is_subcomplex(PolyhedralComplex([p1, p2, p3]))  # True
sage: pc.is_subcomplex(PolyhedralComplex([p1, p2]))  # False
```

join(right)  
The join of this polyhedral complex with another one.

INPUT:

- right – the other polyhedral complex (the right-hand factor)

EXAMPLES:

```python
sage: pc = PolyhedralComplex([Polyhedron(vertices=[[0, 0], [1]])])
sage: pc_join = pc.join(pc)
sage: pc_join
```

(continues on next page)
Polyhedral complex with 1 maximal cell

sage: next(pc_join.maximal_cell_iterator()).vertices()
(A vertex at (0, 0, 0),
A vertex at (0, 0, 1),
A vertex at (0, 1, 1),
A vertex at (1, 0, 0))

maximal_cell_iterator(increasing=False)
An iterator for the maximal cells in this polyhedral complex.

INPUT:

- increasing – (optional, default False) if True, return maximal cells in increasing order of dimension. Otherwise it returns cells in decreasing order of dimension.

Note: Among the cells of a fixed dimension, there is no sorting.

Warning: This may give the wrong answer if the polyhedral complex was constructed with maximality_check set to False.

EXAMPLES:

sage: p1 = Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)])
sage: p2 = Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])
sage: p3 = Polyhedron(vertices=[(1, 2), (0, 2)])
sage: pc = PolyhedralComplex([p1, p2, p3])
sage: len(list(pc.maximal_cell_iterator()))
2

Wrong answer due to maximality_check=False:

sage: pc_invalid = PolyhedralComplex([p1, p2, p3],
.....:     maximality_check=False)
sage: len(list(pc_invalid.maximal_cell_iterator()))
3

maximal_cells()
The maximal cells of this polyhedral complex, in the form of a dictionary: the keys are integers, representing dimension, and the value associated to an integer \(d\) is the set of \(d\)-maximal cells.

Warning: This may give the wrong answer if the polyhedral complex was constructed with maximality_check set to False.

EXAMPLES:
Wrong answer due to `maximality_check=False`:

```
sage: pc_invalid = PolyhedralComplex([p1, p2, p3],
.....:           maximality_check=False)
sage: len(pc_invalid.maximal_cells()[1])
1
```

### maximal_cells_sorted()

Return the sorted list of the maximal cells of this polyhedral complex by non-increasing dimensions.

**EXAMPLES:**

```
sage: pc = PolyhedralComplex([
.....:           Polyhedron(verts=[(1, 1), (0, 0), (1, 2)]),
.....:           Polyhedron(verts=[(1, 2), (0, 0), (0, 2)])])
sage: [p.vertices_list() for p in pc.maximal_cells_sorted()]
[[[0, 0], [0, 2], [1, 2]], [[0, 0], [1, 1], [1, 2]]]
```

### n_maximal_cells(n)

List of maximal cells of dimension n of this polyhedral complex.

**INPUT:**

- n – non-negative integer; the dimension

**Note:** The resulting list need not be sorted. If you want a sorted list of n-cells, use `_n_maximal_cells_sorted()`.

**Warning:** This may give the wrong answer if the polyhedral complex was constructed with `maximality_check` set to `False`.

**EXAMPLES:**

```
sage: p1 = Polyhedron(verts=[(1, 1), (0, 0), (1, 2)])
sage: p2 = Polyhedron(verts=[(1, 2), (0, 0), (0, 2)])
sage: p3 = Polyhedron(verts=[(1, 2), (0, 2)])
sage: pc = PolyhedralComplex([p1, p2, p3])
sage: len(pc.n_maximal_cells(2))
2
sage: len(pc.n_maximal_cells(1))
0
```

Wrong answer due to `maximality_check=False`:

```
sage: pc_invalid = PolyhedralComplex([p1, p2, p3],
.....:           maximality_check=False)
```

(continues on next page)
```python
sage: len(pc_invalid.n_maximal_cells(1))
1
```

**n_skeleton**(n)

The n-skeleton of this polyhedral complex.

The n-skeleton of a polyhedral complex is obtained by discarding all of the cells in dimensions larger than n.

**INPUT:**

- n – non-negative integer; the dimension

**See also:**

`stratify()`

**EXAMPLES:**

```python
sage: pc = PolyhedralComplex([...
....: Polyhedron(vertices=[[1, 1], [0, 0], [1, 2]]),
....: Polyhedron(vertices=[[1, 2], [0, 0], [0, 2]])])
```

```python
sage: pc.n_skeleton(2)
Polyhedral complex with 2 maximal cells
```

```python
sage: pc.n_skeleton(1)
Polyhedral complex with 5 maximal cells
```

```python
sage: pc.n_skeleton(0)
Polyhedral complex with 4 maximal cells
```

**plot**(**kwds)**

Return a plot of the polyhedral complex, if it is of dim at most 3.

**EXAMPLES:**

```python
sage: p1 = Polyhedron(vertices=[[1, 1], [0, 0], [1, 2]])
```

```python
sage: p2 = Polyhedron(vertices=[[1, 2], [0, 0], [0, 2]])
```

```python
sage: pc = PolyhedralComplex([p1, p2])
```

```python
sage: pc.plot()
Graphics object consisting of 10 graphics primitives
```

**product**(right)

The (Cartesian) product of this polyhedral complex with another one.

**INPUT:**

- right – the other polyhedral complex (the right-hand factor)

**OUTPUT:**

- the product self x right

**EXAMPLES:**

```python
sage: pc = PolyhedralComplex([Polyhedron(vertices=[[0], [1]])])
```

```python
sage: pc_square = pc.product(pc)
```

```python
sage: next(pc_square.maximal_cell_iterator()).vertices()
```

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

**relative_boundary_cells()**

Return the maximal cells of the relative-boundary sub-complex.

A point $P$ is in the relative boundary of a set $S$ if $P$ is in the closure of $S$ but not in the relative interior of $S$.

**Warning:** This may give the wrong answer if the polyhedral complex was constructed with `maximality_check` set to False.

EXAMPLES:

```
sage: p1 = Polyhedron(vertices=[[1, 1], (0, 0), (1, 2)])
sage: p2 = Polyhedron(vertices=[[1, 2], (0, 0), (0, 2)])
sage: p3 = Polyhedron(vertices=[[2, 2]])
sage: pc = PolyhedralComplex([p1, p2])
sage: rbd_cells = pc.relative_boundary_cells()
sage: len(rbd_cells)
4
sage: all(p.dimension() == 1 for p in rbd_cells)
True
sage: pc_lower_dim = PolyhedralComplex([p3])
sage: sorted([p.vertices() for p in pc_lower_dim.relative_boundary_cells()])
[(A vertex at (0, 2),), (A vertex at (1, 2),)]
```

Test on polyhedral complex which is not pure:

```
sage: pc_non_pure = PolyhedralComplex([p1, p3, p4])
sage: (set(pc_non_pure.relative_boundary_cells())
....: == set([f.as_polyhedron() for f in p1.faces(1)] + [p3, p4]))
True
```

Test with `maximality_check` == False:

```
sage: pc_invalid = PolyhedralComplex([p2, p3],
....: maximality_check=False)
sage: (set(pc_invalid.relative_boundary_cells())
....: == set([f.as_polyhedron() for f in p2.faces(1)]))
True
```

Test unbounded case:

```
sage: pc3 = PolyhedralComplex([
....: Polyhedron(vertices=[[1,0], [0,1]], rays=[[1,0], [0,1]]),
....: Polyhedron(vertices=[[1,0], [0,-1]], rays=[[1,0], [0,-1]]))
sage: len(pc3.relative_boundary_cells())
4
```
remove_cell\((cell, check=False)\)

Remove cell from self and all the cells that contain cell as a subface.

INPUT:

- cell – a cell of the polyhedral complex
- check – boolean (default: False); if True, raise an error if cell is not a cell of this complex

This does not return anything; instead, it changes the polyhedral complex.

EXAMPLES:

If you add a cell which is already present, there is no effect:

```python
sage: p1 = Polyhedron(vertices=[(1, 1), (0, 0), (1, 2)])
sage: p2 = Polyhedron(vertices=[(1, 2), (0, 0), (0, 2)])
sage: r = Polyhedron(rays=[(1, 0)])
sage: pc = PolyhedralComplex([p1, p2, r])
sage: pc.dimension()
2
sage: pc.remove_cell(Polyhedron(vertices=[(0, 0), (1, 2)]))
sage: pc.dimension()
1
```

set_immutable()

Make this polyhedral complex immutable.

EXAMPLES:

```python
sage: pc = PolyhedralComplex([Polyhedron(vertices=[[0], [1]])])
sage: pc.is_mutable()
True
sage: pc.set_immutable()
sage: pc.is_mutable()
False
```

stratify\((n)\)

Return the pure sub-polyhedral complex which is constructed from the \(n\)-dimensional maximal cells of this polyhedral complex.

See also:

\(n\_skeleton()\)

**Warning:** This may give the wrong answer if the polyhedral complex was constructed with maximality_check set to False.
EXAMPLES:

```python
sage: p1 = Polyhedron(vertices=[[1, 1], [0, 0], [1, 2]])
sage: p2 = Polyhedron(vertices=[[1, 2], [0, 0], [0, 2]])
sage: p3 = Polyhedron(vertices=[[1, 2], [0, 2]])
sage: pc = PolyhedralComplex([p1, p2, p3])
sage: pc.stratify(2) == pc
True
sage: pc.stratify(1)
Polyhedral complex with 0 maximal cells
```

Wrong answer due to `maximality_check=False`:

```python
sage: pc_invalid = PolyhedralComplex([p1, p2, p3],
....:     maximality_check=False)
sage: pc_invalid.stratify(1)
Polyhedral complex with 1 maximal cell
```

**subdivide** (`make_simplicial=False, new_vertices=None, new_rays=None`)

Construct a new polyhedral complex by iterative stellar subdivision of self for each new vertex/ray given.

Currently, subdivision is only supported for bounded polyhedral complex or polyhedral fan.

**INPUT:**

- `make_simplicial` – boolean (default: False): if True, the returned polyhedral complex is simplicial
- `new_vertices, new_rays` – list (optional); new generators to be added during subdivision

**EXAMPLES:**

```python
sage: square_vertices = [(1, 1, 1), (-1, 1, 1), (-1, -1, 1), (1, -1, 1)]
sage: pc = PolyhedralComplex([Polyhedron(vertices=[[0, 0, 0]] + square_vertices),
....: Polyhedron(vertices=[[0, 0, 2]] + square_vertices)])
sage: pc.is_compact() and not pc.is_simplicial_complex()
True
sage: subdivided_pc = pc.subdivide(new_vertices=[[0, 0, 1]])
sage: subdivided_pc
Polyhedral complex with 8 maximal cells
sage: subdivided_pc.is_simplicial_complex()
True
sage: simplicial_pc = pc.subdivide(make_simplicial=True)
sage: simplicial_pc
Polyhedral complex with 4 maximal cells
sage: simplicial_pc.is_simplicial_complex()
True
```

```python
sage: fan = PolyhedralComplex([Polyhedron(rays=square_vertices)])
sage: fan.is_polyhedral_fan() and not fan.is_simplicial_fan()
True
sage: fan.subdivide(new_vertices=[[0, 0, 1]])
Traceback (most recent call last):
...
ValueError: new vertices cannot be used for subdivision
sage: subdivided_fan = fan.subdivide(new_rays=[[0, 0, 1]])
```

(continues on next page)
sage: subdivided_fan
Polyhedral complex with 4 maximal cells
sage: subdivided_fan.is_simplicial_fan()
True
sage: simplicial_fan = fan.subdivide(make_simplicial=True)
sage: simplicial_fan
Polyhedral complex with 2 maximal cells
sage: simplicial_fan.is_simplicial_fan()
True
sage: halfspace = PolyhedralComplex([Polyhedron(rays=[[0, 0, 1]],
......:     lines=[[1, 0, 0], [0, 1, 0]])])
sage: halfspace.is_simplicial_fan()
False
sage: subdiv_halfspace = halfspace.subdivide(make_simplicial=True)
sage: subdiv_halfspace
Polyhedral complex with 4 maximal cells
sage: subdiv_halfspace.is_simplicial_fan()
True

union(right)
The union of this polyhedral complex with another one.

INPUT:

• right – the other polyhedral complex (the right-hand factor)

EXAMPLES:

sage: p1 = Polyhedron(vertices=[(-1, 0), (0, 0), (0, 1)])
sage: p2 = Polyhedron(vertices=[(0, -1), (0, 0), (1, 0)])
sage: p3 = Polyhedron(vertices=[(0, -1), (1, -1), (1, 0)])
sage: pc = PolyhedralComplex([p1]).union(PolyhedralComplex([p3]))
sage: set(pc.maximal_cell_iterator()) == set([p1, p3])
True
sage: pc.union(PolyhedralComplex([p2]))
Polyhedral complex with 3 maximal cells
sage: p4 = Polyhedron(vertices=[(0, -1), (0, 0), (1, 0), (1, -1)])
sage: pc.union(PolyhedralComplex([p4]))
Traceback (most recent call last):
  ... ValueError: the given cells are not face-to-face

union_as_polyhedron()
Return self as a Polyhedron if self is convex.

EXAMPLES:

sage: p1 = Polyhedron(vertices=[[1, 1], (0, 0), (1, 2)])
sage: p2 = Polyhedron(vertices=[[1, 2], (0, 0), (0, 2)])
sage: p3 = Polyhedron(vertices=[[0, 0], (1, 1), (2, 0)])
sage: P = PolyhedralComplex([p1, p2]).union_as_polyhedron()
sage: P.vertices_list()
[[0, 0], [0, 2], [1, 1], [1, 2]]
Todo: Implement the wedge product of two polyhedral complexes.

Todo: Implement the wedge product of two polyhedral complexes.

EXAMPLES:

```python
sage: pc = PolyhedralComplex([Polyhedron(vertices=[[0], [1]])])
sage: pc.wedge(pc)
Traceback (most recent call last):
  ...  
NotImplementedError: wedge is not implemented for polyhedral complex
```

```
sage.geometry.polyhedral_complex.cells_list_to_cells_dict(cells_list)
Helper function that returns the dictionary whose keys are the dimensions, and the value associated to an integer
\(d\) is the set of \(d\)-dimensional polyhedra in the given list.

EXAMPLES:

```python
sage: p1 = Polyhedron(vertices=[[1, 1], [0, 0], [1, 2]])
sage: p2 = Polyhedron(vertices=[[1, 1], [0, 0]])
sage: p3 = Polyhedron(vertices=[[0, 0]])
sage: p4 = Polyhedron(vertices=[[1, 1]])
sage: sage.geometry.polyhedral_complex.cells_list_to_cells_dict([p1, p2, p3, p4])
{0: {A 0-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex, 
     A 0-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex},
 1: {A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices},
 2: {A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 3 vertices}}
```

2.4 Toric geometry

2.4.1 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:


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:

sage: n * m
14
sage: m * n
14

You can also add elements of the same lattice or scale them:

sage: 2 * n
N(2, 4, 6)
sage: n * 2
N(2, 4, 6)
sage: n + n
N(2, 4, 6)
However, you cannot “mix wrong lattices” in your expressions:

```python
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:

```python
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”:

```python
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:

```python
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:

```python
sage: (1,2,3) + (1,2,3)
(1, 2, 3, 1, 2, 3)
```

Instead, use syntax like

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

```python
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
sage: L1 = ToricLattice(3)
sage: L1
3-d lattice N
sage: L1.dual()
sage: L1.dual()
3-d lattice M
```

If we give the name “N” explicitly, the dual lattice will be called “N*”:

```sage
sage: L2 = ToricLattice(3, "N")
sage: L2
3-d lattice N
sage: L2.dual()
sage: L2.dual()
3-d lattice N*
```

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

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

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

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

(continues on next page)
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.

```python
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.

```python
create_object(version, key)
```

Create the toric lattice described by `key`.

See `ToricLattice` for documentation.

**Warning:** You probably should not use this function directly.

```python
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.

```python
Element
```

alias of `sage.geometry.toric_lattice_element.ToricLatticeElement`

```python
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.

OUTPUT:

• 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

2.4. Toric geometry
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:**

```python
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:

```python
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:

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

**intersection(other)**
Return the intersection of `self` and `other`.

**INPUT:**
- `other` - a toric (sub)lattice.dual

**OUTPUT:**
- a toric (sub)lattice.
EXAMPLES:

```
sage: N = ToricLattice(3)
sage: Ns1 = N.submodule([N(2,4,0), N(9,12,0)])
sage: Ns2 = N.submodule([N(1,4,9), N(9,2,0)])
sage: Ns1.intersection(Ns2)
Sublattice <N(54, 12, 0)>
```

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**(sub, check=True, positive_point=None, positive_dual_point=None, **kwds)

Return the quotient of self by the given sublattice sub.

**INPUT:**

- **sub** – sublattice of self;
- **check** – (default: True) whether or not to check that sub is a valid sublattice.

If the quotient is one-dimensional and torsion free, the following two mutually exclusive keyword arguments are also allowed. They decide the sign choice for the (single) generator of the quotient lattice:

- **positive_point** – a lattice point of self not in the sublattice sub (that is, not zero in the quotient lattice). The quotient generator will be in the same direction as positive_point.

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

Further named arguments are passed to the constructor of a toric lattice quotient.

**EXAMPLES:**

```
sage: N = ToricLattice(3)
sage: Ns = N.submodule([N(2,4,0), N(9,12,0)])
sage: Q = N/Ns
sage: Q
Quotient with torsion of 3-d lattice N
by Sublattice <N(1, 8, 0), N(0, 12, 0)>
```

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

```
sage: N = ToricLattice(3)
sage: M = ToricLattice(3, name='M')
sage: Ms = M.submodule([M(2,4,0), M(9,12,0)])
sage: N.quotient(Ms)
Traceback (most recent call last):
...
ValueError: M(1, 8, 0) cannot generate a sublattice of 3-d lattice N
```
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: N.s = N.submodule([[1,2,3], (4,5,6)])
sage: N.s
Sublattice <N(1, 2, 3), N(0, 3, 6)>
sage: N.s_sat = N.s.saturation()
sage: N.s_sat
Sublattice <N(1, 0, -1), N(0, 1, 2)>
sage: N.s_sat is N.s_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 `span_of_basis()`, `submodule()`, and `submodule_with_basis()`.

EXAMPLES:

```
sage: N = ToricLattice(3)
sage: N.s = N.submodule([N.gen(0)])
sage: N.s.span([N.gen(1)])
Sublattice <N(0, 1, 0)>
```
span_of_basis(basis, base_ring=Integer Ring, *args, **kwds)
Return the submodule with the given basis.

INPUT:
• basis – list of elements of the ambient vector space of self.
• base_ring – (default: \(\mathbb{Z}\)) base ring for the generated module.

OUTPUT:
• submodule spanned by basis.

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

See also \(\text{span()}\), \(\text{submodule()}\), and \(\text{submodule_with_basis()}\),

EXAMPLES:

```python
sage: N = ToricLattice(3)
sage: Ns = N.span_of_basis([(1,2,3)])
sage: Ns.span_of_basis([(2,4,0)])
Sublattice \(<N(2, 4, 0)>\)
sage: Ns.span_of_basis([(1/5,2/5,0), (1/7,1/7,0)])
Free module of degree 3 and rank 2 over Integer Ring
User basis matrix:
[1/5 2/5 0]
[1/7 1/7 0]
```

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.
```

class sage.geometry.toric_lattice.ToricLattice_quotient(V, W, check=True, positive_point=None, positive_dual_point=None, **kwds)

Bases: sage.modules.fg_pid.fgp_module.FGP_Module_class

Construct the quotient of a toric lattice \(V\) by its sublattice \(W\).

INPUT:
• \(V\) – ambient toric lattice;
• \(W\) – sublattice of \(V\);
• check – (default: True) whether to check correctness of input or not.
If the quotient is one-dimensional and torsion free, the following two mutually exclusive keyword arguments are also allowed. They decide the sign choice for the (single) generator of the quotient lattice:

- **positive_point** – a lattice point of `self` not in the sublattice `sub` (that is, not zero in the quotient lattice). The quotient generator will be in the same direction as `positive_point`.
- **positive_dual_point** – a dual lattice point. The quotient generator will be chosen such that its lift has a positive product with `positive_dual_point`. Note: if `positive_dual_point` is not zero on the sublattice `sub`, then the notion of positivity will depend on the choice of lift!

Further given named arguments are passed to the constructor of an FGP module.

**OUTPUT:**

- quotient of `V` by `W`.

**EXAMPLES:**

The intended way to get objects of this class is to use `quotient()` method of toric lattices:

```python
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)>
```

Here, `sublattice` happens to be of codimension one in `N`. If you want to prescribe the sign of the quotient generator, you can do either:

```python
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)>
```

```python
sage: M = N.dual()
sage: Q = N.quotient(sublattice, positive_dual_point=M(1,0,0)); Q
1-d lattice, quotient of 3-d lattice N by Sublattice <N(1, 0, 1), N(0, 1, -1)>
```

**Element**

alias of `ToricLattice_quotient_element`

**base_extend(R)**

Return the base change of `self` to the ring `R`.

**INPUT:**

- `R` – either `Z` or `Q`.

**OUTPUT:**

- `self` if `R = Z`, quotient of the base extension of the ambient lattice by the base extension of the sublattice if `R = Q`.

**EXAMPLES:**

```python
```
```python
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:**

```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 = 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:**

```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
```

(continues on next page)
sage: Q.rank()
2

\textbf{dual()}

Return the lattice dual to \texttt{self}.

\textbf{OUTPUT:}

\begin{itemize}
  \item a toric lattice quotient.
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}
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)>
\end{verbatim}

\textbf{gens()}

Return the generators of the quotient.

\textbf{OUTPUT:}

A tuple of ToricLattice_quotient_element generating the quotient.

\textbf{EXAMPLES:}

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

\textbf{is_torsion_free()}

Check if \texttt{self} is torsion-free.

\textbf{OUTPUT:}

\begin{itemize}
  \item True is \texttt{self} has no torsion and False otherwise.
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}
sage: N = ToricLattice(3)
sage: Ns = N.submodule([N(2,4,0), N(9,12,0)])
sage: Q = N/Ns
sage: Q.is_torsion_free()
False
sage: Ns = N.submodule([N(1,4,0)])
sage: Q = N/Ns
sage: Q.is_torsion_free()
True
\end{verbatim}

\textbf{rank()}

Return the rank of \texttt{self}.

\textbf{OUTPUT:}

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

\textbf{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:**

```python
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;

2.4. Toric geometry 265
• gens – list of elements of ambient generating the constructed sublattice;
• see the base class for other available options.

OUTPUT:
• sublattice of a toric lattice with an automatically chosen basis.

See also ToricLattice_sublattice_with_basis if you want to specify an explicit basis.

EXAMPLES:
The intended way to get objects of this class is to use submodule() method of toric lattices:

```
sage: N = ToricLattice(3)
sage: sublattice = N.submodule([(1,1,0), (3,2,1)])
sage: sublattice.has_user_basis()
False
sage: sublattice.basis()
[N(1, 0, 1),
 N(0, 1, -1)]
```

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)

Bases:  
sage.geometry.toric_lattice.ToricLattice_generic,  
sage.modules.free_module.FreeModule_submodule_with_basis_pid

Construct the sublattice of ambient toric lattice with given basis.

INPUT (same as for FreeModule_submodule_with_basis_pid):
• ambient – ambient toric lattice for this sublattice;
• basis – list of linearly independent elements of ambient, these elements will be used as the default basis of the constructed sublattice;
• see the base class for other available options.

OUTPUT:
• sublattice of a toric lattice with a user-specified basis.

See also ToricLattice_sublattice if you do not want to specify an explicit basis.

EXAMPLES:
The intended way to get objects of this class is to use submodule_with_basis() method of toric lattices:
sage: N = ToricLattice(3)

sage: sublattice = N.submodule_with_basis([(1,1,0), (3,2,1)])

sage: sublattice.has_user_basis()
True

sage: sublattice.basis()
[\(N(1, 1, 0)\), \(N(3, 2, 1)\)]

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
```

### 2.4.2 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:

EXAMPLES:

Use `Cone()` to construct cones:

```python
sage: octant = Cone([(1,0,0), (0,1,0), (0,0,1)])
sage: halfspace = Cone([(1,0,0), (0,1,0), (-1,0,0), (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)])
```

For all of the cones above we have provided primitive generating rays, but in fact this is not necessary - a cone can be constructed from any collection of rays (from the same space, of course). If there are non-primitive (or even non-integral) rays, they will be replaced with primitive ones. If there are extra rays, they will be discarded. Of course, this means that `Cone()` has to do some work before actually constructing the cone and sometimes it is not desirable, if you know for sure that your input is already “good”. In this case you can use options `check=False` to force `Cone()` to use exactly the directions that you have specified and `normalize=False` to force it to use exactly the rays that you have specified. However, it is better not to use these possibilities without necessity, since cones are assumed to be represented by a minimal set of primitive generating rays. See `Cone()` for further documentation on construction.

Once you have a cone, you can perform numerous operations on it. The most important ones are, probably, ray accessing methods:

```python
sage: rays = halfspace.rays()
sage: rays
N( 0, 0, 1),
N( 0, 1, 0),
N( 0, -1, 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)
```
The method \texttt{rays()} returns \textit{a PointCollection} with the \(i\)-th element being the primitive integral generator of the \(i\)-th ray. It is possible to convert this collection to a matrix with either rows or columns corresponding to these generators. You may also change the default \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

\begin{verbatim}
sage: for ray in positive_xy: print(ray)
N(1, 0, 0)
N(0, 1, 0)
\end{verbatim}

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:

\begin{verbatim}
sage: positive_xy.dim()
2
sage: positive_xy.lattice_dim()
3
\end{verbatim}

You also may be interested in this dimension:

\begin{verbatim}
sage: dim(positive_xy.linear_subspace())
0
sage: dim(halfspace.linear_subspace())
2
\end{verbatim}

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

\begin{verbatim}
sage: positive_xy.is_strictly_convex()
True
sage: halfspace.is_strictly_convex()
False
\end{verbatim}

You can also perform these checks:

\begin{verbatim}
sage: positive_xy.is simplicial()
True
sage: four_rays.is simplicial()
False
sage: positive_xy.is smooth()
True
\end{verbatim}

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

\begin{verbatim}
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),
\end{verbatim}

(continues on next page)
N(-1, 1, 1)
in 3-d lattice N

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

```
sage: L = four_rays.face_lattice()
sage: [len(s) for s in L.level_sets()]
[1, 4, 4, 1]
sage: face = L.level_sets()[2][0]
sage: face.rays()
N(1, 1, 1),
N(1, -1, 1)
in 3-d lattice N
sage: L.hasse_diagram().neighbors_in(face)
[1-d face of 3-d cone in 3-d lattice N,
 1-d face of 3-d cone in 3-d lattice N]
```

**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:

- [Ful1993]

```
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
```

(continues on next page)
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!
sage: origin = Cone([], lattice=ToricLattice(2))
sage: origin.dim()
0
sage: origin.lattice_dim()
2
sage: origin.lattice()
2-d lattice N

However, the trivial cone in \( n \) dimensions has a predefined constructor for you to use:

sage: origin = cones.trivial(2)
sage: origin.rays()
Empty collection
in 2-d lattice N

Of course, you can also provide lattice in other cases:

sage: L = ToricLattice(3, "L")
sage: c1 = Cone([(1,0,0),(1,1,1)], lattice=L)
sage: c1.rays()
L(1, 0, 0),
L(1, 1, 1)
in 3-d lattice L

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

sage: ray1 = L(1,0,0)
sage: ray2 = L(1,1,1)
sage: c2 = Cone([ray1, ray2])
sage: c2.rays()
L(1, 0, 0),
L(1, 1, 1)
in 3-d lattice L
sage: c1 == c2
True

When the cone in question is not strictly convex, the standard form for the “generating rays” of the linear subspace is “basis vectors and their negatives”, as in the following example:

sage: plane = Cone([(1,0), (0,1), (-1,-1)])
sage: plane.rays()
N( 0, 1),

2.4. Toric geometry
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)
```

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
sage: PointConfiguration.set_engine('internal')
```

We start with a simple case of a non-smooth 2-dimensional cone:

```sage
sage: C = 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
sage: C = Cone([[1,0],[3,4]]).dual().Hilbert_basis()
M(0, 1),
M(4, 3),
M(1, -1),
M(2, -2)
in 2-d lattice M
```

```sage
sage: C = Cone([[1,2,3,4],[0,1,0,7],[3,1,0,2],[0,0,1,0]]).dual()
sage: C.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( 4, -4, 0, 1),
M(-1, 3, 0, 0),
M( 1, -5, 2, 1),
M( 3, -5, 1, 1),
M( 6, -5, 0, 1),
M( 3, -13, 5, 2),
M( 2, -6, 2, 1),
M( 5, -6, 1, 1),
M( 8, -6, 0, 1),
M( 0, 1, 0, 0),
M(-2, 8, 0, -1),
```

(continues on next page)
\[ M(10, -42, 17, 6), \]
\[ M( 7, -28, 11, 4), \]
\[ M( 5, -21, 9, 3), \]
\[ M( 6, -21, 8, 3), \]
\[ M( 5, -14, 5, 2), \]
\[ M( 2, -7, 3, 1), \]
\[ M( 6, -21, 8, 3), \]
\[ M( 5, -14, 5, 2), \]
\[ M( 2, -7, 3, 1), \]
\[ M( 0, 0, 1, 0), \]
\[ M( 1, 0, 0, 0), \]
\[ M(-1, 7, 0, -1), \]
\[ M(-3, 14, 0, -2) \]

in 4-d lattice \( M \)

Not a strictly convex cone:

```
sage: wedge = Cone([ (1,0,0), (1,2,0), (0,0,1), (0,0,-1) ])  
sage: sorted(wedge.semigroup_generators())  
[[(0, 0, -1), (0, 0, 1), (1, 0, 0), (1, 1, 0), (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].

\textbf{Hilbert coefficients}(point, solver, verbose=None, integrality_tolerance=0)

Return the expansion coefficients of point with respect to \( Hilbert\_basis() \).

INPUT:

- **point** – a lattice point in the cone, or something that can be converted to a point. For example, a list or tuple of integers.
- **solver** – (default: None) Specify a Mixed Integer Linear Programming (MILP) solver to be used. If set to None, the default one is used. For more information on MILP solvers and which default solver is used, see the method solve of the class MixedIntegerLinearProgram.
- **verbose** – integer (default: 0). Sets the level of verbosity of the LP solver. Set to 0 by default, which means quiet.
- **integrality_tolerance** – parameter for use with MILP solvers over an inexact base ring; see \( MixedIntegerLinearProgram.get\_values() \).
A \( \mathbf{Z} \)-vector of length \( \text{len}(\text{self}.\text{Hilbert}_\text{basis}()) \) 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:**

```python
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:

```python
sage: N = ToricLattice(2)
sage: cone = Cone([N(1,0),N(1,2)])
sage: cone.Hilbert_basis()
N(1, 0),
N(1, 2),
N(1, 1)
in 2-d lattice N
sage: cone.Hilbert_coefficients( N(1,1) )
(0, 0, 1)
```

The cone need not be strictly convex:

```python
sage: N = ToricLattice(3)
sage: cone = Cone([N(1,0,0),N(1,2,0),N(0,0,1),N(0,0,-1)])
sage: cone.Hilbert_basis()
N(1, 2, 0),
N(1, 0, 0),
N(0, 0, 1),
N(0, 0, -1),
N(1, 1, 0)
in 3-d lattice N
sage: cone.Hilbert_coefficients( N(1,1,3) )
(0, 0, 3, 0, 1)
```

**Z_operators_gens()**

Compute minimal generators of the \( \mathbf{Z} \)-operators on this cone.

The \( \mathbf{Z} \)-operators on a cone generalize the \( \mathbf{Z} \)-matrices over the nonnegative orthant. They are simply negations of the `cross_positive_operators_gens()`.

**OUTPUT:**

A list of \( n \)-by-\( n \) matrices where \( n \) is the ambient dimension of this cone. Each matrix \( L \) in the list has the property that \( s(L(x)) \leq 0 \) whenever \( (x, s) \) is an element of this cone's `discrete_complementarity_set()`.
The returned matrices generate the cone of Z-operators on this cone; that is,

- Any nonnegative linear combination of the returned matrices is a Z-operator on this cone.
- Every Z-operator on this cone is some nonnegative linear combination of the returned matrices.

See also:

cross_positive_operators_gens(), lyapunov_like_basis(), positive_operators_gens()

REFERENCES:

- [BP1994]
- [Or2018b]

adjacent()  
Return faces adjacent to self in the ambient face lattice.

Two distinct faces \( F_1 \) and \( F_2 \) of the same face lattice are adjacent if all of the following conditions hold:

- \( F_1 \) and \( F_2 \) have the same dimension \( d \);
- \( F_1 \) and \( F_2 \) share a facet of dimension \( d - 1 \);
- \( F_1 \) and \( F_2 \) are facets of some face of dimension \( d + 1 \), unless \( d \) is the dimension of the ambient structure.

OUTPUT:

- tuple of cones.

EXAMPLES:

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

```python
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:**

```python
sage: quadrant = Cone([(1,0), (0,1)])
sage: quadrant.ambient_ray_indices()
(0, 1)
sage: quadrant.facets()[1].ambient_ray_indices()
(1,)
```

### an_affine_basis()

Return points in `self` that form a basis for the affine hull.

**EXAMPLES:**

2.4. Toric geometry
sage: quadrant = Cone([[1,0], (0,1)])
sage: quadrant.an_affine_basis()
Traceback (most recent call last):
...  
NotImplementedError: this function is not implemented for unbounded polyhedra
sage: ray = Cone([[1, 1]])
Traceback (most recent call last):
...
SyntaxError: unexpected EOF while parsing
sage: ray.an_affine_basis()
Traceback (most recent call last):
...
NameError: name 'ray' is not defined
sage: line = Cone([[1,0], (-1,0)])
sage: line.an_affine_basis()
Traceback (most recent call last):
...
NotImplementedError: this function is not implemented for unbounded polyhedra

**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 \( \text{discrete_complementarity_set()} \); 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 \( \text{discrete_complementarity_set()} \).

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:

\texttt{lyapunov_like_basis()}, \texttt{positive_operators_gens()}, \texttt{Z_operators_gens()}

REFERENCES:

- [SV1970]
- [Or2018b]

EXAMPLES:
Cross-positive operators on the nonnegative orthant are negations of Z-matrices; that is, matrices whose off-diagonal elements are nonnegative:

```python
sage: K = cones.nonnegative_orthant(2)
sage: K.cross_positive_operators_gens()
```
```
[  [0 1]  [0 0]  [1 0]  [-1 0]  [0 0]  [0 0]
  [0 0],  [1 0],  [0 0],  [0 0],  [0 1],  [0 -1]
]
```

```python
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:

```python
sage: K = cones.trivial(0)
sage: K.cross_positive_operators_gens()
```
```
[]
```

Every operator is a cross-positive operator on the ambient vector space:

```python
sage: K = Cone([(1,),(-1,)])
sage: K.is_full_space()
```
```
True
```

```python
sage: K = Cone([(1,0),(-1,0),(0,1),(0,-1)])
sage: K.cross_positive_operators_gens()
```
```
[  [1 0]  [-1 0]  [0 0]  [0 0]
  [0 0],  [0 0],  [0 0],  [0 0],  [0 0],  [0 0],  [0 1],  [0 -1]
]
```

A non-obvious application is to find the cross-positive operators on the right half-plane [Or2018b]:

```python
sage: K = Cone([(1,0),(0,1),(0,-1)])
sage: K.cross_positive_operators_gens()
```
```
[  [1 0]  [-1 0]  [0 0]  [0 0]
  [0 0],  [0 0],  [0 0],  [0 0],  [0 0],  [0 0],  [0 1],  [0 -1]
]
```

Cross-positive operators on a subspace are Lyapunov-like and vice-versa:

```python
sage: K = Cone([(1,0),(-1,0),(0,1),(0,-1)])
sage: K.is_full_space()
```
```
True
```

```python
sage: lls = span( vector(lls.list())
```

(continues on next page)
...:
for l in K.lyapunov_like_basis() 

sage: cs = span( vector(c.list())
...: for c in K.cross_positive_operators_gens() 

sage: cs == lls
True

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 = cones.nonnegative_orthant(2)
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: cones.trivial(0).discrete_complementarity_set()
()```

`dual()`

Return the dual cone of `self`.

OUTPUT:

- cone.

EXAMPLES:

```
sage: cone = Cone([(1,0), (-1,3)])
sage: cone.dual().rays()
M(0, 1),
M(3, 1)
in 2-d lattice M
```

Now let's look at a more complicated case:

```
sage: cone = Cone([(-2,-1,2), (4,1,0), (-4,-1,-5), (4,1,5)])
sage: cone.is_strictly_convex()
False
sage: cone.dim()
3
sage: cone.dual().rays()
M(7, -18, -2),
M(-4, 0)
in 3-d lattice M
sage: 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)
```

(continues on next page)
**embed**

Return the cone equivalent to the given one, but sitting in `self` as a face.

You may need to use this method before calling methods of `cone` that depend on the ambient structure, such as `ambient_ray_indices()` or `facet_of()`. The cone returned by this method will have `self` as ambient. If `cone` does not represent a valid cone of `self`, `ValueError` exception is raised.

**Note:** This method is very quick if `self` is already the ambient structure of `cone`, so you can use without extra checks and performance hit even if `cone` is likely to sit in `self` but in principle may not.

**INPUT:**
- `cone` – a cone.

**OUTPUT:**
- a cone, equivalent to `cone` but sitting inside `self`.

**EXAMPLES:**

Let’s take a 3-d cone on 4 rays:

```python
sage: c = Cone([(1,0,1), (0,1,1), (-1,0,1), (0,-1,1)])
```

Then any ray generates a 1-d face of this cone, but if you construct such a face directly, it will not “sit” inside the cone:

```python
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 face of the cone, we need to embed it first:

```python
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
```

(continues on next page)
False

```python
sage: e_ray.is_equivalent(ray)
True
sage: e_ray.ambient_ray_indices()
(3,)
sage: e_ray.adjacent()
(1-d face of 3-d cone in 3-d lattice N,
 1-d face of 3-d cone in 3-d lattice N)
sage: e_ray.ambient()
3-d cone in 3-d lattice N
```

Not every cone can be embedded into a fixed ambient cone:

```python
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!
```

**`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:

```python
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:

```python
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...

```python
sage: face = L.level_sets()[1][0]
sage: face.rays()
```
... or you can see the index of the ray of the original cone that corresponds to the above one:

```
sage: face.ambient_ray_indices()
(0,)
sage: quadrant.ray(0)
N(1, 0)
```

An alternative to extracting faces from the face lattice is to use `faces()` method:

```
sage: face is quadrant.faces(dim=1)[0]
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: sorted(D.neighbors(face))
[0-d face of 2-d cone in 2-d lattice N, 2-d cone in 2-d lattice N]
```

However, you can achieve some of this functionality using `facets()`, `facet_of()`, and `adjacent()` methods:

```
sage: 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
```

Note that if `cone` is a face of `supercone`, then the face lattice of `cone` consists of (appropriate) faces of `supercone`:

```
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]
```

(continues on next page)
sage: cone
3-d face of 4-d cone in 4-d lattice N
sage: cone.face_lattice()
Finite poset containing 8 elements with distinguished linear extension
sage: cone.face_lattice().bottom()
0-d face of 4-d cone in 4-d lattice N
sage: cone.face_lattice().top()
3-d face of 4-d cone in 4-d lattice N
sage: cone.face_lattice().top() == cone
True

faces(dim=None, codim=None)
Return faces of self of specified (co)dimension.

INPUT:
• dim – integer, dimension of the requested faces;
• codim – integer, codimension of the requested faces.

Note: You can specify at most one parameter. If you don’t give any, then all faces will be returned.

OUTPUT:
• if either dim or codim is given, the output will be a tuple of cones;
• if neither dim nor codim is given, the output will be the tuple of tuples as above, giving faces of all existing dimensions. If you care about inclusion relations between faces, consider using face_lattice() or adjacent(), facet_of(), and facets().

EXAMPLES:
Let’s take a look at the faces of the first quadrant:

sage: quadrant = Cone([(1,0), (0,1)])

(sage: quadrant.faces())
((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...

(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:

(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:

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:

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,),)
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:
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).facetNormals()  # line in 2d
Empty collection
in 2-d lattice M
sage: Cone([[1,0],(0,1)], lattice=N).facetNormals()  # strictly convex cone
M(0, 1),
M(1, 0)
in 2-d lattice M
sage: Cone([[1,0],(-1,0),(0,1)], lattice=N).facetNormals()  # half space
M(0, 1)
in 2-d lattice M
sage: Cone([[1,0],(0,1),(-1,-1)], lattice=N).facetNormals()  # 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:**

- cone = Cone([-2,-1,2], (4,1,0), (-4,-1,-5), (4,1,5))
- cone.is_strictly_convex()
  False
- cone.dim()
  3
- cone.linear_subspace().dimension()
  1
- lsg = (QQ^3)(cone.linear_subspace().gen(0)); lsg
  (1, 1/4, 5/4)
- cone.facet_normals()
  M(7, -18, -2),
  M(1, -4, 0)
in 3-d lattice M
- [lsg*normal for normal in cone.facet_normals()]
  [0, 0]
sage: octant = Cone([(1,0,0), (0,1,0), (0,0,1)])
sage: octant.facet_of()
()
sage: one_face = octant.facets(1)[0]
sage: len(one_face.facet_of())
2
sage: one_face.facet_of()[1]
2-d face of 3-d cone in 3-d lattice N

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:

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)

incidence_matrix()

Return the incidence matrix.

Note: The columns correspond to facets/facet normals in the order of facet_normals(), the rows correspond to the rays in the order of rays().

EXAMPLES:

sage: octant = Cone([(1,0,0), (0,1,0), (0,0,1)])
sage: octant.incidence_matrix()
[0 1 1]
[1 0 1]
[1 1 0]
sage: halfspace = Cone([(1,0,0), (0,1,0), (-1,-1,0), (0,0,1)])
sage: halfspace.incidence_matrix()
[0]
[1]

(continues on next page)
interior()

Return the interior of self.

OUTPUT:

• either self, an empty polyhedron, or an instance of RelativeInterior.

EXAMPLES:

sage: c = Cone([(1,0,0), (0,1,0)]); c
2-d cone in 3-d lattice N
sage: c.interior()
The empty polyhedron in ZZ^3

sage: origin = cones.trivial(2); origin
0-d cone in 2-d lattice N
sage: origin.interior()
The empty polyhedron in ZZ^2

sage: K = cones.nonnegative_orthant(2); K
2-d cone in 2-d lattice N
sage: K.interior()
Relative interior of 2-d cone in 2-d lattice N

sage: K2 = Cone([(1,0),(-1,0),(0,1),(0,-1)]); K2
2-d cone in 2-d lattice N
sage: K2.interior() is K2
True

interior_contains(*args)

Check if a given point is contained in the interior of self.

For a cone of strictly lower-dimension than the ambient space, the interior is always empty. You probably want to use relative_interior_contains() in this case.

INPUT:

• anything. An attempt will be made to convert all arguments into a single element of the ambient space of self. If it fails, False will be returned.

OUTPUT:

• True if the given point is contained in the interior of self, False otherwise.

EXAMPLES:

sage: c = Cone([(1,0), (0,1)])
sage: c.contains((1,1))
True
sage: c.interior_contains((1,1))
True
sage: c.contains((1,0))
True
sage: c.interior_contains((1,0))
False

**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_compact**()

Checks if the cone has no rays.

OUTPUT:

- True if the cone has no rays, False otherwise.

EXAMPLES:
```python
sage: c0 = cones.trivial(3)
sage: c0.is_trivial()
True
sage: c0.nrays()
0
```

**is_empty()**

Return whether `self` is the empty set.

Because a cone always contains the origin, this method returns `False`.

**EXAMPLES:**

```python
sage: trivial_cone = cones.trivial(3)
sage: trivial_cone.is_empty()
False
```

**is_equivalent(other)**

Check if `self` is “mathematically” the same as `other`.

**INPUT:**

- `other` - cone.

**OUTPUT:**

- True if `self` and `other` define the same cones as sets of points in the same lattice, `False` otherwise.

There are three different equivalences between cones $C_1$ and $C_2$ in the same lattice:

1. They have the same generating rays in the same order. This is tested by $C_1 == C_2$.
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.
• True if self is a face of cone, False otherwise.

EXAMPLES:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: cone1 = Cone([(1,0)])
sage: cone2 = Cone([(1,2)])
sage: quadrant.is_face_of(quadrant)
True
sage: cone1.is_face_of(quadrant)
True
sage: cone2.is_face_of(quadrant)
False
```

Being a face means more than just saturating a facet inequality:

```
sage: octant = Cone([(1,0,0), (0,1,0), (0,0,1)])
sage: cone = Cone([(2,1,0),(1,2,0)])
sage: cone.is_face_of(octant)
False
```

**is_full_dimensional()**
Check if this cone is solid.

A cone is said to be solid if it has nonempty interior. That is, if its extreme rays span the entire ambient space.

An alias is *is_full_dimensional()*. 

OUTPUT:

True if this cone is solid, and False otherwise.

See also:

*is_proper()*

EXAMPLES:

The nonnegative orthant is always solid:

```
sage: quadrant = cones.nonnegative_orthant(2)
sage: quadrant.is_solid()
True
sage: octant = Cone([(1,0,0), (0,1,0), (0,0,1)])
sage: octant.is_solid()
True
```

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_full_space()**
Check if this cone is equal to its ambient vector space.

---

**2.4. Toric geometry**


An alias is `is_universe()`.

OUTPUT:

True if this cone equals its entire ambient vector space and False otherwise.

EXAMPLES:

A single ray in two dimensions is not equal to the entire space:

```
sage: K = Cone([(1,0)])
sage: K.is_full_space()
False
```

Neither is the nonnegative orthant:

```
sage: K = cones.nonnegative_orthant(2)
sage: K.is_full_space()
False
```

The right half-space contains a vector subspace, but it is still not equal to the entire space:

```
sage: K = Cone([(1,0),(-1,0),(0,1)])
sage: K.is_full_space()
False
```

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 \(GL(n, Z)\)-orbit as `other`.

INPUT:

- `other` - cone.

OUTPUT:

- True if `self` and `other` are in the same \(GL(n, Z)\)-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 \(GL(n, Z)\) (and, therefore, correspond to isomorphic affine toric varieties). This is tested by \(C_1.is_isomorphic(C_2)\).

EXAMPLES:

```
sage: conel = Cone([(1,0), (0, 3)])
sage: m = matrix(ZZ, [(1, -5), (-1, 4)]) # a GL(2, ZZ)-matrix
sage: cone2 = Cone( m*r for r in conel.rays() )
sage: conel.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 = cones.nonnegative_orthant(2)
sage: quadrant.is_proper()
True
sage: octant = Cone([(1,0,0), (0,1,0), (0,0,1)])
sage: octant.is_proper()
True
```

However, if we embed the two-dimensional nonnegative quadrant into three-dimensional space, then the resulting cone no longer has interior, so it is not solid, and thus not proper:

```
sage: quadrant = Cone([(1,0,0), (0,1,0)])
sage: quadrant.is_proper()
False
```

Likewise, a half-space contains at least one line, so it is not proper:

```
sage: halfspace = Cone([(1,0), (0,1), (-1,0)])
sage: halfspace.is_proper()
False
```

**is_relatively_open()**

Return whether self is relatively open.

**OUTPUT:**

Boolean.

**EXAMPLES:**

```
sage: K = cones.nonnegative_orthant(3)
sage: K.is_relatively_open()
False
```

```
sage: K1 = Cone([(1,0), (-1,0)]); K1
1-d cone in 2-d lattice N
```

(continues on next page)
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.

An alias is is_full_dimensional().

OUTPUT:
True if this cone is solid, and False otherwise.

See also:

is_proper()

EXAMPLES:

The nonnegative orthant is always solid:

```
sage: quadrant = cones.nonnegative_orthant(2)
sage: quadrant.is_solid()
True
sage: octant = Cone([(1,0,0), (0,1,0), (0,0,1)])
sage: octant.is_solid()
True
```

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: cone1.is_strictly_convex()
True
sage: cone2 = Cone([(1,0), (-1, 0)])
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 = cones.trivial(3)
sage: c0.is_trivial()
True
sage: c0.nrays()
0
```

is_universe() Check if this cone is equal to its ambient vector space.

An alias is is_universe().
OUTPUT:

True if this cone equals its entire ambient vector space and False otherwise.

EXAMPLES:
A single ray in two dimensions is not equal to the entire space:

```
sage: K = Cone([(1,0)])
sage: K.is_full_space()
False
```

Neither is the nonnegative orthant:

```
sage: K = cones.nonnegative_orthant(2)
sage: K.is_full_space()
False
```

The right half-space contains a vector subspace, but it is still not equal to the entire space:

```
sage: K = Cone([(1,0),(-1,0),(0,1)])
sage: K.is_full_space()
False
```

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
```

```
lineality()
```

Return the lineality of this cone.

The lineality of a cone is the dimension of the largest linear subspace contained in that cone.

OUTPUT:

A nonnegative integer; the dimension of the largest subspace contained within this cone.

REFERENCES:

• [Roc1970]

EXAMPLES:

The lineality of the nonnegative orthant is zero, since it clearly contains no lines:

```
sage: K = cones.nonnegative_orthant(3)
sage: K.lineality()
0
```

However, if we add another ray so that the entire x-axis belongs to the cone, then the resulting cone will have lineality one:

```
sage: K = Cone([(1,0,0), (-1,0,0), (0,1,0), (0,0,1)])
sage: K.lineality()
1
```

If our cone is all of \( \mathbb{R}^2 \), then its lineality is equal to the dimension of the ambient space (i.e. two):

```
sage: K = Cone([(1,0,0), (-1,0,0), (0,1,0), (0,0,1)])
sage: K.lineality()
1
```

If our cone is all of \( \mathbb{R}^2 \), then its lineality is equal to the dimension of the ambient space (i.e. two):
Per the definition, the lineality of the trivial cone in a trivial space is zero:

```
sage: K = cones.trivial(0)
sage: K.lineality()
0
```

**linear_subspace()**

Return the largest linear subspace contained inside of *self*.

**OUTPUT:**

- subspace of the ambient space of *self*.

**EXAMPLES:**

```
sage: halfplane = Cone([(1,0), (0,1), (-1,0)])
sage: halfplane.linear_subspace()
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 0]
sage: fullplane = Cone([(1,0), (0,1), (-1,-1)])
sage: fullplane.linear_subspace()
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[1 0]
```

**lines()**

Return lines generating the linear subspace of *self*.

**OUTPUT:**

- tuple of primitive vectors in the lattice of *self* giving directions of lines that span the linear subspace of *self*. These lines are arbitrary, but fixed. If you do not care about the order, see also `line_set()`.

**EXAMPLES:**

```
sage: halfplane = Cone([(1,0), (0,1), (-1,0)])
sage: halfplane.lines()
N(1, 0)
in 2-d lattice N
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:

\texttt{cross_positive_operators_gens()}, \texttt{positive_operators_gens()}, \texttt{Z_operators_gens()}

REFERENCES:

- [Or2017]
- [RNPA2011]

EXAMPLES:

Every transformation is Lyapunov-like on the trivial cone:

```sage
sage: K = cones.trivial(2)
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:

```sage
sage: K = Cone([\( (1,0), (-1,0), (0,1), (0,-1) \) ])
sage: K.is_full_space()
True
sage: M = MatrixSpace(K.lattice().base_field(), K.lattice_dim())
sage: list(M.basis()) == K.lyapunov_like_basis()
True
```

However, in a trivial space, there are no non-trivial linear maps, so there can be no Lyapunov-like basis:

```sage
sage: K = cones.trivial(0)
sage: K.lyapunov_like_basis()
[]
```

The Lyapunov-like transformations on the nonnegative orthant are diagonal matrices:

```sage
sage: K = cones.nonnegative_orthant(1)
sage: K.lyapunov_like_basis()
[[1]]

sage: K = cones.nonnegative_orthant(2)
sage: K.lyapunov_like_basis()
[
  [1 0] [0 0]
  [0 0], [0 1]
]

sage: K = cones.nonnegative_orthant(3)
sage: K.lyapunov_like_basis()
[
  [1 0 0] [0 0 0] [0 0 0]
  [0 0 0] [0 1 0] [0 0 0]
  [0 0 0], [0 0 0], [0 0 1]
]
```

Only the identity matrix is Lyapunov-like on the pyramids defined by the one- and infinity-norms [RNPA2011]:
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 is is_proper(), then that upper bound reduces from \( n^2 \) to \( n \). A Lyapunov rank of \( n - 1 \) is not possible (by Lemma 6 [Or2017]) in either case.

ALGORITHM:
Algorithm 3 [Or2017] is used. Every closed convex cone is isomorphic to a Cartesian product of a proper cone, a subspace, and a trivial cone. The Lyapunov ranks of the subspace and trivial cone are easy to compute. Essentially, we “peel off” those easy parts of the cone and compute their Lyapunov ranks separately. We then compute the rank of the proper cone by counting a lyapunov_like_basis() for it. Summing the individual ranks gives the Lyapunov rank of the original cone.

REFERENCES:
• [GT2014]
• [Or2017]
• [RNPA2011]

EXAMPLES:
The Lyapunov rank of the nonnegative orthant is the same as the dimension of the ambient space [RNPA2011]:
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()
21
sage: K.lattice_dim()**2 - K.lattice_dim() + 1
21
```

A subspace of dimension $m$ in an $n$-dimensional ambient space has Lyapunov rank $n^2 - m(n - m)$ [Or2017]:

```
sage: e1 = vector(QQ, [1,0,0,0,0])
sage: e2 = vector(QQ, [0,1,0,0,0])
sage: z = (0,0,0,0,0)
sage: K = Cone([e1, -e1, e2, -e2, z, z, z])
sage: K.lyapunov_rank()
19
sage: K.lattice_dim()**2 - K.dim() * K.codim()
19
```

Lyapunov rank is additive on a product of proper cones [RNPA2011]:

```
sage: l31 = Cone([(1,0,1), (0,-1,1), (-1,0,1), (0,1,1)])
sage: octant = Cone([(1,0,0), (0,1,0), (0,0,1)])
sage: K = l31.cartesian_product(octant)
sage: K.lyapunov_rank()
4
sage: l31.lyapunov_rank() + octant.lyapunov_rank()
4
```

Two linearly-isomorphic cones have the same Lyapunov rank [RNPA2011]. A cone linearly-isomorphic to the nonnegative octant will have Lyapunov rank 3:
Lyapunov rank is invariant under \texttt{dual()} [RNPA2011]:

\begin{verbatim}
sage: K = Cone([(2,2,4), (-1,9,0), (2,0,6)])
sage: K.lyapunov_rank() == K.dual().lyapunov_rank()
True
\end{verbatim}

\textbf{orthogonal_sublattice}(*\texttt{args}, **\texttt{kwds})

The sublattice (in the dual lattice) orthogonal to the sublattice spanned by the cone.

Let $M = \text{self.dual_lattice()}$ be the lattice dual to the ambient lattice of the given cone $\sigma$. Then, in the notation of [Ful1993], this method returns the sublattice $M(\sigma) \equiv \sigma^\perp \cap M$.

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:

\begin{verbatim}
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.orthogonal_sublattice()
Sublattice <M(1, 0, -1)>
\end{verbatim}

\textbf{plot}(**\texttt{options})

Plot \texttt{self}.

INPUT:

- any options for toric plots (see \texttt{toric_plotter.options}), none are mandatory.

OUTPUT:

- a plot.

EXAMPLES:

\begin{verbatim}
sage: quadrant = Cone([(1,0), (0,1)])
sage: quadrant.plot()
Graphics object consisting of 9 graphics primitives
\end{verbatim}

\textbf{polyhedron}()  

Return the polyhedron associated to \texttt{self}.

Mathematically this polyhedron is the same as \texttt{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([], lattice=ZZ^2)
sage: origin.polyhedron()
A 0-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex
```

**positive_operators_gens(K2=None)**

Compute minimal generators of the positive operators on this cone.

A linear operator on a cone is positive if the image of the cone under the operator is a subset of the cone. This concept can be extended to two cones: the image of the first cone under a positive operator is a subset of the second cone, which may live in a different space.

The positive operators (on one or two fixed cones) themselves form a closed convex cone. This method computes and returns the generators of that cone as a list of matrices.

**INPUT:**

- *K2* – (default: self) the codomain cone; the image of this cone under the returned generators is a subset of K2.

**OUTPUT:**

A list of \(m\)-by-\(n\) matrices where \(m\) is the ambient dimension of K2 and \(n\) is the ambient dimension of this cone. Each matrix \(P\) in the list has the property that \(P(x)\) is an element of K2 whenever \(x\) is an element of this cone.

The returned matrices generate the cone of positive operators from this cone to K2; that is,

- Any nonnegative linear combination of the returned matrices sends elements of this cone to K2.
- Every positive operator on this cone (with respect to K2) is some nonnegative linear combination of the returned matrices.

**ALGORITHM:**

Computing positive operators directly is difficult, but computing their dual is straightforward using the generators of Berman and Gaiha. We construct the dual of the positive operators, and then return the dual of that, which is guaranteed to be the desired positive operators because everything is closed, convex, and polyhedral.

**See also:**

cross_positive_operators_gens(), lyapunov_like_basis(), Z_operators_gens()

**REFERENCES:**

- [BG1972]
EXAMPLES:

Positive operators on the nonnegative orthant are nonnegative matrices:

```python
sage: K = Cone([(1,)])
sage: K.positive_operators_gens()
[[1]]
```

```python
sage: K = Cone([(1,0),(0,1)])
sage: K.positive_operators_gens()
[[0 0]
[0 0], [0 0], [1 0], [0 1]]
```

The trivial cone in a trivial space has no positive operators:

```python
sage: K = cones.trivial(0)
sage: K.positive_operators_gens()
[]
```

Every operator is positive on the trivial cone:

```python
sage: K = cones.trivial(1)
sage: K.positive_operators_gens()
[[1], [-1]]
```

```python
sage: K = cones.trivial(2)
sage: K.is_trivial()
True
```

```python
sage: K = cones.trivial(2)
sage: K.positive_operators_gens()

```

Every operator is positive on the ambient vector space:

```python
sage: K = Cone([(1,0),(-1,1)])
sage: K.is_full_space()
True
```

```python
sage: K = Cone([(1,0),(-1,0),(0,1),(0,-1)])
sage: K.is_full_space()
True
```

```python
sage: K = Cone([(1,0),(-1,0),(0,1),(0,-1)])
sage: K.positive_operators_gens()
```

2.4. Toric geometry
A non-obvious application is to find the positive operators on the right half-plane [Or2018b]:

```python
sage: K = Cone([(1,0),(0,1),(0,-1)])
sage: K.positive_operators_gens()
[[1 0] [0 0] [ 0 0] [0 0] [ 0 0]
 [0 0], [1 0], [-1 0], [0 1], [ 0 -1]]
```

**random_element**(ring=Integer Ring)

Return a random element of this cone.

All elements of a convex cone can be represented as a nonnegative linear combination of its generators. A random element is thus constructed by assigning random nonnegative weights to the generators of this cone. By default, these weights are integral and the resulting random element will live in the same lattice as the cone.

The random nonnegative weights are chosen from ring which defaults to ZZ. When ring is not ZZ, the random element returned will be a vector. Only the rings ZZ and QQ are currently supported.

**INPUT:**

- **ring** – (default: ZZ) the ring from which the random generator weights are chosen; either ZZ or QQ.

**OUTPUT:**

Either a lattice element or vector contained in both this cone and its ambient vector space. If ring is ZZ, a lattice element is returned; otherwise a vector is returned. If ring is neither ZZ nor QQ, then a NotImplementedError is raised.

**EXAMPLES:**

The trivial element () is always returned in a trivial space:

```python
sage: set_random_seed()
sage: K = cones.trivial(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:
```

```python
sage: set_random_seed()
sage: K = cones.trivial(3)
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:
```

```python
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):
  ...  
NotImplementedError: ring must be either ZZ or QQ.
```

**relative_interior()**

Return the relative interior of self.

**OUTPUT:**

- either self or an instance of `RelativeInterior`.

**EXAMPLES:**

```
sage: c = Cone([(1,0,0), (0,1,0)]); c
2-d cone in 3-d lattice N
sage: c.relative_interior()
Relative interior of 2-d cone in 3-d lattice N
sage: origin = cones.trivial(2); origin
0-d cone in 2-d lattice N
sage: origin.relative_interior() is origin
True
sage: K1 = Cone([(1,0), (-1,0)]); K1
1-d cone in 2-d lattice N
sage: K1.relative_interior() is K1
True
sage: K2 = Cone([(1,0), (-1,0), (0,1), (0,-1)]); K2
2-d cone in 2-d lattice N
sage: K2.relative_interior() is K2
True
```

**relative_interior_contains(*args)**

Check if a given point is contained in the relative interior of self.

For a full-dimensional cone the relative interior is simply the interior, so this method will do the same check as `interior_contains()`. For a strictly lower-dimensional cone, the relative interior is the cone without its facets.

**INPUT:**

- anything. An attempt will be made to convert all arguments into a single element of the ambient space of self. If it fails, False will be returned.

**OUTPUT:**

- True if the given point is contained in the relative interior of self. False otherwise.

**EXAMPLES:**

```
sage: c = Cone([(1,0,0), (0,1,0)])
sage: c.contains((1,1,0))
```

(continues on next page)
Relative orthogonal quotient ($C_{\rho}$)

The quotient of the dual spanned lattice by the dual of the supercone’s spanned lattice.

In the notation of [Ful1993], if $C_{\rho} > C_{\sigma} = C_{self}$ is a cone that contains $C_{\sigma}$ as a face, then $M(\rho) = supercone.orthogonal_sublattice()$ is a saturated sublattice of $M(\sigma) = self.orthogonal_sublattice()$. This method returns the quotient lattice. The lifts of the quotient generators are $\dim(\rho) - \dim(\sigma)$ linearly independent $M$-lattice lattice points that, together with $M(\rho)$, generate $M(\sigma)$.

OUTPUT:

- toric lattice quotient.

If we call the output $M_{\rho}$, then

- $M_{\rho}.cover() == self.orthogonal_sublattice()$, and
- $M_{\rho}.relations() == supercone.orthogonal_sublattice()$.

Note:

- $M(\sigma)/M(\rho)$ has no torsion since the sublattice $M(\rho)$ is saturated.
- In the codimension one case, (a lift of) the generator of $M(\sigma)/M(\rho)$ is chosen to be positive on $\sigma$.

Examples:

```
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:
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)>
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],)

Sign choice in the codimension one case:
sage: sigma1 = Cone([(1, 2, 3), (1, -1, 1), (-1, 1, 1), (-1, -1, 1)])  # 3d
sage: sigma2 = Cone([(1, 1, -1), (1, 2, 3), (1, -1, 1), (1, -1, -1)])  # 3d
sage: rho = sigma1.intersection(sigma2)
sage: rho.relative_orthogonal_quotient(sigma1).gens()
(M[-5, -2, 3],)
sage: rho.relative_orthogonal_quotient(sigma2).gens()
(M[5, 2, -3],)

relative_quotient(subcone)
The quotient of the spanned lattice by the lattice spanned by a subcone.

In the notation of [Ful1993], let \( N \) be the ambient lattice and \( N_\sigma \) the sublattice spanned by the given cone \( \sigma \). If \( \rho < \sigma \) is a subcone, then \( N_\rho = \text{rho.sublattice()} \) is a saturated sublattice of \( N_\sigma = \text{self.sublattice()} \). This method returns the quotient lattice. The lifts of the quotient generators are \( \dim(\sigma) - \dim(\rho) \) linearly independent primitive lattice points that, together with \( N_\rho \), 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:
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, 3), N(0, -1, 0, 0), N(-1, -1, 0, 0)>
sage: rho.sublattice()
Sublattice <N(-1, -1, 1, 3), N(0, 1, 0, 0)>
sage: sigma.relative_orthogonal_quotient(rho)
1-d lattice, quotient
of Sublattice <N(1, 1, 3), N(0, -1, 0, 0), N(-1, -1, 0, 0)>
by Sublattice <N(1, 0, -1, -3), N(0, 1, 0, 0)>
sage: sigma.relative_orthogonal_quotient(rho).gens()
(N[1, 0, 0, 0],)

More complicated example:
sage: rho = Cone([(1, 2, 3), (1, -1, 1)])
sage: sigma = Cone([(1, 2, 3), (1, -1, 1), (-1, 1, 1), (-1, -1, 1)])
sage: N_sigma = sigma.sublattice()
sage: N_sigma
Sublattice <N(1, 2, 3), N(1, -1, 1), N(-1, -1, -2)>
sage: N_rho = rho.sublattice()
sage: N_rho
Sublattice <N(1, -1, 1), N(1, 2, 3)>
sage: sigma.relative_quotient(rho).gens()
(N[-1, -1, -2],)
sage: N = rho.lattice()
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)
sage: rho.sublattice()
Sublattice <N(1, -1, 1), N(1, 2, 3)>
sage: sigma1.relative_quotient(rho)
1-d lattice, quotient
of Sublattice <N(1, 2, 3), N(1, -1, 1), N(-1, -1, -2)>
by Sublattice <N(1, 2, 3), N(0, 3, 2)>
sage: sigma1.relative_quotient(rho).gens()
(N[-1, -1, -2],)
sage: sigma2.relative_quotient(rho).gens()
(N[0, 2, 1],)

\textbf{seimgroup\_generators()}

Return generators for the semigroup of lattice points of \texttt{self}.

\textbf{OUTPUT:}

- a \texttt{PointCollection} of lattice points generating the semigroup of lattice points contained in \texttt{self}.

\textbf{Note:} No attempt is made to return a minimal set of generators, see \texttt{Hilbert\_basis()} for that.

\textbf{EXAMPLES:}

The following command ensures that the output ordering in the examples below is independent of TOPCOM, you don't have to use it:

\texttt{sage: PointConfiguration.set_engine('internal')}

We start with a simple case of a non-smooth 2-dimensional cone:

\texttt{sage: Cone([(1,0), (1,2)]).seimgroup\_generators()}
N(1, 1),
N(1, 0),
N(1, 2)
in 2-d lattice N
A non-simplicial cone works, too:

```python
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:

```python
sage: cone = Cone([[1,2,3,4],[0,1,0,7],[3,1,0,2],[0,0,1,0]]).dual()
sage: len(cone.semigroup_generators())
2806
```

The cone need not be strictly convex:

```python
sage: halfplane = Cone(((1,0),(2,1),(-1,0)))
sage: sorted(halfplane.semigroup_generators())
[N(-1, 0), N(0, 1), N(1, 0)]
sage: line = Cone(((1,1,1),(-1,-1,-1)))
sage: sorted(line.semigroup_generators())
[N(-1, -1, -1), N(1, 1, 1)]
sage: wedge = Cone(((1,0,0),(1,2,0),(0,0,1),(0,0,-1))
sage: sorted(wedge.semigroup_generators())
[N(0, 0, -1), N(0, 0, 1), N(1, 0, 0), N(1, 1, 0), N(1, 2, 0)]
```

Nor does it have to be full-dimensional (see trac ticket #11312):

```python
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:

```python
sage: A = matrix(((1, 3, 0), (-1, 0, 1), (1, 1, -2), (15, -2, 0))
sage: A.elementary_divisors()
[1, 1, 1, 0]
sage: cone3d = Cone(((3,0,-1), (1,-1,0), (0,1,0), (0,0,1)))
sage: rays = ( A*vector(v) for v in cone3d.rays() )
sage: gens = Cone(rays).semigroup_generators(); sorted(gens)
[N(-2, -1, 0, 17),
 N(0, 1, -2, 0),
 N(1, -1, 1, 15),
 N(3, -4, 5, 45),
 N(3, 0, 1, -2)]
sage: set(map(tuple,gens)) == set( tuple(A*r) for r in cone3d.semigroup_generators() )
True
```

**ALGORITHM:**

If the cone is not simplicial, it is first triangulated. Each simplicial subcone has the integral points of the spanned 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

\[
\text{par}\langle x_1, \ldots, x_d \rangle = \mathbb{Z}^n \cap \{q_1 x_1 + \cdots + q_d x_d : 0 \leq q_i < 1\}
\]
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
sage: K = Cone([(1,0,0), (0,1,0)])
sage: K.solid_restriction().rays()
N(0, 1),
N(1, 0)
in 2-d lattice N
```

The solid restriction of a single ray has the same representation regardless of the ambient space:

```sage
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
sage: K = cones.trivial(0)
sage: K.solid_restriction()
0-d cone in 0-d lattice N
```

The solid restriction of a solid cone is itself:

```sage
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()
N(1)
in 1-d lattice N
sage: line = Cone([(1,0), (-1,0)])
sage: ssc = line.strict_quotient()
sage: ssc
0-d cone in 1-d lattice N
sage: ssc.rays()
Empty collection
in 1-d lattice N
```

The quotient of the trivial cone is trivial:

```
sage: K = cones.trivial(0)
sage: K.strict_quotient()
0-d cone in 0-d lattice N
sage: K = Cone([(0,0,0,0)])
sage: K.strict_quotient()
0-d cone in 4-d lattice N
```

`sublattice(*args, **kwds)`

The sublattice spanned by the cone.

Let $\sigma$ be the given cone and $N = \text{self.lattice()}$ the ambient lattice. Then, in the notation of [Ful1993], 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 `basis()` method on the result of `rays()`.
• The returned lattice points are usually not rays of the cone. In fact, for a non-smooth cone the rays do not generate the sublattice $N_\sigma$, 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.sublattice()
Sublattice <N(1, 1, 1), N(0, -1, 0), N(-1, -1, 0)>
sage: matrix( cone.sublattice().gens() ).det()
-1

Another example:

sage: c = Cone([(1,2,3), (4,-5,1)])
sage: c
2-d cone in 3-d lattice N
sage: c.rays()
N(1, 2, 3),
N(4, -5, 1)
in 3-d lattice N
sage: c.sublattice()
Sublattice <N(4, -5, 1), N(1, 2, 3)>
sage: c.sublattice(5, -3, 4)
N(5, -3, 4)
sage: c.sublattice(1, 0, 0)
Traceback (most recent call last):
...
TypeError: element [1, 0, 0] is not in free module

\texttt{sublattice\_complement(*args, **kwds)}
A complement of the sublattice spanned by the cone.

In other words, \texttt{sublattice()} and \texttt{sublattice\_complement()} together form a Z-basis for the ambient
\texttt{lattice()}. In the notation of [Ful1993], let $\sigma$ be the given cone and $N =$ \texttt{self.lattice()} the ambient lattice. Then
this method returns

\[ N(\sigma) \overset{\text{def}}{=} N/N_\sigma \]

lifted (non-canonically) to a sublattice of $N$.

\textbf{INPUT:}

\begin{itemize}
  \item either nothing or something that can be turned into an element of this lattice.
\end{itemize}

\textbf{OUTPUT:}

\begin{itemize}
  \item if no arguments were given, a \textit{toric sublattice}, otherwise the corresponding element of it.
\end{itemize}

\textbf{EXAMPLES:}

\begin{verbatim}
sage: C2_Z2 = Cone([(1,0),(1,2)])  # C^2/Z_2
sage: c1, c2 = C2_Z2.facets()
sage: c2.sublattice()
Sublattice <N(1, 2)>
\end{verbatim}

(continues on next page)
A more complicated example:

```python
sage: c = Cone([(1,2,3), (4,-5,1)])
sage: c.sublattice()  # Sublattice <N(4, -5, 1), N(1, 2, 3)>

sage: m = matrix( c.sublattice().gens() + c.sublattice_complement().gens() )
sage: m

[  4  -5   1]
[   1   2   3]
[   2  -3   0]

sage: m.det()  # Sublattice <N(2, -3, 0)>
-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.

ambient_dim()  
Return the dimension of the ambient lattice of self.  
An alias is ambient_dim().  

OUTPUT:  
• integer.

EXAMPLES:

sage: c = Cone([[1,0]])  
sage: c.lattice_dim()  
2  
sage: c.dim()  
1

ambient_vector_space(base_field=None)  
Return the ambient vector space.  
It is the ambient lattice (lattice()) tensored with a field.  

INPUT:  
• base_field – (default: the rationals) a field.

EXAMPLES:

sage: c = Cone([[1,0]])  
sage: c.ambient_vector_space()  
Vector space of dimension 2 over Rational Field  
sage: c.ambient_vector_space(AA)  
Vector space of dimension 2 over Algebraic Real Field

cartesian_product(other, lattice=None)  
Return the Cartesian product of self with other.  

INPUT:  
• other – an IntegralRayCollection;  
• lattice – (optional) the ambient lattice for the result. By default, the direct sum of the ambient lattices of self and other is constructed.

OUTPUT:  
• an IntegralRayCollection.
By the Cartesian product of ray collections \((r_0, \ldots, r_{n-1})\) and \((s_0, \ldots, s_{m-1})\) we understand the ray collection of the form \(((r_0, 0), \ldots, (r_{n-1}, 0), (0, s_0), \ldots, (0, s_{m-1}))\), which is suitable for Cartesian products of cones and fans. The ray order is guaranteed to be as described.

**EXAMPLES:**

```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 \texttt{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 \texttt{self}.

**See also:**

\texttt{dim()}, \texttt{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 = cones.nonnegative_orthant(3)
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:

```python
sage: K = cones.trivial(0)
sage: K.lattice_dim()
0
sage: K.codim()
```

(continues on next page)
0

```python
sage: K = cones.trivial(1)
sage: K.lattice_dim()
1
sage: K.codim()
1

sage: K = cones.trivial(2)
sage: K.lattice_dim()
2
sage: K.codim()
2
```

codimension()

Return the codimension of self.

The codimension of a collection of rays (of a cone/fan) is the difference between the dimension of the ambient space and the dimension of the subspace spanned by those rays (of the cone/fan).

OUTPUT:

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 = cones.nonnegative_orthant(3)
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:

```python
sage: K = cones.trivial(0)
sage: K.lattice_dim()
```
sage: K = cones.trivial(1)
sage: K.lattice_dim()
1
sage: K.codim()
1
sage: K = cones.trivial(2)
sage: K.lattice_dim()
2
sage: K.codim()
2

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 \texttt{lattice()} has a \texttt{dual()} method), the dual lattice is returned. Otherwise, \(\mathbb{Z}^n\) is returned, where \(n\) is the dimension of \texttt{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.
An alias is ambient_dim().

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:

```python
sage: c = Cone([(1,0), (0,1)])
sage: c.ray(0)
N(1, 0)
```

`rays(*args)`

Return (some of the) rays of self.

INPUT:

- ray_list – a list of integers, the indices of the requested rays. If not specified, all rays of self will be returned.

OUTPUT:

- a `PointCollection` of primitive integral ray generators.

EXAMPLES:

```python
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:

```python
sage: c.rays(0, 2)
N( 0, 1),
N(-1, 0)
in 2-d lattice N
```

`span(base_ring=None)`

Return the span of self.

INPUT:

- base_ring – (default: from lattice) the base ring to use for the generated module.

OUTPUT:

A module spanned by the generators of self.

EXAMPLES:

The span of a single ray is a one-dimensional sublattice:

```python
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:

sage: K = cones.nonnegative_orthant(3)
sage: K.span() == K.lattice()
True

By specifying a base_ring, we can obtain a vector space:

sage: K = Cone([(1,0,0),(0,1,0),(0,0,1)])
sage: K.span(base_ring=QQ)
Vector space of degree 3 and dimension 3 over Rational Field
Basis matrix:
[1 0 0]
[0 1 0]
[0 0 1]

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 [CLS2011] for the definition. We return the unique \((d, k)\) with minimal \(k\), see Proposition 10.1.3 of [CLS2011].

EXAMPLES:

sage: ray0 = vector([1,0])
sage: ray1 = vector([2,3])
sage: from sage.geometry.cone import classify_cone_2d
sage: classify_cone_2d(ray0, ray1)
(3, 2)

sage: ray0 = vector([2,4,5])
sage: ray1 = vector([5,19,11])
sage: classify_cone_2d(ray0, ray1)
(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])
1
sage: integral_length([2, 2, 4])
2
sage: integral_length([2/3, 2, 4])
2/3
```

```
sage: from sage.geometry.cone import is_Cone
sage: is_Cone(1)
False
sage: quadrant = Cone([(1,0), (0,1)])
sage: is_Cone(quadrant)
True
```

```python
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())
```

2.4. Toric geometry
\[L^*(0, 1), L^*(0, 1), L^*(3, 2), L^*(3, 14)\]

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()
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**:
Likewise for the number of rays when \( \text{min}_\text{rays} = \text{max}_\text{rays} \):

```
sage: set_random_seed()
sage: K = random_cone(min_rays=3, max_rays=3)
sage: K.nrays()
3
```

If we specify a lattice, then the returned cone will live in it:

```
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:

```
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:

```
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:

```
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
```
2.4.3 Catalog of common polyhedral convex cones

This module provides shortcut functions, grouped under the globally-available `cones` prefix, to create some common cones:

- The nonnegative orthant,
- The rearrangement cone of order $p$,
- The Schur cone,
- The trivial cone.

At the moment, only convex rational polyhedral cones are supported—specifically, those cones that can be built using the `Cone()` constructor. As a result, each shortcut method can be passed either an ambient dimension `ambient_dim`, or a toric lattice (from which the dimension can be inferred) to determine the ambient space.

Here are some typical usage examples:

```sage
sage: cones.nonnegative_orthant(2).rays()
N(1, 0),
N(0, 1)
in 2-d lattice N
```

```sage
sage: cones.rearrangement(2,2).rays()
N( 1, 0),
N( 1, -1),
N(-1, 1)
in 2-d lattice N
```

```sage
sage: cones.schur(3).rays()
N(1, -1, 0),
N(0, 1, -1)
in 3-d lattice N
```

```sage
sage: cones.trivial(3).rays()
Empty collection
in 3-d lattice N
```

To specify some other lattice, pass it as an argument to the function:

```sage
sage: K = cones.nonnegative_orthant(3)
sage: cones.schur(lattice=K.dual().lattice())
2-d cone in 3-d lattice M
```

For more information about these cones, see the documentation for the individual functions and the references therein.

```
sage.geometry.cone_catalog.nonnegative_orthant(ambient_dim=None, lattice=None)

The nonnegative orthant in `ambient_dim` dimensions, or living in `lattice`.

The nonnegative orthant consists of all componentwise-nonnegative vectors. It is the convex-conic hull of the standard basis.

INPUT:

- `ambient_dim` – a nonnegative integer (default: None); the dimension of the ambient space
- `lattice` – a toric lattice (default: None); the lattice in which the cone will live
```

2.4. Toric geometry
If `ambient_dim` is omitted, then it will be inferred from the rank of `lattice`. If the `lattice` is omitted, then the default lattice of rank `ambient_dim` will be used.

A `ValueError` is raised if neither `ambient_dim` nor `lattice` are specified. It is also a `ValueError` to specify both `ambient_dim` and `lattice` unless the rank of `lattice` is equal to `ambient_dim`.

OUTPUT:

A `ConvexRationalPolyhedralCone` living in `lattice` and having `ambient_dim` standard basis vectors as its generators. Each generating ray has the integer ring as its base ring.

A `ValueError` can be raised if the inputs are incompatible or insufficient. See the INPUT documentation for details.

REFERENCES:

• Chapter 2 in [BV2009] (Examples 2.4, 2.14, and 2.23 in particular)

EXAMPLES:

```
sage: cones.nonnegative_orthant(3).rays()
N(1, 0, 0),
N(0, 1, 0),
N(0, 0, 1)
in 3-d lattice N
```

**sage.geometry.cone_catalog.rearrangement**(*p*, `ambient_dim=None`, `lattice=None`)

The rearrangement cone of order `p` in `ambient_dim` dimensions, or living in `lattice`.

The rearrangement cone of order `p` in `ambient_dim` dimensions consists of all vectors of length `ambient_dim` whose smallest `p` components sum to a nonnegative number.

For example, the rearrangement cone of order one has its single smallest component nonnegative. This implies that all components are nonnegative, and that therefore the rearrangement cone of order one is the nonnegative orthant in its ambient space.

When `p` and `ambient_dim` are equal, all components of the cone's elements must sum to a nonnegative number. In other words, the rearrangement cone of order `ambient_dim` is a half-space.

INPUT:

• `p` – a nonnegative integer; the number of components to “rearrange”, between 1 and `ambient_dim` inclusive

• `ambient_dim` – a nonnegative integer (default: `None`); the dimension of the ambient space

• `lattice` – a toric lattice (default: `None`); the lattice in which the cone will live

If `ambient_dim` is omitted, then it will be inferred from the rank of `lattice`. If the `lattice` is omitted, then the default lattice of rank `ambient_dim` will be used.

A `ValueError` is raised if neither `ambient_dim` nor `lattice` are specified. It is also a `ValueError` to specify both `ambient_dim` and `lattice` unless the rank of `lattice` is equal to `ambient_dim`.

It is also a `ValueError` to specify a non-integer `p`.

OUTPUT:

A `ConvexRationalPolyhedralCone` representing the rearrangement cone of order `p` living in `lattice`, with ambient dimension `ambient_dim`. Each generating ray has the integer ring as its base ring.

A `ValueError` can be raised if the inputs are incompatible or insufficient. See the INPUT documentation for details.

ALGORITHM:
Suppose that the ambient space is of dimension \(n\). The extreme directions of the rearrangement cone for \(1 \leq p \leq n - 1\) are given by [Jeong2017] Theorem 5.2.3. When \(2 \leq p \leq n - 2\) (that is, if we ignore \(p = 1\) and \(p = n - 1\)), they consist of

- the standard basis \(\{e_1, e_2, \ldots, e_n\}\) for the ambient space, and
- the \(n\) vectors \((1, 1, \ldots, 1)^T - pe_i\) for \(i = 1, 2, \ldots, n\).

Special cases are then given for \(p = 1\) and \(p = n - 1\) in the theorem. However in SageMath we don't need conically-independent extreme directions. We only need a generating set, because the \(\text{Cone()}\) function will eliminate any redundant generators. And one can easily verify that the special-case extreme directions for \(p = 1\) and \(p = n - 1\) are contained in the conic hull of the \(2n\) generators just described. The half space resulting from \(p = n\) is also covered by this set of generators, so for all valid \(p\) we simply take the conic hull of those \(2n\) vectors.

REFERENCES:

- [GJ2016], Section 4
- [HS2010], Example 2.21
- [Jeong2017], Section 5.2

EXAMPLES:

The rearrangement cones of order one are nonnegative orthants:

```sage
orthant = cones.nonnegative_orthant(6)
sage: cones.rearrangement(1,6).is_equivalent(orthant)
True
```

When \(p\) and \(\text{ambient\_dim}\) are equal, the rearrangement cone is a half-space, so we expect its lineality to be one less than \(\text{ambient\_dim}\) because it will contain a hyperplane but is not the entire space:

```sage
sage: cones.rearrangement(5,5).lineality()
4
```

Jeong’s Proposition 5.2.1 [Jeong2017] states that all rearrangement cones are proper when \(p\) is less than \(\text{ambient\_dim}\):

```sage
sage: all( cones.rearrangement(p, ambient_dim).is_proper() 
.....:     for ambient_dim in range(10) 
.....:         for p in range(1, ambient_dim) )
True
```

Jeong’s Corollary 5.2.4 [Jeong2017] states that if \(p = n - 1\) in an \(n\)-dimensional ambient space, then the Lyapunov rank of the rearrangement cone is \(n\), and that for all other \(p > 1\) its Lyapunov rank is one:

```sage
sage: all( cones.rearrangement(p, ambient_dim).lyapunov_rank() == ambient_dim
.....:     for ambient_dim in range(2, 10) 
.....:         for p in range(2, ambient_dim - 1) )
True
```

```sage
sage: all( cones.rearrangement(p, ambient_dim).lyapunov_rank() == 1 
.....:     for ambient_dim in range(3, 10) 
.....:         for p in range(2, ambient_dim - 1) )
True
```
The Schur cone in \( n \) dimensions induces the majorization ordering on the ambient space. If \( \{e_1, e_2, \ldots, e_n\} \) is the standard basis for the space, then its generators are \( \{e_i - e_{i+1} \mid 1 \leq i \leq n - 1\} \). Its dual is the downward monotonic cone.

**INPUT:**
- \texttt{ambient_dim} – a nonnegative integer (default: None); the dimension of the ambient space
- \texttt{lattice} – a toric lattice (default: None); the lattice in which the cone will live

If \texttt{ambient_dim} is omitted, then it will be inferred from the rank of \texttt{lattice}. If the \texttt{lattice} is omitted, then the default lattice of rank \texttt{ambient_dim} will be used.

A \texttt{ValueError} is raised if neither \texttt{ambient_dim} nor \texttt{lattice} are specified. It is also a \texttt{ValueError} to specify both \texttt{ambient_dim} and \texttt{lattice} unless the rank of \texttt{lattice} is equal to \texttt{ambient_dim}.

**OUTPUT:**
A \texttt{ConvexRationalPolyhedralCone} representing the Schur cone living in \texttt{lattice}, with ambient dimension \texttt{ambient_dim}. Each generating ray has the integer ring as its base ring.

A \texttt{ValueError} can be raised if the inputs are incompatible or insufficient. See the INPUT documentation for details.

**REFERENCES:**
- [GS2010], Section 3.1
- [IS2005], Example 7.3
- [SS2016], Example 7.4

**EXAMPLES:**
Verify the claim [SS2016] that the maximal angle between any two generators of the Schur cone and the non-negative orthant in dimension five is \((3/4) \pi\):

```python
sage: P = cones.schur(5)
sage: Q = cones.nonnegative_orthant(5)
sage: G = ( g.change_ring(QQbar).normalized() for g in P )
sage: H = ( h.change_ring(QQbar).normalized() for h in Q )
sage: actual = max(arccos(u.inner_product(v)) for u in G for v in H)
sage: expected = 3*pi/4
sage: abs(actual - expected).n() < 1e-12
True
```

The dual of the Schur cone is the “downward monotonic cone” [GS2010], whose elements’ entries are in non-increasing order:

```python
sage: set_random_seed()
sage: ambient_dim = ZZ.random_element(10)
sage: K = cones.schur(ambient_dim).dual()
sage: x = K.random_element()
sage: all( x[i] >= x[i+1] for i in range(ambient_dim-1) )
True
```

The trivial cone with no nonzero generators in \texttt{ambient_dim} dimensions, or living in \texttt{lattice}.

```python
sage: trivial = cones.trivial(ambient_dim=None, lattice=None)
```

The trivial cone with no nonzero generators in \texttt{ambient_dim} dimensions, or living in \texttt{lattice}.

```python
sage: trivial = cones.trivial(ambient_dim=None, lattice=None)
```
INPUT:

- ambient_dim – a nonnegative integer (default: None); the dimension of the ambient space
- lattice – a toric lattice (default: None); the lattice in which the cone will live

If ambient_dim is omitted, then it will be inferred from the rank of lattice. If the lattice is omitted, then the default lattice of rank ambient_dim will be used.

A ValueError is raised if neither ambient_dim nor lattice are specified. It is also a ValueError to specify both ambient_dim and lattice unless the rank of lattice is equal to ambient_dim.

OUTPUT:

A ConvexRationalPolyhedralCone representing the trivial cone with no nonzero generators living in lattice, with ambient dimension ambient_dim.

A ValueError can be raised if the inputs are incompatible or insufficient. See the INPUT documentation for details.

EXAMPLES:

Construct the trivial cone, containing only the origin, in three dimensions:

```sage
cones.trivial(3)
```

0-d cone in 3-d lattice N

If a lattice is given, the trivial cone will live in that lattice:

```sage
L = ToricLattice(3, 'M')
sage: cones.trivial(3, lattice=L)
```

0-d cone in 3-d lattice M

### 2.4.4 Rational polyhedral fans

This module was designed as a part of the framework for toric varieties (``variety, fano_variety``). While the emphasis is on complete full-dimensional fans, arbitrary fans are supported. Work with distinct lattices. The default lattice is ToricLattice N of the appropriate dimension. The only case when you must specify lattice explicitly is creation of a 0-dimensional fan, where dimension of the ambient space cannot be guessed.

A rational polyhedral fan is a finite collection of strictly convex rational polyhedral cones, such that the intersection of any two cones of the fan is a face of each of them and each face of each cone is also a cone of the fan.

AUTHORS:

- Andrey Novoseltsev (2010-06-17): substantial improvement during review by Volker Braun.

EXAMPLES:

Use Fan() to construct fans “explicitly”:

```sage
fan = Fan(cones=[[0,1], [1,2]],
....: rays=[[1,0], [0,1], [-1,0]])
sage: fan
```

Rational polyhedral fan in 2-d lattice N

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:

```python
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()`:

```python
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:

```python
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)
```

The last output is not very illuminating. Let’s try to improve it:

```python
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),
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),
```

(continues on next page)
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, 2)
(0, 2, 4)
(0, 4, 5)
```

to see indices of rays of the fan corresponding to each cone.

While the above cycles were over “cones in fan”, it is obvious that we did not get ALL the cones: every face of every cone in a fan must also be in the fan, but all of the above cones were of dimension three. The reason for this behaviour is that in many cases it is enough to work with generating cones of the fan, i.e. cones which are not faces of bigger cones. When you do need to work with lower dimensional cones, you can easily get access to them using `cones()`:

```
sage: [cone.ambient_ray_indices() for cone in fan1.cones(2)]
[(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (0, 4),
 (2, 4), (3, 4), (1, 5), (3, 5), (4, 5), (0, 5)]
```

In fact, you do not have to type `.cones`:

```
sage: [cone.ambient_ray_indices() for cone in fan1(2)]
[(0, 1), (0, 2), (1, 2), (1, 3), (2, 3), (0, 4),
 (2, 4), (3, 4), (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()`:

```
sage: L = fan1.cone_lattice()
sage: L
Finite lattice containing 28 elements with distinguished linear extension
sage: L.bottom()
0-d cone of Rational polyhedral fan in 3-d lattice M
sage: L.top()
(continues on next page)
```
Rational polyhedral fan in 3-d lattice \( \mathbb{M} \)

```
sage: cone = L.level_sets()[2][0]
sage: cone
2-d cone of Rational polyhedral fan in 3-d lattice \( \mathbb{M} \)
sage: sorted(L.hasse_diagram().neighbors(cone))
[1-d cone of Rational polyhedral fan in 3-d lattice \( \mathbb{M} \),
  1-d cone of Rational polyhedral fan in 3-d lattice \( \mathbb{M} \),
  3-d cone of Rational polyhedral fan in 3-d lattice \( \mathbb{M} \),
  3-d cone of Rational polyhedral fan in 3-d lattice \( \mathbb{M} \)]
```

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 \texttt{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 \texttt{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 \texttt{fan2} and \texttt{fan4}:

```
sage: fan2.ngenerating_cones()
6
sage: fan2.nrays()
336
```

(continues on next page)
Smoothness does not come for free!

Please take a look at the rest of the available functions below and their complete descriptions. If you need any features that are missing, feel free to suggest them. (Or implement them on your own and submit a patch to Sage for inclusion!)

**class** `sage.geometry.fan.Cone_of_fan(ambient, ambient_ray_indices)`

Bases: `sage.geometry.cone.ConvexRationalPolyhedralCone`

Construct a cone belonging to a fan.

**Warning:** This class does not check that the input defines a valid cone of a fan. You must not construct objects of this class directly.

In addition to all of the properties of “regular” cones, such cones know their relation to the fan.

**INPUT:**
- `ambient` – fan whose cone is constructed;
- `ambient_ray_indices` – increasing list or tuple of integers, indices of rays of `ambient` generating this cone.

**OUTPUT:**
- cone of `ambient`.

**EXAMPLES:**

The intended way to get objects of this class is the following:

```
sage: 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 `self`.

**OUTPUT:**
- increasing tuple of integers.

**EXAMPLES:**

```
sage: P1xP1 = toric_varieties.P1xP1()
sage: cone = P1xP1.fan().generating_cone(0)
```

(continues on next page)
sage: cone.star_generator_indices()
(0,)

star_generator_indices()
Return indices of generating cones of the “ambient fan” containing self.

OUTPUT:
• increasing tuple of integers.

EXAMPLES:
```python
sage: P1xP1 = toric_varieties.P1xP1()
sage: cone = P1xP1.fan().generating_cone(0)
sage: cone.star_generator_indices()
(2-d cone of Rational polyhedral fan in 2-d lattice N,)
```

sage.geometry.fan.FaceFan(polytope, lattice=None)
Construct the face fan of the given rational polytope.

INPUT:
• polytope – a polytope over \( \mathbb{Q} \) 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:
```python
sage: diamond = lattice_polytope.cross_polytope(2)
sage: P1xP1 = FaceFan(diamond)
sage: P1xP1.rays()
M( 1, 0),
M( 0, 1),
M(-1, 0),
M( 0, -1)
in 2-d lattice M
sage: for cone in P1xP1: print(cone.rays())
M(-1, 0),
M( 0, -1)
in 2-d lattice M
M( 0, 1),
M(-1, 0)
in 2-d lattice M
M(1, 0),
M(0, 1)
in 2-d lattice M
M(1, 0),
```

M(0, -1) in 2-d lattice M

sage.geometry.fan.Fan(cones=None, rays=None, lattice=None, check=True, normalize=True, is_complete=None, virtual_rays=None, discard_faces=False, allow_arrangement=False)

Construct a rational polyhedral fan.

**Note:** Approximate time to construct a fan consisting of \( n \) cones is \( n^2/5 \) seconds. That is half an hour for 100 cones. This time can be significantly reduced in the future, but it is still likely to be \( \sim n^2 \) (with, say, /500 instead of /5). If you know that your input does form a valid fan, use check=False option to skip consistency checks.

**INPUT:**

- **cones** – list of either Cone objects or lists of integers interpreted as indices of generating rays in rays. These must be only maximal cones of the fan, unless discard_faces=True or allow_arrangement=True option is specified;

- **rays** – list of rays given as list or vectors convertible to the rational extension of lattice. If cones are given by Cone objects rays may be determined automatically. You still may give them explicitly to ensure a particular order of rays in the fan. In this case you must list all rays that appear in cones. You can give “extra” ones if it is convenient (e.g. if you have a big list of rays for several fans), but all “extra” rays will be discarded;

- **lattice** – ToricLattice, \( \mathbb{Z}^n \), 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), unless allow_arrangement=True option is specified. If you know for sure that the input is correct, you may significantly decrease construction time using check=False option;

- **normalize** – you can further speed up construction using normalize=False option. In this case cones must be a list of sorted tuples and rays must be immutable primitive vectors in lattice. In general, you should not use this option, it is designed for code optimization and does not give as drastic improvement in speed as the previous one;

- **is_complete** – every fan can determine on its own if it is complete or not, however it can take quite a bit of time for “big” fans with many generating cones. On the other hand, in some situations it is known in advance that a certain fan is complete. In this case you can pass is_complete=True option to speed up some computations. You may also pass is_complete=False option, although it is less likely to be beneficial. Of course, passing a wrong value can compromise the integrity of data structures of the fan and lead to wrong results, so you should be very careful if you decide to use this option;

- **virtual_rays** – (optional, computed automatically if needed) a list of ray generators to be used for virtual_rays();

- **discard_faces** – by default, the fan constructor expects the list of maximal cones, unless allow_arrangement=True option is specified. If you provide “extra” ones and leave allow_arrangement=False (default) and check=True (default), an exception will be raised. If you provide “extra” cones and set allow_arrangement=False (default) and check=False, you may get wrong results as assumptions on internal data structures will be invalid. If you want the fan constructor to select the maximal cones from the given input, you may provide discard_faces=True option (it works both for check=True and check=False);

- **allow_arrangement** – by default (allow_arrangement=False), the fan constructor expects that the intersection of any two given cones is a face of each. If allow_arrangement=True option is specified, then construct a rational polyhedral fan from the cone arrangement, so that the union of the cones in the
polyhedral fan equals to the union of the given cones, and each given cone is the union of some cones in the polyhedral fan.

OUTPUT:

• a fan.

See also:

In 2 dimensions you can cyclically order the rays. Hence the rays determine a unique maximal fan without having to specify the cones, and you can use `Fan2d()` to construct this fan from just the rays.

EXAMPLES:

Let’s construct a fan corresponding to the projective plane in several ways:

```sage
cone1 = Cone([(1,0), (0,1)])
cone2 = Cone([(0,1), (-1,-1)])
cone3 = Cone([(-1,-1), (1,0)])
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)
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])
P2.ngenerating_cones()
```

3

Looks better. An alternative way is

```sage
rays = [(1,0), (0,1), (-1,-1)]
cones = [(0,1), (1,2), (2,0)]
P2a = Fan(cones, rays)
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
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:

```python
sage: P2c = Fan(cones, rays, check=False, normalize=False)
Traceback (most recent call last):
  ...  
AttributeError: 'tuple' object has no attribute 'parent'
```

Yet another way is to use functions `FaceFan()` and `NormalFan()` to construct fans from lattice polytopes.

We have not yet used `lattice` argument, since it was determined automatically:

```python
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:

```python
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
```

In the following examples, we test the `allow_arrangement=True` option. See trac ticket #25122.

The intersection of the two cones is not a face of each. Therefore, they do not belong to the same rational polyhedral fan:

```python
sage: c1 = Cone([(-2,-1,1), (-2,1,1), (2,1,1), (2,-1,1)])
sage: c2 = Cone([(-1,-2,1), (-1,2,1), (1,2,1), (1,-2,1)])
sage: c1.intersection(c2).is_face_of(c1)
False
sage: c1.intersection(c2).is_face_of(c2)
False
sage: Fan([c1, c2])
Traceback (most recent call last):
  ...
ValueError: these cones cannot belong to the same fan!
```
Let’s construct the fan using `allow_arrangement=True` option:

```python
sage: fan = Fan([c1, c2], allow_arrangement=True)
sage: fan.ngenerating_cones()
5
```

Another example where cone `c2` is inside cone `c1`:

```python
sage: c1 = Cone([(4, 0, 0), (0, 4, 0), (0, 0, 4)])
sage: c2 = Cone([(2, 1, 1), (1, 2, 1), (1, 1, 2)])
sage: fan = Fan([c1, c2], allow_arrangement=True)
sage: fan.ngenerating_cones()
7
sage: fan.plot()
Graphics3d Object
```

Cones of different dimension:

```python
sage: c1 = Cone([(1, 0), (0, 1)])
sage: c2 = Cone([(2, 1)])
sage: c3 = Cone([-(-1, -2)])
sage: fan = Fan([c1, c2, c3], allow_arrangement=True)
sage: for cone in sorted(fan.generating_cones()):
    print(sorted(cone.rays()))
[N(-1, -2)]
[N(0, 1), N(1, 2)]
[N(1, 0), N(2, 1)]
[N(1, 2), N(2, 1)]
```

A 3-d cone and a 1-d cone:

```python
sage: c3 = Cone([(0, 1, 1), [0, -1, 1], [1, 0, 1], [-1, 0, 1]])
sage: c1 = Cone([(0, 0, 1)])
sage: fan1 = Fan([c1, c3], allow_arrangement=True)
sage: fan1.plot()
Graphics3d Object
```

A 3-d cone and two 2-d cones:

```python
sage: c2v = Cone([(0, 1, 1), [0, -1, 1]])
sage: c2h = Cone([[1, 0, 1], [-1, 0, 1]])
sage: fan2 = Fan([c2v, c2h, c3], allow_arrangement=True)
sage: fan2.is_simplicial()
True
sage: fan2.is_equivalent(fan1)
True
```

```python
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*, \( \mathbb{Z}^n \), or any other object that behaves like these. If not specified, an attempt will be made to determine an appropriate toric lattice automatically.

**EXAMPLES:**

```python
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:

```python
sage: fan1 = Fan2d([(0,1), (1,0)])
sage: fan1.rays()
N(0, 1),
N(1, 0)
in 2-d lattice N

sage: fan2 = Fan2d([(1,0), (0,1)])
sage: fan2.rays()
N(1, 0),
N(0, 1)
in 2-d lattice N

sage: fan1 == fan2, fan1.is_equivalent(fan2)
(False, True)

sage: fan = Fan2d([(1,1), (-1,-1), (1,-1), (-1,1)])
sage: [ cone.ambient_ray_indices() for cone in fan ]
[(2, 1), (1, 3), (3, 0), (0, 2)]
sage: fan.is_complete()
True
```

`sage.geometry.fan.NormalFan(polytope, lattice=None)`

Construct the normal fan of the given rational polytope.

This returns the inner normal fan. For the outer normal fan, use `NormalFan(-P)`.

**INPUT:**

- `polytope` – a full-dimensional polytope over \( \mathbb{Q} \) or class: `latticepolytope < 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()
```

(continues on next page)
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( 1, 0),
N( 0, 1),
in 2-d lattice N
N( 0, -1),
N(-1, 0)
in 2-d lattice N

sage: cuboctahed = polytopes.cuboctahedron()
sage: NormalFan(cuboctahed)
Rational polyhedral fan in 3-d lattice N

class sage.geometry.fan.RationalPolyhedralFan(cones, rays, lattice, is_complete=None, virtual_rays=None)

Bases: sage.geometry.cone.IntegralRayCollection, collections.abc.Callable, collections.abc.Container

Create a rational polyhedral fan.

**Warning:** This class does not perform any checks of correctness of input nor does it convert input into the standard representation. Use `Fan()` to construct fans from “raw data” or `FaceFan()` and `NormalFan()` to get fans associated to polytopes.

Fans are immutable, but they cache most of the returned values.

**INPUT:**

- `cones` – list of generating cones of the fan, each cone given as a list of indices of its generating rays in `rays`;
- `rays` – list of immutable primitive vectors in `lattice` consisting of exactly the rays of the fan (i.e. no “extra” ones);
- `lattice` – `ToricLattice, Z^n`, or any other object that behaves like these. If `None`, it will be determined as `parent()` of the first ray. Of course, this cannot be done if there are no rays, so in this case you must give an appropriate `lattice` directly;
- `is_complete` – if given, must be `True` or `False` depending on whether this fan is complete or not. By default, it will be determined automatically if necessary;
- `virtual_rays` – if given, must be a list of immutable primitive vectors in `lattice`, see `virtual_rays()` for details. By default, it will be determined automatically if necessary.

**OUTPUT:**
• rational polyhedral fan.

**Gale_transform()**
Return the Gale transform of self.

OUTPUT:
A matrix over \(\mathbb{Z}\).

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:

```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.Stanley_Reisner_ideal( PolynomialRing(QQ,5, 'A, B, C, D, E') )
```

**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,)], 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:

```
sage: F0 = Fan2d([(1,0),(0,1),(-1,0),(0,-1)])
sage: F0.common_refinement(F0) == F0
True
```

A more complex example with complete fans:

```
sage: F1 = Fan([(0),[1]],[(1),(-1,)]]
sage: F2 = Fan2d([(1,0),(1,1),(0,1),(-1,0),(0,-1)])
sage: F3 = F2.cartesian_product(F1)
sage: F4 = F1.cartesian_product(F2)
sage: FF = F3.common_refinement(F4)
sage: FF.ngenerating_cones()
10
sage: F3.ngenerating_cones()
10
sage: F4.ngenerating_cones()
13
```

An example with two non-complete fans with the same support:

```
sage: F5 = Fan2d([(1,0),(1,2),(0,1)])
sage: F6 = Fan2d([(1,0),(2,1),(0,1)])
sage: F5.common_refinement(F6).ngenerating_cones()
3
```

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 [Kly1990], eq. (3.2). This complex computes the homology of $|\Sigma| \subset N_R$ with arbitrary support,

$$
H_i(K) = H_{d-i}(|\Sigma|, \mathbb{Z})_{\text{non-cpt}}
$$
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

$$0 \rightarrow \mathbb{Z} \rightarrow \mathbb{Z}^{\Sigma(d)} \rightarrow \mathbb{Z}^{\Sigma(d-1)} \rightarrow \cdots \rightarrow \mathbb{Z}^{\Sigma(0)} \rightarrow 0$$

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:**

```python
sage: fan = toric_varieties.P(3).fan()
sage: K_normal = fan.complex(); K_normal
Chain complex with at most 4 nonzero terms over Integer Ring
sage: K_normal.homology()
{0: \text{Z}, 1: 0, 2: 0, 3: 0}
sage: K_extended = fan.complex(extended=True); K_extended
Chain complex with at most 5 nonzero terms over Integer Ring
sage: K_extended.homology()
{-1: \text{0}, 0: \text{0}, 1: \text{0}, 2: \text{0}, 3: \text{0}}
```

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: \text{Vector space of dimension 0 over Rational Field},
  0: \text{Vector space of dimension 0 over Rational Field},
  1: \text{Vector space of dimension 0 over Rational Field},
  2: \text{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.is_complete()
False
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:
cone_containing(*points)

Return the smallest cone of self containing all given points.

INPUT:

- either one or more indices of rays of self, or one or more objects representing points of the ambient space of self, or a list of such objects (you CANNOT give a list of indices).

OUTPUT:

- A cone of fan whose ambient fan is self.

Note: We think of the origin as of the smallest cone containing no rays at all. If there is no ray in self that contains all rays, a ValueError exception will be raised.

EXAMPLES:

```python
cone1 = Cone([[0, -1], [1, 0]])
cone2 = Cone([[1, 0], [0, 1]])
f = Fan([cone1, cone2])
f.rays()
N(0, -1),
N(0, 1),
N(1, 0)
in 2-d lattice N
cone = f.cone_containing(0) # ray index
1-d cone of Rational polyhedral fan in 2-d lattice N
cone = 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]
cone = f.cone_containing(0, 2) # ray indices
```
cone_lattice()
Return the cone lattice of self.
This lattice will have the origin as the bottom (we do not include the empty set as a cone) and the fan itself as the top.

OUTPUT:
* finite poset <sage.combinat.posets.posets.FinitePoset of cones of fan, behaving like “regular” cones, but also containing the information about their relation to this fan, namely, the contained rays and containing generating cones. The top of the lattice will be this fan itself (which is not a cone of fan).

See also cones().

EXAMPLES:
Cone lattices can be computed for arbitrary fans:

```python
sage: cone1 = Cone([(1,0), (0,1)])
sage: cone2 = Cone([(-1,0)])
sage: fan = Fan([cone1, cone2])
sage: fan.rays()
N(-1, 0),
N( 0, 1),
N( 1, 0)
in 2-d lattice N
sage: for cone in fan: print(cone.ambient_ray_indices())
(1, 2)
(0,)
sage: L = fan.cone_lattice()
sage: L
Finite poset containing 6 elements with distinguished linear extension
```

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:
For example, you can do

```python
sage: for l in L.level_sets()[:-1]:
    ....: print([f.ambient_ray_indices() for f in l])
[()]
[(0, ), (1, ), (2, )]
[(1, 2)]
```

If the fan is complete, its cone lattice is atomic and coatomic and can (and will!) be computed in a much more efficient way, but the interface is exactly the same:

```python
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)]
```

Let’s also consider the cone lattice of a fan generated by a single cone:

```python
sage: fan = Fan([cone1])
sage: L = fan.cone_lattice()
sage: L
Finite poset containing 5 elements with distinguished linear extension
```

Here these 5 elements correspond to the origin, two rays, one generating cone of dimension two, and the whole fan. While this single cone “is” the whole fan, it is consistent and convenient to distinguish them in the cone lattice.

**cones**(dim=None, codim=None)

Return the specified cones of self.

**INPUT:**

- dim – dimension of the requested cones;
- codim – codimension of the requested cones.

**Note:** You can specify at most one input parameter.

**OUTPUT:**

- tuple of cones of self of the specified (co)dimension, if either dim or codim is given. Otherwise tuple of such tuples for all existing dimensions.

**EXAMPLES:**
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)
()  # Empty cone
sage: fan(3)
()  # Empty cone
sage: fan(codim=4)
()  # Empty cone
```

`contains(cone)`
Check if a given cone is equivalent to a cone of the fan.

**INPUT:**
- `cone` – anything.

**OUTPUT:**
- `False` if `cone` is not a cone or if `cone` is not equivalent to a cone of the fan. `True` otherwise.

**Note:** Recall that a fan is a (finite) collection of cones. A cone is contained in a fan if it is equivalent to one of the cones of the fan. In particular, it is possible that all rays of the cone are in the fan, but the cone itself is not.

If you want to know whether a point is in the support of the fan, you should use `support_contains()`.

**EXAMPLES:**

We first construct a simple fan:

```
sage: cone1 = Cone([(0,-1), (1,0)])
sage: cone2 = Cone([(1,0), (0,1)])
sage: f = Fan([cone1, cone2])
```
Now we check if some cones are in this fan. First, we make sure that the order of rays of the input cone does not matter (check=False option ensures that rays of these cones will be listed exactly as they are given):

```
sage: f.contains(Cone([(1,0), (0,1)], check=False))
True
sage: f.contains(Cone([(0,1), (1,0)], check=False))
True
```

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
```

```
sage: f.contains(Cone([(1,0), (-0,1)]))
True
```

A point is not a cone:

```
sage: n = f.lattice()(1,1); n
N(1, 1)
sage: f.contains(n)
False
```

**embed(cone)**

Return the cone equivalent to the given one, but sitting in `self`.

You may need to use this method before calling methods of `cone` that depend on the ambient structure, such as `ambient_ray_indices()` or `facet_of()`. The cone returned by this method will have `self` as ambient. If `cone` does not represent a valid cone of `self`, `ValueError` exception is raised.

**Note:** This method is very quick if `self` is already the ambient structure of `cone`, so you can use without extra checks and performance hit even if `cone` is likely to sit in `self` but in principle may not.

**INPUT:**

- `cone` – a `cone`.

**OUTPUT:**

- a `cone of fan`, equivalent to `cone` but sitting inside `self`.

**EXAMPLES:**

Let’s take a 3-d fan generated by a cone on 4 rays:

```
sage: f = Fan([Cone([(1,0,1), (0,1,1), (-1,0,1), (0,-1,1)])])
```

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:
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 \(self\).

INPUT:

- \(n\) – integer, the index of a generating cone.

OUTPUT:

- \(cone of fan\).

EXAMPLES:

```
```
\begin{verbatim}
sage: fan = toric_varieties.P1xP1().fan()
sage: fan.generating_cone(0)
2-d cone of Rational polyhedral fan in 2-d lattice N
\end{verbatim}

**generating_cones()**

Return generating cones of self.

**OUTPUT:**

- tuple of cones of fan.

**EXAMPLES:**

\begin{verbatim}
sage: fan = toric_varieties.P1xP1().fan()
sage: fan.is_complete()
True
sage: talpa = Cone([(0, 0), (1, 0), (0, 1), (1, 1)]).
\end{verbatim}

\begin{verbatim}
sage: fan = Fan([cone1, cone2])
sage: fan.is_equivalent(other)
False
\end{verbatim}

**is_complete()**

Check if self is complete.

A rational polyhedral fan is complete if its cones fill the whole space.

**OUTPUT:**

- True if self is complete and False otherwise.

**EXAMPLES:**

\begin{verbatim}
sage: fan = toric_varieties.P1xP1().fan()
sage: fan.is_complete()
True
sage: cone1 = Cone([(1, 0), (0, 1)])
sage: cone2 = Cone([(0, 1)])
sage: fan = Fan([cone1, cone2])
sage: fan.is_complete()
False
\end{verbatim}

**is_equivalent(other)**

Check if self is “mathematically” the same as other.

**INPUT:**

- other - fan.

**OUTPUT:**

- True if self and other define the same fans as collections of equivalent cones in the same lattice, False otherwise.

There are three different equivalences between fans \( F_1 \) and \( F_2 \) in the same lattice:
1. They have the same rays in the same order and the same generating cones in the same order. This is tested by \( F_1 == F_2 \).

2. They have the same rays and the same generating cones without taking into account any order. This is tested by \( F_1.is\_equivalent(F_2) \).

3. They are in the same orbit of \( GL(n, \mathbb{Z}) \) (and, therefore, correspond to isomorphic toric varieties). This is tested by \( F_1.is\_isomorphic(F_2) \).

Note that \( virtual\_rays() \) are included into consideration for all of the above equivalences.

**EXAMPLES:**

```python
sage: fan1 = Fan(cones=[(0,1), (1,2)],
            rays=[[1,0], (0,1), (-1,-1)],
            check=False)
sage: fan2 = Fan(cones=[(2,1), (0,2)],
            rays=[[1,0], (-1,-1), (0,1)],
            check=False)
sage: fan3 = Fan(cones=[(0,1), (1,2)],
            rays=[[1,0], (0,1), (-1,1)],
            check=False)
sage: fan1 == fan2
False
sage: fan1.is_equivalent(fan2)
True
sage: fan1 == fan3
False
sage: fan1.is_equivalent(fan3)
False
```

**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 \( F_1 == F_2 \).

2. They have the same rays and the same generating cones without taking into account any order. This is tested by \( F_1.is\_equivalent(F_2) \).

3. They are in the same orbit of \( GL(n, \mathbb{Z}) \) (and, therefore, correspond to isomorphic toric varieties). This is tested by \( F_1.is\_isomorphic(F_2) \).

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 \( isomorphism() \).

**EXAMPLES:**

Here we pick an \( SL(2, \mathbb{Z}) \) matrix \( m \) and then verify that the image fan is isomorphic:
These fans are “mirrors” of each other:

```python
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)
```

```python
sage: fan1 == fan2
False
sage: fan1.is_equivalent(fan2)
False
sage: fan1.is_isomorphic(fan2)
True
sage: fan1.is_isomorphic(fan1)
True
```

**is_simplicial()**

Check if self is simplicial.

A rational polyhedral fan is simplicial if all of its cones are, i.e. primitive vectors along generating rays of every cone form a part of a rational basis of the ambient space.

OUTPUT:

- True if self is simplicial and False otherwise.

EXAMPLES:

```python
sage: fan = toric_varieties.P1xP1().fan()
sage: fan.is_simplicial()
True
sage: cone1 = Cone([(1,0), (0,1)])
sage: cone2 = Cone([-1,0])
sage: fan = Fan([cone1, cone2])
sage: fan.is_simplicial()
True
```

In fact, any fan in a two-dimensional ambient space is simplicial. This is no longer the case in dimension three:

```python
sage: fan = NormalFan(lattice_polytope.cross_polytope(3))
sage: fan.is_simplicial()
```

(continues on next page)
False

```python
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(ring)
Return the ideal generated by linear relations.

INPUT:

- A polynomial ring in self.nrays() variables.

OUTPUT:
Return the ideal, in the given ring, generated by the linear relations of the rays. In toric geometry, this corresponds to rational equivalence of divisors.

EXAMPLES:

sage: fan = Fan([[0,1,3],[3,4],[2,0],[1,2,4]], [(-3, -2, 1), (0, 0, 1), (3, -2, 1), (-1, -1, 1)])
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...

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:
```python
sage: fan = NormalFan(lattice_polytope.cross_polytope(3))
sage: fan.is_simplicial()
False
sage: fan.ngenerating_cones()
6
sage: new_fan = fan.make_simplicial()
sage: new_fan.is_simplicial()
True
sage: new_fan.ngenerating_cones()
12
```

**ngenerating_cones()**

Return the number of generating cones of self.

**OUTPUT:**

- integer.

**EXAMPLES:**

```python
sage: fan = toric_varieties.P1xP1().fan()
sage: fan.ngenerating_cones()
4
sage: cone1 = Cone([(1,0), (0,1)])
sage: cone2 = Cone([-1,0])
sage: fan = Fan([cone1, cone2])
sage: fan.ngenerating_cones()
2
```

**oriented_boundary**(cone)

Return the facets bounding cone with their induced orientation.

**INPUT:**

- cone – a cone of the fan or the whole fan.

**OUTPUT:**

The boundary cones of cone as a formal linear combination of cones with coefficients ±1. Each summand is a facet of cone and the coefficient indicates whether their (chosen) orientation agrees or disagrees with the “outward normal first” boundary orientation. Note that the orientation of any individual cone is arbitrary. This method once and for all picks orientations for all cones and then computes the boundaries relative to that chosen orientation.

If cone is the fan itself, the generating cones with their orientation relative to the ambient space are returned.

See complex() for the associated chain complex. If you do not require the orientation, use cone.facets() instead.

**EXAMPLES:**

```python
sage: fan = toric_varieties.P(3).fan()
sage: cone = fan(2)[0]
sage: bdry = fan.oriented_boundary(cone); bdry
-1-d cone of Rational polyhedral fan in 3-d lattice N + 1-d cone of Rational polyhedral fan in 3-d lattice N
sage: bdry[0]
(-1, 1-d cone of Rational polyhedral fan in 3-d lattice N)
```

(continues on next page)

(continued from previous page)

```sage
bdry[1]
(1, 1-d cone of Rational polyhedral fan in 3-d lattice N)
sage: fan.oriented_boundary(bdry[0][1])
-0-d cone of Rational polyhedral fan in 3-d lattice N
sage: fan.oriented_boundary(bdry[1][1])
-0-d cone of Rational polyhedral fan in 3-d lattice N
```

If you pass the fan itself, this method returns the orientation of the generating cones which is determined by the order of the rays in `cone.ray_basis()`

```sage
fan.oriented_boundary(fan)
-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
```

A non-full dimensional fan:

```sage
cone = Cone([[4, 5]])
sage: fan = Fan([cone])
sage: fan.oriented_boundary(cone)
0-d cone of Rational polyhedral fan in 2-d lattice N
sage: fan.oriented_boundary(fan)
1-d cone of Rational polyhedral fan in 2-d lattice N
```

plot(**options)

Plot self.

INPUT:

- any options for toric plots (see `toric_plotter.options`), none are mandatory.

OUTPUT:

- a plot.

EXAMPLES:

```sage
fan = toric_varieties.dP6().fan()
sage: fan.plot()
Graphics object consisting of 31 graphics primitives
```

primitive_collections()

Return the primitive collections.

OUTPUT:

Return the subsets \(\{i_1, \ldots, i_k\} \subseteq \{1, \ldots, n\}\) 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:

• [Bat1991]

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.

2.4. Toric geometry
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 \( \mathbb{N}_R \). 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 $N_Q$ “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(1, 0, 0, 0),
N(0, 0, 0, 1)
in 4-d lattice N
sage: f.rays()
N(1, 0, 1, 0),
N(0, 1, 1, 0)
in 4-d lattice N
sage: f.virtual_rays([0])
N(1, 0, 0, 0)
in 4-d lattice N
```

You can also give virtual ray indices directly, without packing them into a list:

```python
sage: f.virtual_rays(0)
N(1, 0, 0, 0)
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())
0
```

```
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`.
```
• 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:

```python
sage: from sage.geometry.fan import discard_faces
sage: len(discard_faces(cones))
3
sage: Sigma.ngenerating_cones()
3
```

```python
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:

```python
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
```

### 2.4.5 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:
Some of the cones of the normal fan fail to be mapped to a single cone of the face fan. We can rectify the situation in the following way:

```
sage: fm = FanMorphism(phi, normal, face, subdivide=True)
sage: fm
Fan morphism defined by the matrix
[1 0]
[0 0]
Domain fan: Rational polyhedral fan in 2-d lattice N
Codomain fan: Rational polyhedral fan in 2-d lattice N
sage: fm.domain_fan().rays()
N( 1, 1),
N( 1, -1),
N(-1, -1),
N(-1, 1),
N( 0, -1),
N( 0, 1)
in 2-d lattice N
sage: normal.rays()
N( 1, 1),
N( 1, -1),
N(-1, -1),
N(-1, 1)
in 2-d lattice N
```

As you see, it was necessary to insert two new rays (to prevent “upper” and “lower” cones of the normal fan from being mapped to the whole x-axis).

**class** `sage.geometry.fan_morphism.FanMorphism(morphism, domain_fan, codomain=None, subdivide=False, check=True, verbose=False)`

Bases: `sage.modules.free_module_morphism.FreeModuleMorphism`

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. \( \phi(\sigma_1) \subseteq \sigma_2 \) (\( \sigma_2 \) 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;
- `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`.

---

**Chapter 2. Polyhedral computations**
We had to subdivide two of the four cones of the normal fan, since they were mapped by $\phi$ into non-strictly convex cones.

It is possible to omit the codomain fan, in which case the image fan will be used instead of it:

```
sage: fm = FanMorphism(phi, normal, face)
sage: fm.domain_fan() is normal
False
sage: fm.domain_fan().ngenerating_cones()
6
```

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
```

codomain_fan\((\text{dim=}\text{None}, \text{codim=}\text{None})\)

Return the codomain fan of \textit{self}.

INPUT:

- \textit{dim} – dimension of the requested cones;
- \textit{codim} – codimension of the requested cones.

OUTPUT:

- \textit{rational polyhedral fan} if no parameters were given, \textit{tuple of cones} otherwise.

EXAMPLES:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: quadrant = Fan([quadrant])
sage: quadrant_bl = quadrant.subdivide([(1,1)])
sage: fm = FanMorphism(identity_matrix(2), quadrant_bl, quadrant)
sage: fm.codomain_fan()
Rational polyhedral fan in 2-d lattice \(N\)
sage: fm.codomain_fan() \text{ is quadrant}
True
```

domain_fan\((\text{dim=}\text{None}, \text{codim=}\text{None})\)

Return the codomain fan of \textit{self}.

INPUT:

- \textit{dim} – dimension of the requested cones;
- \textit{codim} – codimension of the requested cones.

OUTPUT:

- \textit{rational polyhedral fan} if no parameters were given, \textit{tuple of cones} otherwise.

EXAMPLES:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: quadrant = Fan([quadrant])
sage: quadrant_bl = quadrant.subdivide([(1,1)])
sage: fm = FanMorphism(identity_matrix(2), quadrant_bl, quadrant)
sage: fm.domain_fan()
Rational polyhedral fan in 2-d lattice \(N\)
sage: fm.domain_fan() \text{ is quadrant_bl}
True
```
factor()

Factor self into injective * birational * surjective morphisms.

OUTPUT:

• a triple of FanMorphism $\phi_i, \phi_b, \phi_s$, such that $\phi_s$ is surjective, $\phi_b$ is birational, $\phi_i$ is injective, and $\text{self}$ 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 $\text{self}$ maps $\Sigma \to \Sigma'$, 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: 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: 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: 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: 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: phi_b.codomain_fan().rays()
N(-1, -1, 0),
N( 0, 1, 0),
N( 1, 0, 0)
in Sublattice <N(1, 0, 0), N(0, 1, 0)>

Coordinate plane inclusion (injective):

sage: phi_i
Fan morphism defined by the matrix
[1 0 0]
[0 1 0]
Domain fan: Rational polyhedral fan in Sublattice <N(1, 0, 0), N(0, 1, 0)>
Codomain fan: Rational polyhedral fan in 3-d lattice N
sage: phi_i.codomain_fan().rays()
N( 1, 0, 0),
N( 0, 1, 0),
N( 0, 0, 1),
N(-1, -1, -1)
in 3-d lattice N

image_cone(cone)
Return the cone of the codomain fan containing the image of cone.

INPUT:
• cone – a cone equivalent to a cone of the domain_fan() of self.

OUTPUT:
• a cone of the codomain_fan() of self.

EXAMPLES:
Sage: quadrant = Cone([(1,0), (0,1)])
Sage: quadrant = Fan([quadrant])
Sage: quadrant_bl = quadrant.subdivide([(1,1)])
Sage: fm = FanMorphism(identity_matrix(2), quadrant_bl, quadrant)
Sage: fm.image_cone(Cone([(1,0)]))
1-d cone of Rational polyhedral fan in 2-d lattice N
Sage: fm.image_cone(Cone([(1,1)]))
2-d cone of Rational polyhedral fan in 2-d lattice N

**index**(cone=None)

Return the index of self as a map between lattices.

**INPUT:**

- cone – (default: None) a cone of the codomain fan() of self.

**OUTPUT:**

- an integer, infinity, or None.

If no cone was specified, this function computes the index of the image of self in the codomain. If a cone \( \sigma \) was given, the index of self over \( \sigma \) is computed in the sense of Definition 2.1.7 of [HLY2002]: if \( \sigma' \) is any cone of the domain fan() of 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:**

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:

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()

(continues on next page)
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 \( d\mathbb{P}^8 \) (with the rest of this curve being uncovered) and the other to a fixed point of \( d\mathbb{P}^8 \) (thus completely covering this torus orbit in \( d\mathbb{P}^8 \)).

We should therefore expect the following behaviour: all indices over 1-d cones are \( \text{None} \), except for one which is infinite, and all indices over 2-d cones are \( \text{None} \), except for one which is 1:

\[
\begin{align*}
\text{sage: } & \text{[xi.index(cone) for cone in Sigma(1)]} \\
& \text{[None, None, None, None]} \\
\text{sage: } & \text{[xi.index(cone) for cone in Sigma(2)]} \\
& \text{[None, 1, None, None]}
\end{align*}
\]

**is_birational()**

Check if \( \text{self} \) is birational.

**OUTPUT:**

- True if \( \text{self} \) is birational, False otherwise.

For fan morphisms this check is equivalent to \( \text{self.index()} == 1 \) and means that the corresponding map between toric varieties is birational.

**EXAMPLES:**

\[
\begin{align*}
\text{sage: } & \text{Sigma = toric_varieties.dP8().fan()} \\
\text{sage: } & \text{Sigma_p = toric_varieties.P1().fan()} \\
\text{sage: } & \text{phi = FanMorphism(matrix([[1], [-1]]), Sigma, Sigma_p)} \\
\text{sage: } & \text{psi = FanMorphism(matrix([[2], [-2]]), Sigma, Sigma_p)} \\
\text{sage: } & \text{xi = FanMorphism(matrix([[1, 0]]), Sigma_p, Sigma)} \\
\text{sage: } & \text{phi.index(), psi.index(), xi.index()} \\
& (1, 2, +\text{Infinity}) \\
\text{sage: } & \text{phi.is_birational(), psi.is_birational(), xi.is_birational()} \\
& (\text{True}, \text{False}, \text{False})
\end{align*}
\]

**is_bundle()**

Check if \( \text{self} \) is a bundle.

**OUTPUT:**

- True if \( \text{self} \) 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 \( \tilde{\Sigma} \) of \( \Sigma \) such that the following two conditions are satisfied:

1. Cones of \( \Sigma \) are precisely the cones of the form \( \sigma_0 + \tilde{\sigma} \), where \( \sigma_0 \in \Sigma_0 \) and \( \tilde{\sigma} \in \tilde{\Sigma} \).
2. Cones of \( \tilde{\Sigma} \) are in bijection with cones of \( \Sigma' \) induced by \( \phi \) and \( \phi \) maps lattice points in every cone \( \tilde{\sigma} \in \tilde{\Sigma} \) bijectively onto lattice points in \( \phi(\tilde{\sigma}) \).

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.
EXAMPLES:

We consider several maps between fans of a del Pezzo surface and the projective line:

```python
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, $\phi_R : N_R \rightarrow N'_{R}$ 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:**

```python
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_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_injective()
Check if self is injective.

OUTPUT:

• True if self is injective, False otherwise.

Let $\phi : \Sigma \to \Sigma'$ be a fan morphism such that the underlying lattice morphism $\phi : N \to N'$ bijectively maps $N$ to a saturated sublattice of $N'$. Let $\psi : \Sigma \to \Sigma'_0$ be the restriction of $\phi$ to the image. Then $\phi$ is 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.

**See also:**

`factor()`.

**EXAMPLES:**

Consider the fan of the affine plane:

```python
sage: A2 = toric_varieties.A(2).fan()
```

We will map several fans consisting of a single ray into the interior of the 2-cone:

```python
sage: Sigma = Fan([Cone([(1,1)])])
sage: m = identity_matrix(2)
sage: FanMorphism(m, Sigma, A2).is_injective()
False
```

This morphism was not injective since (in the toric varieties interpretation) the 1-dimensional orbit corresponding to the ray was mapped to the 0-dimensional orbit corresponding to the 2-cone.

```python
sage: Sigma = Fan([Cone([(1,1)])])
sage: m = matrix(1, 2, [1,1])
sage: FanMorphism(m, Sigma, A2).is_injective()
True
```

While the fans in this example are close to the previous one, here the ray corresponds to a 0-dimensional orbit.

```python
sage: Sigma = Fan([Cone([(1,1)])])
sage: m = matrix(1, 2, [2,2])
sage: FanMorphism(m, Sigma, A2).is_injective()
False
```

Here the problem is that $m$ maps the domain lattice to a non-saturated sublattice of the codomain. The corresponding map of the toric varieties is a two-sheeted cover of its image.

We also embed the affine plane into the projective one:

```python
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, $\phi_R : N_R \to N'_R$ 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:

\texttt{is\_bundle()}, \texttt{is\_fibration()}, \texttt{preimage\_cones()}, \texttt{is\_complete()}.

EXAMPLES:

We check that the blow up of the affine plane at the origin is surjective:

\begin{verbatim}
  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

  sage: FanMorphism(m, Bl.cones(1), A2).is_surjective()
  True

  sage: F = Fan([Bl.generating_cone(0)])
  sage: FanMorphism(m, F, A2).is_surjective()
  False
\end{verbatim}

\texttt{kernel\_fan()}

Return the subfan of the domain fan mapped into the origin.

OUTPUT:

\begin{itemize}
  \item a \texttt{fan}.
\end{itemize}

\textbf{Note:} The lattice of the kernel fan is the \texttt{kernel()} sublattice of \texttt{self}.

See also:

\texttt{preimage\_fan()}.

EXAMPLES:

\begin{verbatim}
  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)>),)
\end{verbatim}

\texttt{preimage\_cones(\texttt{cone})}

Return cones of the domain fan whose \texttt{image\_cone()} is \texttt{cone}.

INPUT:

\begin{itemize}
  \item \texttt{cone} – a \texttt{cone} equivalent to a cone of the \texttt{codomain\_fan()} of \texttt{self}.
\end{itemize}

OUTPUT:
• a tuple of cones of the domain_fan() of self, sorted by dimension.

See also:

preimage_fan().

EXAMPLES:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: quadrant = Fan([quadrant])
sage: quadrant_bl = quadrant.subdivide([(1,1)])
sage: fm = FanMorphism(identity_matrix(2), quadrant_bl, quadrant)
sage: fm.preimage_cones(Cone([(1,0)]))
(1-d cone of Rational polyhedral fan in 2-d lattice N,)
sage: fm.preimage_cones(Cone([(1,0), (0,1)]))
(1-d cone of Rational polyhedral fan in 2-d lattice N,
 2-d cone of Rational polyhedral fan in 2-d lattice N,
 2-d cone of Rational polyhedral fan in 2-d lattice N)
```

preimage_fan(cone)

Return the subfan of the domain fan mapped into cone.

INPUT:

• cone – a cone equivalent to a cone of the codomain_fan() of self.

OUTPUT:

• a fan.

Note: The preimage fan of cone consists of all cones of the domain_fan() which are mapped into cone, including those that are mapped into its boundary. So this fan is not necessarily generated by preimage_cones() of cone.

See also:

kernel_fan(), preimage_cones().

EXAMPLES:

```
sage: quadrant_cone = Cone([(1,0), (0,1)])
sage: quadrant_fan = Fan([quadrant_cone])
sage: quadrant_bl = quadrant_fan.subdivide([(1,1)])
sage: fm = FanMorphism(identity_matrix(2), quadrant_bl, quadrant_fan)
.....: quadrant_bl, quadrant_fan)
sage: fm.preimage_fan(Cone([(1,0)])).cones()
((0-d cone of Rational polyhedral fan in 2-d lattice N,),
 (1-d cone of Rational polyhedral fan in 2-d lattice N,))
sage: fm.preimage_fan(quadrant_cone).ngenerating_cones()
2
sage: len(fm.preimage_cones(quadrant_cone))
3
```

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 \( \phi : \Sigma \rightarrow \Sigma' \) 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:

```python
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 \( \Sigma_p \):

```python
sage: len(phi.preimage_cones(Cone((1,))))
5
```

Yet only rays are primitive:

```python
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:

```python
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:

```python
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 \emph{domain_fan()} of self.

OUTPUT:

• \texttt{star_generators()} of \texttt{domain_cone} viewed as a cone of \texttt{preimage_fan()} of \texttt{image_cone()} of \texttt{domain_cone}.

EXAMPLES:

```python
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,)
```

2.4.6 Point collections

This module was designed as a part of framework for toric varieties (\emph{variety}, \emph{fano_variety}).

AUTHORS:


• Andrey Novoseltsev (2012-03-06): additions and doctest changes while switching cones to use point collections.

EXAMPLES:

The idea behind \texttt{point collections} is to have a container for points of the same space that

• behaves like a tuple \emph{without significant performance penalty}:

```python
sage: c = Cone([[(0,0,1), (1,0,1), (0,1,1), (1,1,1)]].rays()
sage: c[1]
N(1, 0, 1)
sage: for point in c: point
N(0, 0, 1)
N(1, 0, 1)
N(0, 1, 1)
N(1, 1, 1)
```

• prints in a convenient way and with clear indication of the ambient space:

```python
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:

```python
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.

```
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 $\mathbb{R}$-basis for the vector space spanned by the points of self.

**EXAMPLES:**
sage: c = Cone([\((0,0,1), (1,0,1), (0,1,1), (1,1,1)\)]).rays()
sage: c.basis()
\([\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} \])
in 3-d lattice \(\mathbb{N}\)

Calling this method twice will always return *exactly the same* point collection:

sage: c.basis().basis() is c.basis()
True

**cardinality()**

Return the number of points in `self`.

**OUTPUT:**

• an integer.

**EXAMPLES:**

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:**

sage: c = Cone([\((0,0,1), (1,1,1)\)]).rays()
sage: c.cartesian_product(c)
\([\begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 1 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \begin{bmatrix} 0 \\ 0 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}, \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \\ 1 \\ 1 \end{bmatrix} \])
in 6-d lattice \(\mathbb{N}+\mathbb{N}\)

**column_matrix()**

Return a matrix whose columns are points of `self`.

**OUTPUT:**

• a matrix.

**EXAMPLES:**

2.4. Toric geometry 381
dim()
Return the dimension of the space spanned by points of self.

**Note:** You can use either `dim()` or `dimension()`.

**OUTPUT:**
- an integer.

**EXAMPLES:**
```
sage: c = Cone([(0,0,1), (1,0,1), (0,1,1), (1,1,1)]).rays()
sage: c.column_matrix()
[0 1 0 1]
[0 0 1 1]
[1 1 1 1]
sage: c.dimension()
2
sage: c.dim()
2
```

dimension()
Return the dimension of the space spanned by points of self.

**Note:** You can use either `dim()` or `dimension()`.

**OUTPUT:**
- an integer.

**EXAMPLES:**
```
sage: c = Cone([(0,0,1), (1,1,1)]).rays()
sage: c.dimension()
2
sage: c.dim()
2
```

dual_module()
Return the dual of the ambient module of self.

**OUTPUT:**
- a module. If possible (that is, if the ambient `module()` 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), (1,1,1)]).rays()
sage: c.module()
3-d lattice M
sage: c.dual_module()
```

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:

```
sage: c = Cone([(0,0,1), (1,0,1), (0,1,1), (1,1,1)]).rays()
sage: c.module()
3-d lattice N
```

**static output_format(format=None)**

Return or set the output format for ALL point collections.
INPUT:

- **format** – (optional) if given, must be one of the strings
  - “default” – output one point per line with vertical alignment of coordinates in text mode, same as “tuple” for LaTeX;
  - “tuple” – output `tuple(self)` with lattice information;
  - “matrix” – output `matrix()` with lattice information;
  - “column matrix” – output `column_matrix()` with lattice information;
  - “separated column matrix” – same as “column matrix” for text mode, for LaTeX separate columns by lines (not shown by jsMath).

OUTPUT:

- a string with the current format (only if `format` was omitted).

This function affects both regular and LaTeX output.

EXAMPLES:

```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")
sage: c
[0 1 0 1]
[0 0 1 1]
[1 1 1 1]
in 3-d lattice N
```

Note that the last two outputs are identical, separators are only inserted in the LaTeX mode:
sage: latex(c)
\left(\begin{array}{rrrr}
0 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 \\
1 & 1 & 1 & 1
\end{array}\right)_{N}

Since this is a static method, you can call it for the class directly:

```
sage: from sage.geometry.point_collection import PointCollection
sage: PointCollection.output_format("default")

sage: c
N(0, 0, 1),
N(1, 0, 1),
N(0, 1, 1),
N(1, 1, 1)
in 3-d lattice N
```

**set()**

Return points of self as a frozenset.

**OUTPUT:**

- a frozenset.

**EXAMPLES:**

```
sage: c = Cone([(0,0,1), (1,0,1), (0,1,1), (1,1,1)]).rays()

sage: c.set()
set(N(0, 0, 1), N(0, 1, 1), N(1, 0, 1), N(1, 1, 1))
```

**write_for_palp(f)**

Write self into an open file f in PALP format.

**INPUT:**

- f – a file opened for writing.

**EXAMPLES:**

```
sage: o = lattice_polytope.cross_polytope(3)
sage: from io import StringIO
sage: f = StringIO()
sage: o.vertices().write_for_palp(f)
sage: print(f.getvalue())
6 3
1 0 0
0 1 0
0 0 1
-1 0 0
0 -1 0
0 0 -1
```

`sage.geometry.point_collection.is_PointCollection(x)`

Check if x is a point collection.

**INPUT:**

2.4. Toric geometry 385
• \( x \) – anything.

OUTPUT:
• True if \( x \) is a point collection and False otherwise.

EXAMPLES:

```python
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
```

.. 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.
• \( \text{lattice} \) – the lattice for points. If not given, the toric lattice \( M \) of dimension \( n \) will be used.
• \( \text{permutation} \) – (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:

```python
sage: data = "3 2 regular
1 2
3 4
5 6
2 3 transposed
1 2
3 4
5 6"
sage: print(data)
3 2 regular
1 2
3 4
5 6
2 3 transposed
1 2
3 4
5 6
sage: from io import StringIO
sage: f = StringIO(data)
sage: from sage.geometry.point_collection.read_palp_point_collection

(continues on next page)
```
M(1, 2),
M(3, 4),
M(5, 6)
in 2-d lattice $\mathbb{M}$
sage: read_palp_point_collection(f)
M(1, 4),
M(2, 5),
M(3, 6)
in 2-d lattice $\mathbb{M}$
sage: read_palp_point_collection(f) is None
True

2.4.7 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 adjust_options() and set_rays() again if you decide to change the plotting mode and/or cut-off bounds. Otherwise plots maybe invalid.

```
adjust_options()
```

Adjust plotting options.
This function determines appropriate default values for those options, that were not specified by the user, based on the other options. See *ToricPlotter* for a detailed example.

**OUTPUT:**
- none.

**include_points(points, force=False)**

Try to include `points` into the bounding box of `self`.

**INPUT:**
- `points` – 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:**

```python
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 ray generators.

Ray generators must be specified during construction or using *set_rays()* before calling this method.

**OUTPUT:**
- a plot.

**EXAMPLES:**

```python
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(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:

```sage
def from sage.geometry.toric_plotter import ToricPlotter
def from sage.geometry.toric_plotter import ToricPlotter
def from sage.geometry.toric_plotter import ToricPlotter
def from sage.geometry.toric_plotter import ToricPlotter
def from sage.geometry.toric_plotter import ToricPlotter
sage: from sage.geometry.toric_plotter import ToricPlotter
sage: tp = ToricPlotter(dict(), 2)
sage: tp.plot_labels("u", [(1.5,0)])
```

```sage
def from sage.geometry.toric_plotter import ToricPlotter
sage: from sage.geometry.toric_plotter import ToricPlotter
sage: tp = ToricPlotter(dict(), 2)
sage: tp.plot_labels("u", [(1.5,0)])
```

### plot_lattice()

Plot the lattice (i.e. its points in the cut-off bounds of self).

OUTPUT:

• a plot.

EXAMPLES:

```sage
def from sage.geometry.toric_plotter import ToricPlotter
sage: from sage.geometry.toric_plotter import ToricPlotter
sage: tp = ToricPlotter(dict(), 2)
sage: tp.plot_lattice()
```

### plot_points(points)

Plot given points.

INPUT:

• points – a list of points.

OUTPUT:

• a plot.

EXAMPLES:

```sage
def from sage.geometry.toric_plotter import ToricPlotter
sage: from sage.geometry.toric_plotter import ToricPlotter
sage: tp = ToricPlotter(dict(), 2)
sage: tp.plot_points([(1,0), (0,1)])
```

### 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
def from sage.geometry.toric_plotter import ToricPlotter
sage: from sage.geometry.toric_plotter import ToricPlotter
sage: tp = ToricPlotter(dict(), 2, [(3,4)])
sage: tp.plot_ray_labels()
```

```sage
def from sage.geometry.toric_plotter import ToricPlotter
sage: from sage.geometry.toric_plotter import ToricPlotter
sage: tp = ToricPlotter(dict(), 2)
sage: tp.plot_point([(1,0), (0,1)])
```
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
tp = ToricPlotter(dict(), 2, [(3,4)])
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)])
from sage.geometry.toric_plotter import ToricPlotter
tp = ToricPlotter(dict(), 2, quadrant.rays())
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:
```sage
from sage.geometry.toric_plotter import ToricPlotter
tp = ToricPlotter(dict(), 2)
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

sage.geometry.toric_plotter.color_list(color, n)

Normalize a list of n colors.

INPUT:
• color – anything specifying a Color, a list of such specifications, or the string “rainbow”;
• n - 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:

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!

sage.geometry.toric_plotter.label_list(label, n, math_mode, index_set=None)

Normalize a list of n labels.

INPUT:
• label – None, a string, or a list of string;
• n - 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.

OUTPUT:
• a list of \( n \) labels.

If \( \text{label} \) was a list of \( n \) entries, it is returned without changes. If \( \text{label} \) is \texttt{None}, a list of \( n \) \texttt{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}=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\$']
```

```
sage.geometry.toric_plotter.options(option=\texttt{None}, **kwds)
```

Get or set options for plots of toric geometry objects.

**Note:** This function provides access to global default options. Any of these options can be overridden by passing them directly to plotting functions. See also \texttt{reset\_options()}.

**INPUT:**

• \texttt{None};

**OR:**

• \texttt{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 \texttt{option} argument was given, the current value of \texttt{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:

\begin{verbatim}
sage: toric_plotter.options(wall_color="rainbow")
\end{verbatim}

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:

\begin{verbatim}
sage: toric_plotter.options(wall_color="grey")
\end{verbatim}

\begin{verbatim}
sage.geometry.toric_plotter.reset_options()
\end{verbatim}

Reset options for plots of toric geometry objects.

OUTPUT:

• none.

EXAMPLES:

\begin{verbatim}
sage: toric_plotter.options("show_rays")
True
sage: toric_plotter.options(show_rays=False)
sage: toric_plotter.options("show_rays")
False
\end{verbatim}

Now all toric plots will not show rays, unless explicitly requested. If you want to go back to “default defaults”, use this method:

\begin{verbatim}
sage: toric_plotter.reset_options()
sage: toric_plotter.options("show_rays")
True
\end{verbatim}

\begin{verbatim}
sage.geometry.toric_plotter.sector(ray1, ray2, **extra_options)
\end{verbatim}

Plot a sector between ray1 and ray2 centered at the origin.

\textbf{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.

\textbf{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:
2.4.8 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:

```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

```sage
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
```
• **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:**

```
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:

```
sage: R.<x,y,z> = QQ[]
sage: I = R.ideal([(x+y+z)^3-1,(x+y+z)^3-x,(x+y+z)-3])
sage: GF = I.groebner_fan()
sage: PF = GF.tropical_intersection()
sage: PF.rays()
[-1, 0, 0], [0, -1, 0], [0, 0, -1], [1, 1, 1]
sage: RPF = PF.to_RationalPolyhedralFan()
sage: RPF.Stanley_Reisner_ideal(PolynomialRing(QQ,4,'A, B, C, D'))
Ideal (A*B, A*C, B*C*D) of Multivariate Polynomial Ring in A, B, C, D over Rational...
˓→Field
```

**buchberger()**

Return a lexicographic reduced Groebner basis for the ideal.

**EXAMPLES:**

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: G = R.ideal([x - z^3, y^2 - x + x^2 - z^3*x]).groebner_fan()
sage: G.buchberger()
[-z^3 + y^2, -z^3 + x]
```

**characteristic()**

Return the characteristic of the base ring.

**EXAMPLES:**

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: i1 = ideal(x^3*z + 6*y^3*z - z^2, x^3*y + 6*x^2*z + y^3*z - z^2, y^2 + x^2*z + y^3*z)
sage: gf = i1.groebner_fan()
sage: gf.characteristic()
0
```

**dimension_of_homogeneity_space()**

Return the dimension of the homogeneity space.

**EXAMPLES:**

```
```
```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: G = R.ideal([y^3 - x^2, y^2 - 13*x]).groebner_fan()
sage: G.dimension_of_homogeneity_space()
0
```

`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:**

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: gf = R.ideal([x^3-y,y^3-x-1]).groebner_fan()
sage: gf.gfan()
'Q[x,y]\n{{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,y-x^3\}\n
homogeneity_space()`

Return the homogeneity space of a the list of polynomials that define this Groebner fan.

**EXAMPLES:**

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: G = R.ideal([y^3 - x^2, y^2 - 13*x]).groebner_fan()
sage: H = G.homogeneity_space()
```

`ideal()`

Return the ideal the was used to define this Groebner fan.

**EXAMPLES:**

```
sage: R.<x1,x2> = PolynomialRing(QQ,2)
sage: gf = R.ideal([x1^3-x2,x2^3-2*x1-2]).groebner_fan()
sage: gf.ideal()
Ideal (x1^3 - x2, x2^3 - 2*x1 - 2) of Multivariate Polynomial Ring in x1, x2 over Rational Field
```

`interactive(*args, **kwds)`

See the documentation for `self[0].interactive()`. This does not work with the notebook.

**EXAMPLES:**

```
sage: print("This is not easily doc-testable; please write a good one!")
This is not easily doc-testable; please write a good one!
```

`maximal_total_degree_of_a_groebner_basis()`

Return the maximal total degree of any Groebner basis.

**EXAMPLES:**

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: G = R.ideal([y^3 - x^2, y^2 - 13*x]).groebner_fan()
sage: G.maximal_total_degree_of_a_groebner_basis()
4
```

`minimal_total_degree_of_a_groebner_basis()`

Return the minimal total degree of any Groebner basis.
EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: G = R.ideal([y^3 - x^2, y^2 - 13*x]).groebner_fan()
sage: G.minimal_total_degree_of_a_groebner_basis()
2
```

**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
```

**number_of_reduced_groebner_bases()**

Return the number of reduced Grobner bases.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: G = R.ideal([y^3 - x^2, y^2 - 13*x]).groebner_fan()
sage: G.number_of_reduced_groebner_bases()
3
```

**number_of_variables()**

Return the number of variables.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: G = R.ideal([y^3 - x^2, y^2 - 13*x]).groebner_fan()
sage: G.number_of_variables()
2
```

```sage
R = PolynomialRing(QQ,'x',10)
sage: R.inject_variables(globals())
Defining x0, x1, x2, x3, x4, x5, x6, x7, x8, x9
sage: G = ideal([x0 - x9, sum(R.gens())]).groebner_fan()
sage: G.number_of_variables()
10
```

2.4. Toric geometry
polyhedralfan()

Return a polyhedral fan object corresponding to the reduced Groebner bases.

EXAMPLES:

```
sage: R3.<x,y,z> = PolynomialRing(QQ,3)
sage: gf = R3.ideal([x^8-y^4,y^4-z^2,z^2-1]).groebner_fan()
sage: pf = gf.polyhedralfan()
sage: pf.rays()
[[0, 0, 1], [0, 1, 0], [1, 0, 0]]
```

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 - y + z^9, y - z^11]
```

render(file=None, larger=False, shift=0, rgbcolor=(0, 0, 0), polyfill=<function max_degree at 0x7f0c56b85d30>, 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.
- `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.
- `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.
- `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)
G = R.ideal([y^3 - x^2, y^2 - 13*x,z]).groebner_fan()
test_render = G.render()
```

```sage
R.<x,y,z> = PolynomialRing(QQ,3)
G = R.ideal([x^2*y - z, y^2*z - x, z^2*x - y]).groebner_fan()
test_render = G.render(larger=True)
```

`render3d(verbos=False)`
For a Groebner fan of an ideal in a ring with four variables, this function intersects the fan with the standard simplex perpendicular to (1,1,1,1), creating a 3d polytope, which is then projected into 3 dimensions. The edges of this projected polytope are returned as lines.

EXAMPLES:

```sage
R4.<w,x,y,z> = PolynomialRing(QQ,4)
gf = R4.ideal([w^2-x,x^2-y,y^2-z,z^2-x]).groebner_fan()
three_d = gf.render3d()
```

`ring()`
Return the multivariate polynomial ring.

EXAMPLES:

```sage
R.<x1,x2> = PolynomialRing(QQ,2)
gf = R.ideal([x1^3-x2,x2^3-x1-2]).groebner_fan()
gf.ring()
```

```
Multivariate Polynomial Ring in x1, x2 over Rational Field
```

`tropical_basis(check=True, verbos=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:

```sage
R.<x,y,z> = PolynomialRing(QQ,3, order='lex')
g = R.ideal([y^3-3*x^2, z^3-x-y-2*y^3+2*x^2]).groebner_fan()
g Groebner fan of the ideal:
Ideal (-3*x^2 + y^3, 2*x^2 - x - 2*y^3 - y + z^3) of Multivariate Polynomial Ring in x, y, z over Rational Field
g.tropical_basis()
```

```
[-3*x^2 + y^3, 2*x^2 - x - 2*y^3 - y + z^3, 3/4*x + y^3 + 3/4*y - 3/4*z^3]
```

`tropical_intersection(parameters=[], symmetry_generators=[], *args, **kwds)`
Return information about the tropical intersection of the polynomials defining the ideal.
This is the common refinement of the outward-pointing normal fans of the Newton polytopes of the generators of the ideal. Note that some people use the inward-pointing normal fans.

INPUT:

- **parameters** (optional) - 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^2 + 6*y^2 + z^2, x*y + 6*x*z + y*z + z^2, y^2 + x^2 + y^2 + z^2)

sage: gf = I.groebner_fan()

sage: pf = gf.tropical_intersection()

sage: pf.rays()

\[[-2, 1, 1]\]

sage: R.<x,y,z> = PolynomialRing(QQ,3)

sage: f1 = x*y*z - 1

sage: f2 = f1*(x^2 + y^2 + z^2)

sage: f3 = f2*(x + y + z - 1)

sage: I = R.ideal([f1,f2,f3])

sage: gf = I.groebner_fan()

sage: pf = gf.tropical_intersection(symmetry_generators = '(1,2,0),(1,0,2)')

sage: pf.rays()

\[[-2, 1, 1], [1, -2, 1], [1, 1, -2]\]

sage: R.<x,y,z> = QQ[

sage: I = R.ideal([(x+y+z)^2-1,(x+y+z)-x,(x+y+z)-3])

sage: GF = I.groebner_fan()

sage: TI = GF.tropical_intersection()

sage: TI.rays()

\[[-1, 0, 0], [0, -1, -1], [1, 1, 1]\]

sage: GF = I.groebner_fan()

sage: TI = GF.tropical_intersection(parameters=[y])

sage: TI.rays()

\[[-1, 0, 0]\]
```

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()

\[[(3, 7, 1), (5, 1, 2), (7, 1, 4), (5, 1, 4), (1, 1, 1), (1, 4, 8), (1, 4, 10)]\]

sage: r4.<x,y,z,w> = PolynomialRing(QQ,4)

sage: g4 = r4.ideal([x^3+y,y^3-z,z^2-x,z^3-w]).groebner_fan()

sage: len(g4.weight_vectors())

23
```
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[]
```

```python
sage: inform = InitialForm([0], [[-1, 0]], [y^2 - 1, y^2 - 2, y^2 - 3])
```

```python
sage: inform._cone
[0]
```

**cone()**

The cone associated with the initial form system.

**EXAMPLES:**

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

```python
sage: I = R.ideal([(x+y)^2-1,(x+y)^2-2,(x+y)^2-3])
```

```python
sage: GF = I.groebner_fan()
```

```python
sage: PF = GF.tropical_intersection()
```

```python
sage: pfi0 = PF.initial_form_systems()[0]
```

```python
sage: pfi0.cone()
[0]
```

**initial_forms()**

The initial forms (polynomials).

**EXAMPLES:**

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

```python
sage: I = R.ideal([(x+y)^2-1,(x+y)^2-2,(x+y)^2-3])
```

```python
sage: GF = I.groebner_fan()
```

```python
sage: PF = GF.tropical_intersection()
```

```python
sage: pfi0 = PF.initial_form_systems()[0]
```

```python
sage: pfi0.initial_forms()
[y^2 - 1, y^2 - 2, y^2 - 3]
```

**internal_ray()**

A ray internal to the cone associated with the initial form system.

**EXAMPLES:**

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

```python
sage: I = R.ideal([(x+y)^2-1,(x+y)^2-2,(x+y)^2-3])
```

```python
sage: GF = I.groebner_fan()
```

```python
sage: PF = GF.tropical_intersection()
```

```python
sage: pfi0 = PF.initial_form_systems()[0]
```

```python
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[]
```

```python
sage: I = R.ideal([(x+y)^2-1,(x+y)^2-2,(x+y)^2-3])
```

```python
sage: GF = I.groebner_fan()
```

```python
sage: PF = GF.tropical_intersection()
```

```python
sage: pfi0 = PF.initial_form_systems()[0]
```

```python
sage: pfi0.rays()
(-1, 0)
```
```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]]]
```

```python
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:

```python
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:

```python
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:

```python
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:

```python
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.

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=None)

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> = 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]]

ambient_dim()

Return the ambient dimension of the Groebner fan.

EXAMPLES:

```
sage: R3.<x,y,z> = PolynomialRing(QQ,3)
sage: gf = R3.ideal([x^8-y^4,y^4-z^2,z^2-2]).groebner_fan()
sage: a = gf.polyhedralfan()
sage: a.ambient_dim()
3
```

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:
```python
sage: R.<x,y,z> = QQ[]
sage: f = 1+x+y+x*y
sage: I = R.ideal([f+z*f, 2*f+z*f, 3*f+z^2*f])
sage: GF = I.groebner_fan()
sage: PF = GF.tropical_intersection()
sage: PF.cones()
{1: [[0], [1], [2], [3], [4], [5]], 2: [[0, 1], [0, 2], [0, 3], [0, 4], [1, 2], ...}
```
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:

```python
sage: R.<x,y,z> = QQ[]
sage: f = 1+x+y+x*y
sage: I = R.ideal([f+z*f, 2*f+z^2*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:

```python
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: i2 = ideal(x^2 + 6*y^2 - z^2, x*y + 6*x*z + y*z - z^2, y^2 + x*z + y*z)
sage: gf2 = i2.groebner_fan(verbos=False)
sage: pf = gf2.polyhedralfan()
sage: pf.rays()
[[1, -1, 0], [1, 1, 0], [1, 1, 1], [1, 1, -1], [1, -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:

```python
sage: R.<x,y,z> = QQ[]
sage: f = 1+x+y+x*y
sage: I = R.ideal([f+z*f, 2*f+z^2*f, 3*f+z^2*f])
sage: GF = I.groebner_fan()
sage: PF = GF.tropical_intersection()
sage: fan = PF.to_RationalPolyhedralFan()
sage: tuple(q.facet_normals()) for q in fan
[(M(0, -1, 0), M(-1, 0, 0)), (M(0, 0, -1), M(-1, 0, 0)), (M(0, 0, 1), M(-1, 0, 0, 0)), (M(0, 1, 0), M(-1, 0, 0)), (M(0, 1, 0), M(-1, 0, 0)), (M(0, 1, 0), M(-1, 0, 0))]
```

Here we use the RationalPolyhedralFan’s Gale_transform method on a tropical prevariety.

```python
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:
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=None)
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:

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:

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:

sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: polys = [x^2*y - z, y^2*z - x, z^2*x - y]
sage: from sage.rings.polynomial.groebner_fan import ideal_to_gfan_format
sage: ideal_to_gfan_format(R, polys)
'Q[x, y, z]{x^2*y-z,y^2*z-x,x*z^2-y}'

```
Q[x, y, z]{x^2*y-z,y^2*z-x,x*z^2-y}
```

sage.rings.polynomial.groebner_fan.max_degree(list_of_polys)
Compute the maximum degree of a list of polynomials

EXAMPLES:

```
sage: from sage.rings.polynomial.groebner_fan import max_degree
dsage: R.<x,y> = PolynomialRing(QQ,2)
dsage: p_list = [x^2-y,x*y^10-x]
dsage: max_degree(p_list)
11.0
```

sage.rings.polynomial.groebner_fan.prefix_check(str_list)
Check if any strings in a list are prefixes of another string in the list.

EXAMPLES:

```
sage: from sage.rings.polynomial.groebner_fan import prefix_check
dsage: prefix_check(['z1','z1z1'])
False
dsage: prefix_check(['z1','zz1'])
True
```

sage.rings.polynomial.groebner_fan.ring_to_gfan_format(input_ring)
Converts a ring to gfan's format.

EXAMPLES:

```
sage: R.<w,x,y,z> = QQ[]
sage: from sage.rings.polynomial.groebner_fan import ring_to_gfan_format
sage: ring_to_gfan_format(R)
'Q[w, x, y, z]'
sage: R2.<x,y> = GF(2)[]
sage: ring_to_gfan_format(R2)
'Z/2Z[x, y]'
```

sage.rings.polynomial.groebner_fan.verts_for_normal(normal, poly)
Return the exponents of the vertices of a Newton polytope that make up the supporting hyperplane for the given outward normal.

EXAMPLES:

```
sage: from sage.rings.polynomial.groebner_fan import verts_for_normal
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: f1 = x*y*z - 1
sage: f2 = f1*(x^2 + y^2 + 1)
sage: verts_for_normal([1,1,1],f2)
[(3, 1, 1), (1, 3, 1)]
```
2.5 Base classes for polyhedra

2.5.1 Base class for polyhedra

class sage.geometry.polyhedron.base.Polyhedron_base(parent, Vrep, Hrep, Vrep_minimal=None, Hrep_minimal=None, pref_rep=None, mutable=False, **kwds):


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.

• Vrep_minimal (optional) – see below

• Hrep_minimal (optional) – see below

• pref_rep – string (default: None); one of `Vrep` or Hrep to pick this in case the backend cannot initialize from complete double description

• mutable – ignored

If both Vrep and Hrep are provided, then Vrep_minimal and Hrep_minimal must be set to True.

Hrepresentation(index=None)

Return the objects of the H-representation. Each entry is either an inequality or an 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: next(p.Hrepresentation_generator())
An inequality (-1, 0, 0) x + 1 >= 0
```

Hrepresentation(index=None)

Return an iterator over the objects of the H-representation (inequalities or equations).

EXAMPLES:

```
sage: p = polytopes.hypercube(3)
sage: next(p.Hrepresentation_generator())
An inequality (-1, 0, 0) x + 1 >= 0
```

Hrepresentation(index=None)

Return the objects of the H-representation. Each entry is either an inequality or a equation.

INPUT:

• index – either an integer or None

OUTPUT:

The optional argument is an index running from 0 to self.n_Hrepresentation()-1. If present, the H-representation object at the given index will be returned. Without an argument, returns the list of all H-representation objects.

EXAMPLES:

```
sage: p = polytopes.hypercube(3, backend='field')
sage: p.Hrepresentation(0)
An inequality (-1, 0, 0) x + 1 >= 0
```

sage: p.Hrepresentation(0) == p.Hrepresentation()[0]
True
**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:**

```python
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
  x0 + x1 >= 3
   -x0 - x1 >= -5
    x1 >= 1
      -x0 >= -3
      x0 >= 1
      -x1 >= -3

sage: print(P.Hrepresentation_str(style='<='))
-x0 - x1 - x2 == -6
  -x0 - x1 <= -3
   x0 + x1 <= 5
    -x1 <= -1
     x0 <= 3
     -x0 <= -1
     x1 <= 3

sage: print(P.Hrepresentation_str(style='positive'))
```

(continues on next page)
\begin{align*}
x_0 + x_1 + x_2 &= 6 \\
x_0 + x_1 &\geq 3 \\
5 &\leq x_0 + x_1 \\
x_1 &\geq 1 \\
3 &\leq x_0 \\
x_0 &\geq 1 \\
3 &\leq x_1
\end{align*}

\texttt{sage: print(P.Hrepresentation_str(latex=True))}
\begin{array}{rcl}
x_{0} + x_{1} + x_{2} & = & 6 \\
x_{0} + x_{1} & \geq & 3 \\
-x_{0} - x_{1} & \geq & -5 \\
x_{1} & \geq & 1 \\
-x_{0} & \geq & -3 \\
x_{0} & \geq & 1 \\
-x_{1} & \geq & -3
\end{array}
\texttt{sage: print(P.Hrepresentation_str(align=False))}
\begin{align*}
x_0 + x_1 + x_2 &= 6 \\
x_0 + x_1 &\geq 3 \\
-x_0 - x_1 &\geq -5 \\
x_1 &\geq 1 \\
-x_0 &\geq -3 \\
x_0 &\geq 1 \\
-x_1 &\geq -3
\end{align*}
\texttt{sage: c = polytopes.cube()}
\texttt{sage: c.Hrepresentation_str(separator='\,', style='positive')}
'1 >= x0, 1 >= x1, 1 >= x2, x0 + 1 >= 0, x2 + 1 >= 0, x1 + 1 >= 0'

\textbf{Vrep_generator()}
Return an iterator over the objects of the V-representation (vertices, rays, and lines).

\textbf{EXAMPLES:}
\begin{verbatim}
\texttt{sage: p = polytopes.cyclic_polytope(3,4)
\texttt{sage: vg = p.Vrep_generator()}
\texttt{sage: next(vg)}
A vertex at (0, 0, 0)
\texttt{sage: next(vg)}
A vertex at (1, 1, 1)
\end{verbatim}

\textbf{Vrepresentation(index=None)}
Return the objects of the V-representation. Each entry is either a vertex, a ray, or a line.

See \texttt{sage.geometry.polyhedron.constructor} for a definition of vertex/ray/line.

\textbf{INPUT:}

- index – either an integer or \texttt{None}

\textbf{OUTPUT:}
The optional argument is an index running from 0 to \texttt{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 free module.

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
```

```
a_maximal_chain()
Return a maximal chain of the face lattice in increasing order.

EXAMPLES:

```python
sage: P = polytopes.cube()
sage: P.a_maximal_chain()
[A -1-dimensional face of a Polyhedron in ZZ^3,
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1 vertex,
A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2 vertices,
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4 vertices,
A 3-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 8 vertices]
sage: P = polytopes.cube()
sage: chain = P.a_maximal_chain(); chain
[A -1-dimensional face of a Polyhedron in ZZ^3,
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1 vertex,
A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2 vertices,
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4 vertices,
A 3-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 8 vertices]
sage: [face.ambient_V_indices() for face in chain]
[(), (5,), (0, 5), (0, 3, 4, 5), (0, 1, 2, 3, 4, 5, 6, 7)]
```
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 \texttt{Vrepresentation()} objects: vertices, rays, and lines. The \((i,j)\) matrix entry equals 1 if the \(i\)-th and \(j\)-th \texttt{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 \texttt{V-representation} objects (see \texttt{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]
```

In one dimension higher, that is for a half-space in 3 dimensions, there is no one-dimensional face. Hence nothing is adjacent:

```
sage: Polyhedron(ieqs=[(0,1,0,0)]).vertex_adjacency_matrix()
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
```

EXAMPLES:

In a bounded polygon, every vertex has precisely two adjacent ones:
If the V-representation of the polygon contains vertices and one ray, then each V-representation object is adjacent to two V-representation objects:

```
sage: P = Polyhedron(vertices=[(0, 1), (1, 0), (3, 0), (4, 1)],
                   rays=[(0,1)])
sage: for v in P.Vrep_generator():
    print("{} {}") .format(P.adjacency_matrix().row(v.index()), v))
(0, 1, 0, 0) 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, 1) A vertex at (3, 0)
(1, 0, 0, 1, 0) A vertex at (4, 1)
```

If the V-representation of the polygon contains vertices and two distinct rays, then each vertex is adjacent to two V-representation objects (which can now be vertices or rays). The two rays are not adjacent to each other:

```
sage: P = Polyhedron(vertices=[(0, 1), (1, 0), (3, 0), (4, 1)],
                   rays=[(0,1), (1,1)])
sage: for v in P.Vrep_generator():
    print("{} {}") .format(P.adjacency_matrix().row(v.index()), v))
(0, 1, 0, 0) 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, 0, 1) A ray in the direction (1, 1)
(0, 0, 1, 0, 1) A vertex at (3, 0)
(1, 0, 0, 1, 0) A vertex at (4, 1)
```

The vertex adjacency matrix has base ring integers. This way one can express various counting questions:

```
sage: P = polytopes.cube()
sage: Q = P.stack(P.faces(2)[0])
sage: M = Q.vertex_adjacency_matrix()
sage: sum(M)
(4, 4, 3, 3, 4, 4, 4, 3, 3)
sage: G = Q.vertex_graph()
sage: G.degree()
[4, 4, 3, 3, 4, 4, 4, 3, 3]
```

```
affine_hull(*args, **kwds)
Return the affine hull of self as a polyhedron.

EXAMPLES:
```
```
affine_hull_manifold(name=None, latex_name=None, start_index=0, ambient_space=None, ambient_chart=None, names=None, **kwds)

Return the affine hull of self as a manifold.

If self is full-dimensional, it is just the ambient Euclidean space. Otherwise, it is a Riemannian submanifold of the ambient Euclidean space.

INPUT:

• ambient_space – a EuclideanSpace of the ambient dimension (default: the manifold of ambient_chart, if provided; otherwise, a new instance of EuclideanSpace).

• ambient_chart – a chart on ambient_space.

• names – names for the coordinates on the affine hull.

• optional arguments accepted by affine_hull_projection().

The default chart is determined by the optional arguments of affine_hull_projection().

EXAMPLES:

sage: triangle = Polyhedron([(1,0,0), (0,1,0), (0,0,1)]); triangle
A 2-dimensional polyhedron in ZZ^3 defined as the convex hull of 3 vertices
sage: A = triangle.affine_hull_manifold(name='A'); A
2-dimensional Riemannian submanifold A embedded in the Euclidean space E^3
sage: A.embedding().display()
A → E^3
  (x0, x1) ↦ (x, y, z) = (t0 + x0, t0 + x1, t0 - x0 - x1 + 1)
sage: A.embedding().inverse().display()
E^3 → A
  (x, y, z) ↦ (x0, x1) = (x, y)
sage: A.adapted_chart()
[Chart (E^3, (x0_E3, x1_E3, t0_E3))]
sage: A.normal().display()
n = 1/3*sqrt(3) e_x + 1/3*sqrt(3) e_y + 1/3*sqrt(3) e_z
sage: A.induced_metric()  # Need to call this before volume_form
Riemannian metric gamma on the 2-dimensional Riemannian submanifold A embedded in the Euclidean space E^3
sage: A.volume_form()
2-form eps_gamma on the 2-dimensional Riemannian submanifold A embedded in the Euclidean space E^3

Orthogonal version:

sage: A = triangle.affine_hull_manifold(name='A', orthogonal=True); A
2-dimensional Riemannian submanifold A embedded in the Euclidean space E^3
sage: A.embedding().display()
A → E^3
  (x0, x1) ↦ (x, y, z) = (t0 - 1/2*x0 - 1/3*x1 + 1, t0 + 1/2*x0 - 1/3*x1, t0 + 2/3*x1)
sage: A.embedding().inverse().display()
Arrangement of affine hull of facets:

```python
e^3 \rightarrow A \\
(x, y, z) \mapsto (x_0, x_1) = (-x + y + 1, -1/2*x - 1/2*y + z + 1/2)
```

```python
sage: D = polytopes.dodecahedron()
sage: E3 = EuclideanSpace(3)
sage: submanifolds = [ 
....:   F.as_polyhedron().affine_hull_manifold(name=f'F{i}', orthogonal=True, 
 \ 
˓→ ambient_space=E3)
....:   for i, F in enumerate(D.facets())]
sage: sum(FM.plot({}, srange(-2, 2, 0.1), srange(-2, 2, 0.1), opacity=0.2) 
\ 
˓→ not tested
....:   for FM in submanifolds) + D.plot()
Graphics3d Object
```

Full-dimensional case:

```python
sage: cube = polytopes.cube(); cube
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 8 vertices
sage: cube.affine_hull_manifold()
Euclidean space E^3
```

**affine_hull_projection**

(as\_polyhedron, as\_affine\_map=None, orthogonal=False, orthonormal=False, 
extend=False, minimal=False, return\_all\_data=False, as\_convex\_set=False)

Return the polyhedron projected into its affine hull.

Each polyhedron is contained in some smallest affine subspace (possibly the entire ambient space) – its 
affine hull. We provide an affine linear map that projects the ambient space of the polyhedron to the standard 
Euclidean space of dimension of the polyhedron, which restricts to a bijection from the affine hull.

The projection map is not unique; some parameters control the choice of the map. Other parameters control 
the output of the function.

**INPUT:**

- as\_polyhedron (or as\_convex\_set) – (boolean or the default None) and
- as\_affine\_map – (boolean, default False) control the output

The default as\_polyhedron=None translates to as\_polyhedron=not as\_affine\_map, therefore 
to as\_polyhedron=\textTrue if nothing is specified.

If exactly one of either as\_polyhedron or as\_affine\_map is set, then either a polyhedron or 
the affine transformation is returned. The affine transformation sends the embedded polytope to a 
fulldimensional one. It is given as a pair (A, b), where A is a linear transformation and b is a vector, 
and the affine transformation sends v to A(v)+b.

If both as\_polyhedron and as\_affine\_map are set, then both are returned, encapsulated in an insta-
tion of AffineHullProjectionData.

- return\_all\_data – (boolean, default False)

If set, then as\_polyhedron and as\_affine\_map will set (possibly overridden) and additional (in-
ternal) data concerning the transformation is returned. Everything is encapsulated in an instance of 
AffineHullProjectionData in this case.

- orthogonal – boolean (default: False); if True, provide an orthogonal transformation.
• orthonormal – boolean (default: False); if True, provide an orthonormal transformation. If the
base ring does not provide the necessary square roots, the extend parameter needs to be set to True.

• extend – boolean (default: False); if True, allow base ring to be extended if necessary. This becomes
relevant when requiring an orthonormal transformation.

• minimal – boolean (default: False); if True, when doing an extension, it computes the minimal base
ring of the extension, otherwise the base ring is \( \mathbb{A} \).

OUTPUT:
A full-dimensional polyhedron or an affine transformation, depending on the parameters as_polyhedron
and as_affine_map, or an instance of AffineHullProjectionData containing all data (parameter
return_all_data).

If the output is an instance of AffineHullProjectionData, the following fields may be set:

• image – the projection of the original polyhedron
• projection_map – the affine map as a pair whose first component is a linear transformation and its
second component a shift; see above.
• section_map – an affine map as a pair whose first component is a linear transformation and its second
component a shift. It maps the codomain of affine_map to the affine hull of self. It is a right inverse
of projection_map.

Note that all of these data are compatible.

Todo:
• make the parameters orthogonal and orthonormal work with unbounded polyhedra.

EXAMPLES:

```
sage: triangle = Polyhedron([(1,0,0), (0,1,0), (0,0,1)]); triangle
A 2-dimensional polyhedron in ZZ^3 defined as the convex hull of 3 vertices
sage: triangle.affine_hull_projection()
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 3 vertices
sage: half3d = Polyhedron(vertices=[(3,2,1)], rays=[(1,0,0)])

sage: half3d.affine_hull_projection().Vrepresentation()
(A ray in the direction (1), A vertex at (3))
```

The resulting affine hulls depend on the parameter orthogonal and orthonormal:

```
sage: L = Polyhedron([(1,0),(0,1)]); L
A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: A = L.affine_hull_projection(); A
A 1-dimensional polyhedron in ZZ^1 defined as the convex hull of 2 vertices
sage: A.vertices()
(A vertex at (0), A vertex at (1))
sage: A = L.affine_hull_projection(orthogonal=True); A
A 1-dimensional polyhedron in QQ^1 defined as the convex hull of 2 vertices
sage: A.vertices()
(A vertex at (0), A vertex at (2))
sage: A = L.affine_hull_projection(orthonormal=True)
Traceback (most recent call last):
```
ValueError: the base ring needs to be extended; try with "extend=True"

sage: A = L.affine_hull_projection(orthonormal=True, extend=True); A
A 1-dimensional polyhedron in AA^1 defined as the convex hull of 2 vertices
sage: A.vertices()
(A vertex at (1.414213562373095?), A vertex at (0.?e-18))

More generally:

sage: S = polytopes.simplex(); S
A 3-dimensional polyhedron in ZZ^4 defined as the convex hull of 4 vertices
sage: S.vertices()
(A vertex at (0, 0, 0, 1),
A vertex at (0, 0, 1, 0),
A vertex at (0, 1, 0, 0),
A vertex at (1, 0, 0, 0))

sage: A = S.affine_hull_projection(); A
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 4 vertices
sage: A.vertices()
(A vertex at (0, 0, 0),
A vertex at (0, 0, 1),
A vertex at (0, 1, 0),
A vertex at (1, 0, 0))

sage: A = S.affine_hull_projection(orthogonal=True); A
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 4 vertices
sage: A.vertices()
(A vertex at (0, 0, 0),
A vertex at (2, 0, 0),
A vertex at (1, 3/2, 0),
A vertex at (1, 1/2, 4/3))

sage: A = S.affine_hull_projection(orthonormal=True, extend=True); A
A 3-dimensional polyhedron in AA^3 defined as the convex hull of 4 vertices
sage: A.vertices()
(A vertex at (0.7071067811865475?, 0.4082482904638630?, 1.154700538379252?),
A vertex at (0.7071067811865475?, 1.224744871391589?, 0.?e-18),
A vertex at (1.414213562373095?, 0.?e-18, 0.?e-18),
A vertex at (0.?e-18, 0.?e-18, 0.?e-18))

With the parameter minimal one can get a minimal base ring:

sage: s = polytopes.simplex(3)
sage: s_AA = s.affine_hull_projection(orthonormal=True, extend=True)
sage: s_AA.base_ring()
Algebraic Real Field
sage: s_full = s.affine_hull_projection(orthonormal=True, extend=True, minimal=True)
sage: s_full.base_ring()
Number Field in a with defining polynomial y^4 - 4*y^2 + 1 with a = 0.

-5176380902050415?

More examples with the orthonormal parameter:
sage: P = polytopes.permutahedron(3); P
A 2-dimensional polyhedron in ZZ^3 defined as the convex hull of 6 vertices
sage: set([F.as_polyhedron().affine_hull_projection(orthonormal=True, extend=True).volume() for F in P.affine_hull_projection().faces(1)]) == {1, sqrt(AA(2))}
True
sage: set([F.as_polyhedron().affine_hull_projection(orthonormal=True, extend=True).volume() for F in P.affine_hull_projection(orthonormal=True, extend=True).faces(1)]) == {sqrt(AA(2))}
True
sage: D = polytopes.dodecahedron()
sage: F = D.faces(2)[0].as_polyhedron()
sage: F.affine_hull_projection(orthogonal=True)
A 2-dimensional polyhedron in (Number Field in sqrt5 with defining polynomial x^2 - 5 with sqrt5 = 2.236067977499790?)^2 defined as the convex hull of 5 vertices
sage: F.affine_hull_projection(orthonormal=True, extend=True)
A 2-dimensional polyhedron in AA^2 defined as the convex hull of 5 vertices
sage: 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))
sage: A = P.affine_hull_projection(orthonormal=True); A
A 1-dimensional polyhedron in (Number Field in sqrt2 with defining polynomial x^2 - 2 with sqrt2 = 1.414213562373095?)^2 defined as the convex hull of 2 vertices
sage: A.vertices()
(A vertex at (0), A vertex at (2))
sage: K.<sqrt3> = QuadraticField(3)
sage: P = Polyhedron([2*[K.zero()],2*[sqrt3]])
sage: P.vertices()
(A vertex at (0, 0), A vertex at (sqrt3, sqrt3))
sage: A = P.affine_hull_projection(orthonormal=True)
Traceback (most recent call last):
...
ValueError: the base ring needs to be extended; try with "extend=True"
sage: A = P.affine_hull_projection(orthonormal=True, extend=True); A
A 1-dimensional polyhedron in AA^1 defined as the convex hull of 2 vertices
sage: A.vertices()
(A vertex at (0), A vertex at (2.44948974278318?))
\sqrt{6}.n()
2.44948974278318

The affine hull is combinatorially equivalent to the input:

2.5. Base classes for polyhedra 421
sage: P.is_combinatorially_isomorphic(P.affine_hull_projection())
True
sage: P.is_combinatorially_isomorphic(P.affine_hull_projection(orthogonal=True))
True
sage: P.is_combinatorially_isomorphic(P.affine_hull_projection(orthonormal=True, extend=True))
True

The orthonormal=True parameter preserves volumes; it provides an isometric copy of the polyhedron:

sage: Pentagon = polytopes.dodecahedron().faces(2)[0].as_polyhedron()
sage: P = Pentagon.affine_hull_projection(orthonormal=True, extend=True)
sage: c = P.is_inscribed(certificate=True)
sage: c
(0.4721359549995794?, 0.6498393924658126?)
sage: circumradius = (c-vector(P.vertices()[0])).norm()
sage: p = polytopes.regular_polygon(5)
sage: p.volume()
2.377641290737884?

sage: P.volume()
1.53406271079097?

sage: P.volume() == 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:

sage: Pentagon = polytopes.dodecahedron().faces(2)[0].as_polyhedron()
sage: P orthogonal = Pentagon.affine_hull_projection(orthogonal=True, as_affine_map=True, extend=True)
sage: A, b = Pentagon.affine_hull_projection(orthogonal=True, extend=True)
sage: A, b = P.affine_hull_projection(orthogonal=True)
sage: A, b = P.affine_hull_projection(orthogonal=True, as_affine_map=True, extend=True)
sage: Q = P.affine_hull_projection(orthogonal=True, extend=True)
sage: Q.center()
(0.7071067811865475?, 1.224744871391589?, 1.732050807568878?)
For unbounded, non full-dimensional polyhedra, the `orthogonal=True` and `orthonormal=True` is not implemented:

```
sage: P = Polyhedron(ieqs=[[0, 1, 0], [0, 0, 1], [0, 0, -1]]); P
A 1-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 1 ray
sage: P.is_compact()
False
sage: P.is_full_dimensional()
False
sage: P.affine_hull_projection(orthogonal=True)
Traceback (most recent call last):
... Not ImplementedError: "orthogonal=True" and "orthonormal=True" work only for compact polyhedra
sage: P.affine_hull_projection(orthonormal=True)
Traceback (most recent call last):
... Not ImplementedError: "orthogonal=True" and "orthonormal=True" work only for compact polyhedra
```

Setting `as_affine_map` to `True` without `orthogonal` or `orthonormal` set to `True`:

```
sage: S = polytopes.simplex()
sage: S.affine_hull_projection(as_affine_map=True)
(Vector space morphism represented by the matrix:
[1 0 0]
[0 1 0]
[0 0 1]
[0 0 0]
Domain: Vector space of dimension 4 over Rational Field
Codomain: Vector space of dimension 3 over Rational Field,
(0, 0, 0))
```

If the polyhedron is full-dimensional, it is returned:

```
sage: polytopes.cube().affine_hull_projection()
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 8 vertices
sage: polytopes.cube().affine_hull_projection(as_affine_map=True)
(Vector space morphism represented by the matrix:
[1 0 0]
[0 1 0]
[0 0 1]
Domain: Vector space of dimension 3 over Rational Field
Codomain: Vector space of dimension 3 over Rational Field,
(0, 0, 0))
```

Return polyhedron and affine map:

```
sage: A(P.center()) + b == Q.center()
True
```
```
sage: S = polytopes.simplex(2)
sage: data = S.affine_hull_projection(orthogonal=True,
.....: as_polyhedron=True,
.....: as_affine_map=True); data
AffineHullProjectionData(image=A 2-dimensional polyhedron in QQ^2
defined as the convex hull of 3 vertices,
projection_linear_map=Vector space morphism represented by the matrix:
  [[ -1  -1/2]
   [  1  -1/2]
   [  0   1]]
  Domain: Vector space of dimension 3 over Rational Field
  Codomain: Vector space of dimension 2 over Rational Field,
projection_translation=(1, 1/2),
section_linear_map=None,
section_translation=None)
```

Return all data:

```
sage: data = S.affine_hull_projection(orthogonal=True, return_all_data=True); data
AffineHullProjectionData(image=A 2-dimensional polyhedron in QQ^2
defined as the convex hull of 3 vertices,
projection_linear_map=Vector space morphism represented by the matrix:
  [[ -1  -1/2]
   [  1  -1/2]
   [  0   1]]
  Domain: Vector space of dimension 3 over Rational Field
  Codomain: Vector space of dimension 2 over Rational Field,
projection_translation=(1, 1/2),
section_linear_map=Vector space morphism represented by the matrix:
  [[ -1/2  1/2  0]
   [ -1/3 -1/3  2/3]]
  Domain: Vector space of dimension 2 over Rational Field
  Codomain: Vector space of dimension 3 over Rational Field,
section_translation=(1, 0, 0))
```

The section map is a right inverse of the projection map:

```
sage: data.image.linear_transformation(data.section_linear_map.matrix().
.....: transpose()) + data.section_translation == S
True
```

Same without orthogonal=True:

```
sage: data = S.affine_hull_projection(return_all_data=True); data
AffineHullProjectionData(image=A 2-dimensional polyhedron in ZZ^2
defined as the convex hull of 3 vertices,
projection_linear_map=Vector space morphism represented by the matrix:
  [[ 1  0]
   [ 0  1]
   [ 0  0]]
  Domain: Vector space of dimension 3 over Rational Field
  Codomain: Vector space of dimension 2 over Rational Field,
projection_translation=(0, 0),
section_translation=(0, 0).
```

(continues on next page)
section linear map=Vector space morphism represented by the matrix:

\[
\begin{bmatrix}
1 & 0 & -1 \\
0 & 1 & -1
\end{bmatrix}
\]

Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 3 over Rational Field, section_
→translation=(0, 0, 1))
sage: data.image.linear_transformation(data.section_linear_map.matrix().
→transpose()) + data.section_translation == S
True

sage: P0 = Polyhedron(
....: ieqs=[(0, -1, 0, 1, 1, 1), (0, 1, 1, 0, -1, -1), (0, -1, 1, 1, 0, 0),
....: (0, 1, 0, 0, 0, 0), (0, 0, 1, 1, -1, -1), (0, 0, 0, 0, 0, 1),
....: (0, 0, 0, 1, 0, -1), (0, 0, 1, 0, 0, 0)]

sage: P = P0.intersection(Polyhedron(eqns=[(-1, 1, 1, 1, 1, 1)]))
sage: P.dim()
4
sage: P.affine_hull_projection(orthogonal=True, as_affine_map=True)[0]
Vector space morphism represented by the matrix:

\[
\begin{bmatrix}
0 & 0 & 0 & 1/3 \\
-2/3 & -1/6 & 0 & -1/12 \\
1/3 & -1/6 & 1/2 & -1/12 \\
0 & 1/2 & 0 & -1/12 \\
1/3 & -1/6 & -1/2 & -1/12
\end{bmatrix}
\]

Domain: Vector space of dimension 5 over Rational Field
Codomain: Vector space of dimension 4 over Rational Field

ambient(base_field=None)

Return the ambient vector space.

It is the ambient free module (Vrepresentation_space()) tensored with a field.

INPUT:

• base_field – (default: the fraction field of the base ring) a field.

EXAMPLES:

sage: poly_test = Polyhedron(vertices = [[1,0,0,0],[0,1,0,0]])
sage: poly_test.ambient_vector_space()
Vector space of dimension 4 over Rational Field
sage: poly_test.ambient_vector_space() is poly_test.ambient()
True

sage: poly_test.ambient_vector_space(AA)
Vector space of dimension 4 over Algebraic Real Field
sage: poly_test.ambient_vector_space(RR)
Vector space of dimension 4 over Real Field with 53 bits of precision
sage: poly_test.ambient_vector_space(SR)
Vector space of dimension 4 over Symbolic Ring

ambient_dim()

Return the dimension of the ambient space.

EXAMPLES:
ambient_space()  
Return the ambient free module.

OUTPUT:  
A free module over the base ring of dimension \texttt{ambient_dim()}.  

EXAMPLES:

```python  
sage: poly_test = Polyhedron(\text{vertices} = \{[1,0,0,0],[0,1,0,0]\})  
sage: poly_test.ambient_dim()  
4  
```

ambient_vector_space(\texttt{base_field}=None)  
Return the ambient vector space.

It is the ambient free module (\texttt{Vrepresentation_space()}) tensored with a field.

INPUT:

• \texttt{base_field} – (default: the fraction field of the base ring) a field.

EXAMPLES:

```python  
sage: poly_test = Polyhedron(\text{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  
sage: poly_test.ambient_vector_space()  
Vector space of dimension 4 over Rational Field  
sage: poly_test.ambient_vector_space(AA)  
Vector space of dimension 4 over Algebraic Real Field  
sage: poly_test.ambient_vector_space(RR)  
Vector space of dimension 4 over Real Field with 53 bits of precision  
sage: poly_test.ambient_vector_space(SR)  
Vector space of dimension 4 over Symbolic Ring  
```

an_affine_basis()  
Return points in \texttt{self} that are a basis for the affine span of the polytope.

This implementation of the method \texttt{ConvexSet_base.an_affine_basis()} for polytopes guarantees the following:

• All points are vertices.

• The basis is obtained by considering a maximal chain of faces in the face lattice and picking for each cover relation one vertex that is in the difference. Thus this method is independent of the concrete realization of the polytope.

EXAMPLES:

```python  
sage: P = polytopes.cube()  
sage: P.an_affine_basis()  
(continues on next page)```
A vertex at (-1, -1, -1),
A vertex at (1, -1, -1),
A vertex at (1, -1, 1),
A vertex at (1, 1, -1)]

sage: P = polytopes.permutahedron(5)
sage: P.an_affine_basis()
[A vertex at (1, 2, 3, 5, 4),
A vertex at (2, 1, 3, 5, 4),
A vertex at (1, 3, 2, 5, 4),
A vertex at (4, 1, 3, 5, 2),
A vertex at (4, 2, 5, 3, 1)]

The method is not implemented for unbounded polyhedra:

sage: p = Polyhedron(vertices=[[0,0]],rays=[[1,0],[0,1]])
sage: p.an_affine_basis()
Traceback (most recent call last):
  ... 
NotImplementedError: this function is not implemented for unbounded polyhedra

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:

```
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(verts=[[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(verts=[[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
```

\texttt{base\_extend}(\texttt{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 \texttt{Polyhedron()}. If \texttt{None} (the default), attempt to keep the same backend. Otherwise, use the same defaulting behavior as described there.

OUTPUT:
The same polyhedron, but over a larger base ring and possibly with a changed backend.

EXAMPLES:

```
sage: P = Polyhedron(verts=[[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
```

(continues on next page)
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: 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, 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]
```

boundary_complex()

Return the simplicial complex given by the boundary faces of self, if it is simplicial.

OUTPUT:
A (spherical) simplicial complex

EXAMPLES:
The boundary complex of the octahedron:

```
sage: oc = polytopes.octahedron()
sage: sc_oc = oc.boundary_complex()
sage: fl_oc = oc.face_lattice()
sage: fl_sc = sc_oc.face_poset()
sage: [len(x) for x in fl_oc.level_sets()]
[1, 6, 12, 8, 1]
sage: [len(x) for x in fl_sc.level_sets()]
[6, 12, 8]
sage: sc_oc.euler_characteristic()
2
sage: sc_oc.homology()
{0: 0, 1: 0, 2: ℤ}
```

The polyhedron should be simplicial:

```
sage: c = polytopes.cube()
sage: c.boundary_complex()
Traceback (most recent call last):
  ... Not ImplementedError: this function is only implemented for simplicial polytopes
```

**bounded_edges()**

Return the bounded edges (excluding rays and lines).

**OUTPUT:**

A generator for pairs of vertices, one pair per edge.

**EXAMPLES:**

```
sage: p = Polyhedron(vertices=[[1,0],[0,1]], rays=[[1,0],[0,1]])
sage: [ e for e in p.bounded_edges() ]
[(A vertex at (0, 1), A vertex at (1, 0))]
sage: for e in p.bounded_edges(): print(e)
(A vertex at (0, 1), A vertex at (1, 0))
```

**bounding_box(integral=False, integral_hull=False)**

Return the coordinates of a rectangular box containing the non-empty polytope.

**INPUT:**

- **integral** – Boolean (default: False). Whether to only allow integral coordinates in the bounding box.
- **integral_hull** – Boolean (default: False). If True, return a box containing the integral points of the polytope, or None, None if it is known that the polytope has no integral points.

**OUTPUT:**

A pair of tuples (box_min, box_max) where box_min are the coordinates of a point bounding the coordinates of the polytope from below and box_max bounds the coordinates from above.

**EXAMPLES:**

```
sage: Polyhedron([[1/3,2/3], [2/3,1/3]]).bounding_box()
((1/3, 1/3), (2/3, 2/3))
```
cartesian_product(other)

Return the Cartesian product.

INPUT:

- other – a Polyhedron_base

OUTPUT:

The Cartesian product of self and other with a suitable base ring to encompass the two.

EXAMPLES:

```python
sage: P1 = Polyhedron([[0],[1]], base_ring=ZZ)
sage: P2 = Polyhedron([[0],[1]], base_ring=QQ)
sage: P1.product(P2)
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 4 vertices
```

The Cartesian product is the product in the semiring of polyhedra:

```python
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
```

An alias is cartesian_product():

```python
sage: P1.cartesian_product(P2) == P1.product(P2)
True
```

cdd_Hrepresentation()

Write the inequalities/equations data of the polyhedron in cdd’s H-representation format.

See also:

- write_cdd_Hrepresentation() – export the polyhedron as a H-representation to a file.

OUTPUT: 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
  1 0 1
  1 1 0
  1 1 1
end
```

**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)

centroid(engine='auto', **kwds)

Return the center of the mass of the polytope.

The mass is taken with respect to the induced Lebesgue measure, see volume().

If the polyhedron is not compact, a NotImplementedError is raised.

INPUT:

• engine – either ‘auto’ (default), ‘internal’, ‘TOPCOM’, or ‘normaliz’. The ‘internal’ and ‘TOPCOM’
instruct this package to always use its own triangulation algorithms or TOPCOM’s algorithms, respec-
tively. By default (‘auto’), TOPCOM is used if it is available and internal routines otherwise.

• **kwds – keyword arguments that are passed to the triangulation engine (see triangulate()).

OUTPUT: The centroid as vector.

ALGORITHM:

We triangulate the polytope and find the barycenter of the simplices. We add the individual barycenters
weighted by the fraction of the total mass.

EXAMPLES:

sage: P = polytopes.hypercube(2).pyramid()
sage: P.centroid()
(1/4, 0, 0)

sage: P = polytopes.associahedron(['A',2])
sage: P.centroid()
(2/21, 2/21)

sage: P = polytopes.permutahedron(4, backend='normaliz')  # optional - pynormaliz
                # optional - pynormaliz
sage: P.centroid()
(5/2, 5/2, 5/2, 5/2)

The method is not implemented for unbounded polyhedra:

sage: P = Polyhedron(vertices=[(0,0)],rays=[(1,0),(0,1)])
sage: P.centroid()
Traceback (most recent call last):
...  
NotImplementedError: the polyhedron is not compact

The centroid of an empty polyhedron is not defined:

sage: Polyhedron().centroid()
Traceback (most recent call last):
...  
ZeroDivisionError: rational division by zero
change_ring(base_ring, backend=None)

Return the polyhedron obtained by coercing the entries of the vertices/lines/rays of this polyhedron into the given ring.

This method can also be used to change the backend.

INPUT:

- base_ring – the new base ring
- backend – the new backend or None (default), see Polyhedron(). If None (the default), attempt to keep the same backend. Otherwise, use the same defaulting behavior as described there.

EXAMPLES:

```
sage: P = Polyhedron(vertices=[(1,0), (0,1)], rays=[(1,1)], base_ring=QQ); P
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 2 vertices and 1 ray
sage: P.change_ring(ZZ)
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices and 1 ray
sage: P.change_ring(ZZ) == P
True
sage: P = Polyhedron(vertices=[(-1.3,0), (0,2.3)], base_ring=RDF); P.vertices()
(A vertex at (-1.3, 0.0), A vertex at (0.0, 2.3))
sage: P.change_ring(QQ).vertices()
(A vertex at (-13/10, 0), A vertex at (0, 23/10))
sage: P == P.change_ring(QQ)
True
sage: P.change_ring(ZZ)
Traceback (most recent call last):
... TypeError: cannot change the base ring to the Integer Ring
sage: P = polytopes.regular_polygon(3); P
A 2-dimensional polyhedron in AA^2 defined as the convex hull of 3 vertices
sage: P.vertices()
(A vertex at (0.?e-16, 1.000000000000000?), A vertex at (0.866025403784439?, -0.500000000000000?),
A vertex at (-0.866025403784439?, -0.500000000000000?))
sage: P.change_ring(QQ)
Traceback (most recent call last):
... TypeError: cannot change the base ring to the Rational Field
```

Warning: The base ring RDF should be used with care. As it is not an exact ring, certain computations may break or silently produce wrong results, for example changing the base ring from an exact ring into RDF may cause a loss of data:

```
sage: P = Polyhedron([[2/3,0],[6666666666666667/10^16,0]], base_ring=AA); P
A 1-dimensional polyhedron in AA^2 defined as the convex hull of 2 vertices
sage: Q = P.change_ring(RDF); Q
A 0-dimensional polyhedron in RDF^2 defined as the convex hull of 1 vertex
sage: P.n_vertices() == Q.n_vertices()
False
```
**combinatorial_automorphism_group**\( (\text{vertex\_graph\_only}=\text{False}) \)

Computes the combinatorial automorphism group.

If \( \text{vertex\_graph\_only} \) is True, the automorphism group of the vertex-edge graph of the polyhedron is returned. Otherwise the automorphism group of the vertex-facet graph, which is isomorphic to the automorphism group of the face lattice is returned.

**INPUT:**

- \( \text{vertex\_graph\_only} \) – boolean (default: False); whether to return the automorphism group of the vertex edges graph or of the lattice

**OUTPUT:**

A \texttt{PermutationGroup} that is isomorphic to the combinatorial automorphism group is returned.

- if \( \text{vertex\_graph\_only} \) is True: The automorphism group of the vertex-edge graph of the polyhedron
- if \( \text{vertex\_graph\_only} \) 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.

**NOTE:**

Depending on \( \text{vertex\_graph\_only} \), this method returns groups that are not necessarily isomorphic, see the examples below.

**See also:**

\texttt{is\textunderscore\_combinatorially\textunderscore\_isomorphic(),\:\texttt{graph()},\:\texttt{vertex\textunderscore\_facet\_graph()}}

**EXAMPLES:**

```plaintext
sage: quadrangle = Polyhedron(vertices=[[0,0),(1,0),(0,1),(2,3)])
sage: quadrangle.combinatorial_automorphism_group().is_isomorphic(groups.permutation.Dihedral(4))
True
sage: quadrangle.restricted_automorphism_group()
Permutation Group with generators [(0,1)]
sage: P = Polyhedron(vertices=[[1,0], (1,1)], rays=[[1,0]])
sage: P.combinatorial_automorphism_group(vertex_graph_only=True)
Permutation Group with generators [(A vertex at (1,0),A vertex at (1,1))]
```

Permutations of the vertex graph only exchange vertices with vertices:

```plaintext
sage: Q=Polyhedron([[-123984206864/2768850730773, -101701330976/922950243591, -64154618668/2768850730773, -27484464474675/2768850730773,   ...
```

This shows an example of two polytopes whose vertex-edge graphs are isomorphic, but their face_lattices are not isomorphic:

```plaintext
sage: Q=Polyhedron([[-123984206864/2768850730773, -101701330976/922950243591, -64154618668/2768850730773, -27484464474675/2768850730773,   ...
```

(continues on next page)
...: [4674489456/83665171433, -4026061312/83665171433, 28596876672/83665171433, 
→ -78383796375/83665171433], [857794884940/98972360190089, -10910202223200/ 
→ 98972360190089, 2974263671400/98972360190089, -98320463346111/ 
→ 98972360190089])
sage: C = polytopes.cyclic_polytope(4,8)
sage: C.is_combinatorially_isomorphic(Q)
False
sage: C.combinatorial_automorphism_group(vertex_graph_only=True).is_isomorphic(Q.combinatorial_automorphism_group(vertex_graph_only=True))
True
sage: C.combinatorial_automorphism_group(vertex_graph_only=False).is_isomorphic(Q.combinatorial_automorphism_group(vertex_graph_only=False))
False
The automorphism group of the face lattice is isomorphic to the combinatorial automorphism group:

sage: CG = C.hasse_diagram().automorphism_group()
sage: C.combinatorial_automorphism_group().is_isomorphic(CG)
True
sage: QG = Q.hasse_diagram().automorphism_group()
sage: Q.combinatorial_automorphism_group().is_isomorphic(QG)
True

combinatorial_polyhedron()

Return the combinatorial type of self.

See also: sage.geometry.polyhedron.combinatorial_polyhedron.base.CombinatorialPolyhedron.

EXAMPLES:

sage: polytopes.cube().combinatorial_polyhedron()
A 3-dimensional combinatorial polyhedron with 6 facets
sage: polytopes.cyclic_polytope(4,10).combinatorial_polyhedron()
A 4-dimensional combinatorial polyhedron with 35 facets
sage: Polyhedron(rays=[[0,1], [1,0]]).combinatorial_polyhedron()
A 2-dimensional combinatorial polyhedron with 2 facets

contains(point)

Test whether the polyhedron contains the given point.

See also: interior_contains(), relative_interior_contains().

INPUT:

• point – coordinates of a point (an iterable)

OUTPUT:

Boolean.

EXAMPLES:
sage: P = Polyhedron(vertices=[[1,1],[1,-1],[0,0]])
sage: P.contains([1,0])
True
sage: P.contains(P.center())  # true for any convex set
True

As a shorthand, one may use the usual in operator:

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:

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:

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(vertices = 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 base_ring()

OUTPUT:
The polyhedron dilated by that scalar, possibly coerced to a bigger base ring.

EXAMPLES:

sage: p = Polyhedron(vertices = [[t,t^2,t^3] for t in srange(2,6)])
sage: next(p.vertex_generator())
A vertex at (2, 4, 8)
sage: p2 = p.dilation(2)
sage: next(p2.vertex_generator())
A vertex at (4, 8, 16)
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(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):

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 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:**

```
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):

```
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:**

```
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:**

```
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,)
```

**equations_list()**
Return the linear constraints of the polyhedron. As with inequalities, each constraint is given as \([b -a_1 -a_2 \ldots -a_n]\) where for variables \(x_1, x_2, \ldots, x_n\), the polyhedron satisfies the equation \(b = a_1 x_1 + a_2 x_2 + \ldots + a_n x_n\).

Note: It is recommended to use **equations()** or **equation_generator()** instead to iterate over the list of **Equation** objects.

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_list()
[[-10, 1, 1, 1]]
```

**f_vector**(num_threads=None, parallelization_depth=None)
Return the f-vector.

INPUT:
- **num_threads** – integer (optional); specify the number of threads; otherwise determined by **ncpus()**
- **parallelization_depth** – integer (optional); specify how deep in the lattice the parallelization is done

OUTPUT:
Return a vector whose \(i\)-th entry is the number of \(i - 2\)-dimensional faces of the polytope.

Note: The vertices as given by **Polyhedron_base.vertices()** do not need to correspond to 0-dimensional faces. If a polyhedron contains \(k\) lines they correspond to \(k\)-dimensional faces. See example below

EXAMPLES:

```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)

sage: polytopes.cyclic_polytope(4,10).f_vector()
(1, 10, 45, 70, 35, 1)

sage: polytopes.hypercube(5).f_vector()
(1, 32, 80, 80, 40, 10, 1)
```

Polyhedra with lines do not have 0-faces:
However, the method `Polyhedron_base.vertices()` returns two points that belong to the Vrepresentation:

```python
sage: P = Polyhedron(ieqs=[[1,-1,0],[1,1,0]])
sage: P.vertices()
(A vertex at (1, 0), A vertex at (-1, 0))
sage: P.f_vector()
(1, 0, 2, 1)
```

### `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:

```python
sage: P = Polyhedron(ieqs=[[1,0],[1,1]])
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(ieqs=[[-1,0,1],[1,0,1]])
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:

```python
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].
If dimension is not specified return an iterator over all faces.

**INPUT:**
- `face_dimension` – integer (default `None`), yield only faces of this dimension if specified
- `dual` – boolean (default `None`); if `True`, generate the faces using the vertices; if `False`, generate the faces using the facets; if `None`, pick automatically

**OUTPUT:**
A `FaceIterator_geom`. This class iterates over faces as `PolyhedronFace`. See `face` for details. The order is random but fixed.

**EXAMPLES:**

```python
sage: P = polytopes.cube()
sage: it = P.face_generator()
sage: it
Iterator over the faces of a 3-dimensional polyhedron in ZZ^3
sage: list(it)
[442 Chapter 2. Polyhedral computations]
A 1-dimensional face of a Polyhedron in $\mathbb{Z}^3$ defined as the convex hull of 2 → vertices,
A 0-dimensional face of a Polyhedron in $\mathbb{Z}^3$ defined as the convex hull of 1 → vertex,
A 0-dimensional face of a Polyhedron in $\mathbb{Z}^3$ defined as the convex hull of 1 → vertex,
A 1-dimensional face of a Polyhedron in $\mathbb{Z}^3$ defined as the convex hull of 2 → vertices,
A 1-dimensional face of a Polyhedron in $\mathbb{Z}^3$ defined as the convex hull of 2 → vertices,
A 0-dimensional face of a Polyhedron in $\mathbb{Z}^3$ defined as the convex hull of 1 → vertex,
A 1-dimensional face of a Polyhedron in $\mathbb{Z}^3$ defined as the convex hull of 2 → vertices,
A 1-dimensional face of a Polyhedron in $\mathbb{Z}^3$ defined as the convex hull of 2 → vertices,
A 0-dimensional face of a Polyhedron in $\mathbb{Z}^3$ defined as the convex hull of 1 → vertex,
A 1-dimensional face of a Polyhedron in $\mathbb{Z}^3$ defined as the convex hull of 2 → vertices,
A 1-dimensional face of a Polyhedron in $\mathbb{Z}^3$ defined as the convex hull of 2 → vertices,
A 0-dimensional face of a Polyhedron in $\mathbb{Z}^3$ defined as the convex hull of 1 → vertex,
A 1-dimensional face of a Polyhedron in $\mathbb{Z}^3$ defined as the convex hull of 2 → vertices,
A 1-dimensional face of a Polyhedron in $\mathbb{Z}^3$ defined as the convex hull of 2 → vertices,
A 1-dimensional face of a Polyhedron in $\mathbb{Z}^3$ defined as the convex hull of 2 → vertices

sage: P = polytopes.hypercube(4)
sage: list(P.face_generator(2))[:4]
[A 2-dimensional face of a Polyhedron in $\mathbb{Z}^4$ defined as the convex hull of 4 → vertices,
A 2-dimensional face of a Polyhedron in $\mathbb{Z}^4$ defined as the convex hull of 4 → vertices,
A 2-dimensional face of a Polyhedron in $\mathbb{Z}^4$ defined as the convex hull of 4 → vertices,
A 2-dimensional face of a Polyhedron in $\mathbb{Z}^4$ defined as the convex hull of 4 → vertices]

sage: P = polytopes.hypercube(4)
sage: list(P.face_generator(2))[:4]

If a polytope has more facets than vertices, the dual mode is chosen:

sage: P = polytopes.cross_polytope(3)
sage: list(P.face_generator())
[A 3-dimensional face of a Polyhedron in $\mathbb{Z}^3$ defined as the convex hull of 6 → vertices,
A -1-dimensional face of a Polyhedron in $\mathbb{Z}^3$,
A 0-dimensional face of a Polyhedron in $\mathbb{Z}^3$ defined as the convex hull of 1 → vertex,
A 0-dimensional face of a Polyhedron in $\mathbb{Z}^3$ defined as the convex hull of 1 → vertex,
A 0-dimensional face of a Polyhedron in $\mathbb{Z}^3$ defined as the convex hull of 1 → vertex,
A 0-dimensional face of a Polyhedron in $\mathbb{Z}^3$ defined as the convex hull of 1 → vertex,
A 0-dimensional face of a Polyhedron in $\mathbb{Z}^3$ defined as the convex hull of 1 → vertex,
A 0-dimensional face of a Polyhedron in $\mathbb{Z}^3$ defined as the convex hull of 1 → vertex,
A 0-dimensional face of a Polyhedron in $\mathbb{Z}^3$ defined as the convex hull of 1 → vertex,
A 1-dimensional face of a Polyhedron in $\mathbb{Z}^3$ defined as the convex hull of 2 → vertices,
A 1-dimensional face of a Polyhedron in $\mathbb{Z}^3$ defined as the convex hull of 2 → vertices,
A 1-dimensional face of a Polyhedron in $\mathbb{Z}^3$ defined as the convex hull of 2 → vertices,
A 1-dimensional face of a Polyhedron in $\mathbb{Z}^3$ defined as the convex hull of 2 → vertices,

sage: P = polytopes.cross_polytope(3)
sage: list(P.face_generator())

(continues on next page)
A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2 vertices,
A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2 vertices,
A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2 vertices,
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3 vertices,
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3 vertices,
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3 vertices,
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3 vertices,
A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2 vertices,
A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2 vertices,
A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2 vertices,
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3 vertices,
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3 vertices,
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3 vertices,
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3 vertices,
A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2 vertices,
A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2 vertices,
A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2 vertices,
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3 vertices,
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3 vertices,
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3 vertices,
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3 vertices,
A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2 vertices,
A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2 vertices,
A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2 vertices,
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3 vertices,
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3 vertices,
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3 vertices,
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3 vertices,
A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2 vertices,
A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2 vertices,
A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2 vertices,
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3 vertices,
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3 vertices,
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3 vertices,
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 3 vertices,

The face iterator can also be slightly modified. In non-dual mode we can skip subfaces of the current (proper) face:

```python
sage: P = polytopes.cube()
sage: it = P.face_generator(dual=False)
sage: _ = next(it), next(it)
sage: face = next(it)
sage: face.ambient_H_indices()
(5,)
sage: it.ignore_subfaces()
sage: face = next(it)
sage: face.ambient_H_indices()
(4,)
```
sage: it.ignore_subfaces()
sage: [face.ambient_H_indices() for face in it]
[(3,),
 (2,),
 (1,),
 (0,),
 (2, 3),
 (1, 3),
 (1, 2, 3),
 (1, 2),
 (0, 2),
 (0, 1, 2),
 (0, 1)]

In dual mode we can skip supfaces of the current (proper) face:

sage: P = polytopes.cube()
sage: it = P.face_generator(dual=True)
sage: _, next(it), next(it)
sage: face = next(it)
sage: face.ambient_V_indices()
(7,)

sage: it.ignore_supfaces()
sage: next(it)
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1 vertex

sage: face = next(it)
sage: face.ambient_V_indices()
(5,)

sage: it.ignore_supfaces()
sage: [face.ambient_V_indices() for face in it]
[(4,),
 (3,),
 (2,),
 (1,),
 (0,),
 (1, 6),
 (3, 4),
 (2, 3),
 (0, 3),
 (0, 1, 2, 3),
 (1, 2),
 (0, 1)]

In non-dual mode, we cannot skip supfaces:

sage: it = P.face_generator(dual=False)
sage: _, next(it), next(it)
sage: next(it)
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4 vertices

sage: it.ignore_supfaces()
Traceback (most recent call last):
...
ValueError: only possible when in dual mode

In dual mode, we cannot skip subfaces:

```
sage: it = P.face_generator(dual=True)
sage: _, next(it), next(it)
sage: next(it)
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1 vertex
sage: it.ignore_subfaces()
Traceback (most recent call last):
...
ValueError: only possible when not in dual mode
```

We can only skip sub-/supfaces of proper faces:

```
sage: it = P.face_generator(dual=False)
sage: next(it)
A 3-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 8 vertices
sage: it.ignore_subfaces()
Traceback (most recent call last):
...
ValueError: iterator not set to a face yet
```

See also:
FaceIterator_geom.

ALGORITHM:
See FaceIterator.

`face_lattice()`

Return the face-lattice poset.

OUTPUT:

A `FinitePoset`. Elements are given as `PolyhedronFace`.

In the case of a full-dimensional polytope, the faces are pairs (vertices, inequalities) of the spanning vertices and corresponding saturated inequalities. In general, a face is defined by a pair (V-rep. objects, H-rep. objects). The V-representation objects span the face, and the corresponding H-representation objects are those inequalities and equations that are saturated on the face.

The bottom-most element of the face lattice is the “empty face”. It contains no V-representation object. All H-representation objects are incident.

The top-most element is the “full face”. It is spanned by all V-representation objects. The incident H-representation objects are all equations and no inequalities.

In the case of a full-dimensional polytope, the “empty face” and the “full face” are the empty set (no vertices, all inequalities) and the full polytope (all vertices, no inequalities), respectively.

ALGORITHM:
See `sage.geometry.polyhedron.combinatorial_polyhedron.face_iterator`.
EXAMPLES:

```
sage: square = polytopes.hypercube(2)
sage: fl = square.face_lattice();fl
Finite lattice containing 10 elements
sage: list(f.ambient_V_indices() for f in fl)
[(0,), (0, 1), (1,), (2,), (0, 1, 2, 3), (3,), (0, 3), (2, 3), (1, 2), (1, 3)]
sage: poset_element = fl[5]
sage: a_face = poset_element
sage: a_face
A 1-dimensional face of a Polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: a_face.ambient_V_indices()
(1, 2)
sage: set(a_face.ambient_Vrepresentation()) == set([square.Vrepresentation(1), square.Vrepresentation(2)])
True
sage: a_face.ambient_Vrepresentation()
(A vertex at (1, 1), A vertex at (-1, 1))
sage: a_face.ambient_Hrepresentation()
(An inequality (0, -1) x + 1 >= 0,)
```

A more complicated example:

```
sage: c5_10 = Polyhedron(vertices = [[i,i^2,i^3,i^4,i^5] for i in range(1,11)])
sage: c5_10_fl = c5_10.face_lattice()
sage: [len(x) for x in c5_10_fl.level_sets()]
[1, 10, 45, 100, 105, 42, 1]
```

Note that if the polyhedron contains lines then there is a dimension gap between the empty face and the first non-empty face in the face lattice:

```
sage: line = Polyhedron(vertices=[[0,],], lines=[[1,]])
sage: [ fl.dim() for fl in line.face_lattice() ]
[-1, 1]
```

```
face_split(face)

Return the face splitting of the face face.

Splitting a face correspond to the bipyramid (see bipyramid()) of self where the two new vertices are placed above and below the center of face instead of the center of the whole polyhedron. The two new vertices are placed in the new dimension at height $-1$ and $1$.

INPUT:

* face – a PolyhedronFace or a Vertex

EXAMPLES:

```
sage: pentagon = polytopes.regular_polygon(5)
sage: f = pentagon.faces(1)[0]
sage: fsplit_pentagon = pentagon.face_split(f)
```
```
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: tuple(f.ambient_V_indices() for f in vertex_trunc1.faces(2))
((4, 5, 6, 7, 9),
 (0, 3, 4, 8, 9),
 (0, 1, 6, 7, 8),
 (7, 8, 9),
 (2, 3, 4, 5),
 (1, 2, 5, 6),
 (0, 1, 2, 3))
sage: vertex_trunc1.vertices()
(A vertex at (1, -1, -1),
 A vertex at (1, 1, -1),
 A vertex at (1, 1, 1),
 A vertex at (1, -1, 1),
 A vertex at (-1, -1, 1),
 A vertex at (-1, 1, 1),
 A vertex at (-1, -1, -1),
 A vertex at (-1, 1, -1),
 A vertex at (-1, -1/3, -1),
(continues on next page)
```
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)
vertex_trunc2.f_vector()
(1, 10, 15, 7, 1)
tuple(f.ambient_V_indices() for f in vertex_trunc2.faces(2))
((4, 5, 6, 7, 9),
 (0, 3, 4, 8, 9),
 (0, 1, 6, 7, 8),
 (7, 8, 9),
 (2, 3, 4, 5),
 (1, 2, 5, 6),
 (0, 1, 2, 3))
```

```sage```
vertex_trunc3 = Cube.face_truncation(Cube.faces(0)[0], cut_frac=0.3)
vertex_trunc3.vertices()
(A vertex at (-1.0, -1.0, 1.0),
 A vertex at (-1.0, 1.0, -1.0),
 A vertex at (-1.0, 1.0, 1.0),
 A vertex at (1.0, 1.0, -1.0),
 A vertex at (1.0, 1.0, 1.0),
 A vertex at (1.0, -1.0, 1.0),
 A vertex at (1.0, -1.0, -1.0),
 A vertex at (-0.4, -1.0, -1.0),
 A vertex at (-1.0, -0.4, -1.0),
 A vertex at (-1.0, -1.0, -0.4))
```

```sage```
edge_trunc = Cube.face_truncation(Cube.faces(1)[11])
edge_trunc.f_vector()
(1, 10, 15, 7, 1)
tuple(f.ambient_V_indices() for f in edge_trunc.faces(2))
((0, 5, 6, 7),
 (1, 4, 5, 6, 8),
 (6, 7, 8, 9),
 (0, 2, 3, 7, 9),
 (1, 2, 8, 9),
 (0, 3, 4, 5),
 (1, 2, 3, 4))
```

```sage```
face_trunc = Cube.face_truncation(Cube.faces(2)[2])
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
```

**faces**(face_dimension)

Return the faces of given dimension

**INPUT:**

- face_dimension – integer. The dimension of the faces whose representation will be returned.

**OUTPUT:**

A tuple of PolyhedronFace. See face for details. The order is random but fixed.

**See also:**

face_generator(), facet().

**EXAMPLES:**

Here we find the vertex and face indices of the eight three-dimensional facets of the four-dimensional hypercube:

```
sage: p = polytopes.hypercube(4)
sage: list(f.ambient_V_indices() for f in p.faces(3))
[(0, 5, 6, 7, 8, 9, 14, 15),
 (1, 4, 5, 6, 10, 13, 14, 15),
 (1, 2, 6, 7, 8, 10, 11, 15),
 (8, 9, 10, 11, 12, 13, 14, 15),
 (0, 3, 4, 5, 9, 12, 13, 14),
 (0, 2, 3, 7, 8, 9, 11, 12),
 (1, 2, 3, 4, 10, 11, 12, 13),
 (0, 1, 2, 3, 4, 5, 6, 7)]
```

```
sage: face = p.faces(3)[3]
sage: face.ambient_Hrepresentation()
(An inequality (1, 0, 0, 0) \text{x} + 1 \geq 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):
\quad return rep.index()
sage: [get_idx(_) for _ in face.ambient_Hrepresentation()]
```

(continues on next page)
sage: [get_idx(_) for _ in face.ambient_Vrepresentation()]
[8, 9, 10, 11, 12, 13, 14, 15]

sage: [(get_idx(_) for _ in face.ambient_Vrepresentation()), 
....:  (get_idx(_) for _ in face.ambient_Hrepresentation()) 
....:  for face in p.faces(3) ]

[(
[0, 5, 6, 7, 8, 9, 14, 15], [7]),
([1, 4, 5, 6, 10, 13, 14, 15], [6]),
([1, 2, 6, 7, 8, 10, 11, 15], [5]),
([8, 9, 10, 11, 12, 13, 14, 15], [4]),
([0, 3, 4, 5, 9, 12, 13, 14], [3]),
([0, 2, 3, 7, 8, 9, 11, 12], [2]),
([1, 2, 3, 4, 10, 11, 12, 13], [1]),
([0, 1, 2, 3, 4, 5, 6, 7], [0])]  

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]

The facet adjacency matrix has base ring integers. This way one can express various counting questions:

sage: P = polytopes.cube()
sage: Q = P.stack(P.faces(2)[0])
sage: M = Q.facet_adjacency_matrix()
sage: sum(M)
(4, 4, 4, 4, 3, 3, 3, 3, 4)

facets()  
Return the facets of the polyhedron.

Facets are the maximal nontrivial faces of polyhedra. The empty face and the polyhedron itself are trivial.

A facet of a $d$-dimensional polyhedron is a face of dimension $d - 1$. For $d \neq 0$ the converse is true as well.

OUTPUT:

A tuple of PolyhedronFace. See face for details. The order is random but fixed.

See also:

facets()

EXAMPLES:

Here we find the eight three-dimensional facets of the four-dimensional hypercube:
sage: p = polytopes.hypercube(4)
sage: p.facets()
(A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8→vertices,
A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8→vertices,
A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8→vertices,
A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8→vertices,
A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8→vertices,
A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8→vertices,
A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8→vertices)

This is the same result as explicitly finding the three-dimensional faces:

sage: dim = p.dimension()
sage: p.faces(dim-1)
(A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8→vertices,
A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8→vertices,
A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8→vertices,
A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8→vertices,
A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8→vertices,
A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8→vertices,
A 3-dimensional face of a Polyhedron in ZZ^4 defined as the convex hull of 8→vertices)

The 0-dimensional polyhedron does not have facets:

sage: P = Polyhedron([[0]])
sage: P.facets()
()

`flag_f_vector(*args)`

Return the flag f-vector.

For each $-1 < i_0 < \cdots < i_n < d$ the flag f-vector counts the number of flags $F_0 \subset \cdots \subset F_n$ with $F_j$ of dimension $i_j$ for each $0 \leq j \leq n$, where $d$ is the dimension of the polyhedron.

INPUT:

- `args` – integers (optional): specify an entry of the flag-f-vector; must be an increasing sequence of
OUTPUT:

- a dictionary, if no arguments were given
- an Integer, if arguments were given

EXAMPLES:

Obtain the entire flag-f-vector:

```
sage: P = polytopes.twenty_four_cell()
sage: P.flag_f_vector()
{(-1,): 1,
 (0,): 24,
 (0, 1): 192,
 (0, 1, 2): 576,
 (0, 1, 2, 3): 1152,
 (0, 1, 3): 576,
 (0, 2): 288,
 (0, 2, 3): 576,
 (0, 3): 144,
 (1,): 96,
 (1, 2): 288,
 (1, 2, 3): 576,
 (1, 3): 288,
 (2,): 96,
 (2, 3): 192,
 (3,): 24,
 (4,): 1}
```

Specify an entry:

```
sage: P.flag_f_vector(0,3)
144
sage: P.flag_f_vector(2)
96
```

Leading -1 and trailing entry of dimension are allowed:

```
sage: P.flag_f_vector(-1,0,3)
144
sage: P.flag_f_vector(-1,0,3,4)
144
```

One can get the number of trivial faces:

```
sage: P.flag_f_vector(-1)
1
sage: P.flag_f_vector(4)
1
```

Polyhedra with lines, have 0 entries accordingly:

```
sage: P = (Polyhedron(lines=[[1]]) * polytopes.cross_polytope(3))
sage: P.flag_f_vector()
```

(continues on next page)
{(-1,): 1, 
  (0, 1): 0, 
  (0, 1, 2): 0, 
  (0, 1, 3): 0, 
  (0, 2): 0, 
  (0, 2, 3): 0, 
  (0, 3): 0, 
  (0,): 0, 
  (1, 2): 24, 
  (1, 2, 3): 48, 
  (1, 3): 24, 
  (1,): 6, 
  (2, 3): 24, 
  (2,): 12, 
  (3,): 8, 
  4: 1}

If the arguments are not strictly increasing or out of range, a key error is raised:

```
sage: P.flag_f_vector(-1,0,3,6)
Traceback (most recent call last):
  ...
KeyError: (0, 3, 6)
sage: P.flag_f_vector(-1,3,0)
Traceback (most recent call last):
  ...
KeyError: (3, 0)
```

gale_transform()

Return the Gale transform of a polytope as described in the reference below.

OUTPUT:

A list of vectors, the Gale transform. The dimension is the dimension of the affine dependencies of the vertices of the polytope.

EXAMPLES:

This is from the reference, for a triangular prism:

```
sage: p = Polyhedron(vertices = [[0,0],[0,1],[1,0]])
sage: p2 = p.prism()
sage: p2.gale_transform()
((-1, 0), (0, -1), (1, 1), (-1, -1), (1, 0), (0, 1))
```

REFERENCES:


See also:

:func:`sage.geometry.polyhedron.library.gale_transform_to_polyhedron`.

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 \texttt{ValueError} is raised.

INPUT:

- index – integer. The index of the integral point to be found. If this is not in $[0, \text{self}.\text{integral_point_count()}$, an \texttt{IndexError} is raised.
- **kwds – optional keyword parameters that are passed to \texttt{self.integral_points_count()}.

ALGORITHM:
The function computes each of the components of the requested point in turn. To compute $x_i$, the $i$th component, it bisects the upper and lower bounds on $x_i$ given by the bounding box. At each bisection, it uses \texttt{integral_points_count()} to determine on which side of the bisecting hyperplane the requested point lies.

See also:

\texttt{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: ...

sage: Q = Polyhedron([(1,3), (2, 7), (9, 77)])
sage: [Q.get_integral_point(i) for i in range(Q.integral_points_count())] == sorted(Q.integral_points())
True
sage: Q.get_integral_point(0, explicit_enumeration_threshold=0, triangulation='cddlib')  # optional - latte_int
(1, 3)
sage: Q.get_integral_point(0, explicit_enumeration_threshold=0, triangulation='cddlib', foo=True)  # optional - latte_int
Traceback (most recent call last):
  ... RuntimeError: ...

sage: R = Polyhedron(vertices=[[1/2, 1/3]], rays=[[1, 1]])
sage: R.get_integral_point(0)
Traceback (most recent call last):
  ... ValueError: ...
```

\texttt{graph()}

Return a graph in which the vertices correspond to vertices of the polyhedron, and edges to edges.

..NOTE:
The graph of a polyhedron with lines has no vertices, as the polyhedron has no vertices (`0`-faces).

The method :meth:`Polyhedron_base:vertices` returns the defining points in this case.

EXAMPLES:

```
sage: g3 = polytopes.hypercube(3).vertex_graph(); g3
Graph on 8 vertices
sage: g3.automorphism_group().cardinality()
48
sage: s4 = polytopes.simplex(4).vertex_graph(); s4
Graph on 5 vertices
sage: s4.is_eulerian()
True
```

The graph of an unbounded polyhedron is the graph of the bounded complex:

```
sage: open_triangle = Polyhedron(vertices=[[1,0], [0,1]],
  ....:    rays =[[1,1]])
```

```
sage: open_triangle.vertex_graph()
Graph on 2 vertices
```

The graph of a polyhedron with lines has no vertices:

```
sage: line = Polyhedron(lines=[[0,1]])
```

```
sage: line.vertex_graph()
Graph on 0 vertices
```

`greatest_common_subface_of_Hrep(*Hrepresentatives)`

Return the largest face that is contained in `Hrepresentatives`.

INPUT:

- `Hrepresentatives` – facets or indices of `Hrepresentatives`; the indices are assumed to be the indices of the `Hrepresentation`

OUTPUT: a `PolyhedronFace`

EXAMPLES:

```
sage: P = polytopes.permutahedron(5)
sage: P.meet_of_Hrep()
A 4-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 120 vertices
```

```
sage: P.meet_of_Hrep(1)
A 3-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 24 vertices
```

```
sage: P.meet_of_Hrep(4)
A 3-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 12 vertices
```

```
sage: P.meet_of_Hrep(1,3,7)
A 1-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 2 vertices
```

(continues on next page)
sage: P.meet_of_Hrep(1,3,7).ambient_H_indices()
(0, 1, 3, 7)

The indices are the indices of the Hrepresentation. 0 corresponds to an equation and is ignored:

sage: P.meet_of_Hrep(0)
A 4-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 120 vertices

The input is flexible:

sage: P.meet_of_Hrep(P.facets()[-1], P.inequalities()[2], 7)
A 1-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 2 vertices

The Hrepresentatives must belong to self:

sage: P = polytopes.cube(backend='ppl')
sage: Q = polytopes.cube(backend='field')
sage: f = P.facets()[0]
sage: P.meet_of_Hrep(f)
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4 vertices
sage: Q.meet_of_Hrep(f)
Traceback (most recent call last):
  ...
ValueError: not a facet of ``self``
sage: f = P.inequalities()[0]
sage: P.meet_of_Hrep(f)
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4 vertices
sage: Q.meet_of_Hrep(f)
Traceback (most recent call last):
  ...
ValueError: not a facet of ``self``

**h_star_vector()**

Return the $h^*$-vector of the lattice polytope.

The $h^*$-vector records the coefficients of the polynomial in the numerator of the Ehrhart series of a lattice polytope.

**INPUT:**

- **self** – A lattice polytope.

**OUTPUT:**

A list whose entries give the $h^*$-vector.

**EXAMPLES:**

The $h^*$-vector of a unimodular simplex $S$ (a simplex with volume $\frac{1}{\text{dim}(S)!}$) is always 1. Here we test this on simplices up to dimension 3:
For a less trivial example, we compute the $h^*$-vector of the 0/1-cube, which has the Eulerian numbers $(3, i)$ for $i \in [0, 2]$ as an $h^*$-vector:

```
sage: cube = polytopes.cube(intervals='zero_one', backend='normaliz')  # optional - pynormaliz
sage: cube.h_star_vector()  # optional - pynormaliz
[1, 4, 1]
sage: from sage.combinat.combinat import eulerian_number
sage: [eulerian_number(3,i) for i in range(3)]
[1, 4, 1]
```

`hasse_diagram()`

Return the Hasse diagram of the face lattice of `self`.

This is the Hasse diagram of the poset of the faces of `self`.

OUTPUT: a directed graph

EXAMPLES:

```
sage: P = polytopes.regular_polygon(4).pyramid()
sage: D = P.hasse_diagram(); D
Digraph on 20 vertices
sage: D.degree_polynomial()
x^5 + x^4*y + x*y^4 + y^5 + 4*x^3*y + 8*x^2*y^2 + 4*x*y^3
```

Faces of an mutable polyhedron are not hashable. Hence those are not suitable as vertices of the hasse diagram. Use the combinatorial polyhedron instead:

```
sage: P = polytopes.regular_polygon(4).pyramid()
sage: parent = P.parent()
sage: parent = parent.change_ring(QQ, backend='ppl')
sage: Q = parent._element_constructor_(P, mutable=True)
sage: Q.hasse_diagram()
Traceback (most recent call last):
  ... TypeError: mutable polyhedra are unhashable
sage: C = Q.combinatorial_polyhedron()
sage: D = C.hasse_diagram()
sage: set(D.vertices()) == set(range(20))
True
sage: def index_to_combinatorial_face(n):
    ....:     return C.face_by_face_lattice_index(n)
    ....:     return
sage: D.relabel(index_to_combinatorial_face, inplace=True)
sage: D.vertices()
```

(continues on next page)
hyperplane_arrangement()

Return the hyperplane arrangement defined by the equations and inequalities.

OUTPUT:

A hyperplane arrangement consisting of the hyperplanes defined by the Hrepresentation(). If the polytope is full-dimensional, this is the hyperplane arrangement spanned by the facets of the polyhedron.

EXAMPLES:

```
sage: p = polytopes.hypercube(2)
sage: p.hyperplane_arrangement()
Arrangement <-t0 + 1 | -t1 + 1 | t1 + 1 | t0 + 1>
```

incidence_matrix()

Return the incidence matrix.

Note: The columns correspond to inequalities/equations in the order Hrepresentation(), the rows correspond to vertices/rays/lines in the order Vrepresentation().

See also:

slack_matrix().

EXAMPLES:

```
sage: p = polytopes.cuboctahedron()
sage: p.incidence_matrix()
[0 0 1 1 0 1 0 0 0 0 1 0 0 0]
[0 0 1 0 0 1 0 0 0 0 1 0 0 0]
[0 0 1 1 0 0 1 0 0 0 0 1 0 0]
```

(continued from previous page)
sage: v = p.Vrepresentation(0)
sage: v
A vertex at (-1, -1, 0)
sage: h = p.Hrepresentation(2)
sage: h
An inequality (1, 1, -1) x + 2 >= 0
sage: h.eval(v)  # evaluation (1, 1, -1) * (-1/2, -1/2, 0) + 1
0
sage: h*v  # same as h.eval(v)
0
sage: p.incidence_matrix() [0,2]  # this entry is (v,h)
1
sage: h.contains(v)
True
sage: p.incidence_matrix() [2,0]  # note: not symmetric
0

The incidence matrix depends on the ambient dimension:

sage: simplex = polytopes.simplex(); simplex
A 3-dimensional polyhedron in ZZ^4 defined as the convex hull of 4 vertices
sage: simplex.incidence_matrix()
[1 1 1 0]
[1 1 0 1]
[1 0 1 1]
[1 0 1 1]
sage: simplex = simplex.affine_hull_projection(); simplex
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 4 vertices
sage: simplex.incidence_matrix()
[1 1 0]
[1 0 1]
[0 1 1]
[0 1 1]

An incidence matrix does not determine a unique polyhedron:

sage: P = Polyhedron(vertices=[[0,1],[1,1],[1,0]])
sage: P.incidence_matrix()
[1 1 0]
[1 0 1]
[0 1 1]
sage: Q = Polyhedron(vertices=[[0,1], [1,0]], rays=[[1,1]])
An example of two polyhedra with isomorphic face lattices but different incidence matrices:

```python
sage: Q.incidence_matrix()
[1 1 0]
[1 0 1]
[0 1 1]
sage: R = Polyhedron(vertices=[[0,1], [1,0]], rays=[[1,3/2], [3/2,1]])
sage: R.incidence_matrix()
[1 1 0]
[1 0 1]
[0 1 0]
[0 0 1]
```

The incidence matrix has base ring integers. This way one can express various counting questions:

```python
sage: P = polytopes.twenty_four_cell()
sage: M = P.incidence_matrix()
sage: sum(sum(x) for x in M) == P.flag_f_vector(0,3)
True
```

### 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(vertiecs = Permutations([1,2,3,4]))
sage: ieqs = p3.inequalities()
sage: list(_)
[-6, 0, 1, 1, 1]
```

### inequalities_list()
Return a list of inequalities as coefficient lists.

**Note:** It is recommended to use `inequalities()` or `inequality_generator()` instead to iterate over the list of Inequality objects.

**EXAMPLES:**

2.5. Base classes for polyhedra
```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]
[[0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]]
sage: p3 = Polyhedron(inequalities = Permutations([1,2,3,4]))
sage: ieqs = p3.inequalities_list()
sage: ieqs[0]
[-6, 0, 1, 1]
sage: ieqs[-1]
[-3, 0, 1, 0]
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:

```python
sage: triangle = Polyhedron(inequalities=[[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:

```python
sage: p = Polyhedron(inequalities=[(-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:
Here is a simplex where the naive algorithm of running over all points in a rectangular bounding box no longer works fast enough:

```
sage: v = [(1,0,7,-1), (-2,-2,4,-3), (-1,-1,-1,4), (2,9,0,-5), (-2,-1,5,1)]
sage: simplex = Polyhedron(v); simplex
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 5 vertices
sage: len(simplex.integral_points())
49
```

A case where rounding in the right direction goes a long way:

```
sage: P = 1/10*polytopes.hypercube(14, backend='field')
sage: P.integral_points()
((0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0),)
```

Finally, the 3-d reflexive polytope number 4078:

```
sage: v = [(1,0,0), (0,1,0), (0,0,1), (0,0,-1), (0,-2,1), (-1,2,-1), (-1,2,-2), (-1,1,-2), (-1,-1,2), (-1,-3,2)]
sage: P = Polyhedron(v)
sage: pts1 = P.integral_points()  # Sage's own code
sage: all(P.contains(p) for p in pts1)
True
sage: pts2 = LatticePolytope(v).points()  # PALP
sage: for p in pts1: p.set_immutable()
```

```
integral_points_count(**kwds)
Return the number of integral points in the polyhedron.
This generic version of this method simply calls integral_points.
```

```
sage: P = polytopes.cube()
sage: P.integral_points_count()
27
```

2.5. Base classes for polyhedra 463
We shrink the polyhedron a little bit:

```plaintext
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:

```plaintext
sage: Q = P*RDF(8/9)
sage: Q.integral_points_count()
1
```

Unbounded polyhedra (with or without lattice points) are not supported:

```plaintext
sage: P = Polyhedron(ambient_dim=2, 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**(`function, measure='ambient', **kwds)``

Return the integral of `function` over this polytope.

**INPUT:**

- `self` – Polyhedron
- `function` – a multivariate polynomial or a valid LattE description string for polynomials
- `measure` – string, the measure to use

Allowed values are:

- `ambient` (default): Lebesgue measure of ambient space,
- `induced`: Lebesgue measure of the affine hull,
- `induced_nonnormalized`: Lebesgue measure of the affine hull without the normalization by \(\sqrt{\det(A^T A)}\) (with \(A\) being the affine transformation matrix; see `affine_hull()`).
- `**kwds` – additional keyword arguments that are passed to the engine

**OUTPUT:**

The integral of the polynomial over the polytope

**Note:** The polytope triangulation algorithm is used. This function depends on LattE (i.e., the latte_int optional package).

**EXAMPLES:**
If the polyhedron has floating point coordinates, an inexact result can be obtained if we transform to rational coordinates:

```python
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
0
sage: ixy = P.integrate(x*y, measure='induced'); ixy  # optional - latte_int
0.4714045207910317?
sage: ixy.parent()  # optional - latte_int
Algebraic Real Field
```

Convert to a symbolic expression:

```python
sage: ixy.radical_expression()  # optional - latte_int
1/3*sqrt(2)
```

Another non full-dimensional polytope integration:

```python
sage: R.<x, y, z> = QQ[]
sage: P = polytopes.simplex(2)
sage: V = AA(P.volume(measure='induced')); V.radical_expression()  # optional
1/2*sqrt(3)
sage: P.integrate(R(1), measure='induced') == V  # optional
True
```

Computing the mass center:

```python
sage: (P.integrate(x, measure='induced') / V).radical_expression()  # optional
1/3
```

```
interior()
The interior of self.
```
OUTPUT:

• either an empty polyhedron or an instance of `RelativeInterior`

EXAMPLES:

If the polyhedron is full-dimensional, the result is the same as that of `relative_interior()`:

```sage
sage: P_full = Polyhedron(vertices=[[0,0],[1,1],[1,-1]])
sage: P_full.interior()
Relative interior of a 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 3 vertices
```

If the polyhedron is of strictly smaller dimension than the ambient space, its interior is empty:

```sage
sage: P_lower = Polyhedron(vertices=[[0,1], [0,-1]])
sage: P_lower.interior()
The empty polyhedron in ZZ^2
```

`interior_contains(point)`

Test whether the interior of the polyhedron contains the given point.

See also:
`contains()`, `relative_interior_contains()`.

INPUT:

• point – coordinates of a point

OUTPUT:

True or False.

EXAMPLES:

```sage
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:

```sage
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:

```sage
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 $\mathbb{Z}$-polyhedra might not be a $\mathbb{Z}$-polyhedron. In this case, a $\mathbb{Q}$-polyhedron is returned.

EXAMPLES:

```sage
cube = polytopes.hypercube(3)
oc = polytopes.cross_polytope(3)
cube.intersection(oct*2)
```

A 3-dimensional polyhedron in $\mathbb{Z}^3$ defined as the convex hull of 12 vertices

As a shorthand, one may use:

```sage
cube & oct*2
```

A 3-dimensional polyhedron in $\mathbb{Z}^3$ defined as the convex hull of 12 vertices

The intersection of two $\mathbb{Z}$-polyhedra is not necessarily a $\mathbb{Z}$-polyhedron:

```sage
P = Polyhedron([(0,0),(1,1)], base_ring=ZZ)
P.intersection(P)
```

A 1-dimensional polyhedron in $\mathbb{Z}^2$ defined as the convex hull of 2 vertices

```sage
Q = Polyhedron([(0,1),(1,0)], base_ring=ZZ)
P.intersection(Q)
```

A 0-dimensional polyhedron in $\mathbb{Q}^2$ defined as the convex hull of 1 vertex

```sage
_.Vrepresentation()
```

(A vertex at (1/2, 1/2),)

```is_bipyramid(certificate=False)```

Test whether the polytope is combinatorially equivalent to a bipyramid over some polytope.

INPUT:

• certificate – boolean (default: False); specifies whether to return two vertices of the polytope which are the apices of a bipyramid, if found

OUTPUT:

If certificate is True, returns a tuple containing:

1. Boolean.
2. None or a tuple containing:
   a. The first apex.
   b. The second apex.

If certificate is False returns a boolean.

EXAMPLES:

```sage
P = polytopes.octahedron()
P.is_bipyramid()
```

True

```sage
P.is_bipyramid(certificate=True)
```

(True, [A vertex at (-1, 0, 0), A vertex at (1, 0, 0)])

(continues on next page)
sage: Q = polytopes.cyclic_polytope(3,7)
sage: Q.is_bipyramid()
False
sage: R = Q.bipyramid()
sage: R.is_bipyramid(certificate=True)
(True, [A vertex at (-1, 3, 13, 63), A vertex at (1, 3, 13, 63)])

ALGORITHM:
Assume all faces of a polyhedron to be given as lists of vertices.

A polytope is a bipyramid with apexes \( v \), \( w \) if and only if for each proper face \( v \in F \) there exists a face \( G \) with \( G \setminus \{ w \} = F \setminus \{ v \} \) and vice versa (for each proper face \( w \in F \) there exists ...).

To check this property it suffices to check for all facets of the polyhedron.

```
is_combinatorially_isomorphic(other, algorithm='bipartite_graph')
```

Return whether the polyhedron is combinatorially isomorphic to another polyhedron.

We only consider bounded polyhedra. By definition, they are combinatorially isomorphic if their face lattices are isomorphic.

INPUT:

- other – a polyhedron object
- algorithm (default = bipartite_graph) – the algorithm to use. The other possible value is face_lattice.

OUTPUT:

- True if the two polyhedra are combinatorially isomorphic
- False otherwise

See also:

```
combinatorial_automorphism_group(), vertex_facet_graph()
```

REFERENCES:

For the equivalence of the two algorithms see [KK1995], p. 877-878

EXAMPLES:

The square is combinatorially isomorphic to the 2-dimensional cube:

```
sage: polytopes.hypercube(2).is_combinatorially_isomorphic(polytopes.regular_polygon(4))
True
```

All the faces of the 3-dimensional permutahedron are either combinatorially isomorphic to a square or a hexagon:

```
sage: H = polytopes.regular_polygon(6)
sage: S = polytopes.hypercube(2)
sage: P = polytopes.permutahedron(4)
sage: all(F.as_polyhedron().is_combinatorially_isomorphic(S) or F.as_polyhedron().is_combinatorially_isomorphic(H) for F in P.faces(2))
True
```
Checking that a regular simplex intersected with its reflection through the origin is combinatorially isomorphic to the intersection of a cube with a hyperplane perpendicular to its long diagonal:

```python
sage: def simplex_intersection(k):
    ...:     S1 = Polyhedron([vector(v) - vector(polytopes.simplex(k).center()) for v in polytopes.simplex(k).vertices_list()])
    ...:     S2 = Polyhedron([-vector(v) for v in S1.vertices_list()])
    ...:     return S1.intersection(S2)

sage: def cube_intersection(k):
    ...:     C = polytopes.hypercube(k+1)
    ...:     H = Polyhedron(eqns=[[0]+[1 for i in range(k+1)])
    ...:     return C.intersection(H)

sage: [simplex_intersection(k).is_combinatorially_isomorphic(cube_intersection(k)) for k in range(2,5)]
[True, True, True]

sage: simplex_intersection(2).is_combinatorially_isomorphic(polytopes.regular_polygon(6))
True
sage: simplex_intersection(3).is_combinatorially_isomorphic(polytopes.octahedron())
True
```

Two polytopes with the same $f$-vector, but different combinatorial types:

```python
sage: P = Polyhedron([-605520/1525633, -605520/1525633, -1261500/1525633, -52200/1525633, 11833/1525633],
                [-720/1769, -600/1769, 1500/1769, 0, -31/1769],
                [-216/749, 240/749, -240/749, -432/749, 461/749],
                [-50/181, 50/181, 60/181, -100/181, -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

sage: S = polytopes.simplex(3)
sage: S = S.face_truncation(S.faces(0)[3])
sage: S = S.face_truncation(S.faces(0)[4])
sage: S = S.face_truncation(S.faces(0)[5])
sage: T = polytopes.simplex(3)
sage: T = T.face_truncation(T.faces(0)[3])
sage: T = T.face_truncation(T.faces(0)[4])
sage: T = T.face_truncation(T.faces(0)[4])
sage: T.is_combinatorially_isomorphic(S)
False
sage: T.f_vector(), S.f_vector()
((1, 10, 15, 7, 1), (1, 10, 15, 7, 1))
```

(continues on next page)
```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'

sage: G = Graph()
sage: C.is_combinatorially_isomorphic(G)
Traceback (most recent call last):
  ...  
AssertionError: input `other` must be a polyhedron

sage: H = Polyhedron(eqns=[[0,1,1,1,1]]); H
A 3-dimensional polyhedron in QQ^4 defined as the convex hull of 1 vertex and 3
lines
sage: C.is_combinatorially_isomorphic(H)
Traceback (most recent call last):
  ...  
AssertionError: polyhedron `other` must be bounded
```

**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]]); P
A 2-dimensional polyhedron in ZZ^3 defined as the convex hull of 3 vertices
sage: P.is_empty(), P.is_universe()
(False, False)

sage: Q = Polyhedron(vertices=()); Q
The empty polyhedron in ZZ^0
sage: Q.is_empty(), Q.is_universe()
(True, False)
```

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

```sage
R.is_empty(), R.is_universe()
```

\[(False, True)\]

### is_immutable()

Returns True if the polyhedron is immutable, i.e. it cannot be modified in place.

**EXAMPLES:**

```sage
p = polytopes.cube(backend='field')
p.is_immutable()
```

\[True\]

### 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:**

```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]])
q.is_inscribed(certificate=True)
```

\[(True, (0, 0, 0, 0))\]

```sage
cube = polytopes.cube()
cube.is_inscribed()
```

\[True\]

```sage
translated_cube = Polyhedron(verts=
....: [v.vector() + vector([1,2,3])
....: for v in cube.verts()])
translated_cube.is_inscribed(certificate=True)
```

\[(True, (1, 2, 3))\]
sage: truncated_cube = cube.face_truncation(cube.faces(0)[0])
sage: truncated_cube.is_inscribed()
False

The method is not implemented for non-full-dimensional polytope or unbounded polyhedra:

sage: square = Polyhedron(vertices=[[1,0,0],[0,1,0],[1,1,0],[0,0,0]])
sage: square.is_inscribed()
Traceback (most recent call last):
  ... 
NotImplementedError: this function is implemented for full-dimensional polyhedra only

sage: p = Polyhedron(vertices=[[0,0]],rays=[[1,0],[0,1]])
sage: p.is_inscribed()
Traceback (most recent call last):
  ... 
NotImplementedError: this function is not implemented for unbounded polyhedra

\textbf{is\_lattice\_polytope()}

Return whether the polyhedron is a lattice polytope.

\textbf{OUTPUT:}

\text{True} if the polyhedron is compact and has only integral vertices, \text{False} otherwise.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: polytopes.cross_polytope(3).is_lattice_polytope()
True
sage: polytopes.regular_polygon(5).is_lattice_polytope()
False
\end{verbatim}

\textbf{is\_lawrence\_polytope()}

Return True if self is a Lawrence polytope.

A polytope is called a Lawrence polytope if it has a centrally symmetric (normalized) Gale diagram.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: P = polytopes.hypersimplex(5,2)
sage: L = P.lawrence_polytope()  
sage: L.is_lattice_polytope()  
True
sage: egyptian_pyramid = polytopes.regular_polygon(4).pyramid()  
sage: egyptian_pyramid.lawrence_polytope()  
True
sage: polytopes.octahedron().is_lawrence_polytope()
False
\end{verbatim}

\textbf{REFERENCES:}

For more information, see [BaSt1990].

\textbf{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 \( \text{self} \) can be written as \( Y \oplus Z \).

EXAMPLES:

```
sage: A = polytopes.hypercube(2)
sage: B = Polyhedron( vertices=[(0,1), (1/2,1)])
sage: C = Polyhedron( vertices=[(1,1)])
sage: A.is_minkowski_summand(B)
True
sage: A.is_minkowski_summand(C)
True
sage: B.is_minkowski_summand(C)
True
sage: B.is_minkowski_summand(A)
False
sage: C.is_minkowski_summand(A)
False
sage: C.is_minkowski_summand(B)
False
```

.. automethod:: is_mutable()

Return True if the polyhedron is mutable, i.e. it can be modified in place.

EXAMPLES:

```
sage: p = polytopes.cube(backend='field')
sage: p.is_mutable()
False
```

.. automethod:: is_neighborly(k=None)

Return whether the polyhedron is neighborly.

If the input \( k \) is provided, then return whether the polyhedron is \( k \)-neighborly

A polyhedron is neighborly if every set of \( n \) vertices forms a face for \( n \) up to floor of half the dimension of the polyhedron. It is \( k \)-neighborly if this is true for \( n \) up to \( k \).

INPUT:

- \( k \) – the dimension up to which to check if every set of \( k \) vertices forms a face. If no \( k \) is provided, check up to floor of half the dimension of the polyhedron.

OUTPUT:

- True if every set of up to \( k \) vertices forms a face,
- False otherwise

See also:

.. method:: neighborliness()

EXAMPLES:

```
sage: cube = polytopes.hypercube(3)
sage: cube.is_neighborly()
True
```
Cyclic polytopes are neighborly:

```
sage: all(polytopes.cyclic_polytope(i, i + 1 + j).is_neighborly() for i in \n    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:

```
sage: testpolys = [polytopes.cube(), polytopes.cyclic_polytope(6, 9), polytopes.\n    simplex(6)]
sage: [(P.neighborliness()>=floor(P.dim()/2)) == P.is_neighborly() for P in \n    testpolys]
[True, True, True]
```

### is_prism(certificate=False)

Test whether the polytope is combinatorially equivalent to a prism of some polytope.

**INPUT:**

- **certificate** – boolean (default: False); specifies whether to return two facets of the polytope which are the bases of a prism, if found

**OUTPUT:**

If certificate is True, returns a tuple containing:

1. Boolean.
2. **None** or a tuple containing:
   a. List of the vertices of the first base facet.
   b. List of the vertices of the second base facet.

If certificate is False returns a boolean.

**EXAMPLES:**

```
sage: P = polytopes.cube()
sage: P.is_prism()
True
sage: P.is_prism(certificate=True)
(True,
  [[[A vertex at (1, -1, -1),
    A vertex at (1, 1, -1),
    A vertex at (1, 1, 1),
    A vertex at (1, -1, 1)],[[A vertex at (-1, -1, 1),
    A vertex at (-1, -1, -1),
    A vertex at (-1, 1, -1),
    A vertex at (-1, 1, 1)]]])
sage: Q = polytopes.cyclic_polytope(3,8)
```

ALGORITHM:

See `Polyhedron_base.is_bipyramid()`.

**is_pyramid**(certificate=False)

Test whether the polytope is a pyramid over one of its facets.

INPUT:

- certificate – boolean (default: False); specifies whether to return a vertex of the polytope which is the apex of a pyramid, if found

OUTPUT:

If certificate is True, returns a tuple containing:

1. Boolean.
2. The apex of the pyramid or None.

If certificate is False returns a boolean.

EXAMPLES:

```python
sage: P = polytopes.simplex(3)
sage: P.is_pyramid()
True
sage: P.is_pyramid(certificate=True)
(True, A vertex at (1, 0, 0, 0))
sage: egyptian_pyramid = polytopes.regular_polygon(4).pyramid()
sage: egyptian_pyramid.is_pyramid()
True
sage: Q = polytopes.octahedron()
sage: Q.is_pyramid()
False
```
For the 0-dimensional polyhedron, the output is True, but it cannot be constructed as a pyramid over the empty polyhedron:

```sage
def P = Polyhedron([[0]])
print P.is_pyramid()
True
print Polyhedron().pyramid()
Traceback (most recent call last):
...
ZeroDivisionError: rational division by zero
```

**is_relatively_open()**

Returns whether `self` is relatively open.

**OUTPUT:**

Boolean.

**EXAMPLES:**

```sage
def P = Polyhedron(vertices=[(1,0, (-1,0)]); P
A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
print P.is_relatively_open()
False

def P0 = Polyhedron(vertices=[[1, 2]]); P0
A 0-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex
print P0.is_relatively_open()
True

def Empty = Polyhedron(ambient_dim=2); Empty
The empty polyhedron in ZZ^2
print Empty.is_relatively_open()
True

def Line = Polyhedron(vertices=[[1, 1]], lines=[[1, 0]]); Line
A 1-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 1 line
print Line.is_relatively_open()
True
```

**is_self_dual()**

Return whether the polytope is self-dual.

A polytope is self-dual if its face lattice is isomorphic to the face lattice of its dual polytope.

**EXAMPLES:**

```sage
def polytopes.simplex().is_self_dual()
True
def polytopes.twenty_four_cell().is_self_dual()
True
def polytopes.cube().is_self_dual()
False
def polytopes.hypersimplex(5,2).is_self_dual()
False
```
```python
sage: P = Polyhedron(vertices=[[1/2, 1/3]], rays=[[1, 1]]).is_self_dual()
Traceback (most recent call last):
  ... 
ValueError: polyhedron has to be compact
```

**is_simple()**

Test for simplicity of a polytope.

See [Wikipedia article Simple_polytope](#).

**EXAMPLES:**

```python
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.

A simplex is a bounded polyhedron with \(d + 1\) vertices, where \(d\) is the dimension.

**EXAMPLES:**

```python
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:**

```python
sage: p = polytopes.hypercube(3)
sage: p.is_simplicial()
False
sage: q = polytopes.simplex(5, project=True)
sage: q.is_simplicial()
True
sage: p = Polyhedron([[0,0,0], [1,0,0], [0,1,0], [0,0,1]])
sage: p.is_simplicial()
True
sage: q = Polyhedron([[1,1,1],[-1,1,1],[1,-1,1],[-1,-1,1],[1,1,-1]])
sage: q.is_simplicial()
False
sage: P = polytopes.simplex(); P
A 3-dimensional polyhedron in ZZ^4 defined as the convex hull of 4 vertices
```
sage: P.is_simplicial()
True

The method is not implemented for unbounded polyhedra:

sage: p = Polyhedron(vertices=[(0,0)],rays=[(1,0),(0,1)])
sage: p.is_simplicial()
Traceback (most recent call last):
  ...
NotImplementedError: this function is implemented for polytopes only

**is_universe()**

Test whether the polyhedron is the whole ambient space

**OUTPUT:**

Boolean.

**EXAMPLES:**

sage: P = Polyhedron(vertices=[[1,0,0],[0,1,0],[0,0,1]]); P
A 2-dimensional polyhedron in ZZ^3 defined as the convex hull of 3 vertices
sage: P.is_empty(), P.is_universe()
(False, False)

sage: Q = Polyhedron(vertices=()); Q
The empty polyhedron in ZZ^0
sage: Q.is_empty(), Q.is_universe()
(True, False)

sage: R = Polyhedron(lines=[[1,0],[0,1]]); R
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex and 2 → lines
sage: R.is_empty(), R.is_universe()
(False, True)

**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
A 3-dimensional polyhedron in \( \mathbb{Q}^3 \) 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 \( \mathbb{Z}^3 \) defined as the convex hull of 2 vertices and 2 rays
```

**join_of_Vrep(**Vrepresentatives*)***

Return the smallest face that contains Vrepresentatives.

**INPUT:**

- Vrepresentatives – vertices/rays/lines of self or indices of such

**OUTPUT:** a PolyhedronFace

**Note:** In the case of unbounded polyhedra, the join of rays etc. may not be well-defined.

**EXAMPLES:**

```
sage: P = polytopes.permutahedron(5)
sage: P.join_of_Vrep(1)
A 0-dimensional face of a Polyhedron in \( \mathbb{Z}^5 \) defined as the convex hull of 1 vertex

sage: P.join_of_Vrep()
A -1-dimensional face of a Polyhedron in \( \mathbb{Z}^5 \)

sage: P.join_of_Vrep(0,12,13).ambient_V_indices()
(0, 12, 13, 68)
```

The input is flexible:

```
sage: P.join_of_Vrep(2, P.vertices()[3], P.Vrepresentation(4))
A 2-dimensional face of a Polyhedron in \( \mathbb{Z}^5 \) defined as the convex hull of 6 vertices
```

```
sage: P = polytopes.cube()
sage: a, b = P.faces(0)[:2]
sage: P.join_of_Vrep(a, b)
A 1-dimensional face of a Polyhedron in \( \mathbb{Z}^3 \) defined as the convex hull of 2 vertices
```

In the case of an unbounded polyhedron, the join may not be well-defined:

```
sage: P = Polyhedron(          vertices=[[1,0], [0,1]],          rays=[[1,1]])
sage: P.join_of_Vrep(0)
A 0-dimensional face of a Polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 1 vertex

sage: P.join_of_Vrep(0,1)
A 1-dimensional face of a Polyhedron in \( \mathbb{Q}^2 \) defined as the convex hull of 2 vertices
```

(continues on next page)
sage: P.join_of_Vrep(0,2)
A 1-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 1 vertex and 1 ray
sage: P.join_of_Vrep(1,2)
A 1-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 1 vertex and 1 ray
sage: P.join_of_Vrep(2)
Traceback (most recent call last):
... ValueError: the join is not well-defined

The Vrepresentatives must be of self:

sage: P = polytopes.cube(backend='ppl')
sage: Q = polytopes.cube(backend='field')
sage: v = P.vertices()[0]
sage: P.join_of_Vrep(v)
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1 vertex
sage: Q.join_of_Vrep(v)
Traceback (most recent call last):
... ValueError: not a Vrepresentative of `self`
sage: f = P.faces(0)[0]
sage: P.join_of_Vrep(v)
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1 vertex
sage: Q.join_of_Vrep(v)
Traceback (most recent call last):
... ValueError: not a Vrepresentative of `self`

``lattice_polytope``(envelope=False)
Return an encompassing lattice polytope.

**INPUT:**

- **envelope** – boolean (default: False). If the polyhedron has non-integral vertices, this option decides whether to return a strictly larger lattice polytope or raise a `ValueError`. This option has no effect if the polyhedron has already integral vertices.

**OUTPUT:**

A `LatticePolytope`. If the polyhedron is compact and has integral vertices, the lattice polytope equals the polyhedron. If the polyhedron is compact but has at least one non-integral vertex, a strictly larger lattice polytope is returned.

If the polyhedron is not compact, a `NotImplementedError` is raised.

If the polyhedron is not integral and `envelope=False`, a `ValueError` is raised.

**ALGORITHM:**

For each non-integral vertex, a bounding box of integral points is added and the convex hull of these integral points is returned.

**EXAMPLES:**
First, a polyhedron with integral vertices:

```python
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:

```python
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( 0, 1),
M( 1, 0)
in 2-d lattice M
```

```python
lawrence_extension(v)

Return the Lawrence extension of self on the point v.

Let P be a polytope and v be a vertex of P or a point outside P. The Lawrence extension of P on v is the
convex hull of (v, 1), (v, 2) and (u, 0) for all vertices u in P other than v if v is a vertex.

INPUT:

- v – a vertex of self or a point outside it

EXAMPLES:

```python
sage: P = polytopes.cube()
sage: P.lawrence_extension(P.vertices()[0])
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 9 vertices
sage: P.lawrence_extension([-1,-1,-1])
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 9 vertices
```

REFERENCES:

For more information, see Section 6.6 of [Zie2007].

```python
lawrence_polytope()

Return the Lawrence polytope of self.

Let P be a d-polytope in R^d with n vertices. The Lawrence polytope of P is the polytope whose vertices
```
are the columns of the following \((r + n)\)-by-\(2n\) matrix.

\[
\begin{pmatrix}
V & V \\
I_n & 2I_n
\end{pmatrix},
\]

where \(V\) is the \(r\)-by-\(n\) vertices matrix of \(P\).

EXAMPLES:

```python
sage: P = polytopes.octahedron()
sage: L = P.lawrence_polytope(); L
A 9-dimensional polyhedron in ZZ^9 defined as the convex hull of 12 vertices
sage: V = P.vertices_list()
sage: i = 0
sage: for v in V:
    ....:     v = v + i*[0]
    ....:     P = P.lawrence_extension(v)
    ....:     i = i + 1
sage: P == L
True
```

REFERENCES:

For more information, see Section 6.6 of [Zie2007].

least_common_superface_of_Vrep(*Vrepresentatives)

Return the smallest face that contains \(V\)representatives.

INPUT:

- \(V\)representatives – vertices/rays/lines of \(self\) or indices of such

OUTPUT: a PolyhedronFace

Note: In the case of unbounded polyhedra, the join of rays etc. may not be well-defined.

EXAMPLES:

```python
sage: P = polytopes.permutahedron(5)
sage: P.join_of_Vrep(1)
A 0-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 1
vertex
sage: P.join_of_Vrep()
A -1-dimensional face of a Polyhedron in ZZ^5
sage: P.join_of_Vrep(0,12,13).ambient_V_indices()
(0, 12, 13, 68)
```

The input is flexible:

```python
sage: P.join_of_Vrep(2, P.vertices()[3], P.Vrepresentation(4))
A 2-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 6
vertices
```

```python
sage: P = polytopes.cube()
sage: a, b = P.faces(0)[:2]
sage: P.join_of_Vrep(a, b)
A 1-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 2
vertices
```
In the case of an unbounded polyhedron, the join may not be well-defined:

```sage
sage: P = Polyhedron(vertices=[[1,0], [0,1]], rays=[[1,1]])
sage: P.join_of_Vrep(0)
A 0-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 1 vertex
sage: P.join_of_Vrep(0,1)
A 1-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 2 vertices
sage: P.join_of_Vrep(0,2)
A 1-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 1 vertex and 1 ray
sage: P.join_of_Vrep(1,2)
A 1-dimensional face of a Polyhedron in QQ^2 defined as the convex hull of 1 vertex and 1 ray
sage: P.join_of_Vrep(2)
Traceback (most recent call last):
... ValueError: the join is not well-defined
```

The Vrepresentatives must be of self:

```sage
sage: P = polytopes.cube(backend='ppl')
sage: Q = polytopes.cube(backend='field')
sage: v = P.vertices()[0]
sage: P.join_of_Vrep(v)
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1 vertex
sage: Q.join_of_Vrep(v)
Traceback (most recent call last):
... ValueError: not a Vrepresentative of ``self``
sage: f = P.faces(0)[0]
sage: P.join_of_Vrep(v)
A 0-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 1 vertex
sage: Q.join_of_Vrep(v)
Traceback (most recent call last):
... ValueError: not a Vrepresentative of ``self``
```

**line_generator()**

Return a generator for the lines of the polyhedron.

**EXAMPLES:**

```sage
sage: pr = Polyhedron(rays = [[1,0],[-1,0],[0,1]], vertices = [[-1,-1]])
sage: next(pr.line_generator()).vector()
(1, 0)
```

**linear_transformation(linear_transf, new_base_ring=None)**

Return the linear transformation of self.
INPUT:

- `linear_transf` – a matrix, not necessarily in `base_ring()`
- `new_base_ring` – ring (optional); specify the new base ring; may avoid coercion failure

OUTPUT:

The polyhedron transformed by that matrix, possibly coerced to a bigger base ring.

EXAMPLES:

```python
sage: b3 = polytopes.Birkhoff_polytope(3)
sage: proj_mat = matrix([[0,1,0,0,0,0,0,0,0],[0,0,0,1,0,0,0,0,0],[0,0,0,0,0,1,0,0,0],[0,0,0,0,0,0,0,1,0]])
sage: b3_proj = proj_mat * b3; b3_proj
A 3-dimensional polyhedron in ZZ^4 defined as the convex hull of 5 vertices

sage: square = polytopes.regular_polygon(4)
sage: transf = matrix([[1,1],[0,1]])
sage: sheared = transf * square
sage: sheared.vertices_list()
[[[-1, -1], [1, 0], [-1, 0], [1, 1]]
sage: sheared == square.linear_transformation(transf)
True
```

Specifying the new base ring may avoid coercion failure:

```python
sage: K.<sqrt2> = QuadraticField(2)
sage: L.<sqrt3> = QuadraticField(3)
sage: P = polytopes.cube() * sqrt2
sage: M = matrix([[sqrt3, 0, 0], [0, sqrt3, 0], [0, 0, 1]])
sage: P.linear_transformation(M, new_base_ring=K.composite_fields(L)[0])
A 3-dimensional polyhedron in (Number Field in sqrt2sqrt3 with defining polynomial x^4 - 10*x^2 + 1 with sqrt2sqrt3 = 0.3178372451957823?)^3 defined as the convex hull of 8 vertices
```

Linear transformation without specified new base ring fails in this case:

```python
sage: M*P
Traceback (most recent call last):
... TypeError: unsupported operand parent(s) for *: 'Full MatrixSpace of 3 by 3 dense matrices over Number Field in sqrt3 with defining polynomial x^2 - 3 with sqrt3 = 1.732050807568878?' and 'Full MatrixSpace of 3 by 8 dense matrices over Number Field in sqrt2 with defining polynomial x^2 - 2 with sqrt2 = 1.414213562373095?'
```

`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:

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

meet_of_Hrep(*Hrepresentatives)
Return the largest face that is contained in Hrepresentatives.

INPUT:
• Hrepresentatives – facets or indices of Hrepresentatives; the indices are assumed to be the indices
  of the Hrepresentation

OUTPUT: a PolyhedronFace

EXAMPLES:

sage: P = polytopes.permutahedron(5)
sage: P.meet_of_Hrep()
A 4-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 120 vertices
sage: P.meet_of_Hrep(1)
A 3-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 24 vertices
sage: P.meet_of_Hrep(4)
A 3-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 12 vertices
sage: P.meet_of_Hrep(1,3,7)
A 1-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 2 vertices
sage: P.meet_of_Hrep(1,3,7).ambient_H_indices()
(0, 1, 3, 7)

The indices are the indices of the Hrepresentation. 0 corresponds to an equation and is ignored:

sage: P.meet_of_Hrep(0)
A 4-dimensional face of a Polyhedron in ZZ^5 defined as the convex hull of 120 vertices

The input is flexible:
The Hrepresentatives must belong to self:

```
sage: P = polytopes.cube(backend='ppl')
sage: Q = polytopes.cube(backend='field')
sage: f = P.facets()[0]
sage: P.meet_of_Hrep(f)
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4→vertices
sage: Q.meet_of_Hrep(f)
Traceback (most recent call last):
... ValueError: not a facet of `self`
sage: f = P.inequalities()[0]
sage: P.meet_of_Hrep(f)
A 2-dimensional face of a Polyhedron in ZZ^3 defined as the convex hull of 4→vertices
sage: Q.meet_of_Hrep(f)
Traceback (most recent call last):
... ValueError: not a facet of `self`
```

```
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:

```
sage: X = polytopes.hypercube(3)
sage: Y = Polyhedron(verticess=[(0,0,0), (0,0,1), (0,1,0), (1,0,0)]) / 2
sage: (X-Y)-Y == X
True
```

(continues on next page)
The polyhedra need not be full-dimensional:

```
sage: X2 = Polyhedron(vertices=[(-1,-1,0),(1,-1,0),(-1,1,0),(1,1,0)])
sage: Y2 = Polyhedron(vertices=[(0,0,0), (0,1,0), (1,0,0)]) / 2
sage: (X2+Y2)-Y2 == X2
True
sage: (X2-Y2)+Y2 < X2
True
```

Minus sign is really an alias for `minkowski_difference()`

```
sage: four_cube = polytopes.hypercube(4)
sage: four_simplex = Polyhedron(vertices = [[0, 0, 0, 1], [0, 0, 1, 0], [0, 1, 0, 0], [1, 0, 0, 0]])
sage: four_cube + four_simplex
A 4-dimensional polyhedron in QQ^4 defined as the convex hull of 16 vertices
sage: four_cube.minkowski_difference(four_simplex) == four_cube - four_simplex
True
```

Coercion of the base ring works:

```
sage: poly_spam = Polyhedron([[3,4,5,2],[1,0,0,1],[0,0,0,0],[0,4,3,2],[-3,-3,-3,-3]], base_ring=ZZ)
sage: poly_eggs = Polyhedron([[5,4,5,4],[-4,5,-4,5],[4,-5,4,-5],[0,0,0,0]], base_ring=QQ) / 100
sage: poly_spam - poly_eggs
A 4-dimensional polyhedron in QQ^4 defined as the convex hull of 5 vertices
```

```
minkowski_sum(other)
Return the Minkowski sum.

Minkowski addition of two subsets of a vector space is defined as

\[ X \oplus Y = \bigcup_{y \in Y} (X + y) = \bigcup_{x \in X, y \in Y} (x + y) \]

See `minkowski_difference()` for a partial inverse operation.
```

INPUT:

• other -- a Polyhedron_base

OUTPUT:

The Minkowski sum of self and other

EXAMPLES:

```
sage: X = polytopes.hypercube(3)
sage: Y = Polyhedron(vertices=[(0,0,0), (0,0,1/2), (0,1/2,0), (1/2,0,0)])
sage: X+Y
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 13 vertices
sage: four_cube = polytopes.hypercube(4)
```

(continues on next page)
sage: four_simplex = Polyhedron(vertices = [[0, 0, 0, 1], [0, 0, 1, 0], [0, 1, 0, 0], [1, 0, 0, 0]])
sage: four_cube + four_simplex
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 36 vertices
sage: four_cube.minkowski_sum(four_simplex) == four_cube + four_simplex
True

sage: poly_spam = Polyhedron([[3,4,5,2],[1,0,0,1],[0,0,0,0],[0,4,3,2],[-3,-3,-3,-3]], base_ring=ZZ)
sage: poly_eggs = Polyhedron([[5,4,5,4],[-4,5,-4,5],[4,-5,4,-5],[0,0,0,0]], base_ring=QQ)
sage: poly_spam + poly_spam + poly_eggs
A 4-dimensional polyhedron in QQ^4 defined as the convex hull of 12 vertices

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.

**Warning:** If the polyhedron has lines, return the number of vertices in the Vrepresentation. As the represented polyhedron has no 0-dimensional faces (i.e. vertices), n_vertices corresponds to the number of k-faces, where k is the number of lines:

```
sage: P = Polyhedron(rays=[[1,0,0]],lines=[[0,1,0]])
sage: P.n_vertices()
1
sage: P.faces(0)
()
sage: P.f_vector()
(1, 0, 1, 1)
```
sage: P = Polyhedron(rays=[[1,0,0]],lines=[[0,1,0],[0,1,1]])
sage: P.n_vertices()
1
sage: P.f_vector()
(1, 0, 0, 1, 1)

EXAMPLES:

sage: p = Polyhedron(vertices=[[1,0],[0,1],[1,1]], rays=[[1,1]])
sage: p.n_vertices()
2

neighborliness()

Return the largest $k$, such that the polyhedron is $k$-neighborly.

A polyhedron is $k$-neighborly if every set of $n$ vertices forms a face for $n$ up to $k$.

In case of the $d$-dimensional simplex, it returns $d + 1$.

See also:

is_neighborly()

EXAMPLES:

sage: cube = polytopes.cube()
sage: cube.neighborliness()
1
sage: P = Polyhedron(); P
The empty polyhedron in ZZ^0
sage: P.neighborliness()
0
sage: P = Polyhedron([[0]]); P
A 0-dimensional polyhedron in ZZ^1 defined as the convex hull of 1 vertex
sage: P.neighborliness()
1
sage: S = polytopes.simplex(5); S
A 5-dimensional polyhedron in ZZ^6 defined as the convex hull of 6 vertices
sage: S.neighborliness()
6
sage: C = polytopes.cyclic_polytope(7,10); C
A 7-dimensional polyhedron in QQ^7 defined as the convex hull of 10 vertices
sage: C.neighborliness()
3
sage: C = polytopes.cyclic_polytope(6,11); C
A 6-dimensional polyhedron in QQ^6 defined as the convex hull of 11 vertices
sage: C.neighborliness()
3
sage: [polytopes.cyclic_polytope(5,n).neighborliness() for n in range(6,10)]
[6, 2, 2, 2]

normal_fan(direction='inner')

Return the normal fan of a compact full-dimensional rational polyhedron.

This returns the inner normal fan of self. For the outer normal fan, use direction='outer'.

490 Chapter 2. Polyhedral computations
INPUT:

- direction – either 'inner' (default) or 'outer'; if set to 'inner', use the inner normal vectors to span the cones of the fan, if set to 'outer', use the outer normal vectors.

OUTPUT:

A complete fan of the ambient space as a RationalPolyhedralFan.

See also:

face_fan().

EXAMPLES:

```sage
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
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

sage: P = Polyhedron(vertices=[[0,0],[2,0],[0,2],[2,1],[1,2]])
sage: P.normal_fan(direction=None)
Traceback (most recent call last):
  ... TypeError: the direction should be 'inner' or 'outer'

sage: inner_nf = P.normal_fan()
sage: inner_nf.rays()
N( 1, 0),
N( 0, -1),
N( 0, 1),
N(-1, 0),
N(-1, -1)
```

(continues on next page)
in 2-d lattice N

```python
sage: outer_nf = P.normal_fan(direction='outer')
sage: outer_nf.rays()
N( 1, 0),
N( 1, 1),
N( 0, 1),
N(-1, 0),
N( 0, -1)
in 2-d lattice N
```

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:
```python
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)
```

It works with a polyhedral face as well:
```python
sage: vv = cube.faces(0)[1]
sage: ops_cube2 = cube.one_point_suspension(vv)
sage: ops_cube == ops_cube2
True
```

See also:
`.face_split()`
`.plot(point=None, line=None, polygon=None, wireframe='blue', fill='green', position=None, orthonormal=True, **kwds)`
Return a graphical representation.

INPUT:
• point, line, polygon – Parameters to pass to point (0d), line (1d), and polygon (2d) plot commands. Allowed values are:
  - A Python dictionary to be passed as keywords to the plot commands.
- A string or triple of numbers: The color. This is equivalent to passing the dictionary {'color': ..}
- False: Switches off the drawing of the corresponding graphics object

- **wireframe**
  - Similar to **point**, **line**, and **polygon**, but **fill** is used for the graphics objects in the dimension of the polytope (or of dimension 2 for higher dimensional polytopes) and **wireframe** is used for all lower-dimensional graphics objects (default: 'green' for **fill** and 'blue' for **wireframe**)

- **position**
  - positive number; the position to take the projection point in Schlegel diagrams.

- **orthonormal**
  - Boolean (default: True); whether to use orthonormal projections.

- **kwds**
  - optional keyword parameters that are passed to all graphics objects.

**OUTPUT:**

A (multipart) graphics object.

**EXAMPLES:**

```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()
```

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 0 graphics primitives
sage: cube.plot(point=False, line='red', polygon=False)
Graphics3d Object
sage: hypercube.plot(point=False, line='red', polygon=False)
```

Draw points in red, no lines, and a blue polygon:

```python
sage: square.plot(point={'color':'red'}, line=False, polygon=(0,0,1))
```

(continues on next page)
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))
```

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')
```

It is possible to draw polyhedra up to dimension 4, no matter what the ambient dimension is:

```
sage: hcube = polytopes.hypercube(5)
sage: facet = hcube.facets()[0].as_polyhedron();facet
A 4-dimensional polyhedron in ZZ^5 defined as the convex hull of 16 vertices
```

```
sage: facet.plot()
```

Graphics3d Object

```
polar(in_affine_span=False)
```

Return the polar (dual) polytope.

The original vertices are translated so that their barycenter is at the origin, and then the vertices are used as the coefficients in the polar inequalities.

The polytope must be full-dimensional, unless in_affine_span is True. If in_affine_span is True, then the operation will be performed in the linear/affine span of the polyhedron (after translation).

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[0,0,1],[0,1,0],[1,0,0],[0,0,0],[1,1,1]], base_˓
˓→ring=QQ)
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
```

in_affine_span somewhat ignores equations, performing the polar in the spanned subspace (after translating barycenter to origin):
Embedding the polytope in a higher dimension, commutes with polar in this case:

```
sage: point = Polyhedron([[0]])
sage: P = polytopes.cube().change_ring(QQ)
sage: (P*point).polar(in_affine_span=True) == P.polar()*point
True
```

**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)
sage: P1.product(P2)
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 4 vertices
```

The Cartesian product is the product in the semiring of polyhedra:

```
sage: P1 * P1
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices
sage: P1 * P2
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 4 vertices
sage: P2 * P2
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 4 vertices
sage: 2 * P1
A 1-dimensional polyhedron in ZZ^1 defined as the convex hull of 2 vertices
sage: P1 * 2.0
A 1-dimensional polyhedron in RDF^1 defined as the convex hull of 2 vertices
```

An alias is `cartesian_product()`:
projection(projection=None)
Return a projection object.

INPUT:

• proj – a projection function

OUTPUT:

The identity projection. This is useful for plotting polyhedra.

See also:

schlegel_projection() for a more interesting projection.

EXAMPLES:

```python
sage: p = polytopes.hypercube(3)
sage: proj = p.projection()
sage: proj
The projection of a polyhedron into 3 dimensions
```

pyramid()
Return 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:

```python
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.
The square of the radius, which is in \textit{base_ring}.}

**EXAMPLES:**

```python
sage: p = polytopes.permutahedron(4, project = False)
sage: p.radius_square()
5
```

\texttt{random\_integral\_point(**kwds)}

Return an integral point in this polyhedron chosen uniformly at random.

**INPUT:**

- **kwds** – optional keyword parameters that are passed to \texttt{self.get\_integral\_point()}.  

**OUTPUT:**

The integral point in the polyhedron chosen uniformly at random. If the polyhedron is not compact, a \texttt{ValueError} is raised. If the polyhedron does not contain any integral points, an \texttt{EmptySetError} is raised.

See also:

\texttt{get\_integral\_point()}.  

**EXAMPLES:**

```python
sage: P = Polyhedron(vertices=[(-1,-1),(1,0),(1,1),(0,1)])
sage: P.random_integral_point()
# random
(0, 0)
sage: P.random_integral_point() in P.integral_points()
True
sage: P.random_integral_point(explicit Enumeration_threshold=0, 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)])
sage: Q.random_integral_point()
Traceback (most recent call last):
  ... ValueError: ...

sage: R = Polyhedron(vertices=[(1/2, 0), (1, 1/2), (0, 1/2)])
sage: R.random_integral_point()
Traceback (most recent call last):
  ... EmptySetError: ...
```

\texttt{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 \texttt{rays()} or \texttt{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()
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
sage: p.rays_list() == [list(r) for r in p.ray_generator()]
True

relative_interior()
Return the relative interior of self.

EXAMPLES:

sage: P = Polyhedron(vertices=[[1,0], (-1,0)])
sage: ri_P = P.relative_interior(); ri_P
Relative interior of
a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: (0, 0) in ri_P
True
sage: (1, 0) in ri_P
False
sage: P0 = Polyhedron(vertices=[[1, 2]])
sage: P0.relative_interior() is P0
True

(continues on next page)
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:

```python
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:

```python
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:

```python
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:
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
sage: p = Polyhedron(vertices=[(3,2)], rays=[(1,-1)])
sage: p.representative_point()
(4, 1)
sage: p.center()
(3, 2)
sage: Polyhedron(vertices=[(3,2)]).representative_point()
(3, 2)
```

restricted_automorphism_group(output=’abstract’)

Return the restricted automorphism group.

First, let the linear automorphism group be the subgroup of the affine group $AGL(d, \mathbb{R}) = GL(d, \mathbb{R}) \rtimes \mathbb{R}^d$ preserving the $d$-dimensional polyhedron. The affine group acts in the usual way $\vec{x} \mapsto A\vec{x} + b$ on the ambient space.

The restricted automorphism group is the subgroup of the linear automorphism group generated by permutations of the generators of the same type. That is, vertices can only be permuted with vertices, ray generators with ray generators, and line generators with line generators.

For example, take the first quadrant $Q = \{(x, y) \mid x \geq 0, \ y \geq 0\} \subset \mathbb{Q}^2$.

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

$$\text{Aut}(Q) = \left\{ \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \right\} \simeq \mathbb{Z}_2$$

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 `self.Vrepresentation(0)` and `self.Vrepresentation(1)`.
- "matrix" – return a matrix group representing affine transformations. When acting on affine vectors, you should append a 1 to every vector. If the polyhedron is not full dimensional, the returned matrices act as the identity on the orthogonal complement of the affine space spanned by the polyhedron.
- "matrixlist" – like `matrix`, but return the list of elements of the matrix group. Useful for fields without a good implementation of matrix groups or to avoid the overhead of creating the group.

**OUTPUT:**

- For `output="abstract"` and `output="permutation"`: a `PermutationGroup`
- For `output="matrix"`: a `MatrixGroup`
- For `output="matrixlist"`: a list of matrices.

**REFERENCES:**

- [BSS2009]

**EXAMPLES:**

A cross-polytope example:

```python
sage: P = polytopes.cross_polytope(3)
sage: P.restricted_automorphism_group() == PermutationGroup([(3,4), (2,3), (4,5), (2,5), (1,2), (5,6), (1,6)])
True
sage: P.restricted_automorphism_group(output="permutation") == PermutationGroup([(2,3), (1,2), (3,4), (1,4), (0,1), (4,5), (0,5)])
True
sage: mgens = [ [1,0,0,0], [0,1,0,0], [0,0,1,0], [0,0,0,1] ]
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
sage: all(t.matrix() in matrix_g for t in poly_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
([1 0 0 0 0] [-87/55 -82/55 -2/5 38/55 98/55 12/11]
[0 1 0 0 0] [-142/55 -27/55 -2/5 38/55 98/55 12/11]
[0 0 1 0 0] [-142/55 -82/55 3/5 38/55 98/55 12/11]
[0 0 0 1 0] [-142/55 -82/55 -2/5 93/55 98/55 12/11]
[0 0 0 0 1] [-142/55 -82/55 -2/5 38/55 153/55 12/11]
[1 0 0 0 0], [0 0 0 0 0 1])
sage: g = AffineGroup(5, QQ)(G[1])
sage: g
[ -87/55 -82/55 -2/5 38/55 98/55] [12/11]
[ -142/55 -27/55 -2/5 38/55 98/55] [12/11]
[-142/55 -82/55 -2/5 93/55 98/55] [12/11]
[-142/55 -82/55 -2/5 38/55 153/55] [12/11]
sage: g^2
[1 0 0 0 0] [0]
[0 1 0 0 0] [0]
x |-> [0 0 1 0 0] x + [0]
[0 0 0 1 0] [0]
[0 0 0 0 1] [0]
sage: g(list(P.vertices()[0]))
(7, 8, 9, 10, 11)
sage: g(list(P.vertices()[1]))
(1, 2, 3, 4, 5)
```

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:

```python
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 - initial_points[1] for pt in initial_points]
sage: Polyhedron(vertices=points).restricted_automorphism_group()
```

(continues on next page)
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 with sqrt5 = 2.236067977499790?)^3 defined as the convex hull of 20 vertices
sage: G = P.restricted_automorphism_group(output="matrixlist")
sage: len(G)
120
```

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: (facet=None, position=None)

Return the Schlegel projection.

- The facet is orthonormally transformed into its affine hull.
- The position specifies a point coming out of the barycenter of the facet from which the other vertices will be projected into the facet.

**INPUT:**

- facet – a PolyhedronFace. The facet into which the Schlegel diagram is created. The default is the first facet.
- position – a positive number. Determines a relative distance from the barycenter of facet. A value close to 0 will place the projection point close to the facet and a large value further away. Default is 1. If the given value is too large, an error is returned.

**OUTPUT:**

A :class:`Projection` object.

**EXAMPLES:**

```
sage: p = polytopes.hypercube(3)
sage: sch_proj = p.schlegel_projection()
sage: schlegel_edge_indices = sch_proj.lines
sage: schlegel_edges = [sch_proj.coordinates_of(x) for x in schlegel_edge_indices]
sage: len([x for x in schlegel_edges if x[0][0] > 0])
8
```

The Schlegel projection preserves the convexity of facets, see trac ticket #30015:
The same truncated cube but see inside the tetrahedral facet:

```
sage: tfcube.facets()[4]
A 3-dimensional face of a Polyhedron in QQ^4 defined as the convex hull of 4 vertices
sage: sp = tfcube.schlegel_projection(tfcube.facets()[4])
sage: sp.plot()
Graphics3d Object
```

A different values of position changes the projection:

```
sage: sp = tfcube.schlegel_projection(tfcube.facets()[4], 1/2)
sage: sp.plot()
Graphics3d Object
```

A value which is too large give a projection point that sees more than one facet resulting in an error:

```
sage: sp = tfcube.schlegel_projection(tfcube.facets()[4], 5)
Traceback (most recent call last):
...
ValueError: the chosen position is too large
```

**show(****kwds***)

Display graphics immediately

This method attempts to display the graphics immediately, without waiting for the currently running code (if any) to return to the command line. Be careful, calling it from within a loop will potentially launch a large number of external viewer programs.

INPUT:

- **kwds** – optional keyword arguments. See **plot()** for the description of available options.

OUTPUT:

This method does not return anything. Use **plot()** if you want to generate a graphics object that can be saved or further transformed.

EXAMPLES:

```
sage: square = polytopes.hypercube(2)
sage: square.show(point='red')
```

**simpliciality()**

Return the largest integer $k$ such that the polytope is $k$-simplicial.
A polytope is \( k \)-simplicial, if every \( k \)-face is a simplex. If \( self \) is a simplex, returns its dimension.

**EXAMPLES:**

```
sage: polytopes.cyclic_polytope(10,4).simpliciality()
sage: polytopes.hypersimplex(5,2).simpliciality()
sage: polytopes.cross_polytope(4).simpliciality()
sage: polytopes.simplex(3).simpliciality()
sage: polytopes.simplex(1).simpliciality()
```

The method is not implemented for unbounded polyhedra:

```
sage: p = Polyhedron(vertices=[[0,0]],rays=[[1,0],[0,1]])
sage: p.simpliciality()
Traceback (most recent call last):
  ... Not ImplementedError: this function is implemented for polytopes only
```

**simplicity()**

Return the largest integer \( k \) such that the polytope is \( k \)-simple.

A polytope \( P \) is \( k \)-simple, if every \( (d-1-k) \)-face is contained in exactly \( k+1 \) facets of \( P \) for \( 1 \leq k \leq d-1 \). Equivalently it is \( k \)-simple if the polar/dual polytope is \( k \)-simplicial. If \( self \) is a simplex, it returns its dimension.

**EXAMPLES:**

```
sage: polytopes.hypersimplex(4,2).simplicity()
sage: polytopes.hypersimplex(5,2).simplicity()
sage: polytopes.hypersimplex(6,2).simplicity()
sage: polytopes.simplex(3).simplicity()
sage: polytopes.simplex(1).simplicity()
```

The method is not implemented for unbounded polyhedra:

```
sage: p = Polyhedron(vertices=[[0,0]],rays=[[1,0],[0,1]])
sage: p.simplicity()
Traceback (most recent call last):
  ... Not ImplementedError: this function is implemented for polytopes only
```

**slack_matrix()**

Return the slack matrix.

The entries correspond to the evaluation of the Hrepresentation elements on the Vrepresentation elements.
Note: The columns correspond to inequalities/equations in the order \texttt{Hrepresentation()}, the rows correspond to vertices/rays/lines in the order \texttt{Vrepresentation()}.

See also: \texttt{incidence_matrix()}.

EXAMPLES:

```python
sage: P = polytopes.cube()
sage: P.slack_matrix()
[0 2 2 2 0 0]
[0 0 2 2 0 2]
[0 0 0 2 2 2]
[0 2 0 2 2 0]
[2 2 0 0 2 0]
[2 2 2 0 0 0]
[2 0 2 0 0 2]
[2 0 0 0 2 2]

sage: P = polytopes.cube(intervals='zero_one')
sage: P.slack_matrix()
[0 1 1 1 0 0]
[0 0 1 1 0 1]
[0 0 0 1 1 1]
[0 1 0 1 1 0]
[1 1 0 0 1 0]
[1 1 1 0 0 0]
[1 0 1 0 0 1]
[1 0 0 0 1 1]

sage: P = polytopes.dodecahedron().faces(2)[0].as_polyhedron()
sage: P.slack_matrix()
[1/2*sqrt5 - 1/2 0 0 1 1/2*sqrt5 - 1/2 0]
[0 0 1/2*sqrt5 - 1/2 1/2*sqrt5 - 1/2 1]
[0 1/2*sqrt5 - 1/2 1 0 1/2*sqrt5 - 1/2]
[1 1/2*sqrt5 - 1/2 0 1/2*sqrt5 - 1/2 0]
[1/2*sqrt5 - 1/2 1 1/2*sqrt5 - 1/2 0 0]

sage: P = Polyhedron(rays=[[1, 0], [0, 1]])
sage: P.slack_matrix()
[0 0]
[0 1]
[1 0]
```

\texttt{stack}(\texttt{face}, position=None)

Return a new polyhedron formed by stacking onto a face. Stacking a face adds a new vertex located slightly outside of the designated face.
INPUT:

- `face` – a PolyhedronFace

- `position` – a positive number. Determines a relative distance from the barycenter of `face`. A value close to 0 will place the new vertex close to the face and a large value further away. Default is 1. If the given value is too large, an error is returned.

OUTPUT:

A Polyhedron object

EXAMPLES:

```python
sage: cube = polytopes.cube()
sage: square_face = cube.facets()[2]
sage: stacked_square = cube.stack(square_face)
sage: stacked_square.f_vector()
(1, 9, 16, 9, 1)

sage: edge_face = cube.faces(1)[3]
sage: stacked_edge = cube.stack(edge_face)
sage: stacked_edge.f_vector()
(1, 9, 17, 10, 1)

sage: cube.stack(cube.faces(0)[0])
Traceback (most recent call last):
...
ValueError: cannot stack onto a vertex

sage: stacked_square_half = cube.stack(square_face, position=1/2)
sage: stacked_square_half.f_vector()
(1, 9, 16, 9, 1)

sage: hexaprism = polytopes.regular_polygon(6).prism()
sage: hexaprism.f_vector()
(1, 12, 18, 8, 1)
sage: square_face = hexaprism.faces(2)[2]
sage: stacked_hexaprism = hexaprism.stack(square_face)
sage: stacked_hexaprism.f_vector()
(1, 13, 22, 11, 1)

sage: hexaprism.stack(square_face, position=4)
Traceback (most recent call last):
...
ValueError: the chosen position is too large

sage: s = polytopes.simplex(7)
sage: f = s.faces(3)[69]
sage: sf = s.stack(f); sf
A 7-dimensional polyhedron in QQ^8 defined as the convex hull of 9 vertices

sage: sf.vertices()
(A vertex at (-4, -4, -4, -4, 17/4, 17/4, 17/4, 17/4),
 A vertex at (0, 0, 0, 0, 0, 0, 0, 1),
 A vertex at (0, 0, 0, 0, 0, 1, 0, 0),
(continues on next page)
```
It is possible to stack on unbounded faces:

```
sage: Q = Polyhedron(vertices=[[0,1],[1,0]],rays=[[1,1]])
sage: E = Q.faces(1)
sage: Q.stack(E[0],1/2).Vrepresentation()
[(A vertex at (0, 1),
  A vertex at (1, 0),
  A ray in the direction (1, 1),
  A vertex at (2, 0))]
sage: Q.stack(E[1],1/2).Vrepresentation()
[(A vertex at (0, 1),
  A vertex at (0, 2),
  A vertex at (1, 0),
  A ray in the direction (1, 1))]
sage: Q.stack(E[2],1/2).Vrepresentation()
[(A vertex at (0, 0),
  A vertex at (0, 1),
  A vertex at (1, 0),
  A ray in the direction (1, 1))]
```

Stacking requires a proper face:

```
sage: Q.stack(Q.faces(2)[0])
Traceback (most recent call last):
...
ValueError: can only stack on proper face
```

`subdirect_sum(other)`
Return the subdirect sum of `self` and `other`.

The subdirect sum of two polyhedron is a projection of the join of the two polytopes. It is obtained by placing the two objects in orthogonal subspaces intersecting at the origin.

INPUT:

- `other` – a `Polyhedron_base`

EXAMPLES:

```
sage: P1 = Polyhedron([[1],[2]], base_ring=ZZ)
sage: P2 = Polyhedron([[3],[4]], base_ring=QQ)
sage: sds = P1.subdirect_sum(P2);sds
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 4 vertices
sage: sds.vertices()
[(A vertex at (0, 3),
  A vertex at (0, 4),
  A vertex at (1, 0),
  A vertex at (1, 1))]
```
A vertex at (1, 0),
A vertex at (2, 0)

See also:
join() direct_sum()

tikz(view=[0, 0, 1], angle=0, scale=1, edge_color='blue!95!black', facet_color='blue!95!black',
opacity=0.8, vertex_color='green', axis=False)
Return a string tikz_pic consisting of a tikz picture of self according to a projection view and an angle
angle obtained via the threejs viewer.

INPUT:
• view - list (default: [0,0,1]) representing the rotation axis (see note below).
• angle - integer (default: 0) angle of rotation in degree from 0 to 360 (see note below).
• scale - integer (default: 1) 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: This is a wrapper of a method of the projection object self.projection(). See tikz() for more
detail.

The inputs view and angle can be obtained by visualizing it using .show(aspect_ratio=1). This will
open an interactive view in your default browser, where you can rotate the polytope. Once the desired view
angle is found, click on the information icon in the lower right-hand corner and select Get Viewpoint. This
will copy a string of the form \[x,y,z,angle\] to your local clipboard. Go back to Sage and type
\[\text{Img} = \text{P.tikz((x,y,z),angle)}\].

The inputs view and angle can also be obtained from the viewer Jmol:

1) Right click on the image
2) Select ``Console``
3) Select the tab `State``
4) Scroll to the line `moveto``

It reads something like:

```
moveto 0.0 {x y z angle} Scale
```

The view is then \[x,y,z\] and angle is angle. The following number is the scale.
Jmol performs a rotation of angle degrees along the vector \[x,y,z\] and show the result from the z-axis.

EXAMPLES:
sage: co = polytopes.cuboctahedron()
sage: Img = co.tikz([0,0,1], 0)
sage: print('
'.join(Img.splitlines()[:9]))
\begin{tikzpicture}%
  [x={(1.000000cm, 0.000000cm)},
  y={(0.000000cm, 1.000000cm)},
  z={(0.000000cm, 0.000000cm)},
  scale=1.000000,
  back/.style={loosely dotted, thin},
  edge/.style={color=blue!95!black, thick},
  facet/.style={fill=blue!95!black, fill opacity=0.800000},
  vertex/.style={inner sep=1pt,circle,draw=green!25!black, fill=green!75!black, thick}]
sage: print('
'.join(Img.splitlines()[12:21]))
%% with the command: ._tikz_3d_in_3d and parameters:
%% view = [0, 0, 1]
%% angle = 0
%% scale = 1
%% edge_color = blue!95!black
%% facet_color = blue!95!black
%% opacity = 0.8
%% vertex_color = green
%% axis = False
sage: print('
'.join(Img.splitlines()[22:26]))
%% Coordinate of the vertices:
%%
\coordinate (-1.00000, -1.00000, 0.00000) at (-1.00000, -1.00000, 0.00000);
\coordinate (-1.00000, 0.00000, -1.00000) at (-1.00000, 0.00000, -1.00000);

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:

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:

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:

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:

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:

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:

```
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)
```

Return a triangulation of the polytope.

INPUT:

- engine – either ‘auto’ (default), ‘internal’, ‘TOPCOM’, or ‘normaliz’. The ‘internal’ and ‘TOPCOM’ instruct this package to always use its own triangulation algorithms or TOPCOM’s algorithms, respectively. By default (‘auto’), TOPCOM is used if it is available and internal routines otherwise.

The remaining keyword parameters are passed through to the `PointConfiguration` constructor:

- connected – boolean (default: True). Whether the triangulations should be connected to the regular triangulations via bistellar flips. These are much easier to compute than all triangulations.
- fine – boolean (default: False). Whether the triangulations must be fine, that is, make use of all points of the configuration.
- regular – boolean or None (default: None). Whether the triangulations must be regular. A regular triangulation is one that is induced by a piecewise-linear convex support function. In other words, the shadows of the faces of a polyhedron in one higher dimension.
  - True: Only regular triangulations.
  - False: Only non-regular triangulations.
  - None (default): Both kinds of triangulation.
- star – either None (default) or a point. Whether the triangulations must be star. A triangulation is star if all maximal simplices contain a common point. The central point can be specified by its index (an integer) in the given points or by its coordinates (anything iterable.)

OUTPUT:

A triangulation of the convex hull of the vertices as a `Triangulation`. The indices in the triangulation correspond to the `Vrepresentation()` objects.

EXAMPLES:

```
sage: cube = polytopes.hypercube(3)
sage: triangulation = cube.triangulate(....: engine='internal') # to make doctest independent of TOPCOM
sage: triangulation
(<0,1,2,7>, <0,1,5,7>, <0,2,3,7>, <0,3,4,7>, <0,4,5,7>, <1,5,6,7>)
sage: simplex_indices = triangulation[0]; simplex_indices
(0, 1, 2, 7)
sage: simplex_vertices = [ cube.Vrepresentation(i) for i in simplex_indices ]
sage: simplex_vertices
[A vertex at (1, -1, -1),
 A vertex at (1, 1, -1),
 A vertex at (1, 1, 1),
 A vertex at (-1, 1, 1)]
```

(continues on next page)
It is possible to use 'normaliz' as an engine. For this, the polyhedron should have the backend set to normaliz:

```
sage: P = Polyhedron(vertices=[[0,0,1],[1,0,1],[0,1,1],[1,1,1]],backend='normaliz')  # optional - pynormaliz
sage: P.triangulate(engine='normaliz')  # optional - pynormaliz
(<0,1,2>, <1,2,3>)
sage: P = Polyhedron(vertices=[[0,0,1],[1,0,1],[0,1,1],[1,1,1]])
sage: P.triangulate(engine='normaliz')
Traceback (most recent call last):
  ...
TypeError: the polyhedron's backend should be 'normaliz'
```

The normaliz engine can triangulate pointed cones:

```
sage: C1 = Polyhedron(rays=[[0,0,1],[1,0,1],[0,1,1],[1,1,1]],backend='normaliz')  # optional - pynormaliz
sage: C1.triangulate(engine='normaliz')  # optional - pynormaliz
(<0,1,2>, <1,2,3>)
sage: C2 = Polyhedron(rays=[[1,0,1],[0,0,1],[0,1,1],[1,1,10/9]],backend='normaliz')  # optional - pynormaliz
sage: C2.triangulate(engine='normaliz')  # optional - pynormaliz
(<0,1,2>, <1,2,3>)
```

They can also be affine cones:

```
sage: K = Polyhedron(vertices=[[1,1,1]],rays=[[1,0,0],[0,1,0],[1,1,-1],[1,1,1]],backend='normaliz')  # optional - pynormaliz
sage: K.triangulate(engine='normaliz')  # optional - pynormaliz
(<0,1,2>, <0,1,3>)
```

`truncation(cut_frac=None)`

Return a new polyhedron formed from two points on each edge between two vertices.

**INPUT:**

- `cut_frac` – integer, how deeply to cut into the edge. Default is $\frac{1}{3}$.

**OUTPUT:**

A Polyhedron object, truncated as described above.

**EXAMPLES:**

```
sage: cube = polytopes.hypercube(3)
sage: trunc_cube = cube.truncation()
sage: trunc_cube.n_vertices()
24
sage: trunc_cube.n_inequalities()
14
```
**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 (0,0,0))
sage: half_plane.vertex_adjacency_matrix()
[[0 0 1]
 [0 0 0]
 [1 0 0]]
```

In one dimension higher, that is for a half-space in 3 dimensions, there is no one-dimensional face. Hence nothing is adjacent:

```
sage: Polyhedron(ieqs=[(0,1,0,0)]).vertex_adjacency_matrix()
[[0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]
 [0 0 0 0]]
```

**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("{} {}\).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:

```python
sage: P = Polyhedron(vertices=[(0, 1), (1, 0), (3, 0), (4, 1)],
....: rays=[(0,1)])
sage: for v in P.Vrep_generator():
    print("{} {}\).format(P.adjacency_matrix().row(v.index()), v))
(0, 1, 0, 0, 1) A ray in the direction (0, 1)
(1, 0, 1, 0, 0) A vertex at (0, 1)
(0, 1, 0, 1, 0) A vertex at (1, 0)
(0, 0, 1, 0, 1) A vertex at (3, 0)
(1, 0, 0, 1, 0) A vertex at (4, 1)
```

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:

```python
sage: P = Polyhedron(vertices=[(0, 1), (1, 0), (3, 0), (4, 1)],
....: rays=[(0,1), (1,1)])
sage: for v in P.Vrep_generator():
    print("{} {}\).format(P.adjacency_matrix().row(v.index()), v))
(0, 1, 0, 0, 1) A ray in the direction (0, 1)
(1, 0, 1, 0, 0) A vertex at (0, 1)
(0, 1, 0, 0, 1) A vertex at (1, 0)
(0, 0, 0, 0, 1) A ray in the direction (1, 1)
(0, 0, 1, 1, 0) A vertex at (3, 0)
(1, 0, 0, 1, 0) A vertex at (4, 1)
```

The vertex adjacency matrix has base ring integers. This way one can express various counting questions:

```python
sage: P = polytopes.cube()
sage: Q = P.stack(P.faces(2)[0])
sage: M = Q.vertex_adjacency_matrix()
sage: sum(M)
(4, 4, 3, 3, 4, 4, 3, 3)
sage: G = Q.vertex_graph()
sage: G.degree()
[4, 4, 3, 3, 4, 4, 3, 3]
```

`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: \( f(v) = v \cdot \text{dot_product}(f) \).

- 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 a directed edge from a vertex to a face if and only if the vertex is incident to the face.

**INPUT:**

- labels – boolean (default: True); decide how the nodes of the graph are labelled. Either with the original vertices/facets of the Polyhedron or with integers.

**OUTPUT:**

- a bipartite DiGraph. If `labels` is True, then the nodes of the graph will actually be the vertices and facets of `self`, otherwise they will be integers.

See also:

`combinatorial_automorphism_group()`, `is_combinatorially_isomorphic()`.

**EXAMPLES:**

```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),
   ... (continues on next page)
```
A vertex at (-1, -1, 1),
A vertex at (-1, 1, -1),
A vertex at (-1, 1, 1),
A vertex at (1, -1, -1),
A vertex at (1, -1, 1),
A vertex at (1, 1, -1),
A vertex at (1, 1, 1),
An inequality (-1, 0, 0) x + 1 >= 0,
An inequality (0, -1, 0) x + 1 >= 0,
An inequality (0, 0, -1) x + 1 >= 0,
An inequality (0, 0, 1) x + 1 >= 0,
An inequality (0, 1, 0) x + 1 >= 0,
An inequality (1, 0, 0) x + 1 >= 0.

$sage$: G.automorphism_group().is_isomorphic(P.hasse_diagram().automorphism_group())
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
$sage$: G2 = copy(G)
$sage$: G2.reverse_edges(G2.edges())
$sage$: G2.is_isomorphic(H)
True

vertex_generator()

Return a generator for the vertices of the polyhedron.

Warning: If the polyhedron has lines, return a generator for the vertices of the Vrepresentation. However, the represented polyhedron has no 0-dimensional faces (i.e. vertices):

$sage$: P = Polyhedron(rays=[[1,0,0]],lines=[[0,1,0]])
$sage$: list(P.vertex_generator())
[A vertex at (0, 0, 0)]
$sage$: P.faces(0)
[]

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.

..NOTE:
The graph of a polyhedron with lines has no vertices, as the polyhedron has no vertices ('0'-faces).

The method :meth:`Polyhedron_base:vertices` returns the defining points in this case.

EXAMPLES:

sage: g3 = polytopes.hypercube(3).vertex_graph(); g3
Graph on 8 vertices
sage: g3.automorphism_group().cardinality()
48
sage: s4 = polytopes.simplex(4).vertex_graph(); s4
Graph on 5 vertices
sage: s4.is_eulerian()
True

The graph of an unbounded polyhedron is the graph of the bounded complex:

sage: open_triangle = Polyhedron(vertices=[[1,0], [0,1]],
    ....:     rays=[[1,1]])

sage: open_triangle.vertex_graph()
Graph on 2 vertices

The graph of a polyhedron with lines has no vertices:

sage: line = Polyhedron(lines=[[0,1]])
sage: line.vertex_graph()
Graph on 0 vertices

vertices()
Return all vertices of the polyhedron.

OUTPUT:
A tuple of vertices.
Warning: If the polyhedron has lines, return the vertices of the \texttt{Vrepresentation}. However, the represented polyhedron has no 0-dimensional faces (i.e. vertices):

```
sage: P = Polyhedron(rays=[[1,0,0]],lines=[[0,1,0]])
sage: P.vertices()
(A vertex at (0, 0, 0),)
sage: P.faces(0)
()
```

EXAMPLES:

```
sage: triangle = Polyhedron(verteces=[[1,0],[0,1],[1,1]])
sage: triangle.vertices()
(A vertex at (0, 1), A vertex at (1, 0), A vertex at (1, 1))
sage: a_simplex = Polyhedron(ieqs = [
.....: [0,1,0,0],[0,0,1,0],[0,0,0,1]
.....: ], eqns = [[1,-1,-1,-1,-1]])
sage: a_simplex.vertices()
(A vertex at (1, 0, 0, 0), A vertex at (0, 1, 0, 0),
A vertex at (0, 0, 1, 0), A vertex at (0, 0, 0, 1))
```

\texttt{vertices\_list()}

Return a list of vertices of the polyhedron.

Note: It is recommended to use \texttt{vertex\_generator()} instead to iterate over the list of \texttt{Vertex} objects.

```
sage: P = Polyhedron(rays=[[1,0,0]],lines=[[0,1,0]])
sage: P.vertices_list()
[[0, 0, 0]]
sage: P.faces(0)
()
```

EXAMPLES:

```
sage: triangle = Polyhedron(verteces=[[1,0],[0,1],[1,1]])
sage: triangle.vertices_list()
[[0, 1], [1, 0], [1, 1]]
sage: a_simplex = Polyhedron(ieqs = [
.....: [0,1,0,0],[0,0,1,0],[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.

Warning: If the polyhedron has lines, return the coordinates of the vertices of the Vrepresentation. However, the represented polyhedron has no 0-dimensional faces (i.e. vertices):

```
sage: P = Polyhedron(rays=[[1,0,0]], lines=[[0,1,0]])
sage: P.vertices_matrix()
[0]
[0]
[0]
sage: P.faces(0)
()```

EXAMPLES:

```
sage: triangle = Polyhedron(vertices=[[1,0],[0,1],[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(measure='ambient', engine='auto', **kwds)
Return the volume of the polytope.

INPUT:

• measure – string. The measure to use. Allowed values are:
  – ambient (default): Lebesgue measure of ambient space (volume)
  – induced: Lebesgue measure of the affine hull (relative volume)
  – induced_rational: Scaling of the Lebesgue measure for rational polytopes, such that the unit hypercube has volume 1
  – induced_lattice: Scaling of the Lebesgue measure, such that the volume of the hypercube is factorial(n)

• engine – string. The backend to use. Allowed values are:
  – 'auto' (default): choose engine according to measure
- 'internal': see `triangulate()`
- 'TOPCOM': see `triangulate()`
- 'lrs': use David Avis's lrs program (optional)
- 'latte': use LattE integrale program (optional)
- 'normaliz': use Normaliz program (optional)

- **kwds** – keyword arguments that are passed to the triangulation engine

OUTPUT:
The volume of the polytope

EXAMPLES:

```python
sage: polytopes.hypercube(3).volume()
8
sage: (polytopes.hypercube(3)*2).volume()
64
sage: polytopes.twenty_four_cell().volume()
2
```

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:

```python
sage: I3 = polytopes.hypercube(3)
sage: I3.volume(engine='lrs') # optional - lrslib
8.0
sage: C24 = polytopes.twenty_four_cell()
sage: C24.volume(engine='lrs') # optional - lrslib
2.0
```

If the base ring is exact, the answer is exact:

```python
sage: P5 = polytopes.regular_polygon(5)
sage: P5.volume()
2.377641290737884?
sage: polytopes.icosahedron().volume()
5/12*sqrt5 + 5/4
sage: numerical_approx(_)
# abs tol 1e9
2.18169499062491
```

When considering lower-dimensional polytopes, we can ask for the ambient (full-dimensional), the induced measure (of the affine hull) or, in the case of lattice polytopes, for the induced rational measure. This is controlled by the parameter `measure`. Different engines may have different ideas on the definition of volume of a lower-dimensional object:

```python
sage: P = Polyhedron([[0, 0], [1, 1]])
sage: P.volume()
0
sage: P.volume(measure='induced')
1.414213562373095?
sage: P.volume(measure='induced_rational') # optional -- latte_int
1
```

(continues on next page)
sage: S = polytopes.regular_polygon(6); S
A 2-dimensional polyhedron in AA^2 defined as the convex hull of 6 vertices
sage: edge = S.faces(1)[4].as_polyhedron()
(A vertex at (0.866025403784439?, 1/2), A vertex at (0, 1))

sage: edge.vertices()
(A vertex at (0.866025403784439?, 0.5000000000000000), A vertex at (0, 1))

sage: edge.volume()
0
sage: edge.volume(measure='induced')
1

sage: P = Polyhedron(backend='normaliz',vertices=[[1,0,0],[0,0,1],[-1,1,1],[-1,2,0]])
# optional - pynormaliz
sage: P.volume()  # optional - pynormaliz
0
sage: P.volume(measure='induced')  # optional - pynormaliz
2.5980762113533167
sage: P.volume(measure='induced',engine='normaliz')  # optional - pynormaliz
2.5980762113533167
sage: P.volume(measure='induced_rational')  # optional - pynormaliz, latte_int
3/2
sage: P.volume(measure='induced_rational',engine='normaliz')  # optional - pynormaliz
3/2
sage: P.volume(measure='induced_lattice')  # optional - pynormaliz
3

The same polytope without normaliz backend:

sage: P = Polyhedron(backend='normaliz',vertices=[[1,0,0],[0,0,1],[-1,1,1],[-1,2,0]])
# optional - pynormaliz
sage: P.volume(measure='induced_lattice',engine='latte')  # optional - latte_int
3
sage: Dexact = polytopes.dodecahedron()
sage: v = Dexact.faces(2)[0].as_polyhedron().volume(measure='induced', engine='internal'); v
1.53406271079097?
sage: RDF(v)  # abs tol 1e-9
1.53406271079097?

sage: Dinexact = polytopes.dodecahedron(exact=False)
sage: w = Dinexact.faces(2)[2].as_polyhedron().volume(measure='induced', engine='internal'); RDF(w)  # abs tol 1e-9
1.5340627082974878

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
The volume of a full-dimensional unbounded polyhedron is infinity:

```
sage: P = Polyhedron(vertices=[[1, 0], [0, 1]], rays=[[1, 1]])
sage: P.volume()
+Infinity
```

The volume of a non full-dimensional unbounded polyhedron depends on the measure used:

```
sage: P = Polyhedron(ieqs=[[1,1,1],[-1,-1,-1],[3,1,0]]); P
A 1-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 1 ray
sage: P.volume()
0
sage: P.volume(measure='induced')
0
sage: P.volume(measure='ambient')
0
sage: P.volume(measure='induced_rational')  # optional - pynormaliz
+Infinity
sage: P.volume(measure='induced_rational',engine='latte')  # optional - latte_int
+Infinity
```

The volume in 0-dimensional space is taken by counting measure:

```
sage: P = Polyhedron(vertices=[]); P
A 0-dimensional polyhedron in ZZ^0 defined as the convex hull of 1 vertex
sage: P.volume()
1
```

The empty polyhedron in ZZ^0:

```
sage: P = Polyhedron(vertices=[]); P
The empty polyhedron in ZZ^0
sage: P.volume()
0
```
wedge(face, width=1)

Return the wedge over a face of the polytope self.

The wedge over a face $F$ of a polytope $P$ with width $w \neq 0$ is defined as:

$$(P \times \mathbb{R}) \cap \{a^\top x + |wx_d+1| \leq b\}$$

where $\{x|a^\top x = b\}$ is a supporting hyperplane defining $F$.

INPUT:
- face – a PolyhedronFace of self, the face which we take the wedge over
- width – a nonzero number (default: 1); specifies how wide the wedge will be

OUTPUT:
A (bounded) polyhedron

EXAMPLES:

```
sage: P_4 = polytopes.regular_polygon(4)
sage: W1 = P_4.wedge(P_4.faces(1)[0]); W1
A 3-dimensional polyhedron in AA^3 defined as the convex hull of 6 vertices
sage: triangular_prism = polytopes.regular_polygon(3).prism()
sage: W1.is_combinatorially_isomorphic(triangular_prism)
True
sage: Q = polytopes.hypersimplex(4,2)
sage: W2 = Q.wedge(Q.faces(2)[7]); W2
A 4-dimensional polyhedron in QQ^5 defined as the convex hull of 9 vertices
sage: W2.vertices()
(A vertex at (1, 1, 0, 0, 1),
 A vertex at (1, 1, 0, 0, -1),
 A vertex at (1, 0, 1, 0, 1),
 A vertex at (1, 0, 1, 0, -1),
 A vertex at (1, 0, 0, 1, 1),
 A vertex at (1, 0, 0, 1, -1),
 A vertex at (0, 1, 1, 0, 0),
 A vertex at (0, 0, 1, 1, 0))
sage: W3 = Q.wedge(Q.faces(1)[11]); W3
A 4-dimensional polyhedron in QQ^5 defined as the convex hull of 10 vertices
sage: W3.vertices()
(A vertex at (1, 1, 0, 0, -2),
 A vertex at (1, 1, 0, 0, 2),
 A vertex at (1, 0, 1, 0, -2),
 A vertex at (1, 0, 1, 0, 2),
 A vertex at (1, 0, 0, 1, 1),
 A vertex at (1, 0, 0, 1, -1),
 A vertex at (0, 1, 1, 0, 0),
 A vertex at (0, 1, 1, 0, 1))
sage: C_3_7 = polytopes.cyclic_polytope(3,7)
```

(continues on next page)
sage: P_6 = polytopes.regular_polygon(6)
sage: W4 = P_6.wedge(P_6.faces(1)[0])
sage: W4.is_combinatorially_isomorphic(C_3_7.polar())
True

REFERENCES:
For more information, see Chapter 15 of [HoDaCG17].

write_cdd_Hrepresentation(filename)
Export the polyhedron as a H-representation to a file.

INPUT:
• filename – the output file.

See also:
cdd_Hrepresentation() – return the H-representation of the polyhedron as a string.

EXAMPLES:

```
sage: from sage.misc.temporary_file import tmp_filename
sage: filename = tmp_filename(ext='.ext')
sage: polytopes.cube().write_cdd_Hrepresentation(filename)
```

write_cdd_Vrepresentation(filename)
Export the polyhedron as a V-representation to a file.

INPUT:
• filename – the output file.

See also:
cdd_Vrepresentation() – return the V-representation of the polyhedron as a string.

EXAMPLES:

```
sage: from sage.misc.temporary_file import tmp_filename
sage: filename = tmp_filename(ext='.ext')
sage: polytopes.cube().write_cdd_Vrepresentation(filename)
```

sage.geometry.polyhedron.base.is_Polyhedron(X)
Test whether X is a Polyhedron.

INPUT:
• X – anything.

OUTPUT:
Boolean.

EXAMPLES:

```
sage: p = polytopes.hypercube(2)
sage: from sage.geometry.polyhedron.base import is_Polyhedron
sage: is_Polyhedron(p)
True
```

(continues on next page)
2.5.2 Base class for polyhedra over $\mathbb{Q}$

class sage.geometry.polyhedron.base_QQ.Polyhedron_QQ(
    parent, Vrep, Hrep, Vrep_minimal=None,
    Hrep_minimal=None, pref_rep=None,
    mutable=False, **kwds)

Bases: sage.geometry.polyhedron.base.Polyhedron_base

Base class for Polyhedra over $\mathbb{Q}$

ehrhart_polynomial(engine=None, variable='t', verbose=False, dual=False, irrational_primal=None,
    irrational_all_primal=None, maxdet=None, no_decomposition=None,
    compute_vertex_cones=None, smith_form=None, dualization=None,
    triangulation=None, triangulation_max_height=None, **kwds)

Return the Ehrhart polynomial of this polyhedron.

The polyhedron must be a lattice polytope. Let $P$ be a lattice polytope in $\mathbb{R}^d$ 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 Ehrhart polynomial of $P$. For more information see the Wikipedia article Ehrhart_polynomial. The Ehrhart polynomial may be computed using either LattE Integrale or Normaliz by setting engine to ‘latte’ or ‘normaliz’ respectively.

**INPUT:**

- **engine** – string; The backend to use. Allowed values are:
  - None (default); When no input is given the Ehrhart polynomial is computed using LattE Integrale (optional)
  - ‘latte’; use LattE integrale program (optional)
  - ‘normaliz’; use Normaliz program (optional package pynormaliz). The backend of self must be set to ‘normaliz’.
- **variable** – string (default: ‘t’); The variable in which the Ehrhart polynomial should be expressed.
- When the engine is ‘latte’, the additional input values are:
  - **verbose** - boolean (default: False); If True, print the whole output of the LattE command.
- dualization - string; either 'cdd' or '4ti2'
- triangulation - string; 'cddlib', '4ti2' or 'topcom'
- triangulation_max_height - integer; use a uniform distribution of height from 1 to this number

OUTPUT:

A univariate polynomial in variable over a rational field.

See also:

latte the interface to LattE Integrale PyNormaliz

EXAMPLES:

To start, we find the Ehrhart polynomial of a three-dimensional simplex, first using engine='latte'. Leaving the engine unspecified sets the engine to 'latte' by default:

```python
 sage: simplex = Polyhedron(vertices=[(0,0,0),(3,3,3),(-3,2,1),(1,-1,-2)])
 sage: simplex = simplex.change_ring(QQ)
 sage: poly = simplex.ehrhart_polynomial(engine='latte')  # optional - latte_int
 sage: poly
 7/2*t^3 + 2*t^2 - 1/2*t + 1
 sage: poly(1)  # optional - latte_int
 6
 sage: len(simplex.integral_points())  # optional - latte_int
 6
 sage: poly(2)  # optional - latte_int
 36
 sage: len((2*simplex).integral_points())  # optional - latte_int
 36
```

Now we find the same Ehrhart polynomial, this time using engine='normaliz'. To use the Normaliz engine, the simplex must be defined with backend='normaliz':

```python
 sage: simplex = Polyhedron(vertices=[(0,0,0),(3,3,3),(-3,2,1),(1,-1,-2)], backend='normaliz')  # optional - pynormaliz
 sage: simplex = simplex.change_ring(QQ)  # optional - pynormaliz
 sage: poly = simplex.ehrhart_polynomial(engine='normaliz')  # optional - pynormaliz
 sage: poly
 7/2*t^3 + 2*t^2 - 1/2*t + 1
```

If the engine='normaliz', the backend should be 'normaliz', otherwise it returns an error:

```python
 sage: simplex = Polyhedron(vertices=[(0,0,0),(3,3,3),(-3,2,1),(1,-1,-2)])
 sage: simplex = simplex.change_ring(QQ)
 sage: poly = simplex.ehrhart_polynomial(engine='normaliz')  # optional - pynormaliz
 Traceback (most recent call last):
 ... TypeError: The backend of the polyhedron should be 'normaliz'
```

The polyhedron should be compact:
The polyhedron should have integral vertices:

```python
sage: L = Polyhedron(backend='normaliz',vertices=[[0],[1/2]])
sage: L.ehrhart_polynomial()
```

```python
Traceback (most recent call last):
... ValueError: Ehrhart polynomial only defined for compact polyhedra
```

The polyhedron should have integral vertices:

```python
sage: L = Polyhedron(backend='normaliz',vertices=[[0],[1/2]])
sage: L.ehrhart_quasipolynomial()
```

```python
Traceback (most recent call last):
... TypeError: the polytope has nonintegral vertices, use ehrhart_quasipolynomial
```

`ehrhart_quasipolynomial`

```python
ehrhart_quasipolynomial (variable='t', engine=None, verbose=False, dual=None, irrational_primal=None, irrational_all_primal=None, maxdet=None, no_decomposition=None, compute_vertex_cones=None, smith_form=None, dualization=None, triangulation=None, triangulation_max_height=None, **kwds)
```

Compute the Ehrhart quasipolynomial of this polyhedron with rational vertices.

If the polyhedron is a lattice polytope, returns the Ehrhart polynomial, a univariate polynomial in variable over a rational field. If the polyhedron has rational, nonintegral vertices, returns a tuple of polynomials in variable over a rational field. The Ehrhart counting function of a polytope \( P \) with rational vertices is given by a quasipolynomial. That is, there exists a positive integer \( l \) and \( l \) polynomials \( ehr_{P,i} \) for \( i \in \{1, \ldots, l\} \) such that if \( t \) is equivalent to \( i \) mod \( l \) then \( tP \cap \mathbb{Z}^d = ehr_{P,i}(t) \).

**INPUT:**

- `variable` – string (default: ‘\( t \)’); The variable in which the Ehrhart polynomial should be expressed.
- `engine` – string; The backend to use. Allowed values are:
  - `None` (default); When no input is given the Ehrhart polynomial is computed using Normaliz (optional)
  - ‘latte’; use LattE Integrale program (requires optional package ‘latte_int’)  
  - ‘normaliz’; use the Normaliz program (requires optional package ‘pynormaliz’). The backend of `self` must be set to ‘normaliz’.
- When the engine is ‘latte’, the additional input values are:
  - `verbose` - boolean (default: False); If True, print the whole output of the LattE command.

The following options are passed to the LattE command, for details consult the LattE documentation:

- `dual` - boolean; triangulate and signed-decompose in the dual space
- `irrational_primal` - boolean; triangulate in the dual space, signed-decompose in the primal space using irrationalization.
- `irrational_all_primal` - boolean; triangulate and signed-decompose in the primal space using irrationalization.
– maxdet – integer; decompose down to an index (determinant) of maxdet instead of index 1 (unimodular cones).
– no_decomposition – boolean; do not signed-decompose simplicial cones.
– compute_vertex_cones – string; either ‘cdd’ or ‘lrs’ or ‘4ti2’
– smith_form – string; either ‘ilio’ or ‘lidia’
– dualization – string; either ‘cdd’ or ‘4ti2’
– triangulation – string; ‘cddlib’, ‘4ti2’ or ‘topcom’
– triangulation_max_height – integer; use a uniform distribution of height from 1 to this number

OUTPUT:
A univariate polynomial over a rational field or a tuple of such polynomials.

See also:
latte the interface to LattE Integrale PyNormaliz

Warning: If the polytope has rational, non integral vertices, it must have backend='normaliz'.

EXAMPLES:
As a first example, consider the line segment [0,1/2]. If we dilate this line segment by an even integral factor \( k \), then the dilated line segment will contain \( k/2 + 1 \) lattice points. If \( k \) is odd then there will be \( k/2 + 1/2 \) lattice points in the dilated line segment. Note that it is necessary to set the backend of the polytope to ‘normaliz’:

```
sage: line_seg = Polyhedron(vertices=[[0],[1/2]],backend='normaliz') # optional
    # pynormaliz
sage: line_seg
    # optional
    # pynormaliz
A 1-dimensional polyhedron in QQ^1 defined as the convex hull of 2 vertices
sage: line_seg.ehrhart_quasipolynomial() # optional
    # pynormaliz
(1/2*t + 1, 1/2*t + 1/2)
```

For a more exciting example, let us look at the subpolytope of the 3 dimensional permutahedron fixed by the reflection across the hyperplane \( x_1 = x_4 \):

```
sage: verts = [[3/2, 3, 4, 3/2],
     ....: [3/2, 4, 3, 3/2],
     ....: [5/2, 1, 4, 5/2],
     ....: [5/2, 4, 1, 5/2],
     ....: [7/2, 1, 2, 7/2],
     ....: [7/2, 2, 1, 7/2]]
sage: subpoly = Polyhedron(verts=verts, backend='normaliz') # optional
    # pynormaliz
sage: eq = subpoly.ehrhart_quasipolynomial() # optional
    # pynormaliz
(4*t^2 + 3*t + 1, 4*t^2 + 2*t)
sage: eq = subpoly.ehrhart_quasipolynomial() # optional
    # pynormaliz
(continues on next page)```
(continued from previous page)

\[(4t^2 + 3t + 1, 4t^2 + 2t)\]

\[
sage: even_ep = eq[0] \quad \# \text{optional - pynormaliz}
\]

\[
sage: odd_ep = eq[1] \quad \# \text{optional - pynormaliz}
\]

\[
sage: even_ep(2) \quad \# \text{optional - pynormaliz}
\]

\[23\]

\[
sage: ts = 2\times\text{subpoly} \quad \# \text{optional - pynormaliz}
\]

\[
sage: ts.\text{integral_points_count()} \quad \# \text{optional - pynormaliz \ latte_}
\]

\[\rightarrow \text{int}\]

\[23\]

\[
sage: odd_ep(1) \quad \# \text{optional - pynormaliz}
\]

\[6\]

\[
sage: \text{subpoly.\text{integral_points_count()}} \quad \# \text{optional - pynormaliz \ latte_}
\]

\[\rightarrow \text{int}\]

\[6\]

A polytope with rational nonintegral vertices must have backend='normaliz':

\[
sage: \text{line}_\text{seg} = \text{Polyhedron(\text{vertices}=[[0],[1/2]])}
\]

\[
sage: \text{line}_\text{seg}.\text{ehrhart_quasipolynomial()}
\]

Traceback (most recent call last):
...
TypeError: The backend of the polyhedron should be 'normaliz'

The polyhedron should be compact:

\[
sage: C = \text{Polyhedron(backend='normaliz',rays=[[1/2,1],[2,1]])} \quad \# \text{optional - pynormaliz}
\]

\[
sage: C.\text{ehrhart_quasipolynomial()} \quad \# \text{optional - pynormaliz}
\]

Traceback (most recent call last):
...
ValueError: Ehrhart quasipolynomial only defined for compact polyhedra

If the polytope happens to be a lattice polytope, the Ehrhart polynomial is returned:

\[
sage: \text{simplex} = \text{Polyhedron(\text{vertices}=[[0,0,0],[3,3,3],[-3,2,1],[1,-1,-2]],\text{backend='normaliz'})} \quad \# \text{optional - pynormaliz}
\]

\[
sage: \text{simplex} = \text{simplex.\text{change\_ring(QQ)}} \quad \# \text{optional - pynormaliz}
\]

\[
sage: \text{poly} = \text{simplex.\text{ehrhart\_quasipolynomial(\text{engine='normaliz'})}} \quad \# \text{optional - pynormaliz}
\]

\[
sage: \text{poly} \quad \# \text{optional - pynormaliz}
\]

\[7/2t^3 + 2t^2 - 1/2t + 1\]

\[
sage: \text{simplex.\text{ehrhart\_polynomial()}} \quad \# \text{optional - pynormaliz \ latte\_int}
\]

\[7/2t^3 + 2t^2 - 1/2t + 1\]

\[
\text{integral\_points\_count(\text{verbose=False, use\_Hrepresentation=False,}}
\]

\[\text{explicit\_enumeration\_threshold=1000, preprocess=True, **kwds})\]

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 \texttt{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:

\texttt{lattice} the interface to \texttt{LattE} interfaces

EXAMPLES:

\begin{verbatim}
sage: P = polytopes.cube()
sage: P.integral_points_count()
27
sage: P.integral_points_count(explicit Enumeration_threshold=0) # optional - \texttt{latte_int}
27
\end{verbatim}

We enlarge the polyhedron to force the use of the generating function methods implemented in \texttt{LattE integrale}, rather than explicit enumeration.

\begin{verbatim}
sage: (1000000000*P).integral_points_count(\texttt{verbose=True}) # optional - \texttt{latte_int} This is \texttt{LattE integrale}... ... Total time:... 8000000012000000006000000001
\end{verbatim}

We shrink the polyhedron a little bit:

\begin{verbatim}
sage: Q = P*(8/9)
sage: Q.integral_points_count()
1
sage: Q.integral_points_count(explicit Enumeration_threshold=0) # optional - \texttt{latte_int}
1
\end{verbatim}

Unbounded polyhedra (with or without lattice points) are not supported:

\begin{verbatim}
sage: P = Polyhedron(\texttt{vertices=[[1/2, 1/3]], \texttt{rays=[[1, 1]]})
sage: P.integral_points_count()
Traceback (most recent call last):
... NotImplementedError: ...
sage: P = Polyhedron(\texttt{vertices=[[1, 1]], \texttt{rays=[[1, 1]]})
sage: P.integral_points_count()
Traceback (most recent call last):
... NotImplementedError: ...
\end{verbatim}

“Fibonacci” knapsacks (preprocessing helps a lot):

\begin{verbatim}
sage: \texttt{def fibonacci\_knapsack}(d, b, backend=\texttt{None}):
....: \texttt{lp} = \texttt{MixedIntegerLinearProgram}(base_ring=\texttt{QQ})
....: \texttt{x} = \texttt{lp.new\_variable}(nonnegative=\texttt{True})
....: \texttt{lp.add\_constraint(lp.sum(fibonacci(i+3)*x[i] \textbf{for i in range(d)}) <= b)
....: \texttt{return lp.polyhedron(backend=backend)}
\end{verbatim}

(continues on next page)
2.5.3 Base class for polyhedra over $\mathbb{Z}$

class sage.geometry.polyhedron.base_ZZ.Polyhedron_ZZ(parent, Vrep, Hrep, Vrep_minimal=None, Hrep_minimal=None, pref_rep=None, mutable=False, **kwds)

Bases: sage.geometry.polyhedron.base_QQ.Polyhedron_QQ
Base class for Polyhedra over $\mathbb{Z}$

ehrhart_polynomial(engine=None, variable='t', verbose=False, dual=None, irrational_primal=None, irrational_all_primal=None, maxdet=None, no_decomposition=None, compute_vertex_cones=None, smith_form=None, dualization=None, triangulation=None, triangulation_max_height=None, **kwds)

Return the Ehrhart polynomial of this polyhedron.

Let $P$ be a lattice polytope in $\mathbb{R}^d$ 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 Ehrhart polynomial of $P$. For more information see the Wikipedia article Ehrhart_polynomial.

The Ehrhart polynomial may be computed using either LattE Integrale or Normaliz by setting engine to ‘latte’ or ‘normaliz’ respectively.

INPUT:

- engine – string; The backend to use. Allowed values are:
  - None (default); When no input is given the Ehrhart polynomial is computed using LattE Integrale (optional)
  - 'latte'; use LattE integrale program (optional)
  - 'normaliz'; use Normaliz program (optional). The backend of self must be set to ‘normaliz’.
- variable – string (default: ‘t’); The variable in which the Ehrhart polynomial should be expressed.
- When the engine is ‘latte’ or None, the additional input values are:
  - verbose - boolean (default: False); if True, print the whole output of the LattE command.
  - dual - boolean; triangulate and signed-decompose in the dual space
  - irrational_primal - boolean; triangulate in the dual space, signed-decompose in the primal space using irrationalization.
  - irrational_all_primal - boolean; Triangulate and signed-decompose in the primal space using irrationalization.
  - maxdet – integer; decompose down to an index (determinant) of maxdet instead of index 1 (unimodular cones).
  - no_decomposition – boolean; do not signed-decompose simplicial cones.
  - compute_vertex_cones – string; either ‘cdd’ or ‘lrs’ or ‘4ti2’
  - smith_form – string; either ‘ilio’ or ‘lidia’
The Ehrhart polynomial as a univariate polynomial in \( t \) over a rational field.

See also:

- \texttt{latte} the interface to LattE Integrale PyNormaliz

**EXAMPLES:**

To start, we find the Ehrhart polynomial of a three-dimensional \texttt{simplex}, first using \texttt{engine='latte'}.

```python
sage: simplex = Polyhedron(vertices=[(0,0,0),(3,3,3),(-3,2,1),(1,-1,-2)])
sage: poly = simplex.ehrhart_polynomial(engine = 'latte')  # optional - latte-
7/2*t^3 + 2*t^2 - 1/2*t + 1
sage: poly(1)  # optional - latte-
6
sage: len(simplex.integral_points())  # optional - latte-
6
sage: poly(2)  # optional - latte-
36
sage: len((2*simplex).integral_points())  # optional - latte-
36
```

Now we find the same Ehrhart polynomial, this time using \texttt{engine='normaliz'}. To use the Normaliz engine, the \texttt{simplex} must be defined with \texttt{backend='normaliz'}:

```python
sage: simplex = Polyhedron(vertices=[(0,0,0),(3,3,3),(-3,2,1),(1,-1,-2)],
backend='normaliz')  # optional - pynormaliz
sage: poly = simplex.ehrhart_polynomial(engine='normaliz')  # optional - pynormaliz
7/2*t^3 + 2*t^2 - 1/2*t + 1
```

If the \texttt{engine='normaliz'}, the backend should be 'normaliz', otherwise it returns an error:

```python
sage: simplex = Polyhedron(vertices=[(0,0,0),(3,3,3),(-3,2,1),(1,-1,-2)])
sage: simplex.ehrhart_polynomial(engine='normaliz')  # optional - pynormaliz
Traceback (most recent call last):
  ...
TypeError: The polyhedron's backend should be 'normaliz'
```
Now we find the Ehrhart polynomials of the unit hypercubes of dimensions three through six. They are computed first with `engine='latte'` and then with `engine='normaliz'`. The degree of the Ehrhart polynomial matches the dimension of the hypercube, and the coefficient of the leading monomial equals the volume of the unit hypercube:

```sage
def hypercube(d):
    return Polyhedron(vertices=list(product([0,1],repeat=d)))

sage: hypercube(3).ehrhart_polynomial()
# optional - latte_int
\(t^3 + 3t^2 + 3t + 1\)

sage: hypercube(4).ehrhart_polynomial()
# optional - latte_int
\(t^4 + 4t^3 + 6t^2 + 4t + 1\)

sage: hypercube(5).ehrhart_polynomial()
# optional - latte_int
\(t^5 + 5t^4 + 10t^3 + 10t^2 + 5t + 1\)

sage: hypercube(6).ehrhart_polynomial()
# optional - latte_int
\(t^6 + 6t^5 + 15t^4 + 20t^3 + 15t^2 + 6t + 1\)

sage: def hypercube(d):
    return Polyhedron(vertices=list(product([0,1],repeat=d)),backend='normaliz')
# optional - pynormaliz

sage: hypercube(3).ehrhart_polynomial(engine='normaliz')
# optional - pynormaliz
\(t^3 + 3t^2 + 3t + 1\)

sage: hypercube(4).ehrhart_polynomial(engine='normaliz')
# optional - pynormaliz
\(t^4 + 4t^3 + 6t^2 + 4t + 1\)

sage: hypercube(5).ehrhart_polynomial(engine='normaliz')
# optional - pynormaliz
\(t^5 + 5t^4 + 10t^3 + 10t^2 + 5t + 1\)

sage: hypercube(6).ehrhart_polynomial(engine='normaliz')
# optional - pynormaliz
\(t^6 + 6t^5 + 15t^4 + 20t^3 + 15t^2 + 6t + 1\)
```

An empty polyhedron:

```sage
p = Polyhedron(ambient_dim=3, vertices=[])
p.ehrhart_polynomial()

\(0\)

parent(_)
Univariate Polynomial Ring in t over Rational Field
```

The polyhedron should be compact:

```sage
C = Polyhedron(rays=[[1,2],[2,1]])
C.ehrhart_polynomial()
Traceback (most recent call last):
...
ValueError: Ehrhart polynomial only defined for compact polyhedra
```

`fibration_generator(dim)`

Generate the lattice polytope fibrations.

For the purposes of this function, a lattice polytope fiber is a sub-lattice polytope. Projecting the plane spanned by the subpolytope to a point yields another lattice polytope, the base of the fibration.

INPUT:

- `dim` – integer. The dimension of the lattice polytope fiber.

OUTPUT:
A generator yielding the distinct lattice polytope fibers of given dimension.

EXAMPLES:

```
sage: P = Polyhedron(toric_varieties.P4_11169().fan().rays(), base_ring=ZZ)
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:**

```
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:

```
sage: polytopes.cross_polytope(3).is_lattice_polytope()
True
sage: polytopes.regular_polygon(5).is_lattice_polytope()
False
```

**is_reflexive()**

A lattice polytope is reflexive if it contains the origin in its interior and its polar with respect to the origin is a lattice polytope.

Equivalently, it is reflexive if it is of the form \( \{ x \in \mathbb{R}^d : Ax \leq 1 \} \) for some integer matrix \( A \) and \( d \) the ambient dimension.

EXAMPLES:

```
sage: p = Polyhedron(vertices=[(1,0,0),(0,1,0),(0,0,1),(-1,-1,-1)], base_ring=ZZ)
sage: p.is_reflexive()
True
sage: polytopes.hypercube(4).is_reflexive()
True
sage: p = Polyhedron(vertices=[(1,0), (0,2), (-1,0), (0,-1)], base_ring=ZZ)
sage: p.is_reflexive()
False
sage: p = Polyhedron(vertices=[(1,0), (0,2), (-1,0)], base_ring=ZZ)
sage: p.is_reflexive()
False
```

An error is raised, if the polyhedron is not compact:

```
sage: p = Polyhedron(rays=[(1,)], base_ring=ZZ)
sage: p.is_reflexive()
Traceback (most recent call last):
...
ValueError: the polyhedron is not compact
```

**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:

```
sage: square = Polyhedron(vertices=[(0,0),(1,0),(0,1),(1,1)])
sage: square.minkowski_decompositions()
((A 0-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex, A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices), (A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices, A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices))
```
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 ZZ^2 defined as the convex hull of 1 vertex,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 7 vertices),
(A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices),
(A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 7 vertices),
(A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 5 vertices,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices),
(A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 7 vertices),
(A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 5 vertices,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 3 vertices),
(A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 7 vertices),
(A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 5 vertices,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 3 vertices),
(A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 7 vertices),
(A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 5 vertices,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices),
(A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 7 vertices),
(A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 5 vertices,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 6 vertices))
sage: [ len(square.dilation(i).minkowski_decompositions())
  ....:   for i in range(6) ]
[1, 2, 5, 8, 13, 18]
sage: [ ceil((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 ZZ^3 defined as the convex hull of 4 vertices
sage: type(_)
<class 'sage.geometry.polyhedron.parent.Polyhedra_ZZ_ppl_with_category.element_class'>
sage: p.polar().base_ring()
Integer Ring
```
2.5.4 Base class for polyhedra over RDF

class sage.geometry.polyhedron.base_RDF.Polyhedron_RDF(parent, Vrep, Hrep, Vrep_minimal=None, Hrep_minimal=None, pref_rep=None, mutable=False, **kwds)

Bases: sage.geometry.polyhedron.base.Polyhedron_base

Base class for polyhedra over RDF.

2.6 Backends for Polyhedra

2.6.1 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:

```python
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, 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, parent, 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:

```python
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(ambient_dim, 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,
Vrep_minimal=None,
Hrep_minimal=None, pref_rep=None,
mutable=False, **kwds)

Bases: sage.geometry.polyhedron.base.Polyhedron_base

Base class for the cdd backend.

2.6.2 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, pref_rep=None,
mutable=False, **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)
sage: TestSuite(p).run()
```
2.6.3 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
• Jean-Philippe Labbé (2019-04): Expose normaliz features and added functionalities

class sage.geometry.polyhedron.backend_normaliz.Polyhedron_QQ_normaliz:
    (parent, Vrep, Hrep, normaliz_cone=None, normaliz_data=None, normaliz_field=None, **kwds)

    Bases: sage.geometry.polyhedron.backend_normaliz.Polyhedron_normaliz, sage.geometry.polyhedron.base_QQ.Polyhedron_QQ

    Polyhedra over 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)  # optional - pynormaliz
    sage: TestSuite(p).run()  # optional - pynormaliz

    ehrhart_series(variable='t')

    Return the Ehrhart series of a compact rational polyhedron.

    The Ehrhart series is the generating function where the coefficient of \(t^k\) is number of integer lattice points inside the \(k\)-th dilation of the polytope.

    INPUT:

    • variable – string (default: 't')

    OUTPUT:

    A rational function.

    EXAMPLES:

    sage: S = Polyhedron(vertices=[[0,1],[1,0]], backend='normaliz')  # optional - pynormaliz
    sage: ES = S.ehrhart_series()  # optional - pynormaliz
    sage: ES.numerator()  # optional - pynormaliz
    1
    sage: ES.denominator().factor()  # optional - pynormaliz
    (continues on next page)
\[(t - 1)^2\]

\[\text{sage: } C = \text{Polyhedron(}\text{vertices} = [[0,0,0],[0,0,1],[0,1,0],[0,1,1],[1,0,0],[1,0,1],[1,1,0],[1,1,1]],\text{backend='normaliz'})\] # optional - pynormaliz
\[\text{sage: } ES = C.ehrhart_series()\] # optional - pynormaliz
\[\text{sage: } ES.numerator()\] # optional - pynormaliz
\[t^2 + 4*t + 1\]
\[\text{sage: } ES.denominator().factor()\] # optional - pynormaliz
\[(t - 1)^4\]

The following example is from the Normaliz manual contained in the file `rational.in`:

\[\text{sage: rat_poly = Polyhedron(}\text{vertices}=[[1/2,1/2],[-1/3,-1/3],[1/4,-1/2]],\text{backend='normaliz'})\] # optional - pynormaliz
\[\text{sage: } ES = \text{rat_poly.ehrhart_series()\] # optional - pynormaliz
\[\text{sage: } ES.numerator()\] # optional - pynormaliz
\[2*t^6 + 3*t^5 + 4*t^4 + 3*t^3 + t^2 + t + 1\]
\[\text{sage: } ES.denominator().factor()\] # optional - pynormaliz
\[(-1) * (t + 1)^2 * (t - 1)^3 * (t^2 + 1) * (t^2 + t + 1)\]

The polyhedron should be compact:

\[\text{sage: } C = \text{Polyhedron(}\text{backend='normaliz'},\text{rays}=[[1,2],[2,1]])\] # optional - pynormaliz
\[\text{sage: } C.ehrhart_series()\] # optional - pynormaliz

Traceback (most recent call last):
...
NotImplementedError: Ehrhart series can only be computed for compact polyhedron

See also:

hilbert_series()

hilbert_series(grading, variable=’t’)

Return the Hilbert series of the polyhedron with respect to grading.

INPUT:

• grading – vector. The grading to use to form the Hilbert series

• variable – string (default: ‘t’)

OUTPUT:

A rational function.

EXAMPLES:

\[\text{sage: } C = \text{Polyhedron(}\text{backend='normaliz'},\text{rays}=[0,0,1],[0,1,1],[1,0,1],[1,1,1])\] # optional - pynormaliz
\[\text{sage: } HS = C.hilbert_series([1,1])\] # optional - pynormaliz
\[\text{sage: } HS.numerator()\] # optional - pynormaliz
t^2 + 1
\begin{verbatim}
sage: HS.denominator().factor() # optional - pynormaliz
(-1) * (t + 1) * (t - 1)^3 * (t^2 + t + 1)
\end{verbatim}

By changing the grading, you can get the Ehrhart series of the square lifted at height 1:

\begin{verbatim}
sage: C.hilbert_series([0,0,1]) # optional - pynormaliz
(t + 1)/(-t^3 + 3*t^2 - 3*t + 1)
\end{verbatim}

Here is an example 2cone.in from the Normaliz manual:

\begin{verbatim}
sage: C = Polyhedron(backend='normaliz',rays=[[1,3],[2,1]]) # optional ~-> pynormaliz
sage: HS = C.hilbert_series([1,1]) # optional - pynormaliz
sage: HS.numerator() # optional - pynormaliz
t^5 + t^4 + t^3 + t^2 + 1
sage: HS.denominator().factor() # optional - pynormaliz
(t + 1) * (t - 1)^2 * (t^2 + 1) * (t^2 + t + 1)
\end{verbatim}

Here is the magic square example form the Normaliz manual:

\begin{verbatim}
sage: eq = [[0,1,1,1,-1,-1,-1, 0, 0, 0], ....: [0,1,1,1, 0, 0, 0,-1,-1,-1], ....: [0,0,1,1,-1, 0, 0,-1, 0, 0], ....: [0,1,0,1, 0,-1, 0, 0,-1, 0], ....: [0,1,1,0, 0, 0,-1, 0, 0,-1], ....: [0,0,1,1, 0,-1, 0, 0, 0,-1], ....: [0,1,1,0, 0,-1, 0,-1, 0, 0], ....: [0,1,1,0, 0,-1, 0,-1, 0, 0]]
sage: magic_square = Polyhedron(eqns=eq,backend='normaliz') &-> Polyhedron(rays=identity_matrix(9).rows()) # optional - pynormaliz
sage: grading = [1,1,1,0,0,0,0,0,0]
sage: magic_square.hilbert_series(grading) # optional - pynormaliz
(t^6 + 2*t^3 + 1)/(-t^9 + 3*t^6 - 3*t^3 + 1)
\end{verbatim}

See also:

\texttt{ehrhart_series()}

\texttt{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.

\textbf{INPUT:}

- \texttt{threshold} – integer (default: 10000); use the naïve algorithm as long as the bounding box is smaller than this

\textbf{OUTPUT:}

The list of integral points in the polyhedron. If the polyhedron is not compact, a \texttt{ValueError} is raised.
EXAMPLES:

\begin{verbatim}
    sage: Polyhedron(vertices=[(-1,-1), (1,0), (1,1), (0,1)],
                  backend='normaliz').integral_points()
    ((-1, -1), (0, 0), (0, 1), (1, 0), (1, 1))
    sage: simplex = Polyhedron([(1,2,3), (2,3,7), (-2,-3,-11)],
                             backend='normaliz')
    sage: simplex.integral_points()
    ((-2, -3, -11), (0, 0, -2), (1, 2, 3), (2, 3, 7))
    sage: simplex = Polyhedron([(1,2,3,5), (2,3,7,5), (-2,-3,-11,5)],
                             backend='normaliz')
    sage: simplex.integral_points()
    ((-2, -3, -11, 5), (0, 0, -2, 5), (1, 2, 3, 5), (2, 3, 7, 5))
    sage: point = Polyhedron([(2,3,7)], backend='normaliz')
    sage: point.integral_points()
    ((2, 3, 7),)
    sage: empty = Polyhedron(backend='normaliz')
    sage: empty.integral_points()
    ()
    sage: v = [(1,0,7,-1), (-2,-2,4,-3), (-1,-1,-1,4), (2,9,0,-5), (-2,-1,5,1)]
    sage: simplex = Polyhedron(v, backend='normaliz'); simplex
    A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 5 vertices
    sage: len(simplex.integral_points())
    49
    sage: P = Polyhedron(vertices=((0, 0), (178933,37121)) + 1/1000*polytopes.hypercube(2))
    sage: P = Polyhedron(vertices=P.vertices_list(), backend='normaliz')
\end{verbatim}

The polyhedron need not be full-dimensional:

\begin{verbatim}
    sage: simplex = Polyhedron([(1,2,3,5), (2,3,7,5), (-2,-3,-11,5)],
                             backend='normaliz')
    sage: simplex.integral_points()
    ((-2, -3, -11, 5), (0, 0, -2, 5), (1, 2, 3, 5), (2, 3, 7, 5))
    sage: point = Polyhedron([(2,3,7)], backend='normaliz')
    sage: point.integral_points()
    ((2, 3, 7),)
    sage: empty = Polyhedron(backend='normaliz')
    sage: empty.integral_points()
    ()
    sage: v = [(1,0,7,-1), (-2,-2,4,-3), (-1,-1,-1,4), (2,9,0,-5), (-2,-1,5,1)]
    sage: simplex = Polyhedron(v, backend='normaliz'); simplex
    A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 5 vertices
    sage: len(simplex.integral_points())
    49
    sage: P = Polyhedron(vertices=((0, 0), (178933,37121)) + 1/1000*polytopes.hypercube(2))
    sage: P = Polyhedron(vertices=P.vertices_list(), backend='normaliz')
\end{verbatim}

Here is a simplex where the naive algorithm of running over all points in a rectangular bounding box no longer works fast enough:

\begin{verbatim}
    sage: v = [(1,0,7,-1), (-2,-2,4,-3), (-1,-1,-1,4), (2,9,0,-5), (-2,-1,5,1)]
    sage: simplex = Polyhedron(v, backend='normaliz'); simplex
    A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 5 vertices
    sage: len(simplex.integral_points())
    49
    sage: P = Polyhedron(vertices=((0, 0), (178933,37121)) + 1/1000*polytopes.hypercube(2))
    sage: P = Polyhedron(vertices=P.vertices_list(), backend='normaliz')
\end{verbatim}

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):

\begin{verbatim}
    sage: P = Polyhedron(vertices=((0, 0), (178933,37121)) + 1/1000*polytopes.hypercube(2))
    sage: P = Polyhedron(vertices=P.vertices_list(), backend='normaliz')
\end{verbatim}
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, backend='normaliz') # optional - pynormaliz
sage: pts1 = P.integral_points() # optional - pynormaliz
sage: all(P.contains(p) for p in pts1) # optional - pynormaliz
True
sage: pts2 = LatticePolytope(v).points() # PALP
sage: for p in pts1: p.set_immutable() # optional - pynormaliz
sage: set(pts1) == set(pts2) # optional - pynormaliz
True
```

`integral_points_generators()`

Return the integral points generators of the polyhedron.

Every integral point in the polyhedron can be written as a (unique) non-negative linear combination of integral points contained in the three defining parts of the polyhedron: the integral points (the compact part), the recession cone, and the lineality space.

**OUTPUT:**

A tuple consisting of the integral points, the Hilbert basis of the recession cone, and an integral basis for the lineality space.

**EXAMPLES:**

Normaliz gives a nonnegative integer basis of the lineality space:

```python
sage: P = Polyhedron(backend='normaliz', lines=[[2,2]]) # optional - pynormaliz
sage: P.integral_points_generators() # optional - pynormaliz
(((0, 0),), (), ((1, 1),))
```

A recession cone generated by two rays:

```python
sage: C = Polyhedron(backend='normaliz', rays=[[1,2],[2,1]]) # optional - pynormaliz
sage: C.integral_points_generators() # optional - pynormaliz
```
Empty polyhedron:

```python
sage: P = Polyhedron(backend='normaliz')  # optional - pynormaliz
sage: P.integral_points_generators()     # optional - pynormaliz
(((), (), ()),)
```

```python
class sage.geometry.polyhedron.backend_normaliz.Polyhedron_ZZ_normaliz(parent, Vrep, Hrep, normaliz_cone=None, normaliz_data=None, normaliz_field=None, **kwds)
Bases: sage.geometry.polyhedron.backend_normaliz.Polyhedron_QQ_normaliz, sage.geometry.polyhedron.base.ZZ.Polyhedron_ZZ
Polyhedra over \( \mathbb{Z} \) with normaliz.
INPUT:

- \( \text{Vrep} \) – a list [vertices, rays, lines] or None
- \( \text{Hrep} \) – a list [ieqs, eqns] or None

EXAMPLES:

```python
sage: p = Polyhedron(backend='normaliz',
                  vertices=[(0,0),(1,0),(0,1)],
                  # optional - pynormaliz
                  rays=[(1,1)], lines=[],
                  backend='normaliz', base_ring=ZZ)
```

```python
class sage.geometry.polyhedron.backend_normaliz.Polyhedron_normaliz(parent, Vrep, Hrep, normaliz_cone=None, normaliz_data=None, normaliz_field=None, **kwds)
Bases: sage.geometry.polyhedron.base.Polyhedron_base
Polyhedra with normaliz
INPUT:

- \( \text{parent} \) – Polyhedra the parent
- \( \text{Vrep} \) – a list [vertices, rays, lines] or None; the V-representation of the polyhedron; if None, the polyhedron is determined by the H-representation
- \( \text{Hrep} \) – a list [ieqs, eqns] or None; the H-representation of the polyhedron; if None, the polyhedron is determined by the V-representation
- \( \text{normaliz\_cone} \) – a PyNormaliz wrapper of a normaliz cone

Only one of \( \text{Vrep}, \text{Hrep}, \text{normaliz\_cone} \) can be different from None.

EXAMPLES:
Two ways to get the full space:

```
sage: Polyhedron(eqns=[[0, 0, 0]], backend='normaliz')  # optional -
    pynormaliz
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')  # optional -
    pynormaliz
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 homoge-

```
sage: P = Polyhedron(vertices=[[1, 1]], rays=[[0, 1]],
    backend='normaliz')
    pynormaliz
sage: P.n_inequalities()  # optional -
    pynormaliz
1
sage: P.equations()  # optional -
    pynormaliz
(An equation (1, 0) x - 1 == 0,)
```

The empty polyhedron:

```
sage: P=Polyhedron(ieqs=[[-2, 1, 1], [-3, -1, -1], [-4, 1, -2]],
    backend='normaliz')
    pynormaliz
sage: P  # optional -
    pynormaliz
The empty polyhedron in QQ^2
sage: P.Vrepresentation()  # optional -
    pynormaliz
()  # optional -
    pynormaliz
(An equation -1 == 0,)
```

```
integral_hull()
Return the integral hull in the polyhedron.
This is a new polyhedron that is the convex hull of all integral points.
EXAMPLES:
Unbounded example from Normaliz manual, “a dull polyhedron”:
```
```
sage: P = Polyhedron(ieqs=[[1, 0, 2], [3, 0, -2], [3, 2, -2]], # optional -
          backend='normaliz')
   ....:   pynormaliz
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:
```
sage: P = Polyhedron(vertices=[[1/2, 1/3]], rays=[[1, 1]], # optional -
          lines=[[-1, 1]], backend='normaliz')
   ....:   pynormaliz
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:
```
sage: P = Polyhedron(backend='normaliz') # optional -
          pynormaliz
sage: PI = P.integral_hull()              # optional -
          pynormaliz
sage: PI.Vrepresentation()               # optional -
          pynormaliz
   ()
```

### 2.6.4 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

```python
class sage.geometry.polyhedron.backend_polymake.Polyhedron_QQ_polymake(parent, Vrep, Hrep, # polymake_polytope=None, **kwds)
```

**Bases:** `sage.geometry.polyhedron.backend_polymake.Polyhedron_polymake, sage.geometry.polyhedron.base_QQ.Polyhedron_QQ`

Polyhedra over $\mathbb{Q}$ 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()
```

```python
class sage.geometry.polyhedron.backend_polymake.Polyhedron_ZZ_polymake(
    parent, Vrep, Hrep,
    polymake_polytope=None,
    **kwds)
```

Bases: `sage.geometry.polyhedron.backend_polymake.Polyhedron_polymake`, `sage.geometry.polyhedron.base.ZZ.Polyhedron_ZZ`

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()
```

```python
class sage.geometry.polyhedron.backend_polymake.Polyhedron_polymake(
    parent, Vrep, Hrep,
    polymake_polytope=None,
    **kwds)
```

Bases: `sage.geometry.polyhedron.base.Polyhedron_base`

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:
A lower-dimensional affine cone; we test that there are no mysterious inequalities coming in from the homoge-
nization:

```python
sage: P = Polyhedron(verts=[(1, 1)], rays=[(0, 1)],
                  backend='polymake')
```

1

```
sage: P.n_inequalities()
```

(An equation \((1, 0) x - 1 == 0,)\)

The empty polyhedron:

```python
sage: P = Polyhedron(eqns=[[1, 0, 0]], backend='polymake')
```

```
sage: P.Vrepresentation()  
```

()`

```
sage: P.Hrepresentation()  
```

(An equation \(-1 == 0,)\)

The full polyhedron:

```python
sage: Polyhedron(eqns=[[0, 0, 0]], backend='polymake')
```

A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2

```python
sage: Polyhedron(ieqs=[[0, 0, 0]], backend='polymake')
```

A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2

Quadratic fields work:
2.6.5 The PPL (Parma Polyhedra Library) backend for polyhedral computations

class sage.geometry.polyhedron.backend_ppl.Polyhedron_QQ_ppl

Bases: sage.geometry.polyhedron.backend_ppl.Polyhedron_ppl, sage.geometry.polyhedron.base_QQ.Polyhedron_QQ

Polyhedra over \( \mathbb{Q} \) with ppl

INPUT:

- \( \text{Vrep} \) – a list [vertices, rays, lines] or None.
- \( \text{Hrep} \) – a list [ieqs, eqns] or None.

EXAMPLES:

```python
sage: p = Polyhedron(vertices=[[0,0],[1,0],[0,1]], rays=[[1,1]], lines=[],
                     backend='ppl', base_ring=QQ)
sage: TestSuite(p).run()
```

class sage.geometry.polyhedron.backend_ppl.Polyhedron_ZZ_ppl

Bases: sage.geometry.polyhedron.backend_ppl.Polyhedron_ppl, sage.geometry.polyhedron.base_ZZ.Polyhedron_ZZ

Polyhedra over \( \mathbb{Z} \) with ppl

INPUT:

- \( \text{Vrep} \) – a list [vertices, rays, lines] or None.
- \( \text{Hrep} \) – a list [ieqs, eqns] or None.

EXAMPLES:

```python
sage: p = Polyhedron(vertices=[[0,0],[1,0],[0,1]], rays=[[1,1]], lines=[],
                     backend='ppl', base_ring=ZZ)
sage: TestSuite(p).run()
```

class sage.geometry.polyhedron.backend_ppl.Polyhedron_ppl

Bases: sage.geometry.polyhedron.base_mutable.Polyhedron_mutable

Polyhedra with ppl

INPUT:

- \( \text{Vrep} \) – 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')
sage: TestSuite(p).run()
```

### Hrepresentation(index=None)

Return the objects of the H-representation. Each entry is either an inequality or a equation.

**INPUT:**

• index – either an integer or None

**OUTPUT:**

The optional argument is an index running from 0 to self.n_Hrepresentation()-1. If present, the H-representation object at the given index will be returned. Without an argument, returns the list of all H-representation objects.

**EXAMPLES:**

```
sage: p = polytopes.hypercube(3)
sage: p.Hrepresentation(0)
An inequality (-1, 0, 0) x + 1 >= 0
sage: p.Hrepresentation(0) == p.Hrepresentation()[0]
True
```

```
sage: P = p.parent()
sage: p = P._element_constructor_(p, mutable=True)
sage: p.Hrepresentation(0)
An inequality (0, 0, -1) x + 1 >= 0
sage: p._clear_cache()
sage: p.Hrepresentation(0)
An inequality (0, 0, -1) x + 1 >= 0
sage: TestSuite(p).run()
```

### Vrepresentation(index=None)

Return the objects of the V-representation. Each entry is either a vertex, a ray, or a line.

See `sage.geometry.polyhedron.constructor` for a definition of vertex/ray/line.

**INPUT:**

• index – either an integer or None

**OUTPUT:**

The optional argument is an index running from 0 to self.n_Vrepresentation()-1. If present, the V-representation object at the given index will be returned. Without an argument, returns the list of all V-representation objects.

**EXAMPLES:**

```
sage: p = polytopes.cube()
sage: p.Vrepresentation(0)
A vertex at (1, -1, -1)
```

2.6. Backends for Polyhedra
```python
sage: P = p.parent()
sage: p = P._element_constructor_(p, mutable=True)
sage: p.Vrepresentation(0)
A vertex at (-1, -1, -1)
sage: p._clear_cache()
sage: p.Vrepresentation(0)
A vertex at (-1, -1, -1)
sage: TestSuite(p).run()
```

```python
set_immutable()

Make this polyhedron immutable. This operation cannot be undone.

EXAMPLES:

```python
sage: p = Polyhedron([[1, 1]], mutable=True)
sage: p.is_mutable()
True
sage: hasattr(p, "_Vrepresentation")
False
sage: p.set_immutable()
```

### 2.6.6 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 backend for polyhedral computations. But as far as this module is concerned, inequality always means without a constant term and the origin is always a point of the cone.

EXAMPLES:

```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
\[
\begin{bmatrix}
1 & 0 & 1 \\
0 & 1 & 1 \\
-1 & -1 & 1
\end{bmatrix}
\begin{bmatrix}
2/3 & -1/3 & -1/3 \\
1/3 & 1/3 & 1/3
\end{bmatrix}
\]
```

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
[(\text{exact field value}), (\text{exact field value}), (\text{exact field value})]
```

(continues on next page)
class sage.geometry.polyhedron.double_description.DoubleDescriptionPair

    Bases: object

    Base class for a double description pair \((A, R)\)

    Warning: You should use the Problem.initial_pair() or Problem.run() to generate double description pairs for a set of inequalities, and not generate DoubleDescriptionPair instances directly.

INPUT:

- problem – instance of Problem.
- A_rows – list of row vectors of the matrix \(A\). These encode the inequalities.
- R_cols – list of column vectors of the matrix \(R\). These encode the rays.

R_by_sign(a)

    Classify the rays into those that are positive, zero, and negative on \(a\).

    INPUT:


    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)])
    sage: DD = StandardAlgorithm(A).run()
cone()
Return the cone defined by $A$.

This method is for debugging only. Assumes that the base ring is $\mathbb{Q}$.

OUTPUT:
The cone defined by the inequalities as a `Polyhedron()`, using the PPL backend.

EXAMPLES:
```
sage: from sage.geometry.polyhedron.double_description import StandardAlgorithm
sage: A = matrix(QQ, [(1,0,1), (0,1,1), (-1,-1,1)])
sage: DD, _ = StandardAlgorithm(A).initial_pair()
sage: DD.cone().Hrepresentation()
(An inequality (-1, -1, 1) x + 0 >= 0,
 An inequality (0, 1, 1) x + 0 >= 0,
 An inequality (1, 0, 1) x + 0 >= 0)
```
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]
  [ 1 0 0]
  [ 0 -1 1]
A = [0 1 0], R = [1 0 0]
sage: DD.dual()
Double description pair (A, R) defined by
  [0 1 1]
  [1 0 0]
  [0 0 1]
  [0 1 0]
A = [1 0 0], R = [1 0 -1]
  [0 0 1]
```
first_coordinate_plane()
Restrict to the first coordinate plane.

OUTPUT:
A new double description pair with the constraint $x_0 = 0$ added.

EXAMPLES:
sage: A = matrix([[1, 1], [-1, 1]])
sage: from sage.geometry.polyhedron.double_description import StandardAlgorithm
sage: DD, _ = StandardAlgorithm(A).initial_pair()
sage: DD
Double description pair (A, R) defined by
A = [ 1 1], R = [ 1/2 -1/2]
[-1 1] [ 1/2 1/2]
sage: DD.first_coordinate_plane()
Double description pair (A, R) defined by
A = [ 1 1]
[-1 1]
A = [ 0]
[1/2]
[ 1 0]
inner_product_matrix()
Return the inner product matrix between the rows of \( A \) and the columns of \( R \).

OUTPUT:
A matrix over the base ring. There is one row for each row of \( A \) and one column for each column of \( R \).

EXAMPLES:

sage: from sage.geometry.polyhedron.double_description import StandardAlgorithm
sage: A = matrix(QQ, [(1,0,1), (0,1,1), (-1,-1,1)])
sage: alg = StandardAlgorithm(A)
sage: DD, _ = alg.initial_pair()
sage: DD.inner_product_matrix()
[1 0 0]
[0 1 0]
[0 0 1]
is_extremal(ray)
Test whether the ray is extremal.

EXAMPLES:

sage: from sage.geometry.polyhedron.double_description import StandardAlgorithm
sage: A = matrix(QQ, [(0,1,0), (1,0,0), (0,-1,1), (-1,0,1)])
sage: DD = StandardAlgorithm(A).run()
sage: DD.is_extremal(DD.R[0])
True
matrix_space(nrows, ncols)
Return a matrix space of size nrows and ncols over the base ring of self.

These matrix spaces are cached to avoid their creation in the very demanding add_inequality() and more precisely are_adjacent().

EXAMPLES:

sage: from sage.geometry.polyhedron.double_description import Problem
sage: A = matrix(QQ, [(1,0,1), (0,1,1), (-1,-1,1)])
sage: DD, _ = Problem(A).initial_pair()
sage: DD.matrix_space(2,2)
Full MatrixSpace of 2 by 2 dense matrices over Rational Field

(continues on next page)
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 with sqrt2 = 1.414213562373095?

verify()
Validate the double description pair.
This method used the PPL backend to check that the double description pair is valid. An assertion is triggered if it is not. Does nothing if the base ring is not $\mathbb{Q}$.

EXAMPLES:

sage: from sage.geometry.polyhedron.double_description import DoubleDescriptionPair, Problem
sage: A = matrix(QQ, [(1,0,1), (0,1,1), (-1,-1,1)])

sage: alg = Problem(A)
sage: DD = DoubleDescriptionPair(alg, [(1, 0, 3), (0, 1, 1), (-1, -1, 1)], [(2/3, -1/3, 1/3), (-1/3, 2/3, 1/3), (-1/3, -1/3, 1/3)])

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)
Bases: object

Base class for implementations of the double description algorithm

It does not make sense to instantiate the base class directly, it just provides helpers for implementations.
INPUT:

• A – a matrix. The rows of the matrix are interpreted as homogeneous inequalities $Ax \geq 0$. Must have maximal rank.

A()

Return the rows of the defining matrix $A$.

OUTPUT:

The matrix $A$ whose rows are the inequalities.

EXAMPLES:

```
sage: A = matrix([[1, 1], [-1, 1]])
sage: from sage.geometry.polyhedron.double_description import Problem
sage: Problem(A).A()
((1, 1), (-1, 1))
```

A_matrix()

Return the defining matrix $A$.

OUTPUT:

Matrix whose rows are the inequalities.

EXAMPLES:

```
sage: A = matrix([[1, 1], [-1, 1]])
sage: from sage.geometry.polyhedron.double_description import Problem
sage: Problem(A).A_matrix()
[ 1 1]
[-1 1]
```

base_ring()

Return the base field.

OUTPUT:

A field.

EXAMPLES:

```
sage: A = matrix(AA, [[1, 1], [-1, 1]])
sage: from sage.geometry.polyhedron.double_description import Problem
sage: Problem(A).base_ring()
Algebraic Real Field
```

dim()

Return the ambient space dimension.

OUTPUT:

Integer. The ambient space dimension of the cone.

EXAMPLES:

```
sage: A = matrix(QQ, [[1, 1], [-1, 1]])
sage: from sage.geometry.polyhedron.double_description import Problem
sage: Problem(A).dim()
2
```
initial_pair()

Return an initial double description pair.

Picks an initial set of rays by selecting a basis. This is probably the most efficient way to select the initial set.

INPUT:

• pair_class – subclass of `DoubleDescriptionPair`.

OUTPUT:

A pair consisting of a `DoubleDescriptionPair` instance and the tuple of remaining unused inequalities.

EXAMPLES:

```python
sage: A = matrix([(-1, 1), (-1, 2), (1/2, -1/2), (1/2, 2)])
sage: from sage.geometry.polyhedron.double_description import Problem
sage: DD, remaining = Problem(A).initial_pair()
sage: DD.verify()
sage: remaining
[(1/2, -1/2), (1/2, 2)]
```

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:

```python
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)]
```

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:

```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: StandardAlgorithm(A).run()
Double description pair \((A, R)\) defined by
\[
\begin{bmatrix}
0 & 1 & 0 \\
0 & 1 & 1
\end{bmatrix}
\begin{bmatrix}
0 & 0 & 1 \\
1 & 1 & 0
\end{bmatrix}
\]
```
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
[ -1  1  0]
[ -1  2  1]
[ 1/2 -1/2 -1]
```

`sage.geometry.polyhedron.double_description.random_inequalities(d, n)`
Random collections of inequalities for testing purposes.

INPUT:
- `d` – integer. The dimension.
- `n` – integer. The number of random inequalities to generate.

OUTPUT:
A random set of inequalities as a `StandardAlgorithm` instance.

EXAMPLES:
```
sage: from sage.geometry.polyhedron.double_description import random_inequalities
sage: P = random_inequalities(5, 10)
sage: P.run().verify()
```
2.6.7 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.

**EXAMPLES:**

```python
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:

```python
sage: Vrep2Hrep(QQ, 2, [(-1/2,0)], [(-1/2,2/3), (1/2,-1/3)], []
[1 2 3]
[2 4 3]
```

```python
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 Hrep2Vrep
sage: Hrep2Vrep(QQ, 2, [(1,2,3), (2,4,3)], []
[-1/2|-1/2 1/2|
```

(continues on next page)
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| -2/3]
[ 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| -1/3 -1/6]
sage: Hrep2Vrep(QQ, 2, [(1,2,3), (7,8,9)], [])
[-2/3]
[ 1/3]
sage: Hrep2Vrep(QQ, 2, [(1,0,0)], [])
# universe
[0|1 0]
[0|0 1]
sage: Hrep2Vrep(QQ, 2, [(-1,0,0)], [])
# empty
[]
sage: Hrep2Vrep(QQ, 2, [], [])
# universe
[0|1 0]
[0|0 1]

verify(inequalities, equations)

Compare result to PPL if the base ring is QQ.

This method is for debugging purposes and compares the computation with another backend if available.

INPUT:

• inequalities, equations – see Hrep2Vrep.

EXAMPLES:

```python
sage: from sage.geometry.polyhedron.double_description_inhomogeneous import Hrep2Vrep
sage: H = Hrep2Vrep(QQ, 1, [(1,2)], [])
sage: H.verify([(1,2)], [])
```

class sage.geometry.polyhedron.double_description_inhomogeneous.PivotedInequalities(base_ring, dim)

Bases: sage.structure.sage_object.SageObject

Base class for inequalities that may contain linear subspaces

INPUT:

• base_ring – a field.

• dim – integer. The ambient space dimension.

EXAMPLES:

```python
sage: from sage.geometry.polyhedron.double_description_inhomogeneous import PivotedInequalities
...:
    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 sage.geometry.polyhedron.double_description_inhomogeneous.Vrep2Hrep(base_ring, dim, vertices, rays, lines)

Bases: sage.geometry.polyhedron.double_description_inhomogeneous.PivotedInequalities

Convert V-representation to a minimal H-representation.

INPUT:

• base_ring – a field.
• dim – integer. The ambient space dimension.
• vertices – list of vertices. Each vertex is given as list of \( \text{dim} \) coordinates.
• 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 \( \text{dim} \) 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]

(continues on next page)
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)])
[]

\textbf{verify}(\textit{vertices, rays, lines})

Compare result to PPL if the base ring is \texttt{QQ}.

This method is for debugging purposes and compares the computation with another backend if available.

\textbf{INPUT:}

* \textit{vertices, rays, lines} – see \texttt{Vrep2Hrep}.

\textbf{EXAMPLES:}

\begin{verbatim}
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)
\end{verbatim}
Chapter Three

Triangulations

3.1 Triangulations of a point configuration

A point configuration is a finite set of points in Euclidean space or, more generally, in projective space. A triangulation is a simplicial decomposition of the convex hull of a given point configuration such that all vertices of the simplices end up lying on points of the configuration. That is, there are no new vertices apart from the initial points.

Note that points that are not vertices of the convex hull need not be used in the triangulation. A triangulation that does make use of all points of the configuration is called fine, and you can restrict yourself to such triangulations if you want. See `PointConfiguration` and `restrict_to_fine_triangulations()` for more details.

Finding a single triangulation and listing all connected triangulations is implemented natively in this package. However, for more advanced options [TOPCOM] needs to be installed. 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
```

(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
```

Graphics object consisting of 12 graphics primitives
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.

```python
sage: list( p_fine.triangulations() )
[(<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>),
 (<0,1,2>, <0,1,4>, <0,2,4>, <1,2,3>)]
```

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)
```

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
```

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 ]
```

(continues on next page)
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, fine, regular, star, defined_affine):

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>)
```

### 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:
```python
sage: pc = PointConfiguration([(0,0),(1,0),(0,1),(1,1)])
sage: pc.bistellar_flips()
(((<0,1,3>, <0,2,3>), (<0,1,2>, <1,2,3>)),)

sage: Tpos, Tneg = pc.bistellar_flips()[0]
sage: Tpos.plot(axes=False)
Graphics object consisting of 11 graphics primitives

sage: Tneg.plot(axes=False)
Graphics object consisting of 11 graphics primitives
```

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),(0,2,0),(0,0,2),(0,2,2),(1,1,1)])
sage: pc.bistellar_flips()
(((<0,1,3>, <0,2,3>), (<0,1,2>, <1,2,3>)),)

sage: Tpos, Tneg = pc.bistellar_flips()[0]
sage: Tpos.plot(axes=False)
Graphics3d Object
```

circuits()

Return the circuits of the point configuration.

Roughly, a circuit is a minimal linearly dependent subset of the points. That is, a circuit is a partition

\[ \{0, 1, \ldots, n - 1\} = C_+ \cup C_0 \cup C_- \]

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: sorted(p.circuits())
[((0,), (1, 2), (3, 4)), ((0,), (3, 4), (1, 2)), ((1, 2), (0,), (3, 4))]```

circuits_support()

A generator for the supports of the circuits of the point configuration.

See circuits() for details.

**OUTPUT:**

A generator for the supports \(C_- \cup C_+\) (returned as a Python tuple) for all circuits of the point configuration.

**EXAMPLES:**
**contained_simplex** *(large=True, initial_point=None, point_order=None)*

Return a simplex contained in the point configuration.

**INPUT:**

- `large` – boolean. Whether to attempt to return a large simplex.
- `initial_point` – a *Point* or *None* (default). A specific point to start with when picking the simplex vertices.
- `point_order` – a list or tuple of (some or all) *Point* s or *None* (default).

**OUTPUT:**

A tuple of points that span a simplex of dimension `dim()`. If `large=True`, the simplex is constructed by successively picking the farthest point. This will ensure that the simplex is not unnecessarily small, but will in general not return a maximal simplex. If a `point_order` is specified, the simplex is greedily constructed by considering the points in this order. The `large` option and `initial_point` is ignored in this case. The `point_order` may contain only a subset of the points; in this case, the dimension of the simplex will be the dimension of this subset.

**EXAMPLES:**

```python
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],[0,1,0],[1,0,1],[1,1,1],[-1,-1,1]])
sage: pc.contained_simplex()
(P(-1, -1, 1), P(1, 1, 0), P(0, 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))
```

Lower-dimensional example:

```python
sage: pc.contained_simplex(point_order = [pc[0],pc[3],pc[4]])
(P(0, 1), P(1, 1))
```

**convex_hull()**

Return the convex hull of the point configuration.

**EXAMPLES:**

```python
sage: p = PointConfiguration([(0,0,0],[0,1,0],[1,0,1],[1,1,1],[-1,-1,1]])
sage: p.convex_hull()
A 2-dimensional polyhedron 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:

```sage
cp = PointConfiguration([(0,0),(1,0),(2,1),(1,2),(0,1)])
cp.distance(cp.point(0), p) for p in cp.points() ]
[0, 1, 5, 5, 1]
cp = PointConfiguration([(0,0,1),(1,0,1),(2,1,1),(1,2,1),(0,1,1)], projective=True)
cp.distance(cp.point(0), p) for p in cp.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-Studi distance function

\[
d_{FS}(x, y) = \arccos \sqrt{\frac{(x \cdot y)^2}{|x|^2 |y|^2}}
\]

INPUT:
• x, y – two points of the point configuration.

OUTPUT:

The distance \(1 - \cos d_{FS}(x, y)^2\). Note that this distance lies in the same field as the entries of x, y. That is, the distance of rational points will be rational and so on.

EXAMPLES:

```sage
cp = PointConfiguration([(0,0),(1,0),(2,1),(1,2),(0,1)])
cp.distance_FS(cp.point(0), p) for p in cp.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
sage: triangle = PointConfiguration([[0,0], [1,-1], [1,0], [1,1]])
sage: triangle.point(2)
P(1, 0)
sage: triangle.face_codimension(2)
1
sage: triangle.face_codimension([1,0])
1
```

This also works for degenerate cases like the tip of the pyramid over a square (which saturates four inequalities):

```python
sage: pyramid = PointConfiguration([[1,0,0],[0,1,1],[0,1,-1],[0,-1,-1],[0,-1,˓→1]])
sage: 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
sage: triangle = PointConfiguration([[-1,0], [0,0], [1,-1], [1,0], [1,1]])
sage: triangle.face_interior()
((1,), (3,), (0, 2, 4))
sage: triangle.face_interior(dim=0)  # the vertices of the convex hull
(0, 2, 4)
```

(continues on next page)
farthest_point(points, among=None)
Return the point with the most distance from points.

INPUT:
• 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:

```python
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:

```python
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:

```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>)
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([...])
```

(continues on next page)
sage: p = PointConfiguration(U.columns())
sage: triangulation = p.placing_triangulation(); triangulation
(<0,2,3,4,6,7>, <0,2,3,4,6,12>, <0,2,3,4,7,13>, <0,2,3,4,12,13>,
 <0,2,3,6,7,13>, <0,2,3,6,12,13>, <0,2,4,6,7,13>, <0,2,4,6,12,13>,
 <0,3,4,6,7,12>, <0,3,4,7,12,13>, <0,3,6,7,12,13>, <0,4,6,7,12,13>,
 <1,3,4,5,6,12>, <1,3,4,6,11,12>, <1,3,4,7,11,13>, <1,3,4,11,12,13>,
 <1,3,6,7,11,13>, <1,3,6,11,12,13>, <1,4,6,7,11,13>, <1,4,6,11,12,13>,
 <3,4,6,7,11,12>, <3,4,7,11,12,13>, <3,6,7,11,12,13>, <4,6,7,11,12,13>)
sage: sum(p.volume(t) for t in triangulation)
42
sage: p0 = PointConfiguration([(0,0),(+1,0),(-1,0),(0,+1),(0,-1)])
sage: p0.pushing_triangulation(point_order=[1,2,0,3,4])
(<1,2,3>, <1,2,4>)
sage: p0.pushing_triangulation(point_order=[0,1,2,3,4])
(<0,1,2>, <0,1,4>, <0,2,3>, <0,2,4>)

The same triangulation with renumbered points 0->4, 1->0, etc:

sage: p1 = PointConfiguration([(0,0),(+1,0),(-1,0),(0,+1),(0,-1)])
sage: p1.pushing_triangulation(point_order=[4,0,1,2,3])
(<0,2,4>, <0,3,4>, <1,2,4>, <1,3,4>)

plot(**kwds)

Produce a graphical representation of the point configuration.

EXAMPLES:

sage: p = PointConfiguration([(0,0),(0,1),(1,0),(1,1),(-1,1)])
sage: p.plot(axes=False)

Graphics object consisting of 5 graphics primitives

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:
• *negative* – integer. The indices of points.

OUTPUT:
A tuple of all circuits with $C_\text{–} = \text{negative}$.

EXAMPLES:

```python
sage: p = PointConfiguration([(1,0,0),(0,1,0),(0,0,1),(-2,0,-1),(-2,-1,0),(-3,-
˓→1,-1),(1,1,1),(-1,0,0),(0,0,0)])
sage: sorted(p.positive_circuits(8))
[(0, 1, 2, 5), (0, 1, 4), (0, 2, 3), (0, 3, 4, 6), (0, 5, 6), (0, 7)]
sage: p.positive_circuits(0,5,6)
((8,),)
```

**pushing_triangulation** *(point_order=None)*

Construct the placing (pushing) triangulation.

INPUT:
• point_order – list of points or integers. The order in which the points are to be placed. If not given, the points will be placed in some arbitrary order that attempts to produce a small number of simplices.

OUTPUT:
A Triangulation.

EXAMPLES:

```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> )
sage: pc.placing_triangulation(point_order=(3,2,1,4,0)) ( (<0,1,4>, <1,2,3>, <1,3,4> )
sage: pc.placing_triangulation(point_order=[pc[1],pc[3],pc[4],pc[0]]) ( (<0,1,4>, <1,3,4> )
sage: U=matrix([ 0, 0, 0, 0, 0, 2, 4,-1, 1, 1, 0, 0, 1, 0,
˓→0, 0, 0, 1, 0, 0,-1, 0, 0, 0, 0, 0, 0, 0],
˓→0, 2, 0, 0, 0, 0,-1, 0, 1, 0, 1, 0, 0, 1],
˓→0, 1, 1, 0, 0, 1, 0,-2, 1, 0, 0,-1, 1, 1],
˓→0, 0, 0, 0, 1, 0,-1, 0, 0, 0, 0, 0, 0, 0])
sage: p = PointConfiguration(U.columns())
sage: triangulation = p.placing_triangulation(); triangulation
( (<0,2,3,4,6,7>, <0,2,3,4,6,12>, <0,2,3,4,7,13>, <0,2,3,4,12,13>,
  <0,2,3,6,7,13>, <0,2,3,6,12,13>, <0,2,4,6,7,13>, <0,2,4,6,12,13>,
  <0,3,4,6,7,12>, <0,3,4,7,12,13>, <0,3,6,7,12,13>, <0,4,6,7,12,13>,
  <1,3,4,5,6,12>, <1,3,4,6,11,12>, <1,3,4,7,11,13>, <1,3,4,11,12,13>,
  <1,3,6,7,11,13>, <1,3,6,11,12,13>, <1,4,6,7,11,13>, <1,4,6,11,12,13>,
  <3,4,6,7,11,12>, <3,4,7,11,12,13>, <3,6,7,11,12,13>, <4,6,7,11,12,13> )
sage: sum(p.volume(t) for t in triangulation)
42
```

(continues on next page)
The same triangulation with renumbered points 0->4, 1->0, etc:

```python
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>)
```

```
restrict_to_connected_triangulations(connected=True)
Restrict to connected triangulations.

NOTE:
Finding non-connected triangulations requires the optional TOPCOM package.

INPUT:
  • connected – boolean. Whether to restrict to triangulations that are connected by bistellar flips to the regular triangulations.

OUTPUT:
A new `PointConfiguration` with the same points, but whose triangulations will all be in the connected component. See `PointConfiguration` for details.

EXAMPLES:
```
sage: p = PointConfiguration([[(0,0),(0,1),(1,0),(1,1),(-1,-1)]])
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_all = p.restrict_to_connected_triangulations(connected=False)  # optional - topcom
sage: len(p_all.triangulations_list())  # optional - topcom
4
sage: p == p_all.restrict_to_connected_triangulations(connected=True)  # optional - topcom
True
sage: PointConfiguration.set_engine('internal')
```

```
restrict_to_fine_triangulations(fine=True)
Restrict to fine triangulations.

INPUT:
  • fine – boolean. Whether to restrict to fine triangulations.

OUTPUT:
A new `PointConfiguration` with the same points, but whose triangulations will all be fine. See `PointConfiguration` for details.
EXAMPLES:

```sage
sage: p = PointConfiguration([[0,0],[0,1],[1,0],[1,1],[-1,-1]])
sage: p
A point configuration in affine 2-space over Integer Ring consisting of 5 points. The triangulations of this point configuration are assumed to be connected, not necessarily fine, not necessarily regular.

sage: len(p.triangulations_list())
4
sage: p_fine = p.restrict_to_fine_triangulations()
sage: len(p.triangulations_list())
4
sage: p == p_fine.restrict_to_fine_triangulations(fine=False)
True
```

**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
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: 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:**

```python
sage: p = PointConfiguration([[(0,0), (0,1), (1,0), (1,1), (-1,-1)]])
sage: len(list(p.triangulations()))
4
sage: p_star = p.restrict_to_star_triangulations(0)
sage: p_star is p.restrict_to_star_triangulations((0,0))
True
sage: p_star.triangulations_list()
[[(0,1,3), (0,1,4), (0,2,3), (0,2,4)]

sage: p_newstar = p_star.restrict_to_star_triangulations(1)  # pick different origin
sage: p_newstar.triangulations_list()
[[(1,2,3), (1,2,4)]

sage: p == p_star.restrict_to_star_triangulations(star=None)
True
```

**restricted_automorphism_group()**

Return the restricted automorphism group.

First, let the linear automorphism group be the subgroup of the affine group $AGL(d, \mathbb{R}) = GL(d, \mathbb{R}) \times \mathbb{R}^d$ 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 nonnegative integers. The indexing of the permutation group is chosen to be shifted by +1. That is, the transposition $(i,j)$ in the permutation group corresponds to exchange of self[i-1] and self[j-1].

**EXAMPLES:**

```python
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:

```python
sage: square = PointConfiguration([[(3/4,3/4), (1,1), (1,-1), (-1,-1), (-1,1)]]

sage: square.restricted_automorphism_group()
```

(continues on next page)
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]])
poly = p.secondary_polytope()
poly.vertices_matrix()
[1 1 3 3 5]
[3 5 1 4 1]
[4 2 5 2 4]
[2 4 2 5 4]
[5 3 4 1 1]
poly.Vrepresentation()
(A vertex at (1, 3, 4, 2, 5),
 A vertex at (1, 5, 2, 4, 3),
 A vertex at (3, 1, 5, 2, 4),
 A vertex at (3, 4, 2, 5, 1),
 A vertex at (5, 1, 4, 4, 1))
poly.Hrepresentation()
(An equation (0, 0, 1, 2, 1) x - 13 == 0,
 An equation (1, 0, 0, 2, 2) x - 15 == 0,
 An equation (0, 1, 0, -3, -2) x + 13 == 0,
 An inequality (0, 0, 0, -1, -1) x + 7 >= 0,
 An inequality (0, 0, 0, 1, 0) x - 2 >= 0,
 An inequality (0, 0, 0, -2, -1) x + 11 >= 0,
 An inequality (0, 0, 0, 3, 2) x - 14 >= 0)
```

classmethod set_engine(engine='auto')
Set the engine used to compute triangulations.

INPUT:

• engine – either ‘auto’ (default), ‘internal’, or ‘topcom’. The latter two instruct this package to always use its own triangulation algorithms or TOPCOM’s algorithms, respectively. By default (‘auto’), internal routines are used.

EXAMPLES:

```sage
p = PointConfiguration([[0,0],[1,0],[1,0],[1,1],[-1,-1]])
p.set_engine('internal')  # to make doctests independent of TOPCOM
p.triangulate()
```
(continued from previous page)

```python
(<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)
sage: pc_nostar = pc.restrict_to_star_triangulations(None)
sage: pc_nostar
A point configuration in affine 2-space over Integer Ring consisting of 4 points. The triangulations of this point configuration are assumed to be connected, not necessarily fine, not necessarily regular.
sage: pc_nostar.star_center()
Traceback (most recent call last):
  ...
ValueError: The point configuration has no star center defined.
```

**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:**

```python
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)]
```

```
sage: p.set_engine('internal')  # optional - topcom
```

\textbf{triangulations} (\texttt{verbose} = \texttt{False})

Returns all triangulations.

- \texttt{verbose} – boolean (default: \texttt{False}). Whether to print out the TOPCOM interaction, if any.

\textbf{OUTPUT:}

A generator for the triangulations satisfying all the restrictions imposed. Each triangulation is returned as a \texttt{Triangulation} object.

\textbf{EXAMPLES:}

```
sage: p = PointConfiguration([[0,0],[0,1],[1,0],[1,1],[-1,-1]])
sage: iter = p.triangulations()
sage: next(iter)
(<1,3,4>, <2,3,4>)
sage: next(iter)
(<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>)
sage: next(iter)
(<1,2,3>, <1,2,4>)
sage: next(iter)
(<0,1,2>, <0,1,4>, <0,2,4>, <1,2,3>)
sage: p.triangulations_list()
[(<1,3,4>, <2,3,4>),
 (<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>),
 (<1,2,3>, <1,2,4>),
 (<0,1,2>, <0,1,4>, <0,2,4>, <1,2,3>)]
```

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>)
```

(continues on next page)
triangulations_list(verbosе=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)

Find n! times the n-volume of a simplex of dimension n.

INPUT:

• simplex (optional argument) – a simplex from a triangulation T specified as a list of point indices.

OUTPUT:

• If a simplex was passed as an argument: n!*(volume of simplex).

• Without argument: n!*(the total volume of the convex hull).

EXAMPLES:

The volume of the standard simplex should always be 1:

```
sage: p = PointConfiguration([[0,0],[1,0],[0,1],[1,1]])
sage: p.volume([0,1,2])
1
sage: simplex = p.triangulate()[0]  # first simplex of triangulation
sage: p.volume(simplex)
1
```
The square can be triangulated into two minimal simplices, so in the “integral” normalization its volume equals two:

```python
sage: p.volume()
2
```

**Note:** We return \(n! \times \text{(metric volume of the simplex)}\) 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)
```

(continues on next page)
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:

```
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:

```
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)
[]
```

```python
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()`.

**Warning:** You should not create `Point` objects manually. The constructor of `PointConfiguration_base` takes care of this for you.
```

INPUT:

- `point_configuration` – `PointConfiguration_base`. The point configuration to which the point belongs.
- `i` – integer. The index of the point in the point configuration.
- `projective` – the projective coordinates of the point.
- `affine` – the affine coordinates of the point.
- `reduced` – the reduced (with linearities removed) coordinates of the point.

EXAMPLES:

```
sage: pc = PointConfiguration([[0,0]])
sage: from sage.geometry.triangulation.base import Point
sage: Point(pc, 123, (0,0,1), (0,0), ()
P(0, 0)
```

3.2. Base classes for triangulations 585
affine()
Return the affine coordinates of the point in the ambient space.

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)
```

index()
Return the index of the point in the point configuration.

EXAMPLES:

```sage
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:

```sage
sage: pc = PointConfiguration([ (0,0), (1,0), (0,1) ])
sage: p = pc.point(0)
sage: p
P(0, 0)
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:
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)

reduced_affine()
Return the affine coordinates of the point on the hyperplane spanned by the point configuration.

OUTPUT:
A tuple containing the coordinates.

EXAMPLES:

sage: pc = PointConfiguration([[10, 0, 1], [10, 0, 0], [10, 2, 3]])
sage: p = pc.point(2); p
P(10, 2, 3)
sage: p.affine()
(10, 2, 3)
sage: p.projective()
(10, 2, 3, 1)
sage: p.reduced_affine()
(2, 2)
sage: p.reduced_projective()
(2, 2, 1)
sage: p.reduced_affine_vector()
(2, 2)

reduced_affine_vector()
Return the affine coordinates of the point on the hyperplane spanned by the point configuration.

OUTPUT:
A tuple containing the coordinates.

EXAMPLES:

sage: pc = PointConfiguration([[10, 0, 1], [10, 0, 0], [10, 2, 3]])
sage: p = pc.point(2); p
P(10, 2, 3)
sage: p.affine()
(10, 2, 3)
sage: p.projective()
(10, 2, 3, 1)
sage: p.reduced_affine()
(2, 2)
sage: p.reduced_projective()
(2, 2, 1)

(continues on next page)
(continued from previous page)

```
(2, 2, 1)
sage: p.reduced_affine_vector()
(2, 2)
```

**reduced_projective()**

Return the projective coordinates of the point on the hyperplane spanned by the point configuration.

**OUTPUT:**

A tuple containing the coordinates.

**EXAMPLES:**

```
sage: pc = PointConfiguration([[10, 0, 1], [10, 0, 0], [10, 2, 3]])
sage: p = pc.point(2); p
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)
```

**reduced_projective_vector()**

Return the affine coordinates of the point on the hyperplane spanned by the point configuration.

**OUTPUT:**

A tuple containing the coordinates.

**EXAMPLES:**

```
sage: pc = PointConfiguration([[10, 0, 1], [10, 0, 0], [10, 2, 3]])
sage: p = pc.point(2); p
P(10, 2, 3)
sage: p.affine()
(10, 2, 3)
sage: p.projective()
(10, 2, 3, 1)
sage: p.reduced_affine()
(2, 2)
sage: p.reduced_projective()
(2, 2, 1)
sage: p.reduced_affine_vector()
(2, 2)
sage: type(p.reduced_affine_vector())
<type 'sage.modules.vector_rational_dense.Vector_rational_dense'>
```

**class** `sage.geometry.triangulation.base.PointConfiguration_base`

**Bases:** `sage.structure.parent.Parent`

The cython abstract base class for `PointConfiguration`. 
ambient_dim()
Return the dimension of the ambient space of the point configuration.
See also dimension()
EXAMPLES:
```
sage: p = PointConfiguration([[0,0,0]])
sage: p.ambient_dim()
3
sage: p.dim()
0
```

base_ring()
Return the base ring, that is, the ring containing the coordinates of the points.
OUTPUT:
A ring.
EXAMPLES:
```
sage: p = PointConfiguration([(0,0)])
sage: p.base_ring()
Integer Ring
sage: p = PointConfiguration([(1/2,3)])
sage: p.base_ring()
Rational Field
sage: p = PointConfiguration([(0.2, 5)])
sage: p.base_ring()
Real Field with 53 bits of precision
```

dim()
Return the actual dimension of the point configuration.
See also ambient_dim()
EXAMPLES:
```
sage: p = PointConfiguration([[0,0,0]])
sage: p.ambient_dim()
3
sage: p.dim()
0
```

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, 1, 0, 0, 1, 0],
               [0, 0, 0, 1, 0, 0, -1, 0, 0, 0, 0, 0, 0, 0],
               [0, 2, 0, 0, 0, 0, -1, 0, 1, 0, 1, 0, 0, 1],
               [0, 1, 1, 0, 0, 1, 0, -2, 1, 0, 0, -1, 1, 1],
               [0, 0, 0, 0, 1, 0, -1, 0, 0, 0, 0, 0, 0, 0])

sage: pc = PointConfiguration(U.columns())

sage: pc.simplex_to_int([1, 3, 4, 7, 10, 13])
1678

sage: pc.int_to_simplex(1678)
(1, 3, 4, 7, 10, 13)
```

**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 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:

```python
sage: pconfig = PointConfiguration([[0,0],[0,1],[1,0],[1,1],[-1,-1]])
sage: list(pconfig)
[Point(0, 0), Point(0, 1), Point(1, 0), Point(1, 1), Point(-1, -1)]
sage: [ p for p in pconfig.points() ]
[Point(0, 0), Point(0, 1), Point(1, 0), Point(1, 1), Point(-1, -1)]
sage: pconfig.point(0)
Point(0, 0)
sage: pconfig[0]
Point(0, 0)
sage: pconfig.point(1)
Point(0, 1)
sage: pconfig.point( pconfig.n_points()-1 )
Point(-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:

```python
sage: pconfig = PointConfiguration([[0,0],[0,1],[1,0],[1,1],[-1,-1]])
sage: list(pconfig)
[Point(0, 0), Point(0, 1), Point(1, 0), Point(1, 1), Point(-1, -1)]
sage: [ p for p in pconfig.points() ]
[Point(0, 0), Point(0, 1), Point(1, 0), Point(1, 1), Point(-1, -1)]
sage: pconfig.point(0)
Point(0, 0)
sage: pconfig[0]
Point(0, 0)
sage: pconfig.point(1)
Point(0, 1)
sage: pconfig.point( pconfig.n_points()-1 )
Point(-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()}.  

EXAMPLES:

```python
sage: p = PointConfiguration([[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
```

(continues on next page)

(continued from previous page)

```python
sage: p.reduced_projective_vector_space()
Vector space of dimension 2 over Rational Field
```

**reduced_projective_vector_space()**

Return the vector space that is spanned by the homogeneous coordinates.

**OUTPUT:**

A vector space over the fraction field of `base_ring()`.

**EXAMPLES:**

```python
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
```

**simplex_to_int(simplex)**

Returns an integer that uniquely identifies the given simplex.

See also the inverse method `int_to_simplex()`.

The enumeration is compatible with [PUNTOS].

**INPUT:**

- `simplex` – iterable, for example a list. The elements are the vertex indices of the simplex.

**OUTPUT:**

An integer that uniquely specifies the simplex.

**EXAMPLES:**

```python
sage: U=matrix([
      ....: [ 0, 0, 0, 0, 2, 4,-1, 1, 1, 0, 1, 0],
      ....: [ 0, 0, 0, 1, 0,-1, 0, 0, 0, 0, 0, 0],
      ....: [ 0, 2, 0, 0, 0,-1, 0, 1, 0, 1, 0, 1],
      ....: [ 0, 1, 1, 0, 0, 1, 0,-2, 1, 0, 0, 1],
      ....: [ 0, 0, 0, 0, 1, 0,-1, 0, 0, 0, 0, 0]
      ....: ])
sage: pc = PointConfiguration(U.columns())
sage: pc.simplex_to_int([1,3,4,7,10,13])
1678
sage: pc.int_to_simplex(1678)
(1, 3, 4, 7, 10, 13)
```

592 Chapter 3. Triangulations
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.

class `sage.geometry.triangulation.element.Triangulation`:

A triangulation of a `PointConfiguration`.

**Warning:** You should never create `Triangulation` objects manually. See `triangulate()` and `triangulations()` to triangulate point configurations.

- **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]])
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')
sage: triangulation
(<0,1,2,7>, <0,1,5,7>, <0,2,3,7>, <0,3,4,7>, <0,4,5,7>, <1,5,6,7>)
sage: triangulation.boundary()
frozenset({(0, 1, 2), (0, 1, 5), (0, 2, 3), (0, 3, 4), (0, 4, 5), (1, 2, 7), (1, 5, 6), (1, 6, 7), (2, 3, 7), (3, 4, 7), (4, 5, 7), (5, 6, 7)})
sage: triangulation.interior_facets()
frozenset({(0, 1, 7), (0, 2, 7), (0, 3, 7), (0, 4, 7), (0, 5, 7)})
```

enumerate_simplices()
Return the enumerated simplices.

OUTPUT:
A tuple of integers that uniquely specifies the triangulation.

EXAMPLES:

```
sage: pc = PointConfiguration(matrix([
.....: [ 0, 0, 0, 0, 0, 2, 4,-1, 1, 1, 0, 0, 1, 0],
.....: [ 0, 0, 0, 1, 0, 0,-1, 0, 0, 0, 0, 0, 0, 0],
.....: [ 0, 2, 0, 0, 0, 0,-1, 0, 1, 0, 0, 0, 0, 0],
.....: [ 0, 1, 1, 0, 0, 1,-2, 1, 0, 0,-1, 1, 1, 1],
.....: [ 0, 0, 0, 0, 1, 0,-1, 0, 0, 0, 0, 0, 0, 0]
]).columns())
sage: triangulation = pc.lexicographic_triangulation()
sage: triangulation.enumerate_simplices()
(1678, 1688, 1769, 1779, 1895, 1905, 2112, 2143, 2234, 2360, 2555, 2580, 2610, 2626, 2650, 2652, 2654, 2661, 2663, 2667, 2685, 2755, 2757, 2759, 2766, 2768, 2772, 2811, 2881, 2883, 2885, 2892, 2894, 2898)
```

You can recreate the triangulation from this list by passing it to the constructor:

```
sage: from sage.geometry.triangulation.point_configuration import Triangulation
sage: Triangulation([1678, 1688, 1769, 1779, 1895, 1905, 2112, 2143,
.....: 2234, 2360, 2555, 2580, 2610, 2626, 2650, 2652, 2654, 2661, 2663,
.....: 2667, 2685, 2755, 2757, 2759, 2766, 2768, 2772, 2811, 2881, 2883, 2885, 2892, 2894, 2898],
```
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, 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

\textbf{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.

\textbf{OUTPUT:}

The phi vector of self.

\textbf{EXAMPLES:}

\begin{verbatim}
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)
\end{verbatim}

\textbf{interior\_facets()}

Return the interior facets of the triangulation.

\textbf{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.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: triangulation = polytopes.cube().triangulate(engine='internal')
sage: triangulation
(<0,1,2,7>, <0,1,5,7>, <0,2,3,7>, <0,3,4,7>, <0,4,5,7>, <1,5,6,7>)
sage: triangulation.boundary()
frozenset({(0, 1, 2), (0, 1, 5), (0, 2, 3), (0, 3, 4), (0, 4, 5), (1, 2, 7), (1, 5, 6), (1, 6, 7), (2, 3, 7), (3, 4, 7), (4, 5, 7), (5, 6, 7)})
sage: triangulation.interior_facets()
frozenset({(0, 1, 7), (0, 2, 7), (0, 3, 7), (0, 4, 7), (0, 5, 7), (1, 5, 7)})
\end{verbatim}

\textbf{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: triangulation
(<0,1,3>, <1,2,3>)
sage: N = triangulation.normal_cone(); N
4-d cone in 4-d lattice
sage: N.rays()
( 0, 0, 0, -1),
( 0, 0, 1, 1),
( 0, 0, -1, -1),
( 1, 0, 0, 1),
(-1, 0, 0, -1),
( 0, 1, 0, -1),
( 0, -1, 0, 1)
in Ambient free module of rank 4
over the principal ideal domain Integer Ring
sage: N.dual().rays()
(1, -1, 1, -1)
in Ambient free module of rank 4
over the principal ideal domain Integer Ring
```

```plot(**kwds)`

Produce a graphical representation of the triangulation.

EXAMPLES:

```sage
sage: p = PointConfiguration([[0,0],[0,1],[1,0],[1,1],[-1,-1]])
sage: triangulation = p.triangulate()
sage: triangulation
(<1,3,4>, <2,3,4>)
sage: triangulation.plot(axes=False)
Graphics object consisting of 12 graphics primitives
```

```point_configuration()`

Returns the point configuration underlying the triangulation.

EXAMPLES:

```sage
sage: pconfig = PointConfiguration([[0,0],[0,1],[1,0]])
sage: pconfig
A point configuration in affine 2-space over Integer Ring
consisting of 3 points. The triangulations of this point
configuration are assumed to be connected, not necessarily
```
sage: triangulation = pconfig.triangulate()
sage: triangulation
(<0,1,2>)
sage: triangulation.point_configuration()
A point configuration in affine 2-space over Integer Ring consisting of 3 points. The triangulations of this point configuration are assumed to be connected, not necessarily fine, not necessarily regular.
sage: pconfig == triangulation.point_configuration()
True

simplicial_complex()
Return a simplicial complex from a triangulation of the point configuration.

OUTPUT:
A SimplicialComplex.

EXAMPLES:
sage: p = polytopes.cuboctahedron()
sage: sc = p.triangulate(engine='internal').simplicial_complex()
sage: sc
Simplicial complex with 12 vertices and 16 facets

Any convex set is contractable, so its reduced homology groups vanish:
sage: sc.homology()
{0: 0, 1: 0, 2: 0, 3: 0}

sage.geometry.triangulation.element.triangulation_render_2d(triangulation, **kwds)
Return a graphical representation of a 2-d triangulation.

INPUT:
• triangulation – a Triangulation.

• **kwds – keywords that are passed on to the graphics primitives.

OUTPUT:
A 2-d graphics object.

EXAMPLES:
sage: points = PointConfiguration([[0,0],[0,1],[1,0],[1,1],[-1,-1]])
sage: triang = points.triangulate()
sage: triang.plot(axes=False, aspect_ratio=1)  # indirect doctest
Graphics object consisting of 12 graphics primitives

sage.geometry.triangulation.element.triangulation_render_3d(triangulation, **kwds)
Return a graphical representation of a 3-d triangulation.

INPUT:
• triangulation – a Triangulation.

• **kwds – keywords that are passed on to the graphics primitives.
OUTPUT:

A 3-d graphics object.

EXAMPLES:

```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)  # indirect doctest
Graphics3d Object
```
4.1 Convex Sets

class sage.geometry.convex_set.AffineHullProjectionData(image=None, projection_linear_map=None, projection_translation=None, section_linear_map=None, section_translation=None):

    Bases: object

class sage.geometry.convex_set.ConvexSet_base:

    Bases: sage.structure.sage_object.SageObject, sage.sets.set.Set_base

    Abstract base class for convex sets.

    affine_hull(*args, **kwds)
    Return the affine hull of self as a polyhedron.

    EXAMPLES:

        sage: from sage.geometry.convex_set import ConvexSet_compact
        sage: class EmbeddedDisk(ConvexSet_compact):
        ....:     def an_affine_basis(self):
        ....:         return [vector([1, 0, 0]), vector([1, 1, 0]), vector([1, 0, 1])]
        sage: O = EmbeddedDisk()
        sage: O.dim()  # doctest: +ELLIPSIS
        2
        sage: O.affine_hull()
        A 2-dimensional polyhedron in QQ^3 defined as the convex hull of 1 vertex and 2 lines

    affine_hull_projection(as_convex_set=None, as_affine_map=False, orthogonal=False, orthonormal=False, extend=False, minimal=False, return_all_data=False, **kwds)

    Return self projected into its affine hull.

    Each convex set is contained in some smallest affine subspace (possibly the entire ambient space) – its affine hull. We provide an affine linear map that projects the ambient space of the convex set to the standard Euclidean space of dimension of the convex set, which restricts to a bijection from the affine hull.

    The projection map is not unique; some parameters control the choice of the map. Other parameters control the output of the function.

    EXAMPLES:
sage: P = Polyhedron(vertices=[[1, 0], [0, 1]])
sage: ri_P = P.relative_interior(); ri_P
Relative interior of a 1-dimensional polyhedron in ZZ^2 defined as the convex
hull of 2 vertices
sage: ri_P.affine_hull_projection(as_affine_map=True)
(Vector space morphism represented by the matrix:
[1]
[0]
Domain: Vector space of dimension 2 over Rational Field
Codomain: Vector space of dimension 1 over Rational Field,
(0))
sage: P_aff = P.affine_hull_projection(); P_aff
A 1-dimensional polyhedron in ZZ^1 defined as the convex hull of 2 vertices
sage: ri_P_aff = ri_P.affine_hull_projection(); ri_P_aff
Relative interior of a 1-dimensional polyhedron in QQ^1 defined as the convex
hull of 2 vertices
sage: ri_P_aff.closure() == P_aff
True

ambient()

Return the ambient convex set or space.

The default implementation delegates to ambient_vector_space().

EXAMPLES:

sage: from sage.geometry.convex_set import ConvexSet_base
sage: class ExampleSet(ConvexSet_base):
    ....: def ambient_vector_space(self, base_field=None):
    ....:     return (base_field or QQ)^2001
sage: ExampleSet().ambient()
Vector space of dimension 2001 over Rational Field

ambient_dim()

Return the dimension of the ambient convex set or space.

The default implementation obtains it from ambient().

EXAMPLES:

sage: from sage.geometry.convex_set import ConvexSet_base
sage: class ExampleSet(ConvexSet_base):
    ....: def ambient(self):
    ....:     return QQ^7
sage: ExampleSet().ambient_dim()
7

ambient_dimension()

Return the dimension of the ambient convex set or space.

This is the same as ambient_dim().

EXAMPLES:

sage: from sage.geometry.convex_set import ConvexSet_base
sage: class ExampleSet(ConvexSet_base):
    ....: def ambient(self):
    ....:     return QQ^7
sage: ExampleSet().ambient_dimension()
7
ambient_vector_space(base_field=None)
Return the ambient vector space.
Subclasses must provide an implementation of this method.
The default implementations of ambient(), ambient_dim(), ambient_dimension() use this method.
EXAMPLES:

sage: from sage.geometry.convex_set import ConvexSet_base
sage: C = ConvexSet_base()
Traceback (most recent call last):
...  
NotImplementedError: <abstract method ambient_vector_space at ...>

an_affine_basis()
Return points that form an affine basis for the affine hull.
The points are guaranteed to lie in the topological closure of self.
EXAMPLES:

sage: from sage.geometry.convex_set import ConvexSet_base
sage: C = ConvexSet_base()
Traceback (most recent call last):
...  
TypeError: 'NotImplementedType' object is not callable

an_element()
Return a point of self.
If self is empty, an EmptySetError will be raised.
The default implementation delegates to _some_elements_().
EXAMPLES:

sage: from sage.geometry.convex_set import ConvexSet_compact
sage: class BlueBox(ConvexSet_compact):
    ....:    def _some_elements_(self):
    ....:        yield 'blue'
    ....:        yield 'cyan'
sage: BlueBox().an_element()
'blue'

cardinality()
Return the cardinality of this set.
OUTPUT:
Either an integer or Infinity.

4.1. Convex Sets
603
EXAMPLES:

```python
sage: p = LatticePolytope([], lattice=ToricLattice(3).dual()); p
-1-d lattice polytope in 3-d lattice M
sage: p.cardinality()
0
sage: q = Polyhedron(ambient_dim=2); q
The empty polyhedron in ZZ^2
sage: q.cardinality()
0
sage: r = Polyhedron(rays=[(1, 0)]); r
A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex and 1
  ray
sage: r.cardinality()
+Infinity
```

cartesian_product(other)
Return the Cartesian product.

INPUT:

• other – another convex set

OUTPUT:

The Cartesian product of self and other.

closure()
Return the topological closure of self.

EXAMPLES:

```python
sage: from sage.geometry.convex_set import ConvexSet_closed
sage: C = ConvexSet_closed()
sage: C.closure()
is C
True
```

codim()
Return the codimension of self in self.ambient().

EXAMPLES:

```python
sage: P = Polyhedron(vertices=[(1,2,3)], rays=[(1,0,0)])
sage: P.codimension()
2
```

An alias is codim():

```python
sage: P.codim()
2
```

codimension()
Return the codimension of self in self.ambient().

EXAMPLES:

```python
sage: P = Polyhedron(vertices=[(1,2,3)], rays=[(1,0,0)])
sage: P.codimension()
2
```
An alias is \texttt{codim()}:

\begin{verbatim}
sage: P.codim()
sage: P.codim()
2
\end{verbatim}

\textbf{contains}(\texttt{point})
Test whether \texttt{self} contains the given \texttt{point}.

INPUT:
• \texttt{point} – a point or its coordinates

\textbf{dilation}(\texttt{scalar})
Return the dilated (uniformly stretched) set.

INPUT:
• \texttt{scalar} – A scalar, not necessarily in \texttt{base_ring()}

EXAMPLES:

\begin{verbatim}
sage: from sage.geometry.convex_set import ConvexSet_compact
class GlorifiedPoint(ConvexSet_compact):
    def __init__(self, p):
        self._p = p
    def ambient_vector_space(self):
        return self._p.parent().vector_space()
    def linear_transformation(self, linear_transf):
        return GlorifiedPoint(linear_transf * self._p)
sage: P = GlorifiedPoint(vector([2, 3]))
sage: P.dilation(10)._p
sage: P.dilation(10)._p
(20, 30)
\end{verbatim}

\textbf{dim()}
Return the dimension of \texttt{self}.

Subclasses must provide an implementation of this method or of the method \texttt{an_affine_basis()}.

\textbf{dimension()}
Return the dimension of \texttt{self}.

This is the same as \texttt{dim()}

EXAMPLES:

\begin{verbatim}
sage: from sage.geometry.convex_set import ConvexSet_base
class ExampleSet(ConvexSet_base):
    def dim(self):
        return 42
sage: ExampleSet().dimension()
sage: ExampleSet().dimension()
42
\end{verbatim}

\textbf{interior()}
Return the topological interior of \texttt{self}.

EXAMPLES:

\begin{verbatim}
sage: from sage.geometry.convex_set import ConvexSet_open
sage: C = ConvexSet_open()
\end{verbatim}

(continues on next page)
sage: C.interior() is C
True

**intersection**(other)
Return the intersection of self and other.

**INPUT:**
- other – another convex set

**OUTPUT:**
The intersection.

**is_closed()**
Return whether self is closed.

The default implementation of this method only knows that the empty set, a singleton set, and the ambient space are closed.

**OUTPUT:**
Boolean.

**EXAMPLES:**

```python
sage: from sage.geometry.convex_set import ConvexSet_base
class ExampleSet(ConvexSet_base):
    ...: def dim(self):
    ...:     return 0
sage: ExampleSet().is_closed()
True
```

**is_compact()**
Return whether self is compact.

The default implementation of this method only knows that a non-closed set cannot be compact, and that the empty set and a singleton set are compact.

**OUTPUT:**
Boolean.

```python
sage: from sage.geometry.convex_set import ConvexSet_base
class ExampleSet(ConvexSet_base):
    ...: def dim(self):
    ...:     return 0
sage: ExampleSet().is_compact()
True
```

**is_empty()**
Test whether self is the empty set.

**OUTPUT:**
Boolean.

**EXAMPLES:**

```python
sage: p = LatticePolytope([], lattice=ToricLattice(3).dual()); p
-1-d lattice polytope in 3-d lattice M
sage: p.is_empty()
True
```
is_finite()

Test whether self is a finite set.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: p = LatticePolytope([], lattice=ToricLattice(3).dual()); p
-1-d lattice polytope in 3-d lattice M
sage: p.is_finite()
True
sage: q = Polyhedron(ambient_dim=2); q
The empty polyhedron in ZZ^2
sage: q.is_finite()
True
sage: r = Polyhedron(rays=[(1, 0)]); r
A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex and 1 →-ray
sage: r.is_finite()
False
```

is_full_dimensional()

Return whether self is full dimensional.

OUTPUT:

Boolean. Whether the polyhedron is not contained in any strict affine subspace.

EXAMPLES:

```
sage: c = Cone([(1,0)])
sage: c.is_full_dimensional()
False
sage: polytopes.hypercube(3).is_full_dimensional()
True
sage: Polyhedron(vertices=[(1,2,3)], rays=[(1,0,0)]).is_full_dimensional()
False
```

is_open()

Return whether self is open.

The default implementation of this method only knows that the empty set and the ambient space are open.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: from sage.geometry.convex_set import ConvexSet_base
sage: class ExampleSet(ConvexSet_base):
....:     def is_empty(self):
....:         return False
....:     def is_universe(self):
....:         return True
```

(continues on next page)
is_relatively_open()  
Return whether self is relatively open.

The default implementation of this method only knows that open sets are also relatively open, and in addition singletons are relatively open.

OUTPUT:
Boolean.

EXAMPLES:

```
sage: from sage.geometry.convex_set import ConvexSet_base
sage: class ExampleSet(ConvexSet_base):
    ....: def is_open(self):
    ....:     return True
sage: ExampleSet().is_relatively_open()
```

is_universe()  
Test whether self is the whole ambient space.

OUTPUT:
Boolean.

linear_transformation(linear_transf)  
Return the linear transformation of self.

INPUT:
• linear_transf – a matrix

relative_interior()  
Return the relative interior of self.

EXAMPLES:

```
sage: from sage.geometry.convex_set import ConvexSet_relatively_open
sage: C = ConvexSet_relatively_open()
```

some_elements()  
Return a list of some points of self.

If self is empty, an empty list is returned; no exception will be raised.

The default implementation delegates to _some_elements_().

EXAMPLES:

```
sage: from sage.geometry.convex_set import ConvexSet_compact
sage: class BlueBox(ConvexSet_compact):
    ....:     def _some_elements_(self):
    ....:         yield 'blue'
```
....:     yield 'cyan'
sage: BlueBox().some_elements()
['blue', 'cyan']

\texttt{translation}(\text{\textit{displacement}})
\text{Return the translation of } \text{\textit{self}} \text{ by a displacement vector.}

\textbf{INPUT:}
\begin{itemize}
\item \textit{displacement} – a displacement vector or a list/tuple of coordinates that determines a displacement vector
\end{itemize}

\textbf{class \texttt{sage.geometry.convex_set.ConvexSet_closed}}
\text{Bases: \texttt{sage.geometry.convex_set.ConvexSet_base}}

Abstract base class for closed convex sets.

\textbf{is\_closed}()
\text{Return whether } \text{\textit{self}} \text{ is closed.}

\textbf{OUTPUT:}
Boolean.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: hcube = polytopes.hypercube(5)
sage: hcube.is_closed()
True
\end{verbatim}

\textbf{is\_open}()
\text{Return whether } \text{\textit{self}} \text{ is open.}

\textbf{OUTPUT:}
Boolean.

\textbf{EXAMPLES:}

\begin{verbatim}
sage: hcube = polytopes.hypercube(5)
sage: hcube.is_open()
False

sage: zerocube = polytopes.hypercube(0)
sage: zerocube.is_open()
True
\end{verbatim}

\textbf{class \texttt{sage.geometry.convex_set.ConvexSet_compact}}
\text{Bases: \texttt{sage.geometry.convex_set.ConvexSet_closed}}

Abstract base class for compact convex sets.

\textbf{is\_compact}()
\text{Return whether } \text{\textit{self}} \text{ is compact.}

\textbf{OUTPUT:}
Boolean.

\textbf{EXAMPLES:}
is_relatively_open()
Return whether self is open.

OUTPUT:
Boolean.

EXAMPLES:

```python
sage: hcube = polytopes.hypercube(5)
sage: hcube.is_open()
False
sage: zerocube = polytopes.hypercube(0)
sage: zerocube.is_open()
True
```

is_universe()
Return whether self is the whole ambient space

OUTPUT:
Boolean.

EXAMPLES:

```python
sage: cross3 = lattice_polytope.cross_polytope(3)
sage: cross3.is_universe()
False
sage: point0 = LatticePolytope([[]]); point0
0-d reflexive polytope in 0-d lattice M
sage: point0.is_universe()
True
```

class sage.geometry.convex_set.ConvexSet_open
Bases: sage.geometry.convex_set.ConvexSet_relatively_open

Abstract base class for open convex sets.

is_closed()
Return whether self is closed.

OUTPUT:
Boolean.

EXAMPLES:

```python
sage: from sage.geometry.convex_set import ConvexSet_open
sage: class OpenBall(ConvexSet_open):
    .....:     def dim(self):
    .....:         return 3
    .....:     def is_universe(self):
    .....:         return False
```
sage: OpenBall().is_closed()
False

is_open()
Return whether self is open.

OUTPUT:
Boolean.

EXAMPLES:

sage: from sage.geometry.convex_set import ConvexSet_open
gage: b = ConvexSet_open()
sage: b.is_open()
True

class sage.geometry.convex_set.ConvexSet_relatively_open
Bases: sage.geometry.convex_set.ConvexSet_base

Abstract base class for relatively open convex sets.

is_open()
Return whether self is open.

OUTPUT:
Boolean.

EXAMPLES:

sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: ri_segment = segment.relative_interior()
sage: ri_segment.is_open()
False

is_relatively_open()
Return whether self is relatively open.

OUTPUT:
Boolean.

EXAMPLES:

sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: ri_segment = segment.relative_interior()
sage: ri_segment.is_relatively_open()
True
4.2 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
```

class `sage.geometry.linear_expression.LinearExpression`

```
Bases: sage.structure.element.ModuleElement

A linear expression.

A linear expression is just a linear polynomial in some (fixed) variables.

EXAMPLES:
```

```python
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
```

(continues on next page)
sage: L.zero()
0*x + 0*y + 0*z + 0
sage: a = L([12, 2/3, -1], -2)
sage: a - m
11*x - 4/3*y - 4*z - 6
sage: LZ.<x,y,z> = LinearExpressionModule(ZZ)
sage: a - LZ([2, -1, 3], 1)
10*x + 5/3*y - 4*z - 3

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)
sage: a = x + 2*y + 3*z + 4; a
x + 2*y + 3*z + 4
sage: a.change_ring(RDF)
1.0*x + 2.0*y + 3.0*z + 4.0
```

**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)
sage: linear = L([1, 2, 3], 4); linear
x + 2*y + 3*z + 4
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)
sage: linear = L([1, 2, 3], 4); linear
x + 2*y + 3*z + 4
sage: linear.A()
(1, 2, 3)
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)
sage: linear = L([1, 2, 3], 4); linear
x + 2*y + 3*z + 4
sage: linear.coefficients()
[4, 1, 2, 3]
```

**evaluate(point)**

Evaluate the linear expression.
INPUT:
• point – list/tuple/iterable of coordinates; the coordinates of a point

OUTPUT:
The linear expression $Ax + b$ evaluated at the point $x$.

EXAMPLES:

```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
```

\textbf{monomial_coefficients}(\texttt{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))
```

\textbf{class} \texttt{sage.geometry.linear_expression.LinearExpressionModule}(\texttt{base_ring, names=()})

\textbf{Bases:} \texttt{sage.structure.parent.Parent}, \texttt{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
Module of linear expressions in variables x, y, z over Rational Field
sage: L.an_element()
x + 0*y + 0*z + 0
```

\textbf{Element}

alias of \texttt{LinearExpression}

\textbf{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:

```python
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L = LinearExpressionModule(QQ, ('x', 'y', 'z'))
sage: L.ambient_module()
Vector space of dimension 3 over Rational Field
sage: M = LinearExpressionModule(ZZ, ('r', 's'))
sage: M.ambient_module()
Ambient free module of rank 2 over the principal ideal domain Integer Ring
sage: M.ambient_vector_space()
Vector space of dimension 2 over Rational Field
```

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:

```python
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L = LinearExpressionModule(QQ, ('x', 'y', 'z'))
sage: L.ambient_vector_space()
Vector space of dimension 3 over Rational Field
sage: M = LinearExpressionModule(ZZ, ('r', 's'))
sage: M.ambient_module()
Ambient free module of rank 2 over the principal ideal domain Integer Ring
sage: M.ambient_vector_space()
Vector space of dimension 2 over Rational Field
```

basis()  
Return a basis of self.

EXAMPLES:

```python
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L = LinearExpressionModule(QQ, ('x', 'y', 'z'))
sage: list(L.basis())
[x + 0*y + 0*z + 0,  
  0*x + y + 0*z + 0,  
  0*x + 0*y + z + 0,  
  0*x + 0*y + 0*z + 1]
```

change_ring(base_ring)  
Return a new module with a changed base ring.

INPUT:
• base_ring – a ring; the new base ring

OUTPUT:
A new linear expression over the new base ring.

EXAMPLES:

```python
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: M.<y> = LinearExpressionModule(ZZ)
```

```python
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'))
```

```python
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'))
```

```python
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'))
```

```python
sage: L.ngens()
3
```

random_element()

Return a random element.
4.3 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()

    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:

```python
sage: from sage.geometry.newton_polygon import NewtonPolygon
sage: NP = NewtonPolygon([[0,0], [1,1], [2,5]])
```

```python
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:

```python
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:

```python
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)
```

```python
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:
from sage.geometry.newton_polygon import NewtonPolygon

NP = NewtonPolygon([(0,0), (1,1), (2,5)]); NP

Finite Newton polygon with 3 vertices: (0, 0), (1, 1), (2, 5)

v = NP.vertices(); v

[(0, 0), (1, 1), (2, 5)]

class sage.geometry.newton_polygon.ParentNewtonPolygon

Bases: 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, ∞). These polygons are attached to polynomials or series over discrete valuation rings (e.g. padics).

EXAMPLES:

We specify here a Newton polygon by its vertices:

sage: from sage.geometry.newton_polygon import NewtonPolygon
sage: NewtonPolygon([(0,0), (1,1), (3,5)])

Finite Newton polygon with 3 vertices: (0, 0), (1, 1), (3, 5)

We note that the convex hull of the vertices is automatically computed:

sage: NewtonPolygon([(0,0), (1,1), (2,8), (3,5)])

Finite Newton polygon with 3 vertices: (0, 0), (1, 1), (3, 5)

Note that the value +Infinity is allowed as the second coordinate of a vertex:

sage: NewtonPolygon([(0,0), (1,Infinity), (2,8), (3,5)])

Finite Newton polygon with 2 vertices: (0, 0), (3, 5)

If last_slope is set, the returned Newton polygon is infinite and ends with an infinite line having the specified slope:

sage: NewtonPolygon([(0,0), (1,1), (2,8), (3,5)], last_slope=3)

Infinite Newton polygon with 3 vertices: (0, 0), (1, 1), (3, 5) ending by an infinite line of slope 3

Specifying a last slope may discard some vertices:

sage: NewtonPolygon([(0,0), (1,1), (2,8), (3,5)], last_slope=3/2)

Infinite Newton polygon with 2 vertices: (0, 0), (1, 1) ending by an infinite line of slope 3/2

(continues on next page)
Next, we define a Newton polygon by its slopes:

```python
sage: NP = NewtonPolygon([0, 1/2, 1/2, 2/3, 2/3, 2/3, 1, 1])
```

Finite Newton polygon with 5 vertices: (0, 0), (1, 0), (3, 1), (6, 3), (8, 5)

```python
sage: NP.slopes()
[0, 1/2, 1/2, 2/3, 2/3, 2/3, 1, 1]
```

By default, slopes are automatically sorted:

```python
sage: NP2 = NewtonPolygon([0, 1, 1/2, 2/3, 1/2, 2/3, 1, 2/3])
```

Finite Newton polygon with 5 vertices: (0, 0), (1, 0), (3, 1), (6, 3), (8, 5)

```python
sage: NP == NP2
True
```

except if the contrary is explicitly mentioned:

```python
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 than or equal last_slope (if specified) are discarded:

```python
sage: NP = NewtonPolygon([0, 1/2, 1/2, 2/3, 2/3, 2/3, 1, 1], last_slope=2/3)
```

Infinite Newton polygon with 3 vertices: (0, 0), (1, 0), (3, 1) ending by an
downward infinite line of slope 2/3

```python
sage: NP.slopes()
[0, 1/2, 1/2]
```

Be careful, do not confuse Newton polygons provided by this class with Newton polytopes. Compare:

```python
sage: NP = NewtonPolygon([(0,0), (1,45), (3,6)]); NP
```

Finite Newton polygon with 2 vertices: (0, 0), (3, 6)

```python
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

```python
sage: p.newton_polytope().vertices()
```

(A vertex at (0, 0), A vertex at (1, 45), A vertex at (3, 6))

**Element**

alias of `NewtonPolygon_element`
4.4 Relative Interiors of Polyhedra and Cones

class sage.geometry.relative_interior.RelativeInterior(polyhedron)
    Bases: sage.geometry.convex_set.ConvexSet_relatively_open

The relative interior of a polyhedron or cone

This class should not be used directly. Use methods relative_interior(), interior(), relative_interior(), interior() instead.

EXAMPLES:

```
sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: segment.relative_interior()
Relative interior of
da 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: octant = Cone([(1,0,0), (0,1,0), (0,0,1)])
sage: octant.relative_interior()
Relative interior of 3-d cone in 3-d lattice N
```

ambient()
    Return the ambient convex set or space.

EXAMPLES:

```
sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: ri_segment = segment.relative_interior(); ri_segment
Relative interior of
da 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: ri_segment.ambient()
Vector space of dimension 2 over Rational Field
```

ambient_dim()
    Return the dimension of the ambient space.

EXAMPLES:

```
sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: ri_segment = segment.relative_interior(); ri_segment
Relative interior of
da 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: ri_segment.ambient_dim()
2
```

ambient_vector_space(base_field=None)
    Return the ambient vector space.

EXAMPLES:

```
sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: ri_segment = segment.relative_interior(); ri_segment
Relative interior of
da 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: ri_segment.ambient_vector_space()
Vector space of dimension 2 over Rational Field
```
an_affine_basis()
Return points that form an affine basis for the affine hull.

The points are guaranteed to lie in the topological closure of self.

EXAMPLES:

```
sage: segment = Polyhedron([[1, 0], [0, 1]])
sage: segment.relative_interior().an_affine_basis()
[A vertex at (1, 0), A vertex at (0, 1)]
```

closure()
Return the topological closure of self.

EXAMPLES:

```
sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: ri_segment = segment.relative_interior(); ri_segment
Relative interior of a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: ri_segment.closure() is segment
True
```

dilation(scalar)
Return the dilated (uniformly stretched) set.

INPUT:

• scalar – A scalar

EXAMPLES:

```
sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: ri_segment = segment.relative_interior(); ri_segment
Relative interior of a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: A = ri_segment.dilation(2); A
Relative interior of a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: A.closure().vertices()
(A vertex at (2, 4), A vertex at (6, 8))
sage: B = ri_segment.dilation(-1/3); B
Relative interior of a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: B.closure().vertices()
(A vertex at (-1, -4/3), A vertex at (-1/3, -2/3))
sage: C = ri_segment.dilation(0); C
A 0-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex
sage: C.vertices()
(A vertex at (0, 0),)
```

dim()
Return the dimension of self.

EXAMPLES:
```python
sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: segment.dim()
1
sage: ri_segment = segment.relative_interior(); ri_segment
Relative interior of
a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: ri_segment.dim()
1

interior()
Return the interior of self.

EXAMPLES:
```n
```python
sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: ri_segment = segment.relative_interior(); ri_segment
Relative interior of a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: ri_segment.interior()
The empty polyhedron in ZZ^2

sage: octant = Cone([(1,0,0), (0,1,0), (0,0,1)])
sage: ri_octant = octant.relative_interior(); ri_octant
Relative interior of 3-d cone in 3-d lattice N
sage: ri_octant.interior() is ri_octant
True
```

```python
is_closed()
Return whether self is closed.

OUTPUT:
Boolean.

EXAMPLES:
```n
```python
sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: ri_segment = segment.relative_interior(); ri_segment
Relative interior of a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: ri_segment.is_closed()
False
```

```python
is_universe()
Return whether self is the whole ambient space

OUTPUT:
Boolean.

EXAMPLES:
```n
```python
sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: ri_segment = segment.relative_interior(); ri_segment
Relative interior of
a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
```
sage: ri_segment.is_universe()
False

linear_transformation(linear_transf, **kwds)
Return the linear transformation of self.

By [Roc1970], Theorem 6.6, the linear transformation of a relative interior is the relative interior of the linear transformation.

INPUT:
- linear_transf – a matrix
- **kwds – passed to the linear_transformation() method of the closure of self.

EXAMPLES:

```
sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: ri_segment = segment.relative_interior(); ri_segment
Relative interior of a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: T = matrix([[1, 1]])
sage: A = ri_segment.linear_transformation(T); A
Relative interior of a 1-dimensional polyhedron in ZZ^1 defined as the convex hull of 2 vertices
sage: A.closure().vertices()
((A vertex at (3), A vertex at (7)))
```

relative_interior()
Return the relative interior of self.

As self is already relatively open, this method just returns self.

EXAMPLES:

```
sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: ri_segment = segment.relative_interior(); ri_segment
Relative interior of a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: ri_segment.relative_interior() is ri_segment
True
```

translation(displacement)
Return the translation of self by a displacement vector.

INPUT:
- displacement – a displacement vector or a list/tuple of coordinates that determines a displacement vector

EXAMPLES:

```
sage: segment = Polyhedron([[1, 2], [3, 4]])
sage: ri_segment = segment.relative_interior(); ri_segment
Relative interior of a 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: t = vector([100, 100])
```

(continues on next page)
4.5 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)

```python
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 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:

```
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:

```
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:

```
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) of valency 1 when we define sigma:

```
sage: s2 = PermutationGroupElement('(1,3,5,8)(2,4,6)')
sage: r2 = PermutationGroupElement('(1,2)(3,4)(5,6)(8,15)')
```

(continues on next page)
sage: R2 = RibbonGraph(s2, r2); R1
Ribbon graph of genus 1 and 1 boundary components

sage: R2.sigma()
(1,3,5,8)(2,4,6)

This example is constructed by taking the bipartite graph of type \((3, 3)\):

sage: s3 = PermutationGroupElement('(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,15)(16,17,18)')
sage: r3 = PermutationGroupElement('(1,16)(2,13)(3,10)(4,17)(5,14)(6,11)(7,18)(8,15)(9,12)')
sage: R3 = RibbonGraph(s3, r3); R3
Ribbon graph of genus 1 and 3 boundary components

The labeling of the darts can omit some numbers:

sage: s4 = PermutationGroupElement('(3,5,10,12)')
sage: r4 = PermutationGroupElement('(3,10)(5,12)')
sage: R4 = RibbonGraph(s4, r4); R4
Ribbon graph of genus 1 and 1 boundary components

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.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]]
sage: S.boundary()
[[5, 14, 15, 8, 9, 12], [6, 11, 12, 9, 14, 5], [8, 15, 11, 6]]
sage: R5.homology_basis()
[[[5, 14], [13, 2], [1, 16], [17, 4]],
 [[6, 11], [10, 3], [1, 16], [17, 4]],
 [[8, 15], [13, 2], [1, 16], [18, 7]],
 [[9, 12], [10, 3], [1, 16], [18, 7]]]
We construct a ribbon graph corresponding to a genus 0 surface with 5 boundary components:

```python
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 $(2, 3)$:

```python
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:

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
Ribbon graph of genus 1 and 1 boundary components
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 *(σ, ρ)*, we contract the edge corresponding to the *k*-th transposition of ρ.

**INPUT:**

- *k* – non-negative integer; the position in ρ of the transposition that is going to be contracted

**OUTPUT:**

- a ribbon graph resulting from the contraction of that edge

**EXAMPLES:**

We start again with the one-holed torus ribbon graph:

```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: S1 = R1.contract_edge(1); S1
Ribbon graph of genus 1 and 1 boundary components
sage: S1.sigma()
(1,6,2,5)
sage: S1.rho()
(1,2)(5,6)
```

However, this ribbon graphs is formed only by loops and hence it cannot be longer reduced, we get an error if we try to contract a loop:

```python
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 *σ*. We contract precisely this edge and get a ribbon graph with no vertices 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); R2
Ribbon graph of genus 1 and 3 boundary components
sage: R2.sigma()
(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,15)(16,17,18,19)
sage: R2c = R2.contract_edge(9); 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
sage: E1 = R1.extrude_edge(1, 1, 2); E1
Ribbon graph of genus 1 and 1 boundary components
sage: E1.sigma()
(1,3,5)(2,8,6)(4,7)
sage: E1.rho()
(1,2)(3,4)(5,6)(7,8)
sage: E2 = R1.extrude_edge(1, 1, 3); E2
Ribbon graph of genus 1 and 1 boundary components
sage: E2.sigma()
(1,3,5)(2,8)(4,6,7)
sage: E2.rho()
(1,2)(3,4)(5,6)(7,8)

We can also extrude a contractible edge from a vertex. This new edge will end at a vertex of valency 1:

sage: E1p = R1.extrude_edge(0, 0, 0); E1p
Ribbon graph of genus 1 and 1 boundary components
sage: E1p.sigma()
(1,3,5,8)(2,4,6)
sage: E1p.rho()
(1,2)(3,4)(5,6)(7,8)
In the following example we first extrude one edge from a vertex of valency 3 generating a new vertex of valency 2. Then we extrude a new edge from this vertex of valency 2:

```
sage: s1 = PermutationGroupElement('(1,3,5)(2,4,6)')
sage: r1 = PermutationGroupElement('(1,2)(3,4)(5,6)')
sage: R1 = RibbonGraph(s1,r1); R1
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)
```

### genius() 
Return the genus of the thickening of `self`. 

**OUTPUT:**

- `g` – non-negative integer representing the genus of the thickening of the ribbon graph

**EXAMPLES:**

```
sage: s1 = PermutationGroupElement('(1,3,5)(2,4,6)')
sage: r1 = PermutationGroupElement('(1,2)(3,4)(5,6)')
sage: R1 = RibbonGraph(s1,r1)
sage: R1.genus()
1

sage: s3 = PermutationGroupElement('(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,15,\rightarrow 16)(17,18,19,20)(21,22,23,24)')
sage: R3 = RibbonGraph(s3,r3); R3.genus()
3
```

### homology_basis() 
Return an oriented basis of the first homology group of the graph. 

**OUTPUT:**

- A 2-dimensional array of ordered edges in the graph (given by pairs). The length of the first dimension is $\mu$. Each row corresponds to an element of the basis and is a circle contained in the graph.

**EXAMPLES:**

```
sage: R = RibbonGraph(0,6); R
Ribbon graph of genus 0 and 6 boundary components
sage: R.mu()
5
sage: R.homology_basis()
(continues on next page)
### 4.5. Ribbon Graphs

```python
[[[3, 4], [2, 1]],
 [[5, 6], [2, 1]],
 [[7, 8], [2, 1]],
 [[9, 10], [2, 1]],
 [[11, 12], [2, 1]]

sage: R = RibbonGraph(1,1); R
Ribbon graph of genus 1 and 1 boundary components

sage: R.mu()
2

sage: R.homology_basis()
[[[2, 5], [4, 1]], [[3, 6], [4, 1]]]

sage: H = R.reduced(); H
Ribbon graph of genus 1 and 1 boundary components

sage: H.sigma()
(2,3,5,6)

sage: H.rho()
(2,5)(3,6)

sage: H.homology_basis()
[[[2, 5]], [[3, 6]]]

sage: s3 = PermutationGroupElement('1,2,3,4,5,6,7,8,9,10,11,27,25,23,12,24,26,28')

sage: r3 = PermutationGroupElement('1,12,2,13,3,14,4,15,5,16,6,17,8,19,9,20,10,21,11,22,23,24,25,26,27,28')

sage: R3 = RibbonGraph(s3,r3); R3
Ribbon graph of genus 5 and 4 boundary components

sage: R3.mu()
13

sage: R3.homology_basis()
[[[2, 13], [12, 1]],
 [[3, 14], [12, 1]],
 [[4, 15], [12, 1]],
 [[5, 16], [12, 1]],
 [[6, 17], [12, 1]],
 [[7, 18], [12, 1]],
 [[8, 19], [12, 1]],
 [[9, 20], [12, 1]],
 [[10, 21], [12, 1]],
 [[11, 22], [12, 1]],
 [[12, 23], [12, 1]],
 [[13, 24], [12, 1]],
 [[25, 26], [12, 1]],
 [[27, 28], [12, 1]]]

sage: H3 = R3.reduced(); H3
Ribbon graph of genus 5 and 4 boundary components

sage: H3.sigma()
(2,3,5,6,7,8,9,10,11,27,25,23,24,26,28,13,14,15,16,17,18,19,20,21,22)

sage: H3.rho()

sage: H3.homology_basis()
[[[2, 13]],

```
make_generic()
Return a ribbon graph equivalent to self but where every vertex has valency 3.

OUTPUT:
• a ribbon graph that is equivalent to self but is generic in the sense that all vertices have valency 3

EXAMPLES:

```
sage: R = RibbonGraph(1,3); R
Ribbon graph of genus 1 and 3 boundary components
sage: R.sigma()
(1,2,3,9,7)(4,8,10,5,6)
sage: R.rho()
(1,4)(2,5)(3,6)(7,8)(9,10)
sage: G = R.make_generic(); G
Ribbon graph of genus 1 and 3 boundary components
sage: G.sigma()
(2,3,11)(5,6,13)(7,8,15)(9,16,17)(10,14,19)(12,18,21)(20,22)
sage: G.rho()
(2,5)(3,6)(7,8)(9,10)(11,12)(13,14)(15,16)(17,18)(19,20)(21,22)
sage: R.genus() == G.genus() and R.number_boundaries() == G.number_boundaries()
True
```

```
sage: R = RibbonGraph(5,4); R
Ribbon graph of genus 5 and 4 boundary components
sage: R.sigma()
(1,2,3,4,5,6,7,8,9,10,11,27,25,23)(12,24,26,28,13,14,15,16,17,18,19,20,21,22)
sage: R.rho()
sage: G = R.reduced(); G
Ribbon graph of genus 5 and 4 boundary components
sage: G.sigma()
(2,3,4,5,6,7,8,9,10,11,27,25,23,24,26,28,13,14,15,16,17,18,19,20,21,22)
sage: G.rho()
sage: G.genus() == R.genus() and G.number_boundaries() == R.number_boundaries()
True
```

\texttt{sage}: \text{R = RibbonGraph(0,6); R}\\
Ribbon graph of genus 0 and 6 boundary components\\
\texttt{sage}: \text{R.sigma()}\\
(1,11,9,7,5,3)(2,4,6,8,10,12)\\
\texttt{sage}: \text{R.rho()}\\
(1,2)(3,4)(5,6)(7,8)(9,10)(11,12)\\
\texttt{sage}: \text{G = R.reduced(); G}\\
Ribbon graph of genus 0 and 6 boundary components\\
\texttt{sage}: \text{G.sigma()}\\
(3,4,6,8,10,12,11,9,7,5)\\
\texttt{sage}: \text{G.rho()}\\
(3,4)(5,6)(7,8)(9,10)(11,12)\\
\texttt{sage}: \text{G.genus()} == \text{R.genus()} \text{ and G.number_boundaries()} == \text{R.number_boundaries()}\\
\text{True}

\texttt{mu()}\\
Return the rank of the first homology group of the thickening of the ribbon graph.

\textbf{EXAMPLES:}
\begin{verbatim}
\texttt{sage}: s1 = PermutationGroupElement('\( (1,3,5)(2,4,6) \)')
\texttt{sage}: r1 = PermutationGroupElement('\( (1,2)(3,4)(5,6) \)')
\texttt{sage}: R1 = RibbonGraph(s1,r1); R1
Ribbon graph of genus 1 and 1 boundary components
\texttt{sage}: R1.mu()
2
\end{verbatim}

\texttt{normalize()}\\
Return an equivalent graph such that the enumeration of its darts exhausts all numbers from 1 to the number of darts.

\textbf{OUTPUT:}
\begin{itemize}
  \item a ribbon graph equivalent to \texttt{self} such that the enumeration of its darts exhausts all numbers from 1 to the number of darts.
\end{itemize}

\textbf{EXAMPLES:}
\begin{verbatim}
\texttt{sage}: s0 = PermutationGroupElement('\( (1,22,3,4,5,6,7,15)(8,16,9,10,11,12,13,14) \)')
\texttt{sage}: r0 = PermutationGroupElement('\( (1,8)(22,9)(3,10)(4,11)(5,12)(6,13)(7,14)(15,16) \)')
\texttt{sage}: R0 = RibbonGraph(s0,r0); R0
Ribbon graph of genus 3 and 2 boundary components
\texttt{sage}: R0 = R0.normalize(); R0; R0.sigma(); R0.rho()
Ribbon graph of genus 3 and 2 boundary components\\
(1,16,2,3,4,5,6,14)(7,15,8,9,10,11,12,13)\\
(1,7)(2,9)(3,10)(4,11)(5,12)(6,13)(8,16)(14,15)
\texttt{sage}: s1 = PermutationGroupElement('\( (5,10,12)(30,34,78) \)')
\texttt{sage}: r1 = PermutationGroupElement('\( (5,30)(10,34)(12,78) \)')
\texttt{sage}: R1 = RibbonGraph(s1,r1); R1
Ribbon graph of genus 1 and 1 boundary components
\texttt{sage}: R1 = R1.normalize(); R1; R1.sigma(); R1.rho()
Ribbon graph of genus 1 and 1 boundary components
\end{verbatim}

(continues on next page)
number_boundaries()

Return number of boundary components of the thickening of the ribbon graph.

EXAMPLES:
The first example is the ribbon graph corresponding to the torus with one hole:

```
sage: s1 = PermutationGroupElement('(1,3,5)(2,4,6)')
sage: r1 = PermutationGroupElement('(1,2)(3,4)(5,6)')
sage: R1 = RibbonGraph(s1,r1)
sage: R1.number_boundaries()
1
```

This example is constructed by taking the bipartite graph of type (3, 3):

```
sage: s2 = PermutationGroupElement('(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,\rightarrow 15)(16,17,18)')
sage: r2 = PermutationGroupElement('(1,16)(2,13)(3,10)(4,17)(5,14)(6,11)(7,\rightarrow 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:

```
sage: s1 = PermutationGroupElement('(1,3,5)(2,4,6)')
sage: r1 = PermutationGroupElement('(1,2)(3,4)(5,6)')
sage: R1 = RibbonGraph(s1,r1); R1
Ribbon graph of genus 1 and 1 boundary components
sage: G1 = R1.reduced(); G1
Ribbon graph of genus 1 and 1 boundary components
sage: G1.sigma()
(3,5,4,6)
sage: G1.rho()
(3,4)(5,6)
```

```
sage: s2 = PermutationGroupElement('(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,\rightarrow 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)

`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: B23 = RibbonGraph(2,3,bipartite=True); B23; B23.sigma(); B23.rho()
Ribbon graph of genus 1 and 1 boundary components
(1,2,3)(4,5,6)(7,8)(9,10)(11,12)
(1,8)(2,10)(3,12)(4,7)(5,9)(6,11)

sage: B32 = RibbonGraph(3,2,bipartite=True); B32; B32.sigma(); B32.rho()
Ribbon graph of genus 1 and 1 boundary components
(1,2)(3,4)(5,6)(7,8,9)(10,11,12)
(1,9)(2,12)(3,8)(4,11)(5,7)(6,10)

sage: B33 = RibbonGraph(3,3,bipartite=True); B33; B33.sigma(); B33.rho()
Ribbon graph of genus 1 and 3 boundary components
(1,2,3)(4,5,6)(7,8,9)(10,11,12)(13,14,15)(16,17,18)
(1,12)(2,15)(3,18)(4,11)(5,14)(6,17)(7,10)(8,13)(9,16)

sage: B24 = RibbonGraph(2,4,bipartite=True); B24; B24.sigma(); B24.rho()
Ribbon graph of genus 1 and 2 boundary components
(1,2,3,4)(5,6,7,8)(9,10)(11,12)(13,14)(15,16)
(1,18)(2,12)(3,14)(4,16)(5,9)(6,11)(7,13)(8,15)

sage: B47 = RibbonGraph(4,7, bipartite=True); B47; B47.sigma(); B47.rho()
Ribbon graph of genus 9 and 1 boundary components
(1,2,3,4,5,6,7)(8,9,10,11,12,13,14)(15,16,17,18,19,20,21)(22,23,24,25,26,27,28)(29,
→30,31,32)(33,34,35,36)(37,38,39,40)(41,42,43,44)(45,46,47,48)(49,50,51,52)(53,54,
→55,56)
(1,32)(2,36)(3,40)(4,44)(5,48)(6,52)(7,56)(8,31)(9,35)(10,39)(11,43)(12,47)(13,
→37)(25,41)(26,45)(27,49)(28,53)

sage.geometry.ribbon_graph.make_ribbon(g, r)
Return a ribbon graph whose thickening has genus \( g \) and \( r \) boundary components.

INPUT:

- \( g \) – 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:

```python
sage: from sage.geometry.ribbon_graph import make_ribbon
sage: R = make_ribbon(0,1); R
Ribbon graph of genus 0 and 1 boundary components
sage: R.sigma() ()
```

(sage continued on next page)
4.6 Pseudolines

This module gathers everything that has to do with pseudolines, and for a start a `PseudolineArrangement` class that can be used to describe an arrangement of pseudolines in several different ways, and to translate one description into another, as well as to display Wiring diagrams via the `show` method.

In the following, we try to stick to the terminology given in [Fe1997], which can be checked in case of doubt. And please fix this module’s documentation afterwards :-)

**Definition**

A *pseudoline* can not be defined by itself, though it can be thought of as a $x$-monotone curve in the plane. A *set* of pseudolines, however, represents a set of such curves that pairwise intersect exactly once (and hence mimic the behaviour of straight lines in general position). We also assume that those pseudolines are in general position, that is that no three of them cross at the same point.

The present class is made to deal with a combinatorial encoding of a pseudolines arrangement, that is the ordering in which a pseudoline $l_i$ of an arrangement $l_0, \ldots, l_{n-1}$ crosses the $n-1$ other lines.

**Warning:** It is assumed through all the methods that the given lines are numbered according to their $y$-coordinate on the vertical line $x = -\infty$. For instance, it is not possible that the first transposition be $(0, 2)$ (or equivalently that the first line $l_0$ crosses is $l_2$ and conversely), because one of them would have to cross $l_1$ first.
4.6.1 Encodings

Permutations

An arrangement of pseudolines can be described by a sequence of \( n \) lists of length \( n - 1 \), where the \( i \) list is a permutation of \( \{0, \ldots, n-1\} \) 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

sage: p.show()
```

Sequence of transpositions

An arrangement of pseudolines can also be described as a sequence of \( \binom{n}{2} \) transpositions (permutations of two elements). In this sequence, the transposition \((2, 3)\) appears before \((8, 2)\) iif \( l_2 \) crosses \( l_3 \) before it crosses \( l_8 \). This encoding is easy to obtain by reading the wiring diagram from left to right (see the `show` method).

```sage
sage: from sage.geometry.pseudolines import PseudolineArrangement

sage: transpositions = [(3, 2), (3, 1), (0, 3), (2, 1), (0, 2), (0, 1)]

sage: p = PseudolineArrangement(transpositions)

sage: p
Arrangement of pseudolines of size 4

sage: p.show()
```

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 \( 0 \rightarrow 1 \) 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
```
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)]))
# not tested
sage: print(l[:5])
[(96, 278.0130613051349), (74, 332.92512282478714), (13, 155.65820951249867), (209, 34.75396221755307), (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: p.show(figsize=[20,8])
```

**Author**

Nathann Cohen

### 4.6.3 Methods

**class** `sage.geometry.pseudolines.PseudolineArrangement(seq, encoding='auto')`

Bases: object

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 $0..(n - 1)$. 

4.6. Pseudolines 641
felsner_matrix()

Return 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]]
```

```python
sage: p = PseudolineArrangement(permutations)
```

```python
sage: p.felsner_matrix()
[[0, 0, 0], [0, 0, 1], [0, 1, 1], [1, 1, 1]]
```

permutations()

Return the arrangements as $n$ permutations of size $n - 1$.

See the pseudolines module’s documentation for more information on this encoding.

EXAMPLES:

```python
sage: from sage.geometry.pseudolines import PseudolineArrangement
sage: permutations = [[3, 2, 1], [3, 2, 0], [3, 1, 0], [2, 1, 0]]
```

```python
sage: p = PseudolineArrangement(permutations)
```

```python
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]]
```

```python
sage: p = PseudolineArrangement(permutations)
```

```python
sage: p.show(figsize=[7, 5])
```

transpositions()

Return 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]]
```

```python
sage: p1 = PseudolineArrangement(permutations)
```

```python
sage: transpositions = [(3, 2), (3, 1), (0, 3), (2, 1), (0, 2), (0, 1)]
```

```python
sage: p2 = PseudolineArrangement(transpositions)
```

```python
sage: p1 == p2
```

True

```python
sage: p1.transpositions()
```

(continues on next page)
4.7 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 $\mathbb{R}^3$:

    ```python
    sage: V = VoronoiDiagram([[1, 3, .3], [2, -2, 1], [-1, 2, -.1]]); V
    The Voronoi diagram of 3 points of dimension 3 in the Real Double Field
    sage: VoronoiDiagram([])
    The empty Voronoi diagram.
    ```

    Get the Voronoi diagram of a regular pentagon in $\mathbb{A}^2$. All cells meet at the origin:

    ```python
    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:

    ```python
    sage: polytopes.dodecahedron().vertices_list()[0][0].parent()
    Number Field in sqrt5 with defining polynomial $x^2 - 5$ with sqrt5 = 2.
    → 23607977499790?
    sage: VoronoiDiagram(polytopes.dodecahedron().vertices_list())
    Traceback (most recent call last):
    ... NotImplementedError: Base ring of the Voronoi diagram must be one of QQ, RDF, AA.
    ```

    ALGORITHM:
We use hyperplanes tangent to the paraboloid one dimension higher to get a convex polyhedron and then project back to one dimension lower.

**Todo:**
- The dual construction: Delaunay triangulation
- improve 2d-plotting
- implement 3d-plotting
- more general constructions, like Voroi diagrams with weights (power diagrams)

**REFERENCES:**
- [Mat2002] Ch.5.7, p.118.

**AUTHORS:**
- Moritz Firsching (2012-09-21)

### ambient_dim()
Return the ambient dimension of the points.

**EXAMPLES:**

```sage
sage: V = VoronoiDiagram([[.5, 3], [2, 5], [4, 5], [4, -1]])
sage: V.ambient_dim()
2
sage: V = VoronoiDiagram([[1, 2, 3, 4, 5, 6]]); V.ambient_dim()
6
```

### base_ring()
Return the base_ring of the regions of the Voronoi diagram.

**EXAMPLES:**

```sage
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], [0.6493, 0.879]]
sage: V = VoronoiDiagram(P); S=V.plot()
sage: show(S, xmin=0, xmax=1, ymin=0, ymax=1, aspect_ratio=1, axes=false)
sage: S=V.plot(cell_colors={0:'red', 1:'blue', 2:'green', 3:'white', 4:'yellow'})
sage: show(S, xmin=0, xmax=1, ymin=0, ymax=1, aspect_ratio=1, axes=false)
sage: S=V.plot(cell_colors=['red','blue','white','white'])
sage: show(S, xmin=0, xmax=1, ymin=0, ymax=1, aspect_ratio=1, axes=false)
sage: S=V.plot(cell_colors='something else')
Traceback (most recent call last):
  ... AssertionError: 'cell_colors' must be a list or a dictionary

Trying to plot a Voronoi diagram of dimension other than 2 gives an error:

```
sage: VoronoiDiagram([[1, 2, 3], [6, 5, 4]]).plot()
Traceback (most recent call last):
  ... NotImplementedError: Plotting of 3-dimensional Voronoi diagrams not implemented
```

points()  
Return the input points (as a PointConfiguration).

EXAMPLES:

```
sage: V = VoronoiDiagram([[.5, 3], [2, 5], [4, 5], [4, -1]]); V.points()
A point configuration in affine 2-space over Real Field with 53 bits of precision consisting of 4 points. The triangulations of this point configuration are assumed to be connected, not necessarily fine, not necessarily regular.
```

regions()  
Return the Voronoi regions of the Voronoi diagram as a dictionary of polyhedra.

EXAMPLES:

```
sage: V = VoronoiDiagram([[1, 3, .3], [2, -2, 1], [-1, 2, -1]])
sage: P = V.points()
sage: V.regions() == {P[0]: Polyhedron(base_ring=RDF, lines=[(-RDF(0.375), RDF(0.13888888900000001), RDF(1.5277777799999999)), ....: rays=[(RDF(9), -RDF(1), -RDF(20)), (RDF(4.5), RDF(1), -RDF(25))], ....: vertices=[(-RDF(1.1074999999999999), RDF(1.149444444), RDF(9.01388888900000004)), ....: P[1]: Polyhedron(base_ring=RDF, lines=[(-RDF(0.375), ....: RDF(0.13888888900000001), RDF(1.52777777999999999)), ....: rays=[(RDF(9), -RDF(1), -RDF(20)), (-RDF(2.25), RDF(1), RDF(2.5))], ....: ....: RDF(20)), (-RDF(2.25), -RDF(1), RDF(2.5))], ....: (continues on next page)
```
P[2]: Polyhedron(base_ring=RDF, lines=[(-RDF(0.375),
    RDF(0.13888888900000001), RDF(1.5277777799999999))],
    rays=[(RDF(4.5), RDF(1), -RDF(25)), (-RDF(2.25), -RDF(1), RDF(2.5))],
    vertices=[(-RDF(1.0749999999999999), RDF(1.1494444444), RDF(9.0138888890000004))])}
True
5.1 Find isomorphisms between fans

**exception**  `sage.geometry.fan_isomorphism.FanNotIsomorphicError`

Bases: `Exception`

Exception to return if there is no fan isomorphism

`sage.geometry.fan_isomorphism.fan_2d_cyclically_ordered_rays(fan)`

Return the rays of a 2-dimensional fan in cyclic order.

**INPUT:**

- `fan` – a 2-dimensional fan.

**OUTPUT:**

A `PointCollection` containing the rays in one particular cyclic order.

**EXAMPLES:**

```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
```

```python
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)
frozenset({[ 1 0 -1]
          [ 0 1 -1]})
```

```python
sage: fan = toric_varieties.dP7().fan()
sage: sorted(fan_2d_echelon_forms(fan))
[[ 1 0 -1 -1 0]
  [ 0 1 1 -1 -1],
  [ 1 0 -1 0 1],
  [ 0 1 0 -1 -1],
  [ 0 1 1 0 -1],
  [ 0 1 1 0 -1],
  [ 0 1 1 0 -1],
  [ 0 1 1 0 -1]]
```

`sage.geometry.fan_isomorphism.fan_isomorphic_necessary_conditions(fan1, fan2)`
Check necessary (but not sufficient) conditions for the fans to be isomorphic.

INPUT:
- `fan1, fan2` – two fans.

OUTPUT:
Boolean. False if the two fans cannot be isomorphic. True if the two fans may be isomorphic.

EXAMPLES:

```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:

```python
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], [-1 -1], [0 1], [1 0], [1 0], [-1 -1], [0 1]
]
sage: m1 = matrix([[1, 0], [0, -5], [-3, 4]])
sage: m2 = matrix([[3, 0], [1, 0], [-2, 1]])
sage: m1.elementary_divisors() == m2.elementary_divisors() == [1,1,0]
True
sage: fan1 = Fan([Cone([m1*vector([23, 14]), m1*vector([3,100])]),
                Cone([m1*vector([-1,-14]), m1*vector([-100, -5])])])
sage: fan2 = Fan([Cone([m2*vector([23, 14]), m2*vector([3,100])]),
                Cone([m2*vector([-1,-14]), m2*vector([-100, -5])])])
sage: sorted(fan_isomorphism_generator(fan1, fan2))
[
[-12  1  -5],
[ -4  0  -1],
[- 5  0  -1]
]
sage: m0 = identity_matrix(ZZ, 2)
sage: m1 = matrix([[1, 0], [0, -5], [-3, 4]])
sage: m2 = matrix([[3, 0], [1, 0], [-2, 1]])
sage: m1.elementary_divisors() == m2.elementary_divisors() == [1,1,0]
True
sage: fan0 = Fan([Cone([m0*vector([1,0]), m0*vector([1,1])]),
                Cone([m0*vector([1,0]), m0*vector([0,1])])])
sage: fan1 = Fan([Cone([m1*vector([1,0]), m1*vector([1,1])]),
                Cone([m1*vector([1,1]), m1*vector([0,1])])])
sage: fan2 = Fan([Cone([m2*vector([1,0]), m2*vector([1,1])]),
                Cone([m2*vector([1,1]), m2*vector([0,1])])])
sage: sorted(fan_isomorphism_generator(fan0, fan0))
[
[ 0  1], [ 1  0]
[ 1  0], [ 0  1]
]
sage: sorted(fan_isomorphism_generator(fan1, fan1))

(continues on next page)

5.1. Find isomorphisms between fans
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: 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),(0,2,-1),(1,-1,1),(3,-1,-1)])
sage: fan1.is_isomorphic(fan2)
True

5.2 Construction of finite atomic and coatomic lattices from incidences

This module provides the function \texttt{lattice\_from\_incidences()} for computing finite atomic and coatomic lattices in the sense of partially ordered sets where any two elements have meet and joint. For example, the face lattice of a polyhedron.

\texttt{sage.geometry.hasse\_diagram.lattice\_from\_incidences(atom\_to\_coatoms, coatom\_to\_atoms, 
face\_constructor=None, required\_atoms=None, 
key=None, **kwds)}

Compute an atomic and coatomic lattice from the incidence between atoms and coatoms.

\textbf{INPUT:}

- \texttt{atom\_to\_coatoms} – list, \texttt{atom\_to\_coatom[i]} should list all coatoms over the \texttt{i}-th atom;
- \texttt{coatom\_to\_atoms} – list, \texttt{coatom\_to\_atom[i]} should list all atoms under the \texttt{i}-th coatom;
- \texttt{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;
- \texttt{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.
- \texttt{key} – any hashable value (default: None). It is passed down to \texttt{FinitePoset}.
- all other keyword arguments will be passed to \texttt{face\_constructor} on each call.

\textbf{OUTPUT:}

- \texttt{finite poset} with elements constructed by \texttt{face\_constructor}.

\textbf{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 \texttt{atom\_to\_coatoms} and to finish with coatoms in the order corresponding to \texttt{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.

\textbf{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

\[
\text{sage: } \text{atom_to_coatoms} = [(0,1), (0,2), (1,2)]
\]
\[
\text{sage: } \text{coatom_to_atoms} = [(0,1), (0,2), (1,2)]
\]

and we can compute the lattice as

\[
\text{sage: } \text{from sage.geometry.cone import lattice_from_incidences}
\]
\[
\text{sage: } \text{L} = \text{lattice_from_incidences}(
\text{....: } \text{atom_to_coatoms, coatom_to_atoms})
\]
\[
\text{sage: } \text{L}
\]
Finite lattice containing 8 elements with distinguished linear extension

\[
\text{sage: for level in L.level_sets(): print(level)}
\]
\[
[[()], ([0, 1, 2])]
[[([0,]), ([0, 1]), ([1,], [0, 2]), ([2,], [1, 2])]
[[([0, 1], [0,]), ([0, 2], [1,]), ([1, 2], [2,])]
[[([0, 1, 2], []])]
\]

For more involved examples see the source code of \text{sage.geometry.cone}.
\text{ConvexRationalPolyhedralCone.face_lattice()} \text{ and sage.geometry.fan. RationalPolyhedralFan._compute_cone_lattice()}. 

## 5.3 Cython helper methods to compute integral points in polyhedra.

\text{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:

\[
\text{sage: from sage.geometry.integral_points import InequalityCollection}
\]
\[
\text{sage: P_QQ = Polyhedron(identity_matrix(3).columns() + [(-2, -1, -1)], base_ring=QQ)}
\]
\[
\text{sage: ieq = InequalityCollection(P_QQ, [0,1,2], [0]*3,[1]*3); ieq}
\]
The collection of inequalities

\[
\text{integer: (3, -2, -2) x + 2 >= 0}
\]
integer: (-1, 4, -1) x + 1 >= 0
integer: (-1, -1, 4) x + 1 >= 0
integer: (-1, -1, -1) x + 1 >= 0

```python
sage: P_RR = Polyhedron(identity_matrix(2).columns() + [(-2.7, -1)], base_ring=RDF)
sage: InequalityCollection(P_RR, [0,1], [0]², [1]²)
```

The collection of inequalities

integer: (-1, -1) x + 1 >= 0
generic: (-1.0, 3.7) x + 1.0 >= 0
generic: (1.0, -1.35) x + 1.35 >= 0

```python
sage: line = Polyhedron(eqns=[(2,3,7)])
sage: InequalityCollection(line, [0,1], [0]², [1]²)
```

The collection of inequalities

integer: (3, 7) x + 2 >= 0
integer: (-3, -7) x + -2 >= 0

### 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:**

```python
sage: from sage.geometry.integral_points import InequalityCollection
sage: line = Polyhedron(eqns=[(2,3,7)])
sage: ieq = InequalityCollection(line, [0,1], [0]², [1]²)
sage: ieq.prepare_next_to_inner_loop([3,4])
sage: ieq.prepare_inner_loop([3,4])
sage: ieq.are_satisfied(3)
False
```

### prepare_inner_loop(p)

Peel off the inner loop.

In the inner loop of `rectangular_box_points()`, we have to repeatedly evaluate $Ax + b \geq 0$. To speed up computation, we pre-evaluate

$$c = Ax - A_0 x_0 + b = b + \sum_{i=1} A_i x_i$$

and only test $A_0 x_0 + c \geq 0$ in the inner loop.

You must call `prepare_next_to_inner_loop()` before calling this method.

**INPUT:**

- `p` – the coordinates of the point to loop over. Only the `p[1:]` entries are used.

**EXAMPLES:**
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 >= 0

ieq.prepare_next_to_inner_loop([2,1,3])
ieq.prepare_inner_loop([2,1,3])

print_cache(ieq)
Cached inner loop: 3 * x_0 + 42 >= 0
Cached next-to-inner loop: 3 * x_0 + 7 * x_1 + 35 >= 0

prepare_next_to_inner_loop(p)
Peel off the next-to-inner loop.

In the next-to-inner loop of rectangular_box_points(), we have to repeatedly evaluate $Ax - A_0 x_0 + b$. To speed up computation, we pre-evaluate

$$c = b + \sum_{i=2} A_i x_i$$

and only compute $Ax - A_0 x_0 + b = A_1 x_1 + c \geq 0$ in the next-to-inner loop.

INPUT:

- p – the point coordinates. Only p[2:] coordinates are potentially used by this method.

EXAMPLES:

[i]

satisfied_as_equalities(inner_loop_variable)
Return the inequalities (by their index) that are satisfied as equalities.

INPUT:

- inner_loop_variable – Integer. the 0-th coordinate of the lattice point.

OUTPUT:

A set of integers in ascending order. Each integer is the index of a H-representation object of the polyhedron (either a inequality or an equation).

EXAMPLES:

[i]
swap_ineq_to_front(i)

Swap the i-th entry of the list to the front of the list of inequalities.

INPUT:

• i – Integer. The Inequality_int to swap to the beginning of the list of integral inequalities.

EXAMPLES:

```python
sage: from sage.geometry.integral_points import InequalityCollection
sage: P_QQ = Polyhedron(identity_matrix(3).columns() + [(-2, -1, -1)], base_ring=QQ)
sage: iec = InequalityCollection(P_QQ, [0, 1, 2], [0]*3, [1]*3)
sage: iec
The collection of inequalities
integer: (3, -2, -2) x + 2 >= 0
integer: (-1, 4, -1) x + 1 >= 0
integer: (-1, -1, 4) x + 1 >= 0
integer: (-1, -1, -1) x + 1 >= 0
sage: iec.swap_ineq_to_front(3)
sage: iec
The collection of inequalities
integer: (-1, -1, -1) x + 1 >= 0
integer: (3, -2, -2) x + 2 >= 0
integer: (-1, 4, -1) x + 1 >= 0
integer: (-1, -1, 4) x + 1 >= 0
```

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
- max_abs_coordinates – the maximum of the coordinates that one wants to evaluate the coordinates on; used for overflow checking

OUTPUT:

Inequality $Ax + b \geq 0$. A OverflowError is raised if a machine integer is not long enough to hold the results. A ValueError is raised if some of the input is not integral.

EXAMPLES:

```python
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.
```python
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))
```

```python
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 `semigroup_generators()` for a description of the algorithm.

**INPUT:**

- `spanning_points` – a non-empty list of linearly independent rays (`Z`-vectors or toric lattice elements), not necessarily primitive lattice points.

**OUTPUT:**

The tuple of all lattice points in the half-open parallelotope spanned by the rays $r_i$,

$$par([r_i]) = \sum_{0 \leq a_i < 1} a_i r_i$$

By half-open parallelotope, we mean that the points in the facets not meeting the origin are omitted.

**EXAMPLES:**

Note how the points on the outward-facing 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), (0, 1), (1, 0), (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(0, 1), N(1, 0), N(1, 1))
```

A non-smooth cone:

```python
sage: c = Cone([(1, 0), (1, 2)])
sage: parallelotope_points(c.rays(), c.lattice())
(N(0, 0), N(1, 1))
```

A `ValueError` is raised if the `spanning_points` are not linearly independent:

5.3. Cython helper methods to compute integral points in polyhedra. 657
sage: rays = list(map(ToricLattice(2), [(1,1)]*2))
sage: parallelotope_points(rays, ToricLattice(2))
Traceback (most recent call last):
  ...ValueError: The spanning points are not linearly independent!

sage.geometry.integral_points.print_cache(inequality_collection)
Print the cached values in Inequality_int (for debugging/doctesting only).

EXAMPLES:

sage: from sage.geometry.integral_points import InequalityCollection, print_cache
sage: P = Polyhedron(ieqs=[[2,3,7]])
sage: ieq = InequalityCollection(P, [0,1], [0]*2,[1]*2); ieq
The collection of inequalities
integer: (3, 7) x + 2 >= 0
sage: ieq.prepare_next_to_inner_loop([3,5])
sage: ieq.prepare_inner_loop([3,5])
sage: print_cache(ieq)
Cached inner loop: 3 * x_0 + 37 >= 0
Cached next-to-inner loop: 3 * x_0 + 7 * x_1 + 2 >= 0

sage.geometry.integral_points.ray_matrix_normal_form(R)
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:

sage: from sage.geometry.integral_points import ray_matrix_normal_form
sage: R = column_matrix(ZZ,[[3,3,3]])
sage: ray_matrix_normal_form(R)
([3], 3, [1])

sage.geometry.integral_points.rectangular_box_points(box_min, box_max, polyhedron=None, count_only=False, return_saturated=False)
Return the integral points in the lattice bounding box that are also contained in the given polyhedron.

INPUT:

• box_min – 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 (`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):
  \[ Ax \leq b \iff a_1 x_1 \leq b - \sum_{i=2}^{d} a_i x_i \]
  so we only have to evaluate \( a_1 \times x_1 \) in the inner loop.
- Coordinates are permuted to make the longest box edge the inner loop. The inner loop is optimized to run very fast, so its best to do as much work as possible there.
- Continuously reorder inequalities and test the most restrictive inequalities first.
- Use convexity and only find first and last allowed point in the inner loop. The points in-between must be points of the polyhedron, too.

EXAMPLES:

```
sage: from sage.geometry.integral_points import rectangular_box_points
sage: rectangular_box_points([0,0,0],[1,2,3])
((0, 0, 0), (0, 0, 1), (0, 0, 2), (0, 0, 3),
 (0, 1, 0), (0, 1, 1), (0, 1, 2), (0, 1, 3),
 (0, 2, 0), (0, 2, 1), (0, 2, 2), (0, 2, 3),
 (1, 0, 0), (1, 0, 1), (1, 0, 2), (1, 0, 3),
 (1, 1, 0), (1, 1, 1), (1, 1, 2), (1, 1, 3),
 (1, 2, 0), (1, 2, 1), (1, 2, 2), (1, 2, 3))
```

```
sage: from sage.geometry.integral_points import rectangular_box_points
sage: rectangular_box_points([0,0,0],[1,2,3], count_only=True)
24
```

```
sage: cell24 = polytopes.twenty_four_cell()
sage: rectangular_box_points([-1]*4, [1]*4, cell24)
((-1, 0, 0, 0), (0, -1, 0, 0), (0, 0, -1, 0), (0, 0, 0, -1),
 (0, 0, 0, 0),
 (0, 0, 0, 1), (0, 0, 1, 0), (0, 1, 0, 0), (1, 0, 0, 0))
```

```
sage: d = 3
sage: dilated_cell24 = d*cell24
```

(continues on next page)
sage: len( rectangular_box_points([-d]^4, [d]^4, dilated_cell24) )
305

sage: d = 6
sage: dilated_cell24 = d*cell24
sage: len( rectangular_box_points([-d]^4, [d]^4, dilated_cell24) )
3625

sage: rectangular_box_points([-d]^4, [d]^4, dilated_cell24, count_only=True)
3625

sage: polytope = Polyhedron([(-4,-3,-2,-1),(3,1,1,1),(1,2,1,1),(1,1,3,0),(1,3,2,4)])

sage: pts = rectangular_box_points([-4]^4, [4]^4, polytope); pts
((-4, -3, -2, -1), (-1, 0, 0, 1), (0, 1, 1, 1), (1, 1, 1, 1), (1, 1, 3, 0),
 (1, 2, 1, 1), (1, 2, 2, 2), (1, 3, 2, 4), (2, 1, 1, 1), (3, 1, 1, 1))

sage: all(polytope.contains(p) for p in pts)
True

sage: set(map(tuple,pts)) == 
   ((-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:

sage: polytope = Polyhedron([[1], [10*pi.n()]], base_ring=RDF)

sage: len( rectangular_box_points([-100], [100], polytope) )
31

sage: halfplane = Polyhedron(ieqs=[(-1,1,0)])

sage: rectangular_box_points([0,-1+10^50], [0,1+10^50])
((0, 99999999999999999999999999999999999999999999999999),
 (0, 10000000000000000000000000000000000000000000000000),
 (0, 10000000000000000000000000000000000000000000000001))

sage: len( rectangular_box_points([-100+10^50], [1,100+10^50], halfplane) )
201

Using a PPL polyhedron:

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, 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 (-1, 0, 0) x + 1 >= 0
sage: cube.Hrepresentation(1)
An inequality (0, -1, 0) x + 1 >= 0
sage: cube.Hrepresentation(2)
An inequality (0, 0, -1) x + 1 >= 0
```

```python
sage: rectangular_box_points([0]*3, [1]*3, cube, return_saturated=True)
[((0, 0, 0), frozenset()),
 ((0, 0, 1), frozenset({2})),
 ((0, 1, 0), frozenset({1})),
 ((0, 1, 1), frozenset({1, 2})),
 ((1, 0, 0), frozenset({0})),
 ((1, 0, 1), frozenset({0, 2})),
 ((1, 1, 0), frozenset({0, 1})),
 ((1, 1, 1), frozenset({0, 1, 2}))]
```

```python
sage.geometry.integral_points.simplex_points(vertces)
Return the integral points in a lattice simplex.

INPUT:

• `vertices` – an iterable of integer coordinate vectors. The indices of vertices that span the simplex under consideration.

OUTPUT:

A tuple containing the integral point coordinates as \( \mathbf{Z} \)-vectors.

EXAMPLES:

```python
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:

```python
sage: simplex = Polyhedron([(1,2,3,5), (2,3,7,5), (-2,-3,-11,5)])
```

```python
sage: simplex_points(simplex.Vrepresentation())
((2, 3, 7, 5), (0, 0, -2, 5), (-2, -3, -11, 5), (1, 2, 3, 5))
```

```python
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:

```python
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]
 [0 1 0]
 [0 0 z]]
```

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:

```python
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, y, 1]])
sage: less_generators(m)
[ 1 0 0]
[ 0 z -1]
[ 0 y 1]
```
• Index
• Module Index
• Search Page
sage.geometry.cone, 268
sage.geometry.cone_catalog, 329
sage.geometry.convex_set, 601
sage.geometry.fan, 333
sage.geometry.fan_isomorphism, 647
sage.geometry.fan_morphism, 364
sage.geometry.hasse_diagram, 651
sage.geometry.hyperplane_arrangement.affine_subspace, 53
sage.geometry.hyperplane_arrangement.arrangement, 3
sage.geometry.hyperplane_arrangement.check_freeness, 661
sage.geometry.hyperplane_arrangement.hyperplane, 46
sage.geometry.hyperplane_arrangement.library, 40
sage.geometry.hyperplane_arrangement.plot, 56
sage.geometry.integral_points, 652
sage.geometry.lattice_polytope, 155
sage.geometry.linear_expression, 612
sage.geometry.newton_polygon, 618
sage.geometry.point_collection, 379
sage.geometry.polyhedral_complex, 228
sage.geometry.polyhedron.backend_cdd, 538
sage.geometry.polyhedron.backend_field, 539
sage.geometry.polyhedron.backend_normaliz, 540
sage.geometry.polyhedron.backend_ppl_make, 547
sage.geometry.polyhedron.backend_ppl, 550
sage.geometry.polyhedron.base, 411
sage.geometry.polyhedron.base_QQ, 526
sage.geometry.polyhedron.base_RDF, 538
sage.geometry.polyhedron.base_ZZ, 532
sage.geometry.polyhedron.cdd_file_format, 152
sage.geometry.polyhedron.constructor, 103
sage.geometry.polyhedron.double_description, 552
sage.geometry.polyhedron.double_description_inhomogeneous, 560
sage.geometry.polyhedron.face, 140
sage.geometry.polyhedron.lattice_euclidean_group_element, 207
sage.geometry.polyhedron.library, 61
sage.geometry.polyhedron.modules.formal_polyhedra_module, 153
sage.geometry.polyhedron.palp_database, 209
sage.geometry.polyhedron.parent, 111
sage.geometry.polyhedron.plot, 131
sage.geometry.polyhedron.ppl_lattice_polygon, 211
sage.geometry.polyhedron.ppl_lattice_polytope, 215
sage.geometry.polyhedron.representation, 117
sage.geometry.pseudolines, 639
sage.geometry.relative_interior, 622
sage.geometry.ribbon_graph, 626
sage.geometry.toric_lattice, 252
sage.geometry.toric_plotter, 387
sage.geometry.triangulation.base, 584
sage.geometry.triangulation.element, 593
sage.geometry.triangulation.point_configuration, 565
sage.geometry.voronoi_diagram, 643
sage.rings.polynomial.groebner_fan, 396
INDEX

A

A() (sage.geometry.linear_expression.LinearExpression method), 613
A() (sage.geometry.polyhedron.double_description.Problem method), 557
A() (sage.geometry.polyhedron.representation.Hrepresentation method), 118
A_matrix() (sage.geometry.polyhedron.double_description.Problem method), 557
a_maximal_chain() (sage.geometry.polyhedron.base.Polyhedron_base method), 414
add_cell() (sage.geometry.polyhedral_complex.PolyhedralComplex method), 231
add_hyperplane() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 8
add_inequality() (sage.geometry.polyhedron.double_description.StandardDoubleDescriptionPair method), 559
adjacency_graph() (sage.geometry.triangulation.element.Triangulation method), 593
adjacency_matrix() (sage.geometry.polyhedron.base.Polyhedron_base method), 414
adjacent() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 278
adjacent() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 158
adjacent() (sage.geometry.polyhedron.representation.Hrepresentation method), 118
adjacent() (sage.geometry.polyhedron.representation.Vrepresentation method), 128
adjust_options() (sage.geometry.toric_plotter.ToricPlotter method), 388
affine() (sage.geometry.triangulation.base.Point method), 585
affine_hull() (sage.geometry.convex_set.ConvexSet_base method), 601
affine_hull() (sage.geometry.polyhedron.base.Polyhedron_base method), 416
affine_hull_manifold() (sage.geometry.polyhedron.base.Polyhedron_base method), 417
affine_hull_projection() (sage.geometry.convex_set.ConvexSet_base method), 601
affine_hull_projection() (sage.geometry.polyhedron.base.Polyhedron_base method), 418
affine_lattice_polytope() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope method), 216
affine_space() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope method), 217
affine_tangent_cone() (sage.geometry.polyhedron.face.PolyhedronFace method), 142
alexander_whitney() (sage.geometry.polyhedral_complex.PolyhedralComplex method), 232
all_cached_data() (in module sage.geometry.lattice_polytope), 232
all_facet_equations() (in module sage.geometry.lattice_polytope), 199
all_nef_partitions() (in module sage.geometry.lattice_polytope), 200
all_points() (in module sage.geometry.lattice_polytope), 201
all_polars() (in module sage.geometry.lattice_polytope), 201
ambient() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 279
ambient() (sage.geometry.convex_set.ConvexSet_base method), 602
ambient() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 160
ambient() (sage.geometry.polyhedron.base.Polyhedron_base method), 425
ambient() (sage.geometry.polyhedron.face.PolyhedronFace method), 142
ambient() (sage.geometry.relative_interior.RelativeInterior method), 622
ambient() (sage.geometry.polyhedron.base.Polyhedron_base method), 426
ambient_dim() (sage.geometry.cone.IntegralRayCollection method), 318
ambient_dim() (sage.geometry.convex_set.ConvexSet_base method), 602
ambient_dim() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 160
ambient_dim() (sage.geometry.polyhedron_base.Polyhedron_base method), 144
ambient_dim() (sage.geometry.polyhedron.face.PolyhedronFace method), 145
ambient_dim() (sage.geometry.polyhedron.parent.Polyhedra_base method), 113
ambient_dim() (sage.geometry.relative_interior.RelativeInterior method), 622
ambient_dim() (sage.geometry.triangulation.base.PointConfiguration method), 589
ambient_dim() (sage.geometry.voronoi_diagram.VoronoiDiagram method), 644
ambient_dim() (sage.rings.polynomial.groebner_fan.PolyhedralCone method), 404
ambient_dim() (sage.rings.polynomial.groebner_fan.PolyhedralFan method), 405
ambient_dimension() (sage.geometry.convex_set.ConvexSet_base method), 602
ambient_dimension() (sage.geometry.polyhedron_base.Polyhedron_base method), 145
ambient_dimension() (sage.geometry.polyhedron.face.PolyhedronFace method), 144
ambient_dimension() (sage.geometry.relative_interior.RelativeInterior method), 622
ambient_face_indices() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 160
ambient_H_indices() (sage.geometry.polyhedron.face.PolyhedronFace method), 143
ambient_Hrepresentation() (sage.geometry.polyhedron.face.PolyhedronFace method), 143
ambient_module() (sage.geometry.linear_expression.LinearExpressionModule method), 615
ambient_module() (sage.geometry.toric_lattice.ToricLattice_ambient method), 256
ambient_ordered_point_indices() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 161
ambient_point_indices() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 161
ambient_ray_indices() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 279
ambient_space() (sage.geometry.polyhedron_base.Polyhedron_base method), 426
ambient_space() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope_PPL_class method), 217
ambient_space() (sage.geometry.polyhedron.face.PolyhedronFace method), 145
ambient_space() (sage.geometry.polyhedron.parent.Polyhedra_base method), 113
ambient_space() (sage.geometry.polyhedron_base.Polyhedron_base method), 426
ambient_space() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope_PPL_class method), 217
ambient_V_indices() (sage.geometry.polyhedron_face.PolyhedronFace method), 144
ambient_vector_space() (sage.geometry.linear_expression.LinearExpressionModule method), 616
ambient_vector_space() (sage.geometry.polyhedron_base.Polyhedron_base method), 426
ambient_vector_space() (sage.geometry.polyhedron_face.PolyhedronFace method), 145
ambient_vector_space() (sage.geometry.polyhedron.parent.Polyhedra_base method), 114
ambient_vertex_indices() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 162
ambient_Vrepresentation() (sage.geometry.polyhedron.face.PolyhedronFace method), 144
AmbientVectorSpace (class in sage.geometry.hyperplane_arrangement.hyperplane), 47
an_affine_basis() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 279
an_affine_basis() (sage.geometry.convex_set.ConvexSet_base method), 603
an_affine_basis() (sage.geometry.convex_set.ConvexSet_base method), 603
an_affine_basis() (sage.geometry.polyhedron_base.Polyhedron_base method), 426
an_affine_basis() (sage.geometry.polyhedron_base.Polyhedron_base method), 426
an_affine_basis() (sage.geometry.polyhedron_parent.Polyhedra_base method), 114
an_element() (sage.geometry.polyhedron.double_description.DoubleDescriptionPair method), 668
are_adjacent() (sage.geometry.polyhedron.double_description.DoubleDescriptionPair method), 668

Index
Index  669

are_satisfied() (sage.geometry.integral_points.InequalityCollection method), 429
Birkhoff_polytope()  62
associahedron() (sage.geometry.polyhedron.library.Polytopes method), 145
as_polyhedron() (sage.geometry.polyhedron.face.PolyhedronFace method), 65
base_extend() (sage.geometry.polyhedron.base.Polyhedron_base method), 569
barycentric_subdivision() (sage.geometry.polyhedron.base.Polyhedron_base method), 427
base_projection() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope_PPL_class method), 217
base_projection_matrix() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope_PPL_class method), 218
base_ring() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope_PPL_class method), 38
base_ring() (sage.geometry.polyhedron.base.Polyhedron_base method), 429
base_ring() (sage.geometry.polyhedron.double_description.Problem method), 557
base_ring() (sage.geometry.triangulation.base.PointConfiguration method), 589
base_ring() (sage.geometry.voronoi_diagram.VoronoiDiagram method), 644
basis() (sage.geometry.linear_expression.LinearExpression method), 616
basis() (sage.geometry.point_collection.PointCollection method), 380
bigraphical() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangements method), 43
bipartite_ribbon_graph() (in module sage.geometry.ribbon_graph), 637
bipyramid() (sage.geometry.polyhedron.base.Polyhedron_base method), 549
bitruncated_six_hundred_cell() (sage.geometry.polyhedron.library.Polytopes method), 64
boundary() (sage.geometry.ribon_graph.RibbonGraph method), 629
boundary() (sage.geometry.triangulation.element.Triangulation method), 9
boundary_complex() (sage.geometry.polyhedron.base.Polyhedron_base method), 429
boundary_point_indices() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 162
boundary_points() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 163
boundary_subcomplex() (sage.geometry.polyhedral_complex.PolyhedralComplex method), 233
bounded_edges() (sage.geometry.polyhedron.base.Polyhedron_base method), 430
bounded_regions() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangements method), 9
bounding_box() (sage.geometry.polyhedron.base.Polyhedron_base method), 430
braid() (sage.geometry.hyperplane_arrangement.library.HyperplaneArrangementLibrary method), 119
buchsberger() (sage.rings.polynomial.groebner_fan.GroebnerFan method), 397
buckyball() (sage.geometry.polyhedron.library.Polytopes method), 65
C
cantellated_one_hundred_twenty_cell() (sage.geometry.polyhedron.library.Polytopes method), 65
cantitruncated_one_hundred_twenty_cell() (sage.geometry.polyhedron.library.Polytopes method), 66
cantitruncated_six_hundred_cell() (sage.geometry.polyhedron.library.Polytopes method), 67
cantitruncated_six_hundred_cell() (sage.geometry.polyhedron.library.Polytopes method), 67
cantitruncated_six_hundred_cell() (sage.geometry.polyhedron.library.Polytopes method), 67
cantitruncated_six_hundred_cell() (sage.geometry.polyhedron.library.Polytopes method), 67
cantitruncated_six_hundred_cell() (sage.geometry.polyhedron.library.Polytopes method), 67
cantitruncated_six_hundred_cell() (sage.geometry.polyhedron.library.Polytopes method), 67
cantitruncated_six_hundred_cell() (sage.geometry.polyhedron.library.Polytopes method), 67
buckyball() (sage.geometry.polyhedron.library.Polytopes method), 65
cardinality() (sage.geometry.convex_set.ConvexSet_base method), 603
catalan() (sage.geometry.polyhedron.base.Polyhedron_base method), 40

cartesian_product() (sage.geometry.point_collection.PointCollection method), 381

cartesian_product() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 280

cartesian_product() (sage.geometry.cone.IntegralRayCollection method), 318

cartesian_product() (sage.geometry.convex_set.ConvexSet_base method), 604

cartesian_product() (sage.geometry.fan.RationalPolyhedralFan method), 345

cartesian_product() (sage.geometry.point_collection.PointCollection method), 381

cartesian_product() (sage.geometry.polyhedron.base.Polyhedron_base method), 431

catalan() (sage.geometry.hyperplane_arrangement.library.HyperplaneArrangementLibrary method), 40

cdd_Hrepresentation() (in module sage.geometry.polyhedron.cdd_file_format), 152

cdd_Hrepresentation() (sage.geometry.polyhedron.base.Polyhedron_base method), 431

cdd_Vrepresentation() (in module sage.geometry.polyhedron.cdd_file_format), 152

cdd_Vrepresentation() (sage.geometry.polyhedron.base.Polyhedron_base method), 431

cell_iterator() (sage.geometry.polyhedral_complex.PolyhedralComplex method), 234

cells() (sage.geometry.polyhedral_complex.PolyhedralComplex method), 234

cells_list_to_cells_dict() (in module sage.geometry.polyhedral_complex), 252

cells_sorted() (sage.geometry.polyhedral_complex.PolyhedralComplex method), 235

center() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 10

center() (sage.geometry.polyhedron.base.Polyhedron_base method), 432

centroid() (sage.geometry.polyhedron.base.Polyhedron_base method), 433

chain_complex() (sage.geometry.polyhedral_complex.PolyhedralComplex method), 235

change_ring() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangements method), 39

classify_cone_2d() (in module sage.geometry.cone), 11

closed_faces() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementLibrary method), 11

closure() (sage.geometry.convex_set.ConvexSet_base method), 604

closure() (sage.geometry.relative_interior.RelativeInterior method), 623

codim() (sage.geometry.cone.IntegralRayCollection method), 319

codim() (sage.geometry.convex_set.ConvexSet_base method), 604

codimension() (sage.geometry.cone.IntegralRayCollection method), 320

codimension() (sage.geometry.convex_set.ConvexSet_base method), 604

codomain_dim() (sage.geometry.polyhedron.lattice_euclidean_group_element.LatticeEuclideanGroupElement method), 208

codomain_fan() (sage.geometry.fan_morphism.FanMorphism method), 368

coefficients() (sage.geometry.linear_expression.LinearExpression method), 614

color_list() (in module sage.geometry.toric_plotter), 151

combinatorial_automorphism_group() (sage.geometry.polyhedron.base.Polyhedron_base method), 435

combinatorial_face_to_polyhedral_face() (in module sage.geometry.polyhedron.face), 151

change_ring() (sage.geometry.polyhedron.parent.Polyhedron_base method), 114

common_refinement() (in module sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangements)
complex() (sage.geometry.fan.RationalPolyhedralFan method), 345

contains() (sage.geometry.polyhedron.representation.Inequality method), 121

contains_origin() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope method), 219

contract_edge() (sage.geometry.ribbon_graph.RibbonGraph method), 219

cone() (in module sage.geometry.cone), 271

cone() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 15

cone_lattice() (sage.geometry.polyhedron.double_description.DoubleDescriptionPair method), 554

convex_hull() (in module sage.geometry.polyhedron.lattice_polytope), 201

convex_hull() (sage.geometry.polyhedron.base.Polyhedron_base method), 437

convex_hull() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 571

ConvexRationalPolyhedralCone (class in sage.geometry.cone), 274

ConvexSet_base (class in sage.geometry.convex_set), 601

ConvexSet_closed (class in sage.geometry.convex_set), 609

ConvexSet_compact (class in sage.geometry.convex_set), 609

ConvexSet_open (class in sage.geometry.convex_set), 610

ConvexSet_relatively_open (class in sage.geometry.convex_set), 611

coord_indices_of() (sage.geometry.polyhedron.plot.Projection method), 131

cross_positive_operators_gens() (sage.geometry.toric_lattice.ToricLattice_quotient method), 281

cross_polytope() (in module sage.geometry.lattice_polytope), 202

cross_polytope() (sage.geometry.polyhedron.library.Polytopes method), 67

cross_positive_operators_gens() (sage.geometry.polyhedron.library.Polytopes method), 68

cyclic_polytope() (sage.geometry.polyhedron.library.Polytopes method), 69

Index
cyclic_sort_vertices_2d() (in module sage.geometry.polyhedron.plot), 139

D
defining_polynomial()
(dag.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 15
degree_on_basis() (sage.geometry.polyhedron.modules.formal_polyhedra_module.FormalPolyhedraModule method), 154
deletion() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 16
Delta() (sage.geometry.lattice_polytope.NefPartition method), 191
Delta_polar() (sage.geometry.lattice_polytope.NefPartition method), 192
Deltas() (sage.geometry.lattice_polytope.NefPartition method), 192
dense_coefficient_list()
(sage.geometry.linear_expression.LinearExpression method), 614
derivation_module_basis()
(sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 16
derivation_module_free_chain()
(sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 17
dilation()
(sage.geometry.convex_set.ConvexSet_base method), 605
dilation()
(sage.geometry.polyhedron.base.Polyhedron_base method), 438
dilation()
(sage.geometry.polyhedron.base.Polyhedron_base method), 438
dimension()
(sage.geometry.convex_set.ConvexSet_base method), 605
dimension()
(sage.geometry.hyperplane_arrangement.affine_subspace.AffineSubspace method), 54
dimension()
(sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 17
dimension()
(sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 17
dimension()
(sage.geometry.hyperplane_arrangement.hyperplane.HyperplaneArrangementElement method), 18
dimension()
(sage.geometry.point_collection.PointCollection method), 164
dimension()
(sage.geometry.point_collection.PointCollection method), 164
direct_sum()
(sage.geometry.toric_lattice.ToricLattice_quotient method), 258
direct_sum()
(sage.geometry.toric_lattice.ToricLattice_quotient method), 258
discard_faces()
(in module sage.geometry.fan), 363
discrete_complementarity_set()
(sage.geometry.polyhedron.polyhedron.Polyhedron method), 237
disjoint_union()
(sage.geometry.polyhedron.polyhedron.Polyhedron method), 237
distance()
(sage.geometry.triangulation.point_configuration.PointConfiguration method), 571
distance_affine()
(sage.geometry.triangulation.point_configuration.PointConfiguration method), 572
distance_between_regions()
(sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 18
distance_enumerator()
(sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 18
distance_FS()
(sage.geometry.triangulation.point_configuration.PointConfiguration method), 572
distances()
(sage.geometry.lattice_polytope.LatticePolytopeClass method), 164
dodecahedron()
(sage.geometry.polyhedron.library.PolyhedronClass method), 69
does_backend_handle_base_ring()
(in module sage.geometry.polyhedron.parent), 116
domain_dim()
(sage.geometry.polyhedron.lattice.euclidean_group.Element method), 208
domain_dim()
(sage.geometry.polyhedron.lattice.euclidean_group.Element method), 208
domain_fan()
(sage.geometry.fan.morphism.FanMorphism method), 368
DoubleDescriptionPair (class in
Hrepresentation (class in sage.geometry.polyhedron.representation), 118
Hrepresentation() (sage.geometry.polyhedron.backend_ppl.Polyhedron_ppl method), 551
Hrepresentation() (sage.geometry.polyhedron.base.Polyhedron_base method), 411
Hrepresentation_space() (sage.geometry.polyhedron.base.Polyhedron_base method), 411
Hrepresentation_space() (sage.geometry.polyhedron.parent.Polyhedra_base method), 113
Hrepresentation_str() (sage.geometry.polyhedron.base.Polyhedron_base method), 412
hypercube() (sage.geometry.polyhedron.library.Polytopes method), 77
Hyperplane (class in sage.geometry.hyperplane_arrangement.hyperplane), 48
hyperplane_arrangement() (sage.geometry.polyhedron.base.Polyhedron_base method), 459
HyperplaneArrangementElement (class in sage.geometry.hyperplane_arrangement.arrangement), 8
hyperplane_arrangement_library (sage.geometry.hyperplane_arrangement.arrangement.geometry.point_collection.PointCollection method), 382
HyperplaneArrangements (class in sage.geometry.hyperplane_arrangement.arrangement), 38
hyperplanes() (sage.geometry.hyperplane_arrangement.arrangement.geometry.polyhedron_base method), 22
hypersimplex() (sage.geometry.polyhedron.library.Polytopes method), 78

icosahedron() (sage.geometry.polyhedron.library.Polytopes method), 79
icosidodecahedron() (sage.geometry.polyhedron.library.Polytopes method), 79
icosidodecahedron_V2() (sage.geometry.polyhedron.library.Polytopes method), 80
ideal() (sage.rings.polynomial.groebner_fan.GroebnerFan method), 398
ideal() (sage.rings.polynomial.groebner_fan.ReducedGroebnerBasis method), 408
ideal_to_gfan_format() (in module sage.rings.polynomial.groebner_fan), 409
identity() (sage.geometry.polyhedron.plot.Projection method), 131
image_cone() (sage.geometry.fan_morphism.FanMorphism method), 370
incidence_matrix() (sage.geometry.convex.RationalPolyhedralCones method), 291
incidence_matrix() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 459
incident() (sage.geometry.polyhedron.representation.Hrepresentation method), 119
incident() (sage.geometry.polyhedron.representation.Vrepresentation method), 128
include_points() (sage.geometry.toric_plotter.ToricPlotter method), 389
index() (sage.geometry.polyhedron.representation.PolyhedronRepresentation method), 124
index() (sage.geometry.triangulation.base.Point method), 586
inequalities() (sage.geometry.polyhedron.base.Polyhedron_base method), 461
inequalities_list() (sage.geometry.polyhedron.base.Polyhedron_base method), 461
initial_form_systems() (sage.rings.polynomial.groebner_fan.TropicalPrevariety method), 409
initial_forms() (sage.rings.polynomial.groebner_fan.InitialForm method), 403
initial_pair() (sage.geometry.polyhedron.double_description.Problem method), 557
Inequality (class in sage.geometry.polyhedron.representation), 121
Inequality_generator() (sage.geometry.polyhedron.representation.Hrepresentation method), 462
Inequality_generic (class in sage.geometry.integral_points), 655
Inequality_int (class in sage.geometry.integral_points), 655
InequalityCollection (class in sage.geometry.integral_points), 652
initial_form_systems() (sage.rings.polynomial.groebner_fan.TropicalPrevariety method), 409
initial_forms() (sage.rings.polynomial.groebner_fan.InitialForm method), 403
initial_pair() (sage.geometry.polyhedron.double_description.Problem method), 557
InitialForm (class in sage.geometry.polyhedron.double_description.Problem method), 402
inner_product_matrix() (sage.geometry.polyhedron.double_description.DoubleDescription method), 555
int_to_simplex() (sage.geometry.triangulation.base.PointCollection method), 589
is_mutable() (sage.geometry.polyhedron.base.Polyhedron_base method), 473
is_NefPartition() (in module sage.geometry.lattice_polytope), 202
is_neighboring() (sage.geometry.polyhedron.base.Polyhedron_base method), 27
is_neighborly() (sage.geometry.polyhedron.base.Polyhedron_base method), 473
is_neighborly() (sage.geometry.polyhedron.base.Polyhedron_base method), 473
is_open() (sage.geometry.convex_set.ConvexSet_base method), 607
is_open() (sage.geometry.convex_set.ConvexSet_closed method), 609
is_open() (sage.geometry.convex_set.ConvexSet_open method), 611
is_open() (sage.geometry.convex_set.ConvexSet_relatively_open method), 611
is_PointCollection() (in module sage.geometry.point_collection), 385
is_polyhedral_fan() (sage.geometry.polyhedral_complex.PolyhedralComplex method), 243
is_polyhedron() (in module sage.geometry.polyhedron.base), 525
is_prism() (sage.geometry.polyhedron.base.Polyhedron_base method), 474
is_proper() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 297
is_pure() (sage.geometry.polyhedral_complex.PolyhedralComplex method), 243
is_pyramid() (sage.geometry.polyhedron.base.Polyhedron_base method), 475
is_ray() (sage.geometry.polyhedron.representation.Ray method), 126
is_ray() (sage.geometry.polyhedron.representation.Vrepresentation method), 129
is_reflexive() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 298
is_reflexive() (sage.geometry.polyhedron.base.ZZ.Polyhedron_ZZ sagesage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope method), 223
is_reflexive() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope method), 223
is_relative() (sage.geometry.polyhedron.base.Polyhedron_base method), 476
is_relative() (sage.geometry.polyhedron.face.PolyhedronFace method), 476
is_relative() (sage.geometry.polyhedron.face.PolyhedronFace method)
sage.geometry.hyperplane_arrangement.library, 46
sage.geometry.hyperplane_arrangement.plot, 40
sage.geometry.integral_points, 652
sage.geometry.lattice_polytope, 155
sage.geometry.linear_expression, 612
sage.geometry.newton_polygon, 618
sage.geometry.point_collection, 379
sage.geometry.polyhedral_complex, 228
sage.geometry.polyhedron.backend_cdd, 538
sage.geometry.polyhedron.backend_field, 540
sage.geometry.polyhedron.backend_normaliz, 547
sage.geometry.polyhedron.backend_ppl, 550
sage.geometry.polyhedron.base, 411
sage.geometry.polyhedron.base_QQ, 526
sage.geometry.polyhedron.base_RDF, 538
sage.geometry.polyhedron.base_ZZ, 532
sage.geometry.polyhedron.cdd_file_format, 152
sage.geometry.polyhedron.constructor, 103
sage.geometry.polyhedron.double_description, 552
sage.geometry.polyhedron.double_description_inhomogeneous, 560
sage.geometry.polyhedron.face, 140
sage.geometry.polyhedron.lattice_euclidean_group_element, 207
sage.geometry.polyhedron.library, 61
sage.geometry.polyhedron.modules.formal_polyhedra_module, 153
sage.geometry.polyhedron.palp_database, 209
sage.geometry.polyhedron.parent, 111
sage.geometry.polyhedron.plot, 131
sage.geometry.polyhedron.ppl_lattice_polytope, 211
sage.geometry.polyhedron.ppl_lattice_polytope, 215
sage.geometry.polyhedron.representation, 117
sage.geometry.pseudolines, 639
sage.geometry.relative_interior, 622
sage.geometry.ribbon_graph, 626
sage.geometry.toric_lattice, 252
sage.geometry.toric_plotter, 387
sage.geometry.triangulation.base, 584
sage.geometry.triangulation.element, 593
sage.geometry.triangulation.point_configuration, 565
sage.geometry.voronoi_diagram, 643
sage.rings.polynomial.groebner_fan, 396
module() (sage.geometry.point_collection.PointCollection method), 383
monomial_coefficients() (sage.geometry.linear_expression.LinearExpression method), 615
mu() (sage.geometry.ribbon_graph.RibbonGraph method), 652
n_ambient_Hrepresentation() (sage.geometry.polyhedron.face.PolyhedronFace method), 147
n_ambient_Vrepresentation() (sage.geometry.polyhedron.face.PolyhedronFace method), 147
n_boundaries() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 28
n_equations() (sage.geometry.polyhedron.base.Polyhedron_base method), 488
n_facets() (sage.geometry.polyhedron.base.Polyhedron_base method), 488
n_Hrepresentation() (sage.geometry.polyhedron.base.Polyhedron_base method), 488
n_hyperplanes() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangement method), 28
n_integral_hull() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope method), 224
n_integral_points() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope method), 224
n_lines() (sage.geometry.polyhedron.base.Polyhedron_base method), 489
n_lines() (sage.geometry.polyhedron.face.PolyhedronFace method), 148
n_maximal_cells() (sage.geometry.polyhedral_complex.PolyhedralComplex method), 246
n_inequalities() (sage.geometry.triangulation.base.PointConfiguration_base method), 590
n_rays() (sage.geometry.polyhedron.base.Polyhedron_base method), 489
n_rays() (sage.geometry.polyhedron.face.PolyhedronFace method), 148
n_regions() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangement method), 29
n_skeleton() (sage.geometry.polyhedral_complex.PolyhedralComplex method), 247
n_vertices() (sage.geometry.polyhedron.base.Polyhedron_base method), 489
n_vertices() (sage.geometry.polyhedron.face.PolyhedronFace method), 148
n_vertices() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytopePPL_class method), 225
n_Vrepresentation() (sage.geometry.polyhedron.base.Polyhedron_base method), 488
nabla() (sage.geometry.lattice_polytope.NefPartition method), 194
nabla_polar() (sage.geometry.lattice_polytope.NefPartition method), 195
nablas() (sage.geometry.lattice_polytope.NefPartition method), 195
nef_partitions() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 176
nef_x() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 178
NefPartition (class in sage.geometry.lattice_polytope), 189
neighborliness() (sage.geometry.polyhedron.base.Polyhedron_base method), 490
neighbors() (sage.geometry.polyhedron.representation.Hrepresentation method), 120
neighbors() (sage.geometry.polyhedron.representation.Vrepresentation method), 130
NewtonPolygon (in module sage.geometry.newton_polygon), 618
NewtonPolygon_element (class in sage.geometry.newton_polygon), 618
nfacets() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 178
ngenerating_cones() (sage.geometry.fan.RationalPolyhedralFan method), 359
ngens() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangement method), 39
ngens() (sage.geometry.linear_expression.LinearExpressionModule method), 617
nonnegative_orthant() (in module sage.geometry.cone_catalog), 329
normal() (sage.geometry.hyperplane_arrangement.hyperplane.Hyperplane method), 50
normal_cone() (sage.geometry.polyhedron.face.Polyhedron_face method), 148
normal_cone() (sage.geometry.triangulation.element.Triangulation method), 596
normal_fan() (sage.geometry.polyhedron.base.Polyhedron_base method), 490
normal_form() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 179
NormalFan() (in module sage.geometry.fan), 343
normalize() (sage.geometry.ribbon_graph.RibbonGraph method), 635
normalize_rays() (in module sage.geometry.cone), 305
nparts() (sage.geometry.lattice_polytope.NefPartition method), 196
npoints() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 180
nreyns() (sage.geometry.cone.IntegralRayCollection method), 322
number_boundaries() (sage.geometry.ribbon_graph.RibbonGraph method), 636
number_of_reduced_groebner_bases() (sage.rings.polynomial.groebner_fan.GroebnerFan method), 399
number_of_variables() (sage.rings.polynomial.groebner_fan.GroebnerFan method), 399
nvertices() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 181
octahedron() (sage.geometry.polyhedron.library.Polytopes method), 80
omnitruncated_one_hundred_twenty_cell() (sage.geometry.polyhedron.library.Polytopes method), 81
omnitruncated_six_hundred_cell() (sage.geometry.polyhedron.library.Polytopes method), 81
one_hundred_twenty_cell() (sage.geometry.polyhedron.library.Polytopes method), 82
one_point_suspension() (sage.geometry.polyhedron.base.Polyhedron_base method), 492
ordered_vertices() (sage.geometry.polyhedron.ppl_lattice_polygon.LatticePolygon method), 212
oriented_boundary() (sage.geometry.fan.RationalPolyhedralFan method), 359
orlik_solomon_algebra() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangement method), 29
orlik_terao_algebra() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangement method), 29
orthogonal_projection() (sage.geometry.hyperplane_arrangement.hyperplane.Hyperplane method), 50
orthogonal_sublattice() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 305

Index 683
outer_normal() (sage.geometry.polyhedron.representation method), 122
output_format() (sage.geometry.point_collection.PointCollection static method), 383

pair_class (sage.geometry.polyhedron.double_description.Polyhedron) attribute), 558
pair_class (sage.geometry.polyhedron.double_description.Polyhedron) attribute), 558

PALPreader (class in sage.geometry.polyhedron.palp_database) 209
parallelootope() (sage.geometry.polyhedron.library.Polytope method), 82
parallelootope_points() (in module sage.geometry.integral_points), 657
parent() (sage.geometry.lattice_polytope.LatticePolytope class) method), 181
ParentNewtonPolygon (class in sage.geometry.newton_polygon), 620
part() (sage.geometry.lattice_polytope.NefPartition method), 196
part_of() (sage.geometry.lattice_polytope.NefPartition method), 196
part_of_point() (sage.geometry.lattice_polytope.NefPartition method), 197
parts() (sage.geometry.lattice_polytope.NefPartition method), 198
pentakis_dodecahedron() (sage.geometry.polyhedron.library.Polytopes method), 83
permutahedron() (sage.geometry.polyhedron.library.Polytopes method), 83
permutations() (sage.geometry.pseudolines.PseudolineArrangement method), 642
PivotedInequalities (class in sage.geometry.polyhedron.double_description_inhomogeneous), 561
placing_triangulation() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 574
plot() (in module sage.geometry.hyperplane_arrangement.plot), 58
plot() (sage.geometry.cone.ConvexRationalPolyhedralComplex method), 305
plot() (sage.geometry.cone.IntegralRayCollection method), 322
plot() (sage.geometry.fan.RationalPolyhedralFan method), 360
plot() (sage.geometry.newton_polygon.NewtonPolygon_element method), 618
plot() (sage.geometry.polyhedron.base.Polyhedron_base method), 492
plot() (sage.geometry.polyhedron.ppl_lattice_polygon.LatticePolygon_PPL method), 212
plot() (sage.geometry.toric_lattice.ToricLattice_ambient attribute), 257
plot() (sage.geometry.toric_lattice.ToricLattice_sublattice_with_basis method), 267
plot() (sage.geometry.triangulation.element.Triangulation method), 597
plot() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 575
plot3d() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 181
plot() (sage.geometry.voronoi_diagram.VoronoiDiagram method), 644
plot_hyperplane() (in module sage.geometry.hyperplane_arrangement.plot), 58
plot_labels() (sage.geometry.toric_plotter.ToricPlotter method), 389
plot_lattice() (sage.geometry.toric_plotter.ToricPlotter method), 390
plot_points() (sage.geometry.toric_plotter.ToricPlotter method), 390
plot_ray_labels() (sage.geometry.toric_plotter.ToricPlotter method), 390
plot_walls() (sage.geometry.toric_plotter.ToricPlotter method), 391
poincare_polynomial() (sage.geometry.hyperplane_arrangement.arrangement.Hyperplane polynomial) (sage.geometry.hyperplane_arrangement.arrangement.Hyperplane polynomial) (sage.geometry.hyperplane_arrangement.arrangement.Hyperplane polynomial) method), 30
Point (class in sage.geometry.triangulation.base), 585
plot() (in module sage.geometry.polyhedron.ppl_lattice_polytope), 228
point() (sage.geometry.hyperplane_arrangement.affine_subspace.AffineSubspace method), 55
point() (sage.geometry.hyperplane_arrangement.hyperplane.Hyperplane method), 51
point() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 183
plot() (sage.geometry.triangulation.base.PointConfiguration_base method), 590
point_configuration() (sage.geometry.triangulation.base.Point method), 586
Polyhedron_RDF (class in sage.geometry.polyhedron.base_RDF), 538
Polyhedron_RDF_cdd (class in sage.geometry.polyhedron.backend_cdd), 538
Polyhedron_ZZ (class in sage.geometry.polyhedron.base_ZZ), 532
Polyhedron_ZZ_normaliz (class in sage.geometry.polyhedron.backend_normaliz), 545
Polyhedron_ZZ_polymake (class in sage.geometry.polyhedron.backend_polymake), 548
Polyhedron_ZZ_ppl (class in sage.geometry.polyhedron.backend_ppl), 550
PolyhedronFace (class in sage.geometry.polyhedron.face), 141
PolyhedronRepresentation (class in sage.geometry.polyhedron.representation), 124
Polytopes (class in sage.geometry.polyhedron.library), 62
poset_of_regions() (sage.geometry.pseudolines.PseudolineArrangement method), 30
positive_circuits() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 575
positive_integer_relations() (in module sage.geometry.lattice_polytope), 203
positive_operators_gens() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 306
prefix_check() (in module sage.rings.polynomial.groebner_fan), 410
preimage_cones() (sage.geometry.fan.morphism.FanMorphism method), 376
preimage_fan() (sage.geometry.fan.morphism.FanMorphism method), 377
prepare_inner_loop() (sage.geometry.integral_points.InequalityCollection method), 653
prepare_next_to_inner_loop() (sage.geometry.integral_points.InequalityCollection method), 654
primitive() (sage.geometry.hyperplane_arrangement.hyperplane.Hyperplane method), 52
primitive_collections() (sage.geometry.fan.RationalPolyhedralFan method), 360
primitive_preimage_cones() (sage.geometry.fan.morphism.FanMorphism method), 377
print_cache() (in module sage.geometry.integral_points), 658
prism() (sage.geometry.polyhedron.base.Polyhedron_base method), 495
Problem (class in sage.geometry.polyhedron.double_description), 556
product() (sage.geometry.polyhedral_complex.PolyhedralComplex method), 247
product() (sage.geometry.polyhedron.base.Polyhedron_base method), 495
project_points() (in module sage.geometry.polyhedron.library), 102
projection() (sage.geometry.polyhedron.base.Polyhedron_base method), 496
projection_func_identity() (in module sage.geometry.polyhedron.plot), 140
Projection (class in sage.geometry.polyhedron.plot), 131
projection() (sage.geometry.polyhedron.base.Polyhedron_base method), 496
PseudolineArrangement (class in sage.geometry.pseudolines), 641
pushing_triangulation() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 576
pyramid() (sage.geometry.polyhedron.base.Polyhedron_base method), 496
quotient() (sage.geometry.toric_lattice.ToricLattice_generic method), 259
R_by_sign() (sage.geometry.polyhedron.double_description.DoubleDescriptionPair method), 553
radius() (sage.geometry.polyhedron.base.Polyhedron_base method), 496
Radius_square() (sage.geometry.polyhedron.base.Polyhedron_base method), 496
random_cone() (in module sage.geometry.cone), 326
Random_element() (sage.geometry.cone.ConvexRationalPolyhedralCone method), 308
Random_element() (sage.geometry.linear_expression.LinearExpressionMethod method), 617
random_inequalities() (in module sage.geometry.polyhedron.double_description), 559
random_integral_point() (sage.geometry.polyhedron.base.Polyhedron_base method), 497
Index
representative_point()

reset_options() (in module sage.geometry.toric_plotter), 395

restrict_to_connected_triangulations()

restrict_to_fine_triangulations()

restrict_to_regular_triangulations()

remove_cell() (sage.geometry.polyhedral_complex.PolyhedralComplex method), 248

ring() (sage.rings.polynomial.groebner_fan.GroebnerFan method), 401

ring() (sage.rings.polynomial.groebner_fan.GroebnerFan method), 400

render3d() (sage.rings.polynomial.groebner_fan.GroebnerFan method), 401

render_6d() (sage.geometry.polyhedron.plot.Projection method), 131

render_1d() (sage.geometry.polyhedron.plot.Projection method), 132

render_2d() (sage.geometry.polyhedron.plot.Projection method), 132

render_3d() (sage.geometry.polyhedron.plot.Projection method), 132

render_fill_2d() (sage.geometry.polyhedron.plot.Projection method), 133

render_line_1d() (sage.geometry.polyhedron.plot.Projection method), 133

render_points_1d() (sage.geometry.polyhedron.plot.Projection method), 134

render_points_2d() (sage.geometry.polyhedron.plot.Projection method), 134

render_solid() (sage.geometry.polyhedron.base.Polyhedron_base method), 499

render_solid_3d() (sage.geometry.polyhedron.plot.Projection method), 135

runcinated_one_hundred_twenty_cell()

runcitruncated_one_hundred_twenty_cell()

runcitruncated_six_hundred_cell()
<table>
<thead>
<tr>
<th>Module Name</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>sage.geometry.polyhedron.library.Polytopes</td>
<td>88</td>
</tr>
<tr>
<td>sage.geometry.convex_set</td>
<td>601</td>
</tr>
<tr>
<td>sage.geometry.fan</td>
<td>333</td>
</tr>
<tr>
<td>sage.geometry.fan_isomorphism</td>
<td>647</td>
</tr>
<tr>
<td>sage.geometry.fan_morphism</td>
<td>364</td>
</tr>
<tr>
<td>sage.geometry.hasse_diagram</td>
<td>651</td>
</tr>
<tr>
<td>sage.geometry.hyperplane_arrangement.affine_subspace</td>
<td>53</td>
</tr>
<tr>
<td>sage.geometry.hyperplane_arrangement.arrangement</td>
<td>3</td>
</tr>
<tr>
<td>sage.geometry.hyperplane_arrangement.check_freeness</td>
<td>661</td>
</tr>
<tr>
<td>sage.geometry.hyperplane_arrangement.hyperplane</td>
<td>46</td>
</tr>
<tr>
<td>sage.geometry.hyperplane_arrangement.library</td>
<td>40</td>
</tr>
<tr>
<td>sage.geometry.hyperplane_arrangement.plot</td>
<td>56</td>
</tr>
<tr>
<td>sage.geometry.integral_points</td>
<td>652</td>
</tr>
<tr>
<td>sage.geometry.lattice_polytope</td>
<td>155</td>
</tr>
<tr>
<td>sage.geometry.linear_expression</td>
<td>612</td>
</tr>
<tr>
<td>sage.geometry.newton_polygon</td>
<td>618</td>
</tr>
<tr>
<td>sage.geometry.point_collection</td>
<td>379</td>
</tr>
<tr>
<td>sage.geometry.polyhedral_complex</td>
<td>228</td>
</tr>
<tr>
<td>sage.geometry.polyhedron.backend_cdd</td>
<td>538</td>
</tr>
<tr>
<td>sage.geometry.polyhedron.backend_field</td>
<td>539</td>
</tr>
<tr>
<td>sage.geometry.polyhedron.backend_normaliz</td>
<td>540</td>
</tr>
<tr>
<td>sage.geometry.polyhedron.backend_ppl</td>
<td>547</td>
</tr>
<tr>
<td>sage.geometry.polyhedron.base</td>
<td>411</td>
</tr>
<tr>
<td>sage.geometry.polyhedron.base_QQ</td>
<td>526</td>
</tr>
<tr>
<td>sage.geometry.polyhedron.base_RDF</td>
<td>538</td>
</tr>
<tr>
<td>sage.geometry.polyhedron.base_ZZ</td>
<td>532</td>
</tr>
<tr>
<td>sage.geometry.polyhedron.cdd_file_format</td>
<td>152</td>
</tr>
<tr>
<td>sage.geometry.polyhedron.constructor</td>
<td>103</td>
</tr>
<tr>
<td>sage.geometry.polyhedron.double_description</td>
<td>552</td>
</tr>
<tr>
<td>sage.geometry.polyhedron.double_description_inhomogeneous</td>
<td>560</td>
</tr>
<tr>
<td>sage.geometry.polyhedron.face</td>
<td>140</td>
</tr>
<tr>
<td>sage.geometry.polyhedron.lattice_euclidean_group_element</td>
<td>207</td>
</tr>
<tr>
<td>sage.geometry.polyhedron.library</td>
<td>61</td>
</tr>
<tr>
<td>sage.geometry.polyhedron.modules.formal_polyhedra_module</td>
<td>153</td>
</tr>
<tr>
<td>sage.geometry.polyhedron.palp_database</td>
<td>209</td>
</tr>
<tr>
<td>sage.geometry.polyhedron.parent</td>
<td>111</td>
</tr>
<tr>
<td>sage.geometry.polyhedron.ppl_lattice_polygon</td>
<td>131</td>
</tr>
<tr>
<td>sage.geometry.polyhedron.ppl_lattice_polytope</td>
<td>211</td>
</tr>
<tr>
<td>sage.geometry.polyhedron.ppl_lattice_polytope</td>
<td>215</td>
</tr>
<tr>
<td>sage.geometry.polyhedron.representation</td>
<td>117</td>
</tr>
<tr>
<td>sage.geometry.pseudolines</td>
<td>639</td>
</tr>
<tr>
<td>sage.geometry.relative_interior</td>
<td>622</td>
</tr>
<tr>
<td>sage.geometry.ribbon_graph</td>
<td>626</td>
</tr>
<tr>
<td>sage.geometry.toric_lattice</td>
<td>252</td>
</tr>
<tr>
<td>sage.geometry.toric_plotter</td>
<td>387</td>
</tr>
<tr>
<td>sage.geometry.triangulation.base</td>
<td>584</td>
</tr>
<tr>
<td>sage.geometry.triangulation.element</td>
<td>593</td>
</tr>
<tr>
<td>sage.geometry.triangulation.point_configuration</td>
<td>565</td>
</tr>
<tr>
<td>sage.geometry.voronoi_diagram</td>
<td>643</td>
</tr>
<tr>
<td>sage.rings.polynomial.groebner_fan</td>
<td>396</td>
</tr>
</tbody>
</table>
T

tetrahedron() (sage.geometry.polyhedron.library.Polytopes method), 93
tikz() (sage.geometry.polyhedron.base.Polyhedron_base method), 509
tikz() (sage.geometry.polyhedron.plot.Projection method), 136
to_linear_program() (sage.geometry.polyhedron.base.Polyhedron_base method), 510
to_RationalPolyhedralFan() (sage.rings.polynomial.groebner_fan.PolyhedralFan method), 407
to_symmetric_space() (sage.geometry.polyhedron.library.Polytopes method), 58

ToricLattice_ambient (class in sage.geometry.toric_lattice), 256
ToricLattice_generic (class in sage.geometry.toric_lattice), 257
ToricLattice_quotient (class in sage.geometry.toric_lattice), 261
ToricLattice_quotient_element (class in sage.geometry.toric_lattice), 265
ToricLattice_sublattice (class in sage.geometry.toric_lattice), 265
ToricLattice_sublattice_with_basis (class in sage.geometry.toric_lattice), 266
ToricLatticeFactory (class in sage.geometry.toric_lattice), 254
ToricPlotter (class in sage.geometry.toric_plotter), 387
translation() (sage.geometry.convex_set.ConvexSet_base method), 609
translation() (sage.geometry.polyhedron.base.Polyhedron_base method), 511
translation() (sage.geometry.relative_interior.RelativeInterior method), 625
transpositions() (sage.geometry.pseudolines.PseudolineArrangement method), 642
traverse_boundary() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 188
triangulate() (sage.geometry.polyhedron.base.Polyhedron_base method), 512
triangulate() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 581
Triangulation (class in sage.geometry.triangulation.point_configuration.PointConfiguration), 593
triangulation_render_2d() (in module sage.geometry.triangulation.element), 598
triangulation_render_3d() (in module sage.geometry.triangulation.element), 598
sage.geometry.triangulation.element, 598

triangulations() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 582

triangulations_list() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 583

trivial() (in module sage.geometry.cone_catalog), 332

tropical_basis() (sage.rings.polynomial.groebner_fan.GroebnerFan method), 401

tropical_intersection() (sage.rings.polynomial.groebner_fan.GroebnerFan method), 401

TropicalPrevariety (class in sage.rings.polynomial.groebner_fan), 409

truncated_cube() (sage.geometry.polyhedron.library.Polytopes method), 93

truncated_dodecahedron() (sage.geometry.polyhedron.library.Polytopes method), 94

truncated_icosidodecahedron() (sage.geometry.polyhedron.library.Polytopes method), 95

truncated_octahedron() (sage.geometry.polyhedron.library.Polytopes method), 96

truncated_tetrahedron() (sage.geometry.polyhedron.library.Polytopes method), 97

truncation() (sage.geometry.polyhedron.base.Polyhedron_base method), 513

twenty_four_cell() (sage.geometry.polyhedron.library.Polytopes method), 97

type() (sage.geometry.polyhedron.representation.Equation method), 117

type() (sage.geometry.polyhedron.representation.Inequality method), 122

type() (sage.geometry.polyhedron.representation.Line method), 123

type() (sage.geometry.polyhedron.representation.Ray method), 126

type() (sage.geometry.polyhedron.representation.Vertex method), 127

U

unbounded_regions() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 34

union() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 251

union() (sage.geometry.polyhedral_complex.PolyhedralComplex method), 251

universe() (sage.geometry.polyhedron.parent.Polyhedra_base method), 116

V

varchenko_matrix() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 35

vector() (sage.geometry.polyhedron.representation.PolyhedronRepresentation method), 125

verify() (sage.geometry.polyhedron.double_description.DoubleDescriptionPair method), 556

verify() (sage.geometry.polyhedron.double_description_inhomogeneous.Hrep2Vrep method), 561

verify() (sage.geometry.polyhedron.double_description_inhomogeneous.Vrep2Hrep method), 563

Vertex (class in sage.geometry.polyhedron.representation), 126

vertex() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 188

vertex_adjacency_matrix() (sage.geometry.polyhedron.base.Polyhedron_base method), 513

vertex_digraph() (sage.geometry.polyhedron.base.Polyhedron_base method), 515

vertex_facet_graph() (sage.geometry.polyhedron.base.Polyhedron_base method), 516

vertex_facet_pairing_matrix() (sage.geometry.polyhedron.base.Polyhedron_base method), 188

vertex_generator() (sage.geometry.polyhedron.base.Polyhedron_base method), 517

vertex_generator() (sage.geometry.polyhedron.face.PolyhedronFace method), 151

vertex_graph() (sage.geometry.fan.RationalPolyhedralFan method), 362

vertex_graph() (sage.geometry.polyhedron.base.Polyhedron_base method), 518

vertices() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 36

vertices() (sage.geometry.lattice_polytope.LatticePolytopeClass method), 36

vertices() (sage.geometry.polyhedron.base.Polyhedron_base method), 189

vertices() (sage.geometry.newton_polygon.NewtonPolygon_element method), 619

vertices() (sage.geometry.polyhedron.base.Polyhedron_base method), 151
vertices() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope_PPL_class method), 227
vertices_list() (sage.geometry.polyhedron.base.Polyhedron_base method), 519
vertices_matrix() (sage.geometry.polyhedron.base.Polyhedron_base method), 519
vertices_saturating() (sage.geometry.polyhedron.ppl_lattice_polytope.LatticePolytope_PPL_class method), 227
verts_for_normal() (in module sage.rings.polynomial.groebner_fan), 410
virtual_rays() (sage.geometry.fan.RationalPolyhedralFan method), 362
volume() (sage.geometry.polyhedron.base.Polyhedron_base method), 520
volume() (sage.geometry.triangulation.point_configuration.PointConfiguration method), 583
VoronoiDiagram (class in sage.geometry.voronoi_diagram), 643
Vrep2Hrep (class in sage.geometry.polyhedron.double_description_inhomogeneous), 562
Vrep_generator() (sage.geometry.polyhedron.base.Polyhedron_base method), 413
Vrepresentation (class in sage.geometry.polyhedron.representation), 128
Vrepresentation() (sage.geometry.polyhedron.backend_ppl.Polyhedron_ppl method), 551
Vrepresentation() (sage.geometry.polyhedron.base.Polyhedron_base method), 413
Vrepresentation_space() (sage.geometry.polyhedron.base.Polyhedron_base method), 414
Vrepresentation_space() (sage.geometry.polyhedron.parent.Polyhedra_base method), 113

W
wedge() (sage.geometry.polyhedral_complex.PolyhedralComplex method), 252
wedge() (sage.geometry.polyhedron.base.Polyhedron_base method), 523
weight_vectors() (sage.rings.polynomial.groebner_fan.GroebnerFan method), 402
whitney_data() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 37
whitney_number() (sage.geometry.hyperplane_arrangement.arrangement.HyperplaneArrangementElement method), 37
write_cdd_Hrepresentation() (sage.geometry.polyhedron.base.Polyhedron_base method), 525
write_cdd_Vrepresentation() (sage.geometry.polyhedron.base.Polyhedron_base method), 525