Sage Reference Manual: Combinatorial Geometry

Release 6.3

The Sage Development Team

August 11, 2014

CONTENTS

Sage includes classes for convex rational polyhedral cones and fans, Groebner fans, lattice and reflexive polytopes (with integral coordinates), and generic polytopes and polyhedra (with rational or numerical coordinates).

CHAPTER

ONE

TORIC LATTICES

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

All toric lattices are isomorphic to $\mathbf{Z}^n$ for some n, but will prevent you from doing "wrong" operations with objects from different lattices.

AUTHORS:

- Andrey Novoseltsev (2010-05-27): initial version.
- Andrey Novoseltsev (2010-07-30): sublattices and quotients.

EXAMPLES:

The simplest way to create a toric lattice is to specify its dimension only:

```
sage: N = ToricLattice(3)
sage: N
3-d lattice N
```

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:

```
sage: n + m
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for '+':
'3-d lattice N' and '3-d lattice M'
sage: n * n
Traceback (most recent call last):
...
TypeError: elements of the same toric lattice cannot be multiplied!
sage: n == m
False
```

Note that `n` and `m` are not equal to each other even though they are both "just (1,2,3)." Moreover, you cannot easily convert elements between toric lattices:

```
sage: M(n)
Traceback (most recent call last):
...
TypeError: N(1, 2, 3) cannot be converted to 3-d lattice M!
```

If you really need to consider elements of one lattice as elements of another, you can either use intermediate conversion to "just a vector":

```
sage: ZZ3 = ZZ^3
sage: n_in_M = M(ZZ3(n))
sage: n_in_M
M(1, 2, 3)
sage: n == n_in_M
False
sage: n_in_M == m
True
```

Or you can create a homomorphism from one lattice to any other:

```
sage: h = N.hom(identity_matrix(3), M)
sage: h(n)
M(1, 2, 3)
```

Warning: While integer vectors (elements of $\mathbf{Z}^n$) are printed as `(1,2,3)`, in the code `(1,2,3)` is a `tuple`, which has nothing to do neither with vectors, nor with toric lattices, so the following is probably not what you want while working with toric geometry objects:

```
sage: (1,2,3) + (1,2,3)
(1, 2, 3, 1, 2, 3)
```

Instead, use syntax like

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

class sage.geometry.toric_lattice.**ToricLatticeFactory**

Bases: sage.structure.factory.UniqueFactory

Create a lattice for toric geometry objects.

INPUT:

- `rank` – nonnegative integer, the only mandatory parameter;
- `name` – string;
- `dual_name` – string;
- `latex_name` – string;
- `latex_dual_name` – string.

OUTPUT:

- lattice.

A toric lattice is uniquely determined by its rank and associated names. There are four such "associated names" whose meaning should be clear from the names of the corresponding parameters, but the choice of default values is a little bit involved. So here is the full description of the "naming algorithm":

1. If no names were given at all, then this lattice will be called "N" and the dual one "M". These are the standard choices in toric geometry.
2. If `name` was given and `dual_name` was not, then `dual_name` will be `name` followed by "*".
3. If LaTeX names were not given, they will coincide with the "usual" names, but if `dual_name` was constructed automatically, the trailing star will be typeset as a superscript.

EXAMPLES:

Let's start with no names at all and see how automatic names are given:

```
sage: L1 = ToricLattice(3)
sage: L1
3-d lattice N
sage: L1.dual()
3-d lattice M
```

If we give the name "N" explicitly, the dual lattice will be called "N*":

```
sage: L2 = ToricLattice(3, "N")
sage: L2
3-d lattice N
sage: L2.dual()
3-d lattice N*
```

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

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

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

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

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

```
sage: L1 == L2
False
sage: L1 == L3
True
sage: L1 is L3
True
sage: L1 == L4
False
```

The reason for this is that `L2` and `L4` have different names either for dual lattices or for LaTeX typesetting.

create_key(*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.

TESTS:

```
sage: ToricLattice.create_key(3)
(3, 'N', 'M', 'N', 'M')
sage: N = ToricLattice(3)
sage: loads(dumps(N)) is N
True
sage: TestSuite(N).run()
```

create_object(*version*, *key*)

Create the toric lattice described by `key`.

See `ToricLattice` for documentation.

> **Warning:** You probably should not use this function directly.

TESTS:

```
sage: key = ToricLattice.create_key(3)
sage: ToricLattice.create_object(1, key)
3-d lattice N
```

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

Bases: sage.geometry.toric_lattice.ToricLattice_generic, sage.modules.free_module.FreeModule_ambient_pid

Create a toric lattice.

See ToricLattice for documentation.

Warning: There should be only one toric lattice with the given rank and associated names. Using this class directly to create toric lattices may lead to unexpected results. Please, use ToricLattice to create toric lattices.

TESTS:

```
sage: N = ToricLattice(3, "N", "M", "N", "M")
sage: N
3-d lattice N
sage: TestSuite(N).run()
```

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()
```

class sage.geometry.toric_lattice.**ToricLattice_generic**(*base_ring*, *rank*, *degree*, *sparse=False*)

Bases: `sage.modules.free_module.FreeModule_generic_pid`

Abstract base class for toric lattices.

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.

TESTS:

```
sage: print ToricLattice(3).construction()
None
```

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 $\mathbf{Z}$-modules. If `other` is a `ToricLattice`, another toric lattice will be returned.

EXAMPLES:

```
sage: K = ToricLattice(3, 'K')
sage: L = ToricLattice(3, 'L')
sage: N = K.direct_sum(L); N
6-d lattice K+L
sage: N, N.dual(), latex(N), latex(N.dual())
(6-d lattice K+L, 6-d lattice K*+L*, K \oplus L, K^* \oplus L^*)
```

With default names:

```
sage: N = ToricLattice(3).direct_sum(ToricLattice(2))
sage: N, N.dual(), latex(N), latex(N.dual())
(5-d lattice N+N, 5-d lattice M+M, N \oplus N, M \oplus M)
```

If `other` is not a `ToricLattice`, fall back to sum of modules:

```
sage: ToricLattice(3).direct_sum(ZZ^2)
Free module of degree 5 and rank 5 over Integer Ring
Echelon basis matrix:
[1 0 0 0 0]
[0 1 0 0 0]
[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*)

Return the quotient of `self` by the given sublattice `sub`.

INPUT:

- `sub` – sublattice of self;
- `check` – (default: True) whether or not to check that `sub` is a valid sublattice.

If the quotient is one-dimensional and torsion free, the following two mutually exclusive keyword arguments are also allowed. They decide the sign choice for the (single) generator of the quotient lattice:

- `positive_point` – a lattice point of `self` not in the sublattice `sub` (that is, not zero in the quotient lattice). The quotient generator will be in the same direction as `positive_point`.
- `positive_dual_point` – a dual lattice point. The quotient generator will be chosen such that its lift has a positive product with `positive_dual_point`. Note: if `positive_dual_point` is not zero on the sublattice `sub`, then the notion of positivity will depend on the choice of lift!

EXAMPLES:

```
sage: N = ToricLattice(3)
sage: Ns = N.submodule([N(2,4,0), N(9,12,0)])
sage: Q = N/Ns
sage: Q
Quotient with torsion of 3-d lattice N
by Sublattice <N(1, 8, 0), N(0, 12, 0)>
```

See ToricLattice_quotient for more examples.

saturation()

Return the saturation of self.

OUTPUT:

- a toric lattice.

EXAMPLES:

```
sage: N = ToricLattice(3)
sage: Ns = N.submodule([(1,2,3), (4,5,6)])
sage: Ns
Sublattice <N(1, 2, 3), N(0, 3, 6)>
sage: Ns_sat = Ns.saturation()
sage: Ns_sat
Sublattice <N(1, 0, -1), N(0, 1, 2)>
sage: Ns_sat is Ns_sat.saturation()
True
```

span(**args*, ***kwds*)

Return the span of the given generators.

INPUT:

- gens – list of elements of the ambient vector space of self.

OUTPUT:

- submodule spanned by gens.

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

See also span_of_basis(), submodule(), and submodule_with_basis(),

EXAMPLES:

```
sage: N = ToricLattice(3)
sage: Ns = N.submodule([N.gen(0)])
sage: Ns.span([N.gen(1)])
Sublattice <N(0, 1, 0)>
sage: Ns.submodule([N.gen(1)])
Traceback (most recent call last):
...
ArithmeticError: Argument gens (= [N(0, 1, 0)])
does not generate a submodule of self.
```

span_of_basis(**args*, ***kwds*)

Return the submodule with the given basis.

INPUT:

- basis – list of elements of the ambient vector space of self.

OUTPUT:

- submodule spanned by basis.

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

See also span(), submodule(), and submodule_with_basis(),

EXAMPLES:

```
sage: N = ToricLattice(3)
sage: Ns = N.span_of_basis([(1,2,3)])
sage: Ns.span_of_basis([(2,4,0)])
Sublattice <N(2, 4, 0)>
sage: Ns.span_of_basis([(1/5,2/5,0), (1/7,1/7,0)])
Sublattice <(1/5, 2/5, 0), (1/7, 1/7, 0)>
```

Of course the input basis vectors must be linearly independent:

```
sage: Ns.span_of_basis([(1,2,0), (2,4,0)])
Traceback (most recent call last):
...
ValueError: The given basis vectors must be linearly independent.
```

class sage.geometry.toric_lattice.**ToricLattice_quotient**(*V*, *W*, *check=True*, *positive_point=None*, *positive_dual_point=None*)

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!

OUTPUT:

- quotient of V by W.

EXAMPLES:

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

```
sage: N = ToricLattice(3)
sage: sublattice = N.submodule([(1,1,0), (3,2,1)])
sage: Q = N/sublattice
sage: Q
```

```
1-d lattice, quotient of 3-d lattice N by Sublattice <N(1, 0, 1), N(0, 1, -1)>
sage: Q.gens()
(N[0, 0, 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:

```
sage: Q = N.quotient(sublattice, positive_point=N(0,0,-1)); Q
1-d lattice, quotient of 3-d lattice N by Sublattice <N(1, 0, 1), N(0, 1, -1)>
sage: Q.gens()
(N[0, 0, -1],)
```

or:

```
sage: M = N.dual()
sage: Q = N.quotient(sublattice, positive_dual_point=M(0,0,-1)); Q
1-d lattice, quotient of 3-d lattice N by Sublattice <N(1, 0, 1), N(0, 1, -1)>
sage: Q.gens()
(N[0, 0, -1],)
```

TESTS:

```
sage: loads(dumps(Q)) == Q
True
sage: loads(dumps(Q)).gens() == Q.gens()
True
```

Element
alias of `ToricLattice_quotient_element`

base_extend(*R*)
Return the base change of `self` to the ring `R`.

INPUT:

- `R` – either $\mathbf{Z}$ or $\mathbf{Q}$.

OUTPUT:

- `self` if $R = \mathbf{Z}$, quotient of the base extension of the ambient lattice by the base extension of the sublattice if $R = \mathbf{Q}$.

EXAMPLES:

```
sage: N = ToricLattice(3)
sage: Ns = N.submodule([N(2,4,0), N(9,12,0)])
sage: Q = N/Ns
sage: Q.base_extend(ZZ) is Q
True
sage: Q.base_extend(QQ)
Vector space quotient V/W of dimension 1 over Rational Field where
V: Vector space of dimension 3 over Rational Field
W: Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[1 0 0]
[0 1 0]
```

coordinate_vector(*x*, *reduce=False*)
Return coordinates of x with respect to the optimized representation of self.

INPUT:

- x – element of self or convertable to self.
- reduce – (default: False); if True, reduce coefficients modulo invariants.

OUTPUT:

The coordinates as a vector.

EXAMPLES:

```
sage: N = ToricLattice(3)
sage: Q = N.quotient(N.span([N(1,2,3), N(0,2,1)]), positive_point=N(0,-1,0))
sage: q = Q.gen(0); q
N[0, -1, 0]
sage: q.vector()  # indirect test
(1)
sage: Q.coordinate_vector(q)
(1)
```

dimension()

Return the rank of self.

OUTPUT:

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

EXAMPLES:

```
sage: N = ToricLattice(3)
sage: Ns = N.submodule([N(2,4,0), N(9,12,0)])
sage: Q = N/Ns
sage: Q.ngens()
2
sage: Q.rank()
1
sage: Ns = N.submodule([N(1,4,0)])
sage: Q = N/Ns
sage: Q.ngens()
2
sage: Q.rank()
2
```

dual()

Return the lattice dual to self.

OUTPUT:

- a toric lattice quotient.

EXAMPLES:

```
sage: N = ToricLattice(3)
sage: Ns = N.submodule([(1, -1, -1)])
sage: Q = N / Ns
sage: Q.dual()
Sublattice <M(1, 0, 1), M(0, 1, -1)>
```

gens()

Return the generators of the quotient.

OUTPUT:

A tuple of ToricLattice_quotient_element generating the quotient.

EXAMPLES:

```
sage: N = ToricLattice(3)
sage: Q = N.quotient(N.span([N(1,2,3), N(0,2,1)]), positive_point=N(0,-1,0))
sage: Q.gens()
(N[0, -1, 0],)
```

is_torsion_free()

Check if `self` is torsion-free.

OUTPUT:

- `True` is `self` has no torsion and `False` otherwise.

EXAMPLES:

```
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
```

rank()

Return the rank of `self`.

OUTPUT:

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

EXAMPLES:

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

TESTS:

```
sage: N = ToricLattice(3)
sage: sublattice = N.submodule([(1,1,0), (3,2,1)])
sage: Q = N/sublattice
sage: e = Q(1,2,3)
sage: e
N[1, 2, 3]
sage: e2 = Q(N(2,3,3))
sage: e2
N[2, 3, 3]
sage: e == e2
True
sage: e.vector()
(4)
sage: e2.vector()
(4)
```

set_immutable()

Make `self` immutable.

OUTPUT:

•none.

Note: Elements of toric lattice quotients are always immutable, so this method does nothing, it is introduced for compatibility purposes only.

EXAMPLES:

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

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

Bases: sage.geometry.toric_lattice.ToricLattice_sublattice_with_basis, sage.modules.free_module.FreeModule_submodule_pid

Construct the sublattice of `ambient` toric lattice generated by `gens`.

INPUT (same as for `FreeModule_submodule_pid`):

•`ambient` – ambient `toric lattice` for this sublattice;

•`gens` – list of elements of `ambient` generating the constructed sublattice;

•see the base class for other available options.

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()
```

Now we plot both the ambient lattice and its sublattice:

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

sage.geometry.toric_lattice.**is_ToricLattice**(*x*)

Check if `x` is a toric lattice.

INPUT:

- `x` – anything.

OUTPUT:

- `True` if `x` is a toric lattice and `False` otherwise.

EXAMPLES:

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

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

CHAPTER

TWO

CONVEX RATIONAL POLYHEDRAL CONES

This module was designed as a part of framework for toric varieties (`variety`, `fano_variety`). While the emphasis is on strictly convex cones, non-strictly convex cones are supported as well. Work with distinct lattices (in the sense of discrete subgroups spanning vector spaces) is supported. The default lattice is `ToricLattice` N of the appropriate dimension. The only case when you must specify lattice explicitly is creation of a 0-dimensional cone, where dimension of the ambient space cannot be guessed.

AUTHORS:

- Andrey Novoseltsev (2010-05-13): initial version.
- Andrey Novoseltsev (2010-06-17): substantial improvement during review by Volker Braun.
- Volker Braun (2010-06-21): various spanned/quotient/dual lattice computations added.
- Volker Braun (2010-12-28): Hilbert basis for cones.
- Andrey Novoseltsev (2012-02-23): switch to PointCollection container.

EXAMPLES:

Use `Cone()` to construct cones:

```
sage: octant = Cone([(1,0,0), (0,1,0), (0,0,1)])
sage: halfspace = Cone([(1,0,0), (0,1,0), (-1,-1,0), (0,0,1)])
sage: positive_xy = Cone([(1,0,0), (0,1,0)])
sage: four_rays = Cone([(1,1,1), (1,-1,1), (-1,-1,1), (-1,1,1)])
```

For all of the cones above we have provided primitive generating rays, but in fact this is not necessary - a cone can be constructed from any collection of rays (from the same space, of course). If there are non-primitive (or even non-integral) rays, they will be replaced with primitive ones. If there are extra rays, they will be discarded. Of course, this means that `Cone()` has to do some work before actually constructing the cone and sometimes it is not desirable, if you know for sure that your input is already "good". In this case you can use options `check=False` to force `Cone()` to use exactly the directions that you have specified and `normalize=False` to force it to use exactly the rays that you have specified. However, it is better not to use these possibilities without necessity, since cones are assumed to be represented by a minimal set of primitive generating rays. See `Cone()` for further documentation on construction.

Once you have a cone, you can perform numerous operations on it. The most important ones are, probably, ray accessing methods:

```
sage: rays = halfspace.rays()
sage: rays
N( 0,  0, 1),
N( 0,  1, 0),
```

```
N( 0, -1, 0),
N( 1,  0, 0),
N(-1,  0, 0)
in 3-d lattice N
sage: rays.set()
frozenset([N(1, 0, 0), N(-1, 0, 0), N(0, 1, 0), N(0, 0, 1), N(0, -1, 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 `rays()` returns a `PointCollection` with the i-th element being the primitive integral generator of the i-th ray. It is possible to convert this collection to a matrix with either rows or columns corresponding to these generators. You may also change the default `output_format()` of all point collections to be such a matrix.

If you want to do something with each ray of a cone, you can write

```
sage: for ray in positive_xy: print ray
N(1, 0, 0)
N(0, 1, 0)
```

There are two dimensions associated to each cone - the dimension of the subspace spanned by the cone and the dimension of the space where it lives:

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

You also may be interested in this dimension:

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

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

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

You can also perform these checks:

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

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

```
sage: face = four_rays.faces(dim=2)[0]
sage: face
2-d face of 3-d cone in 3-d lattice N
sage: face.rays()
N(-1, -1, 1),
N(-1,  1, 1)
in 3-d lattice N
sage: face.ambient_ray_indices()
(2, 3)
sage: four_rays.rays(face.ambient_ray_indices())
N(-1, -1, 1),
N(-1,  1, 1)
in 3-d lattice N
```

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

```
sage: L = four_rays.face_lattice()
sage: map(len, 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:

`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, $\mathbf{Z}^n$, or any other object that behaves like these. If not specified, an attempt will be made to determine an appropriate toric lattice automatically;
- check – by default the input data will be checked for correctness (e.g. that all rays have the same number of components) and generating rays will be constructed from rays. If you know that the input is a minimal set of generators of a valid cone, you may significantly decrease construction time using check=False option;
- normalize – you can further speed up construction using normalize=False option. In this case rays must be a list of immutable primitive rays in lattice. In general, you should not use this option, it is designed for code optimization and does not give as drastic improvement in speed as the previous one.

OUTPUT:

- convex rational polyhedral cone determined by rays.

EXAMPLES:

Let's define a cone corresponding to the first quadrant of the plane (note, you can even mix objects of different types to represent rays, as long as you let this function to perform all the checks and necessary conversions!):

```
sage: quadrant = Cone([(1,0), [0,1]])
sage: quadrant
2-d cone in 2-d lattice N
sage: quadrant.rays()
N(1, 0),
N(0, 1)
in 2-d lattice N
```

If you give more rays than necessary, the extra ones will be discarded:

```
sage: Cone([(1,0), (0,1), (1,1), (0,1)]).rays()
N(0, 1),
N(1, 0)
in 2-d lattice N
```

However, this work is not done with check=False option, so use it carefully!

```
sage: Cone([(1,0), (0,1), (1,1), (0,1)], check=False).rays()
N(1, 0),
N(0, 1),
N(1, 1),
N(0, 1)
in 2-d lattice N
```

Even worse things can happen with normalize=False option:

```
sage: Cone([(1,0), (0,1)], check=False, normalize=False)
Traceback (most recent call last):
...
AttributeError: 'tuple' object has no attribute 'parent'
```

You can construct different "not" cones: not full-dimensional, not strictly convex, not containing any rays:

```
sage: one_dimensional_cone = Cone([(1,0)])
sage: one_dimensional_cone.dim()
1
sage: half_plane = Cone([(1,0), (0,1), (-1,0)])
sage: half_plane.rays()
N( 0, 1),
```

```
N( 1, 0),
N(-1, 0)
in 2-d lattice N
sage: half_plane.is_strictly_convex()
False
sage: origin = Cone([(0,0)])
sage: origin.rays()
Empty collection
in 2-d lattice N
sage: origin.dim()
0
sage: origin.lattice_dim()
2
```

You may construct the cone above without giving any rays, but in this case you must provide `lattice` explicitly:

```
sage: origin = Cone([])
Traceback (most recent call last):
...
ValueError: lattice must be given explicitly if there are no rays!
sage: origin = Cone([], lattice=ToricLattice(2))
sage: origin.dim()
0
sage: origin.lattice_dim()
2
sage: origin.lattice()
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),
N( 0, -1),
N( 1,  0),
N(-1,  0)
in 2-d lattice N
```

The cone can also be specified by a Polyhedron_base:

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

TESTS:

```
sage: N = ToricLattice(2)
sage: Nsub = N.span([ N(1,2) ])
sage: Cone(Nsub.basis())
1-d cone in Sublattice <N(1, 2)>
sage: Cone([N(0)])
0-d cone in 2-d lattice N
```

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

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

Create a convex rational polyhedral cone.

Warning: This class does not perform any checks of correctness of input nor does it convert input into the standard representation. Use Cone() to construct cones.

Cones are immutable, but they cache most of the returned values.

INPUT:

The input can be either:

- rays – list of immutable primitive vectors in lattice;
- lattice – ToricLattice, $\mathbf{Z}^n$, or any other object that behaves like these. If None, it will be determined as parent() of the first ray. Of course, this cannot be done if there are no rays, so in this case you must give an appropriate lattice directly.

or (these parameters must be given as keywords):

- ambient – ambient structure of this cone, a bigger cone or a fan, this cone *must be a face of* ambient;
- ambient_ray_indices – increasing list or tuple of integers, indices of rays of ambient generating this cone.

In both cases, the following keyword parameter may be specified in addition:

- PPL – either None (default) or a C_Polyhedron representing the cone. This serves only to cache the polyhedral data if you know it already. The polyhedron will be set immutable.

OUTPUT:

- convex rational polyhedral cone.

Note: Every cone has its ambient structure. If it was not specified, it is this cone itself.

Hilbert_basis()

Return the Hilbert basis of the cone.

Given a strictly convex cone $C \subset \mathbf{R}^d$, the Hilbert basis of C is the set of all irreducible elements in the semigroup $C \cap \mathbf{Z}^d$. It is the unique minimal generating set over $\mathbf{Z}$ for the integral points $C \cap \mathbf{Z}^d$.

If the cone C is not strictly convex, this method finds the (unique) minimial set of lattice points that need to be added to the defining rays of the cone to generate the whole semigroup $C \cap \mathbf{Z}^d$. But because the rays of the cone are not unique nor necessarily minimal in this case, neither is the returned generating set (consisting of the rays plus additional generators).

See also `semigroup_generators()` if you are not interested in a minimal set of generators.

OUTPUT:

- a `PointCollection`. The rays of `self` are the first `self.nrays()` entries.

EXAMPLES:

The following command ensures that the output ordering in the examples below is independent of TOPCOM, you don't have to use it:

```
sage: PointConfiguration.set_engine('internal')
```

We start with a simple case of a non-smooth 2-dimensional cone:

```
sage: Cone([ (1,0), (1,2) ]).Hilbert_basis()
N(1, 0),
N(1, 2),
N(1, 1)
in 2-d lattice N
```

Two more complicated example from GAP/toric:

```
sage: Cone([[1,0],[3,4]]).dual().Hilbert_basis()
M(0,  1),
M(4, -3),
M(3, -2),
M(2, -1),
M(1,  0)
in 2-d lattice M
sage: cone = Cone([[1,2,3,4],[0,1,0,7],[3,1,0,2],[0,0,1,0]]).dual()
sage: cone.Hilbert_basis()            # long time
M(10,  -7,  0,  1),
M(-5,  21,  0, -3),
M( 0,  -2,  0,  1),
M(15, -63, 25,  9),
M( 2,  -3,  0,  1),
M( 1,  -4,  1,  1),
M(-1,   3,  0,  0),
M( 4,  -4,  0,  1),
M( 1,  -5,  2,  1),
M( 3,  -5,  1,  1),
M( 6,  -5,  0,  1),
M( 3, -13,  5,  2),
M( 2,  -6,  2,  1),
M( 5,  -6,  1,  1),
M( 0,   1,  0,  0),
M( 8,  -6,  0,  1),
M(-2,   8,  0, -1),
M(10, -42, 17,  6),
M( 7, -28, 11,  4),
M( 5, -21,  9,  3),
M( 6, -21,  8,  3),
M( 5, -14,  5,  2),
```

```
M( 2,  -7,  3,  1),
M( 4,  -7,  2,  1),
M( 7,  -7,  1,  1),
M( 0,   0,  1,  0),
M(-3,  14,  0, -2),
M(-1,   7,  0, -1),
M( 1,   0,  0,  0)
in 4-d lattice M
```

Not a strictly convex cone:

```
sage: wedge = Cone([ (1,0,0), (1,2,0), (0,0,1), (0,0,-1) ])
sage: wedge.semigroup_generators()
(N(1, 0, 0), N(1, 1, 0), N(1, 2, 0), N(0, 0, 1), N(0, 0, -1))
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 http://trac.sagemath.org/sage_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].

REFERENCES:

Hilbert_coefficients(*point*)

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.

OUTPUT:

A **Z**-vector of length `len(self.Hilbert_basis())` with nonnegative components.

Note: Since the Hilbert basis elements are not necessarily linearly independent, the expansion coefficients are not unique. However, this method will always return the same expansion coefficients when invoked with the same argument.

EXAMPLES:

```
sage: cone = Cone([(1,0),(0,1)])
sage: cone.rays()
N(1, 0),
N(0, 1)
in 2-d lattice N
sage: cone.Hilbert_coefficients([3,2])
(3, 2)
```

A more complicated example:

```
sage: N = ToricLattice(2)
sage: cone = Cone([N(1,0),N(1,2)])
sage: cone.Hilbert_basis()
N(1, 0),
N(1, 2),
N(1, 1)
in 2-d lattice N
sage: cone.Hilbert_coefficients( N(1,1) )
(0, 0, 1)
```

The cone need not be strictly convex:

```
sage: N = ToricLattice(3)
sage: cone = Cone([N(1,0,0),N(1,2,0),N(0,0,1),N(0,0,-1)])
sage: cone.Hilbert_basis()
N(1, 2,  0),
N(1, 0,  0),
N(0, 0,  1),
N(0, 0, -1),
N(1, 1,  0)
in 3-d lattice N
sage: cone.Hilbert_coefficients( N(1,1,3) )
(0, 0, 3, 0, 1)
```

adjacent()

Return faces adjacent to `self` in the ambient face lattice.

Two *distinct* faces F_1 and F_2 of the same face lattice are **adjacent** if all of the following conditions hold:

- F_1 and F_2 have the same dimension d;
- F_1 and F_2 share a facet of dimension $d-1$;
- F_1 and F_2 are facets of some face of dimension $d+1$, unless d is the dimension of the ambient structure.

OUTPUT:

- `tuple` of `cones`.

EXAMPLES:

```
sage: 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:

```
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:

```
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:

```
sage: fan = Fan(cones=[(0,1), (1,2), (3,)],
...             rays=[(1,0,0), (0,1,0), (-1,0,0), (0,0,1)])
sage: cone = fan.generating_cone(0)
sage: len(cone.adjacent())
0
```

Since now `cone` has smaller dimension than `fan`, it and its adjacent cones must be facets of a bigger one, but since `cone` in this example is generating, it is not contained in any other.

ambient()

Return the ambient structure of `self`.

OUTPUT:

- cone or fan containing `self` as a face.

EXAMPLES:

```
sage: cone = Cone([(1,2,3), (4,6,5), (9,8,7)])
sage: cone.ambient()
3-d cone in 3-d lattice N
sage: cone.ambient() is cone
True
sage: face = cone.faces(1)[0]
sage: face
1-d face of 3-d cone in 3-d lattice N
sage: face.ambient()
3-d cone in 3-d lattice N
sage: face.ambient() is cone
True
```

ambient_ray_indices()

Return indices of rays of the ambient structure generating `self`.

OUTPUT:

- increasing `tuple` of integers.

EXAMPLES:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: quadrant.ambient_ray_indices()
(0, 1)
sage: quadrant.facets()[1].ambient_ray_indices()
(1,)
```

cartesian_product(*other*, *lattice=None*)

Return the Cartesian product of `self` with `other`.

INPUT:

- `other` – a `cone`;
- `lattice` – (optional) the ambient lattice for the Cartesian product cone. By default, the direct sum of the ambient lattices of `self` and `other` is constructed.

OUTPUT:

- a `cone`.

EXAMPLES:

```
sage: c = Cone([(1,)])
sage: c.cartesian_product(c)
2-d cone in 2-d lattice N+N
sage: _.rays()
N+N(1, 0),
N+N(0, 1)
in 2-d lattice N+N
```

contains(**args*)

Check if a given point is contained in `self`.

INPUT:

- anything. An attempt will be made to convert all arguments into a single element of the ambient space of `self`. If it fails, `False` will be returned.

OUTPUT:

- `True` if the given point is contained in `self`, `False` otherwise.

EXAMPLES:

```
sage: c = Cone([(1,0), (0,1)])
sage: c.contains(c.lattice()(1,0))
True
sage: c.contains((1,0))
True
sage: c.contains((1,1))
True
sage: c.contains(1,1)
True
sage: c.contains((-1,0))
False
sage: c.contains(c.dual_lattice()(1,0)) #random output (warning)
False
sage: c.contains(c.dual_lattice()(1,0))
False
sage: c.contains(1)
False
sage: c.contains(1/2, sqrt(3))
True
sage: c.contains(-1/2, sqrt(3))
False
```

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(1,  -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)
in 2-d lattice M
sage: Cone([(1,0),(-1,0),(0,1)], lattice=N).dual().rays()  # half space
M(0, 1)
in 2-d lattice M
sage: Cone([(1,0),(0,1),(-1,-1)], lattice=N).dual().rays()  # whole space
Empty collection
in 2-d lattice M
```

embed(*cone*)

Return the cone equivalent to the given one, but sitting in `self` as a face.

You may need to use this method before calling methods of `cone` that depend on the ambient structure, such as `ambient_ray_indices()` or `facet_of()`. The cone returned by this method will have `self` as ambient. If `cone` does not represent a valid cone of `self`, `ValueError` exception is raised.

Note: This method is very quick if `self` is already the ambient structure of `cone`, so you can use without extra checks and performance hit even if `cone` is likely to sit in `self` but in principle may not.

INPUT:

- `cone` – a `cone`.

OUTPUT:

- a `cone`, equivalent to `cone` but sitting inside `self`.

EXAMPLES:

Let's take a 3-d cone on 4 rays:

```
sage: c = Cone([(1,0,1), (0,1,1), (-1,0,1), (0,-1,1)])
```

Then any ray generates a 1-d face of this cone, but if you construct such a face directly, it will not "sit" inside the cone:

```
sage: ray = Cone([(0,-1,1)])
sage: ray
1-d cone in 3-d lattice N
sage: ray.ambient_ray_indices()
(0,)
sage: ray.adjacent()
()
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:

```
sage: e_ray = c.embed(ray)
sage: e_ray
1-d face of 3-d cone in 3-d lattice N
sage: e_ray.rays()
N(0, -1, 1)
in 3-d lattice N
sage: e_ray is ray
False
sage: e_ray.is_equivalent(ray)
True
sage: e_ray.ambient_ray_indices()
(3,)
sage: e_ray.adjacent()
(1-d face of 3-d cone in 3-d lattice N,
 1-d face of 3-d cone in 3-d lattice N)
sage: e_ray.ambient()
3-d cone in 3-d lattice N
```

Not every cone can be embedded into a fixed ambient cone:

```
sage: c.embed(Cone([(0,0,1)]))
Traceback (most recent call last):
...
ValueError: 1-d cone in 3-d lattice N is not a face
of 3-d cone in 3-d lattice N!
sage: c.embed(Cone([(1,0,1), (-1,0,1)]))
Traceback (most recent call last):
...
ValueError: 2-d cone in 3-d lattice N is not a face
of 3-d cone in 3-d lattice N!
```

face_lattice()

Return the face lattice of `self`.

This lattice will have the origin as the bottom (we do not include the empty set as a face) and this cone itself as the top.

OUTPUT:

- `finite poset` of `cones`.

EXAMPLES:

Let's take a look at the face lattice of the first quadrant:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: L = quadrant.face_lattice()
sage: L
Finite poset containing 4 elements
```

To see all faces arranged by dimension, you can do this:

```
sage: for level in L.level_sets(): print level
[0-d face of 2-d cone in 2-d lattice N]
[1-d face of 2-d cone in 2-d lattice N,
 1-d face of 2-d cone in 2-d lattice N]
[2-d cone in 2-d lattice N]
```

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

```
sage: face = L.level_sets()[1][0]
sage: face.rays()
N(1, 0)
in 2-d lattice N
```

... or you can see the index of the ray of the original cone that corresponds to the above one:

```
sage: face.ambient_ray_indices()
(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: D.neighbors(face)
[2-d cone in 2-d lattice N,
 0-d face of 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 poset containing 16 elements
sage: supercone.face_lattice().top()
4-d cone in 4-d lattice N
sage: cone = supercone.facets()[0]
sage: cone
3-d face of 4-d cone in 4-d lattice N
sage: cone.face_lattice()
Finite poset containing 8 elements
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
```

TESTS:

```
sage: C1 = Cone([(0,1)])
sage: C2 = Cone([(0,1)])
sage: C1 == C2
True
sage: C1 is C2
False
```

C1 and C2 are equal, but not identical. We currently want them to have non identical face lattices, even if the faces themselves are equal (see #10998):

```
sage: C1.face_lattice() is C2.face_lattice()
False

sage: C1.facets()[0]
0-d face of 1-d cone in 2-d lattice N
sage: C2.facets()[0]
0-d face of 1-d cone in 2-d lattice N

sage: C1.facets()[0].ambient() is C1
True
sage: C2.facets()[0].ambient() is C1
False
sage: C2.facets()[0].ambient() is C2
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,
  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 orginal 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,),
 (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,),)
```

TESTS:

Now we check that "general" cones whose dimension is smaller than the dimension of the ambient space work as expected (see trac ticket #9188):

```
sage: c = Cone([(1,1,1,3),(1,-1,1,3),(-1,-1,1,3)])
sage: c.faces()
((0-d face of 3-d cone in 4-d lattice N,),
 (1-d face of 3-d cone in 4-d lattice N,
  1-d face of 3-d cone in 4-d lattice N,
  1-d face of 3-d cone in 4-d lattice N),
 (2-d face of 3-d cone in 4-d lattice N,
  2-d face of 3-d cone in 4-d lattice N,
  2-d face of 3-d cone in 4-d lattice N),
 (3-d cone in 4-d lattice N,))
```

We also ensure that a call to this function does not break `facets()` method (see trac ticket #9780):

```
sage: cone = toric_varieties.dP8().fan().generating_cone(0)
sage: cone
2-d cone of Rational polyhedral fan in 2-d lattice N
sage: for f in cone.facets(): print f.rays()
N(1, 1)
in 2-d lattice N
N(0, 1)
in 2-d lattice N
sage: len(cone.faces())
3
sage: for f in cone.facets(): print f.rays()
N(1, 1)
in 2-d lattice N
N(0, 1)
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 nomals will be elements of the dual lattice. If it is a general lattice (like `ZZ^n`) that does not have a `dual()` method, the facet normals

will be returned as integral vectors.

EXAMPLES:

```
sage: cone = Cone([(1,0), (-1,3)])
sage: cone.facet_normals()
M(0, 1),
M(3, 1)
in 2-d lattice M
```

Now let's look at a more complicated case:

```
sage: cone = Cone([(-2,-1,2), (4,1,0), (-4,-1,-5), (4,1,5)])
sage: cone.is_strictly_convex()
False
sage: cone.dim()
3
sage: cone.linear_subspace().dimension()
1
sage: lsg = (QQ^3)(cone.linear_subspace().gen(0)); lsg
(1, 1/4, 5/4)
sage: cone.facet_normals()
M(7, -18, -2),
M(1,  -4,  0)
in 3-d lattice M
sage: [lsg*normal for normal in cone.facet_normals()]
[0, 0]
```

A lattice that does not have a `dual()` method:

```
sage: Cone([(1,1),(0,1)], lattice=ZZ^2).facet_normals()
(-1, 1),
( 1, 0)
in Ambient free module of rank 2
over the principal ideal domain Integer Ring
```

We correctly handle the degenerate cases:

```
sage: N = ToricLattice(2)
sage: Cone([], lattice=N).facet_normals()  # empty cone
Empty collection
in 2-d lattice M
sage: Cone([(1,0)], lattice=N).facet_normals()  # ray in 2d
M(1, 0)
in 2-d lattice M
sage: Cone([(1,0),(-1,0)], lattice=N).facet_normals()  # line in 2d
Empty collection
in 2-d lattice M
sage: Cone([(1,0),(0,1)], lattice=N).facet_normals()  # strictly convex cone
M(0, 1),
M(1, 0)
in 2-d lattice M
sage: Cone([(1,0),(-1,0),(0,1)], lattice=N).facet_normals()  # half space
M(0, 1)
in 2-d lattice M
sage: Cone([(1,0),(0,1),(-1,-1)], lattice=N).facet_normals()  # whole space
Empty collection
in 2-d lattice M
```

facet_of()

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

OUTPUT:

- tuple of cones.

EXAMPLES:

```
sage: octant = Cone([(1,0,0), (0,1,0), (0,0,1)])
sage: octant.facet_of()
()
sage: one_face = octant.faces(1)[0]
sage: len(one_face.facet_of())
2
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)
```

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:

```
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:

```
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:

```
sage: cone1.intersection(cone1.dual())
Traceback (most recent call last):
...
ValueError: 2-d lattice N and 2-d lattice M
have different ambient lattices!
sage: cone1.intersection(Cone(cone1.dual().rays(), N)).rays()
N(3, 1),
N(0, 1)
in 2-d lattice N
```

is_equivalent(*other*)

Check if self is "mathematically" the same as other.

INPUT:

- other - cone.

OUTPUT:

- True if self and other define the same cones as sets of points in the same lattice, False otherwise.

There are three different equivalences between cones C_1 and C_2 in the same lattice:

1. They have the same generating rays in the same order. This is tested by `C1 == C2`.
2. They describe the same sets of points. This is tested by `C1.is_equivalent(C2)`.
3. They are in the same orbit of $GL(n, \mathbf{Z})$ (and, therefore, correspond to isomorphic affine toric varieties). This is tested by `C1.is_isomorphic(C2)`.

EXAMPLES:

```
sage: cone1 = Cone([(1,0), (-1, 3)])
sage: cone2 = Cone([(-1,3), (1, 0)])
sage: cone1.rays()
N( 1, 0),
N(-1, 3)
in 2-d lattice N
sage: cone2.rays()
N(-1, 3),
N( 1, 0)
in 2-d lattice N
sage: cone1 == cone2
False
sage: cone1.is_equivalent(cone2)
True
```

is_face_of(*cone*)

Check if `self` forms a face of another `cone`.

INPUT:

- `cone` – cone.

OUTPUT:

- `True` if `self` is a face of `cone`, `False` otherwise.

EXAMPLES:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: cone1 = Cone([(1,0)])
sage: cone2 = Cone([(1,2)])
sage: quadrant.is_face_of(quadrant)
True
sage: cone1.is_face_of(quadrant)
True
sage: cone2.is_face_of(quadrant)
False
```

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_isomorphic(*other*)

Check if `self` is in the same $GL(n, \mathbf{Z})$-orbit as `other`.

INPUT:

- `other` - cone.

OUTPUT:

- `True` if `self` and `other` are in the same $GL(n, \mathbf{Z})$-orbit, `False` otherwise.

There are three different equivalences between cones C_1 and C_2 in the same lattice:

1. They have the same generating rays in the same order. This is tested by `C1 == C2`.
2. They describe the same sets of points. This is tested by `C1.is_equivalent(C2)`.
3. They are in the same orbit of $GL(n, \mathbf{Z})$ (and, therefore, correspond to isomorphic affine toric varieties). This is tested by `C1.is_isomorphic(C2)`.

EXAMPLES:

```
sage: cone1 = Cone([(1,0), (0, 3)])
sage: m = matrix(ZZ, [(1, -5), (-1, 4)]) # a GL(2,ZZ)-matrix
sage: cone2 = Cone([m*r for r in cone1.rays()])
sage: cone1.is_isomorphic(cone2)
True

sage: cone1 = Cone([(1,0), (0, 3)])
sage: cone2 = Cone([(-1,3), (1, 0)])
sage: cone1.is_isomorphic(cone2)
False
```

TESTS:

```
sage: from sage.geometry.cone import classify_cone_2d
sage: classify_cone_2d(*cone1.rays())
(1, 0)
sage: classify_cone_2d(*cone2.rays())
(3, 2)
```

is_simplicial()

Check if `self` is simplicial.

A cone is called **simplicial** if primitive vectors along its generating rays form a part of a *rational* basis of the ambient space.

OUTPUT:

- `True` if `self` is simplicial, `False` otherwise.

EXAMPLES:

```
sage: cone1 = Cone([(1,0), (0, 3)])
sage: cone2 = Cone([(1,0), (0, 3), (-1,-1)])
sage: cone1.is_simplicial()
True
sage: cone2.is_simplicial()
False
```

is_smooth()

Check if `self` is smooth.

A cone is called **smooth** if primitive vectors along its generating rays form a part of an *integral* basis of the ambient space. Equivalently, they generate the whole lattice on the linear subspace spanned by the rays.

OUTPUT:

- `True` if `self` is smooth, `False` otherwise.

EXAMPLES:

```
sage: cone1 = Cone([(1,0), (0, 1)])
sage: cone2 = Cone([(1,0), (-1, 3)])
sage: cone1.is_smooth()
True
sage: cone2.is_smooth()
False
```

The following cones are the same up to a $SL(2, \mathbf{Z})$ coordinate transformation:

```
sage: Cone([(1,0,0), (2,1,-1)]).is_smooth()
True
sage: Cone([(1,0,0), (2,1,1)]).is_smooth()
True
sage: Cone([(1,0,0), (2,1,2)]).is_smooth()
True
```

is_strictly_convex()

Check if `self` is strictly convex.

A cone is called **strictly convex** if it does not contain any lines.

OUTPUT:

- `True` if `self` is strictly convex, `False` otherwise.

EXAMPLES:

```
sage: cone1 = Cone([(1,0), (0, 1)])
sage: cone2 = Cone([(1,0), (-1, 0)])
sage: cone1.is_strictly_convex()
True
sage: cone2.is_strictly_convex()
False
```

is_trivial()

Checks if the cone has no rays.

OUTPUT:

- `True` if the cone has no rays, `False` otherwise.

EXAMPLES:

```
sage: c0 = Cone([], lattice=ToricLattice(3))
sage: c0.is_trivial()
True
sage: c0.nrays()
0
```

lattice_polytope()

Return the lattice polytope associated to `self`.

The vertices of this polytope are primitive vectors along the generating rays of `self` and the origin, if `self` is strictly convex. In this case the origin is the last vertex, so the i-th ray of the cone always corresponds to the i-th vertex of the polytope.

See also `polyhedron()`.

OUTPUT:

- `LatticePolytope`.

EXAMPLES:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: lp = quadrant.lattice_polytope()
doctest:...: DeprecationWarning: lattice_polytope(...) is deprecated!
See http://trac.sagemath.org/16180 for details.
sage: lp
2-d lattice polytope in 2-d lattice N
sage: lp.vertices_pc()
N(1, 0),
N(0, 1),
N(0, 0)
in 2-d lattice N

sage: line = Cone([(1,0), (-1,0)])
sage: lp = line.lattice_polytope()
sage: lp
1-d lattice polytope in 2-d lattice N
sage: lp.vertices_pc()
N( 1, 0),
N(-1, 0)
in 2-d lattice N
```

line_set()

Return a set of lines generating the linear subspace of `self`.

OUTPUT:

- `frozenset` 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. See also `lines()`.

EXAMPLES:

```
sage: halfplane = Cone([(1,0), (0,1), (-1,0)])
sage: halfplane.line_set()
doctest:...: DeprecationWarning:
line_set(...) is deprecated, please use lines().set() instead!
See http://trac.sagemath.org/12544 for details.
frozenset([N(1, 0)])
sage: fullplane = Cone([(1,0), (0,1), (-1,-1)])
sage: fullplane.line_set()
frozenset([N(0, 1), N(1, 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]
```

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
```

orthogonal_sublattice(**args*, ***kwds*)

The sublattice (in the dual lattice) orthogonal to the sublattice spanned by the cone.

Let $M =$ self.dual_lattice() be the lattice dual to the ambient lattice of the given cone σ. Then, in the notation of [Fulton], this method returns the sublattice

$$M(\sigma) \stackrel{\text{def}}{=} \sigma^\perp \cap M \subset M$$

INPUT:

- either nothing or something that can be turned into an element of this lattice.

OUTPUT:

- if no arguments were given, a toric sublattice, otherwise the corresponding element of it.

EXAMPLES:

```
sage: c = Cone([(1,1,1), (1,-1,1), (-1,-1,1), (-1,1,1)])
sage: c.orthogonal_sublattice()
Sublattice <>
sage: c12 = Cone([(1,1,1), (1,-1,1)])
sage: c12.sublattice()
Sublattice <N(1, -1, 1), N(0, 1, 0)>
sage: c12.orthogonal_sublattice()
Sublattice <M(1, 0, -1)>
```

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()
```

polyhedron()

Return the polyhedron associated to self.

Mathematically this polyhedron is the same as self.

OUTPUT:

- Polyhedron_base.

EXAMPLES:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: quadrant.polyhedron()
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull
of 1 vertex and 2 rays
sage: line = Cone([(1,0), (-1,0)])
sage: line.polyhedron()
A 1-dimensional polyhedron in ZZ^2 defined as the convex hull
of 1 vertex and 1 line
```

Here is an example of a trivial cone (see trac ticket #10237):

```
sage: origin = Cone([], lattice=ZZ^2)
sage: origin.polyhedron()
A 0-dimensional polyhedron in ZZ^2 defined as the convex hull
of 1 vertex
```

relative_interior_contains(**args*)

Check if a given point is contained in the relative interior of `self`.

For a full-dimensional cone the relative interior is simply the interior, so this method will do the same check as `interior_contains()`. For a strictly lower-dimensional cone, the relative interior is the cone without its facets.

INPUT:

- anything. An attempt will be made to convert all arguments into a single element of the ambient space of `self`. If it fails, `False` will be returned.

OUTPUT:

- `True` if the given point is contained in the relative interior of `self`, `False` otherwise.

EXAMPLES:

```
sage: c = Cone([(1,0,0), (0,1,0)])
sage: c.contains((1,1,0))
True
sage: c.relative_interior_contains((1,1,0))
True
sage: c.interior_contains((1,1,0))
False
sage: c.contains((1,0,0))
True
sage: c.relative_interior_contains((1,0,0))
False
sage: c.interior_contains((1,0,0))
False
```

relative_orthogonal_quotient(*supercone*)

The quotient of the dual spanned lattice by the dual of the supercone's spanned lattice.

In the notation of [Fulton], if `supercone` = ρ > σ = `self` is a cone that contains σ as a face, then $M(\rho)$ = `supercone.orthogonal_sublattice()` is a saturated sublattice of $M(\sigma)$ = `self.orthogonal_sublattice()`. This method returns the quotient lattice. The lifts of the quotient generators are $\dim(\rho) - \dim(\sigma)$ linearly independent M-lattice lattice points that, together with $M(\rho)$, generate $M(\sigma)$.

OUTPUT:

- toric lattice quotient.

If we call the output `Mrho`, then

- `Mrho.cover() == self.orthogonal_sublattice()`, and
- `Mrho.relations() == supercone.orthogonal_sublattice()`.

Note:

- $M(\sigma)/M(\rho)$ has no torsion since the sublattice $M(\rho)$ is saturated.
- In the codimension one case, (a lift of) the generator of $M(\sigma)/M(\rho)$ is chosen to be positive on σ.

EXAMPLES:

```
sage: 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 [Fulton], let N be the ambient lattice and N_σ the sublattice spanned by the given cone σ. If $\rho < \sigma$ is a subcone, then N_ρ = `rho.sublattice()` is a saturated sublattice of N_σ = `self.sublattice()`. This method returns the quotient lattice. The lifts of the quotient generators are $\dim(\sigma) - \dim(\rho)$ linearly independent primitive lattice lattice points that, together with N_ρ, generate N_σ.

OUTPUT:

- toric lattice quotient.

Note:

- The quotient N_σ/N_ρ of spanned sublattices has no torsion since the sublattice N_ρ is saturated.
- In the codimension one case, the generator of N_σ/N_ρ is chosen to be in the same direction as the image σ/N_ρ

EXAMPLES:

```
sage: sigma = Cone([(1,1,1,3),(1,-1,1,3),(-1,-1,1,3),(-1,1,1,3)])
sage: rho   = Cone([(-1, -1, 1, 3), (-1, 1, 1, 3)])
sage: sigma.sublattice()
Sublattice <N(-1, -1, 1, 3), N(1, 0, 0, 0), N(1, 1, 0, 0)>
sage: rho.sublattice()
Sublattice <N(-1, 1, 1, 3), N(0, -1, 0, 0)>
sage: sigma.relative_quotient(rho)
1-d lattice, quotient
of Sublattice <N(-1, -1, 1, 3), N(1, 0, 0, 0), N(1, 1, 0, 0)>
by Sublattice <N(1, 0, -1, -3), N(0, 1, 0, 0)>
sage: sigma.relative_quotient(rho).gens()
(N[1, 1, 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, 1, 1), N(1, 2, 3), N(0, 1, 1)>
sage: N_rho = rho.sublattice()
sage: N_rho
Sublattice <N(1, -1, 1), N(1, 2, 3)>
sage: sigma.relative_quotient(rho).gens()
(N[0, 1, 1],)
sage: N = rho.lattice()
sage: N_sigma == N.span(N_rho.gens() + tuple(q.lift()
...               for q in sigma.relative_quotient(rho).gens()))
True
```

Sign choice in the codimension one case:

```
sage: sigma1 = Cone([(1, 2, 3), (1, -1, 1), (-1, 1, 1), (-1, -1, 1)])  # 3d
sage: sigma2 = Cone([(1, 1, -1), (1, 2, 3), (1, -1, 1), (1, -1, -1)])  # 3d
sage: rho = sigma1.intersection(sigma2)
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, 1, 1), N(1, 2, 3), N(0, 1, 1)>
by Sublattice <N(1, 2, 3), N(0, 3, 2)>
sage: sigma1.relative_quotient(rho).gens()
(N[0, 1, 1],)
sage: sigma2.relative_quotient(rho).gens()
(N[-1, 0, -2],)
```

semigroup_generators()

Return generators for the semigroup of lattice points of `self`.

OUTPUT:

- a PointCollection of lattice points generating the semigroup of lattice points contained in self.

Note: No attempt is made to return a minimal set of generators, see Hilbert_basis() for that.

EXAMPLES:

The following command ensures that the output ordering in the examples below is independent of TOPCOM, you don't have to use it:

```
sage: PointConfiguration.set_engine('internal')
```

We start with a simple case of a non-smooth 2-dimensional cone:

```
sage: Cone([ (1,0), (1,2) ]).semigroup_generators()
N(1, 1),
N(1, 0),
N(1, 2)
in 2-d lattice N
```

A non-simplicial cone works, too:

```
sage: cone = Cone([(3,0,-1), (1,-1,0), (0,1,0), (0,0,1)])
sage: cone.semigroup_generators()
(N(1, 0, 0), N(0, 0, 1), N(0, 1, 0), N(3, 0, -1), N(1, -1, 0))
```

GAP's toric package thinks this is challenging:

```
sage: cone = Cone([[1,2,3,4],[0,1,0,7],[3,1,0,2],[0,0,1,0]]).dual()
sage: len( cone.semigroup_generators() )
2806
```

The cone need not be strictly convex:

```
sage: halfplane = Cone([(1,0),(2,1),(-1,0)])
sage: halfplane.semigroup_generators()
(N(0, 1), N(1, 0), N(-1, 0))
sage: line = Cone([(1,1,1),(-1,-1,-1)])
sage: 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: wedge.semigroup_generators()
(N(1, 0, 0), N(1, 1, 0), N(1, 2, 0), N(0, 0, 1), N(0, 0, -1))
```

Nor does it have to be full-dimensional (see http://trac.sagemath.org/sage_trac/ticket/11312):

```
sage: Cone([(1,1,0), (-1,1,0)]).semigroup_generators()
N( 0, 1, 0),
N( 1, 1, 0),
N(-1, 1, 0)
in 3-d lattice N
```

Neither full-dimensional nor simplicial:

```
sage: A = matrix([(1, 3, 0), (-1, 0, 1), (1, 1, -2), (15, -2, 0)])
sage: A.elementary_divisors()
[1, 1, 1, 0]
sage: cone3d = Cone([(3,0,-1), (1,-1,0), (0,1,0), (0,0,1)])
sage: rays = [ A*vector(v) for v in cone3d.rays() ]
sage: gens = Cone(rays).semigroup_generators(); gens
```

```
(N(1, -1, 1, 15), N(0, 1, -2, 0), N(-2, -1, 0, 17), 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
```

TESTS:

```
sage: len(Cone(identity_matrix(10).rows()).semigroup_generators())
10

sage: trivial_cone = Cone([], lattice=ToricLattice(3))
sage: trivial_cone.semigroup_generators()
Empty collection
in 3-d lattice N
```

ALGORITHM:

If the cone is not simplicial, it is first triangulated. Each simplicial subcone has the integral points of the spaned parallelotope as generators. This is the first step of the primal Normaliz algorithm, see [Normaliz]. For each simplicial cone (of dimension d), the integral points of the open parallelotope

$$par\langle x_1, \ldots, x_d\rangle = \mathbf{Z}^n \cap \{q_1x_1 + \cdots + q_dx_d : \ 0 \leq q_i < 1\}$$

are then computed [BrunsKoch].

Finally, the the union of the generators of all simplicial subcones is returned.

REFERENCES:

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
```

sublattice(**args*, ***kwds*)

The sublattice spanned by the cone.

Let σ be the given cone and $N =$ `self.lattice()` the ambient lattice. Then, in the notation of [Fulton], this method returns the sublattice

$$N_\sigma \stackrel{\text{def}}{=} span(N \cap \sigma)$$

INPUT:

•either nothing or something that can be turned into an element of this lattice.

OUTPUT:

•if no arguments were given, a `toric sublattice`, otherwise the corresponding element of it.

Note:

•The sublattice spanned by the cone is the saturation of the sublattice generated by the rays of the cone.

•See `sage.geometry.cone.IntegralRayCollection.ray_basis()` if you only need a **Q**-basis.

•The returned lattice points are usually not rays of the cone. In fact, for a non-smooth cone the rays do not generate the sublattice N_σ, but only a finite index sublattice.

EXAMPLES:

```
sage: cone = Cone([(1, 1, 1), (1, -1, 1), (-1, -1, 1), (-1, 1, 1)])
sage: cone.rays().basis()
N( 1,  1, 1),
N( 1, -1, 1),
N(-1, -1, 1)
in 3-d lattice N
sage: cone.rays().basis().matrix().det()
-4
sage: cone.sublattice()
Sublattice <N(-1, -1, 1), N(1, 0, 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(1, 2, 3), N(4, -5, 1)>
sage: c.sublattice(5, -3, 4)
N(5, -3, 4)
sage: c.sublattice(1, 0, 0)
Traceback (most recent call last):
...
TypeError: element (= [1, 0, 0]) is not in free module
```

sublattice_complement(**args*, ***kwds*)

A complement of the sublattice spanned by the cone.

In other words, `sublattice()` and `sublattice_complement()` together form a **Z**-basis for the ambient `lattice()`.

In the notation of [Fulton], let σ be the given cone and $N =$ `self.lattice()` the ambient lattice. Then this method returns

$$N(\sigma) \stackrel{\text{def}}{=} N/N_\sigma$$

lifted (non-canonically) to a sublattice of N.

INPUT:

•either nothing or something that can be turned into an element of this lattice.

OUTPUT:

•if no arguments were given, a `toric sublattice`, otherwise the corresponding element of it.

EXAMPLES:

```
sage: C2_Z2 = Cone([(1,0),(1,2)])      # C^2/Z_2
sage: c1, c2 = C2_Z2.facets()
sage: c2.sublattice()
Sublattice <N(1, 2)>
sage: c2.sublattice_complement()
Sublattice <N(0, 1)>
```

A more complicated example:

```
sage: c = Cone([(1,2,3), (4,-5,1)])
sage: c.sublattice()
Sublattice <N(1, 2, 3), N(4, -5, 1)>
sage: c.sublattice_complement()
Sublattice <N(0, -6, -5)>
sage: m = matrix( c.sublattice().gens() + c.sublattice_complement().gens() )
sage: m
[ 1  2  3]
[ 4 -5  1]
[ 0 -6 -5]
sage: m.det()
-1
```

sublattice_quotient(**args*, ***kwds*)

The quotient of the ambient lattice by the sublattice spanned by the cone.

INPUT:

•either nothing or something that can be turned into an element of this lattice.

OUTPUT:

•if no arguments were given, a `quotient of a toric lattice`, otherwise the corresponding element of it.

EXAMPLES:

```
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*)

Bases: sage.structure.sage_object.SageObject, _abcoll.Hashable,

_abcoll.Iterable

Create a collection of integral rays.

Warning: No correctness check or normalization is performed on the input data. This class is designed for internal operations and you probably should not use it directly.

This is a base class for convex rational polyhedral cones and fans.

Ray collections are immutable, but they cache most of the returned values.

INPUT:

- rays – list of immutable vectors in lattice;
- lattice – ToricLattice, $\mathbf{Z}^n$, or any other object that behaves like these. If None, it will be determined as parent() of the first ray. Of course, this cannot be done if there are no rays, so in this case you must give an appropriate lattice directly. Note that None is *not* the default value - you always *must* give this argument explicitly, even if it is None.

OUTPUT:

- collection of given integral rays.

cartesian_product(*other*, *lattice=None*)

Return the Cartesian product of self with other.

INPUT:

- other – an IntegralRayCollection;
- lattice – (optional) the ambient lattice for the result. By default, the direct sum of the ambient lattices of self and other is constructed.

OUTPUT:

- an IntegralRayCollection.

By the Cartesian product of ray collections $(r_0, \ldots, r_{n-1})$ and $(s_0, \ldots, s_{m-1})$ we understand the ray collection of the form $((r_0, 0), \ldots, (r_{n-1}, 0), (0, s_0), \ldots, (0, s_{m-1}))$, which is suitable for Cartesian products of cones and fans. The ray order is guaranteed to be as described.

EXAMPLES:

```
sage: c = Cone([(1,)])
sage: c.cartesian_product(c)    # indirect doctest
2-d cone in 2-d lattice N+N
sage: _.rays()
N+N(1, 0),
N+N(0, 1)
in 2-d lattice N+N
```

dim()

Return the dimension of the subspace spanned by rays of self.

OUTPUT:

- integer.

EXAMPLES:

```
sage: c = Cone([(1,0)])
sage: c.lattice_dim()
2
```

```
sage: c.dim()
1
```

dual_lattice()
Return the dual of the ambient lattice of `self`.

OUTPUT:

- lattice. If possible (that is, if `lattice()` has a `dual()` method), the dual lattice is returned. Otherwise, $\mathbf{Z}^n$ is returned, where n is the dimension of `lattice()`.

EXAMPLES:

```
sage: c = Cone([(1,0)])
sage: c.dual_lattice()
2-d lattice M
sage: Cone([], ZZ^3).dual_lattice()
Ambient free module of rank 3
over the principal ideal domain Integer Ring
```

lattice()
Return the ambient lattice of `self`.

OUTPUT:

- lattice.

EXAMPLES:

```
sage: c = Cone([(1,0)])
sage: c.lattice()
2-d lattice N
sage: Cone([], ZZ^3).lattice()
Ambient free module of rank 3
over the principal ideal domain Integer Ring
```

lattice_dim()
Return the dimension of the ambient lattice of `self`.

OUTPUT:

- integer.

EXAMPLES:

```
sage: c = Cone([(1,0)])
sage: c.lattice_dim()
2
sage: c.dim()
1
```

nrays()
Return the number of rays of `self`.

OUTPUT:

- integer.

EXAMPLES:

```
sage: c = Cone([(1,0), (0,1)])
sage: c.nrays()
2
```

plot(***options*)
Plot `self`.

INPUT:

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

OUTPUT:

•a plot.

EXAMPLES:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: quadrant.plot()
```

ray(*n*)
Return the `n`-th ray of `self`.

INPUT:

•`n` – integer, an index of a ray of `self`. Enumeration of rays starts with zero.

OUTPUT:

•ray, an element of the lattice of `self`.

EXAMPLES:

```
sage: c = Cone([(1,0), (0,1)])
sage: c.ray(0)
N(1, 0)
```

rays(**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:

```
sage: c = Cone([(1,0), (0,1), (-1, 0)])
sage: c.rays()
N( 0, 1),
N( 1, 0),
N(-1, 0)
in 2-d lattice N
sage: c.rays([0, 2])
N( 0, 1),
N(-1, 0)
in 2-d lattice N
```

You can also give ray indices directly, without packing them into a list:

```
sage: c.rays(0, 2)
N( 0, 1),
N(-1, 0)
in 2-d lattice N
```

sage.geometry.cone.**classify_cone_2d**(*ray0*, *ray1*, *check=True*)

Return (d, k) classifying the lattice cone spanned by the two rays.

INPUT:

- `ray0`, `ray1` – two primitive integer vectors. The generators of the two rays generating the two-dimensional cone.
- `check` – boolean (default: `True`). Whether to check the input rays for consistency.

OUTPUT:

A pair (d, k) of integers classifying the cone up to $GL(2, \mathbf{Z})$ equivalence. See Proposition 10.1.1 of [CLS] for the definition. We return the unique (d, k) with minmial k, see Proposition 10.1.3 of [CLS].

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)
```

TESTS:

Check using the connection between the Hilbert basis of the cone spanned by the two rays (in arbitrary dimension) and the Hirzebruch-Jung continued fraction expansion, see Chapter 10 of [CLS]

```
sage: from sage.geometry.cone import normalize_rays
sage: for i in range(10):
...       ray0 = random_vector(ZZ, 3)
...       ray1 = random_vector(ZZ, 3)
...       if ray0.is_zero() or ray1.is_zero(): continue
...       ray0, ray1 = normalize_rays([ray0, ray1], ZZ^3)
...       d, k = classify_cone_2d(ray0, ray1, check=True)
...       assert (d,k) == classify_cone_2d(ray1, ray0)
...       if d == 0: continue
...       frac = Hirzebruch_Jung_continued_fraction_list(k/d)
...       if len(frac)>100: continue   # avoid expensive computation
...       hilb = Cone([ray0, ray1]).Hilbert_basis()
...       assert len(hilb) == len(frac) + 1
```

sage.geometry.cone.**is_Cone**(*x*)

Check if `x` is a cone.

INPUT:

- `x` – anything.

OUTPUT:

- `True` if `x` is a cone and `False` otherwise.

EXAMPLES:

```
sage: from sage.geometry.cone import is_Cone
sage: is_Cone(1)
False
sage: quadrant = Cone([(1,0), (0,1)])
sage: quadrant
2-d cone in 2-d lattice N
sage: is_Cone(quadrant)
True
```

sage.geometry.cone.**normalize_rays**(*rays*, *lattice*)

Normalize a list of rational rays: make them primitive and immutable.

INPUT:

- `rays` – list of rays which can be converted to the rational extension of `lattice`;
- `lattice` – `ToricLattice`, $\mathbf{Z}^n$, or any other object that behaves like these. If `None`, an attempt will be made to determine an appropriate toric lattice automatically.

OUTPUT:

- list of immutable primitive vectors of the `lattice` in the same directions as original `rays`.

EXAMPLES:

```
sage: from sage.geometry.cone import normalize_rays
sage: normalize_rays([(0, 1), (0, 2), (3, 2), (5/7, 10/3)], None)
[N(0, 1), N(0, 1), N(3, 2), N(3, 14)]
sage: L = ToricLattice(2, "L")
sage: normalize_rays([(0, 1), (0, 2), (3, 2), (5/7, 10/3)], L.dual())
[L*(0, 1), L*(0, 1), L*(3, 2), L*(3, 14)]
sage: ray_in_L = L(0,1)
sage: normalize_rays([ray_in_L, (0, 2), (3, 2), (5/7, 10/3)], None)
[L(0, 1), L(0, 1), L(3, 2), L(3, 14)]
sage: normalize_rays([(0, 1), (0, 2), (3, 2), (5/7, 10/3)], ZZ^2)
[(0, 1), (0, 1), (3, 2), (3, 14)]
sage: normalize_rays([(0, 1), (0, 2), (3, 2), (5/7, 10/3)], ZZ^3)
Traceback (most recent call last):
...
TypeError: cannot convert (0, 1) to
Vector space of dimension 3 over Rational Field!
sage: normalize_rays([], ZZ^3)
[]
```

CHAPTER

THREE

RATIONAL POLYHEDRAL FANS

This module was designed as a part of the framework for toric varieties (`variety`, `fano_variety`). While the emphasis is on complete full-dimensional fans, arbitrary fans are supported. Work with distinct lattices. The default lattice is `ToricLattice` N of the appropriate dimension. The only case when you must specify lattice explicitly is creation of a 0-dimensional fan, where dimension of the ambient space cannot be guessed.

A **rational polyhedral fan** is a *finite* collection of *strictly* convex rational polyhedral cones, such that the intersection of any two cones of the fan is a face of each of them and each face of each cone is also a cone of the fan.

AUTHORS:

- Andrey Novoseltsev (2010-05-15): initial version.
- Andrey Novoseltsev (2010-06-17): substantial improvement during review by Volker Braun.

EXAMPLES:

Use `Fan()` to construct fans "explicitly":

```
sage: fan = Fan(cones=[(0,1), (1,2)],
...             rays=[(1,0), (0,1), (-1,0)])
sage: fan
Rational polyhedral fan in 2-d lattice N
```

In addition to giving such lists of cones and rays you can also create cones first using `Cone()` and then combine them into a fan. See the documentation of `Fan()` for details.

In 2 dimensions there is a unique maximal fan determined by rays, and you can use `Fan2d()` to construct it:

```
sage: fan2d = Fan2d(rays=[(1,0), (0,1), (-1,0)])
sage: fan2d.is_equivalent(fan)
True
```

But keep in mind that in higher dimensions the cone data is essential and cannot be omitted. Instead of building a fan from scratch, for this tutorial we will use an easy way to get two fans assosiated to `lattice polytopes`: `FaceFan()` and `NormalFan()`:

```
sage: fan1 = FaceFan(lattice_polytope.cross_polytope(3))
sage: fan2 = NormalFan(lattice_polytope.cross_polytope(3))
```

Given such "automatic" fans, you may wonder what are their rays and cones:

```
sage: fan1.rays()
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
```

```
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1)
in 3-d lattice M
sage: fan1.generating_cones()
(3-d cone of Rational polyhedral fan in 3-d lattice M,
 3-d cone of Rational polyhedral fan in 3-d lattice M,
 3-d cone of Rational polyhedral fan in 3-d lattice M,
 3-d cone of Rational polyhedral fan in 3-d lattice M,
 3-d cone of Rational polyhedral fan in 3-d lattice M,
 3-d cone of Rational polyhedral fan in 3-d lattice M,
 3-d cone of Rational polyhedral fan in 3-d lattice M,
 3-d cone of Rational polyhedral fan in 3-d lattice M)
```

The last output is not very illuminating. Let's try to improve it:

```
sage: for cone in fan1: print cone.rays()
M(1, 0,  0),
M(0, 1,  0),
M(0, 0, -1)
in 3-d lattice M
M( 0, 1,  0),
M(-1, 0,  0),
M( 0, 0, -1)
in 3-d lattice M
M(1,  0,  0),
M(0, -1,  0),
M(0,  0, -1)
in 3-d lattice M
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1)
in 3-d lattice M
M(1, 0, 0),
M(0, 1, 0),
M(0, 0, 1)
in 3-d lattice M
M( 0, 1, 0),
M( 0, 0, 1),
M(-1, 0, 0)
in 3-d lattice M
M(1,  0, 0),
M(0,  0, 1),
M(0, -1, 0)
in 3-d lattice M
M( 0,  0, 1),
M(-1,  0, 0),
M( 0, -1, 0)
in 3-d lattice M
```

You can also do

```
sage: for cone in fan1: print cone.ambient_ray_indices()
(0, 1, 5)
(1, 3, 5)
(0, 4, 5)
(3, 4, 5)
(0, 1, 2)
```

```
(1, 2, 3)
(0, 2, 4)
(2, 3, 4)
```

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 don't 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 poset containing 28 elements
sage: L.bottom()
0-d cone of Rational polyhedral fan in 3-d lattice M
sage: L.top()
Rational polyhedral fan in 3-d lattice M
sage: cone = L.level_sets()[2][0]
sage: cone
2-d cone of Rational polyhedral fan in 3-d lattice M
sage: sorted(L.hasse_diagram().neighbors(cone))
[1-d cone of Rational polyhedral fan in 3-d lattice M,
 1-d cone of Rational polyhedral fan in 3-d lattice M,
 3-d cone of Rational polyhedral fan in 3-d lattice M,
 3-d cone of Rational polyhedral fan in 3-d lattice M]
```

You can check how "good" a fan is:

```
sage: fan1.is_complete()
True
sage: fan1.is_simplicial()
True
sage: fan1.is_smooth()
True
```

The face fan of the octahedron is really good! Time to remember that we have also constructed its normal fan:

```
sage: fan2.is_complete()
True
sage: fan2.is_simplicial()
False
sage: fan2.is_smooth()
False
```

This one does have some "problems," but we can fix them:

```
sage: fan3 = fan2.make_simplicial()
sage: fan3.is_simplicial()
True
sage: fan3.is_smooth()
False
```

Note that we had to save the result of `make_simplicial()` in a new fan. Fans in Sage are immutable, so any operation that does change them constructs a new fan.

We can also make `fan3` smooth, but it will take a bit more work:

```
sage: cube = lattice_polytope.cross_polytope(3).polar()
sage: sk = cube.skeleton_points(2)
sage: rays = [cube.point(p) for p in sk]
sage: fan4 = fan3.subdivide(new_rays=rays)
sage: fan4.is_smooth()
True
```

Let's see how "different" are `fan2` and `fan4`:

```
sage: fan2.ngenerating_cones()
6
sage: fan2.nrays()
8
sage: fan4.ngenerating_cones()
48
sage: fan4.nrays()
26
```

Smoothness does not come for free!

Please take a look at the rest of the available functions below and their complete descriptions. If you need any features that are missing, feel free to suggest them. (Or implement them on your own and submit a patch to Sage for inclusion!)

class `sage.geometry.fan.`**`Cone_of_fan`**(*ambient*, *ambient_ray_indices*)

Bases: `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)
sage: cone.star_generator_indices()
(0,)
```

TESTS:

A mistake in this function used to cause the problem reported in trac ticket #9782. We check that now everything is working smoothly:

```
sage: f = Fan([(0, 2, 4),
...            (0, 4, 5),
...            (0, 3, 5),
...            (0, 1, 3),
...            (0, 1, 2),
...            (2, 4, 6),
...            (4, 5, 6),
...            (3, 5, 6),
...            (1, 3, 6),
...            (1, 2, 6)],
...           [(-1, 0, 0),
...            (0, -1, 0),
...            (0, 0, -1),
...            (0, 0, 1),
...            (0, 1, 2),
...            (0, 1, 3),
...            (1, 0, 4)])
sage: f.is_complete()
True
sage: X = ToricVariety(f)
sage: X.fan().is_complete()
True
```

star_generators()

Return indices of generating cones of the "ambient fan" containing `self`.

OUTPUT:

- increasing `tuple` of integers.

EXAMPLES:

```
sage: P1xP1 = toric_varieties.P1xP1()
sage: cone = P1xP1.fan().generating_cone(0)
sage: cone.star_generators()
(2-d cone of Rational polyhedral fan in 2-d lattice N,)
```

sage.geometry.fan.**FaceFan**(*polytope*, *lattice=None*)

Construct the face fan of the given rational `polytope`.

INPUT:

- `polytope` – a `polytope` over $\mathbf{Q}$ or a `lattice polytope`. A (not necessarily full-dimensional) polytope contaning the origin in its `relative interior`.
- `lattice` – `ToricLattice`, $\mathbf{Z}^n$, or any other object that behaves like these. If not specified, an attempt will be made to determine an appropriate toric lattice automatically.

OUTPUT:

- `rational polyhedral fan`.

See also `NormalFan()`.

EXAMPLES:

Let's construct the fan corresponding to the product of two projective lines:

```
sage: diamond = lattice_polytope.cross_polytope(2)
sage: P1xP1 = FaceFan(diamond)
sage: P1xP1.rays()
M( 1,  0),
M( 0,  1),
M(-1,  0),
M( 0, -1)
in 2-d lattice M
sage: for cone in P1xP1: print cone.rays()
M(1,  0),
M(0, -1)
in 2-d lattice M
M(-1,  0),
M( 0, -1)
in 2-d lattice M
M(1, 0),
M(0, 1)
in 2-d lattice M
M( 0, 1),
M(-1, 0)
in 2-d lattice M
```

TESTS:

```
sage: cuboctahed = polytopes.cuboctahedron()
sage: FaceFan(cuboctahed)
Rational polyhedral fan in 3-d lattice M
sage: cuboctahed.is_lattice_polytope(), cuboctahed.dilation(2).is_lattice_polytope()
(False, True)
sage: fan1 = FaceFan(cuboctahed)
sage: fan2 = FaceFan(cuboctahed.dilation(2).lattice_polytope())
sage: fan1.is_equivalent(fan2)
True

sage: ray = Polyhedron(vertices=[(-1,-1)], rays=[(1,1)])
sage: FaceFan(ray)
```

```
Traceback (most recent call last):
...
ValueError: face fans are defined only for
polytopes containing the origin as an interior point!

sage: interval_in_QQ2 = Polyhedron([ (0,-1), (0,+1) ])
sage: FaceFan(interval_in_QQ2).generating_cones()
(1-d cone of Rational polyhedral fan in 2-d lattice M,
 1-d cone of Rational polyhedral fan in 2-d lattice M)

sage: FaceFan(Polyhedron([(-1,0), (1,0), (0,1)])) # origin on facet
Traceback (most recent call last):
...
ValueError: face fans are defined only for
polytopes containing the origin as an interior point!
```

sage.geometry.fan.**Fan**(*cones*, *rays=None*, *lattice=None*, *check=True*, *normalize=True*, *is_complete=None*, *virtual_rays=None*, *discard_faces=False*)

Construct a rational polyhedral fan.

Note: Approximate time to construct a fan consisting of n cones is $n^2/5$ seconds. That is half an hour for 100 cones. This time can be significantly reduced in the future, but it is still likely to be $\sim n^2$ (with, say, $/500$ instead of $/5$). If you know that your input does form a valid fan, use `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` option is specified;
- `rays` – list of rays given as list or vectors convertible to the rational extension of `lattice`. If `cones` are given by `Cone` objects `rays` may be determined automatically. You still may give them explicitly to ensure a particular order of rays in the fan. In this case you must list all rays that appear in `cones`. You can give "extra" ones if it is convenient (e.g. if you have a big list of rays for several fans), but all "extra" rays will be discarded;
- `lattice` – `ToricLattice`, $\mathbf{Z}^n$, or any other object that behaves like these. If not specified, an attempt will be made to determine an appropriate toric lattice automatically;
- `check` – by default the input data will be checked for correctness (e.g. that intersection of any two given cones is a face of each). 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. If you provide "extra" ones and leave check=True (default), an exception will be raised. If you provide "extra" cones and set 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).

OUTPUT:

- a fan.

See Also:

In 2 dimensions you can cyclically order the rays. Hence the rays determine a unique maximal fan without having to specify the cones, and you can use Fan2d() to construct this fan from just the rays.

EXAMPLES:

Let's construct a fan corresponding to the projective plane in several ways:

```
sage: cone1 = Cone([(1,0), (0,1)])
sage: cone2 = Cone([(0,1), (-1,-1)])
sage: cone3 = Cone([(-1,-1), (1,0)])
sage: P2 = Fan([cone1, cone2, cone2])
Traceback (most recent call last):
...
ValueError: you have provided 3 cones, but only 2 of them are maximal!
Use discard_faces=True if you indeed need to construct a fan from
these cones.
```

Oops! There was a typo and cone2 was listed twice as a generating cone of the fan. If it was intentional (e.g. the list of cones was generated automatically and it is possible that it contains repetitions or faces of other cones), use discard_faces=True option:

```
sage: P2 = Fan([cone1, cone2, cone2], discard_faces=True)
sage: P2.ngenerating_cones()
2
```

However, in this case it was definitely a typo, since the fan of $\mathbb{P}^2$ has 3 maximal cones:

```
sage: P2 = Fan([cone1, cone2, cone3])
sage: P2.ngenerating_cones()
3
```

Looks better. An alternative way is

```
sage: rays = [(1,0), (0,1), (-1,-1)]
sage: cones = [(0,1), (1,2), (2,0)]
sage: P2a = Fan(cones, rays)
sage: P2a.ngenerating_cones()
3
sage: P2 == P2a
False
```

That may seem wrong, but it is not:

```
sage: P2.is_equivalent(P2a)
True
```

See is_equivalent() for details.

Yet another way to construct this fan is

```
sage: P2b = Fan(cones, rays, check=False)
sage: P2b.ngenerating_cones()
3
sage: P2a == P2b
True
```

If you try the above examples, you are likely to notice the difference in speed, so when you are sure that everything is correct, it is a good idea to use `check=False` option. On the other hand, it is usually **NOT** a good idea to use `normalize=False` option:

```
sage: P2c = Fan(cones, rays, check=False, normalize=False)
Traceback (most recent call last):
...
AttributeError: 'tuple' object has no attribute 'parent'
```

Yet another way is to use functions `FaceFan()` and `NormalFan()` to construct fans from `lattice polytopes`.

We have not yet used `lattice` argument, since if was determined automatically:

```
sage: P2.lattice()
2-d lattice N
sage: P2b.lattice()
2-d lattice N
```

However, it is necessary to specify it explicitly if you want to construct a fan without rays or cones:

```
sage: Fan([], [])
Traceback (most recent call last):
...
ValueError: you must specify the lattice
when you construct a fan without rays and cones!
sage: F = Fan([], [], lattice=ToricLattice(2, "L"))
sage: F
Rational polyhedral fan in 2-d lattice L
sage: F.lattice_dim()
2
sage: F.dim()
0
```

sage.geometry.fan.**Fan2d**(*rays*, *lattice=None*)

Construct the maximal 2-d fan with given `rays`.

In two dimensions we can uniquely construct a fan from just rays, just by cyclically ordering the rays and constructing as many cones as possible. This is why we implement a special constructor for this case.

INPUT:

- `rays` – list of rays given as list or vectors convertible to the rational extension of `lattice`. Duplicate rays are removed without changing the ordering of the remaining rays.
- `lattice` – `ToricLattice`, $\mathbf{Z}^n$, or any other object that behaves like these. If not specified, an attempt will be made to determine an appropriate toric lattice automatically.

EXAMPLES:

```
sage: Fan2d([(0,1), (1,0)])
Rational polyhedral fan in 2-d lattice N
sage: Fan2d([], lattice=ToricLattice(2, 'myN'))
Rational polyhedral fan in 2-d lattice myN
```

The ray order is as specified, even if it is not the cyclic order:

```
sage: fan1 = Fan2d([(0,1), (1,0)])
sage: fan1.rays()
N(0, 1),
N(1, 0)
in 2-d lattice N

sage: fan2 = Fan2d([(1,0), (0,1)])
sage: fan2.rays()
N(1, 0),
N(0, 1)
in 2-d lattice N

sage: fan1 == fan2, fan1.is_equivalent(fan2)
(False, True)

sage: fan = Fan2d([(1,1), (-1,-1), (1,-1), (-1,1)])
sage: [ cone.ambient_ray_indices() for cone in fan ]
[(2, 1), (1, 3), (3, 0), (0, 2)]
sage: fan.is_complete()
True
```

TESTS:

```
sage: Fan2d([(0,1), (0,1)]).generating_cones()
(1-d cone of Rational polyhedral fan in 2-d lattice N,)

sage: Fan2d([(1,1), (-1,-1)]).generating_cones()
(1-d cone of Rational polyhedral fan in 2-d lattice N,
 1-d cone of Rational polyhedral fan in 2-d lattice N)

sage: Fan2d([])
Traceback (most recent call last):
...
ValueError: you must specify a 2-dimensional lattice
when you construct a fan without rays.

sage: Fan2d([(3,4)]).rays()
N(3, 4)
in 2-d lattice N

sage: Fan2d([(0,1,0)])
Traceback (most recent call last):
...
ValueError: the lattice must be 2-dimensional.

sage: Fan2d([(0,1), (1,0), (0,0)])
Traceback (most recent call last):
...
ValueError: only non-zero vectors define rays

sage: Fan2d([(0, -2), (2, -10), (1, -3), (2, -9), (2, -12), (1, 1),
...          (2, 1), (1, -5), (0, -6), (1, -7), (0, 1), (2, -4),
...          (2, -2), (1, -9), (1, -8), (2, -6), (0, -1), (0, -3),
...          (2, -11), (2, -8), (1, 0), (0, -5), (1, -4), (2, 0),
...          (1, -6), (2, -7), (2, -5), (-1, -3), (1, -1), (1, -2),
...          (0, -4), (2, -3), (2, -1)]).cone_lattice()
Finite poset containing 44 elements
```

```
sage: Fan2d([(1,1)]).is_complete()
False
sage: Fan2d([(1,1), (-1,-1)]).is_complete()
False
sage: Fan2d([(1,0), (0,1)]).is_complete()
False
```

sage.geometry.fan.**NormalFan**(*polytope*, *lattice=None*)

Construct the normal fan of the given rational `polytope`.

INPUT:

- `polytope` – a full-dimensional `polytope` over **Q** or:class:*latticepolytope* < *sage.geometry.lattice*$_p$*olytope.LatticePolytopeClass* >.
- `lattice` – `ToricLattice`, $\mathbf{Z}^n$, or any other object that behaves like these. If not specified, an attempt will be made to determine an appropriate toric lattice automatically.

OUTPUT:

- `rational polyhedral fan`.

See also `FaceFan()`.

EXAMPLES:

Let's construct the fan corresponding to the product of two projective lines:

```
sage: square = LatticePolytope([(1,1), (-1,1), (-1,-1), (1,-1)])
sage: P1xP1 = NormalFan(square)
sage: P1xP1.rays()
N( 1,  0),
N( 0,  1),
N( 0, -1),
N(-1,  0)
in 2-d lattice N
sage: for cone in P1xP1: print cone.rays()
N( 0, -1),
N(-1,  0)
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
```

TESTS:

```
sage: cuboctahed.is_lattice_polytope(), cuboctahed.dilation(2).is_lattice_polytope()
(False, True)
sage: fan1 = NormalFan(cuboctahed)
sage: fan2 = NormalFan(cuboctahed.dilation(2).lattice_polytope())
sage: fan1.is_equivalent(fan2)
True
```

class sage.geometry.fan.**RationalPolyhedralFan**(*cones*, *rays*, *lattice*, *is_complete=None*, *virtual_rays=None*)

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

Create a rational polyhedral fan.

Warning: This class does not perform any checks of correctness of input nor does it convert input into the standard representation. Use Fan() to construct fans from "raw data" or FaceFan() and NormalFan() to get fans associated to polytopes.

Fans are immutable, but they cache most of the returned values.

INPUT:

- cones – list of generating cones of the fan, each cone given as a list of indices of its generating rays in rays;
- rays – list of immutable primitive vectors in lattice consisting of exactly the rays of the fan (i.e. no "extra" ones);
- lattice – ToricLattice, $\mathbf{Z}^n$, or any other object that behaves like these. If None, it will be determined as parent() of the first ray. Of course, this cannot be done if there are no rays, so in this case you must give an appropriate lattice directly;
- is_complete – if given, must be True or False depending on whether this fan is complete or not. By default, it will be determined automatically if necessary;
- virtual_rays – if given, must the a list of immutable primitive vectors in lattice, see virtual_rays() for details. By default, it will be determined automatically if necessary.

OUTPUT:

- rational polyhedral fan.

Gale_transform()

Return the Gale transform of self.

OUTPUT:

A matrix over ZZ.

EXAMPLES:

```
sage: fan = toric_varieties.P1xP1().fan()
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:

```
sage: fan = Fan([[0,1,3],[3,4],[2,0],[1,2,4]], [(-3, -2, 1), (0, 0, 1), (3, -2, 1), (-1, -1,
sage: fan.Stanley_Reisner_ideal( PolynomialRing(QQ,5,'A, B, C, D, E') )
Ideal (A*E, C*D, A*B*C, B*D*E) of Multivariate Polynomial Ring in A, B, C, D, E over Rationa
```

cartesian_product(*other*, *lattice=None*)

Return the Cartesian product of `self` with `other`.

INPUT:

- `other` – a `rational polyhedral fan`;
- `lattice` – (optional) the ambient lattice for the Cartesian product fan. By default, the direct sum of the ambient lattices of `self` and `other` is constructed.

OUTPUT:

- a `fan` whose cones are all pairwise Cartesian products of the cones of `self` and `other`.

EXAMPLES:

```
sage: K = ToricLattice(1, 'K')
sage: fan1 = Fan([[0],[1]],[(1,),(-1,)], lattice=K)
sage: L = ToricLattice(2, 'L')
sage: fan2 = Fan(rays=[(1,0),(0,1),(-1,-1)],
...           cones=[[0,1],[1,2],[2,0]], lattice=L)
sage: fan1.cartesian_product(fan2)
Rational polyhedral fan in 3-d lattice K+L
sage: _.ngenerating_cones()
6
```

complex(*base_ring=Integer Ring*, *extended=False*)

Return the chain complex of the fan.

To a d-dimensional fan Σ, one can canonically associate a chain complex $K^\bullet$

$$0 \longrightarrow \mathbf{Z}^{\Sigma(d)} \longrightarrow \mathbf{Z}^{\Sigma(d-1)} \longrightarrow \cdots \longrightarrow \mathbf{Z}^{\Sigma(0)} \longrightarrow 0$$

where the leftmost non-zero entry is in degree 0 and the rightmost entry in degree d. See [Klyachko], eq. (3.2). This complex computes the homology of $|\Sigma| \subset N_{\mathbf{R}}$ with arbitrary support,

$$H_i(K) = H_{d-i}(|\Sigma|, \mathbf{Z})_{\text{non-cpct}}$$

For a complete fan, this is just the non-compactly supported homology of $\mathbf{R}^d$. In this case, $H_0(K) = \mathbf{Z}$ and 0 in all non-zero degrees.

For a complete fan, there is an extended chain complex

$$0 \longrightarrow \mathbf{Z} \longrightarrow \mathbf{Z}^{\Sigma(d)} \longrightarrow \mathbf{Z}^{\Sigma(d-1)} \longrightarrow \cdots \longrightarrow \mathbf{Z}^{\Sigma(0)} \longrightarrow 0$$

where we take the first $\mathbf{Z}$ 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 $\mathbf{Z}$-term at degree -1 or not.
- `base_ring` – A ring (default: `ZZ`). The ring to use instead of $\mathbf{Z}$.

OUTPUT:

The complex associated to the fan as a `ChainComplex`. Raises a `ValueError` if the extended complex is requested for a non-complete fan.

EXAMPLES:

```
sage: fan = toric_varieties.P(3).fan()
sage: K_normal = fan.complex(); K_normal
Chain complex with at most 4 nonzero terms over Integer Ring
sage: K_normal.homology()
{0: Z, 1: 0, 2: 0, 3: 0}
sage: K_extended = fan.complex(extended=True); K_extended
Chain complex with at most 5 nonzero terms over Integer Ring
sage: K_extended.homology()
{0: 0, 1: 0, 2: 0, 3: 0, -1: 0}
```

Homology computations are much faster over $\mathbf{Q}$ if you don't care about the torsion coefficients:

```
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()
{0: Vector space of dimension 0 over Rational Field,
 1: Vector space of dimension 0 over Rational Field,
 2: Vector space of dimension 0 over Rational Field,
 -1: Vector space of dimension 0 over Rational Field}
```

The extended complex is only defined for complete fans:

```
sage: fan = Fan([ Cone([(1,0)]) ])
sage: fan.is_complete()
False
sage: fan.complex(extended=True)
Traceback (most recent call last):
...
ValueError: The extended complex is only defined for complete fans!
```

The definition of the complex does not refer to the ambient space of the fan, so it does not distinguish a fan from the same fan embedded in a subspace:

```
sage: K1 = Fan([Cone([(-1,)]), Cone([(1,)])]).complex()
sage: K2 = Fan([Cone([(-1,0,0)]), Cone([(1,0,0)])]).complex()
sage: K1 == K2
True
```

Things get more complicated for non-complete fans:

```
sage: fan = Fan([Cone([(1,1,1)]),
...              Cone([(1,0,0),(0,1,0)]),
...              Cone([(-1,0,0),(0,-1,0),(0,0,-1)])])
sage: fan.complex().homology()
{0: 0, 1: 0, 2: Z x Z, 3: 0}
sage: fan = Fan([Cone([(1,0,0),(0,1,0)]),
...              Cone([(-1,0,0),(0,-1,0),(0,0,-1)])])
sage: fan.complex().homology()
{0: 0, 1: 0, 2: Z, 3: 0}
sage: fan = Fan([Cone([(-1,0,0),(0,-1,0),(0,0,-1)])])
sage: fan.complex().homology()
{0: 0, 1: 0, 2: 0, 3: 0}
```

REFERENCES:

cone_containing(**points*)

Return the smallest cone of `self` containing all given points.

INPUT:

- either one or more indices of rays of `self`, or one or more objects representing points of the ambient space of `self`, or a list of such objects (you CANNOT give a list of indices).

OUTPUT:

- A `cone of fan` whose ambient fan is `self`.

Note: We think of the origin as of the smallest cone containing no rays at all. If there is no ray in `self` that contains all `rays`, a `ValueError` exception will be raised.

EXAMPLES:

```
sage: cone1 = Cone([(0,-1), (1,0)])
sage: cone2 = Cone([(1,0), (0,1)])
sage: f = Fan([cone1, cone2])
sage: f.rays()
N(0,  1),
N(0, -1),
N(1,  0)
in 2-d lattice N
sage: f.cone_containing(0)  # ray index
1-d cone of Rational polyhedral fan in 2-d lattice N
sage: f.cone_containing(0, 1) # ray indices
Traceback (most recent call last):
...
ValueError: there is no cone in
Rational polyhedral fan in 2-d lattice N
containing all of the given rays! Ray indices: [0, 1]
sage: f.cone_containing(0, 2) # ray indices
2-d cone of Rational polyhedral fan in 2-d lattice N
sage: f.cone_containing((0,1))  # point
1-d cone of Rational polyhedral fan in 2-d lattice N
sage: f.cone_containing([(0,1)]) # point
1-d cone of Rational polyhedral fan in 2-d lattice N
sage: f.cone_containing((1,1))
2-d cone of Rational polyhedral fan in 2-d lattice N
sage: f.cone_containing((1,1), (1,0))
2-d cone of Rational polyhedral fan in 2-d lattice N
sage: f.cone_containing()
0-d cone of Rational polyhedral fan in 2-d lattice N
sage: f.cone_containing((0,0))
0-d cone of Rational polyhedral fan in 2-d lattice N
sage: f.cone_containing((-1,1))
Traceback (most recent call last):
...
ValueError: there is no cone in
Rational polyhedral fan in 2-d lattice N
containing all of the given points! Points: [N(-1, 1)]
```

TESTS:

```
sage: fan = Fan(cones=[(0,1,2,3), (0,1,4)],
...       rays=[(1,1,1), (1,-1,1), (1,-1,-1), (1,1,-1), (0,0,1)])
sage: fan.cone_containing(0).rays()
```

```
N(1, 1, 1)
in 3-d lattice N
```

cone_lattice()

Return the cone lattice of `self`.

This lattice will have the origin as the bottom (we do not include the empty set as a cone) and the fan itself as the top.

OUTPUT:

- finite poset <sage.combinat.posets.posets.FinitePoset of cones of fan, behaving like "regular" cones, but also containing the information about their relation to this fan, namely, the contained rays and containing generating cones. The top of the lattice will be this fan itself (*which is not a* cone of fan).

See also cones().

EXAMPLES:

Cone lattices can be computed for arbitrary fans:

```
sage: cone1 = Cone([(1,0), (0,1)])
sage: cone2 = Cone([(-1,0)])
sage: fan = Fan([cone1, cone2])
sage: fan.rays()
N( 0, 1),
N( 1, 0),
N(-1, 0)
in 2-d lattice N
sage: for cone in fan: print cone.ambient_ray_indices()
(0, 1)
(2,)
sage: L = fan.cone_lattice()
sage: L
Finite poset containing 6 elements
```

These 6 elements are the origin, three rays, one two-dimensional cone, and the fan itself. Since we do add the fan itself as the largest face, you should be a little bit careful with this last element:

```
sage: for face in L: print face.ambient_ray_indices()
Traceback (most recent call last):
...
AttributeError: 'RationalPolyhedralFan'
object has no attribute 'ambient_ray_indices'
sage: L.top()
Rational polyhedral fan in 2-d lattice N
```

For example, you can do

```
sage: for l in L.level_sets()[:-1]:
...       print [f.ambient_ray_indices() for f in l]
[()]
[(0,), (1,), (2,)]
[(0, 1)]
```

If the fan is complete, its cone lattice is atomic and coatomic and can (and will!) be computed in a much more efficient way, but the interface is exactly the same:

```
sage: fan = toric_varieties.P1xP1().fan()
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:

```
sage: fan = Fan([cone1])
sage: L = fan.cone_lattice()
sage: L
Finite poset containing 5 elements
```

Here these 5 elements correspond to the origin, two rays, one generating cone of dimension two, and the whole fan. While this single cone "is" the whole fan, it is consistent and convenient to distinguish them in the cone lattice.

cones (*dim=None*, *codim=None*)

Return the specified cones of `self`.

INPUT:

- `dim` – dimension of the requested cones;
- `codim` – codimension of the requested cones.

Note: You can specify at most one input parameter.

OUTPUT:

- `tuple` of cones of `self` of the specified (co)dimension, if either `dim` or `codim` is given. Otherwise `tuple` of such tuples for all existing dimensions.

EXAMPLES:

```
sage: cone1 = Cone([(1,0), (0,1)])
sage: cone2 = Cone([(-1,0)])
sage: fan = Fan([cone1, cone2])
sage: fan(dim=0)
(0-d cone of Rational polyhedral fan in 2-d lattice N,)
sage: fan(codim=2)
(0-d cone of Rational polyhedral fan in 2-d lattice N,)
sage: for cone in fan.cones(1): cone.ray(0)
N(0, 1)
N(1, 0)
N(-1, 0)
sage: fan.cones(2)
(2-d cone of Rational polyhedral fan in 2-d lattice N,)
```

You cannot specify both dimension and codimension, even if they "agree":

```
sage: fan(dim=1, codim=1)
Traceback (most recent call last):
...
ValueError: dimension and codimension
cannot be specified together!
```

But it is OK to ask for cones of too high or low (co)dimension:

```
sage: fan(-1)
()
sage: fan(3)
()
sage: fan(codim=4)
()
```

contains(*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
sage: 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:

```
sage: ray = Cone([(0,-1,1)])
sage: ray
1-d cone in 3-d lattice N
sage: ray.ambient_ray_indices()
(0,)
sage: ray.adjacent()
()
sage: ray.ambient()
1-d cone in 3-d lattice N
```

If we want to operate with this ray as a part of the fan, we need to embed it first:

```
sage: e_ray = f.embed(ray)
sage: e_ray
1-d cone of Rational polyhedral fan in 3-d lattice N
sage: e_ray.rays()
N(0, -1, 1)
in 3-d lattice N
sage: e_ray is ray
False
sage: e_ray.is_equivalent(ray)
True
sage: e_ray.ambient_ray_indices()
(3,)
sage: e_ray.adjacent()
(1-d cone of Rational polyhedral fan in 3-d lattice N,
 1-d cone of Rational polyhedral fan in 3-d lattice N)
sage: e_ray.ambient()
Rational polyhedral fan in 3-d lattice N
```

Not every cone can be embedded into a fixed fan:

```
sage: f.embed(Cone([(0,0,1)]))
Traceback (most recent call last):
```

```
...
ValueError: 1-d cone in 3-d lattice N does not belong
to Rational polyhedral fan in 3-d lattice N!
sage: f.embed(Cone([(1,0,1), (-1,0,1)]))
Traceback (most recent call last):
...
ValueError: 2-d cone in 3-d lattice N does not belong
to Rational polyhedral fan in 3-d lattice N!
```

generating_cone(*n*)

Return the n-th generating cone of self.

INPUT:

- n – integer, the index of a generating cone.

OUTPUT:

- cone of fan.

EXAMPLES:

```
sage: fan = toric_varieties.P1xP1().fan()
sage: fan.generating_cone(0)
2-d cone of Rational polyhedral fan in 2-d lattice N
```

generating_cones()

Return generating cones of self.

OUTPUT:

- tuple of cones of fan.

EXAMPLES:

```
sage: fan = toric_varieties.P1xP1().fan()
sage: fan.generating_cones()
(2-d cone of Rational polyhedral fan in 2-d lattice N,
 2-d cone of Rational polyhedral fan in 2-d lattice N,
 2-d cone of Rational polyhedral fan in 2-d lattice N,
 2-d cone of Rational polyhedral fan in 2-d lattice N)
sage: cone1 = Cone([(1,0), (0,1)])
sage: cone2 = Cone([(-1,0)])
sage: fan = Fan([cone1, cone2])
sage: fan.generating_cones()
(2-d cone of Rational polyhedral fan in 2-d lattice N,
 1-d cone of Rational polyhedral fan in 2-d lattice N)
```

is_complete()

Check if self is complete.

A rational polyhedral fan is *complete* if its cones fill the whole space.

OUTPUT:

- True if self is complete and False otherwise.

EXAMPLES:

```
sage: fan = toric_varieties.P1xP1().fan()
sage: fan.is_complete()
True
sage: cone1 = Cone([(1,0), (0,1)])
sage: cone2 = Cone([(-1,0)])
```

```
sage: fan = Fan([cone1, cone2])
sage: fan.is_complete()
False
```

is_equivalent(*other*)

Check if `self` is "mathematically" the same as `other`.

INPUT:

- `other` - fan.

OUTPUT:

- `True` if `self` and `other` define the same fans as collections of equivalent cones in the same lattice, `False` otherwise.

There are three different equivalences between fans F_1 and F_2 in the same lattice:

1. They have the same rays in the same order and the same generating cones in the same order. This is tested by `F1 == F2`.
2. They have the same rays and the same generating cones without taking into account any order. This is tested by `F1.is_equivalent(F2)`.
3. They are in the same orbit of $GL(n, \mathbf{Z})$ (and, therefore, correspond to isomorphic toric varieties). This is tested by `F1.is_isomorphic(F2)`.

Note that `virtual_rays()` are included into consideration for all of the above equivalences.

EXAMPLES:

```
sage: fan1 = Fan(cones=[(0,1), (1,2)],
...              rays=[(1,0), (0,1), (-1,-1)],
...              check=False)
sage: fan2 = Fan(cones=[(2,1), (0,2)],
...              rays=[(1,0), (-1,-1), (0,1)],
...              check=False)
sage: fan3 = Fan(cones=[(0,1), (1,2)],
...              rays=[(1,0), (0,1), (-1,1)],
...              check=False)
sage: fan1 == fan2
False
sage: fan1.is_equivalent(fan2)
True
sage: fan1 == fan3
False
sage: fan1.is_equivalent(fan3)
False
```

is_isomorphic(*other*)

Check if `self` is in the same $GL(n, \mathbf{Z})$-orbit as `other`.

There are three different equivalences between fans F_1 and F_2 in the same lattice:

1. They have the same rays in the same order and the same generating cones in the same order. This is tested by `F1 == F2`.
2. They have the same rays and the same generating cones without taking into account any order. This is tested by `F1.is_equivalent(F2)`.
3. They are in the same orbit of $GL(n, \mathbf{Z})$ (and, therefore, correspond to isomorphic toric varieties). This is tested by `F1.is_isomorphic(F2)`.

Note that virtual_rays() are included into consideration for all of the above equivalences.

INPUT:

- other – a fan.

OUTPUT:

- True if self and other are in the same $GL(n, \mathbf{Z})$-orbit, False otherwise.

See Also:

If you want to obtain the actual fan isomorphism, use isomorphism().

EXAMPLES:

Here we pick an $SL(2, \mathbf{Z})$ matrix m and then verify that the image fan is isomorphic:

```
sage: rays = ((1, 1), (0, 1), (-1, -1), (1, 0))
sage: cones = [(0,1), (1,2), (2,3), (3,0)]
sage: fan1 = Fan(cones, rays)
sage: m = matrix([[-2,3],[1,-1]])
sage: fan2 = Fan(cones, [vector(r)*m for r in rays])
sage: fan1.is_isomorphic(fan2)
True
sage: fan1.is_equivalent(fan2)
False
sage: fan1 == fan2
False
```

These fans are "mirrors" of each other:

```
sage: fan1 = Fan(cones=[(0,1), (1,2)],
...              rays=[(1,0), (0,1), (-1,-1)],
...              check=False)
sage: fan2 = Fan(cones=[(0,1), (1,2)],
...              rays=[(1,0), (0,-1), (-1,1)],
...              check=False)
sage: fan1 == fan2
False
sage: fan1.is_equivalent(fan2)
False
sage: fan1.is_isomorphic(fan2)
True
sage: fan1.is_isomorphic(fan1)
True
```

is_simplicial()

Check if self is simplicial.

A rational polyhedral fan is **simplicial** if all of its cones are, i.e. primitive vectors along generating rays of every cone form a part of a *rational* basis of the ambient space.

OUTPUT:

- True if self is simplicial and False otherwise.

EXAMPLES:

```
sage: fan = toric_varieties.P1xP1().fan()
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:

```
sage: fan = NormalFan(lattice_polytope.cross_polytope(3))
sage: fan.is_simplicial()
False
sage: fan.generating_cone(0).nrays()
4
```

is_smooth(*codim=None*)

Check if `self` is smooth.

A rational polyhedral fan is **smooth** if all of its cones are, i.e. primitive vectors along generating rays of every cone form a part of an *integral* basis of the ambient space. In this case the corresponding toric variety is smooth.

A fan in an n-dimensional lattice is smooth up to codimension c if all cones of codimension greater than or equal to c are smooth, i.e. if all cones of dimension less than or equal to $n - c$ are smooth. In this case the singular set of the corresponding toric variety is of dimension less than c.

INPUT:

- `codim` – codimension in which smoothness has to be checked, by default complete smoothness will be checked.

OUTPUT:

- `True` if `self` is smooth (in codimension `codim`, if it was given) and `False` otherwise.

EXAMPLES:

```
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:

Returns the ideal, in the given `ring`, generated by the linear relations of the rays. In toric geometry, this corresponds to rational equivalence of divisors.

EXAMPLES:

```
sage: fan = Fan([[0,1,3],[3,4],[2,0],[1,2,4]], [(-3, -2, 1), (0, 0, 1), (3, -2, 1), (-1, -1,
sage: fan.linear_equivalence_ideal( PolynomialRing(QQ,5,'A, B, C, D, E') )
Ideal (-3*A + 3*C - D + E, -2*A - 2*C - D - E, A + B + C + D + E) of Multivariate Polynomial
```

make_simplicial(***kwds*)

Construct a simplicial fan subdividing `self`.

It is a synonym for `subdivide()` with `make_simplicial=True` option.

INPUT:

- this functions accepts only keyword arguments. See `subdivide()` for documentation.

OUTPUT:

- `rational polyhedral fan`.

EXAMPLES:

```
sage: fan = NormalFan(lattice_polytope.cross_polytope(3))
sage: fan.is_simplicial()
False
sage: fan.ngenerating_cones()
6
sage: new_fan = fan.make_simplicial()
sage: new_fan.is_simplicial()
True
sage: new_fan.ngenerating_cones()
12
```

ngenerating_cones()

Return the number of generating cones of `self`.

OUTPUT:

- integer.

EXAMPLES:

```
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 argrees or disagrees with the "outward normal first" boundary orientation. Note that the orientation of any individial cone is arbitrary. This method once and for all picks orientations for all cones and then computes the boundaries relative to that chosen orientation.

If `cone` is the fan itself, the generating cones with their orientation relative to the ambient space are returned.

See `complex()` for the associated chain complex. If you do not require the orientation, use `cone.facets()` instead.

EXAMPLES:

```
sage: fan = toric_varieties.P(3).fan()
sage: cone = fan(2)[0]
sage: bdry = fan.oriented_boundary(cone);  bdry
1-d cone of Rational polyhedral fan in 3-d lattice N
- 1-d cone of Rational polyhedral fan in 3-d lattice N
sage: bdry[0]
(1, 1-d cone of Rational polyhedral fan in 3-d lattice N)
sage: bdry[1]
(-1, 1-d cone of Rational polyhedral fan in 3-d lattice N)
sage: fan.oriented_boundary(bdry[0][1])
```

```
-0-d cone of Rational polyhedral fan in 3-d lattice N
sage: fan.oriented_boundary(bdry[1][1])
-0-d cone of Rational polyhedral fan in 3-d lattice N
```

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
sage: [cone.rays().basis().matrix().det()
...    for cone in fan.generating_cones()]
[-1, 1, -1, 1]
```

A non-full dimensional fan:

```
sage: cone = Cone([(4,5)])
sage: fan = Fan([cone])
sage: fan.oriented_boundary(cone)
0-d cone of Rational polyhedral fan in 2-d lattice N
sage: fan.oriented_boundary(fan)
1-d cone of Rational polyhedral fan in 2-d lattice N
```

TESTS:

```
sage: fan = toric_varieties.P2().fan()
sage: trivial_cone = fan(0)[0]
sage: fan.oriented_boundary(trivial_cone)
0
```

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()
```

primitive_collections()

Return the primitive collections.

OUTPUT:

Returns the subsets $\{i_1, \ldots, i_k\} \subset \{1, \ldots, n\}$ such that

•The points $\{p_{i_1}, \ldots, p_{i_k}\}$ do not span a cone of the fan.

•If you remove any one p_{i_j} from the set, then they do span a cone of the fan.

Note: By replacing the multiindices $\{i_1, \ldots, i_k\}$ of each primitive collection with the monomials $x_{i_1} \cdots x_{i_k}$ one generates the Stanley-Reisner ideal in $\mathbf{Z}[x_1, \ldots]$.

REFERENCES:

V.V. Batyrev, On the classification of smooth projective toric varieties, Tohoku Math.J. 43 (1991), 569-585

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,
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:

```
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
```

TESTS:

We check that Trac #11902 is fixed:

```
sage: fan = toric_varieties.P2().fan()
sage: fan.subdivide(new_rays=[(0,0)])
Traceback (most recent call last):
...
ValueError: the origin cannot be used for fan subdivision!
```

support_contains(**args*)

Check if a point is contained in the support of the fan.

The support of a fan is the union of all cones of the fan. If you want to know whether the fan contains a given cone, you should use `contains()` instead.

INPUT:

- `*args` – an element of `self.lattice()` or something that can be converted to it (for example, a list of coordinates).

OUTPUT:

- `True` if `point` is contained in the support of the fan, `False` otherwise.

TESTS:

```
sage: cone1 = Cone([(0,-1), (1,0)])
sage: cone2 = Cone([(1,0), (0,1)])
sage: f = Fan([cone1, cone2])
```

We check if some points are in this fan:

```
sage: f.support_contains(f.lattice()(1,0))
True
sage: f.support_contains(cone1)     # a cone is not a point of the lattice
False
sage: f.support_contains((1,0))
True
sage: f.support_contains(1,1)
True
sage: f.support_contains((-1,0))
False
sage: f.support_contains(f.lattice().dual()(1,0)) #random output (warning)
False
sage: f.support_contains(f.lattice().dual()(1,0))
False
sage: f.support_contains(1)
False
sage: f.support_contains(0)    # 0 converts to the origin in the lattice
True
sage: f.support_contains(1/2, sqrt(3))
True
sage: f.support_contains(-1/2, sqrt(3))
False
```

vertex_graph()

Return the graph of 1- and 2-cones.

OUTPUT:

An edge-colored graph. The vertices correspond to the 1-cones (i.e. rays) of the fan. Two vertices are joined by an edge iff the rays span a 2-cone of the fan. The edges are colored by pairs of integers that classify the 2-cones up to $GL(2, \mathbf{Z})$ transformation, see `classify_cone_2d()`.

EXAMPLES:

```
sage: 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)]

sage: g = toric_varieties.Cube_deformation(10).fan().vertex_graph()
sage: g.automorphism_group().order()
```

```
48
sage: g.automorphism_group(edge_labels=True).order()
4
```

virtual_rays(**args*)

Return (some of the) virtual rays of `self`.

Let N be the D-dimensional `lattice()` of a d-dimensional fan Σ in $N_{\mathbf{R}}$. Then the corresponding toric variety is of the form $X \times (\mathbf{C}^*)^{D-d}$. The actual `rays()` of Σ give a canonical choice of homogeneous coordinates on X. This function returns an arbitrary but fixed choice of virtual rays corresponding to a (non-canonical) choice of homogeneous coordinates on the torus factor. Combinatorially primitive integral generators of virtual rays span the $D - d$ dimensions of $N_{\mathbf{Q}}$ "missed" by the actual rays. (In general addition of virtual rays is not sufficient to span N over $\mathbf{Z}$.)

..note:

```
You may use a particular choice of virtual rays by passing optional
argument ``virtual_rays`` to the :func:`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:

```
sage: f = Fan([Cone([(1,0,1,0), (0,1,1,0)])])
sage: f.virtual_rays()
N(0, 0, 0, 1),
N(0, 0, 1, 0)
in 4-d lattice N

sage: f.rays()
N(1, 0, 1, 0),
N(0, 1, 1, 0)
in 4-d lattice N

sage: f.virtual_rays([0])
N(0, 0, 0, 1)
in 4-d lattice N
```

You can also give virtual ray indices directly, without packing them into a list:

```
sage: f.virtual_rays(0)
N(0, 0, 0, 1)
in 4-d lattice N
```

Make sure that trac ticket #16344 is fixed and one can compute the virtual rays of fans in non-saturated lattices:

```
sage: N = ToricLattice(1)
sage: B = N.submodule([(2,)]).basis()
sage: f = Fan([Cone([B[0]])])
sage: len(f.virtual_rays())
0
```

TESTS:

```
sage: N = ToricLattice(4)
sage: for i in range(10):
...       c = Cone([N.random_element() for j in range(i/2)], lattice=N)
...       f = Fan([c])
...       assert matrix(f.rays() + f.virtual_rays()).rank() == 4
...       assert f.dim() + len(f.virtual_rays()) == 4
```

sage.geometry.fan.**discard_faces**(*cones*)

Return the cones of the given list which are not faces of each other.

INPUT:

- cones – a list of cones.

OUTPUT:

- a list of cones, sorted by dimension in decreasing order.

EXAMPLES:

Consider all cones of a fan:

```
sage: Sigma = toric_varieties.P2().fan()
sage: cones = flatten(Sigma.cones())
sage: len(cones)
7
```

Most of them are not necessary to generate this fan:

```
sage: from sage.geometry.fan import discard_faces
sage: len(discard_faces(cones))
3
sage: Sigma.ngenerating_cones()
3
```

sage.geometry.fan.**is_Fan**(*x*)

Check if x is a Fan.

INPUT:

- x – anything.

OUTPUT:

- True if x is a fan and False otherwise.

EXAMPLES:

```
sage: from sage.geometry.fan import is_Fan
sage: is_Fan(1)
False
sage: fan = toric_varieties.P2().fan()
sage: fan
Rational polyhedral fan in 2-d lattice N
sage: is_Fan(fan)
True
```

CHAPTER

FOUR

MORPHISMS BETWEEN TORIC LATTICES COMPATIBLE WITH FANS

This module is a part of the framework for toric varieties (`variety`, `fano_variety`). Its main purpose is to provide support for working with lattice morphisms compatible with fans via `FanMorphism` class.

AUTHORS:

- Andrey Novoseltsev (2010-10-17): initial version.
- **Andrey Novoseltsev (2011-04-11): added tests for injectivity/surjectivity,** fibration, bundle, as well as some related methods.

EXAMPLES:

Let's consider the face and normal fans of the "diamond" and the projection to the x-axis:

```
sage: diamond = lattice_polytope.cross_polytope(2)
sage: face = FaceFan(diamond, lattice=ToricLattice(2))
sage: normal = NormalFan(diamond)
sage: N = face.lattice()
sage: H = End(N)
sage: phi = H([N.0, 0])
sage: phi
Free module morphism defined by the matrix
[1 0]
[0 0]
Domain: 2-d lattice N
Codomain: 2-d lattice N
sage: FanMorphism(phi, normal, face)
Traceback (most recent call last):
...
ValueError: the image of generating cone #1 of the domain fan
is not contained in a single cone of the codomain fan!
```

Some of the cones of the normal fan fail to be mapped to a single cone of the face fan. We can rectify the situation in the following way:

```
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 Σ_1 and Σ_2 be two fans in lattices N_1 and N_2 respectively. Let ϕ be a morphism (i.e. a linear map) from N_1 to N_2. We say that ϕ is *compatible* with Σ_1 and Σ_2 if every cone $\sigma_1 \in \Sigma_1$ is mapped by ϕ into a single cone $\sigma_2 \in \Sigma_2$, i.e. $\phi(\sigma_1) \subset \sigma_2$ (σ_2 may be different for different σ_1).

By a **fan morphism** we understand a morphism between two lattices compatible with specified fans in these lattices. Such morphisms behave in exactly the same way as "regular" morphisms between lattices, but:

- fan morphisms have a special constructor allowing some automatic adjustments to the initial fans (see below);
- fan morphisms are aware of the associated fans and they can be accessed via codomain_fan() and domain_fan();
- fan morphisms can efficiently compute image_cone() of a given cone of the domain fan and preimage_cones() of a given cone of the codomain fan.

INPUT:

- morphism – either a morphism between domain and codomain, or an integral matrix defining such a morphism;
- domain_fan – a fan in the domain;
- codomain – (default: None) either a codomain lattice or a fan in the codomain. If the codomain fan is not given, the image fan (fan generated by images of generating cones) of domain_fan will be used, if possible;
- subdivide – (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:

```
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`.

```
sage: FanMorphism(phi, normal, face)
Traceback (most recent call last):
...
ValueError: the image of generating cone #1 of the domain fan
is not contained in a single cone of the codomain fan!
sage: fm = FanMorphism(phi, normal, face, subdivide=True)
sage: fm.domain_fan() is normal
False
sage: fm.domain_fan().ngenerating_cones()
6
```

We had to subdivide two of the four cones of the normal fan, since they were mapped by `phi` into non-strictly convex cones.

It is possible to omit the codomain fan, in which case the image fan will be used instead of it:

```
sage: fm = FanMorphism(phi, face)
sage: fm.codomain_fan()
Rational polyhedral fan in 2-d lattice N
sage: fm.codomain_fan().rays()
N( 1, 0),
N(-1, 0)
in 2-d lattice N
```

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(*dim=None*, *codim=None*)

Return the codomain fan of `self`.

INPUT:

- `dim` – dimension of the requested cones;
- `codim` – codimension of the requested cones.

OUTPUT:

- `rational polyhedral fan` if no parameters were given, `tuple` of `cones` otherwise.

EXAMPLES:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: quadrant = Fan([quadrant])
sage: quadrant_bl = quadrant.subdivide([(1,1)])
sage: fm = FanMorphism(identity_matrix(2), quadrant_bl, quadrant)
sage: fm.codomain_fan()
Rational polyhedral fan in 2-d lattice N
sage: fm.codomain_fan() is quadrant
True
```

domain_fan(*dim=None*, *codim=None*)

Return the codomain fan of `self`.

INPUT:

- dim – dimension of the requested cones;
- codim – codimension of the requested cones.

OUTPUT:

- rational polyhedral fan if no parameters were given, tuple of cones otherwise.

EXAMPLES:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: quadrant = Fan([quadrant])
sage: quadrant_bl = quadrant.subdivide([(1,1)])
sage: fm = FanMorphism(identity_matrix(2), quadrant_bl, quadrant)
sage: fm.domain_fan()
Rational polyhedral fan in 2-d lattice N
sage: fm.domain_fan() is quadrant_bl
True
```

factor()

Factor self into injective * birational * surjective morphisms.

OUTPUT:

- a triple of FanMorphism (ϕ_i, ϕ_b, ϕ_s), such that ϕ_s is surjective, ϕ_b is birational, ϕ_i is injective, and self is equal to $\phi_i \circ \phi_b \circ \phi_s$.

Intermediate fans live in the saturation of the image of self as a map between lattices and are the image of the domain_fan() and the restriction of the codomain_fan(), i.e. if self maps $\Sigma \to \Sigma'$, then we have factorization into

$$\Sigma \twoheadrightarrow \Sigma_s \to \Sigma_i \hookrightarrow \Sigma.$$

Note:

- Σ_s is the finest fan with the smallest support that is compatible with self: any fan morphism from Σ given by the same map of lattices as self factors through Σ_s.
- Σ_i is the coarsest fan of the largest support that is compatible with self: any fan morphism into Σ' given by the same map of lattices as self factors though Σ_i.

EXAMPLES:

We map an affine plane into a projective 3-space in such a way, that it becomes "a double cover of a chart of the blow up of one of the coordinate planes":

```
sage: A2 = toric_varieties.A2()
sage: P3 = toric_varieties.P(3)
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,  0, 0),
N( 0,  1, 0),
N(-1, -1, 0)
in Sublattice <N(1, 0, 0), N(0, 1, 0)>
```

Coordinate plane inclusion (injective):

```
sage: 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.codomain_fan().rays()
N( 1,  0,  0),
N( 0,  1,  0),
N( 0,  0,  1),
N(-1, -1, -1)
```

```
in 3-d lattice N
```

TESTS:

```
sage: phi_s.matrix() * phi_b.matrix() * phi_i.matrix() == m
True

sage: phi.domain_fan() is phi_s.domain_fan()
True
sage: phi_s.codomain_fan() is phi_b.domain_fan()
True
sage: phi_b.codomain_fan() is phi_i.domain_fan()
True
sage: phi_i.codomain_fan() is phi.codomain_fan()
True

sage: trivialfan2 = Fan([],[],lattice=ToricLattice(2))
sage: trivialfan3 = Fan([],[],lattice=ToricLattice(3))
sage: f = FanMorphism(zero_matrix(2,3), trivialfan2, trivialfan3)
sage: [phi.matrix().dimensions() for phi in f.factor()]
[(0, 3), (0, 0), (2, 0)]
```

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
```

TESTS:

We check that complete codomain fans are handled correctly, since a different algorithm is used in this case:

```
sage: diamond = lattice_polytope.cross_polytope(2)
sage: face = FaceFan(diamond, lattice=ToricLattice(2))
sage: normal = NormalFan(diamond)
sage: N = face.lattice()
sage: fm = FanMorphism(identity_matrix(2),
...           normal, face, subdivide=True)
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 σ was given, the index of `self` over σ is computed in the sense of Definition 2.1.7 of [HLY]: if σ' is any cone of the `domain_fan()` of `self` whose relative interior is mapped to the relative interior of σ, it is the index of the image of $N'(\sigma')$ in $N(\sigma)$, where N' and N are domain and codomain lattices respectively. While that definition was formulated for the case of the finite index only, we extend it to the infinite one as well and return `None` if there is no σ' at all. See examples below for situations when such things happen. Note also that the index of `self` is the same as index over the trivial cone.

EXAMPLES:

```
sage: 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()
N( 1, 0),
N(-1, 0)
in Sublattice <N(1, 0)>
```

We see that one of the rays of the fan of `P1` is mapped to a ray, while the other one to the interior of some 2-d cone. Both rays correspond to single points on `P1`, yet one is mapped to the distinguished point of a torus invariant curve of `dP8` (with the rest of this curve being uncovered) and the other to a fixed point of `dP8` (thus completely covering this torus orbit in `dP8`).

We should therefore expect the following behaviour: all indices over 1-d cones are `None`, except for one which is infinite, and all indices over 2-d cones are `None`, except for one which is 1:

```
sage: [xi.index(cone) for cone in Sigma(1)]
[None, None, None, +Infinity]
```

```
sage: [xi.index(cone) for cone in Sigma(2)]
[None, 1, None, None]
```

TESTS:

```
sage: Sigma = toric_varieties.dP8().fan()
sage: Sigma_p = toric_varieties.Cube_nonpolyhedral().fan()
sage: m = matrix([[2,6,10], [7,11,13]])
sage: zeta = FanMorphism(m, Sigma, Sigma_p, subdivide=True)
sage: [zeta.index(cone) for cone in flatten(Sigma_p.cones())]
[+Infinity, None, None, None, None, None, None, None, None, None,
 4, 4, None, 4, None, None, 2, None, 4, None, 4, 1, 1, 1, 1, 1, 1]
sage: zeta = prod(zeta.factor()[1:])
sage: Sigma_p = zeta.codomain_fan()
sage: [zeta.index(cone) for cone in flatten(Sigma_p.cones())]
[4, 4, 1, 4, 4, 4, 4, 1, 1, 1, 1, 1, 1]
sage: zeta.index() == zeta.index(Sigma_p(0)[0])
True
```

is_birational()

Check if `self` is birational.

OUTPUT:

- `True` if `self` is birational, `False` otherwise.

For fan morphisms this check is equivalent to `self.index() == 1` and means that the corresponding map between toric varieties is birational.

EXAMPLES:

```
sage: Sigma = toric_varieties.dP8().fan()
sage: Sigma_p = toric_varieties.P1().fan()
sage: phi = FanMorphism(matrix([[1], [-1]]), Sigma, Sigma_p)
sage: psi = FanMorphism(matrix([[2], [-2]]), Sigma, Sigma_p)
sage: xi = FanMorphism(matrix([[1, 0]]), Sigma_p, Sigma)
sage: phi.index(), psi.index(), xi.index()
(1, 2, +Infinity)
sage: phi.is_birational(), psi.is_birational(), xi.is_birational()
(True, False, False)
```

is_bundle()

Check if `self` is a bundle.

OUTPUT:

- `True` if `self` is a bundle, `False` otherwise.

Let $\phi : \Sigma \to \Sigma'$ be a fan morphism such that the underlying lattice morphism $\phi : N \to N'$ is surjective. Let Σ_0 be the kernel fan of ϕ. Then ϕ is a **bundle** (or splitting) if there is a subfan $\widehat{\Sigma}$ of Σ such that the following two conditions are satisfied:

1. Cones of Σ are precisely the cones of the form $\sigma_0 + \widehat{\sigma}$, where $\sigma_0 \in \Sigma_0$ and $\widehat{\sigma} \in \widehat{\Sigma}$.
2. Cones of $\widehat{\Sigma}$ are in bijection with cones of Σ' induced by ϕ and ϕ maps lattice points in every cone $\widehat{\sigma} \in \widehat{\Sigma}$ bijectively onto lattice points in $\phi(\widehat{\sigma})$.

If a fan morphism $\phi : \Sigma \to \Sigma'$ is a bundle, then X_Σ is a fiber bundle over $X_{\Sigma'}$ with fibers X_{Σ_0, N_0}, where N_0 is the kernel lattice of ϕ. See [CLS11] for more details.

See Also:

is_fibration(), kernel_fan().

REFERENCES:

EXAMPLES:

We consider several maps between fans of a del Pezzo surface and the projective line:

```
sage: Sigma = toric_varieties.dP8().fan()
sage: Sigma_p = toric_varieties.P1().fan()
sage: phi = FanMorphism(matrix([[1], [-1]]), Sigma, Sigma_p)
sage: psi = FanMorphism(matrix([[2], [-2]]), Sigma, Sigma_p)
sage: xi = FanMorphism(matrix([[1, 0]]), Sigma_p, Sigma)
sage: phi.is_bundle()
True
sage: phi.is_fibration()
True
sage: phi.index()
1
sage: psi.is_bundle()
False
sage: psi.is_fibration()
True
sage: psi.index()
2
sage: xi.is_fibration()
False
sage: xi.index()
+Infinity
```

The first of these maps induces not only a fibration, but a fiber bundle structure. The second map is very similar, yet it fails to be a bundle, as its index is 2. The last map is not even a fibration.

is_dominant()

Return whether the fan morphism is dominant.

A fan morphism ϕ is dominant if it is surjective as a map of vector spaces. That is, $\phi_{\mathbf{R}} : N_{\mathbf{R}} \to N'_{\mathbf{R}}$ is surjective.

If the domain fan is complete, then this implies that the fan morphism is surjective.

If the fan morphism is dominant, then the associated morphism of toric varieties is dominant in the algebraic-geometric sense (that is, surjective onto a dense subset).

OUTPUT:

Boolean.

EXAMPLES:

```
sage: P1 = toric_varieties.P1()
sage: A1 = toric_varieties.A1()
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 σ' the linear map of vector spaces $\phi_{\mathbf{R}}$ induces a bijection between σ and σ', and, in addition, *phi* is `dominant` (that is, $\phi_{\mathbf{R}} : N_{\mathbf{R}} \to N'_{\mathbf{R}}$ is surjective).

If a fan morphism $\phi : \Sigma \to \Sigma'$ is a fibration, then the associated morphism between toric varieties $\tilde{\phi} : X_\Sigma \to X_{\Sigma'}$ is a fibration in the sense that it is surjective and all of its fibers have the same dimension, namely $\dim X_\Sigma - \dim X_{\Sigma'}$. These fibers do *not* have to be isomorphic, i.e. a fibration is not necessarily a fiber bundle. See [HLY02] for more details.

See Also:

`is_bundle()`, `primitive_preimage_cones()`.

REFERENCES:

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.

TESTS:

We check that reviewer's example on Trac 11200 works as expected:

```
sage: P1 = toric_varieties.P1()
sage: A1 = toric_varieties.A1()
sage: phi = FanMorphism(matrix([[1]]), A1.fan(), P1.fan())
sage: phi.is_fibration()
False
```

is_injective()

Check if `self` is injective.

OUTPUT:

- `True` if `self` is injective, `False` otherwise.

Let $\phi : \Sigma \to \Sigma'$ be a fan morphism such that the underlying lattice morphism $\phi : N \to N'$ bijectively maps N to a *saturated* sublattice of N'. Let $\psi : \Sigma \to \Sigma'_0$ be the restriction of ϕ to the image. Then ϕ is **injective** if the map between cones corresponding to ψ (injectively) maps each cone of Σ to a cone of the same dimension.

If a fan morphism $\phi : \Sigma \to \Sigma'$ is injective, then the associated morphism between toric varieties $\tilde{\phi} : X_\Sigma \to X_{\Sigma'}$ is injective.

See Also:

`factor()`.

EXAMPLES:

Consider the fan of the affine plane:

```
sage: A2 = toric_varieties.A(2).fan()
```

We will map several fans consisting of a single ray into the interior of the 2-cone:

```
sage: Sigma = Fan([Cone([(1,1)])])
sage: m = identity_matrix(2)
sage: FanMorphism(m, Sigma, A2).is_injective()
False
```

This morphism was not injective since (in the toric varieties interpretation) the 1-dimensional orbit corresponding to the ray was mapped to the 0-dimensional orbit corresponding to the 2-cone.

```
sage: Sigma = Fan([Cone([(1,)])])
sage: m = matrix(1, 2, [1,1])
sage: FanMorphism(m, Sigma, A2).is_injective()
True
```

While the fans in this example are close to the previous one, here the ray corresponds to a 0-dimensional orbit.

```
sage: Sigma = Fan([Cone([(1,)])])
sage: m = matrix(1, 2, [2,2])
sage: FanMorphism(m, Sigma, A2).is_injective()
False
```

Here the problem is that `m` maps the domain lattice to a non-saturated sublattice of the codomain. The corresponding map of the toric varieties is a two-sheeted cover of its image.

We also embed the affine plane into the projective one:

```
sage: P2 = toric_varieties.P(2).fan()
sage: m = identity_matrix(2)
sage: FanMorphism(m, A2, P2).is_injective()
True
```

is_surjective()

Check if `self` is surjective.

OUTPUT:

- `True` if `self` is surjective, `False` otherwise.

A fan morphism $\phi : \Sigma \to \Sigma'$ is **surjective** if the corresponding map between cones is surjective, i.e. for each cone $\sigma' \in \Sigma'$ there is at least one preimage cone $\sigma \in \Sigma$ such that the relative interior of σ is mapped to the relative interior of σ' and, in addition, $\phi_{\mathbf{R}} : N_{\mathbf{R}} \to N'_{\mathbf{R}}$ is surjective.

If a fan morphism $\phi : \Sigma \to \Sigma'$ is surjective, then the associated morphism between toric varieties $\tilde{\phi} : X_\Sigma \to X_{\Sigma'}$ is surjective.

See Also:

is_bundle(), is_fibration(), preimage_cones(), is_complete().

EXAMPLES:

We check that the blow up of the affine plane at the origin is surjective:

```
sage: A2 = toric_varieties.A(2).fan()
sage: Bl = A2.subdivide([(1,1)])
sage: m = identity_matrix(2)
sage: FanMorphism(m, Bl, A2).is_surjective()
True
```

It remains surjective if we throw away "south and north poles" of the exceptional divisor:

```
sage: FanMorphism(m, Fan(Bl.cones(1)), A2).is_surjective()
True
```

But a single patch of the blow up does not cover the plane:

```
sage: F = Fan([Bl.generating_cone(0)])
sage: FanMorphism(m, F, A2).is_surjective()
False
```

TESTS:

We check that reviewer's example on trac ticket #11200 works as expected:

```
sage: P1 = toric_varieties.P1()
sage: A1 = toric_varieties.A1()
sage: phi = FanMorphism(matrix([[1]]), A1.fan(), P1.fan())
sage: phi.is_surjective()
False
```

kernel_fan()

Return the subfan of the domain fan mapped into the origin.

OUTPUT:

- a fan.

Note: The lattice of the kernel fan is the kernel() sublattice of self.

See Also:

preimage_fan().

EXAMPLES:

```
sage: fan = Fan(rays=[(1,0), (1,1), (0,1)], cones=[(0,1), (1,2)])
sage: fm = FanMorphism(matrix(2, 1, [1,-1]), fan, ToricLattice(1))
sage: fm.kernel_fan()
Rational polyhedral fan in Sublattice <N(1, 1)>
sage: _.rays()
N(1, 1)
in Sublattice <N(1, 1)>
sage: fm.kernel_fan().cones()
((0-d cone of Rational polyhedral fan in Sublattice <N(1, 1)>,),
 (1-d cone of Rational polyhedral fan in Sublattice <N(1, 1)>,))
```

preimage_cones(*cone*)

Return cones of the domain fan whose image_cone() is cone.

INPUT:

- cone – a cone equivalent to a cone of the codomain_fan() of self.

OUTPUT:

- a tuple of cones of the domain_fan() of self, sorted by dimension.

See Also:

preimage_fan().

EXAMPLES:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: quadrant = Fan([quadrant])
sage: quadrant_bl = quadrant.subdivide([(1,1)])
sage: fm = FanMorphism(identity_matrix(2), quadrant_bl, quadrant)
sage: fm.preimage_cones(Cone([(1,0)]))
(1-d cone of Rational polyhedral fan in 2-d lattice N,)
sage: fm.preimage_cones(Cone([(1,0), (0,1)]))
(1-d cone of Rational polyhedral fan in 2-d lattice N,
 2-d cone of Rational polyhedral fan in 2-d lattice N,
 2-d cone of Rational polyhedral fan in 2-d lattice N)
```

TESTS:

We check that reviewer's example from Trac #9972 is handled correctly:

```
sage: N1 = ToricLattice(1)
sage: N2 = ToricLattice(2)
sage: Hom21 = Hom(N2, N1)
sage: pr = Hom21([N1.0,0])
sage: P1xP1 = toric_varieties.P1xP1()
sage: f = FanMorphism(pr, P1xP1.fan())
sage: c = f.image_cone(Cone([(1,0), (0,1)]))
sage: c
1-d cone of Rational polyhedral fan in 1-d lattice N
sage: f.preimage_cones(c)
(1-d cone of Rational polyhedral fan in 2-d lattice N,
 2-d cone of Rational polyhedral fan in 2-d lattice N,
 2-d cone of Rational polyhedral fan in 2-d lattice N)
```

preimage_fan(*cone*)

Return the subfan of the domain fan mapped into cone.

INPUT:

- cone – a cone equivalent to a cone of the codomain_fan() of self.

OUTPUT:

- a fan.

Note: The preimage fan of cone consists of all cones of the domain_fan() which are mapped into cone, including those that are mapped into its boundary. So this fan is not necessarily generated by preimage_cones() of cone.

See Also:

kernel_fan(), preimage_cones().

EXAMPLES:

```
sage: quadrant_cone = Cone([(1,0), (0,1)])
sage: quadrant_fan = Fan([quadrant_cone])
sage: quadrant_bl = quadrant_fan.subdivide([(1,1)])
sage: fm = FanMorphism(identity_matrix(2),
...                    quadrant_bl, quadrant_fan)
sage: fm.preimage_fan(Cone([(1,0)])).cones()
((0-d cone of Rational polyhedral fan in 2-d lattice N,),
 (1-d cone of Rational polyhedral fan in 2-d lattice N,))
sage: fm.preimage_fan(quadrant_cone).ngenerating_cones()
2
sage: len(fm.preimage_cones(quadrant_cone))
3
```

primitive_preimage_cones(*cone*)

Return the primitive cones of the domain fan corresponding to `cone`.

INPUT:

- `cone` – a `cone` equivalent to a cone of the `codomain_fan()` of `self`.

OUTPUT:

- a `cone`.

Let $\phi : \Sigma \to \Sigma'$ be a fan morphism, let $\sigma \in \Sigma$, and let $\sigma' = \phi(\sigma)$. Then σ is a **primitive cone corresponding to** σ' if there is no proper face τ of σ such that $\phi(\tau) = \sigma'$.

Primitive cones play an important role for fibration morphisms.

See Also:

`is_fibration()`, `preimage_cones()`, `preimage_fan()`.

EXAMPLES:

Consider a projection of a del Pezzo surface onto the projective line:

```
sage: Sigma = toric_varieties.dP6().fan()
sage: Sigma.rays()
N( 0,  1),
N(-1,  0),
N(-1, -1),
N( 0, -1),
N( 1,  0),
N( 1,  1)
in 2-d lattice N
sage: Sigma_p = toric_varieties.P1().fan()
sage: phi = FanMorphism(matrix([[1], [-1]]), Sigma, Sigma_p)
```

Under this map, one pair of rays is mapped to the origin, one in the positive direction, and one in the negative one. Also three 2-dimensional cones are mapped in the positive direction and three in the negative one, so there are 5 preimage cones corresponding to either of the rays of the codomain fan `Sigma_p`:

```
sage: len(phi.preimage_cones(Cone([(1,)])))
5
```

Yet only rays are primitive:

```
sage: phi.primitive_preimage_cones(Cone([(1,)]))
(1-d cone of Rational polyhedral fan in 2-d lattice N,
 1-d cone of Rational polyhedral fan in 2-d lattice N)
```

Since all primitive cones are mapped onto their images bijectively, we get a fibration:

```
sage: phi.is_fibration()
True
```

But since there are several primitive cones corresponding to the same cone of the codomain fan, this map is not a bundle, even though its index is 1:

```
sage: phi.is_bundle()
False
sage: phi.index()
1
```

relative_star_generators(*domain_cone*)

Return the relative star generators of `domain_cone`.

INPUT:

- `domain_cone` – a cone of the `domain_fan()` of `self`.

OUTPUT:

- `star_generators()` of `domain_cone` viewed as a cone of `preimage_fan()` of `image_cone()` of `domain_cone`.

EXAMPLES:

```
sage: A2 = toric_varieties.A(2).fan()
sage: Bl = A2.subdivide([(1,1)])
sage: f = FanMorphism(identity_matrix(2), Bl, A2)
sage: for c1 in Bl(1):
...       print f.relative_star_generators(c1)
(1-d cone of Rational polyhedral fan in 2-d lattice N,)
(1-d cone of Rational polyhedral fan in 2-d lattice N,)
(2-d cone of Rational polyhedral fan in 2-d lattice N,
 2-d cone of Rational polyhedral fan in 2-d lattice N)
```

CHAPTER

FIVE

POINT COLLECTIONS

Point collections

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

AUTHORS:

- Andrey Novoseltsev (2011-04-25): initial version, based on cone module.
- Andrey Novoseltsev (2012-03-06): additions and doctest changes while switching cones to use point collections.

EXAMPLES:

The idea behind `point collections` is to have a container for points of the same space that

- behaves like a tuple *without significant performance penalty*:

```
sage: c = Cone([(0,0,1), (1,0,1), (0,1,1), (1,1,1)]).rays()
sage: c[1]
N(1, 0, 1)
sage: for point in c: point
N(0, 0, 1)
N(1, 0, 1)
N(0, 1, 1)
N(1, 1, 1)
```

- prints in a convenient way and with clear indication of the ambient space:

```
sage: c
N(0, 0, 1),
N(1, 0, 1),
N(0, 1, 1),
N(1, 1, 1)
in 3-d lattice N
```

- allows (cached) access to alternative representations:

```
sage: c.set()
frozenset([N(0, 1, 1), N(1, 1, 1), N(0, 0, 1), N(1, 0, 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 Δ is a reflexive polytope. Then the same point collection can be linked as

1. vertices of Δ;
2. facet normals of its polar Δ°;
3. ray generators of the face fan of Δ;
4. ray generators of the normal fan of Δ.

If all these objects are in use and, say, a matrix representation was computed for one of them, it becomes available to all others as well, eliminating the need to spend time and memory four times.

class sage.geometry.point_collection.**PointCollection**
Bases: 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 $\mathbf{R}$-basis for the vector space spanned by the points of self.

EXAMPLES:

```
sage: c = Cone([(0,0,1), (1,0,1), (0,1,1), (1,1,1)]).rays()
sage: c.basis()
N(0, 0, 1),
N(1, 0, 1),
N(0, 1, 1)
in 3-d lattice N
```

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)
N+N(0, 0, 1, 0, 0, 1),
N+N(1, 1, 1, 0, 0, 1),
N+N(0, 0, 1, 1, 1, 1),
N+N(1, 1, 1, 1, 1, 1)
in 6-d lattice N+N
```

column_matrix()
Return a matrix whose columns are points of `self`.

OUTPUT:

- a `matrix`.

EXAMPLES:

```
sage: c = Cone([(0,0,1), (1,0,1), (0,1,1), (1,1,1)]).rays()
sage: c.column_matrix()
[0 1 0 1]
[0 0 1 1]
[1 1 1 1]
```

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), (1,1,1)]).rays()
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()` M of `self` has a `dual()` method), the dual module is returned. Otherwise, R^n is returned, where n is the dimension of M and R is its base ring.

EXAMPLES:

```
sage: c = Cone([(0,0,1), (1,0,1), (0,1,1), (1,1,1)]).rays()
sage: c.dual_module()
3-d lattice M
```

index(**args*)

Return the index of the first occurrence of `point` in `self`.

INPUT:

- `point` – a point of `self`;
- `start` – (optional) an integer, if given, the search will start at this position;
- `stop` – (optional) an integer, if given, the search will stop at this position.

OUTPUT:

- an integer if `point` is in `self[start:stop]`, otherwise a `ValueError` exception is raised.

EXAMPLES:

```
sage: c = Cone([(0,0,1), (1,0,1), (0,1,1), (1,1,1)]).rays()
sage: c.index((0,1,1))
Traceback (most recent call last):
...
ValueError: tuple.index(x): x not in tuple
```

Note that this was not a mistake: the *tuple* `(0,1,1)` is *not* a point of `c`! We need to pass actual element of the ambient module of `c` to get their indices:

```
sage: N = c.module()
sage: c.index(N(0,1,1))
2
sage: c[2]
N(0, 1, 1)
```

matrix()

Return a matrix whose rows are points of `self`.

OUTPUT:

- a `matrix`.

EXAMPLES:

```
sage: c = Cone([(0,0,1), (1,0,1), (0,1,1), (1,1,1)]).rays()
sage: c.matrix()
[0 0 1]
[1 0 1]
[0 1 1]
[1 1 1]
```

module()

Return the ambient module of `self`.

OUTPUT:

- a `module`.

EXAMPLES:

```
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:

```
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 outpus are identical, separators are only inserted in the LaTeX mode:

```
sage: latex(c)
\left(\begin{array}{r|r|r|r}
0 & 1 & 0 & 1 \\
0 & 0 & 1 & 1 \\
1 & 1 & 1 & 1
\end{array}\right)_{N}
```

Since this is a static method, you can call it for the class directly:

```
sage: from sage.geometry.point_collection import PointCollection
sage: PointCollection.output_format("default")
sage: c
N(0, 0, 1),
N(1, 0, 1),
N(0, 1, 1),
N(1, 1, 1)
in 3-d lattice N
```

set()

Return points of `self` as a `frozenset`.

OUTPUT:

- a `frozenset`.

EXAMPLES:

```
sage: c = Cone([(0,0,1), (1,0,1), (0,1,1), (1,1,1)]).rays()
sage: c.set()
frozenset([N(0, 1, 1), N(1, 1, 1), N(0, 0, 1), N(1, 0, 1)])
```

sage.geometry.point_collection.**is_PointCollection**(*x*)

Check if x is a point collection.

INPUT:

- x – anything.

OUTPUT:

- True if x is a point collection and False otherwise.

EXAMPLES:

```
sage: from sage.geometry.point_collection import 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
```

CHAPTER

SIX

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: print 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: print fan.plot()
Graphics object consisting of 19 graphics primitives
sage: toric_plotter.reset_options()
sage: toric_plotter.options("show_rays")
True
sage: print 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()
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: print 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.

TESTS:

```
sage: from sage.geometry.toric_plotter import ToricPlotter
sage: tp = ToricPlotter(dict(), 2)
sage: print tp.show_lattice
None
sage: tp.adjust_options()
sage: print tp.show_lattice
True
```

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:

```
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:

```
sage: from sage.geometry.toric_plotter import ToricPlotter
sage: tp = ToricPlotter(dict(), 2, [(3,4)])
sage: print 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: from sage.geometry.toric_plotter import ToricPlotter
sage: tp = ToricPlotter(dict(), 2)
sage: print tp.plot_labels("u", [(1.5,0)])
Graphics object consisting of 1 graphics primitive
```

plot_lattice()

Plot the lattice (i.e. its points in the cut-off bounds of `self`).

OUTPUT:

- a plot.

EXAMPLES:

```
sage: from sage.geometry.toric_plotter import ToricPlotter
sage: tp = ToricPlotter(dict(), 2)
sage: print tp.plot_lattice()
Graphics object consisting of 1 graphics primitive
```

plot_points(*points*)

Plot given `points`.

INPUT:

- `points` – a list of points.

OUTPUT:

- a plot.

EXAMPLES:

```
sage: from sage.geometry.toric_plotter import ToricPlotter
sage: tp = ToricPlotter(dict(), 2)
sage: print tp.plot_points([(1,0), (0,1)])
Graphics object consisting of 1 graphics primitive
```

plot_ray_labels()

Plot ray labels.

Usually ray labels are plotted together with rays, but in some cases it is desirable to output them separately.

Ray generators must be specified during construction or using `set_rays()` before calling this method.

OUTPUT:

- a plot.

EXAMPLES:

```
sage: from sage.geometry.toric_plotter import ToricPlotter
sage: tp = ToricPlotter(dict(), 2, [(3,4)])
sage: print tp.plot_ray_labels()
Graphics object consisting of 1 graphics primitive
```

plot_rays()

Plot rays and their labels.

Ray generators must be specified during construction or using `set_rays()` before calling this method.

OUTPUT:

- a plot.

EXAMPLES:

```
sage: from sage.geometry.toric_plotter import ToricPlotter
sage: tp = ToricPlotter(dict(), 2, [(3,4)])
sage: print tp.plot_rays()
Graphics object consisting of 2 graphics primitives
```

plot_walls (*walls*)

Plot `walls`, i.e. 2-d cones, and their labels.

Ray generators must be specified during construction or using `set_rays()` before calling this method and these specified ray generators will be used in conjunction with `ambient_ray_indices()` of `walls`.

INPUT:

- `walls` – a list of 2-d cones.

OUTPUT:

- a plot.

EXAMPLES:

```
sage: quadrant = Cone([(1,0), (0,1)])
sage: from sage.geometry.toric_plotter import ToricPlotter
sage: tp = ToricPlotter(dict(), 2, quadrant.rays())
sage: print tp.plot_walls([quadrant])
Graphics object consisting of 2 graphics primitives
```

Let's also check that the truncating polyhedron is functioning correctly:

```
sage: tp = ToricPlotter({"mode": "box"}, 2, quadrant.rays())
sage: print 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
sage: tp = ToricPlotter(dict(), 2)
sage: tp.adjust_options()
sage: print tp.plot_rays()
Traceback (most recent call last):
...
AttributeError: 'ToricPlotter' object has no attribute 'rays'
sage: tp.set_rays([(0,1)])
sage: print 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 label was a list of n entries, it is returned without changes. If label is None, a list of n None's is returned. If label is a string, a list of strings of the form "$label_i$" is returned, where i ranges over index_set. (If math_mode=False, the form "label_i" is used instead.) If n=1, there is no subscript added, unless index_set was specified explicitly.

EXAMPLES:

```
sage: from sage.geometry.toric_plotter import label_list
sage: label_list("u", 3, False)
['u_0', 'u_1', 'u_2']
sage: label_list("u", 3, True)
```

```
['$u_{0}$', '$u_{1}$', '$u_{2}$']
sage: label_list("u", 1, True)
['$u$']
```

sage.geometry.toric_plotter.**options**(*option=None*, ***kwds*)

Get or set options for plots of toric geometry objects.

Note: This function provides access to global default options. Any of these options can be overridden by passing them directly to plotting functions. See also `reset_options()`.

INPUT:

- None;

OR:

- `option` – 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 `option` argument was given, the current value of `option`;
- if other keyword arguments were given, none.

Name Conventions

To clearly distinguish parts of toric plots, in options and methods we use the following name conventions:

Generator A primitive integral vector generating a 1-dimensional cone, plotted as an arrow from the origin (or a line, if the head of the arrow is beyond cut-off bounds for the plot).

Ray A 1-dimensional cone, plotted as a line from the origin to the cut-off bounds for the plot.

Wall A 2-dimensional cone, plotted as a region between rays (in the above sense). Its exact shape depends on the plotting mode (see below).

Chamber A 3-dimensional cone, plotting is not implemented yet.

Plotting Modes

A plotting mode mostly determines the shape of the cut-off region (which is always relevant for toric plots except for trivial objects consisting of the origin only). The following options are available:

Box The cut-off region is a box with edges parallel to coordinate axes.

Generators The cut-off region is determined by primitive integral generators of rays. Note that this notion is well-defined only for rays and walls, in particular you should plot the lattice on your own (`plot_lattice()` will use box mode which is likely to be unsuitable). While this method may not be suitable for general fans, it is quite natural for fans of `CPR-Fano toric varieties. <sage.schemes.toric.fano_variety.CPRFanoToricVariety_field`

Round The cut-off regions is a sphere centered at the origin.

Available Options

Default values for the following options can be set using this function:

- `mode` – "box", "generators", or "round", see above for descriptions;

- `show_lattice` – boolean, whether to show lattice points in the cut-off region or not;
- `show_rays` – boolean, whether to show rays or not;
- `show_generators` – boolean, whether to show rays or not;
- `show_walls` – boolean, whether to show rays or not;
- `generator_color` – a color for generators;
- `label_color` – a color for labels;
- `point_color` – a color for lattice points;
- `ray_color` – a color for rays, a list of colors (one for each ray), or the string "rainbow";
- `wall_color` – a color for walls, a list of colors (one for each wall), or the string "rainbow";
- `wall_alpha` – a number between 0 and 1, the alpha-value for walls (determining their transparency);
- `point_size` – 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;
- `wall_label` – a string or a list of strings used for wall labels;
- `radius` – a positive number, the radius of the cut-off region for "round" mode;
- `xmin`, `xmax`, `ymin`, `ymax`, `zmin`, `zmax` – numbers determining the cut-off region for "box" mode. Note that you cannot exclude the origin - if you try to do so, bounds will be automatically expanded to include it;
- `lattice_filter` – a callable, taking as an argument a lattice point and returning `True` if this point should be included on the plot (useful, e.g. for plotting sublattices);
- `wall_zorder`, `ray_zorder`, `generator_zorder`, `point_zorder`, `label_zorder` – integers, z-orders for different classes of objects. By default all values are negative, so that you can add other graphic objects on top of a toric plot. You may need to adjust these parameters if you want to put a toric plot on top of something else or if you want to overlap several toric plots.

You can see the current default value of any options by typing, e.g.

```
sage: toric_plotter.options("show_rays")
True
```

If the default value is `None`, it means that the actual default is determined later based on the known options. Note, that not all options can be determined in such a way, so you should not set options to `None` unless it was its original state. (You can always revert to this "original state" using `reset_options()`.)

EXAMPLES:

The following line will make all subsequent toric plotting commands to draw "rainbows" from walls:

```
sage: toric_plotter.options(wall_color="rainbow")
```

If you prefer a less colorful output (e.g. if you need black-and-white illustrations for a paper), you can use something like this:

```
sage: toric_plotter.options(wall_color="grey")
```

sage.geometry.toric_plotter.**reset_options**()

Reset options for plots of toric geometry objects.

OUTPUT:

- none.

EXAMPLES:

```
sage: toric_plotter.options("show_rays")
True
sage: toric_plotter.options(show_rays=False)
sage: toric_plotter.options("show_rays")
False
```

Now all toric plots will not show rays, unless explicitly requested. If you want to go back to "default defaults", use this method:

```
sage: toric_plotter.reset_options()
sage: toric_plotter.options("show_rays")
True
```

sage.geometry.toric_plotter.**sector**(*ray1*, *ray2*, ***extra_options*)

Plot a sector between `ray1` and `ray2` centered at the origin.

Note: This function was intended for plotting strictly convex cones, so it plots the smaller sector between `ray1` and `ray2` and, therefore, they cannot be opposite. If you do want to use this function for bigger regions, split them into several parts.

Note: As of version 4.6 Sage does not have a graphic primitive for sectors in 3-dimensional space, so this function will actually approximate them using polygons (the number of vertices used depends on the angle between rays).

INPUT:

- `ray1`, `ray2` – rays in 2- or 3-dimensional space of the same length;
- `extra_options` – a dictionary of options that should be passed to lower level plotting functions.

OUTPUT:

- a plot.

EXAMPLES:

```
sage: from sage.geometry.toric_plotter import sector
sage: print sector((1,0), (0,1))
Graphics object consisting of 1 graphics primitive
sage: print sector((3,2,1), (1,2,3))
Graphics3d Object
```

CHAPTER

SEVEN

GROEBNER FANS

Sage provides much of the functionality of gfan, which is a software package whose main function is to enumerate all reduced Groebner bases of a polynomial ideal. The reduced Groebner bases yield the maximal cones in the Groebner fan of the ideal. Several subcomputations can be issued and additional tools are included. Among these the highlights are:

- Commands for computing tropical varieties.
- Interactive walks in the Groebner fan of an ideal.
- Commands for graphical renderings of Groebner fans and monomial ideals.

AUTHORS:

- Anders Nedergaard Jensen: Wrote the gfan C++ program, which implements algorithms many of which were invented by Jensen, Komei Fukuda, and Rekha Thomas. All the underlying hard work of the Groebner fans functionality of Sage depends on this C++ program.
- William Stein (2006-04-20): Wrote first version of the Sage code for working with Groebner fans.
- Tristram Bogart: the design of the Sage interface to gfan is joint work with Tristram Bogart, who also supplied numerous examples.
- Marshall Hampton (2008-03-25): Rewrote various functions to use gfan-0.3. This is still a work in progress, comments are appreciated on sage-devel@googlegroups.com (or personally at hamptonio@gmail.com).

EXAMPLES:

```
sage: x,y = QQ['x,y'].gens()
sage: i = ideal(x^2 - y^2 + 1)
sage: g = i.groebner_fan()
sage: g.reduced_groebner_bases()
[[x^2 - y^2 + 1], [-x^2 + y^2 - 1]]
```

TESTS:

```
sage: x,y = QQ['x,y'].gens()
sage: i = ideal(x^2 - y^2 + 1)
sage: g = i.groebner_fan()
sage: g == loads(dumps(g))
True
```

REFERENCES:

- Anders N. Jensen; Gfan, a software system for Groebner fans; available at http://www.math.tu-berlin.de/~jensen/software/gfan/gfan.html

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
```

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()

Computes and returns 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*z + 6*y*z - z^2, x*y + 6*x*z + y*z - z^2, y^2 + x*z + y*z)
sage: gf = i1.groebner_fan()
sage: gf.characteristic()
0
```

dimension_of_homogeneity_space()

Return the dimension of the homogeneity space.

EXAMPLES:

```
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*)

Returns 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{{\ny^9-1-y+3*y^3-3*y^6,\nx+1-y^3}\n,\n{\ny^3-1-x,\nx^3-y}\n,\n{\ny-x^3,\nx^9-1-x}\
```

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 F
```

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()

Returns the mixed volume of the generators of this ideal (i.e. 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 Groebner bases.

EXAMPLES:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: G = R.ideal([y^3 - x^2, y^2 - 13*x]).groebner_fan()
sage: G.number_of_reduced_groebner_bases()
3
```

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
```

polyhedralfan()

Returns 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 - z, y - z^11, x - z^9]
sage: X[0].ideal()
Ideal (z^15 - z, y - z^11, x - z^9) of Multivariate Polynomial Ring in x, y, z over Rational
sage: X[:5]
[[z^15 - z, y - z^11, x - z^9],
[-y + z^11, y*z^4 - z, y^2 - z^8, x - z^9],
[-y^2 + z^8, y*z^4 - z, y^2*z^3 - y, y^3 - z^5, x - y^2*z],
[-y^3 + z^5, y*z^4 - z, y^2*z^3 - y, y^4 - z^2, x - y^2*z],
[-y^4 + z^2, y^6*z - y, y^9 - z, x - y^2*z]]
sage: R3.<x,y,z> = PolynomialRing(GF(2477),3)
sage: gf = R3.ideal([300*x^3-y,y^2-z,z^2-12]).groebner_fan()
sage: gf.reduced_groebner_bases()
[[z^2 - 12, y^2 - z, x^3 + 933*y],
[-y^2 + z, y^4 - 12, x^3 + 933*y],
[z^2 - 12, -300*x^3 + y, x^6 - 1062*z],
[-828*x^6 + z, -300*x^3 + y, x^12 + 200]]
```

render(*file=None*, *larger=False*, *shift=0*, *rgbcolor=(0, 0, 0)*, *polyfill=<function max_degree at 0x7fb30a1b62a8>*, *scale_colors=True*)

Render a Groebner fan as sage graphics or save as an xfig file.

More precisely, the output is a drawing of the Groebner fan intersected with a triangle. The corners of the triangle are (1,0,0) to the right, (0,1,0) to the left and (0,0,1) at the top. If there are more than three variables in the ring we extend these coordinates with zeros.

INPUT:

- `file` - a filename if you prefer the output saved to a file. This will be in xfig format.
- `shift` - shift the positions of the variables in the drawing. For example, with shift=1, the corners will be b (right), c (left), and d (top). The shifting is done modulo the number of variables in the polynomial ring. The default is 0.
- `larger` - bool (default: False); if True, make the triangle larger so that the shape of of the Groebner region appears. Affects the xfig file but probably not the sage graphics (?)
- `rgbcolor` - This will not affect the saved xfig file, only the sage graphics produced.
- `polyfill` - Whether or not to fill the cones with a color determined by the highest degree in each reduced Groebner basis for that cone.
- `scale_colors` - if True, this will normalize color values to try to maximize the range

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: G = R.ideal([y^3 - x^2, y^2 - 13*x,z]).groebner_fan()
sage: test_render = G.render()

sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: G = R.ideal([x^2*y - z, y^2*z - x, z^2*x - y]).groebner_fan()
sage: test_render = G.render(larger=True)
```

TESTS:

Testing the case where the number of generators is < 3. Currently, this should raise a `NotImplementedError` error.

```
sage: R.<x,y> = PolynomialRing(QQ, 2)
sage: R.ideal([y^3 - x^2, y^2 - 13*x]).groebner_fan().render()
Traceback (most recent call last):
...
NotImplementedError
```

render3d(*verbose=False*)

For a Groebner fan of an ideal in a ring with four variables, this function intersects the fan with the standard simplex perpendicular to (1,1,1,1), creating a 3d polytope, which is then projected into 3 dimensions. The edges of this projected polytope are returned as lines.

EXAMPLES:

```
sage: R4.<w,x,y,z> = PolynomialRing(QQ,4)
sage: gf = R4.ideal([w^2-x,x^2-y,y^2-z,z^2-x]).groebner_fan()
sage: three_d = gf.render3d()
```

TESTS:

Now test the case where the number of generators is not 4. Currently, this should raise a `NotImplementedError` error.

```
sage: P.<a,b,c> = PolynomialRing(QQ, 3, order="lex")
sage: sage.rings.ideal.Katsura(P, 3).groebner_fan().render3d()
Traceback (most recent call last):
...
NotImplementedError
```

ring()

Return the multivariate polynomial ring.

EXAMPLES:

```
sage: R.<x1,x2> = PolynomialRing(QQ,2)
sage: gf = R.ideal([x1^3-x2,x2^3-x1-2]).groebner_fan()
sage: gf.ring()
Multivariate Polynomial Ring in x1, x2 over Rational Field
```

tropical_basis(*check=True*, *verbose=False*)

Return a tropical basis for the tropical curve associated to this ideal.

INPUT:

- `check` - bool (default: True); if True raises a ValueError exception if this ideal does not define a tropical curve (i.e., the condition that R/I has dimension equal to 1 + the dimension of the homogeneity space is not satisfied).

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3, order='lex')
sage: G = R.ideal([y^3-3*x^2, z^3-x-y-2*y^3+2*x^2]).groebner_fan()
sage: G
Groebner fan of the ideal:
Ideal (-3*x^2 + y^3, 2*x^2 - x - 2*y^3 - y + z^3) of Multivariate Polynomial Ring in x, y, z
sage: G.tropical_basis()
[-3*x^2 + y^3, 2*x^2 - x - 2*y^3 - y + z^3, 3/4*x + y^3 + 3/4*y - 3/4*z^3]
```

tropical_intersection(*parameters*=[ ], *symmetry_generators*=[ ], **args*, ***kwds*)

Returns 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:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: I = R.ideal(x*z + 6*y*z - z^2, x*y + 6*x*z + y*z - z^2, y^2 + x*z + y*z)
sage: gf = I.groebner_fan()
sage: pf = gf.tropical_intersection()
sage: pf.rays()
[[-2, 1, 1]]

sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: f1 = x*y*z - 1
sage: f2 = f1*(x^2 + y^2 + z^2)
sage: f3 = f2*(x + y + z - 1)
sage: I = R.ideal([f1,f2,f3])
sage: gf = I.groebner_fan()
sage: pf = gf.tropical_intersection(symmetry_generators = '(1,2,0),(1,0,2)')
sage: pf.rays()
[[-2, 1, 1], [1, -2, 1], [1, 1, -2]]

sage: R.<x,y,z> = QQ[]
sage: I = R.ideal([(x+y+z)^2-1,(x+y+z)-x,(x+y+z)-3])
sage: GF = I.groebner_fan()
sage: TI = GF.tropical_intersection()
sage: TI.rays()
[[-1, 0, 0], [0, -1, -1], [1, 1, 1]]
sage: GF = I.groebner_fan()
sage: TI = GF.tropical_intersection(parameters=[y])
sage: TI.rays()
[[-1, 0, 0]]
```

weight_vectors()

Returns the weight vectors corresponding to the reduced Groebner bases.

EXAMPLES:

```
sage: r3.<x,y,z> = PolynomialRing(QQ,3)
sage: g = r3.ideal([x^3+y,y^3-z,z^2-x]).groebner_fan()
sage: g.weight_vectors()
[(3, 7, 1), (5, 1, 2), (7, 1, 4), (5, 1, 4), (1, 1, 1), (1, 4, 8), (1, 4, 10)]
sage: r4.<x,y,z,w> = PolynomialRing(QQ,4)
sage: g4 = r4.ideal([x^3+y,y^3-z,z^2-x,z^3 - w]).groebner_fan()
sage: len(g4.weight_vectors())
23
```

class sage.rings.polynomial.groebner_fan.**InitialForm**(*cone*, *rays*, *initial_forms*)

Bases: sage.structure.sage_object.SageObject

A system of initial forms from a polynomial system. To each form is associated a cone and a list of polynomials (the initial form system itself).

This class is intended for internal use inside of the TropicalPrevariety class.

EXAMPLES:

```
sage: from sage.rings.polynomial.groebner_fan import InitialForm
sage: R.<x,y> = QQ[]
sage: inform = InitialForm([0], [[-1, 0]], [y^2 - 1, y^2 - 2, y^2 - 3])
sage: inform._cone
[0]
```

cone()

The cone associated with the initial form system.

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: I = R.ideal([(x+y)^2-1,(x+y)^2-2,(x+y)^2-3])
sage: GF = I.groebner_fan()
sage: PF = GF.tropical_intersection()
sage: pfi0 = PF.initial_form_systems()[0]
sage: pfi0.cone()
[0]
```

initial_forms()

The initial forms (polynomials).

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: I = R.ideal([(x+y)^2-1,(x+y)^2-2,(x+y)^2-3])
sage: GF = I.groebner_fan()
sage: PF = GF.tropical_intersection()
sage: pfi0 = PF.initial_form_systems()[0]
sage: pfi0.initial_forms()
[y^2 - 1, y^2 - 2, y^2 - 3]
```

internal_ray()

A ray internal to the cone associated with the initial form system.

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: I = R.ideal([(x+y)^2-1,(x+y)^2-2,(x+y)^2-3])
sage: GF = I.groebner_fan()
sage: PF = GF.tropical_intersection()
sage: pfi0 = PF.initial_form_systems()[0]
sage: pfi0.internal_ray()
(-1, 0)
```

rays()

The rays of the cone associated with the initial form system.

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: I = R.ideal([(x+y)^2-1,(x+y)^2-2,(x+y)^2-3])
sage: GF = I.groebner_fan()
sage: PF = GF.tropical_intersection()
sage: pfi0 = PF.initial_form_systems()[0]
sage: pfi0.rays()
[[-1, 0]]
```

class sage.rings.polynomial.groebner_fan.**PolyhedralCone**(*gfan_polyhedral_cone*, *ring=Rational Field*)

Bases: sage.structure.sage_object.SageObject

Converts polymake/gfan data on a polyhedral cone into a sage class. Currently (18-03-2008) needs a lot of work.

EXAMPLES:

```
sage: R3.<x,y,z> = PolynomialRing(QQ,3)
sage: gf = R3.ideal([x^8-y^4,y^4-z^2,z^2-2]).groebner_fan()
sage: a = gf[0].groebner_cone()
sage: a.facets()
[[0, 0, 1], [0, 1, 0], [1, 0, 0]]
```

ambient_dim()

Returns the ambient dimension of the Groebner cone.

EXAMPLES:

```
sage: R3.<x,y,z> = PolynomialRing(QQ,3)
sage: gf = R3.ideal([x^8-y^4,y^4-z^2,z^2-2]).groebner_fan()
sage: a = gf[0].groebner_cone()
sage: a.ambient_dim()
3
```

dim()

Returns the dimension of the Groebner cone.

EXAMPLES:

```
sage: R3.<x,y,z> = PolynomialRing(QQ,3)
sage: gf = R3.ideal([x^8-y^4,y^4-z^2,z^2-2]).groebner_fan()
sage: a = gf[0].groebner_cone()
sage: a.dim()
3
```

facets()

Returns the inward facet normals of the Groebner cone.

EXAMPLES:

```
sage: R3.<x,y,z> = PolynomialRing(QQ,3)
sage: gf = R3.ideal([x^8-y^4,y^4-z^2,z^2-2]).groebner_fan()
sage: a = gf[0].groebner_cone()
sage: a.facets()
[[0, 0, 1], [0, 1, 0], [1, 0, 0]]
```

lineality_dim()

Returns 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()

Returns a point in the relative interior of the Groebner cone.

EXAMPLES:

```
sage: R3.<x,y,z> = PolynomialRing(QQ,3)
sage: gf = R3.ideal([x^8-y^4,y^4-z^2,z^2-2]).groebner_fan()
sage: a = gf[0].groebner_cone()
sage: a.relative_interior_point()
[1, 1, 1]
```

class sage.rings.polynomial.groebner_fan.**PolyhedralFan**(*gfan_polyhedral_fan*, *parameter_indices*=[])

Bases: sage.structure.sage_object.SageObject

Converts 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()

Returns 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:

```
sage: R.<x,y,z> = QQ[]
sage: f = 1+x+y+x*y
sage: I = R.ideal([f+z*f, 2*f+z*f, 3*f+z^2*f])
sage: GF = I.groebner_fan()
sage: PF = GF.tropical_intersection()
sage: PF.cones()
{1: [[0], [1], [2], [3], [4], [5]], 2: [[0, 1], [0, 2], [0, 3], [0, 4], [1, 2], [1, 3], [2,
```

dim()

Returns the dimension of the Groebner fan.

EXAMPLES:

```
sage: R3.<x,y,z> = PolynomialRing(QQ,3)
sage: gf = R3.ideal([x^8-y^4,y^4-z^2,z^2-2]).groebner_fan()
sage: a = gf.polyhedralfan()
sage: a.dim()
3
```

f_vector()

The f-vector of the fan.

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: f = 1+x+y+x*y
sage: I = R.ideal([f+z*f, 2*f+z*f, 3*f+z^2*f])
sage: GF = I.groebner_fan()
sage: PF = GF.tropical_intersection()
sage: PF.f_vector()
[1, 6, 12]
```

is_simplicial()

Whether the fan is simplicial or not.

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: f = 1+x+y+x*y
sage: I = R.ideal([f+z*f, 2*f+z*f, 3*f+z^2*f])
sage: GF = I.groebner_fan()
sage: PF = GF.tropical_intersection()
sage: PF.is_simplicial()
True
```

lineality_dim()

Returns the lineality dimension of the fan. This is the dimension of the largest subspace contained in the fan.

EXAMPLES:

```
sage: R3.<x,y,z> = PolynomialRing(QQ,3)
sage: gf = R3.ideal([x^8-y^4,y^4-z^2,z^2-2]).groebner_fan()
sage: a = gf.polyhedralfan()
sage: a.lineality_dim()
0
```

maximal_cones()

A dictionary of the maximal cones in which the keys are the cone dimensions. For each dimension, the value is a list of the maximal cones, where each element consists of a list of ray indices.

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: f = 1+x+y+x*y
sage: I = R.ideal([f+z*f, 2*f+z*f, 3*f+z^2*f])
sage: GF = I.groebner_fan()
sage: PF = GF.tropical_intersection()
sage: PF.maximal_cones()
{2: [[0, 1], [0, 2], [0, 3], [0, 4], [1, 2], [1, 3], [2, 4], [3, 4], [1, 5], [2, 5], [3, 5],
```

rays()

A list of rays of the polyhedral fan.

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: i2 = ideal(x*z + 6*y*z - z^2, x*y + 6*x*z + y*z - z^2, y^2 + x*z + y*z)
sage: gf2 = i2.groebner_fan(verbose = False)
sage: pf = gf2.polyhedralfan()
sage: pf.rays()
```

```
[[-1, 0, 1], [-1, 1, 0], [1, -2, 1], [1, 1, -2], [2, -1, -1]]
```

to_RationalPolyhedralFan()

Converts to the RationalPolyhedralFan class, which is more actively maintained. While the information in each class is essentially the same, the methods and implementation are different.

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: f = 1+x+y+x*y
sage: I = R.ideal([f+z*f, 2*f+z*f, 3*f+z^2*f])
sage: GF = I.groebner_fan()
sage: PF = GF.tropical_intersection()
sage: fan = PF.to_RationalPolyhedralFan()
sage: [tuple(q.facet_normals()) for q in fan]
[(M(0, -1, 0), M(-1, 0, 0)), (M(0, 0, -1), M(-1, 0, 0)), (M(0, 0, 1), M(-1, 0, 0)), (M(0, 1,
```

Here we use the RationalPolyhedralFan's Gale_transform method on a tropical prevariety.

```
sage: fan.Gale_transform()
[ 1  0  0  0  0  1 -2]
[ 0  1  0  0  1  0 -2]
[ 0  0  1  1  0  0 -2]
```

class sage.rings.polynomial.groebner_fan.**ReducedGroebnerBasis**(*groebner_fan*, *gens*, *gfan_gens*)

Bases: sage.structure.sage_object.SageObject, list

A class for representing reduced Groebner bases as produced by gfan.

INPUT:

- groebner_fan - a GroebnerFan object from an ideal
- gens - the generators of the ideal
- gfan_gens - the generators as a gfan string

EXAMPLES:

```
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:

```
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: G = R.ideal([y^3 - x^2, y^2 - 13*x]).groebner_fan()
sage: poly_cone = G[1].groebner_cone()
sage: poly_cone.facets()
```

```
[[-1, 2], [1, -1]]
sage: [g.groebner_cone().facets() for g in G]
[[[0, 1], [1, -2]], [[-1, 2], [1, -1]], [[-1, 1], [1, 0]]]
sage: G[1].groebner_cone(restrict=True).facets()
[[-1, 2], [1, -1]]
```

ideal()

Return the ideal generated by this basis.

EXAMPLES:

```
sage: R.<x,y,z> = PolynomialRing(QQ,3)
sage: G = R.ideal([x - z^3, y^2 - 13*x]).groebner_fan()
sage: G[0].ideal()
Ideal (-13*z^3 + y^2, -z^3 + x) of Multivariate Polynomial Ring in x, y, z over Rational Fie
```

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=[]*)

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()

Returns 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}'
```

sage.rings.polynomial.groebner_fan.**max_degree**(*list_of_polys*)

Computes the maximum degree of a list of polynomials

EXAMPLES:

```
sage: from sage.rings.polynomial.groebner_fan import max_degree
sage: R.<x,y> = PolynomialRing(QQ,2)
sage: p_list = [x^2-y,x*y^10-x]
sage: max_degree(p_list)
11.0
```

sage.rings.polynomial.groebner_fan.**prefix_check**(*str_list*)

Checks if any strings in a list are prefixes of another string in the list.

EXAMPLES:

```
sage: from sage.rings.polynomial.groebner_fan import prefix_check
sage: prefix_check(['z1','z1z1'])
False
sage: prefix_check(['z1','zz1'])
True
```

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*)

Returns 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)]
```

CHAPTER

EIGHT

LATTICE AND REFLEXIVE POLYTOPES

This module provides tools for work with lattice and reflexive polytopes. A *convex polytope* is the convex hull of finitely many points in $\mathbf{R}^n$. The dimension n of a polytope is the smallest n such that the polytope can be embedded in $\mathbf{R}^n$.

A *lattice polytope* is a polytope whose vertices all have integer coordinates.

If L is a lattice polytope, the dual polytope of L is

$$\{y \in \mathbf{Z}^n : x \cdot y \geq -1 \text{ all } x \in L\}$$

A *reflexive polytope* is a lattice polytope, such that its polar is also a lattice polytope, i.e. it is bounded and has vertices with integer coordinates.

This Sage module uses Package for Analyzing Lattice Polytopes (PALP), which is a program written in C by Maximilian Kreuzer and Harald Skarke, which is freely available under the GNU license terms at http://hep.itp.tuwien.ac.at/~kreuzer/CY/. Moreover, PALP is included standard with Sage.

PALP is described in the paper Arxiv math.SC/0204356. Its distribution also contains the application nef.x, which was created by Erwin Riegler and computes nef-partitions and Hodge data for toric complete intersections.

ACKNOWLEDGMENT: polytope.py module written by William Stein was used as an example of organizing an interface between an external program and Sage. William Stein also helped Andrey Novoseltsev with debugging and tuning of this module.

Robert Bradshaw helped Andrey Novoseltsev to realize plot3d function.

Note: IMPORTANT: PALP requires some parameters to be determined during compilation time, i.e., the maximum dimension of polytopes, the maximum number of points, etc. These limitations may lead to errors during calls to different functions of these module. Currently, a ValueError exception will be raised if the output of poly.x or nef.x is empty or contains the exclamation mark. The error message will contain the exact command that caused an error, the description and vertices of the polytope, and the obtained output.

Data obtained from PALP and some other data is cached and most returned values are immutable. In particular, you cannot change the vertices of the polytope or their order after creation of the polytope.

If you are going to work with large sets of data, take a look at `all_*` functions in this module. They precompute different data for sequences of polynomials with a few runs of external programs. This can significantly affect the time of future computations. You can also use dump/load, but not all data will be stored (currently only faces and the number of their internal and boundary points are stored, in addition to polytope vertices and its polar).

AUTHORS:

- Andrey Novoseltsev (2007-01-11): initial version
- Andrey Novoseltsev (2007-01-15): `all_*` functions

- Andrey Novoseltsev (2008-04-01): second version, including:
  - dual nef-partitions and necessary convex_hull and minkowski_sum
  - built-in sequences of 2- and 3-dimensional reflexive polytopes
  - plot3d, skeleton_show
- Andrey Novoseltsev (2009-08-26): dropped maximal dimension requirement
- Andrey Novoseltsev (2010-12-15): new version of nef-partitions
- Andrey Novoseltsev (2013-09-30): switch to PointCollection.
- Maximilian Kreuzer and Harald Skarke: authors of PALP (which was also used to obtain the list of 3-dimensional reflexive polytopes)
- Erwin Riegler: the author of nef.x

sage.geometry.lattice_polytope.**LatticePolytope**(*data*, *desc=None*, *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_matrix()` for the file format;
- `desc` – DEPRECATED (default: "A lattice polytope") a string description of the polytope;
- **`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:

```
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_pc()
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1)
in 3-d lattice M
```

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

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

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

```
sage: points.append((0,0,0))
sage: p = LatticePolytope(points)
sage: p.nvertices()
6
```

You can suppress vertex computation for speed but this can lead to mistakes:

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

Given points must be in the lattice:

```
sage: LatticePolytope(matrix([1/2, 3/2]))
Traceback (most recent call last):
...
ValueError: points
[(1/2), (3/2)]
are not in 1-d lattice M!
```

But it is OK to create polytopes of non-maximal dimension:

```
sage: p = LatticePolytope([(1,0,0), (0,1,0), (0,0,0),
...           (-1,0,0), (0,-1,0), (0,0,0), (0,0,0)])
sage: p
2-d lattice polytope in 3-d lattice M
sage: p.vertices_pc()
M( 1,  0, 0),
M( 0,  1, 0),
M(-1,  0, 0),
M( 0, -1, 0)
in 3-d lattice M
```

An empty lattice polytope can be considered as well:

```
sage: p = LatticePolytope([], lattice=ToricLattice(3).dual()); p
-1-d lattice polytope in 3-d lattice M
sage: p.ambient_dim()
3
sage: p.npoints()
0
sage: p.nfacets()
0
sage: p.points_pc()
Empty collection
in 3-d lattice M
sage: p.faces()
[]
```

class sage.geometry.lattice_polytope.**LatticePolytopeClass**(*points*, *compute_vertices*)

Bases: sage.structure.sage_object.SageObject, _abcoll.Hashable

Construct a lattice polytope from prepared data.

In most cases you should use `LatticePolytope()` for constructing polytopes.

INPUT:

- points – a PointCollection;
- compute_vertices – boolean.

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_pc()
M( 1,  0),
M( 0,  1),
M(-1,  0),
M( 0, -1)
in 2-d lattice M
sage: o.affine_transform(2).vertices_pc()
M( 2,  0),
M( 0,  2),
M(-2,  0),
M( 0, -2)
in 2-d lattice M
sage: o.affine_transform(1,1).vertices_pc()
M(2, 1),
M(1, 2),
M(0, 1),
M(1, 0)
in 2-d lattice M
sage: o.affine_transform(b=1).vertices_pc()
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_pc()
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_pc()
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_pc()
M(0,  1),
M(1,  1),
M(0, -2),
M(1, -2)
in 2-d lattice M
```

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

```
sage: o.affine_transform(a)
Traceback (most recent call last):
...
ValueError: points
[(1/2, 0), (0, 3/2), (-1/2, 0), (0, -3/2)]
are not in 2-d lattice M!
```

ambient_dim()

Return the dimension of the ambient space of this polytope.

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

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.ambient_dim()
3
```

dim()

Return the dimension of this polytope.

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

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

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

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

This function CAN be used for polytopes whose dimension is smaller than the dimension of the ambient space. In this case distances are computed in the affine subspace spanned by the polytope and if the point is given, it must be in this subspace.

EXAMPLES: The matrix of distances for a 3-dimensional octahedron:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.distances()
[0 0 2 2 2 0 1]
```

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

Distances from facets to the point (1,2,3):

```
sage: o.distances([1,2,3])
(1, 3, 5, 7, -5, -3, -1, 1)
```

It is OK to use RATIONAL coordinates:

```
sage: o.distances([1,2,3/2])
(-1/2, 3/2, 7/2, 11/2, -7/2, -3/2, 1/2, 5/2)
sage: o.distances([1,2,sqrt(2)])
Traceback (most recent call last):
...
TypeError: unable to convert sqrt(2) to a rational
```

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()
[0 2 2 0 1]
[2 2 0 0 1]
[0 0 2 2 1]
[2 0 0 2 1]
sage: p.distances((1/2, 3, 0))
(7/2, 9/2, -5/2, -3/2)
sage: p.distances((1, 1, 1))
Traceback (most recent call last):
...
ArithmeticError: vector is not in free module
```

dual_lattice()

Return the dual of the ambient lattice of `self`.

OUTPUT:

- a lattice. If possible (that is, if `lattice()` has a `dual()` method), the dual lattice is returned. Otherwise, $\mathbf{Z}^n$ is returned, where n is the dimension of `self`.

EXAMPLES:

```
sage: LatticePolytope([(1,0)]).dual_lattice()
2-d lattice N
sage: LatticePolytope([], lattice=ZZ^3).dual_lattice()
Ambient free module of rank 3
over the principal ideal domain Integer Ring
```

edges()

Return the sequence of edges of this polytope (i.e. faces of dimension 1).

EXAMPLES: The octahedron has 12 edges:

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

```
sage: o.edges()
[[1, 5], [0, 5], [0, 1], [3, 5], [1, 3], [4, 5], [0, 4], [3, 4], [1, 2], [0, 2], [2, 3], [2,
```

faces (*dim=None*, *codim=None*)

Return the sequence of proper faces of this polytope.

If `dim` or `codim` are specified, returns a sequence of faces of the corresponding dimension or codimension. Otherwise returns the sequence of such sequences for all dimensions.

EXAMPLES: All faces of the 3-dimensional octahedron:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.faces()
[
[[0], [1], [2], [3], [4], [5]],
[[1, 5], [0, 5], [0, 1], [3, 5], [1, 3], [4, 5], [0, 4], [3, 4], [1, 2], [0, 2], [2, 3], [2,
[[0, 1, 5], [1, 3, 5], [0, 4, 5], [3, 4, 5], [0, 1, 2], [1, 2, 3], [0, 2, 4], [2, 3, 4]]
]
```

Its faces of dimension one (i.e., edges):

```
sage: o.faces(dim=1)
[[1, 5], [0, 5], [0, 1], [3, 5], [1, 3], [4, 5], [0, 4], [3, 4], [1, 2], [0, 2], [2, 3], [2,
```

Its faces of codimension two (also edges):

```
sage: o.faces(codim=2)
[[1, 5], [0, 5], [0, 1], [3, 5], [1, 3], [4, 5], [0, 4], [3, 4], [1, 2], [0, 2], [2, 3], [2,
```

It is an error to specify both dimension and codimension at the same time, even if they do agree:

```
sage: o.faces(dim=1, codim=2)
Traceback (most recent call last):
...
ValueError: Both dim and codim are given!
```

The only faces of a zero-dimensional polytope are the empty set and the polytope itself, i.e. it has no proper faces at all:

```
sage: p = LatticePolytope([[1]])
sage: p.vertices_pc()
M(1)
in 1-d lattice M
sage: p.faces()
[]
```

In particular, you an exception will be raised if you try to access faces of the given dimension or codimension, including edges and facets:

```
sage: p.facets()
Traceback (most recent call last):
...
IndexError: list index out of range
```

facet_constant (*i*)

Return the constant in the `i`-th facet inequality of this polytope.

The i-th facet inequality is given by self.facet_normal(i) * X + self.facet_constant(i) >= 0.

INPUT:

- `i` - integer, the index of the facet

OUTPUT:

- integer – the constant in the `i`-th facet inequality.

EXAMPLES:

Let's take a look at facets of the octahedron and some polytopes inside it:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.vertices_pc()
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_normal(0)
N(-1, -1, 1)
sage: o.facet_constant(0)
1
sage: p = LatticePolytope(o.vertices_pc()(1,2,3,4,5))
sage: p.vertices_pc()
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: p.facet_normal(0)
N(-1, 0, 0)
sage: p.facet_constant(0)
0
sage: p = LatticePolytope(o.vertices_pc()(1,2,4,5))
sage: p.vertices_pc()
M(0,  1,  0),
M(0,  0,  1),
M(0, -1,  0),
M(0,  0, -1)
in 3-d lattice M
sage: p.facet_normal(0)
N(0, -1, 1)
sage: p.facet_constant(0)
1
```

This is a 2-dimensional lattice polytope in a 4-dimensional space:

```
sage: p = LatticePolytope([(1,-1,1,3), (-1,-1,1,3), (0,0,0,0)])
sage: p
2-d lattice polytope in 4-d lattice M
sage: p.vertices_pc()
M( 1, -1, 1, 3),
M(-1, -1, 1, 3),
M( 0,  0, 0, 0)
in 4-d lattice M
sage: fns = [p.facet_normal(i) for i in range(p.nfacets())]
sage: fns
[N(11, -1, 1, 3), N(-11, -1, 1, 3), N(0, 1, -1, -3)]
sage: fcs = [p.facet_constant(i) for i in range(p.nfacets())]
```

```
sage: fcs
[0, 0, 11]
```

Now we manually compute the distance matrix of this polytope. Since it is a triangle, each line (corresponding to a facet) should have two zeros (vertices of the corresponding facet) and one positive number (since our normals are inner):

```
sage: matrix([[fns[i] * p.vertex(j) + fcs[i]
...            for j in range(p.nvertices())]
...           for i in range(p.nfacets())])
[22  0  0]
[ 0 22  0]
[ 0  0 11]
```

facet_constants()

Return facet constants of `self`.

OUTPUT:

- an integer vector.

EXAMPLES:

For reflexive polytopes all constants are 1:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.vertices_pc()
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:

```
sage: p = LatticePolytope([(0,0,0,0), (1,1,1,3),
...                        (1,-1,1,3), (-1,-1,1,3)])
sage: p.vertices_pc()
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, 0, 10)
```

facet_normal(*i*)

Return the inner normal to the `i`-th facet of this polytope.

If this polytope is not full-dimensional, facet normals will be orthogonal to the integer kernel of the affine subspace spanned by this polytope.

INPUT:

- `i` – integer, the index of the facet

OUTPUT:

- vectors – the inner normal of the `i`-th facet

EXAMPLES:

Let's take a look at facets of the octahedron and some polytopes inside it:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.vertices_pc()
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_normal(0)
N(-1, -1, 1)
sage: o.facet_constant(0)
1
sage: p = LatticePolytope(o.vertices_pc()(1,2,3,4,5))
sage: p.vertices_pc()
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: p.facet_normal(0)
N(-1, 0, 0)
sage: p.facet_constant(0)
0
sage: p = LatticePolytope(o.vertices_pc()(1,2,4,5))
sage: p.vertices_pc()
M(0,  1,  0),
M(0,  0,  1),
M(0, -1,  0),
M(0,  0, -1)
in 3-d lattice M
sage: p.facet_normal(0)
N(0, -1, 1)
sage: p.facet_constant(0)
1
```

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

```
sage: p = LatticePolytope([(0,0,0,0), (1,1,1,3),
...                        (1,-1,1,3), (-1,-1,1,3)])
sage: p.vertices_pc()
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: ker = p.vertices_pc().column_matrix().integer_kernel().matrix()
sage: ker
[ 0  0  3 -1]
sage: ker * p.facet_normals()
[0 0 0 0]
```

Now we manually compute the distance matrix of this polytope. Since it is a simplex, each line (corresponding to a facet) should consist of zeros (indicating generating vertices of the corresponding facet) and a single positive number (since our normals are inner):

```
sage: matrix([[p.facet_normal(i) * p.vertex(j)
...            + p.facet_constant(i)
...            for j in range(p.nvertices())]
...           for i in range(p.nfacets())])
[ 0  0  0 20]
[ 0  0 20  0]
[ 0 20  0  0]
[10  0  0  0]
```

facet_normals()

Return inner normals to the facets of `self`.

OUTPUT:

- a `point collection` in the `dual_lattice()` of `self`.

EXAMPLES:

Normals to facets of an octahedron are vertices of a cube:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.vertices_pc()
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1)
in 3-d lattice M
sage: o.facet_normals()
N(-1, -1,  1),
N( 1, -1,  1),
N(-1,  1,  1),
N( 1,  1,  1),
N(-1, -1, -1),
N( 1, -1, -1),
N(-1,  1, -1),
N( 1,  1, -1)
in 3-d lattice N
```

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

```
sage: p = LatticePolytope([(0,0,0,0), (1,1,1,3),
...                        (1,-1,1,3), (-1,-1,1,3)])
sage: p.vertices_pc()
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(-10,   0,  1,  3),
N( 10, -10,  0,  0),
N(  0,  10,  1,  3),
N(  0,   0, -1, -3)
in 4-d lattice N
```

facets()

Return the sequence of facets of this polytope (i.e. faces of codimension 1).

EXAMPLES: All facets of the 3-dimensional octahedron:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.facets()
[[0, 1, 5], [1, 3, 5], [0, 4, 5], [3, 4, 5], [0, 1, 2], [1, 2, 3], [0, 2, 4], [2, 3, 4]]
```

Facets are the same as faces of codimension one:

```
sage: o.facets() is o.faces(codim=1)
True
```

index()

Return the index of this polytope in the internal database of 2- or 3-dimensional reflexive polytopes. Databases are stored in the directory of the package.

Note: The first call to this function for each dimension can take a few seconds while the dictionary of all polytopes is constructed, but after that it is cached and fast.

Return type integer

EXAMPLES: We check what is the index of the "diamond" in the database:

```
sage: d = lattice_polytope.cross_polytope(2)
sage: d.index()
3
```

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

```
sage: d.vertices_pc()
M( 1,  0),
M( 0,  1),
M(-1,  0),
M( 0, -1)
in 2-d lattice M
sage: lattice_polytope.ReflexivePolytope(2,3).vertices_pc()
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:

```
sage: d.normal_form_pc()
M( 1,  0),
M( 0,  1),
M( 0, -1),
M(-1,  0)
in 2-d lattice M
sage: lattice_polytope.ReflexivePolytope(2,3).normal_form_pc()
M( 1,  0),
M( 0,  1),
M( 0, -1),
M(-1,  0)
in 2-d lattice M
```

is_reflexive()
Return True if this polytope is reflexive.

EXAMPLES: The 3-dimensional octahedron is reflexive (and 4318 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
```

linearly_independent_vertices()
Return a maximal set of linearly independent vertices.

OUTPUT:

A tuple of vertex indices.

EXAMPLES:

```
sage: L = LatticePolytope([[0, 0], [-1, 1], [-1, -1]])
sage: L.linearly_independent_vertices()
(1, 2)
sage: L = LatticePolytope([[0, 0, 0]])
sage: L.linearly_independent_vertices()
()
sage: L = LatticePolytope([[0, 1, 0]])
sage: L.linearly_independent_vertices()
(0,)
```

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:

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

Now we omit projections:

```
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:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.nfacets()
8
```

The number of facets of an interval is 2:

```
sage: LatticePolytope(([1],[2])).nfacets()
2
```

Now consider a 2-dimensional diamond in a 3-dimensional space:

```
sage: p = LatticePolytope([(1,0,0), (0,1,0), (-1,0,0), (0,-1,0)])
sage: p.nfacets()
4
```

normal_form()

Return the normal form of `self` as a matrix.

EXAMPLES:

We compute the normal form of the "diamond":

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.normal_form()
doctest:...: DeprecationWarning: normal_form() output will change,
please use normal_form_pc().column_matrix() instead
or consider using normal_form_pc() directly!
See http://trac.sagemath.org/15240 for details.
[ 1  0  0  0  0 -1]
[ 0  1  0  0 -1  0]
[ 0  0  1 -1  0  0]
```

normal_form_pc(*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 [KS98] and implemented in PALP. A modified version of the PALP algorithm is discussed in [GK13] 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.

REFERENCES:

EXAMPLES:

We compute the normal form of the "diamond":

```
sage: d = LatticePolytope([(1,0), (0,1), (-1,0), (0,-1)])
sage: d.vertices_pc()
M( 1,  0),
M( 0,  1),
M(-1,  0),
M( 0, -1)
in 2-d lattice M
sage: d.normal_form_pc()
M( 1,  0),
M( 0,  1),
M( 0, -1),
M(-1,  0)
in 2-d lattice M
```

The diamond is the 3rd polytope in the internal database:

```
sage: d.index()
3
sage: d
2-d reflexive polytope #3 in 2-d lattice M
```

You can get it in its normal form (in the default lattice) as

```
sage: lattice_polytope.ReflexivePolytope(2, 3).vertices_pc()
M( 1,  0),
M( 0,  1),
M( 0, -1),
M(-1,  0)
in 2-d lattice M
```

It is not possible to compute normal forms for polytopes which do not span the space:

```
sage: p = LatticePolytope([(1,0,0), (0,1,0), (-1,0,0), (0,-1,0)])
sage: p.normal_form()
Traceback (most recent call last):
...
ValueError: normal form is not defined for
2-d lattice polytope in 3-d lattice M
```

We can perform the same examples using other algorithms:

```
sage: o = lattice_polytope.cross_polytope(2)
sage: o.normal_form_pc(algorithm="palp_native")
M( 1,  0),
M( 0,  1),
M( 0, -1),
M(-1,  0)
```

```
in 2-d lattice M

sage: o = lattice_polytope.cross_polytope(2)
sage: o.normal_form_pc(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_pc()
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 #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:(0.5,0.5,0.5))
- `show_vertices` - (default:True) whether to draw vertices as balls
- `vertex_size` - (default:10)
- `vertex_color` - (default:(1,0,0))
- `show_points` - (default:True) whether to draw other poits as balls
- `point_size` - (default:10)
- `point_color` - (default:(0,0,1))
- `show_vindices` - (default:same as show_vertices) whether to show indices of vertices
- `vindex_color` - (default:(0,0,0)) color for vertex labels
- `vlabels` - (default:None) if specified, must be a list of labels for each vertex, default labels are vertex indicies
- `show_pindices` - (default:same as show_points) whether to show indices of other points
- `pindex_color` - (default:(0,0,0)) color for point labels

- `index_shift` - (default:1.1)) if 1, labels are placed exactly at the corresponding points. Otherwise the label position is computed as a multiple of the point position vector.

EXAMPLES: The default plot of a cube:

```
sage: c = lattice_polytope.cross_polytope(3).polar()
sage: c.plot3d()
```

Plot without facets and points, shown without the frame:

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

Plot with facets of different colors:

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

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

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

TESTS:

```
sage: p = LatticePolytope([[0,0,0],[0,1,1],[1,0,1],[1,1,0]])
sage: p.plot3d()
```

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_pc()
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_pc()
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()

Return all lattice points of this polytope as columns of a matrix.

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.points()
doctest:...: DeprecationWarning: points() output will change,
please use points_pc().column_matrix() instead or
consider using points_pc() directly!
See http://trac.sagemath.org/15240 for details.
[ 1  0  0 -1  0  0  0]
[ 0  1  0  0 -1  0  0]
[ 0  0  1  0  0 -1  0]
```

points_pc()

Return all lattice points of `self`.

OUTPUT:

- a `point collection`.

EXAMPLES:

Lattice points of the octahedron and its polar cube:

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

```
in 3-d lattice N
```

Lattice points of a 2-dimensional diamond in a 3-dimensional space:

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

We check that points of a zero-dimensional polytope can be computed:

```
sage: p = LatticePolytope([[1]])
sage: p.points_pc()
M(1)
in 1-d lattice M
```

polar()

Return the polar polytope, if this polytope is reflexive.

EXAMPLES: The polar polytope to the 3-dimensional octahedron:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: cube = o.polar()
sage: cube
3-d reflexive polytope in 3-d lattice N
```

The polar polytope "remembers" the original one:

```
sage: cube.polar()
3-d reflexive polytope in 3-d lattice M
sage: cube.polar().polar() is cube
True
```

Only reflexive polytopes have polars:

```
sage: p = LatticePolytope([(1,0,0), (0,1,0), (0,0,2),
...                        (-1,0,0), (0,-1,0), (0,0,-1)])
sage: p.polar()
Traceback (most recent call last):
...
ValueError: The given polytope is not reflexive!
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:

```
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
  -1   1  -1
   1   1  -1
```

Since PALP has limits on different parameters determined during compilation, the following code is likely to fail, unless you change default settings of PALP:

```
sage: BIG = lattice_polytope.cross_polytope(7)
sage: BIG
7-d lattice 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
```

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: We construct the one-skeleton graph for the "diamond":

```
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:

```
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:

```
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:

```
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):

```
sage: c.skeleton_points(k=2)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 18, 19, 20, 21, 22, 23, 24, 25, 2
```

3-skeleton includes all points:

```
sage: c.skeleton_points(k=3)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 2
```

It is OK to compute higher dimensional skeletons - you will get the list of all points:

```
sage: c.skeleton_points(k=100)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 2
```

skeleton_show(*normal=None*)

Show the graph of one-skeleton of this polytope. Works only for polytopes in a 3-dimensional space.

INPUT:

- `normal` - a 3-dimensional vector (can be given as a list), which should be perpendicular to the screen. If not given, will be selected randomly (new each time and it may be far from "nice").

EXAMPLES: Show a pretty picture of the octahedron:

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

Does not work for a diamond at the moment:

```
sage: d = lattice_polytope.cross_polytope(2)
sage: d.skeleton_show()
Traceback (most recent call last):
...
NotImplementedError: skeleton view is implemented only in 3-d space
```

traverse_boundary()

Return a list of indices of vertices of a 2-dimensional polytope in their boundary order.

Needed for plot3d function of polytopes.

EXAMPLES:

sage: p = lattice_polytope.cross_polytope(2).polar() sage: p.traverse_boundary() [0, 1, 3, 2]

vertex(*i*)

Return the i-th vertex of this polytope, i.e. the i-th column of the matrix returned by vertices().

EXAMPLES: Note that numeration starts with zero:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.vertices_pc()
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1)
in 3-d lattice M
sage: o.vertex(3)
M(-1, 0, 0)
```

vertex_facet_pairing_matrix()

Return the vertex facet pairing matrix PM.

Return a matrix whose the i, j^{th} entry is the height of the j^{th} vertex over the i^{th} facet. The ordering of the vertices and facets is as in `vertices()` and `facets()`.

EXAMPLES:

```
sage: L = lattice_polytope.cross_polytope(3)
sage: L.vertex_facet_pairing_matrix()
[0 0 2 2 2 0]
[2 0 2 0 2 0]
[0 2 2 2 0 0]
[2 2 2 0 0 0]
[0 0 0 2 2 2]
[2 0 0 0 2 2]
[0 2 0 2 0 2]
[2 2 0 0 0 2]
```

vertices()

Return vertices of this polytope as columns of a matrix.

EXAMPLES: The lattice points of the 3-dimensional octahedron and its polar cube:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o.vertices()
doctest:...: DeprecationWarning: vertices() output will change,
please use vertices_pc().column_matrix() instead or
consider using vertices_pc() directly!
See http://trac.sagemath.org/15240 for details.
[ 1  0  0 -1  0  0]
[ 0  1  0  0 -1  0]
[ 0  0  1  0  0 -1]
sage: cube = o.polar()
sage: cube.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]
```

vertices_pc()
Return vertices of self.

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_pc()
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_pc()
N(-1, -1,  1),
N( 1, -1,  1),
N(-1,  1,  1),
N( 1,  1,  1),
N(-1, -1, -1),
N( 1, -1, -1),
N(-1,  1, -1),
N( 1,  1, -1)
in 3-d lattice N
```

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

Create a nef-partition.

INPUT:

- data – a list of integers, the i-th element of this list must be the part of the i-th vertex of Delta_polar in this nef-partition;
- Delta_polar – a lattice polytope;
- check – by default the input will be checked for correctness, i.e. that data indeed specify a nef-partition. If you are sure that the input is correct, you can speed up construction via check=False option.

OUTPUT:

- a nef-partition of Delta_polar.

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

The **dual nef-partition** is formed by polytopes $\Delta_i \subset M_{\mathbf{R}}$ of E_i, which give a decomposition of the vertex set of $\nabla^\circ \subset M_{\mathbf{R}}$ and their Minkowski sum is Δ, i.e. the polar duality of reflexive polytopes switches convex hull

and Minkowski sum for dual nef-partitions:

$$\begin{aligned}\Delta^\circ &= \mathrm{Conv}\left(\nabla_0, \nabla_1, \ldots, \nabla_{k-1}\right),\\ \nabla &= \nabla_0 + \nabla_1 + \cdots + \nabla_{k-1},\end{aligned}$$

$$\begin{aligned}\Delta &= \Delta_0 + \Delta_1 + \cdots + \Delta_{k-1},\\ \nabla^\circ &= \mathrm{Conv}\left(\Delta_0, \Delta_1, \ldots, \Delta_{k-1}\right).\end{aligned}$$

See Section 4.3.1 in [CK99] and references therein for further details, or [BN08] for a purely combinatorial approach.

REFERENCES:

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 Δ° in the above description:

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

The dual nef-partition (corresponding to the "mirror complete intersection") gives decomposition of the vertex set of ∇°:

```
sage: np.dual()
Nef-partition {4, 5, 6} U {1, 3} U {0, 2, 7}
sage: np.nabla_polar().vertices_pc()
N( 1,  1,  0),
N( 0,  0,  1),
N( 0,  1,  0),
N( 0,  0, -1),
N(-1, -1,  0),
N( 0, -1,  0),
N(-1,  0,  0),
N( 1,  0,  0)
in 3-d lattice N
```

Of course, ∇° is Δ° from the point of view of the dual nef-partition:

```
sage: np.dual().Delta_polar() is np.nabla_polar()
True
sage: np.Delta(1).vertices_pc()
N(0, 0,  1),
N(0, 0, -1)
in 3-d lattice N
sage: np.dual().nabla(1).vertices_pc()
N(0, 0,  1),
N(0, 0, -1)
in 3-d lattice N
```

Instead of constructing nef-partitions directly, you can request all 2-part nef-partitions of a given reflexive polytope (they will be computed using `nef.x` program from PALP):

```
sage: o.nef_partitions()
[
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:

```
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_pc()
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_pc()
N( 1, -1, 0),
N( 1,  0, 0),
N(-1, -1, 0),
N(-1,  0, 0)
in 3-d lattice N
```

Delta_polar()

Return the polytope Δ° corresponding to `self`.

OUTPUT:

- a `lattice polytope`.

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

EXAMPLE:

```
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 Δ_i corresponding to `self`.

OUTPUT:

- a tuple of lattice polytopes.

See nef-partition class documentation for definitions and notation.

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: np = o.nef_partitions()[0]
sage: np
Nef-partition {0, 1, 3} U {2, 4, 5}
sage: np.Delta().vertices_pc()
N(-1, -1,  1),
N( 1, -1,  1),
N(-1,  1,  1),
N( 1,  1,  1),
N(-1, -1, -1),
N( 1, -1, -1),
N(-1,  1, -1),
N( 1,  1, -1)
in 3-d lattice N
sage: [Delta_i.vertices_pc() for Delta_i in np.Deltas()]
[N( 1, -1, 0),
N( 1,  0, 0),
N(-1, -1, 0),
N(-1,  0, 0)
in 3-d lattice N,
N(0, 1,  1),
N(0, 0,  1),
N(0, 0, -1),
N(0, 1, -1)
in 3-d lattice N]
sage: np.nabla_polar().vertices_pc()
N( 1, -1,  0),
N( 0,  1,  1),
N( 1,  0,  0),
N( 0,  0,  1),
N( 0,  0, -1),
N(-1, -1,  0),
N( 0,  1, -1),
N(-1,  0,  0)
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 [BN08].

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, 2, 5, 7} U {1, 3, 4, 6}
sage: np.dual().Delta() is np.nabla()
True
sage: np.dual().nabla(0) is np.Delta(0)
True
```

hodge_numbers()

Return Hodge numbers corresponding to `self`.

OUTPUT:

- a tuple of integers (produced by `nef.x` program from PALP).

EXAMPLES:

Currently, you need to request Hodge numbers when you compute nef-partitions:

```
sage: 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 ∇ or ∇_i corresponding to `self`.

INPUT:

- `i` – an integer. If not given, ∇ will be returned.

OUTPUT:

- a `lattice polytope`.

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

EXAMPLES:

```
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_pc()
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_pc()
```

```
M( 1,  0,  0),
M( 0,  1,  0),
M(-1,  0,  0)
in 3-d lattice M
sage: np.nabla().vertices_pc()
M( 1,  0,  1),
M( 1, -1,  0),
M( 1,  0, -1),
M( 0,  1,  1),
M( 0,  1, -1),
M(-1,  0,  1),
M(-1, -1,  0),
M(-1,  0, -1)
in 3-d lattice M
```

nabla_polar()

Return the polytope ∇° corresponding to `self`.

OUTPUT:

- a lattice polytope.

See nef-partition class documentation for definitions and notation.

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: np = o.nef_partitions()[0]
sage: np
Nef-partition {0, 1, 3} U {2, 4, 5}
sage: np.nabla_polar().vertices_pc()
N( 1, -1,  0),
N( 0,  1,  1),
N( 1,  0,  0),
N( 0,  0,  1),
N( 0,  0, -1),
N(-1, -1,  0),
N( 0,  1, -1),
N(-1,  0,  0)
in 3-d lattice N
sage: np.nabla_polar() is np.dual().Delta_polar()
True
```

nablas()

Return the polytopes ∇_i corresponding to `self`.

OUTPUT:

- a tuple of lattice polytopes.

See nef-partition class documentation for definitions and notation.

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: np = o.nef_partitions()[0]
sage: np
Nef-partition {0, 1, 3} U {2, 4, 5}
sage: np.Delta_polar().vertices_pc()
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_pc() for nabla_i in np.nablas()]
[M( 1, 0, 0),
M( 0, 1, 0),
M(-1, 0, 0)
in 3-d lattice M,
M(0,  0,  1),
M(0, -1,  0),
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)

Return the `i`-th part of `self`.

INPUT:

- `i` – an integer.

OUTPUT:

- a tuple of integers, indices of vertices of Δ° belonging to V_i.

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

EXAMPLES:

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

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 np:

```
sage: [n for n in range(p.npoints())
...       if p.origin() != n and np.part_of_point(n) == 0]
[1, 2, 5, 7, 8, 9, 11, 13]
sage: [n for n in range(p.npoints())
...       if p.origin() != n and np.part_of_point(n) == 1]
[0, 3, 4, 6, 10, 12]
```

parts()
Return all parts of `self`.

OUTPUT:

•a tuple of tuples of integers. The i-th tuple contains indices of vertices of $Delta^circ$ belonging to V_i.

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

EXAMPLES:

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

sage.geometry.lattice_polytope.**ReflexivePolytope**(*dim*, *n*)
Return n-th reflexive polytope from the database of 2- or 3-dimensional reflexive polytopes.

Note:

1. Numeration starts with zero: $0 \leq n \leq 15$ for $\dim = 2$ and $0 \leq n \leq 4318$ for $\dim = 3$.
2. During the first call, all reflexive polytopes of requested dimension are loaded and cached for future use, so the first call for 3-dimensional polytopes can take several seconds, but all consecutive calls are fast.
3. Equivalent to `ReflexivePolytopes(dim)[n]` but checks bounds first.

EXAMPLES: The 3rd 2-dimensional polytope is "the diamond:"

```
sage: ReflexivePolytope(2, 3)
2-d reflexive polytope #3 in 2-d lattice M
sage: lattice_polytope.ReflexivePolytope(2,3).vertices_pc()
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

Base class for all parents.

Parents are the Sage/mathematical analogues of container objects in computer science.

INPUT:

- base – An algebraic structure considered to be the "base" of this parent (e.g. the base field for a vector space).
- category – a category or list/tuple of categories. The category in which this parent lies (or list or tuple thereof). Since categories support more general super-categories, this should be the most specific category possible. If category is a list or tuple, a JoinCategory is created out of them. If category is not specified, the category will be guessed (see CategoryObject), but won't be used to inherit parent's or element's code from this category.
- element_constructor – A class or function that creates elements of this Parent given appropriate input (can also be filled in later with _populate_coercion_lists_())
- gens – Generators for this object (can also be filled in later with _populate_generators_())
- names – Names of generators.
- normalize – Whether to standardize the names (remove punctuation, etc)
- facade – a parent, or tuple thereof, or True

If facade is specified, then Sets().Facade() is added to the categories of the parent. Furthermore, if facade is not True, the internal attribute _facade_for is set accordingly for use by Sets.Facade.ParentMethods.facade_for().

Internal invariants:

- self._element_init_pass_parent == guess_pass_parent(self, self._element_constructor) Ensures that __call__() passes down the parent properly to _element_constructor(). See trac ticket #5979.

Todo

Eventually, category should be Sets by default.

TESTS:

We check that the facade option is compatible with specifying categories as a tuple:

```
sage: class MyClass(Parent): pass
sage: P = MyClass(facade = ZZ, category = (Monoids(), CommutativeAdditiveMonoids()))
sage: P.category()
Join of Category of monoids and Category of commutative additive monoids and Category of facade
```

__call__(*x*)

TESTS:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: lattice_polytope.SetOfAllLatticePolytopesClass().__call__(o)
3-d reflexive polytope in 3-d lattice M
```

_populate_coercion_lists_(*coerce_list=[]*, *action_list=[]*, *convert_list=[]*, *embedding=None*, *convert_method_name=None*, *element_constructor=None*, *init_no_parent=None*, *unpickling=False*)

This function allows one to specify coercions, actions, conversions and embeddings involving this parent.

IT SHOULD ONLY BE CALLED DURING THE __INIT__ method, often at the end.

INPUT:

- `coerce_list` – a list of coercion Morphisms to self and parents with canonical coercions to self
- `action_list` – a list of actions on and by self
- **`convert_list` – a list of conversion Maps to self and** parents with conversions to self
- `embedding` – a single Morphism from self
- `convert_method_name` – a name to look for that other elements can implement to create elements of self (e.g. _integer_)
- `element_constructor` – A callable object used by the __call__ method to construct new elements. Typically the element class or a bound method (defaults to self._element_constructor_).
- `init_no_parent` – if True omit passing self in as the first argument of element_constructor for conversion. This is useful if parents are unique, or element_constructor is a bound method (this latter case can be detected automatically).

__mul__(*x*)

This is a multiplication method that more or less directly calls another attribute `_mul_` (single underscore). This is because `__mul__` can not be implemented via inheritance from the parent methods of the category, but `_mul_` can be inherited. This is, e.g., used when creating twosided ideals of matrix algebras. See trac ticket #7797.

EXAMPLE:

```
sage: MS = MatrixSpace(QQ,2,2)
```

This matrix space is in fact an algebra, and in particular it is a ring, from the point of view of categories:

```
sage: MS.category()
Category of algebras over quotient fields
sage: MS in Rings()
True
```

However, its class does not inherit from the base class `Ring`:

```
sage: isinstance(MS,Ring)
False
```

Its _mul_ method is inherited from the category, and can be used to create a left or right ideal:

```
sage: MS._mul_.__module__
'sage.categories.rings'
sage: MS*MS.1      # indirect doctest
Left Ideal
(
  [0 1]
  [0 0]
)
 of Full MatrixSpace of 2 by 2 dense matrices over Rational Field
sage: MS*[MS.1,2]
Left Ideal
(
  [0 1]
  [0 0],

  [2 0]
  [0 2]
)
 of Full MatrixSpace of 2 by 2 dense matrices over Rational Field
sage: MS.1*MS
Right Ideal
(
  [0 1]
  [0 0]
)
 of Full MatrixSpace of 2 by 2 dense matrices over Rational Field
sage: [MS.1,2]*MS
Right Ideal
(
  [0 1]
  [0 0],

  [2 0]
  [0 2]
)
 of Full MatrixSpace of 2 by 2 dense matrices over Rational Field
```

__contains__(*x*)

True if there is an element of self that is equal to x under ==, or if x is already an element of self. Also, True in other cases involving the Symbolic Ring, which is handled specially.

For many structures we test this by using __call__() and then testing equality between x and the result.

The Symbolic Ring is treated differently because it is ultra-permissive about letting other rings coerce in, but ultra-strict about doing comparisons.

EXAMPLES:

```
sage: 2 in Integers(7)
True
sage: 2 in ZZ
True
sage: Integers(7)(3) in ZZ
True
sage: 3/1 in ZZ
True
sage: 5 in QQ
True
sage: I in RR
```

```
False
sage: SR(2) in ZZ
True
sage: RIF(1, 2) in RIF
True
sage: pi in RIF # there is no element of RIF equal to pi
False
sage: sqrt(2) in CC
True
sage: pi in RR
True
sage: pi in CC
True
sage: pi in RDF
True
sage: pi in CDF
True
```

TESTS:

Check that trac ticket #13824 is fixed:

```
sage: 4/3 in GF(3)
False
sage: 15/50 in GF(25, 'a')
False
sage: 7/4 in Integers(4)
False
sage: 15/36 in Integers(6)
False
```

_coerce_map_from_(*S*)

Override this method to specify coercions beyond those specified in coerce_list.

If no such coercion exists, return None or False. Otherwise, it may return either an actual Map to use for the coercion, a callable (in which case it will be wrapped in a Map), or True (in which case a generic map will be provided).

_convert_map_from_(*S*)

Override this method to provide additional conversions beyond those given in convert_list.

This function is called after coercions are attempted. If there is a coercion morphism in the opposite direction, one should consider adding a section method to that.

This MUST return a Map from S to self, or None. If None is returned then a generic map will be provided.

_get_action_(*S*, *op*, *self_on_left*)

Override this method to provide an action of self on S or S on self beyond what was specified in action_list.

This must return an action which accepts an element of self and an element of S (in the order specified by self_on_left).

_an_element_()

Returns an element of self. Want it in sufficient generality that poorly-written functions won't work when they're not supposed to. This is cached so doesn't have to be super fast.

EXAMPLES:

```
sage: QQ._an_element_()
1/2
```

```
sage: ZZ['x,y,z']._an_element_()
x
```

TESTS:

Since `Parent` comes before the parent classes provided by categories in the hierarchy of classes, we make sure that this default implementation of `_an_element_()` does not override some provided by the categories. Eventually, this default implementation should be moved into the categories to avoid this workaround:

```
sage: S = FiniteEnumeratedSet([1,2,3])
sage: S.category()
Category of facade finite enumerated sets
sage: super(Parent, S)._an_element_
Cached version of <function _an_element_from_iterator at ...>
sage: S._an_element_()
1
sage: S = FiniteEnumeratedSet([])
sage: S._an_element_()
Traceback (most recent call last):
...
EmptySetError
```

_repr_option(*key*)

Metadata about the `_repr_()` output.

INPUT:

- `key` – string. A key for different metadata informations that can be inquired about.

Valid `key` arguments are:

- `'ascii_art'`: The `_repr_()` output is multi-line ascii art and each line must be printed starting at the same column, or the meaning is lost.
- `'element_ascii_art'`: same but for the output of the elements. Used in `sage.misc.displayhook`.
- `'element_is_atomic'`: the elements print atomically, that is, parenthesis are not required when *printing* out any of $x-y$, $x+y$, x^y and x/y.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: ZZ._repr_option('ascii_art')
False
sage: MatrixSpace(ZZ, 2)._repr_option('element_ascii_art')
True
```

_init_category_(*category*)

Initialize the category framework

Most parents initialize their category upon construction, and this is the recommended behavior. For example, this happens when the constructor calls `Parent.__init__()` directly or indirectly. However, some parents defer this for performance reasons. For example, `sage.matrix.matrix_space.MatrixSpace` does not.

EXAMPLES:

```
sage: P = Parent()
sage: P.category()
Category of sets
sage: class MyParent(Parent):
....:     def __init__(self):
....:         self._init_category_(Groups())
sage: MyParent().category()
Category of groups
```

sage.geometry.lattice_polytope.**all_cached_data**(*polytopes*)

Compute all cached data for all given `polytopes` and their polars.

This functions does it MUCH faster than member functions of `LatticePolytope` during the first run. So it is recommended to use this functions if you work with big sets of data. None of the polytopes in the given sequence should be constructed as the polar polytope to another one.

INPUT: a sequence of lattice polytopes.

EXAMPLES: This function has no output, it is just a fast way to work with long sequences of polytopes. Of course, you can use short sequences as well:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: lattice_polytope.all_cached_data([o])
sage: o.faces()
[
[[0], [1], [2], [3], [4], [5]],
[[1, 5], [0, 5], [0, 1], [3, 5], [1, 3], [4, 5], [0, 4], [3, 4], [1, 2], [0, 2], [2, 3], [2, 4]]
[[0, 1, 5], [1, 3, 5], [0, 4, 5], [3, 4, 5], [0, 1, 2], [1, 2, 3], [0, 2, 4], [2, 3, 4]]
]
```

However, you cannot use it for polytopes that are constructed as polar polytopes of others:

```
sage: lattice_polytope.all_cached_data([o.polar()])
Traceback (most recent call last):
...
ValueError: Cannot read face structure for a polytope constructed as polar, use _compute_faces!
```

sage.geometry.lattice_polytope.**all_faces**(*polytopes*)

Compute faces 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:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: lattice_polytope.all_faces([o])
sage: o.faces()
[
[[0], [1], [2], [3], [4], [5]],
[[1, 5], [0, 5], [0, 1], [3, 5], [1, 3], [4, 5], [0, 4], [3, 4], [1, 2], [0, 2], [2, 3], [2, 4]]
[[0, 1, 5], [1, 3, 5], [0, 4, 5], [3, 4, 5], [0, 1, 2], [1, 2, 3], [0, 2, 4], [2, 3, 4]]
]
```

However, you cannot use it for polytopes that are constructed as polar polytopes of others:

```
sage: lattice_polytope.all_faces([o.polar()])
Traceback (most recent call last):
```

```
...
ValueError: Cannot read face structure for a polytope constructed as polar, use _compute_faces!
```

sage.geometry.lattice_polytope.**all_facet_equations**(*polytopes*)

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

`all_facet_equations` and `all_polars` are synonyms.

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

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

Note: member function `is_reflexive` will be called separately for each polytope. It is strictly recommended to call `all_polars` on the sequence of `polytopes` before using this function.

INPUT: a sequence of lattice polytopes.

EXAMPLES: This function has no output, it is just a fast way to work with long sequences of polytopes. Of course, you can use short sequences as well:

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

You cannot use this function for non-reflexive polytopes:

```
sage: p = LatticePolytope([(1,0,0), (0,1,0), (0,0,2),
...                        (-1,0,0), (0,-1,0), (0,0,-1)])
sage: lattice_polytope.all_nef_partitions([o, p])
Traceback (most recent call last):
...
ValueError: nef-partitions can be computed for reflexive polytopes only
```

sage.geometry.lattice_polytope.**all_points**(*polytopes*)

Compute lattice points for all given `polytopes`.

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

INPUT: a sequence of lattice polytopes.

EXAMPLES: This function has no output, it is just a fast way to work with long sequences of polytopes. Of course, you can use short sequences as well:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: lattice_polytope.all_points([o])
sage: o.points_pc()
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1,  0,  0),
M( 0, -1,  0),
M( 0,  0, -1),
M( 0,  0,  0)
in 3-d lattice M
```

sage.geometry.lattice_polytope.**all_polars**(*polytopes*)

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

`all_facet_equations` and `all_polars` are synonyms.

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

INPUT: a sequence of lattice polytopes.

EXAMPLES: This function has no output, it is just a fast way to work with long sequences of polytopes. Of course, you can use short sequences as well:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: lattice_polytope.all_polars([o])
sage: o.polar()
3-d reflexive polytope in 3-d lattice N
```

sage.geometry.lattice_polytope.**always_use_files**(*new_state=None*)

Set or get the way of using PALP for lattice polytopes.

INPUT:

- `new_state` - (default:None) if specified, must be `True` or `False`.

OUTPUT: The current state of using PALP. If `True`, files are used for all calls to PALP, otherwise pipes are used for single polytopes. While the latter may have some advantage in speed, the first method is more reliable when working with large outputs. The initial state is `True`.

EXAMPLES:

```
sage: lattice_polytope.always_use_files()
doctest:...: DeprecationWarning: using PALP via pipes is deprecated and
will be removed, if you have a use case for this,
please email Andrey Novoseltsev
See http://trac.sagemath.org/15240 for details.
True
sage: p = LatticePolytope(([1], [20]))
sage: p.npoints()
20
```

Now let's use pipes instead of files:

```
sage: lattice_polytope.always_use_files(False)
False
sage: p = LatticePolytope(([1], [20]))
sage: p.npoints()
20
```

sage.geometry.lattice_polytope.**convex_hull**(*points*)
Compute the convex hull of the given points.

Note: `points` might not span the space. Also, it fails for large numbers of vertices in dimensions 4 or greater

---

INPUT:

- `points` - a list that can be converted into vectors of the same dimension over ZZ.

OUTPUT: list of vertices of the convex hull of the given points (as vectors).

EXAMPLES: Let's compute the convex hull of several points on a line in the plane:

```
sage: lattice_polytope.convex_hull([[1,2],[3,4],[5,6],[7,8]])
[(1, 2), (7, 8)]
```

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

INPUT:

- `dim` – an integer.

OUTPUT:

- a `lattice polytope`.

EXAMPLES:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: o
3-d reflexive polytope in 3-d lattice M
sage: o.vertices_pc()
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.**filter_polytopes**(*f*, *polytopes*, *subseq=None*, *print_numbers=False*)
Use the function `f` to filter polytopes in a list.

INPUT:

- `f` - filtering function, it must take one argument, a lattice polytope, and return True or False.
- `polytopes` - list of polytopes.
- `subseq` - (default: None) list of integers. If it is specified, only polytopes with these numbers will be considered.
- `print_numbers` - (default: False) if True, the number of the current polytope will be printed on the screen before calling `f`.

OUTPUT: a list of integers – numbers of polytopes in the given list, that satisfy the given condition (i.e. function `f` returns True) and are elements of subseq, if it is given.

EXAMPLES: Consider a sequence of cross-polytopes:

```
sage: polytopes = Sequence([lattice_polytope.cross_polytope(n)
....:                       for n in range(2, 7)], cr=True)
sage: polytopes
[
2-d reflexive polytope #3 in 2-d lattice M,
3-d reflexive polytope in 3-d lattice M,
4-d reflexive polytope in 4-d lattice M,
5-d reflexive polytope in 5-d lattice M,
6-d reflexive polytope in 6-d lattice M
]
```

This filters polytopes of dimension at least 4:

```
sage: lattice_polytope.filter_polytopes(lambda p: p.dim() >= 4, polytopes)
doctest:...: DeprecationWarning: filter_polytopes is deprecated,
use standard tools instead
See http://trac.sagemath.org/15240 for details.
[2, 3, 4]
```

For long tests you can see the current progress:

```
sage: lattice_polytope.filter_polytopes(lambda p: p.nvertices() >= 10, polytopes, print_numbers=
0
1
2
3
4
[3, 4]
```

Here we consider only some of the polytopes:

```
sage: lattice_polytope.filter_polytopes(lambda p: p.nvertices() >= 10, polytopes, [2, 3, 4], pri
2
3
4
[3, 4]
```

sage.geometry.lattice_polytope.**integral_length**(*v*)

Compute the integral length of a given rational vector.

INPUT:

- v - any object which can be converted to a list of rationals

OUTPUT: Rational number r such that v = r u, where u is the primitive integral vector in the direction of v.

EXAMPLES:

```
sage: lattice_polytope.integral_length([1, 2, 4])
1
sage: lattice_polytope.integral_length([2, 2, 4])
2
sage: lattice_polytope.integral_length([2/3, 2, 4])
2/3
```

sage.geometry.lattice_polytope.**is_LatticePolytope**(*x*)

Check if x is a lattice polytope.

INPUT:

- x – anything.

OUTPUT:

- `True` if `x` is a `lattice polytope`, `False` otherwise.

EXAMPLES:

```
sage: from sage.geometry.lattice_polytope import is_LatticePolytope
sage: is_LatticePolytope(1)
False
sage: p = LatticePolytope([(1,0), (0,1), (-1,-1)])
sage: p
2-d reflexive polytope #0 in 2-d lattice M
sage: is_LatticePolytope(p)
True
```

sage.geometry.lattice_polytope.**is_NefPartition**(*x*)

Check if `x` is a nef-partition.

INPUT:

- `x` – anything.

OUTPUT:

- `True` if `x` is a `nef-partition` and `False` otherwise.

EXAMPLES:

```
sage: from sage.geometry.lattice_polytope import is_NefPartition
sage: is_NefPartition(1)
False
sage: o = lattice_polytope.cross_polytope(3)
sage: np = o.nef_partitions()[0]
sage: np
Nef-partition {0, 1, 3} U {2, 4, 5}
sage: is_NefPartition(np)
True
```

sage.geometry.lattice_polytope.**minkowski_sum**(*points1*, *points2*)

Compute the Minkowski sum of two convex polytopes.

Note: Polytopes might not be of maximal dimension.

INPUT:

- `points1, points2` - lists of objects that can be converted into vectors of the same dimension, treated as vertices of two polytopes.

OUTPUT: list of vertices of the Minkowski sum, given as vectors.

EXAMPLES: Let's compute the Minkowski sum of two line segments:

```
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:

```
sage: p = LatticePolytope([(1,0,0), (0,1,0),
...              (-1,-1,0), (0,0,1), (-1,0,-1)])
sage: p.points_pc()
M( 1,  0,  0),
M( 0,  1,  0),
M(-1, -1,  0),
M( 0,  0,  1),
M(-1,  0, -1),
M( 0,  0,  0)
in 3-d lattice M
```

We can compute linear relations between its points in the following way:

```
sage: p.points_pc().matrix().kernel().echelonized_basis_matrix()
[ 1  0  0  1  1  0]
[ 0  1  1 -1 -1  0]
[ 0  0  0  0  0  1]
```

However, the above relations may contain negative and rational numbers. This function transforms them in such a way, that all coefficients are non-negative integers:

```
sage: lattice_polytope.positive_integer_relations(p.points_pc().column_matrix())
[1 0 0 1 1 0]
[1 1 1 0 0 0]
[0 0 0 0 0 1]
sage: lattice_polytope.positive_integer_relations(ReflexivePolytope(2,1).vertices_pc().column_ma
[2 1 1]
```

sage.geometry.lattice_polytope.**projective_space**(*dim*)

Return a simplex of the given dimension, corresponding to P_{dim}.

EXAMPLES: We construct 3- and 4-dimensional simplexes:

```
sage: p = lattice_polytope.projective_space(3)
doctest:...: DeprecationWarning: this function is deprecated,
perhaps toric_varieties.P(n) is what you are looking for?
See http://trac.sagemath.org/15240 for details.
sage: p
3-d reflexive polytope in 3-d lattice M
sage: p.vertices_pc()
M( 1,  0,  0),
M( 0,  1,  0),
M( 0,  0,  1),
M(-1, -1, -1)
in 3-d lattice M
sage: p = lattice_polytope.projective_space(4)
sage: p
4-d reflexive polytope in 4-d lattice M
sage: p.vertices_pc()
M( 1,  0,  0,  0),
M( 0,  1,  0,  0),
M( 0,  0,  1,  0),
M( 0,  0,  0,  1),
M(-1, -1, -1, -1)
in 4-d lattice M
```

sage.geometry.lattice_polytope.**read_all_polytopes**(*file_name*, *desc=None*)

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)
sage: o = lattice_polytope.cross_polytope(3)
sage: result_name = lattice_polytope._palp("poly.x -fe", [d, o])
sage: with open(result_name) as f:
....:     print f.read()
4 2  Vertices of P-dual <-> Equations of P
  -1   1
   1   1
  -1  -1
   1  -1
8 3  Vertices of P-dual <-> Equations of P
  -1  -1   1
   1  -1   1
  -1   1   1
   1   1   1
  -1  -1  -1
   1  -1  -1
  -1   1  -1
   1   1  -1
sage: lattice_polytope.read_all_polytopes(result_name)
[
2-d reflexive polytope #14 in 2-d lattice M,
3-d reflexive polytope in 3-d lattice M
]
sage: os.remove(result_name)
```

sage.geometry.lattice_polytope.**read_palp_matrix**(*data*, *permutation=False*)

Read and return an integer matrix from a string or an opened file.

First input line must start with two integers m and n, the number of rows and columns of the matrix. The rest of the first line is ignored. The next m lines must contain n numbers each.

If m>n, returns the transposed matrix. If the string is empty or EOF is reached, returns the empty matrix, constructed by `matrix()`.

INPUT:

- **`data` – Either a string containing the filename or the file itself** containing the output by PALP.
- `permutation` – (default: `False`) If `True`, try to retrieve the permutation output by PALP. This parameter makes sense only when PALP computed the normal form of a lattice polytope.

OUTPUT:

A matrix or a tuple of a matrix and a permutation.

EXAMPLES:

```
sage: lattice_polytope.read_palp_matrix("2 3 comment \n 1 2 3 \n 4 5 6")
[1 2 3]
```

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

sage.geometry.lattice_polytope.**sage_matrix_to_maxima**(*m*)

Convert a Sage matrix to the string representation of Maxima.

EXAMPLE:

```
sage: m = matrix(ZZ,2)
sage: lattice_polytope.sage_matrix_to_maxima(m)
matrix([0,0],[0,0])
```

sage.geometry.lattice_polytope.**set_palp_dimension**(*d*)

Set the dimension for PALP calls to `d`.

INPUT:

- `d` – an integer from the list [4,5,6,11] or `None`.

OUTPUT:

- none.

PALP has many hard-coded limits, which must be specified before compilation, one of them is dimension. Sage includes several versions with different dimension settings (which may also affect other limits and enable certain features of PALP). You can change the version which will be used by calling this function. Such a change is not done automatically for each polytope based on its dimension, since depending on what you are doing it may be necessary to use dimensions higher than that of the input polytope.

EXAMPLES:

By default, it is not possible to create the 7-dimensional simplex with vertices at the basis of the 8-dimensional space:

```
sage: LatticePolytope(identity_matrix(8))
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: LatticePolytope(identity_matrix(8))
7-d lattice polytope in 8-d lattice M
```

Let's go back to default settings:

```
sage: lattice_polytope.set_palp_dimension(None)
```

sage.geometry.lattice_polytope.**skip_palp_matrix**(*data*, *n=1*)

Skip matrix data in a file.

INPUT:

- `data` - opened file with blocks of matrix data in the following format: A block consisting of m+1 lines has the number m as the first element of its first line.
- `n` - (default: 1) integer, specifies how many blocks should be skipped

If EOF is reached during the process, raises ValueError exception.

EXAMPLE: We create a file with vertices of the square and the cube, but read only the second set:

```
sage: d = lattice_polytope.cross_polytope(2)
sage: o = lattice_polytope.cross_polytope(3)
sage: result_name = lattice_polytope._palp("poly.x -fe", [d, o])
sage: with open(result_name) as f:
....:     print f.read()
4 2  Vertices of P-dual <-> Equations of P
  -1   1
   1   1
  -1  -1
   1  -1
8 3  Vertices of P-dual <-> Equations of P
  -1  -1   1
   1  -1   1
  -1   1   1
   1   1   1
  -1  -1  -1
   1  -1  -1
  -1   1  -1
   1   1  -1
sage: f = open(result_name)
sage: lattice_polytope.skip_palp_matrix(f)
sage: lattice_polytope.read_palp_matrix(f)
[-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:

```
sage: o = lattice_polytope.cross_polytope(3)
sage: lattice_polytope.write_palp_matrix(o.vertices_pc(), 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_pc(), format="%4d")
3 6
```

```
1    0    0   -1    0    0
0    1    0    0   -1    0
0    0    1    0    0   -1
```

CHAPTER

NINE

POLYHEDRA

In this module, a polyhedron is a convex (possibly unbounded) set in Euclidean space cut out by a finite set of linear inequalities and linear equations. Note that the dimension of the polyhedron can be less than the dimension of the ambient space. There are two complementary representations of the same data:

H(alf-space/Hyperplane)-representation This describes a polyhedron as the common solution set of a finite number of

- linear **inequalities** $A\vec{x} + b \geq 0$, and
- linear **equations** $C\vec{x} + d = 0$.

V(ertex)-representation The other representation is as the convex hull of vertices (and rays and lines to all for unbounded polyhedra) as generators. The polyhedron is then the Minkowski sum

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

where

- **vertices** $v_1, \ldots, v_k$ are a finite number of points. Each vertex is specified by an arbitrary vector, and two points are equal if and only if the vector is the same.
- **rays** $r_1, \ldots, r_m$ are a finite number of directions (directions of infinity). Each ray is specified by a 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))
```

9.1 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 Minkowsi sum presentation:

```
sage: half_plane = Polyhedron(ieqs=[(0,1,0)])
sage: half_plane.Hrepresentation()
(An inequality (1, 0) x + 0 >= 0,)
sage: half_plane.Vrepresentation()
(A line in the direction (0, 1), A ray in the direction (1, 0), A vertex at (0, 0))
```

Note how we need a point in the above example to anchor the ray and line. But any point on the boundary of the half-plane would serve the purpose just as well. Sage picked the origin here, but this choice is not unique. Similarly, the choice of ray is arbitrary but necessary to generate the half-plane.

Finally, note that while rays and lines generate unbounded edges of the polyhedron they are not in a one-to-one correspondence with them. For example, the infinite strip has two infinite edges (1-faces) but only one generating line:

```
sage: strip = Polyhedron(vertices=[(1,0),(-1,0)], lines=[(0,1)])
sage: strip.Hrepresentation()
(An inequality (1, 0) x + 1 >= 0, An inequality (-1, 0) x + 1 >= 0)
sage: strip.lines()
(A line in the direction (0, 1),)
sage: strip.faces(1)
(<0,1>, <0,2>)
sage: for face in strip.faces(1):
...       print face, '=', face.as_polyhedron().Vrepresentation()
<0,1> = (A line in the direction (0, 1), A vertex at (-1, 0))
<0,2> = (A line in the direction (0, 1), A vertex at (1, 0))
```

EXAMPLES:

```
sage: trunc_quadr = Polyhedron(vertices=[[1,0],[0,1]], rays=[[1,0],[0,1]])
sage: trunc_quadr
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices and 2 rays
sage: v = trunc_quadr.vertex_generator().next()  # 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_integer_dense.Vector_integer_dense'>
sage: v.polyhedron()
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices and 2 rays
sage: r = trunc_quadr.ray_generator().next()
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.

9.2 Base Rings

The base ring of the polyhedron can be specified by the `base_ring` optional keyword argument. If not specified, a suitable common base ring for all coordinates/coefficients will be chosen automatically. Important cases are:

- `base_ring=QQ` uses a fast implementation for exact rational numbers.
- `base_ring=ZZ` is similar to `QQ`, but the resulting polyhedron object will have extra methods for lattice polyhedra.
- `base_ring=RDF` uses floating point numbers, this is fast but susceptible to numerical errors.

Polyhedra with symmetries often are defined over some algebraic field extension of the rationals. As a simple example, consider the equilateral triangle whose vertex coordinates involve $\sqrt{3}$. An exact way to work with roots in Sage is the `Algebraic Real Field`

```
sage: triangle = Polyhedron([(0,0), (1,0), (1/2, sqrt(3)/2)], base_ring=AA)
sage: triangle.Hrepresentation()
(An inequality (-1, -0.5773502691896258?) x + 1 >= 0,
 An inequality (1, -0.5773502691896258?) x + 0 >= 0,
 An inequality (0, 1.154700538379252?) x + 0 >= 0)
```

Without specifying the `base_ring`, the `sqrt(3)` would be a symbolic ring element and, therefore, the polyhedron defined over the symbolic ring. This is possible as well, but rather slow:

```
sage: Polyhedron([(0,0), (1,0), (1/2, sqrt(3)/2)])
A 2-dimensional polyhedron in (Symbolic Ring)^2 defined as the convex
hull of 3 vertices
```

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)
sage: Polyhedron([(0,0), (1,0), (1/2, sqrt3/2)])
A 2-dimensional polyhedron in (Number Field in sqrt3 with defining
polynomial x^2 - 3)^2 defined as the convex hull of 3 vertices
```

9.3 Appendix

REFERENCES:

Komei Fukuda's FAQ in Polyhedral Computation

AUTHORS:

- Marshall Hampton: first version, bug fixes, and various improvements, 2008 and 2009

- Arnaud Bergeron: improvements to triangulation and rendering, 2008
- Sebastien Barthelemy: documentation improvements, 2008
- Volker Braun: refactoring, handle non-compact case, 2009 and 2010
- Andrey Novoseltsev: added Hasse_diagram_from_incidences, 2010
- Volker Braun: rewrite to use PPL instead of cddlib, 2011
- Volker Braun: Add support for arbitrary subfields of the reals

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

Construct a polyhedron object.

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

INPUT:

- `vertices` – list of point. Each point can be specified as any iterable container of `base_ring` elements. If `rays` or `lines` are specified but no `vertices`, the origin is taken to be the single vertex.
- `rays` – list of rays. Each ray can be specified as any iterable container of `base_ring` elements.
- `lines` – list of lines. Each line can be specified as any iterable container of `base_ring` elements.
- `ieqs` – list of inequalities. Each line can be specified as any iterable container of `base_ring` elements. An entry equal to `[-1,7,3,4]` represents the inequality $7x_1 + 3x_2 + 4x_3 \geq 1$.
- `eqns` – list of equalities. Each line can be specified as any iterable container of `base_ring` elements. An entry equal to `[-1,7,3,4]` represents the equality $7x_1 + 3x_2 + 4x_3 = 1$.
- `base_ring` – a sub-field of the reals implemented in Sage. The field over which the polyhedron will be defined. For `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 $\mathbf{Q}$ or $\mathbf{R}$ coefficients depending on `base_ring`.
  - `'ppl'`: use ppl (`backend_ppl`) with $\mathbf{Z}$ or $\mathbf{Q}$ coefficients depending on `base_ring`.
  - `'field'`: use python implementation (`backend_field`) for any field

Some backends support further optional arguments:

- `minimize` – boolean (default: `True`). Whether to immediately remove redundant H/V-representation data. Currently not used.
- `verbose` – boolean (default: `False`). Whether to print verbose output for debugging purposes. Only supported by the cdd backends.

OUTPUT:

The polyhedron defined by the input data.

EXAMPLES:

Construct some polyhedra:

```
sage: square_from_vertices = Polyhedron(vertices = [[1, 1], [1, -1], [-1, 1], [-1, -1]])
sage: square_from_ieqs = Polyhedron(ieqs = [[1, 0, 1], [1, 1, 0], [1, 0, -1], [1, -1, 0]])
sage: list(square_from_ieqs.vertex_generator())
[A vertex at (1, -1),
 A vertex at (1, 1),
 A vertex at (-1, 1),
 A vertex at (-1, -1)]
sage: list(square_from_vertices.inequality_generator())
[An inequality (1, 0) x + 1 >= 0,
 An inequality (0, 1) x + 1 >= 0,
 An inequality (-1, 0) x + 1 >= 0,
 An inequality (0, -1) x + 1 >= 0]
sage: p = Polyhedron(vertices = [[1.1, 2.2], [3.3, 4.4]], base_ring=RDF)
sage: p.n_inequalities()
2
```

The same polyhedron given in two ways:

```
sage: p = Polyhedron(ieqs = [[0,1,0,0],[0,0,1,0]])
sage: p.Vrepresentation()
(A line in the direction (0, 0, 1),
 A ray in the direction (1, 0, 0),
 A ray in the direction (0, 1, 0),
 A vertex at (0, 0, 0))
sage: q = Polyhedron(vertices=[[0,0,0]], rays=[[1,0,0],[0,1,0]], lines=[[0,0,1]])
sage: q.Hrepresentation()
(An inequality (1, 0, 0) x + 0 >= 0,
 An inequality (0, 1, 0) x + 0 >= 0)
```

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 $a + b + c + d + e + f = 31$

```
sage: positive_coords = Polyhedron(ieqs=[
...       [0, 1, 0, 0, 0, 0, 0], [0, 0, 1, 0, 0, 0, 0], [0, 0, 0, 1, 0, 0, 0],
...       [0, 0, 0, 0, 1, 0, 0], [0, 0, 0, 0, 0, 1, 0], [0, 0, 0, 0, 0, 0, 1]])
sage: P = Polyhedron(ieqs=positive_coords.inequalities() + (
...       [0,0,1,-1,-1,1,0], [0,0,-1,1,-1,1,0]), eqns=[[-31,1,1,1,1,1,1]])
sage: P
A 5-dimensional polyhedron in QQ^6 defined as the convex hull of 7 vertices
sage: P.dim()
5
sage: P.Vrepresentation()
(A vertex at (31, 0, 0, 0, 0, 0), A vertex at (0, 0, 0, 0, 0, 31),
 A vertex at (0, 0, 0, 0, 31, 0), A vertex at (0, 0, 31/2, 0, 31/2, 0),
 A vertex at (0, 31/2, 31/2, 0, 0, 0), A vertex at (0, 31/2, 0, 0, 31/2, 0),
 A vertex at (0, 0, 0, 31/2, 31/2, 0))
```

Note:

- Once constructed, a `Polyhedron` object is immutable.
- Although the option `field=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.

CHAPTER

TEN

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 = p.vertex_generator().next()
sage: v
A vertex at (0, 0, 0)
sage: a = p.equation_generator().next()
sage: a
An equation (0, 0, 1) x + 0 == 0
sage: a.contains(v)
True
```

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

NOTE:

Returns False for any equation.

EXAMPLES:

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

TESTS:

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

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

OUTPUT:

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

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[0,0,0],[1,1,0],[1,2,0]])
sage: repr_obj = p.equation_generator().next()
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()
Returns 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().

TESTS:

```
sage: p = Polyhedron(ieqs = [[0,0,0,2],[0,0,1,0,],[0,10,0,0],
...       [1,-1,0,0],[1,0,-1,0,],[1,0,0,-1]])
sage: pH = p.Hrepresentation(0)
sage: a = list(pH.neighbors())
sage: b = list(pH.adjacent())
sage: a==b
True
```

b()

Returns 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*)

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

NOTES:

- Evaluating on a vertex returns $A\vec{x} + b$
- Evaluating on a ray returns $A\vec{r}$. Only the sign or whether it is zero is meaningful.
- Evaluating on a line returns $A\vec{l}$. Only whether it is zero or not is meaningful.

EXAMPLES:

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

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

EXAMPLES:

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

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

EXAMPLES:

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

is_equation()
Returns True if the object is an equation of the H-representation.

EXAMPLES:

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

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

EXAMPLES:

```
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()
Returns 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/equations)

EXAMPLES:

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

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

interior_contains(*Vobj*)

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

EXAMPLES:

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

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

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

is_inequality()

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

EXAMPLES:

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

type()

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

OUTPUT:

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

EXAMPLES:

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

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

A line (Minkowski summand $\simeq \mathbf{R}$) of the polyhedron. Inherits from Vrepresentation.

evaluated_on(*Hobj*)
Returns $A\vec{\ell}$

EXAMPLES:

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

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

TESTS:

```
sage: p = Polyhedron(ieqs = [[1, 0, 0, 1],[1,1,0,0]])
sage: a = p.line_generator().next()
sage: a.is_line()
True
```

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

OUTPUT:

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

EXAMPLES:

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

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

TESTS:

```
sage: import sage.geometry.polyhedron.representation as P
sage: P.PolyhedronRepresentation()
<class 'sage.geometry.polyhedron.representation.PolyhedronRepresentation'>
```

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()

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

NOTES:

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

EXAMPLES:

```
sage: s = Polyhedron(vertices=[[1],[-1]])
sage: first_vertex = s.vertex_generator().next()
sage: first_vertex.index()
0
sage: first_vertex == s.Vrepresentation(0)
True
```

polyhedron()

Returns the underlying polyhedron.

TESTS:

```
sage: p = Polyhedron(vertices=[[1,2,3]])
sage: v = p.Vrepresentation(0)
sage: v.polyhedron()
A 0-dimensional polyhedron in ZZ^3 defined as the convex hull of 1 vertex
```

vector(*base_ring=None*)

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

INPUT:

- `base_ring` – the base ring of the vector.

OUTPUT:

For a V-representation object, a vector of length `ambient_dim()`. For a H-representation object, a vector of length `ambient_dim()` + 1.

EXAMPLES:

```
sage: s = polytopes.cuboctahedron()
sage: v = s.vertex_generator().next()
sage: v
A vertex at (-1/2, -1/2, 0)
sage: v.vector()
(-1/2, -1/2, 0)
sage: v()
(-1/2, -1/2, 0)
sage: type(v())
<type 'sage.modules.vector_rational_dense.Vector_rational_dense'>
```

Conversion to a different base ring can be forced with the optional argument:

```
sage: v.vector(RDF)
(-0.5, -0.5, 0.0)
sage: vector(RDF, v)
(-0.5, -0.5, 0.0)
```

class `sage.geometry.polyhedron.representation.`**Ray**(*polyhedron_parent*)

Bases: `sage.geometry.polyhedron.representation.Vrepresentation`

A ray of the polyhedron. Inherits from `Vrepresentation`.

evaluated_on(*Hobj*)

Returns $A\vec{r}$

EXAMPLES:

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

type()

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

OUTPUT:

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

EXAMPLES:

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

```
3
sage: repr_obj.type() == repr_obj.INEQUALITY
False
sage: repr_obj.type() == repr_obj.EQUATION
False
sage: repr_obj.type() == repr_obj.VERTEX
False
sage: repr_obj.type() == repr_obj.RAY
True
sage: repr_obj.type() == repr_obj.LINE
False
```

class sage.geometry.polyhedron.representation.**Vertex**(*polyhedron_parent*)

Bases: sage.geometry.polyhedron.representation.Vrepresentation

A vertex of the polyhedron. Inherits from Vrepresentation.

evaluated_on(*Hobj*)

Returns $A\vec{x} + b$

EXAMPLES:

```
sage: p = polytopes.n_cube(3)
sage: v = p.vertex_generator().next()
sage: h = p.inequality_generator().next()
sage: v
A vertex at (-1, -1, -1)
sage: h
An inequality (0, 0, -1) x + 1 >= 0
sage: v.evaluated_on(h)
2
```

is_integral()

Return whether the coordinates of the vertex are all integral.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: p = Polyhedron([(1/2,3,5), (0,0,0), (2,3,7)])
sage: [ v.is_integral() for v in p.vertex_generator() ]
[True, False, True]
```

is_vertex()

Tests if this object is a vertex. By construction it always is.

EXAMPLES:

```
sage: p = Polyhedron(ieqs = [[0,0,1],[0,1,0],[1,-1,0]])
sage: a = p.vertex_generator().next()
sage: a.is_vertex()
True
```

type()

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

OUTPUT:

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

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[0,0,0],[1,1,0],[1,2,0]])
sage: repr_obj = p.vertex_generator().next()
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().

TESTS:

```
sage: p = Polyhedron(vertices = [[0,0],[1,0],[0,3],[1,4]])
sage: v = p.vertex_generator().next()
sage: a = v.neighbors().next()
sage: b = v.adjacent().next()
sage: a==b
True
```

incident()

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

EXAMPLES:

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

is_V()

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

EXAMPLES:

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

```
sage: v.is_V()
True
```

is_incident(*Hobj*)

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

EXAMPLES:

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

is_line()

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

EXAMPLES:

```
sage: p = Polyhedron(ieqs = [[1, 0, 0, 0, 1], [1, 1, 0, 0, 0], [1, 0, 1, 0, 0]])
sage: line1 = p.line_generator().next()
sage: line1.is_line()
True
sage: v1 = p.vertex_generator().next()
sage: v1.is_line()
False
```

is_ray()

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

EXAMPLES:

```
sage: p = Polyhedron(ieqs = [[1, 0, 0, 0, 1], [1, 1, 0, 0, 0], [1, 0, 1, 0, 0]])
sage: r1 = p.ray_generator().next()
sage: r1.is_ray()
True
sage: v1 = p.vertex_generator().next()
sage: v1
A vertex at (-1, -1, 0, -1)
sage: v1.is_ray()
False
```

is_vertex()

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

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[0,0],[1,0],[0,3],[1,3]])
sage: v = p.vertex_generator().next()
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 = p.ray_generator().next()
```

```
sage: r1.is_vertex()
False
```

neighbors()

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

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[0,0],[1,0],[0,3],[1,4]])
sage: v = p.vertex_generator().next()
sage: v.neighbors().next()
A vertex at (0, 3)
```

CHAPTER

ELEVEN

LIBRARY OF COMMONLY USED, FAMOUS, OR INTERESTING POLYTOPES

REFERENCES:

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

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

TESTS:

```
sage: TestSuite(polytopes).run(skip='_test_pickling')
```

Birkhoff_polytope(*n*)

Return the Birkhoff polytope with n! vertices. Each vertex is a (flattened) n by n permutation matrix.

INPUT:

- n – a positive integer giving the size of the permutation matrices.

EXAMPLES:

```
sage: b3 = polytopes.Birkhoff_polytope(3)
sage: b3.n_vertices()
6
```

Kirkman_icosahedron()

A non-uniform icosahedron with interesting properties.

See [Fetter2012] for details.

OUTPUT:

The Kirkman icosahedron, a 3-dimensional polyhedron with 20 vertices, 20 faces, and 38 edges.

EXAMPLES:

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

buckyball(*base_ring=Rational Field*)

Also known as the truncated icosahedron, an Archimedean solid. It has 32 faces and 60 vertices. Rational coordinates are not exact.

EXAMPLES:

```
sage: bb = polytopes.buckyball()
sage: bb.n_vertices()
60
sage: bb.n_inequalities()   # number of facets
32
sage: bb.base_ring()
Rational Field
```

cross_polytope(*dim_n*)

Return a cross-polytope in dimension `dim_n`. These are the generalization of the octahedron.

INPUT:

- `dim_n` – integer. The dimension of the cross-polytope.

OUTPUT:

A Polyhedron object of the `dim_n`-dimensional cross-polytope, with exact coordinates.

EXAMPLES:

```
sage: four_cross = polytopes.cross_polytope(4)
sage: four_cross.is_simple()
False
sage: four_cross.n_vertices()
8
```

cuboctahedron()

An Archimedean solid with 12 vertices and 14 faces. Dual to the rhombic dodecahedron.

EXAMPLES:

```
sage: co = polytopes.cuboctahedron()
sage: co.n_vertices()
12
sage: co.n_inequalities()
14
```

cyclic_polytope(*dim_n*, *points_n*, *base_ring=Rational Field*)

Return a cyclic polytope.

INPUT:

- `dim_n` – positive integer. the dimension of the polytope.
- `points_n` – positive integer. the number of vertices.
- `base_ring` – either `QQ` (default) or `RDF`.

OUTPUT:

A cyclic polytope of dim_n with points_n vertices on the moment curve `(t,t^2,...,t^n)`, as Polyhedron object.

EXAMPLES:

```
sage: c = polytopes.cyclic_polytope(4,10)
sage: c.n_inequalities()
35
```

dodecahedron (*base_ring=Rational Field*)
Return a dodecahedron.

INPUT:

•`base_ring` – Either `QQ` (in which case a rational approximation to the golden ratio is used) or `RDF`.

EXAMPLES:

```
sage: d12 = polytopes.dodecahedron()
sage: d12.n_inequalities()
12
```

great_rhombicuboctahedron (*base_ring=Rational Field*)
Return an Archimedean solid with 48 vertices and 26 faces.

EXAMPLES:

```
sage: gr = polytopes.great_rhombicuboctahedron()
sage: gr.n_vertices()
48
sage: gr.n_inequalities()
26
```

hypersimplex (*dim_n*, *k*, *project=True*)
The hypersimplex in dimension dim_n with d choose k vertices, projected into (dim_n - 1) dimensions.

INPUT:

•`n` – the numbers `(1,...,n)` are permuted

•`project` – If `False`, the polyhedron is left in dimension `n`.

OUTPUT:

A Polyhedron object representing the hypersimplex.

EXAMPLES:

```
sage: h_4_2 = polytopes.hypersimplex(4,2) # combinatorially equivalent to octahedron
sage: h_4_2.n_vertices()
6
sage: h_4_2.n_inequalities()
8
```

icosahedron (*base_ring=Rational Field*)
Return an icosahedron with edge length 1.

INPUT:

•`base_ring` – Either `QQ` or `RDF`.

OUTPUT:

A Polyhedron object of a floating point or rational approximation to the regular 3d icosahedron.

If `base_ring=QQ`, a rational approximation is used and the points are not exactly the vertices of the icosahedron. The icosahedron's coordinates contain the golden ratio, so there is no exact representation possible.

EXAMPLES:

```
sage: ico = polytopes.icosahedron()
sage: sum(sum( ico.vertex_adjacency_matrix() ))/2
30
```

n_cube(*dim_n*)

Return a cube in the given dimension

INPUT:

- dim_n – integer. The dimension of the cube.

OUTPUT:

A Polyhedron object of the dim_n-dimensional cube, with exact coordinates.

EXAMPLES:

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

n_simplex(*dim_n=3*, *project=True*)

Return a rational approximation to a regular simplex in dimension dim_n.

INPUT:

- dim_n – The dimension of the cross-polytope, a positive integer.
- project – Optional argument, whether to project orthogonally. Default is True.

OUTPUT:

A Polyhedron object of the dim_n-dimensional simplex.

EXAMPLES:

```
sage: s5 = polytopes.n_simplex(5)
sage: s5.dim()
5
```

static orthonormal_1(*dim_n=5*)

A matrix of rational approximations to orthonormal vectors to (1,...,1).

INPUT:

- dim_n - the dimension of the vectors

OUTPUT:

A matrix over QQ whose rows are close to an orthonormal basis to the subspace normal to (1,...,1).

EXAMPLES:

```
sage: from sage.geometry.polyhedron.library import Polytopes
sage: m = Polytopes.orthonormal_1(5)
sage: m
[ 70711/100000   -7071/10000             0             0             0]
[    1633/4000     1633/4000 -81649/100000             0             0]
[   7217/25000    7217/25000    7217/25000  -43301/50000             0]
[ 22361/100000  22361/100000  22361/100000  22361/100000  -44721/50000]
```

parallelotope(*generators*)

Return the parallelotope spanned by the generators.

INPUT:

- generators – an iterable of anything convertible to vector (for example, a list of vectors) such that the vectors all have the same dimension.

OUTPUT:

The parallelotope. This is the multi-dimensional generalization of a parallelogram (2 generators) and a parallelepiped (3 generators).

EXAMPLES:

```
sage: polytopes.parallelotope([ (1,0), (0,1) ])
A 2-dimensional polyhedron in QQ^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 QQ^4 defined as the convex hull of 16 vertices
```

pentakis_dodecahedron()

This face-regular, vertex-uniform polytope is dual to the truncated icosahedron. It has 60 faces and 32 vertices.

EXAMPLES:

```
sage: pd = polytopes.pentakis_dodecahedron()
sage: pd.n_vertices()
32
sage: pd.n_inequalities()   # number of facets
60
```

permutahedron(*n*, *project=True*)

The standard permutahedron of (1,...,n) projected into n-1 dimensions.

INPUT:

- `n` – the numbers `(1,...,n)` are permuted
- `project` – If `False` the polyhedron is left in dimension `n`.

OUTPUT:

A Polyhedron object representing the permutahedron.

EXAMPLES:

```
sage: perm4 = polytopes.permutahedron(4)
sage: perm4
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 24 vertices
sage: polytopes.permutahedron(5).show()    # long time
```

static project_1(*fpoint*)

Take a ndim-dimensional point and projects it onto the plane perpendicular to (1,1,...,1).

INPUT:

- `fpoint` - a list of ndim numbers

EXAMPLES:

```
sage: from sage.geometry.polyhedron.library import Polytopes
sage: Polytopes.project_1([1,1,1,1,2])
[1/100000, 1/100000, 1/50000, -559/625]
```

regular_polygon(*n*, *base_ring=Rational Field*)

Return a regular polygon with n vertices. Over the rational field the vertices may not be exact.

INPUT:

- `n` – a positive integer, the number of vertices.
- `field` – either `QQ` or `RDF`.

EXAMPLES:

```
sage: octagon = polytopes.regular_polygon(8)
sage: len(octagon.vertices())
8
```

rhombic_dodecahedron()

This face-regular, vertex-uniform polytope is dual to the cuboctahedron. It has 14 vertices and 12 faces.

EXAMPLES:

```
sage: rd = polytopes.rhombic_dodecahedron()
sage: rd.n_vertices()
14
sage: rd.n_inequalities()
12
```

six_hundred_cell()

Return the standard 600-cell polytope.

OUTPUT:

A Polyhedron object of the 4-dimensional 600-cell, a regular polytope. In many ways this is an analogue of the icosahedron. The coordinates of this polytope are rational approximations of the true coordinates of the 600-cell, some of which involve the (irrational) golden ratio.

EXAMPLES:

```
sage: p600 = polytopes.six_hundred_cell() # not tested - very long time
sage: len(list(p600.bounded_edges())) # not tested - very long time
120
```

small_rhombicuboctahedron()

Return an Archimedean solid with 24 vertices and 26 faces.

EXAMPLES:

```
sage: sr = polytopes.small_rhombicuboctahedron()
sage: sr.n_vertices()
24
sage: sr.n_inequalities()
26
```

twenty_four_cell()

Return the standard 24-cell polytope.

OUTPUT:

A Polyhedron object of the 4-dimensional 24-cell, a regular polytope. The coordinates of this polytope are exact.

EXAMPLES:

```
sage: p24 = polytopes.twenty_four_cell()
sage: v = p24.vertex_generator().next()
sage: for adj in v.neighbors(): print adj
A vertex at (-1/2, -1/2, -1/2, 1/2)
A vertex at (-1/2, -1/2, 1/2, -1/2)
A vertex at (-1, 0, 0, 0)
A vertex at (-1/2, 1/2, -1/2, -1/2)
A vertex at (0, -1, 0, 0)
A vertex at (0, 0, -1, 0)
A vertex at (0, 0, 0, -1)
A vertex at (1/2, -1/2, -1/2, -1/2)
```

CHAPTER

TWELVE

FUNCTIONS FOR PLOTTING POLYHEDRA

class sage.geometry.polyhedron.plot.**Projection**(*polyhedron*, *proj=<function projection_func_identity at 0x7fb30a162320>*)

Bases: sage.structure.sage_object.SageObject

The projection of a Polyhedron.

This class keeps track of the necessary data to plot the input polyhedron.

coord_index_of(*v*)

Convert a coordinate vector to its internal index.

EXAMPLES:

```
sage: p = polytopes.n_cube(3)
sage: proj = p.projection()
sage: proj.coord_index_of(vector((1,1,1)))
7
```

coord_indices_of(*v_list*)

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

EXAMPLES:

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

coordinates_of(*coord_index_list*)

Given a list of indices, return the projected coordinates.

EXAMPLES:

```
sage: p = polytopes.n_simplex(4).projection()
sage: p.coordinates_of([1])
[[0, -81649/100000, 7217/25000, 22361/100000]]
```

identity()

Return the identity projection of the polyhedron.

EXAMPLES:

```
sage: p = polytopes.icosahedron()
sage: from sage.geometry.polyhedron.plot import Projection
sage: pproj = Projection(p)
```

```
sage: ppid = pproj.identity()
sage: ppid.dimension
3
```

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

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

INPUT:

See `plot()`.

OUTPUT:

A 2-d graphics object.

EXAMPLES:

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

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

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

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

INPUT:

See `plot()`.

OUTPUT:

A 2-d graphics object.

EXAMPLES:

```
sage: Polyhedron([(0,), (1,)]).projection().render_1d()
```

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

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

EXAMPLES:

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

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

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

EXAMPLES:

```
sage: p1 = Polyhedron(vertices=[[1,1,1]], rays=[[1,1,1]])
sage: p2 = Polyhedron(vertices=[[2,0,0], [0,2,0], [0,0,2]])
sage: p3 = Polyhedron(vertices=[[1,0,0], [0,1,0], [0,0,1]], rays=[[-1,-1,-1]])
sage: p1.projection().plot() + p2.projection().plot() + p3.projection().plot()
```

It correctly handles various degenerate cases:

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

render_fill_2d(***kwds*)

Return the filled interior (a polygon) of a polyhedron in 2d.

EXAMPLES:

```
sage: cps = [i^3 for i in srange(-2,2,1/5)]
sage: p = Polyhedron(vertices = [[(t^2-1)/(t^2+1),2*t/(t^2+1)] for t in cps])
sage: proj = p.projection()
sage: filled_poly = proj.render_fill_2d()
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.n_cube(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.n_cube(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.n_cube(3).projection()
sage: p_solid = p.render_solid_3d(opacity = .7)
sage: type(p_solid)
<class 'sage.plot.plot3d.base.Graphics3dGroup'>
```

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.n_cube(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(*projection_direction=None*, *height=1.1*)

Return the Schlegel projection.

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

INPUT:

- `projection_direction` – coordinate list/tuple/iterable or `None` (default). The direction of the

Schlegel projection. For a full-dimensional polyhedron, the default is the first facet normal; Otherwise, the vector consisting of the first n primes is chosen.

- `height` – float (default: 1.1). How far outside of the unit sphere the focal point is.

EXAMPLES:

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

TESTS:

```
sage: Projection(cube4).schlegel()
The projection of a polyhedron into 3 dimensions
```

show(**args*, ***kwds*)

x.__init__(...) initializes x; see help(type(x)) for signature

stereographic(*projection_point=None*)

Return the stereographic projection.

INPUT:

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

EXAMPLES:

```
sage: from sage.geometry.polyhedron.plot import Projection
sage: proj = Projection(polytopes.buckyball())  #long time
sage: proj                                      #long time
The projection of a polyhedron into 3 dimensions
sage: proj.stereographic([5,2,3]).plot()        #long time
sage: Projection( polytopes.twenty_four_cell() ).stereographic([2,0,0,0])
The projection of a polyhedron into 3 dimensions
```

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

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

INPUT:

- `view` - list (default: [0,0,1]) representing the rotation axis (see note below).
- `angle` - integer (default: 0) angle of rotation in degree from 0 to 360 (see note below).
- `scale` - integer (default: 2) specifying the scaling of the tikz picture.
- `edge_color` - string (default: 'blue!95!black') representing colors which tikz recognize.
- `facet_color` - string (default: 'blue!95!black') representing colors which tikz recognize.
- `vertex_color` - string (default: 'green') representing colors which tikz recognize.
- `opacity` - real number (default: 0.8) between 0 and 1 giving the opacity of the front facets.
- `axis` - Boolean (default: False) draw the axes at the origin or not.

OUTPUT:

- LatexExpr – containing the TikZ picture.

Note: The inputs `view` and `angle` can be obtained from the viewer Jmol:

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

It reads something like:

```
moveto 0.0 {x y z angle} Scale
```

The `view` is then [x,y,z] and `angle` is angle. The following number is the scale.

Jmol performs a rotation of `angle` degrees along the vector [x,y,z] and show the result from the z-axis.

EXAMPLES:

```
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: open('polytope-tikz1.tex', 'w').write(Image1)    # not tested

sage: P2 = Polyhedron(vertices=[[1, 1],[1, 2],[2, 1]])
sage: Image2 = P2.projection().tikz(scale=3, edge_color='blue!95!black', facet_color='orange
sage: type(Image2)
<class 'sage.misc.latex.LatexExpr'>
sage: print '\n'.join(Image2.splitlines()[:4])
\begin{tikzpicture}%
    [scale=3.000000,
    back/.style={loosely dotted, thin},
    edge/.style={color=blue!95!black, thick},
sage: open('polytope-tikz2.tex', 'w').write(Image2)    # not tested

sage: P3 = Polyhedron(vertices=[[-1, -1, 2],[-1, 2, -1],[2, -1, -1]])
sage: P3
A 2-dimensional polyhedron in ZZ^3 defined as the convex hull of 3 vertices
sage: Image3 = P3.projection().tikz([0.5,-1,-0.1], 55, scale=3, edge_color='blue!95!black',f
sage: print '\n'.join(Image3.splitlines()[:4])
\begin{tikzpicture}%
    [x={(0.658184cm, -0.242192cm)},
    y={(-0.096240cm, 0.912008cm)},
    z={(-0.746680cm, -0.331036cm)},
sage: open('polytope-tikz3.tex', 'w').write(Image3)    # not tested

sage: P=Polyhedron(vertices=[[1,1,0,0],[1,2,0,0],[2,1,0,0],[0,0,1,0],[0,0,0,1]])
sage: P
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 5 vertices
sage: P.projection().tikz()
Traceback (most recent call last):
...
NotImplementedError: The polytope has to live in 2 or 3 dimensions.
```

Todo

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

```
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.
```

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

---

class sage.geometry.polyhedron.plot.**ProjectionFuncSchlegel**(*projection_direction*, *height=1.1*, *center=0*)

The Schlegel projection from the given input point.

EXAMPLES:

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

class sage.geometry.polyhedron.plot.**ProjectionFuncStereographic**(*projection_point*)

The stereographic (or perspective) projection.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.plot import ProjectionFuncStereographic
sage: cube = polytopes.n_cube(3).vertices()
sage: proj = ProjectionFuncStereographic([1.2, 3.4, 5.6])
sage: ppoints = [proj(vector(x)) for x in cube]
sage: ppoints[0]
(-0.0486511..., 0.0859565...)
```

sage.geometry.polyhedron.plot.**cyclic_sort_vertices_2d**(*Vlist*)

Return the vertices/rays in cyclic order if possible.

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

See `vertex_adjacency_matrix()` for a discussion of "adjacent".

EXAMPLES:

```
sage: from sage.geometry.polyhedron.plot import cyclic_sort_vertices_2d
sage: square = Polyhedron([[1,0],[-1,0],[0,1],[0,-1]])
sage: vertices = [v for v in square.vertex_generator()]
sage: vertices
[A vertex at (-1, 0),
 A vertex at (0, -1),
 A vertex at (0, 1),
 A vertex at (1, 0)]
sage: cyclic_sort_vertices_2d(vertices)
[A vertex at (1, 0),
 A vertex at (0, -1),
 A vertex at (-1, 0),
 A vertex at (0, 1)]
```

Rays are allowed, too:

```
sage: P = Polyhedron(vertices=[(0, 1), (1, 0), (2, 0), (3, 0), (4, 1)], rays=[(0,1)])
sage: P.adjacency_matrix()
[0 1 0 1 0]
[1 0 1 0 0]
[0 1 0 0 1]
[1 0 0 0 1]
[0 0 1 1 0]
sage: cyclic_sort_vertices_2d(P.Vrepresentation())
[A vertex at (3, 0),
 A vertex at (1, 0),
 A vertex at (0, 1),
 A ray in the direction (0, 1),
 A vertex at (4, 1)]

sage: P = Polyhedron(vertices=[(0, 1), (1, 0), (2, 0), (3, 0), (4, 1)], rays=[(0,1), (1,1)])
sage: P.adjacency_matrix()
[0 1 0 0 0]
[1 0 1 0 0]
[0 1 0 0 1]
[0 0 0 0 1]
[0 0 1 1 0]
sage: cyclic_sort_vertices_2d(P.Vrepresentation())
[A ray in the direction (1, 1),
 A vertex at (3, 0),
 A vertex at (1, 0),
 A vertex at (0, 1),
 A ray in the direction (0, 1)]

sage: P = Polyhedron(vertices=[(1,2)], rays=[(0,1)], lines=[(1,0)])
sage: P.adjacency_matrix()
[0 0 1]
[0 0 0]
[1 0 0]
sage: cyclic_sort_vertices_2d(P.Vrepresentation())
[A vertex at (0, 2),
 A line in the direction (1, 0),
 A ray in the direction (0, 1)]
```

sage.geometry.polyhedron.plot.**projection_func_identity**(*x*)

The identity projection.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.plot import projection_func_identity
sage: projection_func_identity((1,2,3))
[1, 2, 3]
```

sage.geometry.polyhedron.plot.**render_2d**(*projection*, **args*, ***kwds*)

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

EXAMPLES:

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

```
sage: q4 = p4.projection()
sage: q1.plot() + q2.plot() + q3.plot() + q4.plot()
sage: from sage.geometry.polyhedron.plot import render_2d
sage: q = render_2d(p1.projection())
doctest:...: DeprecationWarning: use Projection.render_2d instead
See http://trac.sagemath.org/16625 for details.
sage: q._objects
[Point set defined by 1 point(s),
 Arrow from (1.0,1.0) to (2.0,2.0),
 Polygon defined by 3 points]
```

sage.geometry.polyhedron.plot.**render_3d**(*projection*, **args*, ***kwds*)

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

Note: This method, `render_3d`, is used in the `show()` method of a polyhedron if it is in 3 dimensions.

EXAMPLES:

```
sage: p1 = Polyhedron(vertices=[[1,1,1]], rays=[[1,1,1]])
sage: p2 = Polyhedron(vertices=[[2,0,0], [0,2,0], [0,0,2]])
sage: p3 = Polyhedron(vertices=[[1,0,0], [0,1,0], [0,0,1]], rays=[[-1,-1,-1]])
sage: p1.projection().plot() + p2.projection().plot() + p3.projection().plot()
```

It correctly handles various degenerate cases:

```
sage: Polyhedron(lines=[[1,0,0],[0,1,0],[0,0,1]]).plot()                                  # whole sp
sage: Polyhedron(vertices=[[1,1,1]], rays=[[1,0,0]], lines=[[0,1,0],[0,0,1]]).plot()      # half spa
sage: Polyhedron(vertices=[[1,1,1]], lines=[[0,1,0],[0,0,1]]).plot()                      # R^2 in R
sage: Polyhedron(rays=[[0,1,0],[0,0,1]], lines=[[1,0,0]]).plot()                          # quadrant
sage: Polyhedron(rays=[[0,1,0]], lines=[[1,0,0]]).plot()                                  # upper ha
sage: Polyhedron(lines=[[1,0,0]]).plot()                                                  # R^1 in R
sage: Polyhedron(rays=[[0,1,0]]).plot()                                                   # Half-lin
sage: Polyhedron(vertices=[[1,1,1]]).plot()                                               # point in
```

sage.geometry.polyhedron.plot.**render_4d**(*polyhedron*, *point_opts={}*, *line_opts={}*, *polygon_opts={}*, *projection_direction=None*)

Return a 3d rendering of the Schlegel projection of a 4d polyhedron projected into 3-dimensional space.

Note: The `show()` method of `Polyhedron()` uses this to draw itself if the ambient dimension is 4.

INPUT:

- `polyhedron` – A `Polyhedron` object.
- `point_opts`, `line_opts`, `polygon_opts` – dictionaries of plot keywords or `False` to disable.
- `projection_direction` – list/tuple/iterable of coordinates or `None` (default). Sets the projetion direction of the Schlegel projection. If it is not given, the center of a facet is used.

EXAMPLES:

```
sage: poly = polytopes.twenty_four_cell()
sage: poly
A 4-dimensional polyhedron in QQ^4 defined as the convex hull of 24 vertices
sage: poly.plot()
sage: poly.plot(projection_direction=[2,5,11,17])
sage: type( poly.plot() )
<class 'sage.plot.plot3d.base.Graphics3dGroup'>
```

TESTS:

```
sage: from sage.geometry.polyhedron.plot import render_4d
sage: p = polytopes.n_cube(4)
sage: q = render_4d(p)
doctest:...: DeprecationWarning: use Polyhedron.schlegel_projection instead
See http://trac.sagemath.org/16625 for details.
doctest:...: DeprecationWarning: use Projection.render_3d instead
See http://trac.sagemath.org/16625 for details.
sage: tach_str = q.tachyon()
sage: tach_str.count('FCylinder')
32
```

CHAPTER

THIRTEEN

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)
(<0,1,2,3,4,5>,)
sage: P.faces(2)
(<0,1,2>, <0,1,3>, <0,2,4>, <0,3,4>, <3,4,5>, <2,4,5>, <1,3,5>, <1,2,5>)
sage: P.faces(1)
(<0,1>, <0,2>, <1,2>, <0,3>, <1,3>, <0,4>, <2,4>, <3,4>, <2,5>, <3,5>, <4,5>, <1,5>)
```

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

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

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

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

```
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)
```

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

Bases: `sage.structure.sage_object.SageObject`

A face of a polyhedron.

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

INPUT:

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

OUTPUT:

A `PolyhedronFace`.

EXAMPLES:

```
sage: octahedron = polytopes.cross_polytope(3)
sage: inequality = octahedron.Hrepresentation(2)
sage: face_h = tuple([ inequality ])
sage: face_v = tuple( inequality.incident() )
sage: face_h_indices = [ h.index() for h in face_h ]
sage: face_v_indices = [ v.index() for v in face_v ]
sage: from sage.geometry.polyhedron.face import PolyhedronFace
sage: face = PolyhedronFace(octahedron, face_v_indices, face_h_indices)
sage: face
<0,1,2>
sage: face.dim()
2
sage: face.ambient_Hrepresentation()
(An inequality (1, 1, 1) x + 1 >= 0,)
sage: face.ambient_Vrepresentation()
(A vertex at (-1, 0, 0), A vertex at (0, -1, 0), A vertex at (0, 0, -1))
```

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.n_cube(2)
sage: for face in square.face_lattice():
...       print face.ambient_Hrepresentation()
(An inequality (1, 0) x + 1 >= 0, An inequality (0, 1) x + 1 >= 0,
 An inequality (-1, 0) x + 1 >= 0, An inequality (0, -1) x + 1 >= 0)
(An inequality (1, 0) x + 1 >= 0, An inequality (0, 1) x + 1 >= 0)
(An inequality (1, 0) x + 1 >= 0, An inequality (0, -1) x + 1 >= 0)
(An inequality (0, 1) x + 1 >= 0, An inequality (-1, 0) x + 1 >= 0)
(An inequality (-1, 0) x + 1 >= 0, An inequality (0, -1) x + 1 >= 0)
(An inequality (1, 0) x + 1 >= 0,)
(An inequality (0, 1) x + 1 >= 0,)
(An inequality (-1, 0) x + 1 >= 0,)
(An inequality (0, -1) x + 1 >= 0,)
()
```

ambient_Vrepresentation(*index=None*)

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

INPUT:

- `index` – optional. Either an integer or `None` (default).

OUTPUT:

If the optional argument is not present, a tuple of V-representation objects. Each entry is either a vertex, a ray, or a line.

If the optional integer `index` is specified, the `index`-th element of the tuple is returned.

EXAMPLES:

```
sage: square = polytopes.n_cube(2)
sage: for fl in square.face_lattice():
...       print fl.ambient_Vrepresentation()
...
()
(A vertex at (-1, -1),)
(A vertex at (-1, 1),)
(A vertex at (1, -1),)
(A vertex at (1, 1),)
(A vertex at (-1, -1), A vertex at (-1, 1))
(A vertex at (-1, -1), A vertex at (1, -1))
(A vertex at (1, -1), A vertex at (1, 1))
(A vertex at (-1, 1), A vertex at (1, 1))
(A vertex at (-1, -1), A vertex at (-1, 1),
 A vertex at (1, -1), A vertex at (1, 1))
```

ambient_dim()

Return the dimension of the containing polyhedron.

EXAMPLES:

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

as_polyhedron()

Return the face as an independent polyhedron.

OUTPUT:

A polyhedron.

EXAMPLES:

```
sage: P = polytopes.cross_polytope(3);  P
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 6 vertices
sage: face = P.faces(2)[3]
sage: face
<0,3,4>
sage: face.as_polyhedron()
A 2-dimensional polyhedron in ZZ^3 defined as the convex hull of 3 vertices

sage: P.intersection(face.as_polyhedron()) == face.as_polyhedron()
True
```

dim()
Return the dimension of the face.

OUTPUT:

Integer.

EXAMPLES:

```
sage: fl = polytopes.dodecahedron().face_lattice()
sage: [ x.dim() for x in fl ]
[-1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
  1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 3]
```

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: face.line_generator().next()
A line in the direction (1, 0)
```

lines()
Return all lines of the face.

OUTPUT:

A tuple of lines.

EXAMPLES:

```
sage: p = Polyhedron(rays = [[1,0],[-1,0],[0,1],[1,1]], vertices = [[-2,-2],[2,3]])
sage: p.lines()
(A line in the direction (1, 0),)
```

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()[10]
sage: face
<0,2>
sage: face.ambient_Hrepresentation()
(An inequality (1, -1, 1, -1) x + 1 >= 0,
 An inequality (1, 1, 1, 1) x + 1 >= 0,
 An inequality (1, 1, 1, -1) x + 1 >= 0,
 An inequality (1, -1, 1, 1) x + 1 >= 0)
sage: face.n_ambient_Hrepresentation()
4
```

n_ambient_Vrepresentation()
Return the number of objects that make up the ambient V-representation of the polyhedron.

See also ambient_Vrepresentation().

OUTPUT:

Integer.

EXAMPLES:

```
sage: p = polytopes.cross_polytope(4)
sage: face = p.face_lattice()[10]
sage: face
<0,2>
sage: face.ambient_Vrepresentation()
(A vertex at (-1, 0, 0, 0), A vertex at (0, 0, -1, 0))
sage: face.n_ambient_Vrepresentation()
2
```

polyhedron()

Return the containing polyhedron.

EXAMPLES:

```
 sage: P = polytopes.cross_polytope(3); P
 A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 6 vertices
sage: face = P.faces(2)[3]
 sage: face
 <0,3,4>
 sage: face.polyhedron()
 A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 6 vertices
```

ray_generator()

Return a generator for the rays of the face.

EXAMPLES:

```
sage: pi = Polyhedron(ieqs = [[1,1,0],[1,0,1]])
sage: face = pi.faces(1)[0]
sage: face.ray_generator().next()
A ray in the direction (1, 0)
```

rays()

Return the rays of the face.

OUTPUT:

A tuple of rays.

EXAMPLES:

```
sage: p = Polyhedron(ieqs = [[0,0,0,1],[0,0,1,0],[1,1,0,0]])
sage: face = p.faces(2)[0]
sage: face.rays()
(A ray in the direction (1, 0, 0), A ray in the direction (0, 1, 0))
```

vertex_generator()

Return a generator for the vertices of the face.

EXAMPLES:

```
sage: triangle = Polyhedron(vertices=[[1,0],[0,1],[1,1]])
sage: face = triangle.faces(1)[0]
sage: for v in face.vertex_generator(): print(v)
A vertex at (0, 1)
A vertex at (1, 0)
```

```
sage: type(face.vertex_generator())
<type 'generator'>
```

vertices()

Return all vertices of the face.

OUTPUT:

A tuple of vertices.

EXAMPLES:

```
sage: triangle = Polyhedron(vertices=[[1,0],[0,1],[1,1]])
sage: face = triangle.faces(1)[0]
sage: face.vertices()
(A vertex at (0, 1), A vertex at (1, 0))
```

CHAPTER

FOURTEEN

GENERATE CDD .EXT / .INE FILE FORMAT

sage.geometry.polyhedron.cdd_file_format.**cdd_Hrepresentation**(*cdd_type*, *ieqs*, *eqns*)

Return a string containing the H-representation in cddlib's ine format.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.cdd_file_format import cdd_Hrepresentation
sage: cdd_Hrepresentation('rational', None, [[0,1]])
'H-representation\nlinearity 1 1\nbegin\n 1 2 rational\n 0 1\nend\n'
```

sage.geometry.polyhedron.cdd_file_format.**cdd_Vrepresentation**(*cdd_type*, *vertices*, *rays*, *lines*)

Return a string containing the V-representation in cddlib's ext format.

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
```

CHAPTER

FIFTEEN

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 [Felsner], which can be checked in case of doubt. And please fix this module's documentation afterwards :-)

Definition

A *pseudoline* can not be defined by itself, though it can be thought of as a x-monotone curve in the plane. A *set* of pseudolines, however, represents a set of such curves that pairwise intersect exactly once (and hence mimic the behaviour of straight lines in general position). We also assume that those pseudolines are in general position, that is that no three of them cross at the same point.

The present class is made to deal with a combinatorial encoding of a pseudolines arrangement, that is the ordering in which a pseudoline l_i of an arrangement $l_0, ..., l_{n-1}$ crosses the $n-1$ other lines.

Warning: It is assumed through all the methods that the given lines are numbered according to their y-coordinate on the vertical line $x = -\infty$. For instance, it is not possible that the first transposition be `(0,2)` (or equivalently that the first line l_0 crosses is l_2 and conversely), because one of them would have to cross l_1 first.

15.1 Encodings

Permutations

An arrangement of pseudolines can be described by a sequence of n lists of length $n-1$, where the i list is a permutation of $\{0, ..., n-1\} \backslash i$ representing the ordering in which the i th pseudoline meets the other ones.

```
sage: from sage.geometry.pseudolines import PseudolineArrangement
sage: permutations = [[3, 2, 1], [3, 2, 0], [3, 1, 0], [2, 1, 0]]
sage: p = PseudolineArrangement(permutations)
sage: p
Arrangement of pseudolines of size 4
sage: p.show()
```

Sequence of transpositions

An arrangement of pseudolines can also be described as a sequence of $\binom{n}{2}$ transpositions (permutations of two elements). In this sequence, the transposition $(2, 3)$ appears before $(8, 2)$ iif l_2 crosses l_3 before it crosses l_8. This encoding is easy to obtain by reading the wiring diagram from left to right (see the `show` method).

```
sage: from sage.geometry.pseudolines import PseudolineArrangement
sage: transpositions = [(3, 2), (3, 1), (0, 3), (2, 1), (0, 2), (0, 1)]
sage: p = PseudolineArrangement(transpositions)
sage: p
Arrangement of pseudolines of size 4
sage: p.show()
```

Note that this ordering is not necessarily unique.

Felsner's Matrix

Felser gave an encoding of an arrangement of pseudolines that takes n^2 bits instead of the $n^2log(n)$ 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

$$\begin{matrix} 3 & 2 & 1 \\ 3 & 2 & 0 \\ 3 & 1 & 0 \\ 2 & 1 & 0 \end{matrix} \Rightarrow \begin{matrix} 0 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 1 & 1 \end{matrix}$$

In order to go back from Felsner's matrix to an encoding by a sequence of transpositions, it is sufficient to look for occurrences of $\begin{matrix} 0 \\ 1 \end{matrix}$ 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 [Felsner].

```
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
```

15.2 Example

Let us define in the plane several lines l_i of equation $y = ax + b$ by picking a coefficient a and b for each of them. We make sure that no two of them are parallel by making sure all of the a chosen are different, and we avoid a common crossing of three lines by adding a random noise to b:

```
sage: n = 20
sage: l = zip(Subsets(20*n,n).random_element(), [randint(0,20*n)+random() for i in range(n)])
sage: print l[:5]                                  # not tested
[(96, 278.0130613051349), (74, 332.92512282478714), (13, 155.65820951249867), (209, 34.7539462217553
sage: l.sort()
```

We can now compute for each i the order in which line i meets the other lines:

```
sage: permutations = [[0..i-1]+[i+1..n-1] for i in range(n)]
sage: 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:

```
sage: from sage.geometry.pseudolines import PseudolineArrangement
sage: p = PseudolineArrangement(permutations)
sage: print p
Arrangement of pseudolines of size 20
sage: p.show(figsize=[20,8])
```

15.2.1 References

15.2.2 Author

Nathann Cohen

15.3 Methods

class sage.geometry.pseudolines.**PseudolineArrangement**(*seq*, *encoding='auto'*)

Creates an arrangement of pseudolines.

INPUT:

- `seq` (a sequence describing the line arrangement). It can be :
  - A list of n permutations of size $n-1$.
  - A list of $\binom{n}{2}$ transpositions
  - A Felsner matrix, given as a sequence of n binary vectors of length $n-1$.
- `encoding` (information on how the data should be interpreted), and can assume any value among 'transpositions', 'permutations', 'Felsner' or 'auto'. In the latter case, the type will be guessed (default behaviour).

Note:

- The pseudolines are assumed to be integers $0..(n-1)$.
- For more information on the different encodings, see the `pseudolines module`'s documentation.

TESTS:

From permutations:

```
sage: from sage.geometry.pseudolines import PseudolineArrangement
sage: permutations = [[3, 2, 1], [3, 2, 0], [3, 1, 0], [2, 1, 0]]
sage: PseudolineArrangement(permutations)
Arrangement of pseudolines of size 4
```

From transpositions

```
sage: from sage.geometry.pseudolines import PseudolineArrangement
sage: transpositions = [(3, 2), (3, 1), (0, 3), (2, 1), (0, 2), (0, 1)]
sage: PseudolineArrangement(transpositions)
Arrangement of pseudolines of size 4
```

From a Felsner matrix:

```
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: matrix = p.felsner_matrix()
sage: PseudolineArrangement(matrix) == p
True
```

TESTS:

Wrong input:

```
sage: PseudolineArrangement([[5, 2, 1], [3, 2, 0], [3, 1, 0], [2, 1, 0]])
Traceback (most recent call last):
...
ValueError: Are the lines really numbered from 0 to n-1?
sage: PseudolineArrangement([(3, 2), (3, 1), (0, 3), (2, 1), (0, 2)])
Traceback (most recent call last):
...
ValueError: A line is numbered 3 but the number of transpositions ...
```

felsner_matrix()

Returns a Felsner matrix describing the arrangement.

See the pseudolines module's documentation for more information on this encoding.

EXAMPLE:

```
sage: from sage.geometry.pseudolines import PseudolineArrangement
sage: permutations = [[3, 2, 1], [3, 2, 0], [3, 1, 0], [2, 1, 0]]
sage: p = PseudolineArrangement(permutations)
sage: p.felsner_matrix()
[[0, 0, 0], [0, 0, 1], [0, 1, 1], [1, 1, 1]]
```

permutations()

Returns the arrangements as n permutations of size $n-1$.

See the pseudolines module's documentation for more information on this encoding.

EXAMPLE:

```
sage: from sage.geometry.pseudolines import PseudolineArrangement
sage: permutations = [[3, 2, 1], [3, 2, 0], [3, 1, 0], [2, 1, 0]]
sage: p = PseudolineArrangement(permutations)
sage: p.permutations()
[[3, 2, 1], [3, 2, 0], [3, 1, 0], [2, 1, 0]]
```

show(***args*)

Displays the pseudoline arrangement as a wiring diagram.

INPUT:

- `**args` – any arguments to be forwarded to the `show` method. In particular, to tune the dimensions, use the `figsize` argument (example below).

EXAMPLE:

```
sage: from sage.geometry.pseudolines import PseudolineArrangement
sage: permutations = [[3, 2, 1], [3, 2, 0], [3, 1, 0], [2, 1, 0]]
sage: p = PseudolineArrangement(permutations)
sage: p.show(figsize=[7,5])
```

TESTS:

```
sage: from sage.geometry.pseudolines import PseudolineArrangement
sage: permutations = [[3, 2, 1], [3, 2, 0], [3, 0, 1], [2, 0, 1]]
sage: p = PseudolineArrangement(permutations)
sage: p.show()
Traceback (most recent call last):
...
ValueError: There has been a problem while plotting the figure...
```

transpositions()

Returns the arrangement as $\binom{n}{2}$ transpositions.

See the pseudolines module's documentation for more information on this encoding.

EXAMPLE:

```
sage: from sage.geometry.pseudolines import PseudolineArrangement
sage: permutations = [[3, 2, 1], [3, 2, 0], [3, 1, 0], [2, 1, 0]]
sage: p1 = PseudolineArrangement(permutations)
sage: transpositions = [(3, 2), (3, 1), (0, 3), (2, 1), (0, 2), (0, 1)]
sage: p2 = PseudolineArrangement(transpositions)
sage: p1 == p2
True
sage: p1.transpositions()
[(3, 2), (3, 1), (0, 3), (2, 1), (0, 2), (0, 1)]
sage: p2.transpositions()
[(3, 2), (3, 1), (0, 3), (2, 1), (0, 2), (0, 1)]
```

CHAPTER

SIXTEEN

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 command:

```
sage: install_package('TOPCOM')      # not tested
```

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 QQ^2 consisting of 5 points. The
triangulations of this point configuration are assumed to
be connected, not necessarily fine, not necessarily regular.
sage: t = p.triangulate()  # a single triangulation
sage: t
(<1,3,4>, <2,3,4>)
sage: len(t)
2
sage: t[0]
(1, 3, 4)
sage: t[1]
```

```
(2, 3, 4)
sage: list(t)
[(1, 3, 4), (2, 3, 4)]
sage: t.plot(axes=False)
sage: list( p.triangulations() )
[(<1,3,4>, <2,3,4>),
 (<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>),
 (<1,2,3>, <1,2,4>),
 (<0,1,2>, <0,1,4>, <0,2,4>, <1,2,3>)]
sage: p_fine = p.restrict_to_fine_triangulations()
sage: p_fine
A point configuration in QQ^2 consisting of 5 points. The
triangulations of this point configuration are assumed to
be connected, fine, not necessarily regular.
sage: list( p_fine.triangulations() )
[(<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>),
 (<0,1,2>, <0,1,4>, <0,2,4>, <1,2,3>)]
```

A 3-dimensional point configuration:

```
sage: p = [[0,-1,-1],[0,0,1],[0,1,0], [1,-1,-1],[1,0,1],[1,1,0]]
sage: points = PointConfiguration(p)
sage: triang = points.triangulate()
sage: triang.plot(axes=False)
```

The standard example of a non-regular triangulation:

```
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 - TO
sage: nonregular = p.restrict_to_regular_triangulations(False).triangulations_list()  # optional - TO
sage: len(regular)     # optional - TOPCOM
16
sage: len(nonregular)  # optional - TOPCOM
2
sage: nonregular[0].plot(aspect_ratio=1, axes=False)    # optional - TOPCOM
```

Note that the points need not be in general position. That is, the points may lie in a hyperplane and the linear dependencies will be removed before passing the data to TOPCOM which cannot handle it:

```
sage: points = [[0,0,0,1],[0,3,0,1],[3,0,0,1],[0,0,1,1],[0,3,1,1],[3,0,1,1],[1,1,2,1]]
sage: points = [ p+[1,2,3] for p in points ]
sage: pc = PointConfiguration(points)
sage: pc.ambient_dim()
7
sage: pc.dim()
3
sage: pc.triangulate()
(<0,1,2,6>, <0,1,3,6>, <0,2,3,6>, <1,2,4,6>, <1,3,4,6>, <2,3,5,6>, <2,4,5,6>)
sage: _ in pc.triangulations()
True
sage: len( pc.triangulations_list() )
26
```

REFERENCES:

AUTHORS:

- Volker Braun: initial version, 2010
- Josh Whitney: added functionality for computing volumes and secondary polytopes of PointConfigurations
- Marshall Hampton: improved documentation and doctest coverage
- Volker Braun: rewrite using Parent/Element and catgories. Added a Point class. More doctests. Less zombies.
- Volker Braun: Cythonized parts of it, added a C++ implementation of the bistellar flip algorithm to enumerate all connected triangulations.
- Volker Braun 2011: switched the triangulate() method to the placing triangulation (faster).

class sage.geometry.triangulation.point_configuration.**PointConfiguration**(*points*, *connected*, *fine*, *regular*, *star*, *defined_affine*)

Bases: 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:

```
sage: p = PointConfiguration([[0,0],[0,1],[1,0],[1,1],[-1,-1]])
sage: p
A point configuration in QQ^2 consisting of 5 points. The
triangulations of this point configuration are assumed to
be connected, not necessarily fine, not necessarily regular.
sage: p.triangulate()  # a single triangulation
(<1,3,4>, <2,3,4>)
```

Element
alias of Triangulation

Gale_transform(*points=None*)
Return the Gale transform of self.

INPUT:

- points – a tuple of points or point indices or None (default). A subset of points for which to compute the Gale transform. By default, all points are used.

OUTPUT:

A matrix over base_ring().

EXAMPLES:

```
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().

TESTS:

```
sage: p = PointConfiguration([[0, 1], [0, 0], [1, 0], [1,1]])
sage: p.an_element()
(<0,1,3>, <1,2,3>)
```

bistellar_flips()
Return the bistellar flips.

OUTPUT:

The bistellar flips as a tuple. Each flip is a pair (T_+, T_-) where T_+ and T_- are partial triangulations of the point configuration.

EXAMPLES:

```
sage: pc = PointConfiguration([(0,0),(1,0),(0,1),(1,1)])
sage: pc.bistellar_flips()
(((<0,1,3>, <0,2,3>), (<0,1,2>, <1,2,3>)),)
sage: Tpos, Tneg = pc.bistellar_flips()[0]
sage: Tpos.plot(axes=False)
sage: Tneg.plot(axes=False)
```

The 3d analog:

```
sage: pc = PointConfiguration([(0,0,0),(0,2,0),(0,0,2),(-1,0,0),(1,1,1)])
sage: pc.bistellar_flips()
(((<0,1,2,3>, <0,1,2,4>), (<0,1,3,4>, <0,2,3,4>, <1,2,3,4>)),)
```

A 2d flip on the base of the pyramid over a square:

```
sage: pc = PointConfiguration([(0,0,0),(0,2,0),(0,0,2),(0,2,2),(1,1,1)])
sage: pc.bistellar_flips()
(((<0,1,3>, <0,2,3>), (<0,1,2>, <1,2,3>)),)
sage: Tpos, Tneg = pc.bistellar_flips()[0]
sage: Tpos.plot(axes=False)
```

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:

```
sage: p = PointConfiguration([(0,0),(+1,0),(-1,0),(0,+1),(0,-1)])
sage: p.circuits()
(((0,), (1, 2), (3, 4)), ((0,), (3, 4), (1, 2)), ((1, 2), (0,), (3, 4)))
```

TESTS:

```
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: len( p.circuits() )     # long time
218
```

circuits_support()

A generator for the supports of the circuits of the point configuration.

See `circuits()` for details.

OUTPUT:

A generator for the supports $C_- \cup C_+$ (returned as a Python tuple) for all circuits of the point configuration.

EXAMPLES:

```
sage: p = PointConfiguration([(0,0),(+1,0),(-1,0),(0,+1),(0,-1)])
sage: list( p.circuits_support() )
[(0, 3, 4), (0, 1, 2), (1, 2, 3, 4)]
```

contained_simplex(*large=True*, *initial_point=None*)

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.

OUTPUT:

A tuple of points that span a simplex of dimension `dim()`. If `large==True`, the simplex is constructed by sucessively picking the farthest point. This will ensure that the simplex is not unneccessarily small, but will in general not return a maximal simplex.

EXAMPLES:

```
sage: pc = PointConfiguration([(0,0),(1,0),(2,1),(1,1),(0,1)])
sage: pc.contained_simplex()
(P(0, 1), P(2, 1), P(1, 0))
sage: pc.contained_simplex(large=False)
(P(0, 1), P(1, 1), P(1, 0))
sage: pc.contained_simplex(initial_point=pc.point(0))
(P(0, 0), P(1, 1), P(1, 0))

sage: pc = PointConfiguration([[0,0],[0,1],[1,0],[1,1],[-1,-1]])
sage: pc.contained_simplex()
(P(-1, -1), P(1, 1), P(0, 1))
```

TESTS:

```
sage: pc = PointConfiguration([[0,0],[0,1],[1,0]])
sage: pc.contained_simplex()
(P(1, 0), P(0, 1), P(0, 0))
sage: pc = PointConfiguration([[0,0],[0,1]])
sage: pc.contained_simplex()
(P(0, 1), P(0, 0))
sage: pc = PointConfiguration([[0,0]])
sage: pc.contained_simplex()
(P(0, 0),)
sage: pc = PointConfiguration([])
sage: pc.contained_simplex()
()
```

convex_hull()

Return the convex hull of the point configuration.

EXAMPLES:

```
sage: p = PointConfiguration([[0,0],[0,1],[1,0],[1,1],[-1,-1]])
sage: p.convex_hull()
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices
```

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: pc = PointConfiguration([(0,0),(1,0),(2,1),(1,2),(0,1)])
sage: [ pc.distance(pc.point(0), p) for p in pc.points() ]
[0, 1, 5, 5, 1]

sage: pc = PointConfiguration([(0,0,1),(1,0,1),(2,1,1),(1,2,1),(0,1,1)], projective=True)
sage: [ pc.distance(pc.point(0), p) for p in pc.points() ]
[0, 1/2, 5/6, 5/6, 1/2]
```

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: pc = PointConfiguration([(0,0),(1,0),(2,1),(1,2),(0,1)])
sage: [ pc.distance_FS(pc.point(0), p) for p in pc.points() ]
[0, 1/2, 5/6, 5/6, 1/2]
```

distance_affine (*x*, *y*)

Returns the distance between two points.

The distance function used in this method is $d_{aff}(x,y)^2$, the square of the usual affine distance function

$$d_{aff}(x,y) = |x-y|$$

INPUT:

- x, y – two points of the point configuration.

OUTPUT:

The metric distance-square $d_{aff}(x,y)^2$. Note that this distance lies in the same field as the entries of x, y. That is, the distance of rational points will be rational and so on.

EXAMPLES:

```
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:

```
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:

```
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):

```
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:

```
sage: triangle = PointConfiguration([[-1,0], [0,0], [1,-1], [1,0], [1,1]]);
sage: triangle.face_interior()
((1,), (3,), (0, 2, 4))
sage: triangle.face_interior(dim=0)     # the vertices of the convex hull
(0, 2, 4)
sage: triangle.face_interior(codim=1)   # interior of facets
(3,)
```

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:

```
sage: pc = PointConfiguration([(0,0),(1,0),(1,1),(0,1)])
sage: pc.farthest_point([ pc.point(0) ])
P(1, 1)
```

lexicographic_triangulation()

Return the lexicographic triangulation.

The algorithm was taken from [PUNTOS].

EXAMPLES:

```
sage: p = PointConfiguration([(0,0),(+1,0),(-1,0),(0,+1),(0,-1)])
sage: p.lexicographic_triangulation()
(<1,3,4>, <2,3,4>)
```

TESTS:

```
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.lexicographic_triangulation()
(<1,3,4,7,10,13>, <1,3,4,8,10,13>, <1,3,6,7,10,13>, <1,3,6,8,10,13>,
 <1,4,6,7,10,13>, <1,4,6,8,10,13>, <2,3,4,6,7,12>, <2,3,4,7,12,13>,
 <2,3,6,7,12,13>, <2,4,6,7,12,13>, <3,4,5,6,9,12>, <3,4,5,8,9,12>,
 <3,4,6,7,11,12>, <3,4,6,9,11,12>, <3,4,7,10,11,13>, <3,4,7,11,12,13>,
 <3,4,8,9,10,12>, <3,4,8,10,12,13>, <3,4,9,10,11,12>, <3,4,10,11,12,13>,
 <3,5,6,8,9,12>, <3,6,7,10,11,13>, <3,6,7,11,12,13>, <3,6,8,9,10,12>,
 <3,6,8,10,12,13>, <3,6,9,10,11,12>, <3,6,10,11,12,13>, <4,5,6,8,9,12>,
 <4,6,7,10,11,13>, <4,6,7,11,12,13>, <4,6,8,9,10,12>, <4,6,8,10,12,13>,
 <4,6,9,10,11,12>, <4,6,10,11,12,13>)
sage: len(_)
34
```

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.

OUTPUT:

A `Triangulation`.

EXAMPLES:

```
sage: pc = PointConfiguration([(0,0),(1,0),(2,1),(1,2),(0,1)])
sage: pc.placing_triangulation()
(<0,1,2>, <0,2,4>, <2,3,4>)

sage: 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
```

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_- = `negative`.

EXAMPLE:

```
sage: p = PointConfiguration([(1,0,0),(0,1,0),(0,0,1),(-2,0,-1),(-2,-1,0),(-3,-1,-1),(1,1,1)
sage: p.positive_circuits(8)
((0, 7), (0, 1, 4), (0, 2, 3), (0, 5, 6), (0, 1, 2, 5), (0, 3, 4, 6))
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.

OUTPUT:

A `Triangulation`.

EXAMPLES:

```
sage: pc = PointConfiguration([(0,0),(1,0),(2,1),(1,2),(0,1)])
sage: pc.placing_triangulation()
(<0,1,2>, <0,2,4>, <2,3,4>)

sage: 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
```

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 QQ^2 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_all = p.restrict_to_connected_triangulations(connected=False)  # optional - TOPCOM
sage: len(p_all.triangulations_list())                                 # optional - TOPCOM
4
sage: p == p_all.restrict_to_connected_triangulations(connected=True)  # optional - TOPCOM
True
```

restrict_to_fine_triangulations (*fine=True*)

Restrict to fine triangulations.

INPUT:

- `fine` – boolean. Whether to restrict to fine triangulations.

OUTPUT:

A new `PointConfiguration` with the same points, but whose triangulations will all be fine. See `PointConfiguration` for details.

EXAMPLES:

```
sage: p = PointConfiguration([[0,0],[0,1],[1,0],[1,1],[-1,-1]])
sage: p
A point configuration in QQ^2 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: p = PointConfiguration([[0,0],[0,1],[1,0],[1,1],[-1,-1]])
sage: p
A point configuration in QQ^2 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_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
```

restrict_to_star_triangulations(*star*)

Restrict to star triangulations with the given point as the center.

INPUT:

- `origin` – `None` or an integer or the coordinates of a point. An integer denotes the index of the central point. If `None` is passed, any restriction on the starshape will be removed.

OUTPUT:

A new `PointConfiguration` with the same points, but whose triangulations will all be star. See `PointConfiguration` for details.

EXAMPLES:

```
sage: p = PointConfiguration([[0,0],[0,1],[1,0],[1,1],[-1,-1]])
sage: len(list( p.triangulations() ))
4
sage: p_star =  p.restrict_to_star_triangulations(0)
sage: p_star is p.restrict_to_star_triangulations((0,0))
True
sage: p_star.triangulations_list()
[(<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>)]
sage: p_newstar = p_star.restrict_to_star_triangulations(1)  # pick different origin
sage: p_newstar.triangulations_list()
[(<1,2,3>, <1,2,4>)]
```

```
sage: p == p_star.restrict_to_star_triangulations(star=None)
True
```

restricted_automorphism_group()

Return the restricted automorphism group.

First, let the linear automorphism group be the subgroup of the Euclidean group $E(d) = GL(d, \mathbf{R}) \ltimes \mathbf{R}^d$ preserving the d-dimensional point configuration. 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 points. See [BSS] for more details and a description of the algorithm.

OUTPUT:

A `PermutationGroup` that is isomorphic to the restricted automorphism group is returned.

Note that in Sage, permutation groups always act on positive integers while lists etc. are indexed by non-negative integers. The indexing of the permutation group is chosen to be shifted by `+1`. That is, the transposition `(i,j)` in the permutation group corresponds to exchange of `self[i-1]` and `self[j-1]`.

EXAMPLES:

```
sage: pyramid = PointConfiguration([[1,0,0],[0,1,1],[0,1,-1],[0,-1,-1],[0,-1,1]])
sage: pyramid.restricted_automorphism_group()
Permutation Group with generators [(3,5), (2,3)(4,5), (2,4)]
sage: DihedralGroup(4).is_isomorphic(_)
True
```

The square with an off-center point in the middle. Note thath the middle point breaks the restricted automorphism group D_4 of the convex hull:

```
sage: square = PointConfiguration([(3/4,3/4),(1,1),(1,-1),(-1,-1),(-1,1)])
sage: square.restricted_automorphism_group()
Permutation Group with generators [(3,5)]
sage: DihedralGroup(1).is_isomorphic(_)
True
```

secondary_polytope()

Calculate the secondary polytope of the point configuration.

For a definition of the secondary polytope, see [GKZ] page 220 Definition 1.6.

Note that if you restricted the admissible triangulations of the point configuration then the output will be the corresponding face of the whole secondary polytope.

OUTPUT:

The secondary polytope of the point configuration as an instance of `Polyhedron_base`.

EXAMPLES:

```
sage: p = PointConfiguration([[0,0],[1,0],[2,1],[1,2],[0,1]])
sage: poly = p.secondary_polytope()
sage: poly.vertices_matrix()
[1 1 3 3 5]
[3 5 1 4 1]
[4 2 5 2 4]
[2 4 2 5 4]
[5 3 4 1 1]
sage: poly.Vrepresentation()
(A vertex at (1, 3, 4, 2, 5),
 A vertex at (1, 5, 2, 4, 3),
```

```
 A vertex at (3, 1, 5, 2, 4),
 A vertex at (3, 4, 2, 5, 1),
 A vertex at (5, 1, 4, 4, 1))
sage: poly.Hrepresentation()
(An equation (0, 0, 1, 2, 1) x - 13 == 0,
 An equation (1, 0, 0, 2, 2) x - 15 == 0,
 An equation (0, 1, 0, -3, -2) x + 13 == 0,
 An inequality (0, 0, 0, -1, -1) x + 7 >= 0,
 An inequality (0, 0, 0, 1, 0) x - 2 >= 0,
 An inequality (0, 0, 0, -2, -1) x + 11 >= 0,
 An inequality (0, 0, 0, 0, 1) x - 1 >= 0,
 An inequality (0, 0, 0, 3, 2) x - 14 >= 0)
```

classmethod set_engine(*engine='auto'*)

Set the engine used to compute triangulations.

INPUT:

- `engine` – either 'auto' (default), 'internal', or 'TOPCOM'. The latter two instruct this package to always use its own triangulation algorithms or TOPCOM's algorithms, respectively. By default ('auto'), TOPCOM is used if it is available and internal routines otherwise.

EXAMPLES:

```
sage: p = PointConfiguration([[0,0],[0,1],[1,0],[1,1],[-1,-1]])
sage: p.set_engine('internal')    # to make doctests independent of TOPCOM
sage: p.triangulate()
(<1,3,4>, <2,3,4>)
sage: p.set_engine('TOPCOM')    # 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:

```
sage: pc = PointConfiguration([(1,0),(-1,0),(0,1),(0,2)], star=(0,1)); pc
A point configuration in QQ^2 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 QQ^2 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:

```
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
```

triangulations(*verbose=False*)

Returns all triangulations.

- verbose – boolean (default: False). Whether to print out the TOPCOM interaction, if any.

OUTPUT:

A generator for the triangulations satisfying all the restrictions imposed. Each triangulation is returned as a Triangulation object.

EXAMPLES:

```
sage: p = PointConfiguration([[0,0],[0,1],[1,0],[1,1],[-1,-1]])
sage: iter = p.triangulations()
sage: iter.next()
(<1,3,4>, <2,3,4>)
sage: iter.next()
(<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>)
sage: iter.next()
(<1,2,3>, <1,2,4>)
sage: iter.next()
(<0,1,2>, <0,1,4>, <0,2,4>, <1,2,3>)
sage: p.triangulations_list()
[(<1,3,4>, <2,3,4>),
 (<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>),
 (<1,2,3>, <1,2,4>),
 (<0,1,2>, <0,1,4>, <0,2,4>, <1,2,3>)]
sage: p_fine = p.restrict_to_fine_triangulations()
sage: p_fine.triangulations_list()
[(<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>),
```

```
    (<0,1,2>, <0,1,4>, <0,2,4>, <1,2,3>)]

Note that we explicitly asked the internal algorithm to
compute the triangulations. Using TOPCOM, we obtain the same
triangulations but in a different order::

   sage: p.set_engine('TOPCOM')                          # optional - TOPCOM
   sage: iter = p.triangulations()                       # optional - TOPCOM
   sage: iter.next()                                     # optional - TOPCOM
   (<0,1,2>, <0,1,4>, <0,2,4>, <1,2,3>)
   sage: iter.next()                                     # optional - TOPCOM
   (<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>)
   sage: iter.next()                                     # optional - TOPCOM
   (<1,2,3>, <1,2,4>)
   sage: iter.next()                                     # optional - TOPCOM
   (<1,3,4>, <2,3,4>)
   sage: p.triangulations_list()                         # optional - TOPCOM
   [(<0,1,2>, <0,1,4>, <0,2,4>, <1,2,3>),
    (<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>),
    (<1,2,3>, <1,2,4>),
    (<1,3,4>, <2,3,4>)]
   sage: p_fine = p.restrict_to_fine_triangulations() # optional - TOPCOM
   sage: p_fine.set_engine('TOPCOM')                     # optional - TOPCOM
   sage: p_fine.triangulations_list()                    # optional - TOPCOM
   [(<0,1,2>, <0,1,4>, <0,2,4>, <1,2,3>),
    (<0,1,3>, <0,1,4>, <0,2,3>, <0,2,4>)]
   sage: p.set_engine('internal')                        # optional - TOPCOM
```

triangulations_list(*verbose=False*)

Return all triangulations.

INPUT:

- `verbose` – boolean. Whether to print out the TOPCOM interaction, if any.

OUTPUT:

A list of triangulations (see `Triangulation`) satisfying all restrictions imposed previously.

EXAMPLES:

```
sage: p = PointConfiguration([[0,0],[0,1],[1,0],[1,1]])
sage: p.triangulations_list()
[(<0,1,2>, <1,2,3>), (<0,1,3>, <0,2,3>)]
sage: 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:

```
sage: p.volume()
2
```

Note: We return n!*(metric volume of the simplex) to ensure that the volume is an integer. Essentially, this normalizes things so that the volume of the standard n-simplex is 1. See [GKZ] page 182.

CHAPTER

SEVENTEEN

BASE CLASSES FOR TRIANGULATIONS

Base classes for triangulations

We provide (fast) cython implementations here.

AUTHORS:

- Volker Braun (2010-09-14): initial version.

class sage.geometry.triangulation.base.**ConnectedTriangulationsIterator**
Bases: 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:

```
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: ci.next()
(9, 10)
sage: ci.next()
(2, 3, 4, 5)
```

```
sage: ci.next()
(7, 8)
sage: ci.next()
(1, 3, 5, 7)
sage: ci.next()
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)
[]
```

next()

x.next() -> the next value, or raise StopIteration

class sage.geometry.triangulation.base.**Point**

Bases: sage.structure.sage_object.SageObject

A point of a point configuration.

Note that the coordinates of the points of a point configuration are somewhat arbitrary. What counts are the abstract linear relations between the points, for example encoded by the circuits().

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)
```

affine()
Return the affine coordinates of the point in the ambient space.

OUTPUT:

A tuple containing the coordinates.

EXAMPLES:

```
sage: pc = PointConfiguration([[10, 0, 1], [10, 0, 0], [10, 2, 3]])
sage: p = pc.point(2); p
P(10, 2, 3)
sage: p.affine()
(10, 2, 3)
sage: p.projective()
(10, 2, 3, 1)
sage: p.reduced_affine()
(2, 2)
sage: p.reduced_projective()
(2, 2, 1)
sage: p.reduced_affine_vector()
(2, 2)
```

index()
Return the index of the point in the point configuration.

EXAMPLES:

```
sage: pc = PointConfiguration([[0, 1], [0, 0], [1, 0]])
sage: p = pc.point(2); p
P(1, 0)
sage: p.index()
2
```

point_configuration()
Return the point configuration to which the point belongs.

OUTPUT:

A PointConfiguration.

EXAMPLES:

```
sage: pc = PointConfiguration([ (0,0), (1,0), (0,1) ])
sage: p = pc.point(0)
sage: p is pc.point(0)
True
sage: p.point_configuration() is pc
True
```

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)
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.

Warning: You should not instantiate this base class, but only its derived class PointConfiguration.

ambient_dim()

Return the dimension of the ambient space of the point configuration.

See also dimension()

EXAMPLES:

```
sage: p = PointConfiguration([[0,0,0]])
sage: p.ambient_dim()
3
sage: p.dim()
0
```

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:

```
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:

```
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:

```
sage: p = PointConfiguration([[0,0],[0,1],[1,0],[1,1],[-1,-1]])
sage: p
A point configuration in QQ^2 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:

```
sage: pconfig = PointConfiguration([[0,0],[0,1],[1,0],[1,1],[-1,-1]])
sage: list(pconfig)
[P(0, 0), P(0, 1), P(1, 0), P(1, 1), P(-1, -1)]
sage: [ p for p in pconfig.points() ]
[P(0, 0), P(0, 1), P(1, 0), P(1, 1), P(-1, -1)]
sage: pconfig.point(0)
P(0, 0)
sage: pconfig[0]
P(0, 0)
sage: pconfig.point(1)
P(0, 1)
sage: pconfig.point( pconfig.n_points()-1 )
P(-1, -1)
```

points()

Return a list of the points.

OUTPUT:

Returns a list of the points. See also the `__iter__()` method, which returns the corresponding generator.

EXAMPLES:

```
sage: pconfig = PointConfiguration([[0,0],[0,1],[1,0],[1,1],[-1,-1]])
sage: list(pconfig)
[P(0, 0), P(0, 1), P(1, 0), P(1, 1), P(-1, -1)]
sage: [ p for p in pconfig.points() ]
[P(0, 0), P(0, 1), P(1, 0), P(1, 1), P(-1, -1)]
sage: pconfig.point(0)
P(0, 0)
sage: pconfig.point(1)
P(0, 1)
sage: pconfig.point( pconfig.n_points()-1 )
P(-1, -1)
```

reduced_affine_vector_space()

Return the vector space that contains the affine points.

OUTPUT:

A vector space over the fraction field of `base_ring()`.

EXAMPLES:

```
sage: p = PointConfiguration([[0,0,0], [1,2,3]])
sage: p.base_ring()
Integer Ring
sage: p.reduced_affine_vector_space()
Vector space of dimension 1 over Rational Field
sage: p.reduced_projective_vector_space()
Vector space of dimension 2 over Rational Field
```

reduced_projective_vector_space()

Return the vector space that is spanned by the homogeneous coordinates.

OUTPUT:

A vector space over the fraction field of `base_ring()`.

EXAMPLES:

```
sage: p = PointConfiguration([[0,0,0], [1,2,3]])
sage: p.base_ring()
Integer Ring
sage: p.reduced_affine_vector_space()
Vector space of dimension 1 over Rational Field
sage: p.reduced_projective_vector_space()
Vector space of dimension 2 over Rational Field
```

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:

```
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)
```

CHAPTER

EIGHTEEN

A TRIANGULATION

A triangulation

In Sage, the PointConfiguration and Triangulation satisfy a parent/element relationship. In particular, each triangulation refers back to its point configuration. If you want to triangulate a point configuration, you should construct a point configuration first and then use one of its methods to triangulate it according to your requirements. You should never have to construct a Triangulation object directly.

EXAMPLES:

First, we select the internal implementation for enumerating triangulations:

```
sage: PointConfiguration.set_engine('internal')   # to make doctests independent of TOPCOM
```

Here is a simple example of how to triangulate a point configuration:

```
sage: p = [[0,-1,-1],[0,0,1],[0,1,0], [1,-1,-1],[1,0,1],[1,1,0]]
sage: points = PointConfiguration(p)
sage: triang = points.triangulate();  triang
(<0,1,2,5>, <0,1,3,5>, <1,3,4,5>)
sage: triang.plot(axes=False)
```

See sage.geometry.triangulation.point_configuration for more details.

class sage.geometry.triangulation.element.**Triangulation**(*triangulation*, *parent*, *check=True*)

Bases: sage.structure.element.Element

A triangulation of a PointConfiguration.

> **Warning:** You should never create Triangulation objects manually. See triangulate() and triangulations() to triangulate point configurations.

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:

```
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.n_cube(3).triangulate(engine='internal')
sage: triangulation
(<0,1,2,7>, <0,1,4,7>, <0,2,4,7>, <1,2,3,7>, <1,4,5,7>, <2,4,6,7>)
sage: triangulation.boundary()
frozenset([(1, 3, 7), (4, 5, 7), (1, 2, 3), (0, 1, 2), (2, 4, 6), (2, 6, 7),
           (2, 3, 7), (1, 5, 7), (0, 1, 4), (1, 4, 5), (4, 6, 7), (0, 2, 4)])
sage: triangulation.interior_facets()
frozenset([(1, 4, 7), (1, 2, 7), (2, 4, 7), (0, 1, 7), (0, 4, 7), (0, 2, 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,  1,  0,  0,  1],
...       [ 0,  1,  1,  0,  0,  1,  0, -2,  1,  0,  0, -1,  1,  1],
...       [ 0,  0,  0,  0,  1,  0, -1,  0,  0,  0,  0,  0,  0,  0]
...    ]).columns())
sage: triangulation = pc.lexicographic_triangulation()
sage: triangulation.enumerate_simplices()
(1678, 1688, 1769, 1779, 1895, 1905, 2112, 2143, 2234, 2360, 2555, 2580,
 2610, 2626, 2650, 2652, 2654, 2661, 2663, 2667, 2685, 2755, 2757, 2759,
 2766, 2768, 2772, 2811, 2881, 2883, 2885, 2892, 2894, 2898)
```

You can recreate the triangulation from this list by passing it to the constructor:

```
sage: from sage.geometry.triangulation.point_configuration import Triangulation
sage: Triangulation([1678, 1688, 1769, 1779, 1895, 1905, 2112, 2143,
...     2234, 2360, 2555, 2580, 2610, 2626, 2650, 2652, 2654, 2661, 2663,
...     2667, 2685, 2755, 2757, 2759, 2766, 2768, 2772, 2811, 2881, 2883,
...     2885, 2892, 2894, 2898], pc)
(<1,3,4,7,10,13>, <1,3,4,8,10,13>, <1,3,6,7,10,13>, <1,3,6,8,10,13>,
 <1,4,6,7,10,13>, <1,4,6,8,10,13>, <2,3,4,6,7,12>, <2,3,4,7,12,13>,
 <2,3,6,7,12,13>, <2,4,6,7,12,13>, <3,4,5,6,9,12>, <3,4,5,8,9,12>,
```

```
<3,4,6,7,11,12>, <3,4,6,9,11,12>, <3,4,7,10,11,13>, <3,4,7,11,12,13>,
<3,4,8,9,10,12>, <3,4,8,10,12,13>, <3,4,9,10,11,12>, <3,4,10,11,12,13>,
<3,5,6,8,9,12>, <3,6,7,10,11,13>, <3,6,7,11,12,13>, <3,6,8,9,10,12>,
<3,6,8,10,12,13>, <3,6,9,10,11,12>, <3,6,10,11,12,13>, <4,5,6,8,9,12>,
<4,6,7,10,11,13>, <4,6,7,11,12,13>, <4,6,8,9,10,12>, <4,6,8,10,12,13>,
<4,6,9,10,11,12>, <4,6,10,11,12,13>)
```

fan(*origin=None*)

Construct the fan of cones over the simplices of the triangulation.

INPUT:

- `origin` – `None` (default) or coordinates of a point. The common apex of all cones of the fan. If `None`, the triangulation must be a star triangulation and the distinguished central point is used as the origin.

OUTPUT:

A `RationalPolyhedralFan`. The coordinates of the points are shifted so that the apex of the fan is the origin of the coordinate system.

Note: If the set of cones over the simplices is not a fan, a suitable exception is raised.

EXAMPLES:

```
sage: pc = PointConfiguration([(0,0), (1,0), (0,1), (-1,-1)], star=0, fine=True)
sage: triangulation = pc.triangulate()
sage: fan = triangulation.fan(); fan
Rational polyhedral fan in 2-d lattice N
sage: fan.is_equivalent( toric_varieties.P2().fan() )
True
```

Toric diagrams (the $\mathbf{Z}_5$ hyperconifold):

```
sage: vertices=[(0, 1, 0), (0, 3, 1), (0, 2, 3), (0, 0, 2)]
sage: interior=[(0, 1, 1), (0, 1, 2), (0, 2, 1), (0, 2, 2)]
sage: points = vertices+interior
sage: pc = PointConfiguration(points, fine=True)
sage: triangulation = pc.triangulate()
sage: fan = triangulation.fan( (-1,0,0) )
sage: fan
Rational polyhedral fan in 3-d lattice N
sage: fan.rays()
N(1, 1, 0),
N(1, 3, 1),
N(1, 2, 3),
N(1, 0, 2),
N(1, 1, 1),
N(1, 1, 2),
N(1, 2, 1),
N(1, 2, 2)
in 3-d lattice N
```

gkz_phi()

Calculate the GKZ phi vector of the triangulation.

The phi vector is a vector of length equals to the number of points in the point configuration. For a fixed

triangulation T, the entry corresponding to the i-th point p_i is

$$\phi_T(p_i) = \sum_{t \in T, t \ni p_i} Vol(t)$$

that is, the total volume of all simplices containing p_i. See also [GKZ] page 220 equation 1.4.

OUTPUT:

The phi vector of self.

EXAMPLES:

```
sage: p = PointConfiguration([[0,0],[1,0],[2,1],[1,2],[0,1]])
sage: p.triangulate().gkz_phi()
(3, 1, 5, 2, 4)
sage: p.lexicographic_triangulation().gkz_phi()
(1, 3, 4, 2, 5)
```

interior_facets()

Return the interior facets of the triangulation.

OUTPUT:

The inward-facing boundary simplices (of dimension $d-1$) of the d-dimensional triangulation as a set. Each boundary is returned by a tuple of point indices.

EXAMPLES:

```
sage: triangulation = polytopes.n_cube(3).triangulate(engine='internal')
sage: triangulation
(<0,1,2,7>, <0,1,4,7>, <0,2,4,7>, <1,2,3,7>, <1,4,5,7>, <2,4,6,7>)
sage: triangulation.boundary()
frozenset([(1, 3, 7), (4, 5, 7), (1, 2, 3), (0, 1, 2), (2, 4, 6), (2, 6, 7),
           (2, 3, 7), (1, 5, 7), (0, 1, 4), (1, 4, 5), (4, 6, 7), (0, 2, 4)])
sage: triangulation.interior_facets()
frozenset([(1, 4, 7), (1, 2, 7), (2, 4, 7), (0, 1, 7), (0, 4, 7), (0, 2, 7)])
```

normal_cone()

Return the (closure of the) normal cone of the triangulation.

Recall that a regular triangulation is one that equals the "crease lines" of a convex piecewise-linear function. This support function is not unique, for example, you can scale it by a positive constant. The set of all piecewise-linear functions with fixed creases forms an open cone. This cone can be interpreted as the cone of normal vectors at a point of the secondary polytope, which is why we call it normal cone. See [GKZ] Section 7.1 for details.

OUTPUT:

The closure of the normal cone. The i-th entry equals the value of the piecewise-linear function at the i-th point of the configuration.

For an irregular triangulation, the normal cone is empty. In this case, a single point (the origin) is returned.

EXAMPLES:

```
sage: triangulation = polytopes.n_cube(2).triangulate(engine='internal')
sage: triangulation
(<0,1,3>, <0,2,3>)
sage: N = triangulation.normal_cone();  N
4-d cone in 4-d lattice
sage: N.rays()
(-1,  0,  0,  0),
```

```
( 1,  0,  1,  0),
(-1,  0, -1,  0),
( 1,  0,  0, -1),
(-1,  0,  0,  1),
( 1,  1,  0,  0),
(-1, -1,  0,  0)
in Ambient free module of rank 4
over the principal ideal domain Integer Ring
sage: N.dual().rays()
(-1, 1, 1, -1)
in Ambient free module of rank 4
over the principal ideal domain Integer Ring
```

TESTS:

```
sage: polytopes.n_simplex(2).triangulate().normal_cone()
3-d cone in 3-d lattice
sage: _.dual().is_trivial()
True
```

plot(***kwds*)

Produce a graphical representation of the triangulation.

EXAMPLES:

```
sage: p = PointConfiguration([[0,0],[0,1],[1,0],[1,1],[-1,-1]])
sage: triangulation = p.triangulate()
sage: triangulation
(<1,3,4>, <2,3,4>)
sage: triangulation.plot(axes=False)
```

point_configuration()

Returns the point configuration underlying the triangulation.

EXAMPLES:

```
sage: pconfig = PointConfiguration([[0,0],[0,1],[1,0]])
sage: pconfig
A point configuration in QQ^2 consisting of 3 points. The
triangulations of this point configuration are assumed to
be connected, not necessarily fine, not necessarily regular.
sage: triangulation = pconfig.triangulate()
sage: triangulation
(<0,1,2>)
sage: triangulation.point_configuration()
A point configuration in QQ^2 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
```

sage.geometry.triangulation.element.**triangulation_render_3d**(*triangulation*, ***kwds*)

Return a graphical representation of a 3-d triangulation.

INPUT:

- `triangulation` – a `Triangulation`.
- `**kwds` – keywords that are passed on to the graphics primitives.

OUTPUT:

A 3-d graphics object.

EXAMPLES:

```
sage: p = [[0,-1,-1],[0,0,1],[0,1,0], [1,-1,-1],[1,0,1],[1,1,0]]
sage: points = PointConfiguration(p)
sage: triang = points.triangulate()
sage: triang.plot(axes=False)      # indirect doctest
```

CHAPTER

NINETEEN

HYPERPLANE ARRANGEMENTS

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

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

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

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

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

19.1 Arrangements

There are several ways to create hyperplane arrangements:

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

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

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

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

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

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

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

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

New arrangements from old:

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

```
sage: a = hyperplane_arrangements.braid(3)
sage: b = a.union(hyperplane_arrangements.semiorder(3))
sage: b == a | hyperplane_arrangements.semiorder(3)    # alternate syntax
True
sage: b == hyperplane_arrangements.Catalan(3)
True
```

```
sage: a
Arrangement <t1 - t2 | t0 - t1 | t0 - t2>
sage: a = hyperplane_arrangements.coordinate(4)
```

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

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

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

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

```
sage: a = hyperplane_arrangements.semiorder(3)
sage: b = a.essentialization();   b
Arrangement of 6 hyperplanes of dimension 2 and rank 2
sage: b.n_regions()
19
sage: b.regions()
(A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 6 vertices,
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices and 1 ray,
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices and 1 ray,
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2 rays,
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices and 1 ray,
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2 rays,
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices and 1 ray,
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2 rays,
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices and 1 ray,
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2 rays,
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices and 1 ray,
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2 rays,
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex and 2 rays)
sage: b.bounded_regions()
(A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 6 vertices,
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices,
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 3 vertices)
sage: b.n_bounded_regions()
7
sage: a.unbounded_regions()
(A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 1 vertex, 2 rays, 1 line,
```

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

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.distance_enumerator(r1)  # generating function for distances from r1
6*x^3 + 6*x^2 + 6*x + 1
```

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

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

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

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

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

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

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

```
sage: charpoly(1)                                   # long time
751
```

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

Miscellaneous methods (see documentation for an explanation):

```
sage: a = hyperplane_arrangements.semiorder(3)
sage: a.has_good_reduction(5)
True
sage: b = a.change_ring(GF(5))
sage: pa = a.intersection_poset()
sage: pb = b.intersection_poset()
sage: pa.is_isomorphic(pb)
True
sage: a.face_vector()
(0, 12, 30, 19)
sage: a.face_vector()
(0, 12, 30, 19)
sage: a.is_central()
False
sage: a.is_linear()
False
sage: a.sign_vector((1,1,1))
(-1, 1, -1, 1, -1, 1)
sage: a.varchenko_matrix()
[          1          h2       h2*h4       h2*h3    h2*h3*h4 h2*h3*h4*h5]
[         h2           1          h4          h3       h3*h4    h3*h4*h5]
[      h2*h4          h4           1       h3*h4          h3       h3*h5]
[      h2*h3          h3       h3*h4           1          h4       h4*h5]
[   h2*h3*h4       h3*h4          h3          h4           1          h5]
[h2*h3*h4*h5    h3*h4*h5       h3*h5       h4*h5          h5           1]
```

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

TESTS:

```
sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: h = H([(1, 106), 106266], [(83, 101), 157866], [(111, 110), 186150], [(453, 221), 532686],
....:       [(407, 237), 516882], [(55, 32), 75620], [(221, 114), 289346], [(452, 115), 474217],
....:       [(406, 131), 453521], [(28, 9), 32446], [(287, 19), 271774], [(241, 35), 244022],
....:       [(231, 1), 210984], [(185, 17), 181508], [(23, -8), 16609])
sage: h.n_regions()
85

sage: H()
Empty hyperplane arrangement of dimension 2

sage: Zero = HyperplaneArrangements(QQ)
sage: Zero
Hyperplane arrangements in 0-dimensional linear space over Rational Field with coordinate
sage: Zero()
Empty hyperplane arrangement of dimension 0
sage: Zero.an_element()
Empty hyperplane arrangement of dimension 0
```

AUTHORS:

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

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

REFERENCES:

class sage.geometry.hyperplane_arrangement.arrangement.**HyperplaneArrangementElement**(*parent*, *hyperplanes*, *check=True*)

Bases: sage.structure.element.Element

An element in a hyperplane arrangement.

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

add_hyperplane(*other*)

The union of self with other.

INPUT:

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

OUTPUT:

A new hyperplane arrangement.

EXAMPLES:

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

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
```

change_ring(*base_ring*)

Return hyperplane arrangement over the new base ring.

INPUT:

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

OUTPUT:

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

EXAMPLES:

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

characteristic_polynomial()

Return the characteristic polynomial of the hyperplane arrangement.

OUTPUT:

The characteristic polynomial in $\mathbf{Q}[x]$.

EXAMPLES:

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

TESTS:

```
sage: H.<s,t,u,v> = HyperplaneArrangements(QQ)
sage: m = matrix([(0, -1, 0, 1, -1), (0, -1, 1, -1, 0), (0, -1, 1, 0, -1),
....:   (0, 1, 0, 0, 0), (0, 1, 0, 1, -1), (0, 1, 1, -1, 0), (0, 1, 1, 0, -1)])
sage: R.<x> = QQ[]
```

```
sage: expected_charpoly = (x - 1) * x * (x^2 - 6*x + 12)
sage: for s in SymmetricGroup(4):   # long time (about a second on a Core i7)
....:     m_perm = [m.column(i) for i in [0, s(1), s(2), s(3), s(4)]]
....:     m_perm = matrix(m_perm).transpose()
....:     charpoly = H(m_perm.rows()).characteristic_polynomial()
....:     assert charpoly == expected_charpoly
```

cone(*variable='t'*)

Return the cone over the hyperplane arrangement.

INPUT:

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

OUTPUT:

A new yperplane 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 $[0, 1, 0, \ldots, 0]$.

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)
```

deletion(*hyperplanes*)

Return the hyperplane arrangement obtained by removing `h`.

INPUT:

- `h` – a hyperplane or hyperplane arrangement

OUTPUT:

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

See Also:

`restriction()`

EXAMPLES:

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

```
sage: A.deletion(x)
Arrangement <y - 1 | y + 1 | x - y | x + y>
sage: h = H([0,1,0], [0,1,1])
sage: A.deletion(h)
Arrangement <y - 1 | y + 1 | x - y>
```

TESTS:

```
sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: A = H([0,1,0], [1,0,1], [-1,0,1], [0,1,-1], [0,1,1])
sage: h = H([0,4,0])
sage: A.deletion(h)
Arrangement <y - 1 | y + 1 | x - y | x + y>
sage: l = H([1,2,3])
sage: A.deletion(l)
Traceback (most recent call last):
...
ValueError: hyperplane is not in the arrangement
```

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 `base_region`, i.e., the number of hyperplanes separated by i hyperplanes from `base_region`.

EXAMPLES:

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

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

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

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

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

INPUT:

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

OUTPUT:

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

See Also:

`whitney_number()`, `whitney_data()`

EXAMPLES:

```
sage: A = hyperplane_arrangements.Shi(3)
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:

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:

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

sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: B = H([(1,0),1], [(1,0),-1])
```

```
sage: B.is_essential()
False
sage: B.essentialization()
Arrangement <-x + 1 | x + 1>
sage: B.essentialization().parent()
Hyperplane arrangements in 1-dimensional linear space over
Rational Field with coordinate x

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

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

TESTS:

```
sage: b = hyperplane_arrangements.coordinate(2)
sage: b.is_essential()
True
sage: b.essentialization() is b
True
```

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 is the intersection of a region with a hyperplane of the arrangehment.

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 number of hyperplanes in the arrangement.

OUTPUT:

An integer.

EXAMPLES:

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

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

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

intersection_poset()

Return the intersection poset of the hyperplane arrangement.

OUTPUT:

The poset of non-empty intersections of hyperplanes.

EXAMPLES:

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

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

is_central()

Test whether the intersection of all the hyperplanes is nonempty.

OUTPUT:

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

EXAMPLES:

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

is_essential()

Test whether the hyperplane arrangement is essential.

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

See Also:

essentialization()

OUTPUT:

A boolean indicating whether the hyperplane arrangement is essential.

EXAMPLES:

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

is_linear()

Test whether all hyperplanes pass through the origin.

OUTPUT:

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

EXAMPLES:

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

sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: c = H(x+1, y+1)
sage: c.is_linear()
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 hyperplane separate the given regions.

EXAMPLES:

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

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

OUTPUT:

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

EXAMPLES:

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

TESTS:

```
sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: A = H([(1,1),0], [(2,3),-1], [(4,5),3])
sage: B = A.change_ring(FiniteField(7))
sage: B.n_bounded_regions()
Traceback (most recent call last):
...
TypeError: base field must have characteristic zero
```

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

OUTPUT:

An integer.

EXAMPLES:

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

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

OUTPUT:

An integer.

EXAMPLES:

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

TESTS:

```
sage: H.<x,y> = HyperplaneArrangements(QQ)
sage: A = H([(1,1), 0], [(2,3), -1], [(4,5), 3])
sage: B = A.change_ring(FiniteField(7))
sage: B.n_regions()
Traceback (most recent call last):
...
TypeError: base field must have characteristic zero
```

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()
```

poincare_polynomial()

Return the Poincare polynomial of the hyperplane arrangement.

OUTPUT:

The Poincare polynomial in $\mathbf{Q}[x]$.

EXAMPLES:

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

rank()

Return the rank.

OUTPUT:

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

EXAMPLES:

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

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

sage: p = polytopes.n_simplex(5)
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.

EXAMPLES:

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

TESTS:

```
sage: A = H([(1,1),0], [(2,3),-1], [(4,5),3])
sage: B = A.change_ring(FiniteField(7))
sage: B.region_containing_point((1,2))
Traceback (most recent call last):
...
ValueError: base field must have characteristic zero

sage: A = H([(1,1),0], [(2,3),-1], [(4,5),3])
sage: A.region_containing_point((1,-1))
Traceback (most recent call last):
...
ValueError: point 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 $\mathbf{R}^n$.

EXAMPLES:

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

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

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

restriction(*hyperplane*)

Return the restriction to a hyperplane.

INPUT:

- hyperplane – a hyperplane of the hyperplane arrangement

OUTPUT:

The restriction of the hyperplane arrangement to the given hyperplane.

EXAMPLES:

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

See Also:

deletion()

sign_vector(*p*)

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

The base field must have characteristic zero.

INPUT:

- p – point as a list/tuple/iterable

OUTPUT:

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

EXAMPLES:

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

TESTS:

```
sage: H.<x,y> = HyperplaneArrangements(GF(3))
sage: A = H(x, y)
sage: A.sign_vector([1, 2])
Traceback (most recent call last):
...
ValueError: characteristic must be zero
```

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

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 $S = \mathbf{Q}[h_1, \ldots, h_s]$, 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)
sage: v = a.varchenko_matrix();  v
[    1    h2    h1]
[   h2     1 h1*h2]
[   h1 h1*h2     1]
sage: factor(det(v))
(h2 - 1) * (h2 + 1) * (h1 - 1) * (h1 + 1)
```

vertices(*exclude_sandwiched=False*)

Return the vertices.

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

INPUT:

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

OUTPUT:

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

EXAMPLES:

```
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()

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

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 Moebius function values $mu(0, x)$ over all x in the intersection poset with $\mathrm{rank}(x) = k$.

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

See [GZ] 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)
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.UniqueRepresenta

Hyperplane arrangements.

For more information on hyperplane arrangements, see sage.geometry.hyperplane_arrangements.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:

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

change_ring(*base_ring*)
Return hyperplane arrangements over a different base ring.

INPUT:

- base_ring – a ring; the new base ring.

OUTPUT:

A new HyperplaneArrangements instance over the new base ring.

EXAMPLES:

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

TESTS:

```
sage: L.change_ring(QQ) is L
True
```

gen(*i*)

Return the i-th coordinate hyperplane.

INPUT:

- `i` – integer

OUTPUT:

A linear expression.

EXAMPLES:

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

gens()

Return the coordinate hyperplanes.

OUTPUT:

A tuple of linear expressions, one for each linear variable.

EXAMPLES:

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

ngens()

Return the number of linear variables.

OUTPUT:

An integer.

EXAMPLES:

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

CHAPTER

TWENTY

LIBRARY OF HYPERPLANE ARRANGEMENTS

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

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

Bases: object

The library of hyperplane arrangements.

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

Return the Catalan arrangement.

INPUT:

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

OUTPUT:

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

EXAMPLES:

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

TESTS:

```
sage: h = hyperplane_arrangements.Catalan(5)
sage: h.characteristic_polynomial()
x^5 - 30*x^4 + 335*x^3 - 1650*x^2 + 3024*x
sage: h.characteristic_polynomial.clear_cache()  # long time
sage: h.characteristic_polynomial()              # long time
x^5 - 30*x^4 + 335*x^3 - 1650*x^2 + 3024*x
```

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

Return the Shi hyperplane arrangement of a graph G.

INPUT:

- G – graph
- K – field (default: $\mathbf{Q}$)

- names – tuple of strings or None (default); the variable names for the ambient space

OUTPUT:

The Shi hyperplane arrangement of the given graph G.

EXAMPLES:

```
sage: 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: $\mathbf{Q}$)
- 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
```

TESTS:

```
sage: a.characteristic_polynomial.clear_cache()  # long time
sage: a.characteristic_polynomial()              # long time
x^3 - 6*x^2 + 9*x
```

REFERENCES:

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

Return the Shi arrangement.

INPUT:

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

OUTPUT:

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

EXAMPLES:

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

TESTS:

```
sage: h = hyperplane_arrangements.Shi(4)
sage: h.characteristic_polynomial()
x^4 - 12*x^3 + 48*x^2 - 64*x
sage: h.characteristic_polynomial.clear_cache()  # long time
sage: h.characteristic_polynomial()              # long time
x^4 - 12*x^3 + 48*x^2 - 64*x
```

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

Return a bigraphical hyperplane arrangement.

INPUT:

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

OUTPUT:

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

EXAMPLES:

```
sage: 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:

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

The braid arrangement.

INPUT:

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

OUTPUT:

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

EXAMPLES:

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

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

Return the coordinate hyperplane arrangement.

INPUT:

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

OUTPUT:

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

EXAMPLES:

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

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

Return the graphical hyperplane arrangement of a graph G.

INPUT:

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

OUTPUT:

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

EXAMPLES:

```
sage: 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
```

TESTS:

```
sage: h = hyperplane_arrangements.graphical(g)
sage: h.characteristic_polynomial()
x^5 - 6*x^4 + 14*x^3 - 15*x^2 + 6*x
sage: h.characteristic_polynomial.clear_cache()  # long time
sage: h.characteristic_polynomial()              # long time
x^5 - 6*x^4 + 14*x^3 - 15*x^2 + 6*x
```

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

Return the linial hyperplane arrangement.

INPUT:

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

OUTPUT:

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

EXAMPLES:

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

TESTS:

```
sage: h = hyperplane_arrangements.linial(5)
sage: h.characteristic_polynomial()
x^5 - 10*x^4 + 45*x^3 - 100*x^2 + 90*x
sage: h.characteristic_polynomial.clear_cache()  # long time
sage: h.characteristic_polynomial()              # long time
x^5 - 10*x^4 + 45*x^3 - 100*x^2 + 90*x
```

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

Return the semiorder arrangement.

INPUT:

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

OUTPUT:

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

EXAMPLES:

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

TESTS:

```
sage: h = hyperplane_arrangements.semiorder(5)
sage: h.characteristic_polynomial()
x^5 - 20*x^4 + 180*x^3 - 790*x^2 + 1380*x
sage: h.characteristic_polynomial.clear_cache()  # long time
sage: h.characteristic_polynomial()              # long time
x^5 - 20*x^4 + 180*x^3 - 790*x^2 + 1380*x
```

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

Construct the parent for the hyperplane arrangements.

For internal use only.

INPUT:

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

OUTPUT:

A new `HyperplaneArrangements` instance.

EXAMPLES:

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

CHAPTER

TWENTYONE

HYPERPLANES

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

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

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

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

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

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

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

```
sage: V(v)
Hyperplane 3*x + 2*y - 5*z - 7
```

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

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

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

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

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

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

Which you can't do with hyperplane arrangements:

```
sage: arrangement = H(h, x, y, x+y-1);  arrangement
Arrangement <y | x | x + y - 1 | 3*x + 2*y - 5*z - 7>
sage: arrangement + x
Traceback (most recent call last):
TypeError: unsupported operand type(s) for +:
'HyperplaneArrangements_with_category.element_class' and
'HyperplaneArrangements_with_category.element_class'
```

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.

TESTS:

```
sage: from sage.geometry.hyperplane_arrangement.hyperplane import AmbientVectorSpace
sage: V = AmbientVectorSpace(QQ, ('x', 'y'))
sage: V.change_ring(QQ) is V
True
```

Element
alias of `Hyperplane`

change_ring(*base_ring*)
Return a ambient vector space with a changed base ring.

INPUT:

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

OUTPUT:

A new `AmbientVectorSpace`.

EXAMPLES:

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

TESTS:

```
sage: V.change_ring(QQ) is V
True
```

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
```

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

Bases: `sage.geometry.linear_expression.LinearExpression`

A hyperplane.

You shoud 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 empty and full space:

```
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: 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: 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.n_cube(3))
A 2-dimensional polyhedron in QQ^3 defined as the convex hull of 3 vertices
```

linear_part()

The linear part of the affine space.

OUTPUT:

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

EXAMPLES:

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

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 is one less than the ambient space dimension.

EXAMPLES:

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

normal()

Return the normal vector.

OUTPUT:

A vector over the base ring.

EXAMPLES:

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

orthogonal_projection(*point*)

Return the orthogonal projection of a point.

INPUT:

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

OUTPUT:

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

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

EXAMPLES:

```
sage: H.<x,y,z> = HyperplaneArrangements(QQ)
sage: h = x + 2*y + 3*z - 4
sage: p1 = h.orthogonal_projection(0);  p1
(2/7, 4/7, 6/7)
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:

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

point()

Return the point closest to the origin.

OUTPUT:

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

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

EXAMPLES:

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

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

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

polyhedron()

Return the hyperplane as a polyhedron.

OUTPUT:

A `Polyhedron()` instance.

EXAMPLES:

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

CHAPTER

TWENTYTWO

AFFINE SUBSPACES OF A VECTOR SPACE

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

EXAMPLES:

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

TESTS:

```
sage: A = AffineSubspace([2], VectorSpace(QQ, 1))
sage: A.point()
(2)
```

```
sage: A.linear_part()
Vector space of dimension 1 over Rational Field
sage: A.linear_part().basis_matrix()
[1]
sage: A = AffineSubspace([], VectorSpace(QQ, 0))
sage: A.point()
()
sage: A.linear_part()
Vector space of dimension 0 over Rational Field
sage: A.linear_part().basis_matrix()
[]
```

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

Bases: sage.structure.sage_object.SageObject

An affine subspace.

INPUT:

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

OUTPUT:

Affine subspace parallel to V and passing through p.

EXAMPLES:

```
sage: from sage.geometry.hyperplane_arrangement.affine_subspace import AffineSubspace
sage: a = AffineSubspace([1,0,0,0], VectorSpace(QQ,4))
sage: a
Affine space p + W where:
  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
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()
Vector space of dimension 3 over Rational Field
```

point()

Return a point `p` in the affine space.

OUTPUT:

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

EXAMPLES:

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

```
sage: A.point()
(2, 3, 1)
```

CHAPTER

TWENTYTHREE

LINEAR EXPRESSIONS

A linear expression is just a linear polynomial in some (fixed) variables. This class only implements linear expressions for others to use.

EXAMPLES:

```
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L.<x,y,z> = LinearExpressionModule(QQ);  L
Module of linear expressions in variables x, y, z over Rational Field
sage: x + 2*y + 3*z + 4
x + 2*y + 3*z + 4
```

You can also pass coefficients and a constant term to construct linear expressions:

```
sage: L([1, 2, 3], 4)
x + 2*y + 3*z + 4
sage: L([(1, 2, 3), 4])
x + 2*y + 3*z + 4
sage: L([4, 1, 2, 3])   # note: constant is first in single-tuple notation
x + 2*y + 3*z + 4
```

The linear expressions are a module under the base ring, so you can add them and multiply them with scalars:

```
sage: m = x + 2*y + 3*z + 4
sage: 2*m
2*x + 4*y + 6*z + 8
sage: m+m
2*x + 4*y + 6*z + 8
sage: m-m
0*x + 0*y + 0*z + 0
```

class sage.geometry.linear_expression.**LinearExpression**(*parent*, *coefficients*, *constant*, *check=True*)

Bases: sage.structure.element.ModuleElement

A linear expression.

A linear expression is just a linear polynomial in some (fixed) variables.

EXAMPLES:

```
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L.<x,y,z> = LinearExpressionModule(QQ)
sage: m = L([1, 2, 3], 4); m
x + 2*y + 3*z + 4
sage: m2 = L([(1, 2, 3), 4]); m2
```

```
x + 2*y + 3*z + 4
sage: m3 = L([4, 1, 2, 3]); m3   # note: constant is first in single-tuple notation
x + 2*y + 3*z + 4
sage: m == m2
True
sage: m2 == m3
True
sage: 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:

```
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:

```
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:

```
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
```

evaluate(*point*)

Evaluate the linear expression.

INPUT:

- `point` – list/tuple/iterable of coordinates; the coordinates of a point

OUTPUT:

The linear expression $Ax + b$ evaluated at the point x.

EXAMPLES:

```
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L.<x,y> = LinearExpressionModule(QQ)
sage: ex = 2*x + 3* y + 4
sage: ex.evaluate([1,1])
9
sage: ex([1,1])    # syntactic sugar
9
```

```
sage: ex([pi, e])
2*pi + 3*e + 4
```

class sage.geometry.linear_expression.**LinearExpressionModule**(*base_ring*, *names=()*)

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

The module of linear expressions.

This is the module of linear polynomials which is the parent for linear expressions.

EXAMPLES:

```
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L = LinearExpressionModule(QQ, ('x', 'y', 'z'))
sage: L
Module of linear expressions in variables x, y, z over Rational Field
sage: L.an_element()
x + 0*y + 0*z + 0
```

Element

alias of LinearExpression

ambient_module()

Return the ambient module.

See Also:

ambient_vector_space()

OUTPUT:

The domain of the linear expressions as a free module over the base ring.

EXAMPLES:

```
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L = LinearExpressionModule(QQ, ('x', 'y', 'z'))
sage: L.ambient_module()
Vector space of dimension 3 over Rational Field
sage: M = LinearExpressionModule(ZZ, ('r', 's'))
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:

```
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
```

change_ring(*base_ring*)

Return a new module with a changed base ring.

INPUT:

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

OUTPUT:

A new linear expression over the new base ring.

EXAMPLES:

```
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: M.<y> = LinearExpressionModule(ZZ)
sage: L = M.change_ring(QQ);  L
Module of linear expressions in variable y over Rational Field
```

TESTS:

```
sage: L.change_ring(QQ) is L
True
```

gen(*i*)

Return the i-th generator.

INPUT:

- `i` – integer

OUTPUT:

A linear expression.

EXAMPLES:

```
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L = LinearExpressionModule(QQ, ('x', 'y', 'z'))
sage: L.gen(0)
x + 0*y + 0*z + 0
```

gens()

Return the generators.

OUTPUT:

A tuple of linear expressions, one for each linear variable.

EXAMPLES:

```
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L = LinearExpressionModule(QQ, ('x', 'y', 'z'))
sage: L.gens()
(x + 0*y + 0*z + 0, 0*x + y + 0*z + 0, 0*x + 0*y + z + 0)
```

ngens()

Return the number of linear variables.

OUTPUT:

An integer.

EXAMPLES:

```
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L = LinearExpressionModule(QQ, ('x', 'y', 'z'))
sage: L.ngens()
3
```

random_element()

Return a random element.

EXAMPLES:

```
sage: from sage.geometry.linear_expression import LinearExpressionModule
sage: L.<x,y,z> = LinearExpressionModule(QQ)
sage: L.random_element()
-1/2*x - 1/95*y + 1/2*z - 12
```

CHAPTER

TWENTYFOUR

BACKENDS FOR POLYHEDRAL COMPUTATIONS

24.1 The cdd backend for polyhedral computations

class sage.geometry.polyhedron.backend_cdd.**Polyhedron_QQ_cdd**(*parent*, *Vrep*, *Hrep*, ***kwds*)

Bases: sage.geometry.polyhedron.backend_cdd.Polyhedron_cdd, sage.geometry.polyhedron.base_QQ.Polyhedron_QQ

Polyhedra over QQ with cdd

INPUT:

- parent – the parent, an instance of Polyhedra.
- Vrep – a list [vertices, rays, lines] or None.
- Hrep – a list [ieqs, eqns] or None.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.parent import Polyhedra
sage: parent = Polyhedra(QQ, 2, backend='cdd')
sage: from sage.geometry.polyhedron.backend_cdd import Polyhedron_QQ_cdd
sage: Polyhedron_QQ_cdd(parent, [ [(1,0),(0,1),(0,0)], [], []], None, 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**(*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:

```
sage: from sage.geometry.polyhedron.parent import Polyhedra
sage: parent = Polyhedra(RDF, 2, backend='cdd')
sage: from sage.geometry.polyhedron.backend_cdd import Polyhedron_RDF_cdd
```

```
sage: Polyhedron_RDF_cdd(parent, [ [(1,0),(0,1),(0,0)], [], []], None, verbose=False)
A 2-dimensional polyhedron in RDF^2 defined as the convex hull of 3 vertices
```

class sage.geometry.polyhedron.backend_cdd.**Polyhedron_cdd**(*parent*, *Vrep*, *Hrep*, ***kwds*)

Bases: sage.geometry.polyhedron.base.Polyhedron_base

Base class for the cdd backend.

24.2 The PPL (Parma Polyhedra Library) backend for polyhedral computations

class sage.geometry.polyhedron.backend_ppl.**Polyhedron_QQ_ppl**(*parent*, *Vrep*, *Hrep*, ***kwds*)

Bases: sage.geometry.polyhedron.backend_ppl.Polyhedron_ppl, sage.geometry.polyhedron.base_QQ.Polyhedron_QQ

Polyhedra over **Q** with ppl

INPUT:

- `Vrep` – a list `[vertices, rays, lines]` or `None`.
- `Hrep` – a list `[ieqs, eqns]` or `None`.

EXAMPLES:

```
sage: p = Polyhedron(vertices=[(0,0),(1,0),(0,1)], rays=[(1,1)], lines=[],
...                  backend='ppl', base_ring=QQ)
sage: TestSuite(p).run(skip='_test_pickling')
```

class sage.geometry.polyhedron.backend_ppl.**Polyhedron_ZZ_ppl**(*parent*, *Vrep*, *Hrep*, ***kwds*)

Bases: sage.geometry.polyhedron.backend_ppl.Polyhedron_ppl, sage.geometry.polyhedron.base_ZZ.Polyhedron_ZZ

Polyhedra over **Z** with ppl

INPUT:

- `Vrep` – a list `[vertices, rays, lines]` or `None`.
- `Hrep` – a list `[ieqs, eqns]` or `None`.

EXAMPLES:

```
sage: p = Polyhedron(vertices=[(0,0),(1,0),(0,1)], rays=[(1,1)], lines=[])
...                  backend='ppl', base_ring=ZZ)
sage: TestSuite(p).run(skip='_test_pickling')
```

class sage.geometry.polyhedron.backend_ppl.**Polyhedron_ppl**(*parent*, *Vrep*, *Hrep*, ***kwds*)

Bases: sage.geometry.polyhedron.base.Polyhedron_base

Polyhedra with ppl

INPUT:

- `Vrep` – a list `[vertices, rays, lines]` or `None`.
- `Hrep` – a list `[ieqs, eqns]` or `None`.

EXAMPLES:

```
sage: p = Polyhedron(vertices=[(0,0),(1,0),(0,1)], rays=[(1,1)], lines=[], backend='ppl')
sage: TestSuite(p).run()
```

24.3 The Python backend

While slower than specialized C/C++ implementations, the implementation is general and works with any exact field in Sage that allows you to define polyhedra.

EXAMPLES:

```
sage: 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*, ***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:

```
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()
```

TESTS:

```
sage: K.<sqrt3> = NumberField(x^2-3)
sage: p = Polyhedron([(0,0), (1,0), (1/2, sqrt3/2)])
sage: TestSuite(p).run()
```

24.4 Double Description Algorithm for Cones

This module implements the double description algorithm for extremal vertex enumeration in a pointed cone following [FukudaProdon]. With a little bit of preprocessing (see double_description_inhomogeneous) this defines a backend for polyhedral computations. But as far as this module is concerned, *inequality* always means without a constant term and the origin is always a point of the cone.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.double_description import StandardAlgorithm
sage: A = matrix(QQ, [(1,0,1), (0,1,1), (-1,-1,1)])
sage: alg = StandardAlgorithm(A);  alg
Pointed cone with inequalities
(1, 0, 1)
(0, 1, 1)
(-1, -1, 1)
sage: DD, _ = alg.initial_pair();  DD
Double description pair (A, R) defined by
    [ 1  0  1]        [ 2/3 -1/3 -1/3]
A = [ 0  1  1],   R = [-1/3  2/3 -1/3]
    [-1 -1  1]        [ 1/3  1/3  1/3]
```

The implementation works over any exact field that is embedded in $\mathbf{R}$, for example:

```
sage: from sage.geometry.polyhedron.double_description import StandardAlgorithm
sage: A = matrix(AA, [(1,0,1), (0,1,1), (-AA(2).sqrt(),-AA(3).sqrt(),1),
....:                 (-AA(3).sqrt(),-AA(2).sqrt(),1)])
sage: alg = StandardAlgorithm(A)
sage: alg.run().R
((-0.41773766770041119?, 0.5822623322995881?, 0.41773766770041119?),
 (-0.2411809548974793?, -0.2411809548974793?, 0.2411809548974793?),
 (0.07665629029830300?, 0.07665629029830300?, 0.2411809548974793?),
 (0.5822623322995881?, -0.41773766770041119?, 0.41773766770041119?))
```

REFERENCES:

class sage.geometry.polyhedron.double_description.**DoubleDescriptionPair**(*problem*, *A_rows*, *R_cols*)

Bases: sage.structure.sage_object.SageObject

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.

TESTS:

```
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: DoubleDescriptionPair(alg,
....:     [(1, 0, 1), (0, 1, 1), (-1, -1, 1)],
....:     [(2/3, -1/3, 1/3), (-1/3, 2/3, 1/3), (-1/3, -1/3, 1/3)])
Double description pair (A, R) defined by
    [ 1  0  1]        [ 2/3 -1/3 -1/3]
A = [ 0  1  1],   R = [-1/3  2/3 -1/3]
    [-1 -1  1]        [ 1/3  1/3  1/3]
```

R_by_sign(*a*)

Classify the rays into those that are positive, zero, and negative on a.

INPUT:

- a – vector. Coefficient vector of a homogeneous inequality.

OUTPUT:

A triple consisting of the rays (columns of R) that are positive, zero, and negative on a. In that order.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.double_description import StandardAlgorithm
sage: A = matrix(QQ, [(1,0,1), (0,1,1), (-1,-1,1)])
sage: DD, _ = StandardAlgorithm(A).initial_pair()
sage: DD.R_by_sign(vector([1,-1,0]))
([(2/3, -1/3, 1/3)], [(-1/3, -1/3, 1/3)], [(-1/3, 2/3, 1/3)])
sage: DD.R_by_sign(vector([1,1,1]))
([(2/3, -1/3, 1/3), (-1/3, 2/3, 1/3)], [], [(-1/3, -1/3, 1/3)])
```

are_adjacent(*r1*, *r2*)

Return whether the two rays are adjacent.

INPUT:

- r1, r2 – two rays.

OUTPUT:

Boolean. Whether the two rays are adjacent.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.double_description import StandardAlgorithm
sage: A = matrix(QQ, [(0,1,0), (1,0,0), (0,-1,1), (-1,0,1)])
sage: DD = StandardAlgorithm(A).run()
sage: DD.are_adjacent(DD.R[0], DD.R[1])
True
sage: DD.are_adjacent(DD.R[0], DD.R[2])
True
sage: DD.are_adjacent(DD.R[0], DD.R[3])
False
```

cone()

Return the cone defined by A.

This method is for debugging only. Assumes that the base ring is $\mathbf{Q}$.

OUTPUT:

The cone defined by the inequalities as a Polyhedron(), using the PPL backend.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.double_description import StandardAlgorithm
sage: A = matrix(QQ, [(1,0,1), (0,1,1), (-1,-1,1)])
sage: DD, _ = StandardAlgorithm(A).initial_pair()
sage: DD.cone().Hrepresentation()
(An inequality (-1, -1, 1) x + 0 >= 0,
 An inequality (0, 1, 1) x + 0 >= 0,
 An inequality (1, 0, 1) x + 0 >= 0)
```

dual()

Return the dual.

OUTPUT:

For the double description pair (A, R) this method returns the dual double description pair (R^T, A^T)

EXAMPLES:

```
sage: from sage.geometry.polyhedron.double_description import Problem
sage: A = matrix(QQ, [(0,1,0), (1,0,0), (0,-1,1), (-1,0,1)])
sage: DD, _ = Problem(A).initial_pair()
sage: DD
Double description pair (A, R) defined by
    [ 0  1  0]        [0 1 0]
A = [ 1  0  0],   R = [1 0 0]
    [ 0 -1  1]        [1 0 1]
sage: DD.dual()
 Double description pair (A, R) defined by
    [0 1 1]        [ 0  1  0]
A = [1 0 0],   R = [ 1  0 -1]
    [0 0 1]        [ 0  0  1]
```

first_coordinate_plane()

Restrict to the first coordinate plane.

OUTPUT:

A new double description pair with the constraint $x_0 = 0$ added.

EXAMPLES:

```
sage: A = matrix([(1, 1), (-1, 1)])
sage: from sage.geometry.polyhedron.double_description import StandardAlgorithm
sage: DD, _ = StandardAlgorithm(A).initial_pair()
sage: DD
Double description pair (A, R) defined by
A = [ 1  1],   R = [ 1/2 -1/2]
    [-1  1]        [ 1/2  1/2]
sage: DD.first_coordinate_plane()
Double description pair (A, R) defined by
    [ 1  1]
A = [-1  1],   R = [  0]
    [-1  0]        [1/2]
    [ 1  0]
```

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
```

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 $\mathbf{Q}$.

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 tuple 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)
((0, 1, 1), (-1, -1, 1))
```

class sage.geometry.polyhedron.double_description.**Problem**(*A*)

Bases: sage.structure.sage_object.SageObject

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.

TESTS:

```
sage: A = matrix([(1, 1), (-1, 1)])
sage: from sage.geometry.polyhedron.double_description import Problem
sage: Problem(A)
Pointed cone with inequalities
(1, 1)
(-1, 1)
```

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:

```
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 [FukudaProdon] for the definition of the "Standard Algorithm".

EXAMPLES:

```
sage: A = matrix(QQ, [(1, 1), (-1, 1)])
sage: from sage.geometry.polyhedron.double_description import StandardAlgorithm
sage: DD = StandardAlgorithm(A).run()
sage: DD.R    # the extremal rays
((1/2, 1/2), (-1/2, 1/2))
```

pair_class
alias of StandardDoubleDescriptionPair

run()
Run the Standard Algorithm.

OUTPUT:

A double description pair (A, R) of all inequalities as a DoubleDescriptionPair. By virtue of the double description algorithm, the columns of R are the extremal rays.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.double_description import StandardAlgorithm
sage: A = matrix(QQ, [(0,1,0), (1,0,0), (0,-1,1), (-1,0,1)])
sage: StandardAlgorithm(A).run()
Double description pair (A, R) defined by
    [ 0  1  0]         [0 0 1 1]
A = [ 1  0  0],    R = [1 0 1 0]
    [ 0 -1  1]         [1 1 1 1]
    [-1  0  1]
```

class sage.geometry.polyhedron.double_description.**StandardDoubleDescriptionPair**(*problem*, *A_rows*, *R_cols*)

Bases: sage.geometry.polyhedron.double_description.DoubleDescriptionPair

Double description pair for the "Standard Algorithm".

See StandardAlgorithm.

TESTS:

```
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: type(DD)
<class 'sage.geometry.polyhedron.double_description.StandardDoubleDescriptionPair'>
```

add_inequality(*a*)

Return a new double description pair with the inequality a added.

INPUT:

- a – vector. An inequality.

OUTPUT:

A new StandardDoubleDescriptionPair instance.

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: newDD = DD.add_inequality(vector([1,0,0]));  newDD
Double description pair (A, R) defined by
    [  -1    1    0]         [   1    1    0    0]
A = [  -1    2    1],    R = [   1    1    1    1]
    [ 1/2 -1/2   -1]         [   0   -1 -1/2   -2]
    [   1    0    0]
```

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 inequalites 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()
```

24.5 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:

```
sage: from sage.geometry.polyhedron.double_description_inhomogeneous     ....:     import Hrep2Vrep,
sage: Hrep2Vrep(QQ, 2, [(1,2,3), (2,4,3)], [])
[-1/2|-1/2  1/2|]
[   0| 2/3 -1/3|]
```

Note that the columns of the printed matrix are the vertices, rays, and lines of the minimal V-representation. Dually, the rows of the following are the inequalities and equations:

```
sage: Vrep2Hrep(QQ, 2, [(-1/2,0)], [(-1/2,2/3), (1/2,-1/3)], [])
[1 2 3]
[2 4 3]
[-----]
```

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 inequalitiy is given as constant term, `dim` coefficients.
- `equations` – list of equations. Same notation as for inequalities.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.double_description_inhomogeneous import Hrep2Vrep
sage: Hrep2Vrep(QQ, 2, [(1,2,3), (2,4,3)], [])
[-1/2|-1/2  1/2|]
[   0| 2/3 -1/3|]
sage: Hrep2Vrep(QQ, 2, [(1,2,3), (2,-2,-3)], [])
[   1 -1/2||   1]
```

```
[   0    0||-2/3]
sage: Hrep2Vrep(QQ, 2, [(1,2,3), (2,2,3)], [])
[-1/2| 1/2|   1]
[   0|   0|-2/3]
sage: Hrep2Vrep(QQ, 2, [(8,7,-2), (1,-4,3), (4,-3,-1)], [])
[ 1  0 -2||]
[ 1  4 -3||]
sage: Hrep2Vrep(QQ, 2, [(1,2,3), (2,4,3), (5,-1,-2)], [])
[-19/5  -1/2| 2/33  1/11|]
[ 22/5     0|-1/33 -2/33|]
sage: Hrep2Vrep(QQ, 2, [(0,2,3), (0,4,3), (0,-1,-2)], [])
[   0| 1/2  1/3|]
[   0|-1/3 -1/6|]
sage: Hrep2Vrep(QQ, 2, [], [(1,2,3), (7,8,9)])
[-2||]
[ 1||]
sage: Hrep2Vrep(QQ, 2, [(1,0,0)], [])    # universe
[0||1 0]
[0||0 1]
sage: Hrep2Vrep(QQ, 2, [(-1,0,0)], [])   # empty
[]
sage: Hrep2Vrep(QQ, 2, [], [])   # empty
[]
```

verify(*inequalities*, *equations*)

Compare result to PPL if the base ring is QQ.

This method is for debugging purposes and compares the computation with another backend if available.

INPUT:

- `inequalities`, `equations` – see `Hrep2Vrep`.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.double_description_inhomogeneous import Hrep2Vrep
sage: H = Hrep2Vrep(QQ, 1, [(1,2)], [])
sage: H.verify([(1,2)], [])
```

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:

```
sage: from sage.geometry.polyhedron.double_description_inhomogeneous ....:     impor
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 `dim` coordinates.
- `rays` – list of rays. Each ray is given as list of `dim` coordinates, not all zero.
- `lines` – list of line generators. Each line is given as list of `dim` coordinates, not all zero.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.double_description_inhomogeneous import Vrep2Hrep
sage: Vrep2Hrep(QQ, 2, [(-1/2,0)], [(-1/2,2/3), (1/2,-1/3)], [])
[1 2 3]
[2 4 3]
[-----]

sage: Vrep2Hrep(QQ, 2, [(1,0), (-1/2,0)], [], [(1,-2/3)])
[ 1/3  2/3    1]
[ 2/3 -2/3   -1]
[--------------]

sage: Vrep2Hrep(QQ, 2, [(-1/2,0)], [(1/2,0)], [(1,-2/3)])
[1 2 3]
[-----]

sage: Vrep2Hrep(QQ, 2, [(1,1), (0,4), (-2,-3)], [], [])
[ 8/13  7/13 -2/13]
[ 1/13 -4/13  3/13]
[ 4/13 -3/13 -1/13]
[-----------------]

sage: Vrep2Hrep(QQ, 2, [(-19/5,22/5), (-1/2,0)], [(2/33,-1/33), (1/11,-2/33)], [])
[10/11 -2/11 -4/11]
[ 66/5 132/5  99/5]
[ 2/11  4/11  6/11]
[-----------------]

sage: Vrep2Hrep(QQ, 2, [(0,0)], [(1/2,-1/3), (1/3,-1/6)], [])
[  0  -6 -12]
[  0  12  18]
[-----------]

sage: Vrep2Hrep(QQ, 2, [(-1/2,0)], [], [(1,-2/3)])
[-----]
```

```
[1 2 3]

sage: Vrep2Hrep(QQ, 2, [(-1/2,0)], [], [(1,-2/3), (1,0)])
[]
```

verify(*vertices*, *rays*, *lines*)

Compare result to PPL if the base ring is QQ.

This method is for debugging purposes and compares the computation with another backend if available.

INPUT:

- `vertices`, `rays`, `lines` – see `Vrep2Hrep`.

EXAMPLES:

```
sage: from sage.geometry.polyhedron.double_description_inhomogeneous import Vrep2Hrep
sage: vertices = [(-1/2,0)]
sage: rays = [(-1/2,2/3), (1/2,-1/3)]
sage: lines = []
sage: V2H = Vrep2Hrep(QQ, 2, vertices, rays, lines)
sage: V2H.verify(vertices, rays, lines)
```

24.6 Base class for polyhedra

class `sage.geometry.polyhedron.base.`**Polyhedron_base**(*parent*, *Vrep*, *Hrep*, ***kwds*)

Bases: `sage.structure.element.Element`

Base class for Polyhedron objects

INPUT:

- `parent` – the parent, an instance of `Polyhedra`.
- `Vrep` – a list `[vertices, rays, lines]` or `None`. The V-representation of the polyhedron. If `None`, the polyhedron is determined by the H-representation.
- `Hrep` – a list `[ieqs, eqns]` or `None`. The H-representation of the polyhedron. If `None`, the polyhedron is determined by the V-representation.

Only one of `Vrep` or `Hrep` can be different from `None`.

TESTS:

```
sage: p = Polyhedron()
sage: TestSuite(p).run()
```

Hrep_generator()

Return an iterator over the objects of the H-representation (inequalities or equations).

EXAMPLES:

```
sage: p = polytopes.n_cube(3)
sage: p.Hrep_generator().next()
An inequality (0, 0, -1) x + 1 >= 0
```

Hrepresentation(*index=None*)

Return the objects of the H-representaton. 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.n_cube(3)
sage: p.Hrepresentation(0)
An inequality (0, 0, -1) x + 1 >= 0
sage: p.Hrepresentation(0) == p.Hrepresentation() [0]
True
```

Hrepresentation_space()

Return the linear space containing the H-representation vectors.

OUTPUT:

A free module over the base ring of dimension `ambient_dim()` + 1.

EXAMPLES:

```
sage: poly_test = Polyhedron(vertices = [[1,0,0,0],[0,1,0,0]])
sage: poly_test.Hrepresentation_space()
Ambient free module of rank 5 over the principal ideal domain Integer Ring
```

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.n_cube(3)
sage: Y = Polyhedron(vertices=[(0,0,0), (0,0,1), (0,1,0), (1,0,0)]) / 2
sage: (X+Y)-Y == X
True
sage: (X-Y)+Y < X
True
```

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.n_cube(4)
sage: four_simplex = Polyhedron(vertices = [[0, 0, 0, 1], [0, 0, 1, 0], [0, 1, 0, 0], [1, 0,
sage: four_cube - four_simplex
A 4-dimensional polyhedron in ZZ^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_r
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_eggs
A 4-dimensional polyhedron in QQ^4 defined as the convex hull of 5 vertices
```

TESTS:

```
sage: X = polytopes.n_cube(2)
sage: Y = Polyhedron(vertices=[(1,1)])
sage: (X-Y).Vrepresentation()
(A vertex at (0, -2), A vertex at (0, 0), A vertex at (-2, 0), A vertex at (-2, -2))

sage: Y = Polyhedron(vertices=[(1,1), (0,0)])
sage: (X-Y).Vrepresentation()
(A vertex at (0, -1), A vertex at (0, 0), A vertex at (-1, 0), A vertex at (-1, -1))

sage: X = X + Y   # now Y is a Minkowski summand of X
sage: (X+Y)-Y == X
True
sage: (X-Y)+Y == X
True
```

Minkowski_sum(*other*)

Return the Minkowski sum.

Minkowski addition of two subsets of a vector space is defined as

$$X \oplus Y = \cup_{y \in Y}(X + y) = \cup_{x \in X, y \in Y}(x + y)$$

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.n_cube(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.n_cube(4)
sage: four_simplex = Polyhedron(vertices = [[0, 0, 0, 1], [0, 0, 1, 0], [0, 1, 0, 0], [1, 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_r
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
```

Vrep_generator()

Returns an iterator over the objects of the V-representation (vertices, rays, and lines).

EXAMPLES:

```
sage: p = polytopes.cyclic_polytope(3,4)
sage: vg = p.Vrep_generator()
sage: vg.next()
A vertex at (0, 0, 0)
sage: vg.next()
A vertex at (1, 1, 1)
```

Vrepresentation(*index=None*)

Return the objects of the V-representation. Each entry is either a vertex, a ray, or a line.

See sage.geometry.polyhedron.constructor for a definition of vertex/ray/line.

INPUT:

- index – either an integer or None.

OUTPUT:

The optional argument is an index running from 0 to $self.n_Vrepresentation() - 1$‘. If present, the V-representation object at the given index will be returned. Without an argument, returns the list of all V-representation objects.

EXAMPLES:

```
sage: p = polytopes.n_simplex(4)
sage: p.Vrepresentation(0)
A vertex at (-7071/10000, 1633/4000, 7217/25000, 22361/100000)
sage: p.Vrepresentation(0) == p.Vrepresentation() [0]
True
```

Vrepresentation_space()

Return the ambient vector space.

OUTPUT:

A free module over the base ring of dimension ambient_dim().

EXAMPLES:

```
sage: poly_test = Polyhedron(vertices = [[1,0,0,0],[0,1,0,0]])
sage: poly_test.Vrepresentation_space()
Ambient free module of rank 4 over the principal ideal domain Integer Ring
```

```
sage: poly_test.ambient_space() is poly_test.Vrepresentation_space()
True
```

adjacency_matrix()

Return the binary matrix of vertex adjacencies.

EXAMPLES:

```
sage: polytopes.n_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,)
```

Its (non-unique) V-representation consists of a vertex, a ray, and a line. The only edge is spanned by the vertex and the line generator, so they are adjacent:

```
sage: half_plane.Vrepresentation()
(A line in the direction (0, 1), A ray in the direction (1, 0), A vertex at (0, 0))
sage: half_plane.vertex_adjacency_matrix()
[0 0 1]
[0 0 0]
[1 0 0]
```

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:

```
sage: P = Polyhedron(vertices=[(0, 1), (1, 0), (3, 0), (4, 1)])
sage: for v in P.Vrep_generator():
...       print P.adjacency_matrix().row(v.index()), v
(0, 1, 0, 1) A vertex at (0, 1)
(1, 0, 1, 0) A vertex at (1, 0)
(0, 1, 0, 1) A vertex at (3, 0)
(1, 0, 1, 0) A vertex at (4, 1)
```

If the V-representation of the polygon contains vertices and one ray, then each V-representation object is adjacent to two V-representation objects:

```
sage: P = Polyhedron(vertices=[(0, 1), (1, 0), (3, 0), (4, 1)],
...                  rays=[(0,1)])
sage: for v in P.Vrep_generator():
...       print P.adjacency_matrix().row(v.index()), v
(0, 1, 0, 0, 1) A ray in the direction (0, 1)
(1, 0, 1, 0, 0) A vertex at (0, 1)
(0, 1, 0, 1, 0) A vertex at (1, 0)
(0, 0, 1, 0, 1) A vertex at (3, 0)
(1, 0, 0, 1, 0) A vertex at (4, 1)
```

If the V-representation of the polygon contains vertices and two distinct rays, then each vertex is adjacent to two V-representation objects (which can now be vertices or rays). The two rays are not adjacent to each other:

```
sage: P = Polyhedron(vertices=[(0, 1), (1, 0), (3, 0), (4, 1)],
...                  rays=[(0,1), (1,1)])
sage: for v in P.Vrep_generator():
...       print P.adjacency_matrix().row(v.index()), v
(0, 1, 0, 0, 0) A ray in the direction (0, 1)
(1, 0, 1, 0, 0) A vertex at (0, 1)
(0, 1, 0, 0, 1) A vertex at (1, 0)
(0, 0, 0, 0, 1) A ray in the direction (1, 1)
(0, 0, 1, 1, 0) A vertex at (3, 0)
```

affine_hull()

Return the affine hull.

Each polyhedron is contained in some smallest affine subspace (possibly the entire ambient space). The affine hull is the same polyhedron but thought of as a full-dimensional polyhedron in this subspace.

OUTPUT:

A full-dimensional polyhedron.

EXAMPLES:

```
sage: triangle = Polyhedron([(1,0,0), (0,1,0), (0,0,1)]);  triangle
A 2-dimensional polyhedron in ZZ^3 defined as the convex hull of 3 vertices
sage: triangle.affine_hull()
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 3 vertices

sage: half3d = Polyhedron(vertices=[(3,2,1)], rays=[(1,0,0)])
sage: half3d.affine_hull().Vrepresentation()
(A ray in the direction (1), A vertex at (3))
```

TESTS:

```
sage: Polyhedron([(2,3,4)]).affine_hull()
A 0-dimensional polyhedron in ZZ^0 defined as the convex hull of 1 vertex
```

ambient_dim()
Return the dimension of the ambient space.

EXAMPLES:

```
sage: poly_test = Polyhedron(vertices = [[1,0,0,0],[0,1,0,0]])
sage: poly_test.ambient_dim()
4
```

ambient_space()
Return the ambient vector space.

OUTPUT:

A free module over the base ring of dimension ambient_dim().

EXAMPLES:

```
sage: poly_test = Polyhedron(vertices = [[1,0,0,0],[0,1,0,0]])
sage: poly_test.Vrepresentation_space()
Ambient free module of rank 4 over the principal ideal domain Integer Ring
sage: poly_test.ambient_space() is poly_test.Vrepresentation_space()
True
```

base_extend(*base_ring*, *backend=None*)
Return a new polyhedron over a larger field.

INPUT:

- base_ring – the new base ring.
- backend – the new backend, see Polyhedron().

OUTPUT:

The same polyhedron, but over a larger base ring.

EXAMPLES:

```
sage: P = Polyhedron(vertices=[(1,0), (0,1)], rays=[(1,1)], base_ring=ZZ);  P
A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices and 1 ray
sage: P.base_extend(QQ)
A 2-dimensional polyhedron in QQ^2 defined as the convex hull of 2 vertices and 1 ray
sage: P.base_extend(QQ) == P
True
```

base_ring()
Return the base ring.

OUTPUT:

Either QQ (exact arithmetic using gmp, default) or RDF (double precision floating-point arithmetic)

EXAMPLES:

```
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:

```
sage: octahedron = polytopes.cross_polytope(3)
sage: cross_poly_4d = octahedron.bipyramid()
sage: cross_poly_4d.n_vertices()
8
sage: q = [list(v) for v in cross_poly_4d.vertex_generator()]
sage: q
[[-1, 0, 0, 0],
 [0, -1, 0, 0],
 [0, 0, -1, 0],
 [0, 0, 0, -1],
 [0, 0, 0, 1],
 [0, 0, 1, 0],
 [0, 1, 0, 0],
 [1, 0, 0, 0]]
```

Now check that bipyramids of cross-polytopes are cross-polytopes:

```
sage: q2 = [list(v) for v in polytopes.cross_polytope(4).vertex_generator()]
sage: [v in q2 for v in q]
[True, True, True, True, True, True, True, True]
```

bounded_edges()

Return the bounded edges (excluding rays and lines).

OUTPUT:

A generator for pairs of vertices, one pair per edge.

EXAMPLES:

```
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*)

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.

OUTPUT:

A pair of tuples `(box_min, box_max)` where `box_min` are the coordinates of a point bounding the coordinates of the polytope from below and `box_max` bounds the coordinates from above.

EXAMPLES:

```
sage: Polyhedron([ (1/3,2/3), (2/3, 1/3) ]).bounding_box()
((1/3, 1/3), (2/3, 2/3))
sage: Polyhedron([ (1/3,2/3), (2/3, 1/3) ]).bounding_box(integral=True)
((0, 0), (1, 1))
sage: polytopes.buckyball().bounding_box()
((-1059/1309, -1059/1309, -1059/1309), (1059/1309, 1059/1309, 1059/1309))
```

cdd_Hrepresentation()

Write the inequalities/equations data of the polyhedron in cdd's H-representation format.

OUTPUT:

A string. If you save the output to filename.ine then you can run the stand-alone cdd via `cddr+ filename.ine`

EXAMPLES:

```
sage: p = polytopes.n_cube(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
```

cdd_Vrepresentation()

Write the vertices/rays/lines data of the polyhedron in cdd's V-representation format.

OUTPUT:

A string. If you save the output to filename.ext then you can run the stand-alone cdd via `cddr+ filename.ext`

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 `interior_point()`.

OUTPUT:

The center of the polyhedron. All rays and lines are ignored. Raises a `ZeroDivisionError` for the empty polytope.

EXAMPLES:

```
sage: p = polytopes.n_cube(3)
sage: p = p + vector([1,0,0])
sage: p.center()
(1, 0, 0)
```

combinatorial_automorphism_group()

Computes the combinatorial automorphism group of the vertex graph of the polyhedron.

OUTPUT:

A `PermutationGroup` that is isomorphic to the combinatorial automorphism group is returned.

Note that in Sage, permutation groups always act on positive integers while `self.Vrepresentation()` is indexed by nonnegative integers. The indexing of the permuta-

tion group is chosen to be shifted by +1. That is, i in the permutation group corresponds to the V-representation object self.Vrepresentation(i-1).

EXAMPLES:

```
sage: quadrangle = Polyhedron(vertices=[(0,0),(1,0),(0,1),(2,3)])
sage: quadrangle.combinatorial_automorphism_group()
Permutation Group with generators [(2,3), (1,2)(3,4)]
sage: quadrangle.restricted_automorphism_group()
Permutation Group with generators [()]
```

Permutations can only exchange vertices with vertices, rays with rays, and lines with lines:

```
sage: P = Polyhedron(vertices=[(1,0,0), (1,1,0)], rays=[(1,0,0)], lines=[(0,0,1)])
sage: P.combinatorial_automorphism_group()
Permutation Group with generators [(3,4)]
```

contains (*point*)

Test whether the polyhedron contains the given point.

See also interior_contains() and 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 #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.n_simplex(3)
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
```

delete()

Delete this 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.

See Also:

recycle()

EXAMPLES:

```
sage: p = Polyhedron([(0,0),(1,0),(0,1)])
sage: p.delete()

sage: vertices = [(0,0,0,0),(1,0,0,0),(0,1,0,0),(1,1,0,0),(0,0,1,0),(0,0,0,1)]
sage: def loop_polyhedra():
...       for i in range(0,100):
...           p = Polyhedron(vertices)

sage: timeit('loop_polyhedra()')                    # not tested - random
5 loops, best of 3: 79.5 ms per loop

sage: def loop_polyhedra_with_recycling():
...       for i in range(0,100):
...           p = Polyhedron(vertices)
...           p.delete()

sage: timeit('loop_polyhedra_with_recycling()')     # not tested - random
5 loops, best of 3: 57.3 ms per loop
```

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 field.

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[t,t^2,t^3] for t in srange(2,6)])
sage: p.vertex_generator().next()
A vertex at (2, 4, 8)
sage: p2 = p.dilation(2)
sage: p2.vertex_generator().next()
A vertex at (4, 8, 16)
sage: p.dilation(2) == p * 2
True
```

TESTS:

Dilation of empty polyhedrons works, see trac ticket #14987:

```
sage: p = Polyhedron(ambient_dim=2); p
The empty polyhedron in ZZ^2
sage: p.dilation(3)
The empty polyhedron in ZZ^2
```

TESTS:

```
sage: p = Polyhedron(vertices=[(1,1)], rays=[(1,0)], lines=[(0,1)])
sage: (-p).rays()
(A ray in the direction (-1, 0),)
sage: (-p).lines()
(A line in the direction (0, 1),)

sage: (0*p).rays()
()
sage: (0*p).lines()
()
```

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 #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
```

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 #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
```

edge_truncation(*cut_frac=1/3*)

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.n_cube(3)
sage: trunc_cube = cube.edge_truncation()
sage: trunc_cube.n_vertices()
24
sage: trunc_cube.n_inequalities()
14
```

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: p3.equation_generator().next()
An equation (0.0, 0.0, 1.0) x + 0.0 == 0
```

equations()

Return all linear constraints of the polyhedron.

OUTPUT:

A tuple of equations.

EXAMPLES:

```
sage: test_p = Polyhedron(vertices = [[1,2,3,4],[2,1,3,4],[4,3,2,1],[3,4,1,2]])
sage: test_p.equations()
(An equation (1, 1, 1, 1) x - 10 == 0,)
```

equations_list()

Return the linear constraints of the polyhedron. As with inequalities, each constraint is given as [b -a1 -a2 ... an] where for variables x1, x2,..., xn, the polyhedron satisfies the equation b = a1*x1 + a2*x2 + ... + an*xn.

Note: It is recommended to use `equations()` or `equation_generator()` instead to iterate over the list of `Equation` objects.

EXAMPLES:

```
sage: test_p = Polyhedron(vertices = [[1,2,3,4],[2,1,3,4],[4,3,2,1],[3,4,1,2]])
sage: test_p.equations_list()
[[-10, 1, 1, 1, 1]]
```

f_vector()

Return the f-vector.

OUTPUT:

Returns a vector whose `i`-th entry is the number of `i`-dimensional faces of the polytope.

EXAMPLES:

```
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)
```

face_lattice()

Return the face-lattice poset.

OUTPUT:

A `FinitePoset`. Elements are given as `PolyhedronFace`.

In the case of a full-dimensional polytope, the faces are pairs (vertices, inequalities) of the spanning vertices and corresponding saturated inequalities. In general, a face is defined by a pair (V-rep. objects, H-rep. objects). The V-representation objects span the face, and the corresponding H-representation objects are those inequalities and equations that are saturated on the face.

The bottom-most element of the face lattice is the "empty face". It contains no V-representation object. All H-representation objects are incident.

The top-most element is the "full face". It is spanned by all V-representation objects. The incident H-representation objects are all equations and no inequalities.

In the case of a full-dimensional polytope, the "empty face" and the "full face" are the empty set (no vertices, all inequalities) and the full polytope (all vertices, no inequalities), respectively.

ALGORITHM:

For a full-dimensional polytope, the basic algorithm is described in `Hasse_diagram_from_incidences()`. There are three generalizations of [KP2002] necessary to deal with more general polytopes, corresponding to the extra H/V-representation objects:

- Lines are removed before calling `Hasse_diagram_from_incidences()`, and then added back to each face V-representation except for the "empty face".
- Equations are removed before calling `Hasse_diagram_from_incidences()`, and then added back to each face H-representation.
- Rays: Consider the half line as an example. The V-representation objects are a point and a ray, which we can think of as a point at infinity. However, the point at infinity has no inequality associated to it, so there is only one H-representation object alltogether. The face lattice does not contain the "face at infinity". This means that in `Hasse_diagram_from_incidences()`, one needs to drop faces with V-representations that have no matching H-representation. In addition, one needs to ensure that every non-empty face contains at least one vertex.

EXAMPLES:

```
sage: square = polytopes.n_cube(2)
sage: square.face_lattice()
Finite poset containing 10 elements
sage: list(_)
[<>, <0>, <1>, <2>, <3>, <0,1>, <0,2>, <2,3>, <1,3>, <0,1,2,3>]
sage: poset_element = _[6]
sage: a_face = poset_element
sage: a_face
<0,2>
sage: a_face.dim()
1
sage: set(a_face.ambient_Vrepresentation()) ==              ...   set([square.Vrepresentation
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]
```

TESTS:

```
sage: c5_20 = Polyhedron(vertices = [[i,i^2,i^3,i^4,i^5]
...       for i in range(1,21)])
sage: c5_20_fl = c5_20.face_lattice() # long time
sage: [len(x) for x in c5_20_fl.level_sets()] # long time
[1, 20, 190, 580, 680, 272, 1]
sage: polytopes.n_cube(2).face_lattice().plot()
sage: level_sets = polytopes.cross_polytope(2).face_lattice().level_sets()
sage: print level_sets[0], level_sets[-1]
```

```
[<>] [<0,1,2,3>]
```

Various degenerate polyhedra:

```
sage: Polyhedron(vertices=[[0,0,0],[1,0,0],[0,1,0]]).face_lattice().level_sets()
[[<>], [<0>, <1>, <2>], [<0,1>, <0,2>, <1,2>], [<0,1,2>]]
sage: Polyhedron(vertices=[(1,0,0),(0,1,0)], rays=[(0,0,1)]).face_lattice().level_sets()
[[<>], [<1>, <2>], [<0,1>, <0,2>, <1,2>], [<0,1,2>]]
sage: Polyhedron(rays=[(1,0,0),(0,1,0)], vertices=[(0,0,1)]).face_lattice().level_sets()
[[<>], [<0>], [<0,1>, <0,2>], [<0,1,2>]]
sage: Polyhedron(rays=[(1,0),(0,1)], vertices=[(0,0)]).face_lattice().level_sets()
[[<>], [<0>], [<0,1>, <0,2>], [<0,1,2>]]
sage: Polyhedron(vertices=[(1,),(0,)]).face_lattice().level_sets()
[[<>], [<0>, <1>], [<0,1>]]
sage: Polyhedron(vertices=[(1,0,0),(0,1,0)], lines=[(0,0,1)]).face_lattice().level_sets()
[[<>], [<0,1>, <0,2>], [<0,1,2>]]
sage: Polyhedron(lines=[(1,0,0)], vertices=[(0,0,1)]).face_lattice().level_sets()
[[<>], [<0,1>]]
sage: Polyhedron(lines=[(1,0),(0,1)], vertices=[(0,0)]).face_lattice().level_sets()
[[<>], [<0,1,2>]]
sage: Polyhedron(lines=[(1,0)], rays=[(0,1)], vertices=[(0,0)])            ...       .face_l
[[<>], [<0,1>], [<0,1,2>]]
sage: Polyhedron(vertices=[(0,)], lines=[(1,)]).face_lattice().level_sets()
[[<>], [<0,1>]]
sage: Polyhedron(lines=[(1,0)], vertices=[(0,0)]).face_lattice().level_sets()
[[<>], [<0,1>]]
```

REFERENCES:

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 random but fixed.

EXAMPLES:

Here we find the vertex and face indices of the eight three-dimensional facets of the four-dimensional hypercube:

```
sage: p = polytopes.n_cube(4)
sage: p.faces(3)
(<0,1,2,3,4,5,6,7>, <0,1,2,3,8,9,10,11>, <0,1,4,5,8,9,12,13>,
 <0,2,4,6,8,10,12,14>, <2,3,6,7,10,11,14,15>, <8,9,10,11,12,13,14,15>,
 <4,5,6,7,12,13,14,15>, <1,3,5,7,9,11,13,15>)

sage: face = p.faces(3)[0]
sage: face.ambient_Hrepresentation()
(An inequality (1, 0, 0, 0) x + 1 >= 0,)
sage: face.vertices()
(A vertex at (-1, -1, -1, -1), A vertex at (-1, -1, -1, 1),
 A vertex at (-1, -1, 1, -1), A vertex at (-1, -1, 1, 1),
 A vertex at (-1, 1, -1, -1), A vertex at (-1, 1, -1, 1),
 A vertex at (-1, 1, 1, -1), A vertex at (-1, 1, 1, 1))
```

You can use the `index()` method to enumerate vertices and inequalities:

```
sage: def get_idx(rep): return rep.index()
sage: map(get_idx, face.ambient_Hrepresentation())
[4]
sage: map(get_idx, face.ambient_Vrepresentation())
[0, 1, 2, 3, 4, 5, 6, 7]

sage: [ (map(get_idx, face.ambient_Vrepresentation()), map(get_idx, face.ambient_Hrepresenta
...     for face in p.faces(3) ]
[([0, 1, 2, 3, 4, 5, 6, 7], [4]),
 ([0, 1, 2, 3, 8, 9, 10, 11], [5]),
 ([0, 1, 4, 5, 8, 9, 12, 13], [6]),
 ([0, 2, 4, 6, 8, 10, 12, 14], [7]),
 ([2, 3, 6, 7, 10, 11, 14, 15], [2]),
 ([8, 9, 10, 11, 12, 13, 14, 15], [0]),
 ([4, 5, 6, 7, 12, 13, 14, 15], [1]),
 ([1, 3, 5, 7, 9, 11, 13, 15], [3])]
```

TESTS:

```
sage: pr = Polyhedron(rays = [[1,0,0],[-1,0,0],[0,1,0]], vertices = [[-1,-1,-1]], lines=[(0,
sage: pr.faces(4)
()
sage: pr.faces(3)
(<0,1,2,3>,)
sage: pr.faces(2)
(<0,1,2>,)
sage: pr.faces(1)
()
sage: pr.faces(0)
()
sage: pr.faces(-1)
()
```

facet_adjacency_matrix()

Return the adjacency matrix for the facets and hyperplanes.

EXAMPLES:

```
sage: polytopes.n_simplex(4).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]
```

facial_adjacencies()

Return the list of face indices (i.e. indices of H-representation objects) and the indices of faces adjacent to them.

Note: Instead of working with face indices, it is recommended that you use the H-representation objects directly (see example).

EXAMPLES:

```
sage: p = polytopes.permutahedron(4)
sage: p.facial_adjacencies()[0:3]
doctest:...: DeprecationWarning:
This method is deprecated.
```

```
Use self.Hrepresentation(i).neighbors() instead.
See http://trac.sagemath.org/11763 for details.
[[0, [1, 2, 5, 10, 12, 13]], [1, [0, 2, 5, 7, 9, 11]], [2, [0, 1, 10, 11]]]
sage: f0 = p.Hrepresentation(0)
sage: f0.index() == 0
True
sage: f0_adjacencies = [f0.index(), [n.index() for n in f0.neighbors()]]
sage: p.facial_adjacencies()[0] == f0_adjacencies
True
```

facial_incidences()

Return the face-vertex incidences in the form $[f_i, [v_{i_0}, v_{i_1}, \ldots, v_{i_2}]]$.

Note: Instead of working with face/vertex indices, it is recommended that you use the H-representation/V-representation objects directly (see examples). Or use `incidence_matrix()`.

OUTPUT:

The face indices are the indices of the H-representation objects, and the vertex indices are the indices of the V-representation objects.

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[5,0,0],[0,5,0],[5,5,0],[0,0,0],[2,2,5]])
sage: p.facial_incidences()
doctest:...: DeprecationWarning:
This method is deprecated. Use self.Hrepresentation(i).incident() instead.
See http://trac.sagemath.org/11763 for details.
[[0, [0, 1, 3, 4]],
 [1, [0, 1, 2]],
 [2, [0, 2, 3]],
 [3, [2, 3, 4]],
 [4, [1, 2, 4]]]

sage: f0 = p.Hrepresentation(0)
sage: f0.index() == 0
True
sage: f0_incidences = [f0.index(), [v.index() for v in f0.incident()]]
sage: p.facial_incidences()[0] == f0_incidences
True

sage: p.incidence_matrix().column(0)
(1, 1, 0, 1, 1)
sage: p.incidence_matrix().column(1)
(1, 1, 1, 0, 0)
sage: p.incidence_matrix().column(2)
(1, 0, 1, 1, 0)
sage: p.incidence_matrix().column(3)
(0, 0, 1, 1, 1)
sage: p.incidence_matrix().column(4)
(0, 1, 1, 0, 1)
```

field()

Return the base ring.

OUTPUT:

Either `QQ` (exact arithmetic using gmp, default) or `RDF` (double precision floating-point arithmetic)

EXAMPLES:

```
sage: triangle = Polyhedron(vertices = [[1,0],[0,1],[1,1]])
sage: triangle.base_ring() == ZZ
True
```

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

REFERENCES:

Lectures in Geometric Combinatorics, R.R.Thomas, 2006, AMS Press.

graph()

Return a graph in which the vertices correspond to vertices of the polyhedron, and edges to edges.

EXAMPLES:

```
sage: g3 = polytopes.n_cube(3).vertex_graph()
sage: len(g3.automorphism_group())
48
sage: s4 = polytopes.n_simplex(4).vertex_graph()
sage: s4.is_eulerian()
True
```

hyperplane_arrangement()

Return the hyperplane arrangement defined by the equations and inequalities.

OUTPUT:

A `hyperplane arrangement` consisting of the hyperplanes defined by the `Hrepresentation()`. If the polytope is full-dimensional, this is the hyperplane arrangement spanned by the facets of the polyhedron.

EXAMPLES:

```
sage: p = polytopes.n_cube(2)
sage: p.hyperplane_arrangement()
Arrangement <-t0 + 1 | -t1 + 1 | t1 + 1 | t0 + 1>
```

ieqs()

Deprecated. Alias for inequalities()

EXAMPLES:

```
sage: p3 = Polyhedron(vertices = Permutations([1,2,3,4]))
sage: p3.ieqs() == p3.inequalities()
doctest:...: DeprecationWarning:
This method is deprecated. Use inequalities() instead.
```

```
See http://trac.sagemath.org/11763 for details.
True
```

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()`

EXAMPLES:

```
sage: p = polytopes.cuboctahedron()
sage: p.incidence_matrix()
[0 0 1 1 0 1 0 0 0 0 1 0 0 0]
[0 0 0 1 0 0 1 0 1 0 1 0 0 0]
[0 0 1 1 1 0 0 1 0 0 0 0 0 0]
[1 0 0 1 1 0 1 0 0 0 0 0 0 0]
[0 0 0 0 0 1 0 0 1 1 1 0 0 0]
[0 0 1 0 0 1 0 1 0 0 0 1 0 0]
[1 0 0 0 0 0 1 0 1 0 0 0 1 0]
[1 0 0 0 1 0 0 1 0 0 0 0 0 1]
[0 1 0 0 0 1 0 0 0 1 0 1 0 0]
[0 1 0 0 0 0 0 0 1 1 0 0 1 0]
[0 1 0 0 0 0 0 1 0 0 0 1 0 1]
[1 1 0 0 0 0 0 0 0 0 0 0 1 1]
sage: v = p.Vrepresentation(0)
sage: v
A vertex at (-1/2, -1/2, 0)
sage: h = p.Hrepresentation(2)
sage: h
An inequality (1, 1, -1) x + 1 >= 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
```

inequalities()

Return all inequalities.

OUTPUT:

A tuple of inequalities.

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[0,0,0],[0,0,1],[0,1,0],[1,0,0],[2,2,2]])
sage: p.inequalities()[0:3]
(An inequality (1, 0, 0) x + 0 >= 0,
 An inequality (0, 1, 0) x + 0 >= 0,
 An inequality (0, 0, 1) x + 0 >= 0)
sage: p3 = Polyhedron(vertices = Permutations([1,2,3,4]))
sage: ieqs = p3.inequalities()
sage: ieqs[0]
```

```
An inequality (0, 1, 1, 1) x - 6 >= 0
sage: list(_)
[-6, 0, 1, 1, 1]
```

inequalities_list()
Return a list of inequalities as coefficient lists.

Note: It is recommended to use `inequalities()` or `inequality_generator()` instead to iterate over the list of `Inequality` objects.

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[0,0,0],[0,0,1],[0,1,0],[1,0,0],[2,2,2]])
sage: p.inequalities_list()[0:3]
[[0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]]
sage: p3 = Polyhedron(vertices = Permutations([1,2,3,4]))
sage: ieqs = p3.inequalities_list()
sage: ieqs[0]
[-6, 0, 1, 1, 1]
sage: ieqs[-1]
[-3, 0, 1, 0, 1]
sage: ieqs == [list(x) for x in p3.inequality_generator()]
True
```

inequality_generator()
Return a generator for the defining inequalities of the polyhedron.

OUTPUT:

A generator of the inequality Hrepresentation objects.

EXAMPLES:

```
sage: triangle = Polyhedron(vertices=[[1,0],[0,1],[1,1]])
sage: for v in triangle.inequality_generator(): print(v)
An inequality (1, 1) x - 1 >= 0
An inequality (0, -1) x + 1 >= 0
An inequality (-1, 0) x + 1 >= 0
sage: [ v for v in triangle.inequality_generator() ]
[An inequality (1, 1) x - 1 >= 0,
 An inequality (0, -1) x + 1 >= 0,
 An inequality (-1, 0) x + 1 >= 0]
sage: [ [v.A(), v.b()] for v in triangle.inequality_generator() ]
[[(1, 1), -1], [(0, -1), 1], [(-1, 0), 1]]
```

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 algorith as long as the bounding box is smaller than this.

OUTPUT:

The list of integral points in the polyhedron. If the polyhedron is not compact, a `ValueError` is raised.

EXAMPLES:

```
sage: Polyhedron(vertices=[(-1,-1),(1,0),(1,1),(0,1)]).integral_points()
((-1, -1), (0, 0), (0, 1), (1, 0), (1, 1))

sage: simplex = Polyhedron([(1,2,3), (2,3,7), (-2,-3,-11)])
sage: simplex.integral_points()
((-2, -3, -11), (0, 0, -2), (1, 2, 3), (2, 3, 7))
```

The polyhedron need not be full-dimensional:

```
sage: simplex = Polyhedron([(1,2,3,5), (2,3,7,5), (-2,-3,-11,5)])
sage: simplex.integral_points()
((-2, -3, -11, 5), (0, 0, -2, 5), (1, 2, 3, 5), (2, 3, 7, 5))

sage: point = Polyhedron([(2,3,7)])
sage: point.integral_points()
((2, 3, 7),)

sage: empty = Polyhedron()
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 = Polyhedron(v); simplex
A 4-dimensional polyhedron in ZZ^4 defined as the convex hull of 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 = 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_pc()         # PALP
sage: for p in pts1: p.set_immutable()
sage: set(pts1) == set(pts2)
True

sage: timeit('Polyhedron(v).integral_points()')   # not tested - random
625 loops, best of 3: 1.41 ms per loop
sage: timeit('LatticePolytope(v).points()')       # not tested - random
25 loops, best of 3: 17.2 ms per loop
```

interior_contains (*point*)

Test whether the interior of the polyhedron contains the given `point`.

See also `contains()` and `relative_interior_contains()`.

INPUT:

- `point` – coordinates of a point.

OUTPUT:

`True` or `False`.

EXAMPLES:

```
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: 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 #10238:

```
sage: empty = Polyhedron(); empty
The empty polyhedron in ZZ^0
sage: empty.interior_contains([])
False
```

intersection(*other*)

Return the intersection of one polyhedron with another.

INPUT:

- other – a Polyhedron.

OUTPUT:

The intersection.

Note that the intersection of two **Z**-polyhedra might not be a **Z**-polyhedron. In this case, a **Q**-polyhedron is returned.

EXAMPLES:

```
sage: cube = polytopes.n_cube(3)
sage: oct = polytopes.cross_polytope(3)
sage: cube.intersection(oct*2)
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 12 vertices
```

As a shorthand, one may use:

```
sage: cube & oct*2
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 12 vertices
```

The intersection of two **Z**-polyhedra is not necessarily a **Z**-polyhedron:

```
sage: P = Polyhedron([(0,0),(1,1)], base_ring=ZZ)
sage: P.intersection(P)
A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices
sage: Q = Polyhedron([(0,1),(1,0)], base_ring=ZZ)
sage: P.intersection(Q)
A 0-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex
sage: _.Vrepresentation()
(A vertex at (1/2, 1/2),)
```

is_Minkowski_summand(Y)

Test whether `Y` is a Minkowski summand.

See `Minkowski_sum()`.

OUTPUT:

Boolean. Whether there exists another polyhedron Z such that `self` can be written as $Y \oplus Z$.

EXAMPLES:

```
sage: A = polytopes.n_cube(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
```

is_compact()

Test for boundedness of the polytope.

EXAMPLES:

```
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:

```
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)
```

is_full_dimensional()
Return whether the polyhedron is full dimensional.

OUTPUT:

Boolean. Whether the polyhedron is not contained in any strict affine subspace.

EXAMPLES:

```
sage: polytopes.n_cube(3).is_full_dimensional()
True
sage: Polyhedron(vertices=[(1,2,3)], rays=[(1,0,0)]).is_full_dimensional()
False
```

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_simple()
Test for simplicity of a polytope.

See Wikipedia article Simple_polytope

EXAMPLES:

```
sage: p = Polyhedron([[0,0,0],[1,0,0],[0,1,0],[0,0,1]])
sage: p.is_simple()
True
sage: p = Polyhedron([[0,0,0],[4,4,0],[4,0,0],[0,4,0],[2,2,2]])
sage: p.is_simple()
False
```

is_simplex()
Return whether the polyhedron is a simplex.

EXAMPLES:

```
sage: Polyhedron([(0,0,0), (1,0,0), (0,1,0)]).is_simplex()
True
sage: polytopes.n_simplex(3).is_simplex()
True
sage: polytopes.n_cube(3).is_simplex()
False
```

is_simplicial()
Tests if the polytope is simplicial

A polytope is simplicial if every facet is a simplex.

See Wikipedia article Simplicial_polytope

EXAMPLES:

```
sage: p = polytopes.n_cube(3)
sage: p.is_simplicial()
```

```
False
sage: q = polytopes.n_simplex(5)
sage: q.is_simplicial()
True
sage: p = Polyhedron([[0,0,0],[1,0,0],[0,1,0],[0,0,1]])
sage: p.is_simplicial()
True
sage: q = Polyhedron([[1,1,1],[-1,1,1],[1,-1,1],[-1,-1,1],[1,1,-1]])
sage: q.is_simplicial()
False
```

The method is not implemented for unbounded polyhedra:

```
sage: 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)
```

lattice_polytope(*envelope=False*)

Return an encompassing lattice polytope.

INPUT:

- `envelope` – boolean (default: `False`). If the polyhedron has non-integral vertices, this option decides whether to return a strictly larger lattice polytope or raise a `ValueError`. This option has no effect if the polyhedron has already integral vertices.

OUTPUT:

A `LatticePolytope`. If the polyhedron is compact and has integral vertices, the lattice polytope equals the polyhedron. If the polyhedron is compact but has at least one non-integral vertex, a strictly larger lattice polytope is returned.

If the polyhedron is not compact, a `NotImplementedError` is raised.

If the polyhedron is not integral and `envelope=False`, a `ValueError` is raised.

ALGORITHM:

For each non-integral vertex, a bounding box of integral points is added and the convex hull of these integral points is returned.

EXAMPLES:

First, a polyhedron with integral vertices:

```
sage: P = Polyhedron( vertices = [(1, 0), (0, 1), (-1, 0), (0, -1)])
sage: lp = P.lattice_polytope(); lp
2-d reflexive polytope #3 in 2-d lattice M
sage: lp.vertices_pc()
M(-1,  0),
M( 0, -1),
M( 0,  1),
M( 1,  0)
in 2-d lattice M
```

Here is a polyhedron with non-integral vertices:

```
sage: P = Polyhedron( vertices = [(1/2, 1/2), (0, 1), (-1, 0), (0, -1)])
sage: lp = P.lattice_polytope()
Traceback (most recent call last):
...
ValueError: Some vertices are not integral. You probably want
to add the argument "envelope=True" to compute an enveloping
lattice polytope.
sage: lp = P.lattice_polytope(True); lp
2-d reflexive polytope #5 in 2-d lattice M
sage: lp.vertices_pc()
M(-1,  0),
M( 0, -1),
M( 0,  1),
M( 1,  0),
M( 1,  1)
in 2-d lattice M
```

line_generator()

Return a generator for the lines of the polyhedron.

EXAMPLES:

```
sage: pr = Polyhedron(rays = [[1,0],[-1,0],[0,1]], vertices = [[-1,-1]])
sage: pr.line_generator().next().vector()
(1, 0)
```

linearities()

Deprecated. Use equations() instead. Returns the linear constraints of the polyhedron. As with inequalities, each constraint is given as [b -a1 -a2 ... an] where for variables x1, x2,..., xn, the polyhedron satisfies the equation b = a1*x1 + a2*x2 + ... + an*xn.

EXAMPLES:

```
sage: test_p = Polyhedron(vertices = [[1,2,3,4],[2,1,3,4],[4,3,2,1],[3,4,1,2]])
sage: test_p.linearities()
doctest:...: DeprecationWarning:
This method is deprecated. Use equations_list() instead.
See http://trac.sagemath.org/11763 for details.
[[-10, 1, 1, 1, 1]]
sage: test_p.linearities() == test_p.equations_list()
True
```

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
```

lrs_volume(*verbose=False*)
Computes the volume of a polytope using lrs.

OUTPUT:

The volume, cast to RDF (although lrs seems to output a rational value this must be an approximation in some cases).

EXAMPLES:

```
sage: polytopes.n_cube(3).lrs_volume() #optional - lrs
doctest:...: DeprecationWarning: use volume(engine='lrs') instead
See http://trac.sagemath.org/13249 for details.
8.0
sage: (polytopes.n_cube(3)*2).lrs_volume() #optional - lrs
64.0
sage: polytopes.twenty_four_cell().lrs_volume() #optional - lrs
2.0
```

REFERENCES:

David Avis's lrs program.

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.n_simplex(4)
sage: p.n_Vrepresentation()
5
sage: p.n_Vrepresentation() == p.n_vertices() + p.n_rays() + p.n_lines()
True
```

n_equations()

Return the number of equations. The representation will always be minimal, so the number of equations is the codimension of the polyhedron in the ambient space.

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[1,0,0],[0,1,0],[0,0,1]])
sage: p.n_equations()
1
```

n_facets()

Return the number of inequalities. The representation will always be minimal, so the number of inequalities is the number of facets of the polyhedron in the ambient space.

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[1,0,0],[0,1,0],[0,0,1]])
sage: p.n_inequalities()
3

sage: p = Polyhedron(vertices = [[t,t^2,t^3] for t in range(6)])
sage: p.n_facets()
8
```

n_inequalities()

Return the number of inequalities. The representation will always be minimal, so the number of inequalities is the number of facets of the polyhedron in the ambient space.

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[1,0,0],[0,1,0],[0,0,1]])
sage: p.n_inequalities()
3

sage: p = Polyhedron(vertices = [[t,t^2,t^3] for t in range(6)])
sage: p.n_facets()
8
```

n_lines()

Return the number of lines. The representation will always be minimal.

EXAMPLES:

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

n_rays()
Return the number of rays. The representation will always be minimal.

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[1,0],[0,1]], rays=[[1,1]])
sage: p.n_rays()
1
```

n_vertices()
Return the number of vertices. The representation will always be minimal.

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[1,0],[0,1],[1,1]], rays=[[1,1]])
sage: p.n_vertices()
2
```

plot(*point=None*, *line=None*, *polygon=None*, *wireframe='blue'*, *fill='green'*, *projection_direction=None*, ***kwds*)
Return a graphical representation.

INPUT:

- `point`, `line`, `polygon` – Parameters to pass to point (0d), line (1d), and polygon (2d) plot commands. Allowed values are:
  - A Python dictionary to be passed as keywords to the plot commands.
  - A string or triple of numbers: The color. This is equivalent to passing the dictionary `{'color':...}`.
  - `False`: Switches off the drawing of the corresponding graphics object
- `wireframe`, `fill` – Similar to `point`, `line`, and `polygon`, but `fill` is used for the graphics objects in the dimension of the polytope (or of dimension 2 for higher dimensional polytopes) and `wireframe` is used for all lower-dimensional graphics objects (default: 'green' for `fill` and 'blue' for `wireframe`)
- `projection_direction` – coordinate list/tuple/iterable or `None` (default). The direction to use for the `schlegel_projection`() of the polytope. If not specified, no projection is used in dimensions < 4 and parallel projection is used in dimension 4.
- `**kwds` – optional keyword parameters that are passed to all graphics objects.

OUTPUT:

A (multipart) graphics object.

EXAMPLES:

```
sage: square = polytopes.n_cube(2)
sage: point = Polyhedron([[1,1]])
sage: line = Polyhedron([[1,1],[2,1]])
sage: cube = polytopes.n_cube(3)
sage: hypercube = polytopes.n_cube(4)
```

By default, the wireframe is rendered in blue and the fill in green:

```
sage: square.plot()
sage: point.plot()
sage: line.plot()
sage: cube.plot()
sage: hypercube.plot()
```

Draw the lines in red and nothing else:

```
sage: square.plot(point=False, line='red', polygon=False)
sage: point.plot(point=False, line='red', polygon=False)
sage: line.plot(point=False, line='red', polygon=False)
sage: cube.plot(point=False, line='red', polygon=False)
sage: hypercube.plot(point=False, line='red', polygon=False)
```

Draw points in red, no lines, and a blue polygon:

```
sage: square.plot(point={'color':'red'}, line=False, polygon=(0,0,1))
sage: point.plot(point={'color':'red'}, line=False, polygon=(0,0,1))
sage: line.plot(point={'color':'red'}, line=False, polygon=(0,0,1))
sage: cube.plot(point={'color':'red'}, line=False, polygon=(0,0,1))
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')
sage: point.plot(fill='green', wireframe='red')
sage: line.plot(fill='green', wireframe='red')
sage: cube.plot(fill='green', wireframe='red')
sage: hypercube.plot(fill='green', wireframe='red')
```

TESTS:

```
sage: for p in square.plot():
...       print p.options()['rgbcolor'], p
blue Point set defined by 4 point(s)
blue Line defined by 2 points
blue Line defined by 2 points
blue Line defined by 2 points
blue Line defined by 2 points
green Polygon defined by 4 points

sage: for p in line.plot():
...       print p.options()['rgbcolor'], p
blue Point set defined by 2 point(s)
green Line defined by 2 points

sage: for p in point.plot():
...       print p.options()['rgbcolor'], p
green Point set defined by 1 point(s)
```

Draw the lines in red and nothing else:

```
sage: for p in square.plot(point=False, line='red', polygon=False):
...       print p.options()['rgbcolor'], p
red Line defined by 2 points
red Line defined by 2 points
red Line defined by 2 points
red Line defined by 2 points
```

Draw vertices in red, no lines, and a blue polygon:

```
sage: for p in square.plot(point={'color':'red'}, line=False, polygon=(0,0,1)):
...       print p.options()['rgbcolor'], p
red Point set defined by 4 point(s)
(0, 0, 1) Polygon defined by 4 points

sage: for p in line.plot(point={'color':'red'}, line=False, polygon=(0,0,1)):
...       print p.options()['rgbcolor'], p
red Point set defined by 2 point(s)

sage: for p in point.plot(point={'color':'red'}, line=False, polygon=(0,0,1)):
...       print p.options()['rgbcolor'], p
red Point set defined by 1 point(s)
```

Draw in red without wireframe:

```
sage: for p in square.plot(wireframe=False, fill="red"):
...       print p.options()['rgbcolor'], p
red Polygon defined by 4 points

sage: for p in line.plot(wireframe=False, fill="red"):
...       print p.options()['rgbcolor'], p
red Line defined by 2 points

sage: for p in point.plot(wireframe=False, fill="red"):
...       print p.options()['rgbcolor'], p
red Point set defined by 1 point(s)
```

The `projection_direction` option:

```
sage: line3d = Polyhedron([(-1,-1,-1), (1,1,1)])
sage: print(line3d.plot(projection_direction=[2,3,4]).description())
Line defined by 2 points:           [(-0.00..., 0.126...), (0.131..., -1.93...)]
Point set defined by 2 point(s):    [(-0.00..., 0.126...), (0.131..., -1.93...)]
```

We try to draw the polytope in 2 or 3 dimensions:

```
sage: type(Polyhedron(ieqs=[(1,)]).plot())
<class 'sage.plot.graphics.Graphics'>
sage: type(polytopes.n_cube(1).plot())
<class 'sage.plot.graphics.Graphics'>
sage: type(polytopes.n_cube(2).plot())
<class 'sage.plot.graphics.Graphics'>
sage: type(polytopes.n_cube(3).plot())
<class 'sage.plot.plot3d.base.Graphics3dGroup'>
```

In 4d a projection to 3d is used:

```
sage: type(polytopes.n_cube(4).plot())
<class 'sage.plot.plot3d.base.Graphics3dGroup'>
sage: type(polytopes.n_cube(5).plot())
Traceback (most recent call last):
...
NotImplementedError: plotting of 5-dimensional polyhedra not implemented
```

If the polyhedron is not full-dimensional, the `affine_hull()` is used if necessary:

```
sage: type(Polyhedron([(0,), (1,)]).plot())
<class 'sage.plot.graphics.Graphics'>
sage: type(Polyhedron([(0,0), (1,1)]).plot())
<class 'sage.plot.graphics.Graphics'>
```

```
sage: type(Polyhedron([(0,0,0), (1,1,1)]).plot())
<class 'sage.plot.plot3d.base.Graphics3dGroup'>
sage: type(Polyhedron([(0,0,0,0), (1,1,1,1)]).plot())
<class 'sage.plot.plot3d.base.Graphics3dGroup'>
sage: type(Polyhedron([(0,0,0,0,0), (1,1,1,1,1)]).plot())
<class 'sage.plot.graphics.Graphics'>
```

polar()

Return the polar (dual) polytope.

The original vertices are translated so that their barycenter is at the origin, and then the vertices are used as the coefficients in the polar inequalities.

EXAMPLES:

```
sage: p = Polyhedron(vertices = [[0,0,1],[0,1,0],[1,0,0],[0,0,0],[1,1,1]], base_ring=QQ)
sage: p
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 5 vertices
sage: p.polar()
A 3-dimensional polyhedron in QQ^3 defined as the convex hull of 6 vertices

sage: cube = polytopes.n_cube(3)
sage: octahedron = polytopes.cross_polytope(3)
sage: cube_dual = cube.polar()
sage: octahedron == cube_dual
True
```

prism()

Return a prism of the original polyhedron.

EXAMPLES:

```
sage: square = polytopes.n_cube(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
```

projection()

Return a projection object.

See also schlegel_projection() for a more interesting projection.

OUTPUT:

The identity projection. This is useful for plotting polyhedra.

EXAMPLES:

```
sage: p = polytopes.n_cube(3)
sage: proj = p.projection()
sage: proj
The projection of a polyhedron into 3 dimensions
```

pyramid()

Returns a polyhedron that is a pyramid over the original.

EXAMPLES:

```
sage: square = polytopes.n_cube(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 ZZ^3 defined as the convex hull of 5 vertices
sage: egyptian_pyramid.n_vertices()
5
sage: for v in egyptian_pyramid.vertex_generator(): print v
A vertex at (0, -1, -1)
A vertex at (0, -1, 1)
A vertex at (0, 1, -1)
A vertex at (0, 1, 1)
A vertex at (1, 0, 0)
```

radius()

Return the maximal distance from the center to a vertex. All rays and lines are ignored.

OUTPUT:

The radius for a rational polyhedron is, in general, not rational. use radius_square() if you need a rational distance measure.

EXAMPLES:

```
sage: p = polytopes.n_cube(4)
sage: p.radius()
2
```

radius_square()

Return the square of the maximal distance from the center() to a vertex. All rays and lines are ignored.

OUTPUT:

The square of the radius, which is in `field()`.

EXAMPLES:

```
sage: p = polytopes.permutahedron(4, project = False)
sage: p.radius_square()
5
```

ray_generator()

Return a generator for the rays of the polyhedron.

EXAMPLES:

```
sage: pi = Polyhedron(ieqs = [[1,1,0],[1,0,1]])
sage: pir = pi.ray_generator()
sage: [x.vector() for x in pir]
[(1, 0), (0, 1)]
```

rays()

Return a list of rays of the polyhedron.

OUTPUT:

A tuple of rays.

EXAMPLES:

```
sage: p = Polyhedron(ieqs = [[0,0,0,1],[0,0,1,0],[1,1,0,0]])
sage: p.rays()
(A ray in the direction (1, 0, 0),
 A ray in the direction (0, 1, 0),
 A ray in the direction (0, 0, 1))
```

rays_list()

Return a list of rays as coefficient lists.

Note: It is recommended to use `rays()` or `ray_generator()` instead to iterate over the list of `Ray` objects.

OUTPUT:

A list of rays as lists of coordinates.

EXAMPLES:

```
sage: p = Polyhedron(ieqs = [[0,0,0,1],[0,0,1,0],[1,1,0,0]])
sage: p.rays_list()
[[1, 0, 0], [0, 1, 0], [0, 0, 1]]
sage: p.rays_list() == [list(r) for r in p.ray_generator()]
True
```

relative_interior_contains(*point*)

Test whether the relative interior of the polyhedron contains the given `point`.

See also `contains()` and `interior_contains()`.

INPUT:

- `point` – coordinates of a point.

OUTPUT:

`True` or `False`.

EXAMPLES:

```
sage: P = Polyhedron(vertices=[(1,0), (-1,0)])
sage: P.contains( (0,0) )
True
sage: P.interior_contains( (0,0) )
False
sage: P.relative_interior_contains( (0,0) )
True
sage: P.relative_interior_contains( (1,0) )
False
```

The empty polyhedron needs extra care, see trac #10238:

```
sage: empty = Polyhedron(); empty
The empty polyhedron in ZZ^0
sage: empty.relative_interior_contains([])
False
```

render_solid(***kwds*)

Return a solid rendering of a 2- or 3-d polytope.

EXAMPLES:

```
sage: p = polytopes.n_cube(3)
sage: p_solid = p.render_solid(opacity = .7)
sage: type(p_solid)
<class 'sage.plot.plot3d.base.Graphics3dGroup'>
```

render_wireframe(***kwds*)

For polytopes in 2 or 3 dimensions, return the edges as a list of lines.

EXAMPLES:

```
sage: p = Polyhedron([[1,2,],[1,1],[0,0]])
sage: p_wireframe = p.render_wireframe()
sage: p_wireframe._objects
[Line defined by 2 points, Line defined by 2 points, Line defined by 2 points]
```

representative_point()

Return a "generic" point.

See also `center()`.

OUTPUT:

A point as a coordinate vector. The point is chosen to be interior as far as possible. If the polyhedron is not full-dimensional, the point is in the relative interior. If the polyhedron is zero-dimensional, its single point is returned.

EXAMPLES:

```
sage: p = Polyhedron(vertices=[(3,2)], rays=[(1,-1)])
sage: p.representative_point()
(4, 1)
sage: p.center()
(3, 2)

sage: Polyhedron(vertices=[(3,2)]).representative_point()
(3, 2)
```

restricted_automorphism_group()

Return the restricted automorphism group.

First, let the linear automorphism group be the subgroup of the Euclidean group $E(d) = GL(d, \mathbf{R}) \ltimes \mathbf{R}^d$ preserving the d-dimensional polyhedron. The Euclidean group acts in the usual way $\vec{x} \mapsto A\vec{x} + b$ on the ambient space.

The restricted automorphism group is the subgroup of the linear automorphism group generated by permutations of 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

$$Q = \left\{ (x,y) \middle| x \geq 0,\ y \geq 0 \right\} \subset \mathbf{Q}^2$$

Then the linear automorphism group is

$$\mathrm{Aut}(Q) = \left\{ \begin{pmatrix} a & 0 \\ 0 & b \end{pmatrix}, \begin{pmatrix} 0 & c \\ d & 0 \end{pmatrix} :\ a,b,c,d \in \mathbf{Q}_{>0} \right\} \subset GL(2,\mathbf{Q}) \subset E(d)$$

Note that there are no translations that map the quadrant Q to itself, so the linear automorphism group is contained in the subgroup of rotations of the whole Euclidean group. The restricted automorphism group is

$$\mathrm{Aut}(Q) = \left\{ \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \right\} \simeq \mathbf{Z}_2$$

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 `self.Vrepresentation()` is indexed by nonnegative integers. The indexing of the permutation group is chosen to be shifted by `+1`. That is, `i` in the permutation group corresponds to the V-representation object `self.Vrepresentation(i-1)`.

REFERENCES:

EXAMPLES:

```
sage: P = polytopes.cross_polytope(3)
sage: AutP = P.restricted_automorphism_group();  AutP
Permutation Group with generators [(3,4), (2,3)(4,5), (2,5), (1,2)(5,6), (1,6)]
sage: P24 = polytopes.twenty_four_cell()
sage: AutP24 = P24.restricted_automorphism_group()
sage: PermutationGroup([
...       '(3,6)(4,7)(10,11)(14,15)(18,21)(19,22)',
...       '(2,3)(7,8)(11,12)(13,14)(17,18)(22,23)',
...       '(2,5)(3,10)(6,11)(8,17)(9,13)(12,16)(14,19)(15,22)(20,23)',
...       '(2,10)(3,5)(6,12)(7,18)(9,14)(11,16)(13,19)(15,23)(20,22)',
...       '(2,11)(3,12)(4,21)(5,6)(9,15)(10,16)(13,22)(14,23)(19,20)',
...       '(1,2)(3,4)(6,7)(8,9)(12,13)(16,17)(18,19)(21,22)(23,24)',
...       '(1,24)(2,13)(3,14)(5,9)(6,15)(10,19)(11,22)(12,23)(16,20)'
...     ]) == AutP24
True
```

Here is the quadrant example mentioned in the beginning:

```
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()
Permutation Group with generators [(2,3)]
```

Also, the polyhedron need not be full-dimensional:

```
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)]
```

Translations do not change the restricted automorphism group. For example, any non-degenerate triangle has the dihedral group with 6 elements, D_6, as its automorphism group:

```
sage: initial_points = [vector([1,0]), vector([0,1]), vector([-2,-1])]
sage: points = initial_points
sage: Polyhedron(vertices=points).restricted_automorphism_group()
Permutation Group with generators [(2,3), (1,2)]
sage: points = [pt - initial_points[0] for pt in initial_points]
sage: Polyhedron(vertices=points).restricted_automorphism_group()
Permutation Group with generators [(2,3), (1,2)]
sage: points = [pt - initial_points[1] for pt in initial_points]
sage: Polyhedron(vertices=points).restricted_automorphism_group()
Permutation Group with generators [(2,3), (1,2)]
sage: points = [pt - 2*initial_points[1] for pt in initial_points]
sage: Polyhedron(vertices=points).restricted_automorphism_group()
Permutation Group with generators [(2,3), (1,2)]
```

Floating-point computations are supported with a simple fuzzy zero implementation:

```
sage: P = Polyhedron(vertices=[(1.0/3.0,0,0),(0,1.0/3.0,0),(0,0,1.0/3.0)], base_ring=RDF)
sage: P.restricted_automorphism_group()
Permutation Group with generators [(2,3), (1,2)]
```

TESTS:

```
sage: p = Polyhedron(vertices=[(1,0), (1,1)], rays=[(1,0)])
sage: p.restricted_automorphism_group()
Permutation Group with generators [(2,3)]
```

schlegel_projection(*projection_dir=None*, *height=1.1*)

Return the Schlegel projection.

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

INPUT:

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

OUTPUT:

A `Projection` object.

EXAMPLES:

```
sage: p = polytopes.n_cube(3)
sage: sch_proj = p.schlegel_projection()
sage: schlegel_edge_indices = sch_proj.lines
sage: schlegel_edges = [sch_proj.coordinates_of(x) for x in schlegel_edge_indices]
sage: len([x for x in schlegel_edges if x[0][0] > 0])
4
```

show(*point=None*, *line=None*, *polygon=None*, *wireframe='blue'*, *fill='green'*, *projection_direction=None*, ***kwds*)

Return a graphical representation.

INPUT:

- `point`, `line`, `polygon` – Parameters to pass to point (0d), line (1d), and polygon (2d) plot commands. Allowed values are:
  - A Python dictionary to be passed as keywords to the plot commands.
  - A string or triple of numbers: The color. This is equivalent to passing the dictionary `{'color':...}`.
  - `False`: Switches off the drawing of the corresponding graphics object
- `wireframe`, `fill` – Similar to `point`, `line`, and `polygon`, but `fill` is used for the graphics objects in the dimension of the polytope (or of dimension 2 for higher dimensional polytopes) and `wireframe` is used for all lower-dimensional graphics objects (default: 'green' for `fill` and 'blue' for `wireframe`)
- `projection_direction` – coordinate list/tuple/iterable or `None` (default). The direction to use for the `schlegel_projection()` of the polytope. If not specified, no projection is used in dimensions < 4 and parallel projection is used in dimension 4.
- `**kwds` – optional keyword parameters that are passed to all graphics objects.

OUTPUT:

A (multipart) graphics object.

EXAMPLES:

```
sage: square = polytopes.n_cube(2)
sage: point = Polyhedron([[1,1]])
sage: line = Polyhedron([[1,1],[2,1]])
sage: cube = polytopes.n_cube(3)
sage: hypercube = polytopes.n_cube(4)
```

By default, the wireframe is rendered in blue and the fill in green:

```
sage: square.plot()
sage: point.plot()
sage: line.plot()
sage: cube.plot()
sage: hypercube.plot()
```

Draw the lines in red and nothing else:

```
sage: square.plot(point=False, line='red', polygon=False)
sage: point.plot(point=False, line='red', polygon=False)
sage: line.plot(point=False, line='red', polygon=False)
sage: cube.plot(point=False, line='red', polygon=False)
sage: hypercube.plot(point=False, line='red', polygon=False)
```

Draw points in red, no lines, and a blue polygon:

```
sage: square.plot(point={'color':'red'}, line=False, polygon=(0,0,1))
sage: point.plot(point={'color':'red'}, line=False, polygon=(0,0,1))
sage: line.plot(point={'color':'red'}, line=False, polygon=(0,0,1))
sage: cube.plot(point={'color':'red'}, line=False, polygon=(0,0,1))
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')
sage: point.plot(fill='green', wireframe='red')
sage: line.plot(fill='green', wireframe='red')
sage: cube.plot(fill='green', wireframe='red')
sage: hypercube.plot(fill='green', wireframe='red')
```

TESTS:

```
sage: for p in square.plot():
...       print p.options()['rgbcolor'], p
blue Point set defined by 4 point(s)
blue Line defined by 2 points
blue Line defined by 2 points
blue Line defined by 2 points
blue Line defined by 2 points
green Polygon defined by 4 points

sage: for p in line.plot():
...       print p.options()['rgbcolor'], p
blue Point set defined by 2 point(s)
green Line defined by 2 points

sage: for p in point.plot():
...       print p.options()['rgbcolor'], p
green Point set defined by 1 point(s)
```

Draw the lines in red and nothing else:

```
sage: for p in square.plot(point=False, line='red', polygon=False):
...       print p.options()['rgbcolor'], p
red Line defined by 2 points
red Line defined by 2 points
red Line defined by 2 points
red Line defined by 2 points
```

Draw vertices in red, no lines, and a blue polygon:

```
sage: for p in square.plot(point={'color':'red'}, line=False, polygon=(0,0,1)):
...       print p.options()['rgbcolor'], p
red Point set defined by 4 point(s)
(0, 0, 1) Polygon defined by 4 points

sage: for p in line.plot(point={'color':'red'}, line=False, polygon=(0,0,1)):
...       print p.options()['rgbcolor'], p
red Point set defined by 2 point(s)

sage: for p in point.plot(point={'color':'red'}, line=False, polygon=(0,0,1)):
...       print p.options()['rgbcolor'], p
red Point set defined by 1 point(s)
```

Draw in red without wireframe:

```
sage: for p in square.plot(wireframe=False, fill="red"):
...       print p.options()['rgbcolor'], p
red Polygon defined by 4 points

sage: for p in line.plot(wireframe=False, fill="red"):
...       print p.options()['rgbcolor'], p
red Line defined by 2 points
```

```
sage: for p in point.plot(wireframe=False, fill="red"):
...       print p.options()['rgbcolor'], p
red Point set defined by 1 point(s)
```

The `projection_direction` option:

```
sage: line3d = Polyhedron([(-1,-1,-1), (1,1,1)])
sage: print(line3d.plot(projection_direction=[2,3,4]).description())
Line defined by 2 points:           [(-0.00..., 0.126...), (0.131..., -1.93...)]
Point set defined by 2 point(s):    [(-0.00..., 0.126...), (0.131..., -1.93...)]
```

We try to draw the polytope in 2 or 3 dimensions:

```
sage: type(Polyhedron(ieqs=[(1,)]).plot())
<class 'sage.plot.graphics.Graphics'>
sage: type(polytopes.n_cube(1).plot())
<class 'sage.plot.graphics.Graphics'>
sage: type(polytopes.n_cube(2).plot())
<class 'sage.plot.graphics.Graphics'>
sage: type(polytopes.n_cube(3).plot())
<class 'sage.plot.plot3d.base.Graphics3dGroup'>
```

In 4d a projection to 3d is used:

```
sage: type(polytopes.n_cube(4).plot())
<class 'sage.plot.plot3d.base.Graphics3dGroup'>
sage: type(polytopes.n_cube(5).plot())
Traceback (most recent call last):
...
NotImplementedError: plotting of 5-dimensional polyhedra not implemented
```

If the polyhedron is not full-dimensional, the `affine_hull()` is used if necessary:

```
sage: type(Polyhedron([(0,), (1,)]).plot())
<class 'sage.plot.graphics.Graphics'>
sage: type(Polyhedron([(0,0), (1,1)]).plot())
<class 'sage.plot.graphics.Graphics'>
sage: type(Polyhedron([(0,0,0), (1,1,1)]).plot())
<class 'sage.plot.plot3d.base.Graphics3dGroup'>
sage: type(Polyhedron([(0,0,0,0), (1,1,1,1)]).plot())
<class 'sage.plot.plot3d.base.Graphics3dGroup'>
sage: type(Polyhedron([(0,0,0,0,0), (1,1,1,1,1)]).plot())
<class 'sage.plot.graphics.Graphics'>
```

simplicial_complex()

Return a simplicial complex from a triangulation of the polytope.

Warning: This first triangulates the polytope using `triangulated_facial_incidences`, and this function may fail in dimensions greater than 3, although it usually doesn't.

OUTPUT:

A simplicial complex.

EXAMPLES:

```
sage: p = polytopes.cuboctahedron()
sage: sc = p.simplicial_complex()
doctest:...: DeprecationWarning:
This method is deprecated. Use triangulate().simplicial_complex() instead.
See http://trac.sagemath.org/11634 for details.
```

```
doctest:...: DeprecationWarning:
This method is deprecated. Use triangulate() instead.
See http://trac.sagemath.org/11634 for details.
sage: sc
Simplicial complex with 12 vertices and 20 facets
```

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*)

Returns a triangulation of the polytope.

INPUT:

- `engine` – either 'auto' (default), 'internal', or 'TOPCOM'. The latter two instruct this package to always use its own triangulation algorithms or TOPCOM's algorithms, respectively. By default ('auto'), TOPCOM is used if it is available and internal routines otherwise.

The remaining keyword parameters are passed through to the `PointConfiguration` constructor:

- `connected` – boolean (default: `True`). Whether the triangulations should be connected to the regular triangulations via bistellar flips. These are much easier to compute than all triangulations.
- `fine` – boolean (default: `False`). Whether the triangulations must be fine, that is, make use of all points of the configuration.
- `regular` – boolean or `None` (default: `None`). Whether the triangulations must be regular. A regular triangulation is one that is induced by a piecewise-linear convex support function. In other words, the shadows of the faces of a polyhedron in one higher dimension.
  - `True`: Only regular triangulations.
  - `False`: Only non-regular triangulations.
  - `None` (default): Both kinds of triangulation.
- `star` – either `None` (default) or a point. Whether the triangulations must be star. A triangulation is star if all maximal simplices contain a common point. The central point can be specified by its index (an integer) in the given points or by its coordinates (anything iterable.)

OUTPUT:

A triangulation of the convex hull of the vertices as a `Triangulation`. The indices in the triangulation correspond to the `Vrepresentation()` objects.

EXAMPLES:

```
sage: cube = polytopes.n_cube(3)
sage: triangulation = cube.triangulate(
...       engine='internal') # to make doctest independent of TOPCOM
sage: triangulation
(<0,1,2,7>, <0,1,4,7>, <0,2,4,7>, <1,2,3,7>, <1,4,5,7>, <2,4,6,7>)
sage: simplex_indices = triangulation[0]; simplex_indices
(0, 1, 2, 7)
sage: simplex_vertices = [ cube.Vrepresentation(i) for i in simplex_indices ]
sage: simplex_vertices
[A vertex at (-1, -1, -1), A vertex at (-1, -1, 1),
 A vertex at (-1, 1, -1), A vertex at (1, 1, 1)]
sage: Polyhedron(simplex_vertices)
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 4 vertices
```

triangulated_facial_incidences()

Return a list of the form [face_index, [v_i_0, v_i_1,...,v_i_{n-1}]] where the face_index refers to the original defining inequality. For a given face, the collection of triangles formed by each list of v_i should triangulate that face.

In dimensions greater than 3, this is computed by randomly lifting each face up a dimension; this does not always work! This should eventually be fixed by using lrs or another program that computes triangulations.

EXAMPLES:

If the figure is already composed of triangles, then all is well:

```
sage: Polyhedron(vertices = [[5,0,0],[0,5,0],[5,5,0],[2,2,5]]
...              ).triangulated_facial_incidences()
doctest:...: DeprecationWarning:
This method is deprecated. Use triangulate() instead.
See http://trac.sagemath.org/11634 for details.
doctest:...: DeprecationWarning:
This method is deprecated. Use self.Hrepresentation(i).incident() instead.
See http://trac.sagemath.org/11763 for details.
[[0, [0, 1, 2]], [1, [0, 1, 3]], [2, [0, 2, 3]], [3, [1, 2, 3]]]
```

Otherwise some faces get split up to triangles:

```
sage: Polyhedron(vertices = [[2,0,0],[4,1,0],[0,5,0],[5,5,0],
...       [1,1,0],[0,0,1]]).triangulated_facial_incidences()
doctest:...: DeprecationWarning:
This method is deprecated. Use triangulate() instead.
See http://trac.sagemath.org/11634 for details.
doctest:...: DeprecationWarning:
This method is deprecated. Use self.Vrepresentation(i).neighbors() instead.
See http://trac.sagemath.org/11763 for details.
[[0, [1, 2, 5]], [0, [2, 5, 3]], [0, [5, 3, 4]], [1, [0, 1, 2]],
 [2, [0, 2, 3]], [3, [0, 3, 4]], [4, [0, 4, 5]], [5, [0, 1, 5]]]
```

vertex_adjacencies()

Return a list of vertex indices and their adjacent vertices.

Note: Instead of working with vertex indices, you can use the V-representation objects directly (see examples).

Two V-representation objects are adjacent if they generate a (1-dimensional) face of the polyhedron. Examples are two vertices of a polytope that bound an edge, or a vertex and a ray of a polyhedron that

generate a bounding half-line of the polyhedron. See vertex_adjacency_matrix() for a more detailed discussion.

OUTPUT:

The vertex indices are the indices of the V-representation objects.

EXAMPLES:

```
sage: permuta3 = Polyhedron(vertices = Permutations([1,2,3,4]))
sage: permuta3.vertex_adjacencies()[0:3]
doctest:...: DeprecationWarning:
This method is deprecated. Use self.Vrepresentation(i).neighbors() instead.
See http://trac.sagemath.org/11763 for details.
[[0, [1, 2, 6]], [1, [0, 3, 7]], [2, [0, 4, 8]]]
sage: v0 = permuta3.Vrepresentation(0)
sage: v0.index() == 0
True
sage: list( v0.neighbors() )
[A vertex at (1, 2, 4, 3), A vertex at (1, 3, 2, 4), A vertex at (2, 1, 3, 4)]
sage: v0_adjacencies = [v0.index(), [v.index() for v in v0.neighbors()]]
sage: permuta3.vertex_adjacencies()[0] == v0_adjacencies
True
```

vertex_adjacency_matrix()

Return the binary matrix of vertex adjacencies.

EXAMPLES:

```
sage: polytopes.n_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,)
```

Its (non-unique) V-representation consists of a vertex, a ray, and a line. The only edge is spanned by the vertex and the line generator, so they are adjacent:

```
sage: half_plane.Vrepresentation()
(A line in the direction (0, 1), A ray in the direction (1, 0), A vertex at (0, 0))
sage: half_plane.vertex_adjacency_matrix()
[0 0 1]
[0 0 0]
[1 0 0]
```

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:

```
sage: P = Polyhedron(vertices=[(0, 1), (1, 0), (3, 0), (4, 1)])
sage: for v in P.Vrep_generator():
...       print P.adjacency_matrix().row(v.index()), v
(0, 1, 0, 1) A vertex at (0, 1)
(1, 0, 1, 0) A vertex at (1, 0)
(0, 1, 0, 1) A vertex at (3, 0)
(1, 0, 1, 0) A vertex at (4, 1)
```

If the V-representation of the polygon contains vertices and one ray, then each V-representation object is adjacent to two V-representation objects:

```
sage: P = Polyhedron(vertices=[(0, 1), (1, 0), (3, 0), (4, 1)],
...                  rays=[(0,1)])
sage: for v in P.Vrep_generator():
...       print P.adjacency_matrix().row(v.index()), v
(0, 1, 0, 0, 1) A ray in the direction (0, 1)
(1, 0, 1, 0, 0) A vertex at (0, 1)
(0, 1, 0, 1, 0) A vertex at (1, 0)
(0, 0, 1, 0, 1) A vertex at (3, 0)
(1, 0, 0, 1, 0) A vertex at (4, 1)
```

If the V-representation of the polygon contains vertices and two distinct rays, then each vertex is adjacent to two V-representation objects (which can now be vertices or rays). The two rays are not adjacent to each other:

```
sage: P = Polyhedron(vertices=[(0, 1), (1, 0), (3, 0), (4, 1)],
...                  rays=[(0,1), (1,1)])
sage: for v in P.Vrep_generator():
...       print P.adjacency_matrix().row(v.index()), v
(0, 1, 0, 0, 0) A ray in the direction (0, 1)
(1, 0, 1, 0, 0) A vertex at (0, 1)
(0, 1, 0, 0, 1) A vertex at (1, 0)
(0, 0, 0, 0, 1) A ray in the direction (1, 1)
(0, 0, 1, 1, 0) A vertex at (3, 0)
```

vertex_generator()

Return a generator for the vertices of the polyhedron.

EXAMPLES:

```
sage: triangle = Polyhedron(vertices=[[1,0],[0,1],[1,1]])
sage: for v in triangle.vertex_generator(): print(v)
A vertex at (0, 1)
A vertex at (1, 0)
A vertex at (1, 1)
sage: v_gen = triangle.vertex_generator()
sage: v_gen.next()   # the first vertex
A vertex at (0, 1)
sage: v_gen.next()   # the second vertex
A vertex at (1, 0)
sage: v_gen.next()   # the third vertex
A vertex at (1, 1)
sage: try: v_gen.next()   # there are only three vertices
... except StopIteration: print "STOP"
STOP
sage: type(v_gen)
<type 'generator'>
sage: [ v for v in triangle.vertex_generator() ]
[A vertex at (0, 1), A vertex at (1, 0), A vertex at (1, 1)]
```

vertex_graph()

Return a graph in which the vertices correspond to vertices of the polyhedron, and edges to edges.

EXAMPLES:

```
sage: g3 = polytopes.n_cube(3).vertex_graph()
sage: len(g3.automorphism_group())
48
sage: s4 = polytopes.n_simplex(4).vertex_graph()
sage: s4.is_eulerian()
True
```

vertex_incidences()

Return the vertex-face incidences in the form $[v_i, [f_{i_0}, f_{i_1}, \ldots, f_{i_2}]]$.

Note: Instead of working with face/vertex indices, you can use the H-representation/V-representation objects directly (see examples).

EXAMPLES:

```
sage: p = polytopes.n_simplex(3)
sage: p.vertex_incidences()
doctest:...: DeprecationWarning:
This method is deprecated. Use self.Vrepresentation(i).incident() instead.
See http://trac.sagemath.org/11763 for details.
[[0, [0, 1, 2]], [1, [0, 1, 3]], [2, [0, 2, 3]], [3, [1, 2, 3]]]
sage: v0 = p.Vrepresentation(0)
sage: v0.index() == 0
True
sage: p.vertex_incidences()[0] == [ v0.index(), [h.index() for h in v0.incident()] ]
True
```

vertices()

Return all vertices of the polyhedron.

OUTPUT:

A tuple of vertices.

EXAMPLES:

```
sage: triangle = Polyhedron(vertices=[[1,0],[0,1],[1,1]])
sage: triangle.vertices()
(A vertex at (0, 1), A vertex at (1, 0), A vertex at (1, 1))
sage: a_simplex = Polyhedron(ieqs = [
...             [0,1,0,0,0],[0,0,1,0,0],[0,0,0,1,0],[0,0,0,0,1]
...         ], eqns = [[1,-1,-1,-1,-1]])
sage: a_simplex.vertices()
(A vertex at (1, 0, 0, 0), A vertex at (0, 1, 0, 0),
 A vertex at (0, 0, 1, 0), A vertex at (0, 0, 0, 1))
```

vertices_list()

Return a list of vertices of the polyhedron.

Note: It is recommended to use vertex_generator() instead to iterate over the list of Vertex objects.

EXAMPLES:

```
sage: triangle = Polyhedron(vertices=[[1,0],[0,1],[1,1]])
sage: triangle.vertices_list()
[[0, 1], [1, 0], [1, 1]]
sage: a_simplex = Polyhedron(ieqs = [
...             [0,1,0,0,0],[0,0,1,0,0],[0,0,0,1,0],[0,0,0,0,1]
...         ], eqns = [[1,-1,-1,-1,-1]])
sage: a_simplex.vertices_list()
[[1, 0, 0, 0], [0, 1, 0, 0], [0, 0, 1, 0], [0, 0, 0, 1]]
sage: a_simplex.vertices_list() == [list(v) for v in a_simplex.vertex_generator()]
True
```

vertices_matrix(*base_ring=None*)

Return the coordinates of the vertices as the columns of a matrix.

INPUT:

- base_ring – A ring or None (default). The base ring of the returned matrix. If not specified, the base ring of the polyhedron is used.

OUTPUT:

A matrix over base_ring whose columns are the coordinates of the vertices. A TypeError is raised if the coordinates cannot be converted to base_ring.

EXAMPLES:

```
sage: triangle = Polyhedron(vertices=[[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(*engine='auto'*, ***kwds*)

Return the volume of the polytope.

- `engine` – string. The backend to use. Allowed values are:
  - `'auto'` (default): see `triangulate()`.
  - `'internal'`: see `triangulate()`.
  - `'TOPCOM'`: see `triangulate()`.
  - `'lrs'`: use David Avis's lrs program (optional).
- `**kwds` – keyword arguments that are passed to the triangulation engine.

OUTPUT:

The volume of the polytope.

EXAMPLES:

```
sage: polytopes.n_cube(3).volume()
8
sage: (polytopes.n_cube(3)*2).volume()
64
sage: polytopes.twenty_four_cell().volume()
2

sage: polytopes.regular_polygon(5, base_ring=RDF).volume()
2.37764129...
sage: P5 = polytopes.regular_polygon(5, base_ring=QQ)
sage: P5.volume()   # rational approximation
338747171409976647350051567375347617527481227949456780132648787001 3/142471941722062242656108
sage: _.n()
2.37764129...
```

Volume of the same polytope, using the optional package lrs:

```
sage: P5.volume(engine='lrs') #optional - lrs
2.37764129...
```

`sage.geometry.polyhedron.base.`**`is_Polyhedron`**`(X)`

Test whether `X` is a Polyhedron.

INPUT:

- `X` – anything.

OUTPUT:

Boolean.

EXAMPLES:

```
sage: p = polytopes.n_cube(2)
sage: from sage.geometry.polyhedron.base import is_Polyhedron
sage: is_Polyhedron(p)
True
sage: is_Polyhedron(123456)
False
```

24.7 Base class for polyhedra over Q

class `sage.geometry.polyhedron.base_QQ.`**`Polyhedron_QQ`**`(parent, Vrep, Hrep, **kwds)`

Bases: `sage.geometry.polyhedron.base.Polyhedron_base`

Base class for Polyhedra over **Q**

TESTS:

```
sage: p = Polyhedron([(0,0)], base_ring=QQ);  p
A 0-dimensional polyhedron in QQ^2 defined as the convex hull of 1 vertex
sage: TestSuite(p).run()
```

24.8 Base class for polyhedra over $\mathbb{Z}$

class sage.geometry.polyhedron.base_ZZ.**Polyhedron_ZZ**(*parent*, *Vrep*, *Hrep*, ***kwds*)

Bases: sage.geometry.polyhedron.base.Polyhedron_base

Base class for Polyhedra over **Z**

TESTS:

```
sage: p = Polyhedron([(0,0)], base_ring=ZZ);  p
A 0-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex
sage: TestSuite(p).run(skip='_test_pickling')
```

Minkowski_decompositions()

Return all Minkowski sums that add up to the polyhedron.

OUTPUT:

A tuple consisting of pairs (X, Y) of **Z**-polyhedra that add up to `self`. All pairs up to exchange of the summands are returned, that is, (Y, X) is not included if (X, Y) already is.

EXAMPLES:

```
sage: square = Polyhedron(vertices=[(0,0),(1,0),(0,1),(1,1)])
sage: square.Minkowski_decompositions()
((A 0-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices),
 (A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices,
  A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices))
```

Example from http://cgi.di.uoa.gr/~amantzaf/geo/

```
sage: Q = Polyhedron(vertices=[(4,0), (6,0), (0,3), (4,3)])
sage: R = Polyhedron(vertices=[(0,0), (5,0), (8,4), (3,2)])
sage: (Q+R).Minkowski_decompositions()
((A 0-dimensional polyhedron in ZZ^2 defined as the convex hull of 1 vertex,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 7 vertices),
 (A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices),
 (A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 7 vertices),
 (A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 5 vertices,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 4 vertices),
 (A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 7 vertices),
 (A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 5 vertices,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 3 vertices),
 (A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 7 vertices),
 (A 1-dimensional polyhedron in ZZ^2 defined as the convex hull of 2 vertices,
  A 2-dimensional polyhedron in ZZ^2 defined as the convex hull of 6 vertices))
```

```
sage: [ len(square.dilation(i).Minkowski_decompositions())
...     for i in range(6) ]
[1, 2, 5, 8, 13, 18]
sage: [ ceil((i^2+2*i-1)/2)+1 for i in range(10) ]
[1, 2, 5, 8, 13, 18, 25, 32, 41, 50]
```

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 **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.n_cube(3)
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:

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()

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
```

polar()

Return the polar (dual) polytope.

The polytope must have the IP-property (see `has_IP_property()`), that is, the origin must be an interior point. In particular, it must be full-dimensional.

OUTPUT:

The polytope whose vertices are the coefficient vectors of the inequalities of `self` with inhomogeneous term normalized to unity.

EXAMPLES:

```
sage: p = Polyhedron(vertices=[(1,0,0),(0,1,0),(0,0,1),(-1,-1,-1)], base_ring=ZZ)
sage: p.polar()
A 3-dimensional polyhedron in ZZ^3 defined as the convex hull of 4 vertices
sage: type(_)
<class 'sage.geometry.polyhedron.backend_ppl.Polyhedra_ZZ_ppl_with_category.element_class'>
sage: p.polar().base_ring()
Integer Ring
```

24.9 Base class for polyhedra over RDF.

class sage.geometry.polyhedron.base_RDF.**Polyhedron_RDF**(*parent*, *Vrep*, *Hrep*, ***kwds*)

Bases: sage.geometry.polyhedron.base.Polyhedron_base

Base class for polyhedra over RDF.

TESTS:

```
sage: p = Polyhedron([(0,0)], base_ring=RDF);  p
A 0-dimensional polyhedron in RDF^2 defined as the convex hull of 1 vertex
sage: TestSuite(p).run()
```

CHAPTER

TWENTYFIVE

INDICES AND TABLES

- Index
- Module Index
- Search Page

BIBLIOGRAPHY

[Fulton] Wiliam Fulton, "Introduction to Toric Varieties", Princeton University Press

[Normaliz] Winfried Bruns, Bogdan Ichim, and Christof Soeger: Normaliz. http://www.mathematik.uni-osnabrueck.de/normaliz/

[BrunsKoch] W. Bruns and R. Koch, Computing the integral closure of an affine semigroup. Uni. Iaggelonicae Acta Math. 39, (2001), 59-70

[Klyachko] A. A. Klyachko, Equivariant Bundles on Toral Varieties. Mathematics of the USSR - Izvestiya 35 (1990), 337-375.

[CLS11] David A. Cox, John Little, and Hal Schenck. *Toric Varieties*. Volume 124 of *Graduate Studies in Mathematics*. American Mathematical Society, Providence, RI, 2011.

[HLY02] Yi Hu, Chien-Hao Liu, and Shing-Tung Yau. Toric morphisms and fibrations of toric Calabi-Yau hypersurfaces. *Adv. Theor. Math. Phys.*, 6(3):457-506, 2002. arXiv:math/0010082v2 [math.AG].

[KS98] Maximilian Kreuzer and Harald Skarke, Classification of Reflexive Polyhedra in Three Dimensions, arXiv:hep-th/9805190

[GK13] Roland Grinis and Alexander Kasprzyk, Normal forms of convex lattice polytopes, arXiv:1301.6641

[BN08] Victor V. Batyrev and Benjamin Nill. Combinatorial aspects of mirror symmetry. In *Integer points in polyhedra — geometry, number theory, representation theory, algebra, optimization, statistics*, volume 452 of *Contemp. Math.*, pages 35–66. Amer. Math. Soc., Providence, RI, 2008. arXiv:math/0703456v2 [math.CO].

[CK99] David A. Cox and Sheldon Katz. *Mirror symmetry and algebraic geometry*, volume 68 of *Mathematical Surveys and Monographs*. American Mathematical Society, Providence, RI, 1999.

[Fetter2012] Hans L. Fetter, "A Polyhedron Full of Surprises", Mathematics Magazine 85 (2012), no. 5, 334-342.

[Felsner] On the Number of Arrangements of Pseudolines, Stefan Felsner, http://page.math.tu-berlin.de/~felsner/Paper/numarr.pdf

[TOPCOM] J. Rambau, TOPCOM <http://www.rambau.wm.uni-bayreuth.de/TOPCOM/>.

[GKZ] Gel'fand, I. M.; Kapranov, M. M.; and Zelevinsky, A. V. "Discriminants, Resultants and Multidimensional Determinants" Birkhauser 1994.

[PUNTOS] Jesus A. De Loera http://www.math.ucdavis.edu/~deloera/RECENT_WORK/puntos2000

[RS] Stanley, Richard: *Hyperplane Arrangements*, Geometric Combinatorics (E. Miller, V. Reiner, and B. Sturmfels, eds.), IAS/Park City Mathematics Series, vol. 13, American Mathematical Society, Providence, RI, 2007, pp. 389-496.

[GZ] Greene; Zaslavsky "On the Interpretation of Whitney Numbers Through Arrangements of Hyperplanes, Zonotopes, Non-Radon Partitions, and Orientations of Graphs" Transactions of the American Mathematical Society, Vol. 280, No. 1. (Nov., 1983), pp. 97-126.

[AR] D. Armstrong, B. Rhoades "The Shi arrangement and the Ish arrangement" Arxiv 1009.1655

[BigraphicalArrangements] S. Hopkins, D. Perkinson. "Bigraphical Arrangements". Arxiv 1212.4398

[FukudaProdon] Komei Fukuda , Alain Prodon: Double Description Method Revisited, Combinatorics and Computer Science, volume 1120 of Lecture Notes in Computer Science, page 91-111. Springer (1996)

[KP2002] Volker Kaibel and Marc E. Pfetsch, "Computing the Face Lattice of a Polytope from its Vertex-Facet Incidences", Computational Geometry: Theory and Applications, Volume 23, Issue 3 (November 2002), 281-290. Available at http://portal.acm.org/citation.cfm?id=763203 and free of charge at http://arxiv.org/abs/math/0106043

[BSS] David Bremner, Mathieu Dutour Sikiric, Achill Schuermann: Polyhedral representation conversion up to symmetries. http://arxiv.org/abs/math/0702239

[PALP] Maximilian Kreuzer, Harald Skarke: "PALP: A Package for Analyzing Lattice Polytopes with Applications to Toric Geometry" Comput.Phys.Commun. 157 (2004) 87-106 Arxiv math/0204356

PYTHON MODULE INDEX

g

r

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z