Chain complexes and homology

Release 10.3

The Sage Development Team

Mar 20, 2024

CONTENTS

Sage includes some tools for algebraic topology, and in particular computing homology groups.

CHAPTER

ONE

CHAIN COMPLEXES

This module implements bounded chain complexes of free R-modules, for any commutative ring R (although the interesting things, like homology, only work if R is the integers or a field).

Fix a ring R. A chain complex over R is a collection of R-modules $\{C_n\}$ indexed by the integers, with R-module maps $d_n : C_n \to C_{n+1}$ such that $d_{n+1} \circ d_n = 0$ for all n. The maps d_n are called *differentials*.

One can vary this somewhat: the differentials may decrease degree by one instead of increasing it: sometimes a chain complex is defined with $d_n : C_n \to C_{n-1}$ for each n. Indeed, the differentials may change dimension by any fixed integer.

Also, the modules may be indexed over an abelian group other than the integers, e.g., $\mathbf{Z}^m$ for some integer $m \geq 1$, in which case the differentials may change the grading by any element of that grading group. The elements of the grading group are generally called degrees, so C_n is the module in degree n and so on.

In this implementation, the ring R must be commutative and the modules C_n must be free R-modules. As noted above, homology calculations will only work if the ring R is either $\mathbf{Z}$ or a field. The modules may be indexed by any free abelian group. The differentials may increase degree by 1 or decrease it, or indeed change it by any fixed amount: this is controlled by the `degree_of_differential` parameter used in defining the chain complex.

AUTHORS:

- John H. Palmieri (2009-04): initial implementation

`sage.homology.chain_complex.`**`ChainComplex`**(*data=None*, *base_ring=None*, *grading_group=None*, *degree_of_differential=1*, *degree=1*, *check=True*)

Define a chain complex.

INPUT:

- `data` – the data defining the chain complex; see below for more details.

The following keyword arguments are supported:

- `base_ring` – a commutative ring (optional), the ring over which the chain complex is defined. If this is not specified, it is determined by the data defining the chain complex.
- `grading_group` – a additive free abelian group (optional, default `ZZ`), the group over which the chain complex is indexed.
- `degree_of_differential` – element of grading_group (optional, default `1`). The degree of the differential.
- `degree` – alias for `degree_of_differential`.
- `check` – boolean (optional, default `True`). If `True`, check that each consecutive pair of differentials are composable and have composite equal to zero.

OUTPUT:

A chain complex.

Warning: Right now, homology calculations will only work if the base ring is either **Z** or a field, so please take this into account when defining a chain complex.

Use data to define the chain complex. This may be in any of the following forms.

1. a dictionary with integers (or more generally, elements of grading_group) for keys, and with `data[n]` a matrix representing (via left multiplication) the differential coming from degree n. (Note that the shape of the matrix then determines the rank of the free modules C_n and C_{n+d}.)
2. a list/tuple/iterable of the form $[C_0, d_0, C_1, d_1, C_2, d_2, ...]$, where each C_i is a free module and each d_i is a matrix, as above. This only makes sense if `grading_group` is **Z** and `degree` is 1.
3. a list/tuple/iterable of the form $[r_0, d_0, r_1, d_1, r_2, d_2, \ldots]$, where r_i is the rank of the free module C_i and each d_i is a matrix, as above. This only makes sense if `grading_group` is **Z** and `degree` is 1.
4. a list/tuple/iterable of the form $[d_0, d_1, d_2, \ldots]$ where each d_i is a matrix, as above. This only makes sense if `grading_group` is **Z** and `degree` is 1.

Note: In fact, the free modules C_i in case 2 and the ranks r_i in case 3 are ignored: only the matrices are kept, and from their shapes, the ranks of the modules are determined. (Indeed, if `data` is a list or tuple, then any element which is not a matrix is discarded; thus the list may have any number of different things in it, and all of the non-matrices will be ignored.) No error checking is done to make sure, for instance, that the given modules have the appropriate ranks for the given matrices. However, as long as `check` is True, the code checks to see if the matrices are composable and that each appropriate composite is zero.

If the base ring is not specified, then the matrices are examined to determine a ring over which they are all naturally defined, and this becomes the base ring for the complex. If no such ring can be found, an error is raised. If the base ring is specified, then the matrices are converted automatically to this ring when defining the chain complex. If some matrix cannot be converted, then an error is raised.

EXAMPLES:

```
sage: ChainComplex()
Trivial chain complex over Integer Ring

sage: C = ChainComplex({0: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0])})
sage: C
Chain complex with at most 2 nonzero terms over Integer Ring

sage: m = matrix(ZZ, 2, 2, [0, 1, 0, 0])
sage: D = ChainComplex([m, m], base_ring=GF(2)); D
Chain complex with at most 3 nonzero terms over Finite Field of size 2
sage: D == loads(dumps(D))
True
sage: D.differential(0)==m, m.is_immutable(), D.differential(0).is_immutable()
(True, False, True)
```

Note that when a chain complex is defined in Sage, new differentials may be created: every nonzero module in the chain complex must have a differential coming from it, even if that differential is zero:

```
sage: IZ = ChainComplex({0: identity_matrix(ZZ, 1)})
sage: diff = IZ.differential()  # the differentials in the chain complex
sage: diff[-1], diff[0], diff[1]
([], [1], [])
sage: IZ.differential(1).parent()
Full MatrixSpace of 0 by 1 dense matrices over Integer Ring
sage: mat = ChainComplex({0: matrix(ZZ, 3, 4)}).differential(1)
sage: mat.nrows(), mat.ncols()
(0, 3)
```

Defining the base ring implicitly:

```
sage: ChainComplex([matrix(QQ, 3, 1), matrix(ZZ, 4, 3)])
Chain complex with at most 3 nonzero terms over Rational Field
sage: ChainComplex([matrix(GF(125, 'a'), 3, 1), matrix(ZZ, 4, 3)])              #␣
↪needs sage.rings.finite_rings
Chain complex with at most 3 nonzero terms over Finite Field in a of size 5^3
```

If the matrices are defined over incompatible rings, an error results:

```
sage: ChainComplex([matrix(GF(125, 'a'), 3, 1), matrix(QQ, 4, 3)])              #␣
↪needs sage.rings.finite_rings
Traceback (most recent call last):
...
TypeError: no common canonical parent for objects with parents:
'Finite Field in a of size 5^3' and 'Rational Field'
```

If the base ring is given explicitly but is not compatible with the matrices, an error results:

```
sage: ChainComplex([matrix(GF(125, 'a'), 3, 1)], base_ring=QQ)                  #␣
↪needs sage.rings.finite_rings
Traceback (most recent call last):
...
TypeError: unable to convert 0 to a rational
```

class sage.homology.chain_complex.**ChainComplex_class**(*grading_group*, *degree_of_differential*, *base_ring*, *differentials*)

Bases: `Parent`

See *ChainComplex()* for full documentation.

The differentials are required to be in the following canonical form:

- All differentials that are not 0×0 must be specified (even if they have zero rows or zero columns), and
- Differentials that are 0×0 must not be specified.
- Immutable matrices over the `base_ring`

This and more is ensured by the assertions in the constructor. The *ChainComplex()* factory function must ensure that only valid input is passed.

EXAMPLES:

```
sage: C = ChainComplex(); C
Trivial chain complex over Integer Ring

sage: D = ChainComplex({0: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0])})
sage: D
Chain complex with at most 2 nonzero terms over Integer Ring
```

Element

alias of *Chain_class*

betti(*deg=None*, *base_ring=None*)

The Betti number of the chain complex.

That is, write the homology in this degree as a direct sum of a free module and a torsion module; the Betti number is the rank of the free summand.

INPUT:

- `deg` – an element of the grading group for the chain complex or None (default `None`); if `None`, then return every Betti number, as a dictionary indexed by degree, or if an element of the grading group, then return the Betti number in that degree
- `base_ring` – a commutative ring (optional, default is the base ring for the chain complex); compute homology with these coefficients – must be either the integers or a field

OUTPUT:

The Betti number in degree `deg` – the rank of the free part of the homology module in this degree.

EXAMPLES:

```
sage: C = ChainComplex({0: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0])})
sage: C.betti(0)
2
sage: [C.betti(n) for n in range(5)]
[2, 1, 0, 0, 0]
sage: C.betti()
{0: 2, 1: 1}

sage: D = ChainComplex({0: matrix(GF(5), [[3, 1],[1, 2]])})
sage: D.betti()
{0: 1, 1: 1}
```

cartesian_product(**factors*, ***kwds*)

Return the direct sum (Cartesian product) of `self` with `D`.

Let C and D be two chain complexes with differentials ∂_C and ∂_D, respectively, of the same degree (so they must also have the same grading group). The direct sum $S = C \oplus D$ is a chain complex given by $S_i = C_i \oplus D_i$ with differential $\partial = \partial_C \oplus \partial_D$.

INPUT:

- `subdivide` – (default: `False`) whether to subdivide the the differential matrices

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: C = ChainComplex([matrix([[-y],[x]]), matrix([[x, y]])])
sage: D = ChainComplex([matrix([[x-y]]), matrix([[0], [0]])])
sage: ascii_art(C.cartesian_product(D))
            [x y 0]       [   -y     0]
            [0 0 0]       [    x     0]
            [0 0 0]       [    0 x - y]
 0 <-- C_2 <-------- C_1 <-------------- C_0 <-- 0

sage: D = ChainComplex({1:matrix([[x-y]]), 4:matrix([[x], [y]])})
sage: ascii_art(D)
            [x]
```

(continues on next page)

(continued from previous page)

```
            [y]                       [x - y]
 0 <-- C_5 <---- C_4 <-- 0 <-- C_2 <-------- C_1 <-- 0
sage: ascii_art(cartesian_product([C, D]))
                                                          [-y]
            [x]                       [    x     y     0]  [ x]
            [y]                       [    0     0 x - y]  [ 0]
 0 <-- C_5 <---- C_4 <-- 0 <-- C_2 <-------------------- C_1 <----- C_0 <-- 0
```

The degrees of the differentials must agree:

```
sage: C = ChainComplex({1:matrix([[x]])}, degree_of_differential=-1)
sage: D = ChainComplex({1:matrix([[x]])}, degree_of_differential=1)
sage: C.cartesian_product(D)
Traceback (most recent call last):
...
ValueError: the degrees of the differentials must match
```

degree_of_differential()

Return the degree of the differentials of the complex

OUTPUT:

An element of the grading group.

EXAMPLES:

```
sage: D = ChainComplex({0: matrix(ZZ, 2, 2, [1,0,0,2])})
sage: D.degree_of_differential()
1
```

differential(*dim=None*)

The differentials which make up the chain complex.

INPUT:

- `dim` – element of the grading group (optional, default `None`); if this is `None`, return a dictionary of all of the differentials, or if this is a single element, return the differential starting in that dimension

OUTPUT:

Either a dictionary of all of the differentials or a single differential (i.e., a matrix).

EXAMPLES:

```
sage: D = ChainComplex({0: matrix(ZZ, 2, 2, [1,0,0,2])})
sage: D.differential(0)
[1 0]
[0 2]
sage: D.differential(-1)
[]
sage: C = ChainComplex({0: identity_matrix(ZZ, 40)})
sage: diff = C.differential()
sage: diff[-1]
40 x 0 dense matrix over Integer Ring (use the '.str()' method to see the␣
↪entries)
sage: diff[0]
40 x 40 dense matrix over Integer Ring (use the '.str()' method to see the␣
↪entries)
```

(continues on next page)

(continued from previous page)

```
sage: diff[1]
[]
```

dual()

The dual chain complex to `self`.

Since all modules in `self` are free of finite rank, the dual in dimension n is isomorphic to the original chain complex in dimension n, and the corresponding boundary matrix is the transpose of the matrix in the original complex. This converts a chain complex to a cochain complex and vice versa.

EXAMPLES:

```
sage: C = ChainComplex({2: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0])})
sage: C.degree_of_differential()
1
sage: C.differential(2)
[3 0 0]
[0 0 0]
sage: C.dual().degree_of_differential()
-1
sage: C.dual().differential(3)
[3 0]
[0 0]
[0 0]
```

free_module(*degree=None*)

Return the free module at fixed `degree`, or their sum.

INPUT:

- `degree` – an element of the grading group or `None` (default).

OUTPUT:

The free module C_n at the given degree n. If the degree is not specified, the sum $\bigoplus C_n$ is returned.

EXAMPLES:

```
sage: C = ChainComplex({0: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0]), 1: matrix(ZZ,
↪ [[0, 1]])})
sage: C.free_module()
Ambient free module of rank 6 over the principal ideal domain Integer Ring
sage: C.free_module(0)
Ambient free module of rank 3 over the principal ideal domain Integer Ring
sage: C.free_module(1)
Ambient free module of rank 2 over the principal ideal domain Integer Ring
sage: C.free_module(2)
Ambient free module of rank 1 over the principal ideal domain Integer Ring
```

free_module_rank(*degree*)

Return the rank of the free module at the given `degree`.

INPUT:

- `degree` – an element of the grading group

OUTPUT:

Integer. The rank of the free module C_n at the given degree n.

EXAMPLES:

```
sage: C = ChainComplex({0: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0]), 1: matrix(ZZ,
↪ [[0, 1]])})
sage: [C.free_module_rank(i) for i in range(-2, 5)]
[0, 0, 3, 2, 1, 0, 0]
```

grading_group()

Return the grading group.

OUTPUT:

The discrete abelian group that indexes the individual modules of the complex. Usually **Z**.

EXAMPLES:

```
sage: G = AdditiveAbelianGroup([0, 3])
sage: C = ChainComplex(grading_group=G, degree=G(vector([1,2])))
sage: C.grading_group()
Additive abelian group isomorphic to Z + Z/3
sage: C.degree_of_differential()
(1, 2)
```

homology(*deg=None*, *base_ring=None*, *generators=False*, *verbose=False*, *algorithm='pari'*)

The homology of the chain complex.

INPUT:

- `deg` – an element of the grading group for the chain complex (default: `None`); the degree in which to compute homology – if this is `None`, return the homology in every degree in which the chain complex is possibly nonzero.
- `base_ring` – a commutative ring (optional, default is the base ring for the chain complex); must be either the integers **Z** or a field
- `generators` – boolean (optional, default `False`); if `True`, return generators for the homology groups along with the groups. See github issue #6100
- `verbose` - boolean (optional, default `False`); if `True`, print some messages as the homology is computed
- `algorithm` - string (optional, default `'pari'`); the options are:
  - `'auto'`
  - `'dhsw'`
  - `'pari'`
  - `'chomp'` (this option is deprecated)

  See below for descriptions.

OUTPUT:

If the degree is specified, the homology in degree `deg`. Otherwise, the homology in every dimension as a dictionary indexed by dimension.

ALGORITHM:

Over a field, just compute ranks and nullities, thus obtaining dimensions of the homology groups as vector spaces. Over the integers, compute Smith normal form of the boundary matrices defining the chain complex according to the value of `algorithm`. If `algorithm` is `'auto'`, then for each relatively small matrix, use the standard Sage method, which calls the Pari package. For any large matrix, reduce it using the Dumas, Heckenbach, Saunders, and Welker elimination algorithm [DHSW2003]: see *dhsw_snf()* for details.

`'no_chomp'` is a synonym for `'auto'`, maintained for backward-compatibility.

`algorithm` may also be `'pari'` or `'dhsw'`, which forces the named algorithm to be used regardless of the size of the matrices.

Finally, if `algorithm` is set to `'chomp'`, then use CHomP. CHomP is available at the web page http://chomp.rutgers.edu/, although the software has not been tested recently in Sage. The use of this option is deprecated; see github issue #33777.

As of this writing, `'pari'` is the fastest standard option.

Warning: This only works if the base ring is the integers or a field. Other values will return an error.

EXAMPLES:

```
sage: C = ChainComplex({0: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0])})
sage: C.homology()
{0: Z x Z, 1: Z x C3}
sage: C.homology(deg=1, base_ring=GF(3))
Vector space of dimension 2 over Finite Field of size 3
sage: D = ChainComplex({0: identity_matrix(ZZ, 4), 4: identity_matrix(ZZ, 30)}
↪)
sage: D.homology()
{0: 0, 1: 0, 4: 0, 5: 0}
```

Generators: generators are given as a list of cycles, each of which is an element in the appropriate free module, and hence is represented as a vector. Each summand of the homology is listed separately, with a corresponding generator:

```
sage: C.homology(1, generators=True)
[(C3, Chain(1:(1, 0))), (Z, Chain(1:(0, 1)))]
```

Tests for github issue #6100, the Klein bottle with generators:

```
sage: d0 = matrix(ZZ, 0,1)
sage: d1 = matrix(ZZ, 1,3, [[0,0,0]])
sage: d2 = matrix(ZZ, 3,2, [[1,1], [1,-1], [-1,1]])
sage: C_k = ChainComplex({0:d0, 1:d1, 2:d2}, degree=-1)
sage: C_k.homology(generators=true)
{0: [(Z, Chain(0:(1)))],
 1: [(C2, Chain(1:(0, 1, -1))), (Z, Chain(1:(0, 1, 0)))],
 2: []}
```

From a torus using a field:

```
sage: T = simplicial_complexes.Torus()                                    #␣
↪needs sage.graphs
sage: C_t = T.chain_complex()                                             #␣
↪needs sage.graphs
sage: C_t.homology(base_ring=QQ, generators=True)                         #␣
↪needs sage.graphs
{0: [(Vector space of dimension 1 over Rational Field,
   Chain(0:(0, 0, 0, 0, 0, 0, 1)))],
 1: [(Vector space of dimension 1 over Rational Field,
   Chain(1:(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, -1, 0, 0, 1))),
  (Vector space of dimension 1 over Rational Field,
   Chain(1:(0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -1, 0, 1, 0, -1,␣
```

(continues on next page)

(continued from previous page)

```
↪0))))],
 2: [(Vector space of dimension 1 over Rational Field,
   Chain(2:(1, -1, 1, -1, 1, -1, -1, 1, -1, 1, -1, 1, 1, -1))))]}
```

nonzero_degrees()

Return the degrees in which the module is non-trivial.

See also *ordered_degrees()*.

OUTPUT:

The tuple containing all degrees n (grading group elements) such that the module C_n of the chain is non-trivial.

EXAMPLES:

```
sage: one = matrix(ZZ, [[1]])
sage: D = ChainComplex({0: one, 2: one, 6:one})
sage: ascii_art(D)
            [1]                                 [1]        [0]        [1]
0 <-- C_7 <---- C_6 <-- 0  ...  0 <-- C_3 <---- C_2 <---- C_1 <---- C_0 <-- 0
sage: D.nonzero_degrees()
(0, 1, 2, 3, 6, 7)
```

ordered_degrees(*start=None*, *exclude_first=False*)

Sort the degrees in the order determined by the differential

INPUT:

- `start` – (default: `None`) a degree (element of the grading group) or `None`
- `exclude_first` – boolean (optional; default: `False`); whether to exclude the lowest degree – this is a handy way to just get the degrees of the non-zero modules, as the domain of the first differential is zero.

OUTPUT:

If `start` has been specified, the longest tuple of degrees

- containing `start` (unless `start` would be the first and `exclude_first=True`),
- in ascending order relative to *degree_of_differential()*, and
- such that none of the corresponding differentials are 0×0.

If `start` has not been specified, a tuple of such tuples of degrees. One for each sequence of non-zero differentials. They are returned in sort order.

EXAMPLES:

```
sage: one = matrix(ZZ, [[1]])
sage: D = ChainComplex({0: one, 2: one, 6:one})
sage: ascii_art(D)
            [1]                                 [1]        [0]        [1]
0 <-- C_7 <---- C_6 <-- 0  ...  0 <-- C_3 <---- C_2 <---- C_1 <---- C_0 <-- 0
sage: D.ordered_degrees()
((-1, 0, 1, 2, 3), (5, 6, 7))
sage: D.ordered_degrees(exclude_first=True)
((0, 1, 2, 3), (6, 7))
sage: D.ordered_degrees(6)
(5, 6, 7)
```

(continues on next page)

(continued from previous page)

```
sage: D.ordered_degrees(5, exclude_first=True)
(6, 7)
```

random_element()

Return a random element.

EXAMPLES:

```
sage: D = ChainComplex({0: matrix(ZZ, 2, 2, [1,0,0,2])})
sage: D.random_element()    # random output
Chain with 1 nonzero terms over Integer Ring
```

rank(*degree*, *ring=None*)

Return the rank of a differential

INPUT:

- `degree` – an element δ of the grading group. Which differential d_δ we want to know the rank of
- `ring` – (optional) a commutative ring S; if specified, the rank is computed after changing to this ring

OUTPUT:

The rank of the differential $d_\delta \otimes_R S$, where R is the base ring of the chain complex.

EXAMPLES:

```
sage: C = ChainComplex({0:matrix(ZZ, [[2]])})
sage: C.differential(0)
[2]
sage: C.rank(0)
1
sage: C.rank(0, ring=GF(2))
0
```

shift(*n=1*)

Shift this chain complex n times.

INPUT:

- `n` – an integer (optional, default 1)

The *shift* operation is also sometimes called *translation* or *suspension*.

To shift a chain complex by n, shift its entries up by n (if it is a chain complex) or down by n (if it is a cochain complex); that is, shifting by 1 always shifts in the opposite direction of the differential. In symbols, if C is a chain complex and $C[n]$ is its n-th shift, then $C[n]_j = C_{j-n}$. The differential in the shift $C[n]$ is obtained by multiplying each differential in C by $(-1)^n$.

Caveat: different sources use different conventions for shifting: what we call $C[n]$ might be called $C[-n]$ in some places. See for example. https://ncatlab.org/nlab/show/suspension+of+a+chain+complex (which uses $C[n]$ as we do but acknowledges $C[-n]$) or 1.2.8 in [Wei1994] (which uses $C[-n]$).

EXAMPLES:

```
sage: # needs sage.graphs
sage: S1 = simplicial_complexes.Sphere(1).chain_complex()
sage: S1.shift(1).differential(2) == -S1.differential(1)
True
sage: S1.shift(2).differential(3) == S1.differential(1)
```

(continues on next page)

(continued from previous page)

```
True
sage: S1.shift(3).homology(4)
Z
```

For cochain complexes, shifting goes in the other direction. Topologically, this makes sense if we grade the cochain complex for a space negatively:

```
sage: # needs sage.graphs
sage: T = simplicial_complexes.Torus()
sage: co_T = T.chain_complex()._flip_()
sage: co_T.homology()
{-2: Z, -1: Z x Z, 0: Z}
sage: co_T.degree_of_differential()
1
sage: co_T.shift(2).homology()
{-4: Z, -3: Z x Z, -2: Z}
```

You can achieve the same result by tensoring (on the left, to get the signs right) with a rank one free module in degree `-n * deg`, if `deg` is the degree of the differential:

```
sage: C = ChainComplex({-2: matrix(ZZ, 0, 1)})
sage: C.tensor(co_T).homology()                                                 #␣
↪needs sage.graphs
{-4: Z, -3: Z x Z, -2: Z}
```

tensor(**factors*, ***kwds*)

Return the tensor product of `self` with `D`.

Let C and D be two chain complexes with differentials ∂_C and ∂_D, respectively, of the same degree (so they must also have the same grading group). The tensor product $S = C \otimes D$ is a chain complex given by

$$S_i = \bigoplus_{a+b=i} C_a \otimes D_b$$

with differential

$$\partial(x \otimes y) = \partial_C x \otimes y + (-1)^{|a| \cdot |\partial_D|} x \otimes \partial_D y$$

for $x \in C_a$ and $y \in D_b$, where $|a|$ is the degree of a and $|\partial_D|$ is the degree of ∂_D.

Warning: If the degree of the differential is even, then this may not result in a valid chain complex.

INPUT:

- `subdivide` – (default: `False`) whether to subdivide the the differential matrices

Todo: Make subdivision work correctly on multiple factors.

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: C1 = ChainComplex({1:matrix([[x]])}, degree_of_differential=-1)
sage: C2 = ChainComplex({1:matrix([[y]])}, degree_of_differential=-1)
```

(continues on next page)

(continued from previous page)

```
sage: C3 = ChainComplex({1:matrix([[z]])}, degree_of_differential=-1)
sage: ascii_art(C1.tensor(C2))
                          [ x]
            [y x]         [-y]
 0 <-- C_0 <------ C_1 <----- C_2 <-- 0
sage: ascii_art(C1.tensor(C2).tensor(C3))
                            [ y  x  0]       [ x]
                            [-z  0  x]       [-y]
            [z y x]         [ 0 -z -y]       [ z]
 0 <-- C_0 <-------- C_1 <----------- C_2 <----- C_3 <-- 0
```

```
sage: C = ChainComplex({2:matrix([[-y],[x]]), 1:matrix([[x, y]])},
....:                  degree_of_differential=-1); ascii_art(C)
                        [-y]
            [x y]       [ x]
 0 <-- C_0 <------ C_1 <----- C_2 <-- 0
sage: T = C.tensor(C)
sage: T.differential(1)
[x y x y]
sage: T.differential(2)
[-y  x  0  y  0  0]
[ x  0  x  0  y  0]
[ 0 -x -y  0  0 -y]
[ 0  0  0 -x -y  x]
sage: T.differential(3)
[ x  y  0  0]
[ y  0 -y  0]
[-x  0  0 -y]
[ 0  y  x  0]
[ 0 -x  0  x]
[ 0  0  x  y]
sage: T.differential(4)
[-y]
[ x]
[-y]
[ x]
```

The degrees of the differentials must agree:

```
sage: C1p = ChainComplex({1:matrix([[x]])}, degree_of_differential=1)
sage: C1.tensor(C1p)
Traceback (most recent call last):
...
ValueError: the degrees of the differentials must match
```

torsion_list(*max_prime*, *min_prime=2*)

Look for torsion in this chain complex by computing its mod p homology for a range of primes p.

INPUT:

- `max_prime` – prime number; search for torsion mod p for all p strictly less than this number
- `min_prime` – prime (optional, default 2); search for torsion mod p for primes at least as big as this

Return a list of pairs (p, d) where p is a prime at which there is torsion and d is a list of dimensions in which this torsion occurs.

The base ring for the chain complex must be the integers; if not, an error is raised.

ALGORITHM:

Let C denote the chain complex. Let P equal `max_prime`. Compute the mod P homology of C, and use this as the base-line computation: the assumption is that this is isomorphic to the integral homology tensored with $\mathbf{F}_P$. Then compute the mod p homology for a range of primes p, and record whenever the answer differs from the base-line answer.

EXAMPLES:

```
sage: C = ChainComplex({0: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0])})
sage: C.homology()
{0: Z x Z, 1: Z x C3}
sage: C.torsion_list(11)                                                    #␣
↪needs sage.rings.finite_rings
[(3, [1])]
sage: C = ChainComplex([matrix(ZZ, 1, 1, [2]), matrix(ZZ, 1, 1), matrix(1, 1,␣
↪[3])])
sage: C.homology(1)
C2
sage: C.homology(3)
C3
sage: C.torsion_list(5)                                                     #␣
↪needs sage.rings.finite_rings
[(2, [1]), (3, [3])]
```

class sage.homology.chain_complex.**Chain_class**(*parent*, *vectors*, *check=True*)

Bases: ModuleElement

A Chain in a Chain Complex

A chain is collection of module elements for each module C_n of the chain complex (C_n, d_n). There is no restriction on how the differentials d_n act on the elements of the chain.

Note: You must use the chain complex to construct chains.

EXAMPLES:

```
sage: C = ChainComplex({0: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0])},
....:                  base_ring=GF(7))
sage: C.category()
Category of chain complexes over Finite Field of size 7
```

is_boundary()

Return whether the chain is a boundary.

OUTPUT:

Boolean. Whether the elements of the chain are in the image of the differentials.

EXAMPLES:

```
sage: C = ChainComplex({0: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0])})
sage: c = C({0: vector([0, 1, 2]), 1: vector([3, 4])})
sage: c.is_boundary()
False
sage: z3 = C({1:(1, 0)})
sage: z3.is_cycle()
True
```

(continues on next page)

(continued from previous page)

```
sage: (2*z3).is_boundary()
False
sage: (3*z3).is_boundary()
True
```

is_cycle()

Return whether the chain is a cycle.

OUTPUT:

Boolean. Whether the elements of the chain are in the kernel of the differentials.

EXAMPLES:

```
sage: C = ChainComplex({0: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0])})
sage: c = C({0: vector([0, 1, 2]), 1: vector([3, 4])})
sage: c.is_cycle()
True
```

vector(*degree*)

Return the free module element in `degree`.

EXAMPLES:

```
sage: C = ChainComplex({0: matrix(ZZ, 2, 3, [3, 0, 0, 0, 0, 0])})
sage: c = C({0: vector([1, 2, 3]), 1: vector([4, 5])})
sage: c.vector(0)
(1, 2, 3)
sage: c.vector(1)
(4, 5)
sage: c.vector(2)
()
```

CHAPTER

TWO

CHAINS AND COCHAINS

This module implements formal linear combinations of cells of a given cell complex (*Chains*) and their dual (*Cochains*). It is closely related to the `sage.topology.chain_complex` module. The main differences are that chains and cochains here are of homogeneous dimension only, and that they reference their cell complex.

class `sage.homology.chains.`**CellComplexReference**(*cell_complex*, *degree*, *cells=None*)

Bases: `object`

Auxiliary base class for chains and cochains

INPUT:

- `cell_complex` – The cell complex to reference
- `degree` – integer. The degree of the (co)chains
- `cells` – tuple of cells or `None`. Does not necessarily have to be the cells in the given degree, for computational purposes this could also be any collection that is in one-to-one correspondence with the cells. If `None`, the cells of the complex in the given degree are used.

EXAMPLES:

```
sage: X = simplicial_complexes.Simplex(2)
sage: from sage.homology.chains import CellComplexReference
sage: c = CellComplexReference(X, 1)
sage: c.cell_complex() is X
True
```

cell_complex()

Return the underlying cell complex

OUTPUT:

A cell complex.

EXAMPLES:

```
sage: X = simplicial_complexes.Simplex(2)
sage: X.n_chains(1).cell_complex() is X
True
```

degree()

Return the dimension of the cells

OUTPUT:

Integer. The dimension of the cells.

EXAMPLES:

```
sage: X = simplicial_complexes.Simplex(2)
sage: X.n_chains(1).degree()
1
```

class sage.homology.chains.**Chains**(*cell_complex*, *degree*, *cells=None*, *base_ring=None*)

Bases: *CellComplexReference*, CombinatorialFreeModule

Class for the free module of chains in a given degree.

INPUT:

- `n_cells` – tuple of n-cells, which thus forms a basis for this module
- `base_ring` – optional (default **Z**)

One difference between chains and cochains is notation. In a simplicial complex, for example, a simplex `(0,1,2)` is written as "(0,1,2)" in the group of chains but as "\chi_(0,1,2)" in the group of cochains.

Also, since the free modules of chains and cochains are dual, there is a pairing $\langle c, z \rangle$, sending a cochain c and a chain z to a scalar.

EXAMPLES:

```
sage: S2 = simplicial_complexes.Sphere(2)
sage: C_2 = S2.n_chains(1)
sage: C_2_co = S2.n_chains(1, cochains=True)
sage: x = C_2.basis()[Simplex((0,2))]
sage: y = C_2.basis()[Simplex((1,3))]
sage: z = x+2*y
sage: a = C_2_co.basis()[Simplex((1,3))]
sage: b = C_2_co.basis()[Simplex((0,3))]
sage: c = 3*a-2*b
sage: z
(0, 2) + 2*(1, 3)
sage: c
-2*\chi_(0, 3) + 3*\chi_(1, 3)
sage: c.eval(z)
6
```

class Element

Bases: IndexedFreeModuleElement

boundary()

Return the boundary of the chain

OUTPUT:

The boundary as a chain in one degree lower.

EXAMPLES:

```
sage: square = cubical_complexes.Cube(2)
sage: C1 = square.n_chains(1, QQ)
sage: from sage.topology.cubical_complex import Cube
sage: chain = C1(Cube([[1, 1], [0, 1]])) - 2 * C1(Cube([[0, 1], [0, 0]]))
sage: chain
-2*[0,1] x [0,0] + [1,1] x [0,1]
sage: chain.boundary()
2*[0,0] x [0,0] - 3*[1,1] x [0,0] + [1,1] x [1,1]
```

is_boundary()

Test whether the chain is a boundary

OUTPUT:

Boolean. Whether the chain is the *boundary()* of a chain in one degree higher.

EXAMPLES:

```
sage: square = cubical_complexes.Cube(2)
sage: C1 = square.n_chains(1, QQ)
sage: from sage.topology.cubical_complex import Cube
sage: chain = C1(Cube([[1, 1], [0, 1]])) - C1(Cube([[0, 1], [0, 0]]))
sage: chain.is_boundary()
False
```

is_cycle()

Test whether the chain is a cycle

OUTPUT:

Boolean. Whether the *boundary()* vanishes.

EXAMPLES:

```
sage: square = cubical_complexes.Cube(2)
sage: C1 = square.n_chains(1, QQ)
sage: from sage.topology.cubical_complex import Cube
sage: chain = C1(Cube([[1, 1], [0, 1]])) - C1(Cube([[0, 1], [0, 0]]))
sage: chain.is_cycle()
False
```

to_complex()

Return the corresponding chain complex element

OUTPUT:

An element of the chain complex, see *sage.homology.chain_complex*.

EXAMPLES:

```
sage: square = cubical_complexes.Cube(2)
sage: C1 = square.n_chains(1, QQ)
sage: from sage.topology.cubical_complex import Cube
sage: chain = C1(Cube([[1, 1], [0, 1]]))
sage: chain.to_complex()
Chain(1:(0, 0, 0, 1))
sage: ascii_art(_)
   d_0  [0]  d_1  [0]  d_2       d_3
0 <---- [0] <---- [0] <---- [0] <---- 0
        [0]       [0]
        [0]       [1]
```

chain_complex()

Return the chain complex.

OUTPUT:

Chain complex, see *sage.homology.chain_complex*.

EXAMPLES:

```
sage: square = cubical_complexes.Cube(2)
sage: CC = square.n_chains(2, QQ).chain_complex(); CC
Chain complex with at most 3 nonzero terms over Rational Field
sage: ascii_art(CC)
            [-1 -1  0  0]       [-1]
            [ 1  0 -1  0]       [ 1]
            [ 0  1  0 -1]       [-1]
            [ 0  0  1  1]       [ 1]
 0 <-- C_0 <-------------- C_1 <----- C_2 <-- 0
```

dual()

Return the cochains.

OUTPUT:

The cochains of the same cells with the same base ring.

EXAMPLES:

```
sage: square = cubical_complexes.Cube(2)
sage: chains = square.n_chains(1, ZZ);  chains
Free module generated by {[0,0] x [0,1], [0,1] x [0,0], [0,1] x [1,1], [1,1]␣
↪x [0,1]} over Integer Ring
sage: chains.dual()
Free module generated by {[0,0] x [0,1], [0,1] x [0,0], [0,1] x [1,1], [1,1]␣
↪x [0,1]} over Integer Ring
sage: type(chains)
<class 'sage.homology.chains.Chains_with_category'>
sage: type(chains.dual())
<class 'sage.homology.chains.Cochains_with_category'>
```

class sage.homology.chains.**Cochains**(*cell_complex*, *degree*, *cells=None*, *base_ring=None*)

Bases: *CellComplexReference*, CombinatorialFreeModule

Class for the free module of cochains in a given degree.

INPUT:

- n_cells – tuple of n-cells, which thus forms a basis for this module
- base_ring – optional (default **Z**)

One difference between chains and cochains is notation. In a simplicial complex, for example, a simplex (0,1,2) is written as "(0,1,2)" in the group of chains but as "\chi_(0,1,2)" in the group of cochains.

Also, since the free modules of chains and cochains are dual, there is a pairing $\langle c, z \rangle$, sending a cochain c and a chain z to a scalar.

EXAMPLES:

```
sage: S2 = simplicial_complexes.Sphere(2)
sage: C_2 = S2.n_chains(1)
sage: C_2_co = S2.n_chains(1, cochains=True)
sage: x = C_2.basis()[Simplex((0,2))]
sage: y = C_2.basis()[Simplex((1,3))]
sage: z = x+2*y
sage: a = C_2_co.basis()[Simplex((1,3))]
sage: b = C_2_co.basis()[Simplex((0,3))]
sage: c = 3*a-2*b
sage: z
```

(continues on next page)

(continued from previous page)

```
(0, 2) + 2*(1, 3)
sage: c
-2*\chi_(0, 3) + 3*\chi_(1, 3)
sage: c.eval(z)
6
```

class Element

Bases: IndexedFreeModuleElement

coboundary()

Return the coboundary of this cochain

OUTPUT:

The coboundary as a cochain in one degree higher.

EXAMPLES:

```
sage: square = cubical_complexes.Cube(2)
sage: C1 = square.n_chains(1, QQ, cochains=True)
sage: from sage.topology.cubical_complex import Cube
sage: cochain = C1(Cube([[1, 1], [0, 1]])) - 2 * C1(Cube([[0, 1], [0,␣
↪0]]))
sage: cochain
-2*\chi_[0,1] x [0,0] + \chi_[1,1] x [0,1]
sage: cochain.coboundary()
-\chi_[0,1] x [0,1]
```

cup_product(*cochain*)

Return the cup product with another cochain.

INPUT:

- cochain – cochain over the same cell complex

EXAMPLES:

```
sage: T2 = simplicial_complexes.Torus()
sage: C1 = T2.n_chains(1, base_ring=ZZ, cochains=True)
sage: def l(i, j):
....:     return C1(Simplex([i, j]))
sage: l1 = l(1, 3) + l(1, 4) + l(1, 6) + l(2, 4) - l(4, 5) + l(5, 6)
sage: l2 = l(1, 6) - l(2, 3) - l(2, 5) + l(3, 6) - l(4, 5) + l(5, 6)
```

The two one-cocycles are cohomology generators:

```
sage: l1.is_cocycle(), l1.is_coboundary()
(True, False)
sage: l2.is_cocycle(), l2.is_coboundary()
(True, False)
```

Their cup product is a two-cocycle that is again non-trivial in cohomology:

```
sage: l12 = l1.cup_product(l2)
sage: l12
\chi_(1, 3, 6) - \chi_(2, 4, 5) - \chi_(4, 5, 6)
sage: l1.parent().degree(), l2.parent().degree(), l12.parent().degree()
(1, 1, 2)
sage: l12.is_cocycle(), l12.is_coboundary()
(True, False)
```

eval(*other*)

Evaluate this cochain on the chain `other`.

INPUT:

- `other` – a chain for the same cell complex in the same dimension with the same base ring

OUTPUT: scalar

EXAMPLES:

```
sage: S2 = simplicial_complexes.Sphere(2)
sage: C_2 = S2.n_chains(1)
sage: C_2_co = S2.n_chains(1, cochains=True)
sage: x = C_2.basis()[Simplex((0,2))]
sage: y = C_2.basis()[Simplex((1,3))]
sage: z = x+2*y
sage: a = C_2_co.basis()[Simplex((1,3))]
sage: b = C_2_co.basis()[Simplex((0,3))]
sage: c = 3*a-2*b
sage: z
(0, 2) + 2*(1, 3)
sage: c
-2*\chi_(0, 3) + 3*\chi_(1, 3)
sage: c.eval(z)
6
```

is_coboundary()

Test whether the cochain is a coboundary

OUTPUT:

Boolean. Whether the cochain is the *coboundary()* of a cochain in one degree lower.

EXAMPLES:

```
sage: square = cubical_complexes.Cube(2)
sage: C1 = square.n_chains(1, QQ, cochains=True)
sage: from sage.topology.cubical_complex import Cube
sage: cochain = C1(Cube([[1, 1], [0, 1]])) - C1(Cube([[0, 1], [0, 0]]))
sage: cochain.is_coboundary()
True
```

is_cocycle()

Test whether the cochain is a cocycle

OUTPUT:

Boolean. Whether the *coboundary()* vanishes.

EXAMPLES:

```
sage: square = cubical_complexes.Cube(2)
sage: C1 = square.n_chains(1, QQ, cochains=True)
sage: from sage.topology.cubical_complex import Cube
sage: cochain = C1(Cube([[1, 1], [0, 1]])) - C1(Cube([[0, 1], [0, 0]]))
sage: cochain.is_cocycle()
True
```

to_complex()

Return the corresponding cochain complex element

OUTPUT:

An element of the cochain complex, see *sage.homology.chain_complex*.

EXAMPLES:

```
sage: square = cubical_complexes.Cube(2)
sage: C1 = square.n_chains(1, QQ, cochains=True)
sage: from sage.topology.cubical_complex import Cube
sage: cochain = C1(Cube([[1, 1], [0, 1]]))
sage: cochain.to_complex()
Chain(1:(0, 0, 0, 1))
sage: ascii_art(_)
   d_2       d_1  [0]  d_0  [0]  d_-1
0 <---- [0] <---- [0] <---- [0] <----- 0
                  [0]       [0]
                  [1]       [0]
```

cochain_complex()

Return the cochain complex.

OUTPUT:

Cochain complex, see *sage.homology.chain_complex*.

EXAMPLES:

```
sage: square = cubical_complexes.Cube(2)
sage: C2 = square.n_chains(2, QQ, cochains=True)
sage: C2.cochain_complex()
Chain complex with at most 3 nonzero terms over Rational Field
sage: ascii_art(C2.cochain_complex())
                                [-1  1  0  0]
                                [-1  0  1  0]
                                [ 0 -1  0  1]
            [-1  1 -1  1]       [ 0  0 -1  1]
 0 <-- C_2 <-------------- C_1 <-------------- C_0 <-- 0
```

dual()

Return the chains

OUTPUT:

The chains of the same cells with the same base ring.

EXAMPLES:

```
sage: square = cubical_complexes.Cube(2)
sage: cochains = square.n_chains(1, ZZ, cochains=True);  cochains
Free module generated by {[0,0] x [0,1], [0,1] x [0,0], [0,1] x [1,1], [1,1]␣
↪x [0,1]} over Integer Ring
sage: cochains.dual()
Free module generated by {[0,0] x [0,1], [0,1] x [0,0], [0,1] x [1,1], [1,1]␣
↪x [0,1]} over Integer Ring
sage: type(cochains)
<class 'sage.homology.chains.Cochains_with_category'>
sage: type(cochains.dual())
<class 'sage.homology.chains.Chains_with_category'>
```

CHAPTER

THREE

MORPHISMS OF CHAIN COMPLEXES

AUTHORS:

- Benjamin Antieau <d.ben.antieau@gmail.com> (2009.06)
- Travis Scrimshaw (2012-08-18): Made all simplicial complexes immutable to work with the homset cache.

This module implements morphisms of chain complexes. The input is a dictionary whose keys are in the grading group of the chain complex and whose values are matrix morphisms.

EXAMPLES:

```
sage: # needs sage.graphs
sage: S = simplicial_complexes.Sphere(1); S
Minimal triangulation of the 1-sphere
sage: C = S.chain_complex()
sage: C.differential()
{0: [], 1: [-1 -1  0]
 [ 1  0 -1]
 [ 0  1  1], 2: []}
sage: f = {0: zero_matrix(ZZ,3,3), 1: zero_matrix(ZZ,3,3)}
sage: G = Hom(C, C)
sage: x = G(f); x
Chain complex endomorphism of
 Chain complex with at most 2 nonzero terms over Integer Ring
sage: x._matrix_dictionary
{0: [0 0 0]
    [0 0 0]
    [0 0 0],
 1: [0 0 0]
    [0 0 0]
    [0 0 0]}
```

class sage.homology.chain_complex_morphism.**ChainComplexMorphism**(*matrices*, *C*, *D*, *check=True*)

Bases: Morphism

An element of this class is a morphism of chain complexes.

dual()

The dual chain map to this one.

That is, the map from the dual of the codomain of this one to the dual of its domain, represented in each degree by the transpose of the corresponding matrix.

EXAMPLES:

```
sage: # needs sage.graphs
sage: X = simplicial_complexes.Simplex(1)
sage: Y = simplicial_complexes.Simplex(0)
sage: g = Hom(X,Y)({0:0, 1:0})
sage: f = g.associated_chain_complex_morphism()
sage: f.in_degree(0)
[1 1]
sage: f.dual()
Chain complex morphism:
  From: Chain complex with at most 1 nonzero terms over Integer Ring
    To: Chain complex with at most 2 nonzero terms over Integer Ring
sage: f.dual().in_degree(0)
[1]
[1]
sage: ascii_art(f.domain())
            [-1]
            [ 1]
 0 <-- C_0 <----- C_1 <-- 0
sage: ascii_art(f.dual().codomain())
            [-1  1]
 0 <-- C_1 <-------- C_0 <-- 0
```

in_degree(*n*)

The matrix representing this morphism in degree n.

INPUT:

- `n` – degree

EXAMPLES:

```
sage: C = ChainComplex({0: identity_matrix(ZZ, 1)})
sage: D = ChainComplex({0: zero_matrix(ZZ, 1), 1: zero_matrix(ZZ, 1)})
sage: f = Hom(C,D)({0: identity_matrix(ZZ, 1), 1: zero_matrix(ZZ, 1)})
sage: f.in_degree(0)
[1]
```

Note that if the matrix is not specified in the definition of the map, it is assumed to be zero:

```
sage: f.in_degree(2)
[]
sage: f.in_degree(2).nrows(), f.in_degree(2).ncols()
(1, 0)
sage: C.free_module(2)
Ambient free module of rank 0 over the principal ideal domain Integer Ring
sage: D.free_module(2)
Ambient free module of rank 1 over the principal ideal domain Integer Ring
```

is_identity()

Return `True` if this is the identity map.

EXAMPLES:

```
sage: # needs sage.graphs
sage: S = SimplicialComplex(is_mutable=False)
sage: H = Hom(S,S)
sage: i = H.identity()
sage: x = i.associated_chain_complex_morphism()
```

(continues on next page)

(continued from previous page)

```
sage: x.is_identity()
True
```

is_injective()

Return `True` if this map is injective.

EXAMPLES:

```
sage: # needs sage.graphs
sage: S1 = simplicial_complexes.Sphere(1)
sage: H = Hom(S1, S1)
sage: flip = H({0:0, 1:2, 2:1})
sage: flip.associated_chain_complex_morphism().is_injective()
True

sage: # needs sage.graphs
sage: pt = simplicial_complexes.Simplex(0)
sage: inclusion = Hom(pt, S1)({0:2})
sage: inclusion.associated_chain_complex_morphism().is_injective()
True
sage: inclusion.associated_chain_complex_morphism(cochain=True).is_injective()
False
```

is_surjective()

Return `True` if this map is surjective.

EXAMPLES:

```
sage: # needs sage.graphs
sage: S1 = simplicial_complexes.Sphere(1)
sage: H = Hom(S1, S1)
sage: flip = H({0:0, 1:2, 2:1})
sage: flip.associated_chain_complex_morphism().is_surjective()
True

sage: # needs sage.graphs
sage: pt = simplicial_complexes.Simplex(0)
sage: inclusion = Hom(pt, S1)({0:2})
sage: inclusion.associated_chain_complex_morphism().is_surjective()
False
sage: inclusion.associated_chain_complex_morphism(cochain=True).is_
↪surjective()
True
```

to_matrix(*deg=None*)

The matrix representing this chain map.

If the degree `deg` is specified, return the matrix in that degree; otherwise, return the (block) matrix for the whole chain map.

INPUT:

- `deg` – (optional, default `None`) the degree

EXAMPLES:

```
sage: C = ChainComplex({0: identity_matrix(ZZ, 1)})
sage: D = ChainComplex({0: zero_matrix(ZZ, 1), 1: zero_matrix(ZZ, 1)})
```

(continues on next page)

(continued from previous page)

```
sage: f = Hom(C,D)({0: identity_matrix(ZZ, 1), 1: zero_matrix(ZZ, 1)})
sage: f.to_matrix(0)
[1]
sage: f.to_matrix()
[1|0|]
[-+-+]
[0|0|]
[-+-+]
[0|0|]
```

sage.homology.chain_complex_morphism.**is_ChainComplexMorphism**(*x*)

Return `True` if and only if `x` is a chain complex morphism.

EXAMPLES:

```
sage: # needs sage.graphs
sage: from sage.homology.chain_complex_morphism import is_ChainComplexMorphism
sage: S = simplicial_complexes.Sphere(14)
sage: H = Hom(S,S)
sage: i = H.identity()                    # long time (8s on sage.math, 2011)
sage: S = simplicial_complexes.Sphere(6)
sage: H = Hom(S,S)
sage: i = H.identity()
sage: x = i.associated_chain_complex_morphism()
sage: x # indirect doctest
Chain complex morphism:
  From: Chain complex with at most 7 nonzero terms over Integer Ring
  To: Chain complex with at most 7 nonzero terms over Integer Ring
sage: is_ChainComplexMorphism(x)
True
```

CHAPTER

FOUR

CHAIN HOMOTOPIES AND CHAIN CONTRACTIONS

Chain homotopies are standard constructions in homological algebra: given chain complexes C and D and chain maps $f, g : C \to D$, say with differential of degree -1, a *chain homotopy* H between f and g is a collection of maps $H_n : C_n \to D_{n+1}$ satisfying

$$\partial_D H + H\partial_C = f - g.$$

The presence of a chain homotopy defines an equivalence relation (*chain homotopic*) on chain maps. If f and g are chain homotopic, then one can show that f and g induce the same map on homology.

Chain contractions are not as well known. The papers [MAR2009], [RMA2009], and [PR2015] provide some references. Given two chain complexes C and D, a *chain contraction* is a chain homotopy $H : C \to C$ for which there are chain maps $\pi : C \to D$ ("projection") and $\iota : D \to C$ ("inclusion") such that

- H is a chain homotopy between 1_C and $\iota\pi$,
- $\pi\iota = 1_D$,
- $\pi H = 0$,
- $H\iota = 0$,
- $HH = 0$.

Such a chain homotopy provides a strong relation between the chain complexes C and D; for example, their homology groups are isomorphic.

class sage.homology.chain_homotopy.**ChainContraction**(*matrices*, *pi*, *iota*)

Bases: *ChainHomotopy*

A chain contraction.

An algebraic gradient vector field $H : C \to C$ (that is a chain homotopy satisfying $HH = 0$) for which there are chain maps $\pi : C \to D$ ("projection") and $\iota : D \to C$ ("inclusion") such that

- H is a chain homotopy between 1_C and $\iota\pi$,
- $\pi\iota = 1_D$,
- $\pi H = 0$,
- $H\iota = 0$.

`H` is defined by a dictionary `matrices` of matrices.

INPUT:

- `matrices` – dictionary of matrices, keyed by dimension
- `pi` – a chain map $C \to D$
- `iota` – a chain map $D \to C$

EXAMPLES:

```
sage: from sage.homology.chain_homotopy import ChainContraction
sage: C = ChainComplex({0: zero_matrix(ZZ, 1), 1: identity_matrix(ZZ, 1)})
sage: D = ChainComplex({0: matrix(ZZ, 0, 1)})
```

The chain complex C is chain homotopy equivalent to D, which is just a copy of $\mathbf{Z}$ in degree 0, and we construct a chain contraction:

```
sage: pi = Hom(C,D)({0: identity_matrix(ZZ, 1)})
sage: iota = Hom(D,C)({0: identity_matrix(ZZ, 1)})
sage: H = ChainContraction({0: zero_matrix(ZZ, 0, 1),
....:                       1: zero_matrix(ZZ, 1),
....:                       2: identity_matrix(ZZ, 1)}, pi, iota)
```

dual()

The chain contraction dual to this one.

This is useful when switching from homology to cohomology.

EXAMPLES:

```
sage: S2 = simplicial_complexes.Sphere(2)                                    #␣
↪needs sage.graphs
sage: phi, M = S2.algebraic_topological_model(QQ)                            #␣
↪needs sage.graphs
sage: phi.iota()                                                             #␣
↪needs sage.graphs
Chain complex morphism:
  From: Chain complex with at most 3 nonzero terms over Rational Field
  To: Chain complex with at most 3 nonzero terms over Rational Field
```

Lifting the degree zero homology class gives a single vertex, but the degree zero cohomology class needs to be detected on every vertex, and vice versa for degree 2:

```
sage: # needs sage.graphs
sage: phi.iota().in_degree(0)
[0]
[0]
[0]
[1]
sage: phi.dual().iota().in_degree(0)
[1]
[1]
[1]
[1]
sage: phi.iota().in_degree(2)
[-1]
[ 1]
[-1]
[ 1]
sage: phi.dual().iota().in_degree(2)
[0]
[0]
[0]
[1]
```

iota()

The chain map ι associated to this chain contraction.

EXAMPLES:

```
sage: S2 = simplicial_complexes.Sphere(2)                                   #␣
↪needs sage.graphs
sage: phi, M = S2.algebraic_topological_model(QQ)                           #␣
↪needs sage.graphs
sage: phi.iota()                                                            #␣
↪needs sage.graphs
Chain complex morphism:
  From: Chain complex with at most 3 nonzero terms over Rational Field
  To: Chain complex with at most 3 nonzero terms over Rational Field
```

Lifting the degree zero homology class gives a single vertex:

```
sage: phi.iota().in_degree(0)                                               #␣
↪needs sage.graphs
[0]
[0]
[0]
[1]
```

Lifting the degree two homology class gives the signed sum of all of the 2-simplices:

```
sage: phi.iota().in_degree(2)                                               #␣
↪needs sage.graphs
[-1]
[ 1]
[-1]
[ 1]
```

pi()

The chain map π associated to this chain contraction.

EXAMPLES:

```
sage: # needs sage.graphs
sage: S2 = simplicial_complexes.Sphere(2)
sage: phi, M = S2.algebraic_topological_model(QQ)
sage: phi.pi()
Chain complex morphism:
  From: Chain complex with at most 3 nonzero terms over Rational Field
  To: Chain complex with at most 3 nonzero terms over Rational Field
sage: phi.pi().in_degree(0)  # Every vertex represents a homology class.
[1 1 1 1]
sage: phi.pi().in_degree(1)  # No homology in degree 1.
[]
```

The degree 2 homology generator is detected on a single simplex:

```
sage: phi.pi().in_degree(2)                                                 #␣
↪needs sage.graphs
[0 0 0 1]
```

class sage.homology.chain_homotopy.**ChainHomotopy**(*matrices*, *f*, *g=None*)

Bases: Morphism

A chain homotopy.

A chain homotopy H between chain maps $f, g : C \to D$ is a sequence of maps $H_n : C_n \to D_{n+1}$ (if the chain complexes are graded homologically) satisfying

$$\partial_D H + H\partial_C = f - g.$$

INPUT:

- `matrices` – dictionary of matrices, keyed by dimension
- `f` – chain map $C \to D$
- `g` (optional) – chain map $C \to D$

The dictionary `matrices` defines `H` by specifying the matrix defining it in each degree: the entry m corresponding to key i gives the linear transformation $C_i \to D_{i+1}$.

If f is specified but not g, then g can be recovered from the defining formula. That is, if g is not specified, then it is defined to be $f - \partial_D H - H\partial_C$.

Note that the degree of the differential on the chain complex C must agree with that for D, and those degrees determine the "degree" of the chain homotopy map: if the degree of the differential is d, then the chain homotopy consists of a sequence of maps $C_n \to C_{n-d}$. The keys in the dictionary `matrices` specify the starting degrees.

EXAMPLES:

```
sage: from sage.homology.chain_homotopy import ChainHomotopy
sage: C = ChainComplex({0: identity_matrix(ZZ, 1)})
sage: D = ChainComplex({0: zero_matrix(ZZ, 1)})
sage: f = Hom(C,D)({0: identity_matrix(ZZ, 1), 1: zero_matrix(ZZ, 1)})
sage: g = Hom(C,D)({0: zero_matrix(ZZ, 1), 1: zero_matrix(ZZ, 1)})
sage: H = ChainHomotopy({0: zero_matrix(ZZ, 0, 1), 1: identity_matrix(ZZ, 1)}, f,␣
↪g)
```

Note that the maps f and g are stored in the attributes `H._f` and `H._g`:

```
sage: H._f
Chain complex morphism:
  From: Chain complex with at most 2 nonzero terms over Integer Ring
  To: Chain complex with at most 2 nonzero terms over Integer Ring
sage: H._f.in_degree(0)
[1]
sage: H._g.in_degree(0)
[0]
```

A non-example:

```
sage: H = ChainHomotopy({0: zero_matrix(ZZ, 0, 1), 1: zero_matrix(ZZ, 1)}, f, g)
Traceback (most recent call last):
...
ValueError: the data do not define a valid chain homotopy
```

dual()

Dual chain homotopy to this one.

That is, if this one is a chain homotopy between chain maps $f, g : C \to D$, then its dual is a chain homotopy between the dual of f and the dual of g, from D^* to C^*. It is represented in each degree by the transpose of the corresponding matrix.

EXAMPLES:

```
sage: from sage.homology.chain_homotopy import ChainHomotopy
sage: C = ChainComplex({1: matrix(ZZ, 0, 2)})  # one nonzero term in degree 1
sage: D = ChainComplex({0: matrix(ZZ, 0, 1)})  # one nonzero term in degree 0
sage: f = Hom(C, D)({})
sage: H = ChainHomotopy({1: matrix(ZZ, 1, 2, (3,1))}, f, f)
sage: H.in_degree(1)
[3 1]
sage: H.dual().in_degree(0)
[3]
[1]
```

in_degree(*n*)

The matrix representing this chain homotopy in degree n.

INPUT:

- n – degree

EXAMPLES:

```
sage: from sage.homology.chain_homotopy import ChainHomotopy
sage: C = ChainComplex({1: matrix(ZZ, 0, 2)})  # one nonzero term in degree 1
sage: D = ChainComplex({0: matrix(ZZ, 0, 1)})  # one nonzero term in degree 0
sage: f = Hom(C, D)({})
sage: H = ChainHomotopy({1: matrix(ZZ, 1, 2, (3,1))}, f, f)
sage: H.in_degree(1)
[3 1]
```

This returns an appropriately sized zero matrix if the chain homotopy is not defined in degree n:

```
sage: H.in_degree(-3)
[]
```

is_algebraic_gradient_vector_field()

An algebraic gradient vector field is a linear map $H : C \to C$ such that $HH = 0$.

(Some authors also require that $H\partial H = H$, whereas some make this part of the definition of "homology gradient vector field. We have made the second choice.) See Molina-Abril and Réal [MAR2009] and Réal and Molina-Abril [RMA2009] for this and related terminology.

See also *is_homology_gradient_vector_field()*.

EXAMPLES:

```
sage: from sage.homology.chain_homotopy import ChainHomotopy
sage: C = ChainComplex({0: zero_matrix(ZZ, 1), 1: identity_matrix(ZZ, 1)})
```

The chain complex C is chain homotopy equivalent to a copy of $\mathbf{Z}$ in degree 0. Two chain maps $C \to C$ will be chain homotopic as long as they agree in degree 0.

```
sage: f = Hom(C,C)({0: identity_matrix(ZZ, 1),
....:               1: matrix(ZZ, 1, 1, [3]),
....:               2: matrix(ZZ, 1, 1, [3])})
sage: g = Hom(C,C)({0: identity_matrix(ZZ, 1),
....:               1: matrix(ZZ, 1, 1, [2]),
....:               2: matrix(ZZ, 1, 1, [2])})
sage: H = ChainHomotopy({0: zero_matrix(ZZ, 0, 1),
....:                    1: zero_matrix(ZZ, 1),
```

(continues on next page)

(continued from previous page)

```
....:                     2: identity_matrix(ZZ, 1)}, f, g)
sage: H.is_algebraic_gradient_vector_field()
True
```

A chain homotopy which is not an algebraic gradient vector field:

```
sage: H = ChainHomotopy({0: zero_matrix(ZZ, 0, 1),
....:                    1: identity_matrix(ZZ, 1),
....:                    2: identity_matrix(ZZ, 1)}, f, g)
sage: H.is_algebraic_gradient_vector_field()
False
```

is_homology_gradient_vector_field()

A homology gradient vector field is an algebraic gradient vector field $H : C \to C$ (i.e., a chain homotopy satisfying $HH = 0$) such that $\partial H \partial = \partial$ and $H \partial H = H$.

See Molina-Abril and Réal [MAR2009] and Réal and Molina-Abril [RMA2009] for this and related terminology.

See also *is_algebraic_gradient_vector_field()*.

EXAMPLES:

```
sage: from sage.homology.chain_homotopy import ChainHomotopy
sage: C = ChainComplex({0: zero_matrix(ZZ, 1), 1: identity_matrix(ZZ, 1)})

sage: f = Hom(C,C)({0: identity_matrix(ZZ, 1),
....:               1: matrix(ZZ, 1, 1, [3]),
....:               2: matrix(ZZ, 1, 1, [3])})
sage: g = Hom(C,C)({0: identity_matrix(ZZ, 1),
....:               1: matrix(ZZ, 1, 1, [2]),
....:               2: matrix(ZZ, 1, 1, [2])})
sage: H = ChainHomotopy({0: zero_matrix(ZZ, 0, 1),
....:                    1: zero_matrix(ZZ, 1),
....:                    2: identity_matrix(ZZ, 1)}, f, g)
sage: H.is_homology_gradient_vector_field()
True
```

CHAPTER

FIVE

HOMSPACES BETWEEN CHAIN COMPLEXES

Note that some significant functionality is lacking. Namely, the homspaces are not actually modules over the base ring. It will be necessary to enrich some of the structure of chain complexes for this to be naturally available. On other hand, there are various overloaded operators. __mul__ acts as composition. One can __add__, and one can __mul__ with a ring element on the right.

EXAMPLES:

```
sage: S = simplicial_complexes.Sphere(2)
sage: T = simplicial_complexes.Torus()
sage: C = S.chain_complex(augmented=True, cochain=True)
sage: D = T.chain_complex(augmented=True, cochain=True)
sage: G = Hom(C, D); G
Set of Morphisms
 from Chain complex with at most 4 nonzero terms over Integer Ring
   to Chain complex with at most 4 nonzero terms over Integer Ring
   in Category of chain complexes over Integer Ring

sage: S = simplicial_complexes.ChessboardComplex(3, 3)
sage: H = Hom(S,S)
sage: i = H.identity()
sage: x = i.associated_chain_complex_morphism(augmented=True); x
Chain complex morphism:
  From: Chain complex with at most 4 nonzero terms over Integer Ring
  To:   Chain complex with at most 4 nonzero terms over Integer Ring
sage: x._matrix_dictionary
{-1: [1],
  0: [1 0 0 0 0 0 0 0 0]
     [0 1 0 0 0 0 0 0 0]
     [0 0 1 0 0 0 0 0 0]
     [0 0 0 1 0 0 0 0 0]
     [0 0 0 0 1 0 0 0 0]
     [0 0 0 0 0 1 0 0 0]
     [0 0 0 0 0 0 1 0 0]
     [0 0 0 0 0 0 0 1 0]
     [0 0 0 0 0 0 0 0 1],
  1: [1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
     [0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
     [0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
     [0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0]
     [0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0]
     [0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0]
     [0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0]
     [0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0]
     [0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0]
```

(continues on next page)

(continued from previous page)

```
     [0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0]
     [0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0]
     [0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0]
     [0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0]
     [0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0]
     [0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0]
     [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0]
     [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0]
     [0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1],
  2: [1 0 0 0 0 0]
     [0 1 0 0 0 0]
     [0 0 1 0 0 0]
     [0 0 0 1 0 0]
     [0 0 0 0 1 0]
     [0 0 0 0 0 1]}

sage: S = simplicial_complexes.Sphere(2)
sage: A = Hom(S, S)
sage: i = A.identity()
sage: x = i.associated_chain_complex_morphism(); x
Chain complex morphism:
  From: Chain complex with at most 3 nonzero terms over Integer Ring
  To: Chain complex with at most 3 nonzero terms over Integer Ring
sage: y = x*4
sage: z = y*y
sage: y + z
Chain complex morphism:
  From: Chain complex with at most 3 nonzero terms over Integer Ring
  To: Chain complex with at most 3 nonzero terms over Integer Ring
sage: f = x._matrix_dictionary
sage: C = S.chain_complex()
sage: G = Hom(C, C)
sage: w = G(f)
sage: w == x
True
```

class sage.homology.chain_complex_homspace.**ChainComplexHomspace**(*X*, *Y*, *category=None*, *base=None*, *check=True*)

Bases: Homset

Class of homspaces of chain complex morphisms.

EXAMPLES:

```
sage: T = SimplicialComplex([[1,2,3,4],[7,8,9]])
sage: C = T.chain_complex(augmented=True, cochain=True)
sage: G = Hom(C, C)
sage: G
Set of Morphisms
 from Chain complex with at most 5 nonzero terms over Integer Ring
   to Chain complex with at most 5 nonzero terms over Integer Ring
   in Category of chain complexes over Integer Ring
```

sage.homology.chain_complex_homspace.**is_ChainComplexHomspace**(*x*)

Return `True` if and only if `x` is a morphism of chain complexes.

EXAMPLES:

```
sage: from sage.homology.chain_complex_homspace import is_ChainComplexHomspace
sage: T = SimplicialComplex([[1,2,3,4],[7,8,9]])
sage: C = T.chain_complex(augmented=True, cochain=True)
sage: G = Hom(C, C)
sage: is_ChainComplexHomspace(G)
True
```

CHAPTER

SIX

KOSZUL COMPLEXES

class sage.homology.koszul_complex.**KoszulComplex**(*R*, *elements*)

Bases: *ChainComplex_class*, UniqueRepresentation

A Koszul complex.

Let R be a ring and consider $x_1, x_2, \ldots, x_n \in R$. The *Koszul complex* $K_*(x_1, \ldots, x_n)$ is given by defining a chain complex structure on the exterior algebra $\bigwedge^n R$ with the basis $e_{i_1} \wedge \cdots \wedge e_{i_a}$. The differential is given by

$$\partial(e_{i_1} \wedge \cdots \wedge e_{i_a}) = \sum_{r=1}^{a} (-1)^{r-1} x_{i_r} e_{i_1} \wedge \cdots \wedge \hat{e}_{i_r} \wedge \cdots \wedge e_{i_a},$$

where $\hat{e}_{i_r}$ denotes the omitted factor.

Alternatively we can describe the Koszul complex by considering the basic complex K_{x_i}

$$0 \to R \xrightarrow{x_i} R \to 0.$$

Then the Koszul complex is given by $K_*(x_1, \ldots, x_n) = \bigotimes_i K_{x_i}$.

INPUT:

- `R` – the base ring
- `elements` – a tuple of elements of `R`

EXAMPLES:

```
sage: R.<x,y,z> = QQ[]
sage: K = KoszulComplex(R, [x,y])
sage: ascii_art(K)
                        [-y]
            [x y]       [ x]
 0 <-- C_0 <------ C_1 <----- C_2 <-- 0
sage: K = KoszulComplex(R, [x,y,z])
sage: ascii_art(K)
                          [-y -z  0]       [ z]
                          [ x  0 -z]       [-y]
            [x y z]       [ 0  x  y]       [ x]
 0 <-- C_0 <-------- C_1 <---------- C_2 <----- C_3 <-- 0
sage: K = KoszulComplex(R, [x+y*z,x+y-z])
sage: ascii_art(K)
                                          [-x - y + z]
            [  y*z + x x + y - z]         [   y*z + x]
 0 <-- C_0 <---------------------- C_1 <------------- C_2 <-- 0
```

REFERENCES:

- Wikipedia article Koszul_complex

CHAPTER

SEVEN

HOCHSCHILD COMPLEXES

class sage.homology.hochschild_complex.**HochschildComplex**(A, M)

Bases: UniqueRepresentation, Parent

The Hochschild complex.

Let A be an algebra over a commutative ring R such that A a projective R-module, and M an A-bimodule. The *Hochschild complex* is the chain complex given by

$$C_n(A, M) := M \otimes A^{\otimes n}$$

with the boundary operators given as follows. For fixed n, define the face maps

$$f_{n,i}(m \otimes a_1 \otimes \cdots \otimes a_n) = \begin{cases} ma_1 \otimes \cdots \otimes a_n & \text{if } i = 0, \\ a_n m \otimes a_1 \otimes \cdots \otimes a_{n-1} & \text{if } i = n, \\ m \otimes a_1 \otimes \cdots \otimes a_i a_{i+1} \otimes \cdots \otimes a_n & \text{otherwise.} \end{cases}$$

We define the boundary operators as

$$d_n = \sum_{i=0}^{n} (-1)^i f_{n,i}.$$

The *Hochschild homology* of A is the homology of this complex. Alternatively, the Hochschild homology can be described by $HH_n(A, M) = \mathrm{Tor}_n^{A^e}(A, M)$, where $A^e = A \otimes A^o$ (A^o is the opposite algebra of A) is the enveloping algebra of A.

Hochschild cohomology is the homology of the dual complex and can be described by $HH^n(A, M) = \mathrm{Ext}_{A^e}^n(A, M)$.

Another perspective on Hochschild homology is that $f_{n,i}$ make the family $C_n(A, M)$ a simplicial object in the category of R-modules, and the degeneracy maps are

$$s_i(a_0 \otimes \cdots \otimes a_n) = a_0 \otimes \cdots \otimes a_i \otimes 1 \otimes a_{i+1} \otimes \cdots \otimes a_n$$

The Hochschild homology can also be constructed as the homology of this simplicial module.

REFERENCES:

- Wikipedia article Hochschild_homology
- https://ncatlab.org/nlab/show/Hochschild+cohomology
- [Red2001]

class Element(*parent*, *vectors*)

Bases: ModuleElement

A chain of the Hochschild complex.

INPUT:

Can be one of the following:

- A dictionary whose keys are the degree and whose d-th value is an element in the degree d module.
- An element in the coefficient module M.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 3)
sage: T = SGA.trivial_representation()
sage: H = SGA.hochschild_complex(T)
sage: H(T.an_element())
Chain(0: 2*B['v'])
sage: H({0: T.an_element()})
Chain(0: 2*B['v'])
sage: H({1: H.module(1).an_element()})
Chain(1: 2*B['v'] # [1, 2, 3] + 2*B['v'] # [1, 3, 2] + 3*B['v'] # [2, 1, 3])
sage: H({0: H.module(0).an_element(), 3: H.module(3).an_element()})
Chain with 2 nonzero terms over Rational Field

sage: F.<x,y> = FreeAlgebra(ZZ)
sage: H = F.hochschild_complex(F)
sage: H(x + 2*y^2)
Chain(0: F[x] + 2*F[y^2])
sage: H({0: x*y - x})
Chain(0: -F[x] + F[x*y])
sage: H(2)
Chain(0: 2*F[1])
sage: H({0: x-y, 2: H.module(2).basis().an_element()})
Chain with 2 nonzero terms over Integer Ring
```

vector(*degree*)

Return the free module element in `degree`.

EXAMPLES:

```
sage: F.<x,y> = FreeAlgebra(ZZ)
sage: H = F.hochschild_complex(F)
sage: a = H({0: x-y, 2: H.module(2).basis().an_element()})
sage: [a.vector(i) for i in range(3)]
[F[x] - F[y], 0, F[1] # F[1] # F[1]]
```

algebra()

Return the defining algebra of `self`.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 3)
sage: T = SGA.trivial_representation()
sage: H = SGA.hochschild_complex(T)
sage: H.algebra()
Symmetric group algebra of order 3 over Rational Field
```

boundary(*d*)

Return the boundary operator in degree `d`.

EXAMPLES:

```
sage: E.<x,y> = ExteriorAlgebra(QQ)
sage: H = E.hochschild_complex(E)
sage: d1 = H.boundary(1)
sage: z = d1.domain().an_element(); z
2*1 # 1 + 2*1 # x + 3*1 # y
sage: d1(z)
0
sage: d1.matrix()
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  2  0  0 -2  0  0  0  0  0  0]

sage: s = SymmetricFunctions(QQ).s()
sage: H = s.hochschild_complex(s)
sage: d1 = H.boundary(1)
sage: x = d1.domain().an_element(); x
2*s[] # s[] + 2*s[] # s[1] + 3*s[] # s[2]
sage: d1(x)
0
sage: y = tensor([s.an_element(), s.an_element()])
sage: d1(y)
0
sage: z = tensor([s[2,1] + s[3], s.an_element()])
sage: d1(z)
0
```

coboundary(*d*)

Return the coboundary morphism of degree `d`.

EXAMPLES:

```
sage: E.<x,y> = ExteriorAlgebra(QQ)
sage: H = E.hochschild_complex(E)
sage: del1 = H.coboundary(1)
sage: z = del1.domain().an_element(); z
2 + 2*x + 3*y
sage: del1(z)
0
sage: del1.matrix()
[ 0  0  0  0]
[ 0  0  0  0]
[ 0  0  0  0]
[ 0  0  0  0]
[ 0  0  0  0]
[ 0  0  0  0]
[ 0  0  0  2]
[ 0  0  0  0]
[ 0  0  0  0]
[ 0  0  0 -2]
[ 0  0  0  0]
[ 0  0  0  0]
[ 0  0  0  0]
```

(continues on next page)

(continued from previous page)

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

coefficients()

Return the coefficients of `self`.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 3)
sage: T = SGA.trivial_representation()
sage: H = SGA.hochschild_complex(T)
sage: H.coefficients()
Trivial representation of Standard permutations of 3 over Rational Field
```

cohomology(*d*)

Return the `d`-th cohomology group.

EXAMPLES:

```
sage: E.<x,y> = ExteriorAlgebra(QQ)
sage: H = E.hochschild_complex(E)
sage: H.cohomology(0)
Vector space of dimension 3 over Rational Field
sage: H.cohomology(1)
Vector space of dimension 4 over Rational Field
sage: H.cohomology(2)
Vector space of dimension 6 over Rational Field

sage: SGA = SymmetricGroupAlgebra(QQ, 3)
sage: T = SGA.trivial_representation()
sage: H = SGA.hochschild_complex(T)
sage: H.cohomology(0)
Vector space of dimension 1 over Rational Field
sage: H.cohomology(1)
Vector space of dimension 0 over Rational Field
sage: H.cohomology(2)
Vector space of dimension 0 over Rational Field
```

When working over general rings (except $\mathbf{Z}$) and we can construct a unitriangular basis for the image quotient, we fallback to a slower implementation using (combinatorial) free modules:

```
sage: R.<x,y> = QQ[]
sage: SGA = SymmetricGroupAlgebra(R, 2)
sage: T = SGA.trivial_representation()
sage: H = SGA.hochschild_complex(T)
sage: H.cohomology(1)
Free module generated by {} over Multivariate Polynomial Ring in x, y over␣
↪Rational Field
```

differential(*d*)

Return the boundary operator in degree `d`.

EXAMPLES:

```
sage: E.<x,y> = ExteriorAlgebra(QQ)
sage: H = E.hochschild_complex(E)
sage: d1 = H.boundary(1)
sage: z = d1.domain().an_element(); z
2*1 # 1 + 2*1 # x + 3*1 # y
sage: d1(z)
0
sage: d1.matrix()
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  2  0  0 -2  0  0  0  0  0  0]

sage: s = SymmetricFunctions(QQ).s()
sage: H = s.hochschild_complex(s)
sage: d1 = H.boundary(1)
sage: x = d1.domain().an_element(); x
2*s[] # s[] + 2*s[] # s[1] + 3*s[] # s[2]
sage: d1(x)
0
sage: y = tensor([s.an_element(), s.an_element()])
sage: d1(y)
0
sage: z = tensor([s[2,1] + s[3], s.an_element()])
sage: d1(z)
0
```

homology(*d*)

Return the d-th homology group.

EXAMPLES:

```
sage: E.<x,y> = ExteriorAlgebra(QQ)
sage: H = E.hochschild_complex(E)
sage: H.homology(0)
Vector space of dimension 3 over Rational Field
sage: H.homology(1)
Vector space of dimension 4 over Rational Field
sage: H.homology(2)
Vector space of dimension 6 over Rational Field

sage: SGA = SymmetricGroupAlgebra(QQ, 3)
sage: T = SGA.trivial_representation()
sage: H = SGA.hochschild_complex(T)
sage: H.homology(0)
Vector space of dimension 1 over Rational Field
sage: H.homology(1)
Vector space of dimension 0 over Rational Field
sage: H.homology(2)
Vector space of dimension 0 over Rational Field
```

When working over general rings (except **Z**) and we can construct a unitriangular basis for the image quotient, we fallback to a slower implementation using (combinatorial) free modules:

```
sage: R.<x,y> = QQ[]
sage: SGA = SymmetricGroupAlgebra(R, 2)
sage: T = SGA.trivial_representation()
```

(continues on next page)

(continued from previous page)

```
sage: H = SGA.hochschild_complex(T)
sage: H.homology(1)
Free module generated by {} over Multivariate Polynomial Ring in x, y over␣
↪Rational Field
```

module(*d*)

Return the module in degree `d`.

EXAMPLES:

```
sage: SGA = SymmetricGroupAlgebra(QQ, 3)
sage: T = SGA.trivial_representation()
sage: H = SGA.hochschild_complex(T)
sage: H.module(0)
Trivial representation of Standard permutations of 3 over Rational Field
sage: H.module(1)
Trivial representation of Standard permutations of 3 over Rational Field
 # Symmetric group algebra of order 3 over Rational Field
sage: H.module(2)
Trivial representation of Standard permutations of 3 over Rational Field
 # Symmetric group algebra of order 3 over Rational Field
 # Symmetric group algebra of order 3 over Rational Field
```

trivial_module()

Return the trivial module of `self`.

EXAMPLES:

```
sage: E.<x,y> = ExteriorAlgebra(QQ)
sage: H = E.hochschild_complex(E)
sage: H.trivial_module()
Free module generated by {} over Rational Field
```

CHAPTER

EIGHT

HOMOLOGY GROUPS

This module defines a *HomologyGroup()* class which is an abelian group that prints itself in a way that is suitable for homology groups.

sage.homology.homology_group.**HomologyGroup**(*n*, *base_ring*, *invfac=None*)

Abelian group on n generators which represents a homology group in a fixed degree.

INPUT:

- n – integer; the number of generators
- base_ring – ring; the base ring over which the homology is computed
- inv_fac – list of integers; the invariant factors – ignored if the base ring is a field

OUTPUT:

A class that can represent the homology group in a fixed homological degree.

EXAMPLES:

```
sage: from sage.homology.homology_group import HomologyGroup
sage: G = AbelianGroup(5, [5,5,7,8,9]); G                                    #␣
↪needs sage.groups
Multiplicative Abelian group isomorphic to C5 x C5 x C7 x C8 x C9
sage: H = HomologyGroup(5, ZZ, [5,5,7,8,9]); H
C5 x C5 x C7 x C8 x C9
sage: AbelianGroup(4)                                                        #␣
↪needs sage.groups
Multiplicative Abelian group isomorphic to Z x Z x Z x Z
sage: HomologyGroup(4, ZZ)
Z x Z x Z x Z

sage: # needs sage.libs.flint (otherwise timeout)
sage: HomologyGroup(100, ZZ)
Z^100
```

class sage.homology.homology_group.**HomologyGroup_class**(*n*, *invfac*)

Bases: AdditiveAbelianGroup_fixed_gens

Discrete Abelian group on n generators. This class inherits from AdditiveAbelianGroup_fixed_gens; see sage.groups.additive_abelian.additive_abelian_group for more documentation. The main difference between the classes is in the print representation.

EXAMPLES:

```
sage: from sage.homology.homology_group import HomologyGroup
sage: G = AbelianGroup(5, [5,5,7,8,9]); G                                    #␣
↪needs sage.groups
Multiplicative Abelian group isomorphic to C5 x C5 x C7 x C8 x C9
sage: H = HomologyGroup(5, ZZ, [5,5,7,8,9]); H
C5 x C5 x C7 x C8 x C9
sage: G == loads(dumps(G))                                                   #␣
↪needs sage.groups
True
sage: AbelianGroup(4)                                                        #␣
↪needs sage.groups
Multiplicative Abelian group isomorphic to Z x Z x Z x Z
sage: HomologyGroup(4, ZZ)
Z x Z x Z x Z
sage: HomologyGroup(100, ZZ)
Z^100
```

CHAPTER

NINE

HOMOLOGY AND COHOMOLOGY WITH A BASIS

This module provides homology and cohomology vector spaces suitable for computing cup products and cohomology operations.

REFERENCES:

- [GDR2003]
- [GDR1999]

AUTHORS:

- John H. Palmieri, Travis Scrimshaw (2015-09)

class sage.homology.homology_vector_space_with_basis.**CohomologyRing**(*base_ring*, *cell_complex*, *category=None*)

Bases: *HomologyVectorSpaceWithBasis*

The cohomology ring.

Note: This is not intended to be created directly by the user, but instead via the `cohomology ring` of a `cell complex`.

INPUT:

- `base_ring` – must be a field
- `cell_complex` – the cell complex whose homology we are computing
- `category` – (optional) a subcategory of modules with basis

EXAMPLES:

```
sage: CP2 = simplicial_complexes.ComplexProjectivePlane()
sage: H = CP2.cohomology_ring(QQ)
sage: H.basis(2)
Finite family {(2, 0): h^{2,0}}
sage: x = H.basis(2)[2,0]
```

The product structure is the cup product:

```
sage: x.cup_product(x)
-h^{4,0}
sage: x * x
-h^{4,0}
```

class Element

Bases: *Element*

cup_product(*other*)

Return the cup product of this element and `other`.

Algorithm: see González-Díaz and Réal [GDR2003], p. 88. Given two cohomology classes, lift them to cocycle representatives via the chain contraction for this complex, using *to_cycle()*. In the sum of their dimensions, look at all of the homology classes γ: lift each of those to a cycle representative, apply the Alexander-Whitney diagonal map to each cell in the cycle, evaluate the two cocycles on these factors, and multiply. The result is the value of the cup product cocycle on this homology class. After this has been done for all homology classes, since homology and cohomology are dual, one can tell which cohomology class corresponds to the cup product.

See also:

CohomologyRing.product_on_basis()

EXAMPLES:

```
sage: RP3 = simplicial_complexes.RealProjectiveSpace(3)
sage: H = RP3.cohomology_ring(GF(2))
sage: c = H.basis()[1,0]
sage: c.cup_product(c)
h^{2,0}
sage: c * c * c
h^{3,0}
```

We can also take powers:

```
sage: RP2 = simplicial_complexes.RealProjectivePlane()
sage: a = RP2.cohomology_ring(GF(2)).basis()[1,0]
sage: a**0
h^{0,0}
sage: a**1
h^{1,0}
sage: a**2
h^{2,0}
sage: a**3
0
```

A non-connected example:

```
sage: K = cubical_complexes.Torus().disjoint_union(cubical_complexes.
↪Sphere(2))
sage: a,b = K.cohomology_ring(QQ).basis(2)
sage: a**0
h^{0,0} + h^{0,1}
```

one()

The multiplicative identity element.

EXAMPLES:

```
sage: H = simplicial_complexes.Torus().cohomology_ring(QQ)
sage: H.one()
h^{0,0}
sage: all(H.one() * x == x == x * H.one() for x in H.basis())
True
```

product_on_basis(*li*, *ri*)

The cup product of the basis elements indexed by `li` and `ri` in this cohomology ring.

INPUT:

- `li`, `ri` – index of a cohomology class

See also:

CohomologyRing.Element.cup_product() – the documentation for this method describes the algorithm.

EXAMPLES:

```
sage: RP3 = simplicial_complexes.RealProjectiveSpace(3)
sage: H = RP3.cohomology_ring(GF(2))
sage: c = H.basis()[1,0]
sage: c.cup_product(c).cup_product(c)  # indirect doctest
h^{3,0}

sage: T = simplicial_complexes.Torus()
sage: x,y = T.cohomology_ring(QQ).basis(1)
sage: x.cup_product(y)
-h^{2,0}
sage: x.cup_product(x)
0

sage: one = T.cohomology_ring(QQ).basis()[0,0]
sage: x.cup_product(one)
h^{1,0}
sage: one.cup_product(y) == y
True
sage: one.cup_product(one)
h^{0,0}
sage: x.cup_product(y) + y.cup_product(x)
0
```

This also works with cubical complexes:

```
sage: T = cubical_complexes.Torus()
sage: x,y = T.cohomology_ring(QQ).basis(1)
sage: x.cup_product(y)
h^{2,0}
sage: x.cup_product(x)
0
```

Δ-complexes:

```
sage: T_d = delta_complexes.Torus()
sage: a,b = T_d.cohomology_ring(QQ).basis(1)
sage: a.cup_product(b)
h^{2,0}
sage: b.cup_product(a)
-h^{2,0}
sage: RP2 = delta_complexes.RealProjectivePlane()
sage: w = RP2.cohomology_ring(GF(2)).basis()[1,0]
sage: w.cup_product(w)
h^{2,0}
```

and simplicial sets:

```
sage: from sage.topology.simplicial_set_examples import RealProjectiveSpace
sage: RP5 = RealProjectiveSpace(5)                                          #␣
↪needs sage.groups
sage: x = RP5.cohomology_ring(GF(2)).basis()[1,0]                           #␣
↪needs sage.groups
sage: x**4                                                                  #␣
↪needs sage.groups
h^{4,0}
```

A non-connected example:

```
sage: K = cubical_complexes.Torus().disjoint_union(cubical_complexes.Torus())
sage: a,b,c,d = K.cohomology_ring(QQ).basis(1)
sage: x,y = K.cohomology_ring(QQ).basis(0)
sage: a.cup_product(x) == a
True
sage: a.cup_product(y)
0
```

class sage.homology.homology_vector_space_with_basis.**CohomologyRing_mod2**(*base_ring*, *cell_complex*)

Bases: *CohomologyRing*

The mod 2 cohomology ring.

Based on *CohomologyRing*, with Steenrod operations included.

Note: This is not intended to be created directly by the user, but instead via the `cohomology ring` of a `cell complex`.

Todo: Implement Steenrod operations on (co)homology at odd primes, and thereby implement this class over $\mathbf{F}_p$ for any p.

INPUT:

- `base_ring` – must be the field `GF(2)`
- `cell_complex` – the cell complex whose homology we are computing

EXAMPLES:

Mod 2 cohomology operations are defined on both the left and the right:

```
sage: CP2 = simplicial_complexes.ComplexProjectivePlane()
sage: Hmod2 = CP2.cohomology_ring(GF(2))
sage: y = Hmod2.basis(2)[2,0]
sage: y.Sq(2)
h^{4,0}

sage: # needs sage.groups
sage: Y = simplicial_sets.RealProjectiveSpace(6).suspension()
sage: H_Y = Y.cohomology_ring(GF(2))
sage: b = H_Y.basis()[2,0]
sage: b.Sq(1)
```

(continues on next page)

(continued from previous page)

```
h^{3,0}
sage: b.Sq(2)
0
sage: c = H_Y.basis()[4,0]
sage: c.Sq(1)
h^{5,0}
sage: c.Sq(2)
h^{6,0}
sage: c.Sq(3)
h^{7,0}
sage: c.Sq(4)
0
```

Cohomology can be viewed as a left module over the Steenrod algebra, and also as a right module:

```
sage: # needs sage.groups
sage: RP4 = simplicial_sets.RealProjectiveSpace(4)
sage: H = RP4.cohomology_ring(GF(2))
sage: x = H.basis()[1,0]
sage: Sq(0,1) * x
h^{4,0}
sage: Sq(3) * x
0
sage: x * Sq(3)
h^{4,0}
```

class Element

Bases: *Element*

Sq(i)

Return the result of applying Sq^i to this element.

INPUT:

- `i` – nonnegative integer

Warning: The main implementation is only for simplicial complexes and simplicial sets; cubical complexes are converted to simplicial complexes first. Note that this converted complex may be large and so computations may be slow. There is no implementation for Δ-complexes.

This cohomology operation is only defined in characteristic 2. Odd primary Steenrod operations are not implemented.

Algorithm: see González-Díaz and Réal [GDR1999], Corollary 3.2.

EXAMPLES:

```
sage: RP2 = simplicial_complexes.RealProjectiveSpace(2)
sage: x = RP2.cohomology_ring(GF(2)).basis()[1,0]
sage: x.Sq(1)
h^{2,0}

sage: K = RP2.suspension()
sage: K.set_immutable()
sage: y = K.cohomology_ring(GF(2)).basis()[2,0]
sage: y.Sq(1)
h^{3,0}
```

(continues on next page)

(continued from previous page)

```
sage: # long time
sage: # needs sage.groups
sage: RP4 = simplicial_complexes.RealProjectiveSpace(4)
sage: H = RP4.cohomology_ring(GF(2))
sage: x = H.basis()[1,0]
sage: y = H.basis()[2,0]
sage: z = H.basis()[3,0]
sage: x.Sq(1) == y
True
sage: z.Sq(1)
h^{4,0}
```

This calculation is much faster with simplicial sets (on one machine, 20 seconds with a simplicial complex, 4 ms with a simplicial set).

```
sage: RP4_ss = simplicial_sets.RealProjectiveSpace(4)                    #␣
↪needs sage.groups
sage: z_ss = RP4_ss.cohomology_ring(GF(2)).basis()[3,0]                  #␣
↪needs sage.groups
sage: z_ss.Sq(1)                                                         #␣
↪needs sage.groups
h^{4,0}
```

steenrod_module_map(*deg_domain*, *deg_codomain*, *side='left'*)

Return a component of the module structure map $A \otimes H \to H$, where H is this cohomology ring and A is the Steenrod algebra.

INPUT:

- `deg_domain` – the degree of the domain in the cohomology ring
- `deg_codomain` – the degree of the codomain in the cohomology ring
- `side` – (default `'left'`) whether we are computing the action as a left module action or a right module

We will write this with respect to the left action; for the right action, just switch all of the the tensors. Writing m for `deg_domain` and n for `deg_codomain`, this returns $A^{n-m} \otimes H^m \to H^n$, one single component of the map making H into an A-module.

Warning: This is only implemented in characteristic two. The main implementation is only for simplicial complexes and simplicial sets; cubical complexes are converted to simplicial complexes first. Note that this converted complex may be large and so computations may be slow. There is no implementation for Δ-complexes.

ALGORITHM:

Use the Milnor basis for the truncated Steenrod algebra A, and for cohomology, use the basis with which it is equipped. For each pair of basis elements a and h, compute the product $a \otimes h$, and use this to assemble a matrix defining the action map via multiplication on the appropriate side. That is, if `side` is `'left'`, return a matrix suitable for multiplication on the left, etc.

EXAMPLES:

```
sage: # needs sage.groups
sage: RP4 = simplicial_sets.RealProjectiveSpace(4)
```

(continues on next page)

(continued from previous page)

```
sage: H = RP4.cohomology_ring(GF(2))
sage: H.steenrod_module_map(1, 2)
[1]
sage: H.steenrod_module_map(1, 3)
[0]
sage: H.steenrod_module_map(1, 4, 'left')
[1 0]
sage: H.steenrod_module_map(1, 4, 'right')
[1]
[1]
```

Products of projective spaces:

```
sage: RP3 = simplicial_sets.RealProjectiveSpace(3)
sage: K = RP3.product(RP3)
sage: H = K.cohomology_ring(GF(2))
sage: H
Cohomology ring of RP^3 x RP^3 over Finite Field of size 2
```

There is one column for each element $a \otimes b$, where a is a basis element for the Steenrod algebra and b is a basis element for the cohomology algebra. There is one row for each basis element of the cohomology algebra. Unfortunately, the chosen basis for this truncated polynomial algebra is not the monomial basis:

```
sage: x1, x2 = H.basis(1)
sage: x1 * x1
h^{2,0} + h^{2,1}
sage: x2 * x2
h^{2,2}
sage: x1 * x2
h^{2,0}

sage: H.steenrod_module_map(1, 2)
[1 0]
[1 0]
[0 1]
sage: H.steenrod_module_map(1, 3, 'left')
[0 0]
[0 0]
[0 0]
[0 0]
sage: H.steenrod_module_map(1, 3, 'right')
[0 0 0 0]
[0 0 0 0]
sage: H.steenrod_module_map(2, 3)
[0 0 0]
[1 1 0]
[0 0 0]
[0 0 0]
```

class sage.homology.homology_vector_space_with_basis.**HomologyVectorSpaceWithBasis**(*base_ring*, *cell_complex*, *cohomology=False*, *category=None*)

Bases: CombinatorialFreeModule

Homology (or cohomology) vector space.

This provides enough structure to allow the computation of cup products and cohomology operations. See the class *CohomologyRing* (which derives from this) for examples.

It also requires field coefficients (hence the "VectorSpace" in the name of the class).

Note: This is not intended to be created directly by the user, but instead via the methods homology_with_basis() and cohomology_ring() for the class of cell complexes.

INPUT:

- base_ring – must be a field
- cell_complex – the cell complex whose homology we are computing
- cohomology – (default: False) if True, return the cohomology as a module
- category – (optional) a subcategory of modules with basis

EXAMPLES:

Homology classes are denoted by h_{d,i} where d is the degree of the homology class and i is their index in the list of basis elements in that degree. Cohomology classes are denoted h^{1,0}:

```
sage: RP2 = cubical_complexes.RealProjectivePlane()
sage: RP2.homology_with_basis(GF(2))
Homology module of Cubical complex with 21 vertices and 81 cubes
 over Finite Field of size 2
sage: RP2.cohomology_ring(GF(2))
Cohomology ring of Cubical complex with 21 vertices and 81 cubes
 over Finite Field of size 2
sage: simplicial_complexes.Torus().homology_with_basis(QQ)
Homology module of Minimal triangulation of the torus
 over Rational Field
```

To access a basis element, use its degree and index (0 or 1 in the 1st cohomology group of a torus):

```
sage: H = simplicial_complexes.Torus().cohomology_ring(QQ)
sage: H.basis(1)
Finite family {(1, 0): h^{1,0}, (1, 1): h^{1,1}}
sage: x = H.basis()[1,0]; x
h^{1,0}
sage: y = H.basis()[1,1]; y
h^{1,1}
sage: 2*x-3*y
2*h^{1,0} - 3*h^{1,1}
```

You can compute cup products of cohomology classes:

```
sage: x.cup_product(y)
-h^{2,0}
sage: y.cup_product(x)
h^{2,0}
sage: x.cup_product(x)
0
```

This works with simplicial, cubical, and Δ-complexes, and also simplicial sets:

```
sage: Torus_c = cubical_complexes.Torus()
sage: H = Torus_c.cohomology_ring(GF(2))
sage: x,y = H.basis(1)
sage: x.cup_product(x)
0
sage: x.cup_product(y)
h^{2,0}
sage: y.cup_product(y)
0

sage: Klein_d = delta_complexes.KleinBottle()
sage: H = Klein_d.cohomology_ring(GF(2))
sage: u,v = sorted(H.basis(1))
sage: u.cup_product(u)
h^{2,0}
sage: u.cup_product(v)
0
sage: v.cup_product(v)
h^{2,0}
```

An isomorphism between the rings for the cubical model and the Δ-complex model can be obtained by sending x to $u + v$, y to v.

```
sage: # needs sage.groups
sage: X = simplicial_sets.RealProjectiveSpace(6)
sage: H_X = X.cohomology_ring(GF(2))
sage: a = H_X.basis()[1,0]
sage: a**6
h^{6,0}
sage: a**7
0
```

All products of positive-dimensional elements in a suspension should be zero:

```
sage: # needs sage.groups
sage: Y = X.suspension()
sage: H_Y = Y.cohomology_ring(GF(2))
sage: b = H_Y.basis()[2,0]
sage: b**2
0
sage: B = sorted(H_Y.basis())[1:]
sage: B
[h^{2,0}, h^{3,0}, h^{4,0}, h^{5,0}, h^{6,0}, h^{7,0}]
sage: import itertools
sage: [a*b for (a,b) in itertools.combinations(B, 2)]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

The basis elements in the simplicial complex case have been chosen differently; apply the change of basis $x \mapsto a+b$,

$y \mapsto b$ to see the same product structure.

```
sage: Klein_s = simplicial_complexes.KleinBottle()
sage: H = Klein_s.cohomology_ring(GF(2))
sage: a,b = H.basis(1)
sage: a.cup_product(a)
0
sage: a.cup_product(b)
h^{2,0}
sage: (a+b).cup_product(a+b)
h^{2,0}
sage: b.cup_product(b)
h^{2,0}
```

class Element

Bases: IndexedFreeModuleElement

eval(*other*)

Evaluate `self` at `other`.

INPUT:

- `other` – an element of the dual space; if `self` is an element of cohomology in dimension n, then `other` should be an element of homology in dimension n, and vice versa

This just calls the *eval()* method on the representing chains and cochains.

EXAMPLES:

```
sage: T = simplicial_complexes.Torus()
sage: homology = T.homology_with_basis(QQ)
sage: cohomology = T.cohomology_ring(QQ)
sage: a1, a2 = homology.basis(1)
sage: alpha1, alpha2 = cohomology.basis(1)
sage: a1.to_cycle()
(0, 3) - (0, 6) + (3, 6)
sage: alpha1.to_cycle()
-\chi_(1, 3) - \chi_(1, 4) - \chi_(2, 3) - \chi_(2, 4) - \chi_(2, 5) + \
↪chi_(3, 6)
sage: a1.eval(alpha1)
1
sage: alpha2.to_cycle()
\chi_(1, 3) + \chi_(1, 4) + \chi_(1, 6) + \chi_(2, 4) - \chi_(4, 5) + \
↪chi_(5, 6)
sage: alpha2.eval(a1)
0
sage: (2 * alpha2).eval(a1 + a2)
2
```

to_cycle()

(Co)cycle representative of this homogeneous (co)homology class.

EXAMPLES:

```
sage: S2 = simplicial_complexes.Sphere(2)
sage: H = S2.homology_with_basis(QQ)
sage: h20 = H.basis()[2,0]; h20
h_{2,0}
sage: h20.to_cycle()
-(0, 1, 2) + (0, 1, 3) - (0, 2, 3) + (1, 2, 3)
```

Chains are written as linear combinations of simplices σ. Cochains are written as linear combinations of characteristic functions χ_σ for those simplices:

```
sage: S2.cohomology_ring(QQ).basis()[2,0].to_cycle()
\chi_(1, 2, 3)
sage: S2.cohomology_ring(QQ).basis()[0,0].to_cycle()
\chi_(0,) + \chi_(1,) + \chi_(2,) + \chi_(3,)
```

basis(*d=None*)

Return (the degree `d` homogeneous component of) the basis of this graded vector space.

INPUT:

- `d` – (optional) the degree

EXAMPLES:

```
sage: RP2 = simplicial_complexes.ProjectivePlane()
sage: H = RP2.homology_with_basis(QQ)
sage: H.basis()
Finite family {(0, 0): h_{0,0}}
sage: H.basis(0)
Finite family {(0, 0): h_{0,0}}
sage: H.basis(1)
Finite family {}
sage: H.basis(2)
Finite family {}
```

complex()

The cell complex whose homology is being computed.

EXAMPLES:

```
sage: H = simplicial_complexes.Simplex(2).homology_with_basis(QQ)
sage: H.complex()
The 2-simplex
```

contraction()

The chain contraction associated to this homology computation.

That is, to work with chain representatives of homology classes, we need the chain complex C associated to the cell complex, the chain complex H of its homology (with trivial differential), chain maps $\pi : C \to H$ and $\iota : H \to C$, and a chain contraction ϕ giving a chain homotopy between 1_C and $\iota \circ \pi$.

OUTPUT: ϕ

See *ChainContraction* for information about chain contractions, and see *algebraic_topological_model()* for the construction of this particular chain contraction ϕ.

EXAMPLES:

```
sage: H = simplicial_complexes.Simplex(2).homology_with_basis(QQ)
sage: H.contraction()
Chain homotopy between:
  Chain complex endomorphism of Chain complex with at most 3 nonzero terms␣
↪over Rational Field
  and Chain complex endomorphism of Chain complex with at most 3 nonzero␣
↪terms over Rational Field
```

From the chain contraction, one can also recover the maps π and ι:

```
sage: phi = H.contraction()
sage: phi.pi()
Chain complex morphism:
  From: Chain complex with at most 3 nonzero terms over Rational Field
  To: Chain complex with at most 1 nonzero terms over Rational Field
sage: phi.iota()
Chain complex morphism:
  From: Chain complex with at most 1 nonzero terms over Rational Field
  To: Chain complex with at most 3 nonzero terms over Rational Field
```

degree_on_basis(*i*)

Return the degree of the basis element indexed by `i`.

EXAMPLES:

```
sage: H = simplicial_complexes.Torus().homology_with_basis(GF(7))
sage: H.degree_on_basis((2,0))
2
```

dual()

Return the dual space.

If `self` is homology, return the cohomology ring. If `self` is cohomology, return the homology as a vector space.

EXAMPLES:

```
sage: T = simplicial_complexes.Torus()
sage: hom = T.homology_with_basis(GF(2))
sage: coh = T.cohomology_ring(GF(2))
sage: hom.dual() is coh
True
sage: coh.dual() is hom
True
```

class sage.homology.homology_vector_space_with_basis.**HomologyVectorSpaceWithBasis_mod2**(*base_cell_complex*, *category*=

Bases: *HomologyVectorSpaceWithBasis*

Homology vector space mod 2.

Based on *HomologyVectorSpaceWithBasis*, with Steenrod operations included.

Note: This is not intended to be created directly by the user, but instead via the method `homology_with_basis()` for the class of `cell complexes`.

Todo: Implement Steenrod operations on (co)homology at odd primes, and thereby implement this class over $\mathbf{F}_p$ for any p.

INPUT:

- `base_ring` – must be the field `GF(2)`
- `cell_complex` – the cell complex whose homology we are computing
- `category` – (optional) a subcategory of modules with basis

This does not include the `cohomology` argument present for *HomologyVectorSpaceWithBasis*: use *CohomologyRing_mod2* for cohomology.

EXAMPLES:

Mod 2 cohomology operations are defined on both the left and the right:

```
sage: # needs sage.groups
sage: RP4 = simplicial_sets.RealProjectiveSpace(5)
sage: H = RP4.homology_with_basis(GF(2))
sage: x4 = H.basis()[4,0]
sage: x4 * Sq(1)
h_{3,0}
sage: Sq(1) * x4
h_{3,0}
sage: Sq(2) * x4
h_{2,0}
sage: Sq(3) * x4
h_{1,0}
sage: Sq(0,1) * x4
h_{1,0}
sage: x4 * Sq(0,1)
h_{1,0}
sage: Sq(3) * x4
h_{1,0}
sage: x4 * Sq(3)
0
```

class Element

Bases: *Element*

sage.homology.homology_vector_space_with_basis.**is_GF2**(*R*)

Return `True` iff `R` is isomorphic to the field $\mathbf{F}_2$.

EXAMPLES:

```
sage: from sage.homology.homology_vector_space_with_basis import is_GF2
sage: is_GF2(GF(2))
True
sage: is_GF2(GF(2, impl='ntl'))
True
sage: is_GF2(GF(3))
False
```

sage.homology.homology_vector_space_with_basis.**sum_indices**(*k*, *i_k_plus_one*, *S_k_plus_one*)

This is a recursive function for computing the indices for the nested sums in González-Díaz and Réal [GDR1999], Corollary 3.2.

In the paper, given indices i_n, i_{n-1}, ..., i_{k+1}, given k, and given $S(k+1)$, the number $S(k)$ is defined to be

$$S(k) = -S(k+1) + floor(k/2) + floor((k+1)/2) + i_{k+1},$$

and i_k ranges from $S(k)$ to $i_{k+1} - 1$. There are two special cases: if $k = 0$, then $i_0 = S(0)$. Also, the initial case of $S(k)$ is $S(n)$, which is set in the method `Sq()` before calling this function. For this function, given k, i_{k+1}, and $S(k+1)$, return a list consisting of the allowable possible indices $[i_k, i_{k-1}, ..., i_1, i_0]$ given by the above formula.

INPUT:

- `k` – non-negative integer
- `i_k_plus_one` – the positive integer i_{k+1}
- `S_k_plus_one` – the integer $S(k+1)$

EXAMPLES:

```
sage: from sage.homology.homology_vector_space_with_basis import sum_indices
sage: sum_indices(1, 3, 3)
[[1, 0], [2, 1]]
sage: sum_indices(0, 4, 2)
[[2]]
```

CHAPTER

TEN

ALGEBRAIC TOPOLOGICAL MODEL FOR A CELL COMPLEX

This file contains two functions, *algebraic_topological_model()* and *algebraic_topological_model_delta_complex()*. The second works more generally: for all simplicial, cubical, and Δ-complexes. The first only works for simplicial and cubical complexes, but it is faster in those cases.

AUTHORS:

- John H. Palmieri (2015-09)

sage.homology.algebraic_topological_model.**algebraic_topological_model**(*K*, *base_ring=None*)

Algebraic topological model for cell complex `K` with coefficients in the field `base_ring`.

INPUT:

- `K` – either a simplicial complex or a cubical complex
- `base_ring` – coefficient ring; must be a field

OUTPUT: a pair `(phi, M)` consisting of

- chain contraction `phi`
- chain complex M

This construction appears in a paper by Pilarczyk and Réal [PR2015]. Given a cell complex K and a field F, there is a chain complex C associated to K with coefficients in F. The *algebraic topological model* for K is a chain complex M with trivial differential, along with chain maps $\pi : C \to M$ and $\iota : M \to C$ such that

- $\pi\iota = 1_M$, and
- there is a chain homotopy ϕ between 1_C and $\iota\pi$.

In particular, π and ι induce isomorphisms on homology, and since M has trivial differential, it is its own homology, and thus also the homology of C. Thus ι lifts homology classes to their cycle representatives.

The chain homotopy ϕ satisfies some additional properties, making it a *chain contraction*:

- $\phi\phi = 0$,
- $\pi\phi = 0$,
- $\phi\iota = 0$.

Given an algebraic topological model for K, it is then easy to compute cup products and cohomology operations on the cohomology of K, as described in [GDR2003] and [PR2015].

Implementation details: the cell complex K must have an `n_cells()` method from which we can extract a list of cells in each dimension. Combining the lists in increasing order of dimension then defines a filtration of the complex: a list of cells in which the boundary of each cell consists of cells earlier in the list. This is required

by Pilarczyk and Réal's algorithm. There must also be a `chain_complex()` method, to construct the chain complex C associated to this chain complex.

In particular, this works for simplicial complexes and cubical complexes. It doesn't work for Δ-complexes, though: the list of their n-cells has the wrong format.

Note that from the chain contraction `phi`, one can recover the chain maps π and ι via `phi.pi()` and `phi.iota()`. Then one can recover C and M from, for example, `phi.pi().domain()` and `phi.pi().codomain()`, respectively.

EXAMPLES:

```
sage: from sage.homology.algebraic_topological_model import algebraic_topological_
↪model
sage: RP2 = simplicial_complexes.RealProjectivePlane()
sage: phi, M = algebraic_topological_model(RP2, GF(2))
sage: M.homology()
{0: Vector space of dimension 1 over Finite Field of size 2,
 1: Vector space of dimension 1 over Finite Field of size 2,
 2: Vector space of dimension 1 over Finite Field of size 2}
sage: T = cubical_complexes.Torus()
sage: phi, M = algebraic_topological_model(T, QQ)
sage: M.homology()
{0: Vector space of dimension 1 over Rational Field,
 1: Vector space of dimension 2 over Rational Field,
 2: Vector space of dimension 1 over Rational Field}
```

If you want to work with cohomology rather than homology, just dualize the outputs of this function:

```
sage: M.dual().homology()
{0: Vector space of dimension 1 over Rational Field,
 1: Vector space of dimension 2 over Rational Field,
 2: Vector space of dimension 1 over Rational Field}
sage: M.dual().degree_of_differential()
1
sage: phi.dual()
Chain homotopy between:
  Chain complex endomorphism of
    Chain complex with at most 3 nonzero terms over Rational Field
  and Chain complex morphism:
    From: Chain complex with at most 3 nonzero terms over Rational Field
    To:   Chain complex with at most 3 nonzero terms over Rational Field
```

In degree 0, the inclusion of the homology M into the chain complex C sends the homology generator to a single vertex:

```
sage: K = simplicial_complexes.Simplex(2)
sage: phi, M = algebraic_topological_model(K, QQ)
sage: phi.iota().in_degree(0)
[0]
[0]
[1]
```

In cohomology, though, one needs the dual of every degree 0 cell to detect the degree 0 cohomology generator:

```
sage: phi.dual().iota().in_degree(0)
[1]
[1]
[1]
```

sage.homology.algebraic_topological_model.**algebraic_topological_model_delta_complex**(*K*, *base_ring*

Algebraic topological model for cell complex `K` with coefficients in the field `base_ring`.

This has the same basic functionality as *algebraic_topological_model()*, but it also works for Δ-complexes. For simplicial and cubical complexes it is somewhat slower, though.

INPUT:

- `K` – a simplicial complex, a cubical complex, or a Δ-complex
- `base_ring` – coefficient ring; must be a field

OUTPUT: a pair `(phi, M)` consisting of

- chain contraction `phi`
- chain complex M

See *algebraic_topological_model()* for the main documentation. The difference in implementation between the two: this uses matrix and vector algebra. The other function does more of the computations "by hand" and uses cells (given as simplices or cubes) to index various dictionaries. Since the cells in Δ-complexes are not as nice, the other function does not work for them, while this function relies almost entirely on the structure of the associated chain complex.

EXAMPLES:

```
sage: from sage.homology.algebraic_topological_model import algebraic_topological_
↪model_delta_complex as AT_model
sage: RP2 = simplicial_complexes.RealProjectivePlane()
sage: phi, M = AT_model(RP2, GF(2))
sage: M.homology()
{0: Vector space of dimension 1 over Finite Field of size 2,
 1: Vector space of dimension 1 over Finite Field of size 2,
 2: Vector space of dimension 1 over Finite Field of size 2}
sage: T = delta_complexes.Torus()
sage: phi, M = AT_model(T, QQ)
sage: M.homology()
{0: Vector space of dimension 1 over Rational Field,
 1: Vector space of dimension 2 over Rational Field,
 2: Vector space of dimension 1 over Rational Field}
```

If you want to work with cohomology rather than homology, just dualize the outputs of this function:

```
sage: M.dual().homology()
{0: Vector space of dimension 1 over Rational Field,
 1: Vector space of dimension 2 over Rational Field,
 2: Vector space of dimension 1 over Rational Field}
sage: M.dual().degree_of_differential()
1
sage: phi.dual()
Chain homotopy between:
  Chain complex endomorphism of Chain complex with at most 3 nonzero terms over␣
↪Rational Field
  and Chain complex morphism:
    From: Chain complex with at most 3 nonzero terms over Rational Field
    To:   Chain complex with at most 3 nonzero terms over Rational Field
```

In degree 0, the inclusion of the homology M into the chain complex C sends the homology generator to a single vertex:

```
sage: K = delta_complexes.Simplex(2)
sage: phi, M = AT_model(K, QQ)
sage: phi.iota().in_degree(0)
[0]
[0]
[1]
```

In cohomology, though, one needs the dual of every degree 0 cell to detect the degree 0 cohomology generator:

```
sage: phi.dual().iota().in_degree(0)
[1]
[1]
[1]
```

CHAPTER

ELEVEN

INDUCED MORPHISMS ON HOMOLOGY

This module implements morphisms on homology induced by morphisms of simplicial complexes. It requires working with field coefficients.

See *InducedHomologyMorphism* for documentation.

AUTHORS:

- John H. Palmieri (2015.09)

class sage.homology.homology_morphism.**InducedHomologyMorphism**(*map*, *base_ring=None*, *cohomology=False*)

Bases: Morphism

An element of this class is a morphism of (co)homology groups induced by a map of simplicial complexes. It requires working with field coefficients.

INPUT:

- `map` – the map of simplicial complexes
- `base_ring` – a field (optional, default `QQ`)
- `cohomology` – boolean (optional, default `False`). If `True`, return the induced map in cohomology rather than homology.

Note: This is not intended to be used directly by the user, but instead via the method `induced_homology_morphism()`.

EXAMPLES:

```
sage: S1 = simplicial_complexes.Sphere(1)
sage: H = Hom(S1, S1)
sage: f = H({0:0, 1:2, 2:1})  # f switches two vertices
sage: f_star = f.induced_homology_morphism(QQ, cohomology=True)
sage: f_star
Graded algebra endomorphism of
 Cohomology ring of Minimal triangulation of the 1-sphere over Rational Field
  Defn: induced by:
        Simplicial complex endomorphism of Minimal triangulation of the 1-sphere
          Defn: 0 |--> 0
                1 |--> 2
                2 |--> 1
sage: f_star.to_matrix(1)
[-1]
sage: f_star.to_matrix()
```

(continues on next page)

(continued from previous page)

```
[ 1| 0]
[--+--]
[ 0|-1]

sage: T = simplicial_complexes.Torus()
sage: y = T.homology_with_basis(QQ).basis()[(1,1)]
sage: y.to_cycle()
(0, 5) - (0, 6) + (5, 6)
```

Since $(0,2) - (0,5) + (2,5)$ is a cycle representing a homology class in the torus, we can define a map $S^1 \to T$ inducing an inclusion on H_1:

```
sage: Hom(S1, T)({0:0, 1:2, 2:5})
Simplicial complex morphism:
  From: Minimal triangulation of the 1-sphere
  To:   Minimal triangulation of the torus
  Defn: 0 |--> 0
        1 |--> 2
        2 |--> 5
sage: g = Hom(S1, T)({0:0, 1:2, 2: 5})
sage: g_star = g.induced_homology_morphism(QQ)
sage: g_star.to_matrix(0)
[1]
sage: g_star.to_matrix(1)
[-1]
[ 0]
sage: g_star.to_matrix()
[ 1| 0]
[--+--]
[ 0|-1]
[ 0| 0]
[--+--]
[ 0| 0]
```

We can evaluate such a map on (co)homology classes:

```
sage: H = S1.homology_with_basis(QQ)
sage: a = H.basis()[(1,0)]
sage: g_star(a)
-h_{1,0}

sage: T = S1.product(S1, is_mutable=False)
sage: diag = Hom(S1,T).diagonal_morphism()
sage: b,c = list(T.cohomology_ring().basis(1))
sage: diag_c = diag.induced_homology_morphism(cohomology=True)
sage: diag_c(b)
h^{1,0}
sage: diag_c(c)
h^{1,0}
```

base_ring()

The base ring for this map.

EXAMPLES:

```
sage: K = simplicial_complexes.Simplex(2)
sage: H = Hom(K,K)
```

(continues on next page)

(continued from previous page)

```
sage: id = H.identity()
sage: id.induced_homology_morphism(QQ).base_ring()
Rational Field
sage: id.induced_homology_morphism(GF(13)).base_ring()
Finite Field of size 13
```

is_identity()

Return `True` if this is the identity map on (co)homology.

EXAMPLES:

```
sage: S1 = simplicial_complexes.Sphere(1)
sage: H = Hom(S1, S1)
sage: flip = H({0:0, 1:2, 2:1})
sage: flip.induced_homology_morphism(QQ).is_identity()
False
sage: flip.induced_homology_morphism(GF(2)).is_identity()
True
sage: rotate = H({0:1, 1:2, 2:0})
sage: rotate.induced_homology_morphism(QQ).is_identity()
True
```

is_injective()

Return `True` if this map is injective on (co)homology.

EXAMPLES:

```
sage: S1 = simplicial_complexes.Sphere(1)
sage: K = simplicial_complexes.Simplex(2)
sage: H = Hom(S1, K)
sage: f = H({0:0, 1:1, 2:2})
sage: f.induced_homology_morphism().is_injective()
False
sage: f.induced_homology_morphism(cohomology=True).is_injective()
True

sage: T = simplicial_complexes.Torus()
sage: g = Hom(S1, T)({0:0, 1:3, 2: 6})
sage: g_star = g.induced_homology_morphism(QQ)
sage: g.is_injective()
True
```

is_surjective()

Return `True` if this map is surjective on (co)homology.

EXAMPLES:

```
sage: S1 = simplicial_complexes.Sphere(1)
sage: K = simplicial_complexes.Simplex(2)
sage: H = Hom(S1, K)
sage: f = H({0:0, 1:1, 2:2})
sage: f.induced_homology_morphism().is_surjective()
True
sage: f.induced_homology_morphism(cohomology=True).is_surjective()
False
```

to_matrix(*deg=None*)

The matrix for this map.

If degree `deg` is specified, return the matrix just in that degree; otherwise, return the block matrix representing the entire map.

INPUT:

- `deg` – (optional, default `None`) the degree

EXAMPLES:

```
sage: S1 = simplicial_complexes.Sphere(1)
sage: S1_b = S1.barycentric_subdivision()
sage: S1_b.set_immutable()
sage: d = {(0,): 0, (0,1): 1, (1,): 2, (1,2): 0, (2,): 1, (0,2): 2}
sage: f = Hom(S1_b, S1)(d)
sage: h = f.induced_homology_morphism(QQ)
sage: h.to_matrix(1)
[2]
sage: h.to_matrix()
[1|0]
[-+-]
[0|2]
```

CHAPTER

TWELVE

UTILITY FUNCTIONS FOR MATRICES

The actual computation of homology groups ends up being linear algebra with the differentials thought of as matrices. This module contains some utility functions for this purpose.

sage.homology.matrix_utils.**dhsw_snf**(*mat*, *verbose=False*)

Preprocess a matrix using the "Elimination algorithm" described by Dumas et al. [DHSW2003], and then call `elementary_divisors` on the resulting (smaller) matrix.

Note: 'snf' stands for 'Smith Normal Form'.

INPUT:

- `mat` – an integer matrix, either sparse or dense.

(They use the transpose of the matrix considered here, so they use rows instead of columns.)

ALGORITHM:

Go through `mat` one column at a time. For each column, add multiples of previous columns to it until either

- it's zero, in which case it should be deleted.
- its first nonzero entry is 1 or -1, in which case it should be kept.
- its first nonzero entry is something else, in which case it is deferred until the second pass.

Then do a second pass on the deferred columns.

At this point, the columns with 1 or -1 in the first entry contribute to the rank of the matrix, and these can be counted and then deleted (after using the 1 or -1 entry to clear out its row). Suppose that there were N of these.

The resulting matrix should be much smaller; we then feed it to Sage's `elementary_divisors` function, and prepend N 1's to account for the rows deleted in the previous step.

EXAMPLES:

```
sage: from sage.homology.matrix_utils import dhsw_snf
sage: mat = matrix(ZZ, 3, 4, range(12))
sage: dhsw_snf(mat)
[1, 4, 0]
sage: mat = random_matrix(ZZ, 20, 20, x=-1, y=2)
sage: mat.elementary_divisors() == dhsw_snf(mat)
True
```

CHAPTER

THIRTEEN

INTERFACE TO CHOMP

This module is deprecated: see github issue #33777.

CHomP stands for "Computation Homology Program", and is good at computing homology of simplicial complexes, cubical complexes, and chain complexes. It can also compute homomorphisms induced on homology by maps. See the CHomP web page http://chomp.rutgers.edu/ for more information.

AUTHOR:

- John H. Palmieri

class sage.interfaces.chomp.**CHomP**

Bases: object

Interface to the CHomP package.

Parameters

- **program** (*string*) – which CHomP program to use
- **complex** – a simplicial or cubical complex
- **subcomplex** – a subcomplex of complex or None (the default)
- **base_ring** (ring; optional, default $\mathbf{Z}$) – ring over which to perform computations – must be $\mathbf{Z}$ or $\mathbf{F}_p$.
- **generators** (*boolean; optional, default False*) – if True, also return list of generators
- **verbose** (*boolean; optional, default False*) – if True, print helpful messages as the computation progresses
- **extra_opts** (*string*) – options passed directly to program

Returns

homology groups as a dictionary indexed by dimension

The programs homsimpl, homcubes, and homchain are available through this interface. homsimpl computes the relative or absolute homology groups of simplicial complexes. homcubes computes the relative or absolute homology groups of cubical complexes. homchain computes the homology groups of chain complexes. For consistency with Sage's other homology computations, the answers produced by homsimpl and homcubes in the absolute case are converted to reduced homology.

Note also that CHomP can only compute over the integers or $\mathbf{F}_p$. CHomP is fast enough, though, that if you want rational information, you should consider using CHomP with integer coefficients, or with mod p coefficients for a sufficiently large p, rather than using Sage's built-in homology algorithms.

See also the documentation for the functions *homchain()*, *homcubes()*, and *homsimpl()* for more examples, including illustrations of some of the optional parameters.

EXAMPLES:

```
sage: from sage.interfaces.chomp import CHomP
sage: T = cubical_complexes.Torus()
sage: CHomP()('homcubes', T) # optional - CHomP
{0: 0, 1: Z x Z, 2: Z}
```

Relative homology of a segment relative to its endpoints:

```
sage: edge = simplicial_complexes.Simplex(1)
sage: ends = edge.n_skeleton(0)
sage: CHomP()('homsimpl', edge)  # optional - CHomP
{0: 0}
sage: CHomP()('homsimpl', edge, ends)  # optional - CHomP
{0: 0, 1: Z}
```

Homology of a chain complex:

```
sage: C = ChainComplex({3: 2 * identity_matrix(ZZ, 2)}, degree=-1)
sage: CHomP()('homchain', C)  # optional - CHomP
{2: C2 x C2}
```

help(*program*)

Print a help message for `program`, a program from the CHomP suite.

Parameters

program (*string*) – which CHomP program to use

Returns

nothing – just print a message

EXAMPLES:

```
sage: from sage.interfaces.chomp import CHomP
sage: CHomP().help('homcubes')   # optional - CHomP
doctest:...: DeprecationWarning: the CHomP interface is deprecated
See https://github.com/sagemath/sage/issues/33777 for details.
HOMCUBES, ver. ... Copyright (C) ... by Pawel Pilarczyk...
```

sage.interfaces.chomp.**have_chomp**(*program='homsimpl'*)

Return True if this computer has `program` installed.

The first time it is run, this function caches its result in the variable `_have_chomp` – a dictionary indexed by program name – and any subsequent time, it just checks the value of the variable.

This program is used in the routine CHomP.__call__.

If this computer doesn't have CHomP installed, you may obtain it from http://chomp.rutgers.edu/.

EXAMPLES:

```
sage: from sage.interfaces.chomp import have_chomp
sage: have_chomp() # random -- depends on whether CHomP is installed
doctest:...: DeprecationWarning: the CHomP interface is deprecated; hence so is␣
↪this function
See https://github.com/sagemath/sage/issues/33777 for details.
True
sage: 'homsimpl' in sage.interfaces.chomp._have_chomp
True
```

(continues on next page)

(continued from previous page)

```
sage: sage.interfaces.chomp._have_chomp['homsimpl'] == have_chomp()
True
```

sage.interfaces.chomp.**homchain**(*complex=None*, ***kwds*)

Compute the homology of a chain complex using the CHomP program `homchain`.

This function is deprecated: see github issue #33777.

Parameters

- **complex** – a chain complex
- **generators** (*boolean; optional, default False*) – if True, also return list of generators
- **verbose** (*boolean; optional, default False*) – if True, print helpful messages as the computation progresses
- **help** (*boolean; optional, default False*) – if True, just print a help message and exit
- **extra_opts** (*string*) – options passed directly to `homchain`

Returns

homology groups as a dictionary indexed by dimension

EXAMPLES:

```
sage: from sage.interfaces.chomp import homchain
sage: C = cubical_complexes.Sphere(3).chain_complex()
sage: homchain(C)[3]    # optional - CHomP
doctest:...: DeprecationWarning: the CHomP interface is deprecated
See https://github.com/sagemath/sage/issues/33777 for details.
Z
```

Generators: these are given as a list after the homology group. Each generator is specified as a cycle, an element in the appropriate free module over the base ring:

```
sage: C2 = delta_complexes.Sphere(2).chain_complex()
sage: homchain(C2, generators=True)[2]  # optional - CHomP
(Z, [(1, -1)])
sage: homchain(C2, generators=True, base_ring=GF(2))[2]  # optional - CHomP
(Vector space of dimension 1 over Finite Field of size 2, [(1, 1)])
```

sage.interfaces.chomp.**homcubes**(*complex=None*, *subcomplex=None*, ***kwds*)

Compute the homology of a cubical complex using the CHomP program `homcubes`. If the argument `subcomplex` is present, compute homology of `complex` relative to `subcomplex`.

This function is deprecated: see github issue #33777.

Parameters

- **complex** – a cubical complex
- **subcomplex** – a subcomplex of `complex` or None (the default)
- **base_ring** (ring; optional, default **Z**) – ring over which to perform computations – must be **Z** or $\mathbf{F}_p$.
- **generators** (*boolean; optional, default False*) – if True, also return list of generators

- **verbose** (*boolean; optional, default False*) – if True, print helpful messages as the computation progresses
- **help** (*boolean; optional, default False*) – if True, just print a help message and exit
- **extra_opts** (*string*) – options passed directly to `homcubes`

Returns
homology groups as a dictionary indexed by dimension

EXAMPLES:

```
sage: from sage.interfaces.chomp import homcubes
sage: S = cubical_complexes.Sphere(3)
sage: homcubes(S)[3]    # optional - CHomP
doctest:...: DeprecationWarning: the CHomP interface is deprecated
See https://github.com/sagemath/sage/issues/33777 for details.
Z
```

Relative homology:

```
sage: C3 = cubical_complexes.Cube(3)
sage: bdry = C3.n_skeleton(2)
sage: homcubes(C3, bdry)    # optional - CHomP
{0: 0, 1: 0, 2: 0, 3: Z}
```

Generators: these are given as a list after the homology group. Each generator is specified as a linear combination of cubes:

```
sage: homcubes(cubical_complexes.Sphere(1), generators=True, base_
↪ring=GF(2))[1][1]    # optional - CHomP
[[[1,1] x [0,1]] + [[0,1] x [1,1]] + [[0,1] x [0,0]] + [[0,0] x [0,1]]]
```

`sage.interfaces.chomp.`**homsimpl**(*complex=None*, *subcomplex=None*, ***kwds*)

Compute the homology of a simplicial complex using the CHomP program `homsimpl`. If the argument `subcomplex` is present, compute homology of `complex` relative to `subcomplex`.

This function is deprecated: see github issue #33777.

Parameters

- **complex** – a simplicial complex
- **subcomplex** – a subcomplex of `complex` or None (the default)
- **base_ring** (ring; optional, default **Z**) – ring over which to perform computations – must be **Z** or $\mathbf{F}_p$.
- **generators** (*boolean; optional, default False*) – if True, also return list of generators
- **verbose** (*boolean; optional, default False*) – if True, print helpful messages as the computation progresses
- **help** (*boolean; optional, default False*) – if True, just print a help message and exit
- **extra_opts** (*string*) – options passed directly to `program`

Returns
homology groups as a dictionary indexed by dimension

EXAMPLES:

```
sage: from sage.interfaces.chomp import homsimpl
sage: T = simplicial_complexes.Torus()
sage: M8 = simplicial_complexes.MooreSpace(8)
sage: M4 = simplicial_complexes.MooreSpace(4)
sage: X = T.disjoint_union(T).disjoint_union(T).disjoint_union(M8).disjoint_
↪union(M4)
sage: homsimpl(X)[1]  # optional - CHomP
doctest:...: DeprecationWarning: the CHomP interface is deprecated
See https://github.com/sagemath/sage/issues/33777 for details.
Z^6 x C4 x C8
```

Relative homology:

```
sage: S = simplicial_complexes.Simplex(3)
sage: bdry = S.n_skeleton(2)
sage: homsimpl(S, bdry)[3]   # optional - CHomP
Z
```

Generators: these are given as a list after the homology group. Each generator is specified as a linear combination of simplices:

```
sage: homsimpl(S, bdry, generators=True)[3]   # optional - CHomP
(Z, [(0, 1, 2, 3)])

sage: homsimpl(simplicial_complexes.Sphere(1), generators=True)   # optional -␣
↪CHomP
{0: 0, 1: (Z, [(0, 1) - (0, 2) + (1, 2)])}
```

sage.interfaces.chomp.**process_generators_chain**(*gen_string*, *dim*, *base_ring=None*)

Process CHomP generator information for simplicial complexes.

This function is deprecated: see github issue #33777.

Parameters

- **gen_string** (*string*) – generator output from CHomP
- **dim** (*integer*) – dimension in which to find generators
- **base_ring** (*optional, default ZZ*) – base ring over which to do the computations

Returns

list of generators in each dimension, as described below

`gen_string` has the form

```
[H_0]
a1

[H_1]
a2
a3

[H_2]
a1 - a2
```

For each homology group, each line lists a homology generator as a linear combination of generators `ai` of the group of chains in the appropriate dimension. The elements `ai` are indexed starting with $i = 1$. Each generator is

converted, using regular expressions, from a string to a vector (an element in the free module over `base_ring`), with `ai` representing the unit vector in coordinate $i-1$. For example, the string `a1 - a2` gets converted to the vector `(1, -1)`.

Therefore the return value is a list of vectors.

EXAMPLES:

```
sage: from sage.interfaces.chomp import process_generators_chain
sage: s = "[H_0]\na1\n\n[H_1]\na2\na3\n"
sage: process_generators_chain(s, 1)
doctest:...: DeprecationWarning: the CHomP interface is deprecated
See https://github.com/sagemath/sage/issues/33777 for details.
[(0, 1), (0, 0, 1)]
sage: s = "[H_0]\na1\n\n[H_1]\n5 * a2 - a1\na3\n"
sage: process_generators_chain(s, 1, base_ring=ZZ)
[(-1, 5), (0, 0, 1)]
sage: process_generators_chain(s, 1, base_ring=GF(2))
[(1, 1), (0, 0, 1)]
```

`sage.interfaces.chomp.`**`process_generators_cubical`**(*gen_string*, *dim*)

Process CHomP generator information for cubical complexes.

This function is deprecated: see github issue #33777.

Parameters

- **`gen_string`** (*string*) – generator output from CHomP
- **`dim`** (*integer*) – dimension in which to find generators

Returns

list of generators in each dimension, as described below

`gen_string` has the form

```
The 2 generators of H_1 follow:
generator 1
-1 * [(0,0,0,0,0)(0,0,0,0,1)]
1 * [(0,0,0,0,0)(0,0,1,0,0)]
...
generator 2
-1 * [(0,1,0,1,1)(1,1,0,1,1)]
-1 * [(0,1,0,0,1)(0,1,0,1,1)]
...
```

Each line consists of a coefficient multiplied by a cube; the cube is specified by its "bottom left" and "upper right" corners.

For technical reasons, we remove the first coordinate of each tuple, and using regular expressions, the remaining parts get converted from a string to a pair `(coefficient, Cube)`, with the cube represented as a product of tuples. For example, the first line in "generator 1" gets turned into

```
(-1, [0,0] x [0,0] x [0,0] x [0,1])
```

representing an element in the free abelian group with basis given by cubes. Each generator is a list of such pairs, representing the sum of such elements. These are reassembled in `CHomP.__call__()` to actual elements in the free module generated by the cubes of the cubical complex in the appropriate dimension.

Therefore the return value is a list of lists of pairs, one list of pairs for each generator.

EXAMPLES:

```
sage: from sage.interfaces.chomp import process_generators_cubical
sage: s = "The 2 generators of H_1 follow:\ngenerator 1:\n-1 * [(0,0,0,0,0)(0,0,0,
↪0,1)]\n1 * [(0,0,0,0,0)(0,0,1,0,0)]"
sage: process_generators_cubical(s, 1)
doctest:...: DeprecationWarning: the CHomP interface is deprecated
See https://github.com/sagemath/sage/issues/33777 for details.
[[(-1, [0,0] x [0,0] x [0,0] x [0,1]), (1, [0,0] x [0,1] x [0,0] x [0,0])]]
sage: len(process_generators_cubical(s, 1))  # only one generator
1
```

sage.interfaces.chomp.**process_generators_simplicial**(*gen_string*, *dim*, *complex*)

Process CHomP generator information for simplicial complexes.

This function is deprecated: see github issue #33777

Parameters

- **gen_string** (*string*) – generator output from CHomP
- **dim** (*integer*) – dimension in which to find generators
- **complex** – simplicial complex under consideration

Returns

list of generators in each dimension, as described below

`gen_string` has the form

```
The 2 generators of H_1 follow:
generator 1
-1 * (1,6)
1 * (1,4)
...
generator 2
-1 * (1,6)
1 * (1,4)
...
```

where each line contains a coefficient and a simplex. Each line is converted, using regular expressions, from a string to a pair `(coefficient, Simplex)`, like

```
(-1, (1,6))
```

representing an element in the free abelian group with basis given by simplices. Each generator is a list of such pairs, representing the sum of such elements. These are reassembled in `CHomP.__call__()` to actual elements in the free module generated by the simplices of the simplicial complex in the appropriate dimension.

Therefore the return value is a list of lists of pairs, one list of pairs for each generator.

EXAMPLES:

```
sage: from sage.interfaces.chomp import process_generators_simplicial
sage: s = "The 2 generators of H_1 follow:\ngenerator 1:\n-1 * (1,6)\n1 * (1,4)"
sage: process_generators_simplicial(s, 1, simplicial_complexes.Torus())
doctest:...: DeprecationWarning: the CHomP interface is deprecated
See https://github.com/sagemath/sage/issues/33777 for details.
[[(-1, (1, 6)), (1, (1, 4))]]
```

CHAPTER

FOURTEEN

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

h

i

INDEX

E

F

G

H

I

K

M

N

O

P

R

S

T

V