Contents

Chapter 1

Functions

1.1 module – module/ideal with HNF

- **Classes**
  - **Submodule**
  - **Module**
  - **Ideal**
  - **Ideal_with_generator**

1.1.1 Submodule – submodule as matrix representation

Initialize (Constructor)

Submodule(`row`: ***integer***, `column`: ***integer***, `compo`: ***compo*=0**, `coeff_ring`: ***CommutativeRing*=0**, `ishnf`: ***True/False*=None**)
→ ***Submodule***

Create a submodule with matrix representation.

Submodule is subclass of **RingMatrix**.
We assume that `coeff_ring` is a PID (principal ideal domain). Then, we have the HNF(hermite normal form) corresponding to a matrices.

If `ishnf` is True, we assume that the input matrix is a HNF.

Attribute

ishnf If the matrix is a HNF, then `ishnf` should be True, otherwise False.

Methods

1.1.1.1 getGenerators – generator of module

getGenerators(`self`) → ***list***

Return a (current) generator of the module `self`.

Return the list of vectors consisting of a generator.

1.1.1.2 isSubmodule – Check whether submodule of self

isSubmodule(`self`, `other`: ***Submodule***) → ***True/False***

Return True if the submodule instance is a submodule of the `other`, or False otherwise.

1.1.1.3 isEqual – Check whether self and other are same module

isEqual(`self`, `other`: ***Submodule***) → ***True/False***

Return True if the submodule instance is `other` as module, or False otherwise.
.

You should use the method for equality test of module, not matrix. For equality test of matrix simply, use `self==other`.

1.1.1.4 isContain – Check whether other is in self

isContains(`self`, `other`: ***vector.Vector***) → ***True/False***

Determine whether `other` is in `self` or not.
.

If you want to represent `other` as linear combination with the HNF generator of `self`, use **represent_element**.

1.1.1.5 toHNF - change to HNF

toHNF(`self`) → ***(None)***

Rewrite `self` to HNF (hermite normal form), and set True to its `ishnf`.

Note that HNF do not always give basis of `self`.(i.e. HNF may be redundant.)

1.1.1.6 sumOfSubmodules - sum as submodule

sumOfSubmodules(`self`, `other`: ***Submodule***) → ***Submodule***

Return a module which is sum of two subspaces.

1.1.1.7 intersectionOfSubmodules - intersection as submodule

intersectionOfSubmodules(`self`, `other`: ***Submodule***) → ***Submodule***

Return a module which is intersection of two subspaces.

1.1.1.8 represent_element – represent element as linear combination

represent_element(`self`, `other`: ***vector.Vector***) → ***vector.Vector/False***

Represent `other` as a linear combination with HNF generators.

If `other` not in `self`, return False. Note that this method calls **toHNF**.

The method returns coefficients as an instance of **Vector**.

1.1.1.9 linear_combination – compute linear combination

linear_combination(`self`, `coeff`: ***list***) → ***vector.Vector***

For given **Z**-coefficients `coeff`, return a vector corresponding to a linear combination of (current) basis.

`coeff` must be a list of instances in **RingElement** whose size is the column of `self`.

Examples

```
>>> A = module.Submodule(4, 3, [1,2,3]+[4,5,6]+[7,8,9]+[10,11,12])
>>> A.toHNF()
>>> print A
9 1
6 1
3 1
0 1
>>> A.getGenerator
[Vector([9L, 6L, 3L, 0L]), Vector([1L, 1L, 1L, 1L])]
>>> V = vector.Vector([10,7,4,1])
>>> A.represent_element(V)
Vector([1L, 1L])
>>> V == A.linear_combination([1,1])
True
>>> B = module.Submodule(4, 1, [1,2,3,4])
>>> C = module.Submodule(4, 2, [2,-4]+[4,-3]+[6,-2]+[8,-1])
>>> print B.intersectionOfSubmodules(C)
2
4
6
8
```

1.1.2 fromMatrix(class function) - create submodule

fromMatrix(`cls`, `mat`: ***RingMatrix***, `ishnf`: ***True/False*=None**)
→ ***Submodule***

Create a Submodule instance from a matrix instance `mat`, whose class can be any of subclasses of Matrix.

Please use this method if you want a Submodule instance for sure.

1.1.3 Module - module over a number field

Initialize (Constructor)

Module(pair_mat_repr: ***list/matrix***, number_field: ***algfield.NumberField***, base: ***list/matrix.SquareMatrix*=None**, ishnf: ***bool*=False**)
$\to$ ***Module***

Create a new module object over a number field.

A module is a finitely generated sub $\mathbf{Z}$-module. Note that we do not assume rank of a module is deg(number_field).
We represent a module as generators respect to base module over $\mathbf{Z}[\theta]$, where θ is a solution of `number_field`.**polynomial**.

`pair_mat_repr` should be one of the following form:

- $[M,\ d]$, where M is a list of integral tuple/vectors whose size is the degree of `number_field` and d is a denominator.
- $[M,\ d]$, where M is an integral matrix whose the number of row is the degree of `number_field` and d is a denominator.
- a rational matrix whose the number of row is the degree of `number_field`.

Also, `base` should be one of the following form:

- a list of rational tuple/vectors whose size is the degree of `number_field`
- a square non-singular rational matrix whose size is the degree of `number_field`

The module is internally represented as $\frac{1}{d}M$ with respect to **base**, where d is **denominator** and M is **mat_repr**. If `ishnf` is True, we assume that `mat_repr` is a HNF.

Attribute

mat_repr : an instance of **Submodule** M whose size is the degree of `number_field`

denominator : an integer d

base : a square non-singular rational matrix whose size is the degree of `number_field`

number_field : the number field over which the module is defined

Operations

operator	explanation
`M==N`	Return whether `M` and `N` are equal or not as module.
`c in M`	Check whether some element of `M` equals `c`.
`M+N`	Return the sum of `M` and `N` as module.
`M*N`	Return the product of `M` and `N` as the ideal computation. `N` must be module or scalar(i.e. an element of **number_field**). If you want to compute the intersection of M and N, see **intersect**.
`M**c`	Return `M` to `c` based on the ideal multiplication.
`repr(M)`	Return the repr string of the module `M`.
`str(M)`	Return the str string of the module `M`.

Examples

```
>>> F = algfield.NumberField([2,0,1])
>>> M_1 = module.Module([matrix.RingMatrix(2,2,[1,0]+[0,2]), 2], F)
>>> M_2 = module.Module([matrix.RingMatrix(2,2,[2,0]+[0,5]), 3], F)
>>> print M_1
([1, 0]+[0, 2], 2)
 over
([1L, 0L]+[0L, 1L], NumberField([2, 0, 1]))
>>> print M_1 + M_2
([1L, 0L]+[0L, 2L], 6)
 over
([Rational(1, 1), Rational(0, 1)]+[Rational(0, 1), Rational(1, 1)],
NumberField([2, 0, 1]))
>>> print M_1 * 2
([1L, 0L]+[0L, 2L], 1L)
 over
([Rational(1, 1), Rational(0, 1)]+[Rational(0, 1), Rational(1, 1)],
NumberField([2, 0, 1]))
>>> print M_1 * M_2
([2L, 0L]+[0L, 1L], 6L)
 over
([Rational(1, 1), Rational(0, 1)]+[Rational(0, 1), Rational(1, 1)],
NumberField([2, 0, 1]))
>>> print M_1 ** 2
([1L, 0L]+[0L, 2L], 4L)
 over
([Rational(1, 1), Rational(0, 1)]+[Rational(0, 1), Rational(1, 1)],
NumberField([2, 0, 1]))
```

Methods

1.1.3.1 toHNF - change to hermite normal form(HNF)

toHNF(`self`) → ***(None)***

Change `self`.**mat_repr** to the hermite normal form(HNF).

1.1.3.2 copy - create copy

copy(`self`) → ***Module***

Create copy of `self`.

1.1.3.3 intersect - intersection

intersect(`self`, `other`: ***Module***) → ***Module***

Return intersection of `self` and `other`.

1.1.3.4 issubmodule - Check submodule

submodule(`self`, `other`: ***Module***) → ***True/False***

Check `self` is submodule of `other`.

1.1.3.5 issupermodule - Check supermodule

supermodule(`self`, `other`: ***Module***) → ***True/False***

Check `self` is supermodule of `other`.

1.1.3.6 represent_element - Represent as linear combination

represent_element(`self`, `other`: ***algfield.BasicAlgNumber***)
→ ***list/False***

Represent other as a linear combination with generators of self. If other is not in self, return False.

Note that we do not assume self.mat_repr is HNF.
The output is a list of integers if other is in self.

1.1.3.7 change_base_module - Change base

change_base_module(self, other_base: *list/matrix.RingSquareMatrix*) → *Module*

Return the module which is equal to self respect to other_base.

other_base follows the form base.

1.1.3.8 index - size of module

index(self) → *rational.Rational*

Return the order of a residue group over self.base. That is, return $[M:N]$ if $N \subset M$ or $[N:M]^{-1}$ if $MsubsetN$, where M is the module self and N is the module corresponding to self.base.

1.1.3.9 smallest_rational - a Z-generator in the rational field

smallest_rational(self) → *rational.Rational*

Return the **Z**-generator of intersection of the module self and the rational field.

Examples

```
>>> F = algfield.NumberField([1,0,2])
>>> M_1=module.Module([matrix.RingMatrix(2,2,[1,0]+[0,2]), 2], F)
>>> M_2=module.Module([matrix.RingMatrix(2,2,[2,0]+[0,5]), 3], F)
>>> print M_1.intersect(M_2)
([2L, 0L]+[0L, 5L], 1L)
 over
([Rational(1, 1), Rational(0, 1)]+[Rational(0, 1), Rational(1, 1)],
```

```
 NumberField([2, 0, 1]))
>>> M_1.represent_element( F.createElement( [[2,4], 1] ) )
[4L, 4L]
>>> print M_1.change_base_module( matrix.FieldSquareMatrix(2, 2, [1,0]+[0,1]) / 2 )
([1L, 0L]+[0L, 2L], 1L)
 over
([Rational(1, 2), Rational(0, 1)]+[Rational(0, 1), Rational(1, 2)],
 NumberField([2, 0, 1]))
>>> M_2.index()
Rational(10, 9)
>>> M_2.smallest_rational()
Rational(2, 3)
```

1.1.4 Ideal - ideal over a number field

Initialize (Constructor)

Ideal(`pair_mat_repr`: ***list/matrix***, `number_field`: ***algfield.NumberField***, `base`: ***list/matrix.SquareMatrix***=**None**, `ishnf`: ***bool***=**False**)
→ ***Ideal***

Create a new ideal object over a number field.

Ideal is subclass of **Module**.

Refer to initialization of **Module**.

Methods

1.1.4.1 inverse – inverse

inverse(`self`) → ***Ideal***

Return the inverse ideal of `self`.

This method calls `self`.**number_field**.**integer_ring**.

1.1.4.2 issubideal – Check subideal

issubideal(`self`, `other`: ***Ideal***) → ***Ideal***

Check `self` is subideal of `other`.

1.1.4.3 issuperideal – Check superideal

issuperideal(`self`, `other`: ***Ideal***) → ***Ideal***

Check `self` is superideal of `other`.

1.1.4.4 gcd – greatest common divisor

gcd(`self`, `other`: ***Ideal***) → ***Ideal***

Return the greatest common divisor(gcd) of `self` and `other` as ideal.

This method simply executes `self`+`other`.

1.1.4.5 lcm – least common multiplier

lcm(`self`, `other`: ***Ideal***) → ***Ideal***

Return the least common multiplier(lcm) of `self` and `other` as ideal.

This method simply calls the method **intersect**.

1.1.4.6 norm – norm

norm(`self`) → ***rational.Rational***

Return the norm of `self`.

This method calls `self`.**number_field**.**integer_ring**.

1.1.4.7 isIntegral – Check integral

isIntegral(`self`) → ***True/False***

Determine whether `self` is an integral ideal or not.

Examples

```
>>> M = module.Ideal([matrix.RingMatrix(2, 2, [1,0]+[0,2]), 2], F)
>>> print M.inverse()
([-2L, 0L]+[0L, 2L], 1L)
 over
([Rational(1, 1), Rational(0, 1)]+[Rational(0, 1), Rational(1, 1)],
 NumberField([2, 0, 1]))
>>> print M * M.inverse()
([1L, 0L]+[0L, 1L], 1L)
 over
([Rational(1, 1), Rational(0, 1)]+[Rational(0, 1), Rational(1, 1)],
 NumberField([2, 0, 1]))
>>> M.norm()
Rational(1, 2)
>>> M.isIntegral()
False
```

1.1.5 Ideal_with_generator - ideal with generator

Initialize (Constructor)

Ideal_with_generator(generator: *list*) → ***Ideal_with_generator***

Create a new ideal given as a generator.

`generator` is a list of instances in **BasicAlgNumber**, which represent generators, over a same number field.

Attribute

generator : generators of the ideal

number_field : the number field over which generators are defined

Operations

operator	explanation
`M==N`	Return whether `M` and `N` are equal or not as module.
`c in M`	Check whether some element of `M` equals `c`.
`M+N`	Return the sum of `M` and `N` as ideal with generators.
`M*N`	Return the product of `M` and `N` as ideal with generators.
`M**c`	Return `M` to `c` based on the ideal multiplication.
`repr(M)`	Return the repr string of the ideal `M`.
`str(M)`	Return the str string of the ideal `M`.

Examples

```
>>> F = algfield.NumberField([2,0,1])
>>> M_1 = module.Ideal_with_generator([
 F.createElement([[1,0], 2]), F.createElement([[0,1], 1])
])
>>> M_2 = module.Ideal_with_generator([
 F.createElement([[2,0], 3]), F.createElement([[0,5], 3])
])
>>> print M_1
[BasicAlgNumber([[1, 0], 2], [2, 0, 1]), BasicAlgNumber([[0, 1], 1], [2, 0, 1])]
>>> print M_1 + M_2
[BasicAlgNumber([[1, 0], 2], [2, 0, 1]), BasicAlgNumber([[0, 1], 1], [2, 0, 1]),
```

```
 BasicAlgNumber([[2, 0], 3], [2, 0, 1]), BasicAlgNumber([[0, 5], 3], [2, 0, 1])]
>>> print M_1 * M_2
[BasicAlgNumber([[1L, 0L], 3L], [2, 0, 1]), BasicAlgNumber([[0L, 5L], 6], [2, 0, 1]),
BasicAlgNumber([[0L, 2L], 3], [2, 0, 1]), BasicAlgNumber([[-10L, 0L], 3], [2, 0, 1])]
>>> print M_1 ** 2
[BasicAlgNumber([[1L, 0L], 4], [2, 0, 1]), BasicAlgNumber([[0L, 1L], 2], [2, 0, 1]),
BasicAlgNumber([[0L, 1L], 2], [2, 0, 1]), BasicAlgNumber([[-2L, 0L], 1], [2, 0, 1])]
```

Methods

1.1.5.1 copy - create copy

copy(`self`) → ***Ideal_with_generator***

Create copy of `self`.

1.1.5.2 to_HNFRepresentation - change to ideal with HNF

to_HNFRepresentation(`self`) → ***Ideal***

Transform `self` to the corresponding ideal as HNF(hermite normal form) representation.

1.1.5.3 twoElementRepresentation - Represent with two element

twoElementRepresentation(`self`) → ***Ideal_with_generator***

Transform `self` to the corresponding ideal as HNF(hermite normal form) representation.

If `self` is not a prime ideal, this method is not efficient.

1.1.5.4 smallest_rational - a Z-generator in the rational field

smallest_rational(`self`) → ***rational.Rational***

Return the **Z**-generator of intersection of the module `self` and the rational field.

This method calls **to_HNFRepresentation**.

1.1.5.5 inverse – inverse

inverse(`self`) → ***Ideal***

Return the inverse ideal of `self`.

This method calls **to_HNFRepresentation**.

1.1.5.6 norm – norm

norm(`self`) → ***rational.Rational***

Return the norm of `self`.

This method calls **to_HNFRepresentation**.

1.1.5.7 intersect - intersection

intersect(`self`, `other`: ***Ideal_with_generator***) → ***Ideal***

Return intersection of `self` and `other`.

This method calls **to_HNFRepresentation**.

1.1.5.8 issubideal – Check subideal

issubideal(`self`, `other`: ***Ideal_with_generator***) → ***Ideal***

Check `self` is subideal of `other`.

This method calls **to_HNFRepresentation**.

1.1.5.9 issuperideal – Check superideal

issuperideal(`self`, `other`: ***Ideal_with_generator***) → ***Ideal***

This method calls **to_HNFRepresentation**.

Examples

```
>>> M = module.Ideal_with_generator([
F.createElement([[2,0], 3]), F.createElement([[0,2], 3]), F.createElement([[1,0], 3])
])
>>> print M.to_HNFRepresentation()
([2L, 0L, 0L, -4L, 1L, 0L]+[0L, 2L, 2L, 0L, 0L, 1L], 3L)
 over
([1L, 0L]+[0L, 1L], NumberField([2, 0, 1]))
>>> print M.twoElementRepresentation()
[BasicAlgNumber([[1L, 0], 3], [2, 0, 1]), BasicAlgNumber([[3, 2], 3], [2, 0, 1])]
>>> M.norm()
Rational(1, 9)
```