Returns
DomainMatrix containing elements of rows

Examples

```
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy.abc import x, y, z
>>> A = DomainMatrix.from_list_sympy(1, 3, [[x, y, z]])
>>> A
DomainMatrix([[x, y, z]], (1, 3), ZZ[x,y,z])
```

See also:

sympy.polys.constructor.construct_domain (page 2427), *from_dict_sympy* (page 2675)

classmethod from_rep(*rep*)

Create a new DomainMatrix efficiently from DDM/SDM.

Parameters
rep: SDM or DDM

The internal sparse or dense representation of the matrix.

Returns
DomainMatrix

A *DomainMatrix* (page 2671) wrapping *rep*.

Examples

Create a *DomainMatrix* (page 2671) with an dense internal representation as *DDM* (page 2690):

```
>>> from sympy.polys.domains import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy.polys.matrices.ddm import DDM
>>> drep = DDM([[ZZ(1), ZZ(2)], [ZZ(3), ZZ(4)]], (2, 2), ZZ)
>>> dM = DomainMatrix.from_rep(drep)
>>> dM
DomainMatrix([[1, 2], [3, 4]], (2, 2), ZZ)
```

Create a *DomainMatrix* (page 2671) with a sparse internal representation as *SDM* (page 2692):

```
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy.polys.matrices.sdm import SDM
>>> from sympy import ZZ
>>> drep = SDM({0:{1:ZZ(1)},1:{0:ZZ(2)}}, (2, 2), ZZ)
>>> dM = DomainMatrix.from_rep(drep)
>>> dM
DomainMatrix({0: {1: 1}, 1: {0: 2}}, (2, 2), ZZ)
```

Notes

This takes ownership of rep as its internal representation. If rep is being mutated elsewhere then a copy should be provided to `from_rep`. Only minimal verification or checking is done on *rep* as this is supposed to be an efficient internal routine.

hstack(**B*)

Horizontally stack the given matrices.

Parameters
B: DomainMatrix

Matrices to stack horizontally.

Returns
DomainMatrix

DomainMatrix by stacking horizontally.

Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
```

```
>>> A = DomainMatrix([[ZZ(1), ZZ(2)], [ZZ(3), ZZ(4)]], (2, 2), ZZ)
>>> B = DomainMatrix([[ZZ(5), ZZ(6)], [ZZ(7), ZZ(8)]], (2, 2), ZZ)
>>> A.hstack(B)
DomainMatrix([[1, 2, 5, 6], [3, 4, 7, 8]], (2, 4), ZZ)
```

```
>>> C = DomainMatrix([[ZZ(9), ZZ(10)], [ZZ(11), ZZ(12)]], (2, 2), ZZ)
>>> A.hstack(B, C)
DomainMatrix([[1, 2, 5, 6, 9, 10], [3, 4, 7, 8, 11, 12]], (2, 6), ZZ)
```

See also:

unify (page 2688)

inv()

Finds the inverse of the DomainMatrix if exists

Returns
DomainMatrix

DomainMatrix after inverse

Raises
ValueError

If the domain of DomainMatrix not a Field

DMNonSquareMatrixError

If the DomainMatrix is not a not Square DomainMatrix

Examples

```
>>> from sympy import QQ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
...     [QQ(2), QQ(-1), QQ(0)],
...     [QQ(-1), QQ(2), QQ(-1)],
...     [QQ(0), QQ(0), QQ(2)]], (3, 3), QQ)
>>> A.inv()
DomainMatrix([[2/3, 1/3, 1/6], [1/3, 2/3, 1/3], [0, 0, 1/2]], (3, 3),␣
↪QQ)
```

See also:

neg (page 2681)

property is_lower

Says whether this matrix is lower-triangular. True can be returned even if the matrix is not square.

property is_upper

Says whether this matrix is upper-triangular. True can be returned even if the matrix is not square.

lu()

Returns Lower and Upper decomposition of the DomainMatrix

Returns

(L, U, exchange)

L, U are Lower and Upper decomposition of the DomainMatrix, exchange is the list of indices of rows exchanged in the decomposition.

Raises

ValueError

If the domain of DomainMatrix not a Field

Examples

```
>>> from sympy import QQ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
...     [QQ(1), QQ(-1)],
...     [QQ(2), QQ(-2)]], (2, 2), QQ)
>>> A.lu()
(DomainMatrix([[1, 0], [2, 1]], (2, 2), QQ), DomainMatrix([[1, -1],␣
↪[0, 0]], (2, 2), QQ), [])
```

See also:

lu_solve (page 2679)

lu_solve(*rhs*)

Solver for DomainMatrix x in the A*x = B

Parameters
rhs : DomainMatrix B

Returns
DomainMatrix

x in A*x = B

Raises
DMShapeError

If the DomainMatrix A and rhs have different number of rows

ValueError

If the domain of DomainMatrix A not a Field

Examples

```
>>> from sympy import QQ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
...     [QQ(1), QQ(2)],
...     [QQ(3), QQ(4)]], (2, 2), QQ)
>>> B = DomainMatrix([
...     [QQ(1), QQ(1)],
...     [QQ(0), QQ(1)]], (2, 2), QQ)
```

```
>>> A.lu_solve(B)
DomainMatrix([[-2, -1], [3/2, 1]], (2, 2), QQ)
```

See also:

lu (page 2679)

matmul(*B*)

Performs matrix multiplication of two DomainMatrix matrices

Parameters
A, B: DomainMatrix

to multiply

Returns
DomainMatrix

DomainMatrix after multiplication

Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
...     [ZZ(1), ZZ(2)],
...     [ZZ(3), ZZ(4)]], (2, 2), ZZ)
>>> B = DomainMatrix([
...     [ZZ(1), ZZ(1)],
...     [ZZ(0), ZZ(1)]], (2, 2), ZZ)
```

```
>>> A.matmul(B)
DomainMatrix([[1, 3], [3, 7]], (2, 2), ZZ)
```

See also:

mul (page 2681), *pow* (page 2683), *add* (page 2672), *sub* (page 2686)

mul(*b*)

Performs term by term multiplication for the second DomainMatrix w.r.t first DomainMatrix. Returns a DomainMatrix whose rows are list of DomainMatrix matrices created after term by term multiplication.

Parameters

A, B: DomainMatrix

matrices to multiply term-wise

Returns

DomainMatrix

DomainMatrix after term by term multiplication

Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
...     [ZZ(1), ZZ(2)],
...     [ZZ(3), ZZ(4)]], (2, 2), ZZ)
>>> B = DomainMatrix([
...     [ZZ(1), ZZ(1)],
...     [ZZ(0), ZZ(1)]], (2, 2), ZZ)
```

```
>>> A.mul(B)
DomainMatrix([[DomainMatrix([[1, 1], [0, 1]], (2, 2), ZZ),
DomainMatrix([[2, 2], [0, 2]], (2, 2), ZZ)],
[DomainMatrix([[3, 3], [0, 3]], (2, 2), ZZ),
DomainMatrix([[4, 4], [0, 4]], (2, 2), ZZ)]], (2, 2), ZZ)
```

See also:

matmul (page 2680)

neg()

Returns the negative of DomainMatrix

Parameters

A : Represents a DomainMatrix

Returns

DomainMatrix

DomainMatrix after Negation

Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
...     [ZZ(1), ZZ(2)],
...     [ZZ(3), ZZ(4)]], (2, 2), ZZ)
```

```
>>> A.neg()
DomainMatrix([[-1, -2], [-3, -4]], (2, 2), ZZ)
```

nullspace()

Returns the nullspace for the DomainMatrix

Returns

DomainMatrix

The rows of this matrix form a basis for the nullspace.

Examples

```
>>> from sympy import QQ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
...     [QQ(1), QQ(-1)],
...     [QQ(2), QQ(-2)]], (2, 2), QQ)
>>> A.nullspace()
DomainMatrix([[1, 1]], (1, 2), QQ)
```

classmethod ones(*shape*, *domain*)

Returns a DomainMatrix of 1s, of size shape, belonging to the specified domain

Examples

```
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy import QQ
>>> DomainMatrix.ones((2,3), QQ)
DomainMatrix([[1, 1, 1], [1, 1, 1]], (2, 3), QQ)
```

pow(*n*)

Computes A**n

Parameters

A : DomainMatrix

n : exponent for A

Returns

DomainMatrix

DomainMatrix on computing A**n

Raises

NotImplementedError

if n is negative.

Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
...     [ZZ(1), ZZ(1)],
...     [ZZ(0), ZZ(1)]], (2, 2), ZZ)
```

```
>>> A.pow(2)
DomainMatrix([[1, 2], [0, 1]], (2, 2), ZZ)
```

See also:

matmul (page 2680)

rowspace()

Returns the rowspace for the DomainMatrix

Returns

DomainMatrix

The rows of this matrix form a basis for the rowspace.

Examples

```
>>> from sympy import QQ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
...     [QQ(1), QQ(-1)],
...     [QQ(2), QQ(-2)]], (2, 2), QQ)
>>> A.rowspace()
DomainMatrix([[1, -1]], (1, 2), QQ)
```

rref()

Returns reduced-row echelon form and list of pivots for the DomainMatrix

Returns

(DomainMatrix, list)

reduced-row echelon form and list of pivots for the DomainMatrix

Raises

ValueError

If the domain of DomainMatrix not a Field

Examples

```
>>> from sympy import QQ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
...     [QQ(2), QQ(-1), QQ(0)],
...     [QQ(-1), QQ(2), QQ(-1)],
...     [QQ(0), QQ(0), QQ(2)]], (3, 3), QQ)
```

```
>>> rref_matrix, rref_pivots = A.rref()
>>> rref_matrix
DomainMatrix([[1, 0, 0], [0, 1, 0], [0, 0, 1]], (3, 3), QQ)
>>> rref_pivots
(0, 1, 2)
```

See also:

convert_to (page 2673), *lu* (page 2679)

scc()

Compute the strongly connected components of a DomainMatrix

Returns

List of lists of integers

Each list represents a strongly connected component.

Explanation

A square matrix can be considered as the adjacency matrix for a directed graph where the row and column indices are the vertices. In this graph if there is an edge from vertex `i` to vertex `j` if `M[i, j]` is nonzero. This routine computes the strongly connected components of that graph which are subsets of the rows and columns that are connected by some nonzero element of the matrix. The strongly connected components are useful because many operations such as the determinant can be computed by working with the submatrices corresponding to each component.

Examples

Find the strongly connected components of a matrix:

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> M = DomainMatrix([[ZZ(1), ZZ(0), ZZ(2)],
...                   [ZZ(0), ZZ(3), ZZ(0)],
...                   [ZZ(4), ZZ(6), ZZ(5)]], (3, 3), ZZ)
>>> M.scc()
[[1], [0, 2]]
```

Compute the determinant from the components:

```
>>> MM = M.to_Matrix()
>>> MM
Matrix([
[1, 0, 2],
[0, 3, 0],
[4, 6, 5]])
>>> MM[[1], [1]]
Matrix([[3]])
>>> MM[[0, 2], [0, 2]]
Matrix([
[1, 2],
[4, 5]])
>>> MM.det()
-9
>>> MM[[1], [1]].det() * MM[[0, 2], [0, 2]].det()
-9
```

The components are given in reverse topological order and represent a permutation of the rows and columns that will bring the matrix into block lower-triangular form:

```
>>> MM[[1, 0, 2], [1, 0, 2]]
Matrix([
[3, 0, 0],
[0, 1, 2],
[6, 4, 5]])
```

See also:

sympy.matrices.matrices.MatrixBase.strongly_connected_components (page 1314), *sympy.utilities.iterables.strongly_connected_components*

(page 2094)

sub(*B*)

Subtracts two DomainMatrix matrices of the same Domain

Parameters

A, B: DomainMatrix

matrices to substract

Returns

DomainMatrix

DomainMatrix after Substraction

Raises

DMShapeError

If the dimensions of the two DomainMatrix are not equal

ValueError

If the domain of the two DomainMatrix are not same

Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
...     [ZZ(1), ZZ(2)],
...     [ZZ(3), ZZ(4)]], (2, 2), ZZ)
>>> B = DomainMatrix([
...     [ZZ(4), ZZ(3)],
...     [ZZ(2), ZZ(1)]], (2, 2), ZZ)
```

```
>>> A.sub(B)
DomainMatrix([[-3, -1], [1, 3]], (2, 2), ZZ)
```

See also:

add (page 2672), *matmul* (page 2680)

to_Matrix()

Convert DomainMatrix to Matrix

Returns

Matrix

MutableDenseMatrix for the DomainMatrix

Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
...     [ZZ(1), ZZ(2)],
...     [ZZ(3), ZZ(4)]], (2, 2), ZZ)
```

```
>>> A.to_Matrix()
Matrix([
    [1, 2],
    [3, 4]])
```

See also:

from_Matrix (page 2675)

to_dense()

Return a dense DomainMatrix representation of *self*.

Examples

```
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy import QQ
>>> A = DomainMatrix({0: {0: 1}, 1: {1: 2}}, (2, 2), QQ)
>>> A.rep
{0: {0: 1}, 1: {1: 2}}
>>> B = A.to_dense()
>>> B.rep
[[1, 0], [0, 2]]
```

to_field()

Returns a DomainMatrix with the appropriate field

Returns

DomainMatrix

DomainMatrix with the appropriate field

Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([
...     [ZZ(1), ZZ(2)],
...     [ZZ(3), ZZ(4)]], (2, 2), ZZ)
```

```
>>> A.to_field()
DomainMatrix([[1, 2], [3, 4]], (2, 2), QQ)
```

to_sparse()

Return a sparse DomainMatrix representation of *self*.

Examples

```
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy import QQ
>>> A = DomainMatrix([[1, 0],[0, 2]], (2, 2), QQ)
>>> A.rep
[[1, 0], [0, 2]]
>>> B = A.to_sparse()
>>> B.rep
{0: {0: 1}, 1: {1: 2}}
```

transpose()

Matrix transpose of `self`

unify(**others, fmt=None*)

Unifies the domains and the format of self and other matrices.

Parameters

others : DomainMatrix

fmt: string 'dense', 'sparse' or `None` (default)

The preferred format to convert to if self and other are not already in the same format. If *None* or not specified then no conversion if performed.

Returns

Tuple[DomainMatrix]

Matrices with unified domain and format

Examples

Unify the domain of DomainMatrix that have different domains:

```
>>> from sympy import ZZ, QQ
>>> from sympy.polys.matrices import DomainMatrix
>>> A = DomainMatrix([[ZZ(1), ZZ(2)]], (1, 2), ZZ)
>>> B = DomainMatrix([[QQ(1, 2), QQ(2)]], (1, 2), QQ)
>>> Aq, Bq = A.unify(B)
>>> Aq
DomainMatrix([[1, 2]], (1, 2), QQ)
>>> Bq
DomainMatrix([[1/2, 2]], (1, 2), QQ)
```

Unify the format (dense or sparse):

```
>>> A = DomainMatrix([[ZZ(1), ZZ(2)]], (1, 2), ZZ)
>>> B = DomainMatrix({0:{0: ZZ(1)}}, (2, 2), ZZ)
>>> B.rep
{0: {0: 1}}
```

```
>>> A2, B2 = A.unify(B, fmt='dense')
>>> B2.rep
[[1, 0], [0, 0]]
```

See also:

convert_to (page 2673), *to_dense* (page 2687), *to_sparse* (page 2687)

vstack(**B*)

Vertically stack the given matrices.

Parameters

B: DomainMatrix

Matrices to stack vertically.

Returns

DomainMatrix

DomainMatrix by stacking vertically.

Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
```

```
>>> A = DomainMatrix([[ZZ(1), ZZ(2)], [ZZ(3), ZZ(4)]], (2, 2), ZZ)
>>> B = DomainMatrix([[ZZ(5), ZZ(6)], [ZZ(7), ZZ(8)]], (2, 2), ZZ)
>>> A.vstack(B)
DomainMatrix([[1, 2], [3, 4], [5, 6], [7, 8]], (4, 2), ZZ)
```

```
>>> C = DomainMatrix([[ZZ(9), ZZ(10)], [ZZ(11), ZZ(12)]], (2, 2), ZZ)
>>> A.vstack(B, C)
DomainMatrix([[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12]], (6,␣
↪2), ZZ)
```

See also:

unify (page 2688)

classmethod zeros(*shape*, *domain*, ***, *fmt='sparse'*)

Returns a zero DomainMatrix of size shape, belonging to the specified domain

Examples

```
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy import QQ
>>> DomainMatrix.zeros((2, 3), QQ)
DomainMatrix({}, (2, 3), QQ)
```

DDM Class Reference

class sympy.polys.matrices.ddm.**DDM**(*rowslist*, *shape*, *domain*)

Dense matrix based on polys domain elements

This is a list subclass and is a wrapper for a list of lists that supports basic matrix arithmetic +, -, , *.

add(*b*)

a + b

charpoly()

Coefficients of characteristic polynomial of a

det()

Determinant of a

hstack(**B*)

Horizontally stacks *DDM* (page 2690) matrices.

Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices.sdm import DDM
```

```
>>> A = DDM([[ZZ(1), ZZ(2)], [ZZ(3), ZZ(4)]], (2, 2), ZZ)
>>> B = DDM([[ZZ(5), ZZ(6)], [ZZ(7), ZZ(8)]], (2, 2), ZZ)
>>> A.hstack(B)
[[1, 2, 5, 6], [3, 4, 7, 8]]
```

```
>>> C = DDM([[ZZ(9), ZZ(10)], [ZZ(11), ZZ(12)]], (2, 2), ZZ)
>>> A.hstack(B, C)
[[1, 2, 5, 6, 9, 10], [3, 4, 7, 8, 11, 12]]
```

inv()

Inverse of a

is_lower()

Says whether this matrix is lower-triangular. True can be returned even if the matrix is not square.

is_upper()

Says whether this matrix is upper-triangular. True can be returned even if the matrix is not square.

is_zero_matrix()

Says whether this matrix has all zero entries.

lu()

L, U decomposition of a

lu_solve(*b*)

x where a*x = b

matmul(*b*)

a @ b (matrix product)

neg()

-a

rref()

Reduced-row echelon form of a and list of pivots

scc()

Strongly connected components of a square matrix *a*.

Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices.sdm import DDM
>>> A = DDM([[ZZ(1), ZZ(0)], [ZZ(0), ZZ(1)]], (2, 2), ZZ)
>>> A.scc()
[[0], [1]]
```

See also:

sympy.polys.matrices.domainmatrix.DomainMatrix.scc (page 2684)

sub(*b*)

a - b

vstack(**B*)

Vertically stacks *DDM* (page 2690) matrices.

Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices.sdm import DDM
```

```
>>> A = DDM([[ZZ(1), ZZ(2)], [ZZ(3), ZZ(4)]], (2, 2), ZZ)
>>> B = DDM([[ZZ(5), ZZ(6)], [ZZ(7), ZZ(8)]], (2, 2), ZZ)
>>> A.vstack(B)
[[1, 2], [3, 4], [5, 6], [7, 8]]
```

```
>>> C = DDM([[ZZ(9), ZZ(10)], [ZZ(11), ZZ(12)]], (2, 2), ZZ)
>>> A.vstack(B, C)
[[1, 2], [3, 4], [5, 6], [7, 8], [9, 10], [11, 12]]
```

SDM Class Reference

class sympy.polys.matrices.sdm.**SDM**(*elemsdict*, *shape*, *domain*)

Sparse matrix based on polys domain elements

This is a dict subclass and is a wrapper for a dict of dicts that supports basic matrix arithmetic +, -, , *.

In order to create a new *SDM* (page 2692), a dict of dicts mapping non-zero elements to their corresponding row and column in the matrix is needed.

We also need to specify the shape and *Domain* (page 2504) of our *SDM* (page 2692) object.

We declare a 2x2 *SDM* (page 2692) matrix belonging to QQ domain as shown below. The 2x2 Matrix in the example is

$$A = \begin{bmatrix} 0 & \frac{1}{2} \\ 0 & 0 \end{bmatrix}$$

```
>>> from sympy.polys.matrices.sdm import SDM
>>> from sympy import QQ
>>> elemsdict = {0:{1:QQ(1, 2)}}
>>> A = SDM(elemsdict, (2, 2), QQ)
>>> A
{0: {1: 1/2}}
```

We can manipulate *SDM* (page 2692) the same way as a Matrix class

```
>>> from sympy import ZZ
>>> A = SDM({0:{1: ZZ(2)}, 1:{0:ZZ(1)}}, (2, 2), ZZ)
>>> B  = SDM({0:{0: ZZ(3)}, 1:{1:ZZ(4)}}, (2, 2), ZZ)
>>> A + B
{0: {0: 3, 1: 2}, 1: {0: 1, 1: 4}}
```

Multiplication

```
>>> A*B
{0: {1: 8}, 1: {0: 3}}
>>> A*ZZ(2)
{0: {1: 4}, 1: {0: 2}}
```

add(*B*)

Adds two *SDM* (page 2692) matrices

Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices.sdm import SDM
>>> A = SDM({0:{1: ZZ(2)}, 1:{0:ZZ(1)}}, (2, 2), ZZ)
>>> B = SDM({0:{0: ZZ(3)}, 1:{1:ZZ(4)}}, (2, 2), ZZ)
>>> A.add(B)
{0: {0: 3, 1: 2}, 1: {0: 1, 1: 4}}
```

charpoly()

Returns the coefficients of the characteristic polynomial of the *SDM* (page 2692) matrix. These elements will be domain elements. The domain of the elements will be same as domain of the *SDM* (page 2692).

Examples

```
>>> from sympy import QQ, Symbol
>>> from sympy.polys.matrices.sdm import SDM
>>> from sympy.polys import Poly
>>> A = SDM({0:{0:QQ(1), 1:QQ(2)}, 1:{0:QQ(3), 1:QQ(4)}}, (2, 2), QQ)
>>> A.charpoly()
[1, -5, -2]
```

We can create a polynomial using the coefficients using *Poly* (page 2378)

```
>>> x = Symbol('x')
>>> p = Poly(A.charpoly(), x, domain=A.domain)
>>> p
Poly(x**2 - 5*x - 2, x, domain='QQ')
```

convert_to(K)

Converts the *Domain* (page 2504) of a *SDM* (page 2692) matrix to K

Examples

```
>>> from sympy import ZZ, QQ
>>> from sympy.polys.matrices.sdm import SDM
>>> A = SDM({0:{1: ZZ(2)}, 1:{0:ZZ(1)}}, (2, 2), ZZ)
>>> A.convert_to(QQ)
{0: {1: 2}, 1: {0: 1}}
```

copy()

Returns the copy of a *SDM* (page 2692) object

Examples

```
>>> from sympy.polys.matrices.sdm import SDM
>>> from sympy import QQ
>>> elemsdict = {0:{1:QQ(2)}, 1:{}}
>>> A = SDM(elemsdict, (2, 2), QQ)
>>> B = A.copy()
>>> B
{0: {1: 2}, 1: {}}
```

det()

Returns determinant of A

Examples

```
>>> from sympy import QQ
>>> from sympy.polys.matrices.sdm import SDM
>>> A = SDM({0:{0:QQ(1), 1:QQ(2)}, 1:{0:QQ(3), 1:QQ(4)}}, (2, 2), QQ)
>>> A.det()
-2
```

classmethod eye(*shape*, *domain*)

Returns a identity *SDM* (page 2692) matrix of dimensions size x size, belonging to the specified domain

Examples

```
>>> from sympy.polys.matrices.sdm import SDM
>>> from sympy import QQ
>>> I = SDM.eye((2, 2), QQ)
>>> I
{0: {0: 1}, 1: {1: 1}}
```

classmethod from_ddm(*ddm*)

converts object of *DDM* (page 2690) to *SDM* (page 2692)

Examples

```
>>> from sympy.polys.matrices.ddm import DDM
>>> from sympy.polys.matrices.sdm import SDM
>>> from sympy import QQ
>>> ddm = DDM( [[QQ(1, 2), 0], [0, QQ(3, 4)]], (2, 2), QQ)
>>> A = SDM.from_ddm(ddm)
>>> A
{0: {0: 1/2}, 1: {1: 3/4}}
```

classmethod from_list(*ddm*, *shape*, *domain*)

Parameters

ddm:

list of lists containing domain elements

shape:

Dimensions of *SDM* (page 2692) matrix

domain:

Represents *Domain* (page 2504) of *SDM* (page 2692) object

Returns

SDM (page 2692) containing elements of ddm

Examples

```
>>> from sympy.polys.matrices.sdm import SDM
>>> from sympy import QQ
>>> ddm = [[QQ(1, 2), QQ(0)], [QQ(0), QQ(3, 4)]]
>>> A = SDM.from_list(ddm, (2, 2), QQ)
>>> A
{0: {0: 1/2}, 1: {1: 3/4}}
```

hstack(**B*)

Horizontally stacks *SDM* (page 2692) matrices.

Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices.sdm import SDM
```

```
>>> A = SDM({0: {0: ZZ(1), 1: ZZ(2)}, 1: {0: ZZ(3), 1: ZZ(4)}}, (2,␣
↪2), ZZ)
>>> B = SDM({0: {0: ZZ(5), 1: ZZ(6)}, 1: {0: ZZ(7), 1: ZZ(8)}}, (2,␣
↪2), ZZ)
>>> A.hstack(B)
{0: {0: 1, 1: 2, 2: 5, 3: 6}, 1: {0: 3, 1: 4, 2: 7, 3: 8}}
```

```
>>> C = SDM({0: {0: ZZ(9), 1: ZZ(10)}, 1: {0: ZZ(11), 1: ZZ(12)}}, (2,
↪ 2), ZZ)
>>> A.hstack(B, C)
{0: {0: 1, 1: 2, 2: 5, 3: 6, 4: 9, 5: 10}, 1: {0: 3, 1: 4, 2: 7, 3: 8,
↪ 4: 11, 5: 12}}
```

inv()

Returns inverse of a matrix A

Examples

```
>>> from sympy import QQ
>>> from sympy.polys.matrices.sdm import SDM
>>> A = SDM({0:{0:QQ(1), 1:QQ(2)}, 1:{0:QQ(3), 1:QQ(4)}}, (2, 2), QQ)
>>> A.inv()
{0: {0: -2, 1: 1}, 1: {0: 3/2, 1: -1/2}}
```

is_lower()

Says whether this matrix is lower-triangular. True can be returned even if the matrix is not square.

is_upper()

Says whether this matrix is upper-triangular. True can be returned even if the matrix is not square.

is_zero_matrix()

Says whether this matrix has all zero entries.

lu()

Returns LU decomposition for a matrix A

Examples

```
>>> from sympy import QQ
>>> from sympy.polys.matrices.sdm import SDM
>>> A = SDM({0:{0:QQ(1), 1:QQ(2)}, 1:{0:QQ(3), 1:QQ(4)}}, (2, 2), QQ)
>>> A.lu()
({0: {0: 1}, 1: {0: 3, 1: 1}}, {0: {0: 1, 1: 2}, 1: {1: -2}}, [])
```

lu_solve(*b*)

Uses LU decomposition to solve Ax = b,

Examples

```
>>> from sympy import QQ
>>> from sympy.polys.matrices.sdm import SDM
>>> A = SDM({0:{0:QQ(1), 1:QQ(2)}, 1:{0:QQ(3), 1:QQ(4)}}, (2, 2), QQ)
>>> b = SDM({0:{0:QQ(1)}, 1:{0:QQ(2)}}, (2, 1), QQ)
>>> A.lu_solve(b)
{1: {0: 1/2}}
```

matmul(*B*)

Performs matrix multiplication of two SDM matrices

Parameters
A, B: SDM to multiply

Returns
SDM

SDM after multiplication

Raises
DomainError

If domain of A does not match with that of B

Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices.sdm import SDM
>>> A = SDM({0:{1: ZZ(2)}, 1:{0:ZZ(1)}}, (2, 2), ZZ)
>>> B = SDM({0:{0:ZZ(2), 1:ZZ(3)}, 1:{0:ZZ(4)}}, (2, 2), ZZ)
>>> A.matmul(B)
{0: {0: 8}, 1: {0: 2, 1: 3}}
```

mul(*b*)

Multiplies each element of A with a scalar b

Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices.sdm import SDM
>>> A = SDM({0:{1: ZZ(2)}, 1:{0:ZZ(1)}}, (2, 2), ZZ)
>>> A.mul(ZZ(3))
{0: {1: 6}, 1: {0: 3}}
```

neg()

Returns the negative of a *SDM* (page 2692) matrix

Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices.sdm import SDM
>>> A = SDM({0:{1: ZZ(2)}, 1:{0:ZZ(1)}}, (2, 2), ZZ)
>>> A.neg()
{0: {1: -2}, 1: {0: -1}}
```

classmethod new(*sdm*, *shape*, *domain*)

Parameters

sdm: A dict of dicts for non-zero elements in SDM

shape: tuple representing dimension of SDM

domain: Represents :py:class:`~.Domain` of SDM

Returns

An *SDM* (page 2692) object

Examples

```
>>> from sympy.polys.matrices.sdm import SDM
>>> from sympy import QQ
>>> elemsdict = {0:{1: QQ(2)}}
>>> A = SDM.new(elemsdict, (2, 2), QQ)
>>> A
{0: {1: 2}}
```

nullspace()

Returns nullspace for a *SDM* (page 2692) matrix A

Examples

```
>>> from sympy import QQ
>>> from sympy.polys.matrices.sdm import SDM
>>> A = SDM({0:{0:QQ(1), 1:QQ(2)}, 1:{0: QQ(2), 1: QQ(4)}}, (2, 2),␣
↪QQ)
>>> A.nullspace()
({0: {0: -2, 1: 1}}, [1])
```

rref()

Returns reduced-row echelon form and list of pivots for the *SDM* (page 2692)

Examples

```
>>> from sympy import QQ
>>> from sympy.polys.matrices.sdm import SDM
>>> A = SDM({0:{0:QQ(1), 1:QQ(2)}, 1:{0:QQ(2), 1:QQ(4)}}, (2, 2), QQ)
>>> A.rref()
({0: {0: 1, 1: 2}}, [0])
```

scc()

Strongly connected components of a square matrix A.

Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices.sdm import SDM
>>> A = SDM({0:{0: ZZ(2)}, 1:{1:ZZ(1)}}, (2, 2), ZZ)
>>> A.scc()
[[0], [1]]
```

See also:

sympy.polys.matrices.domainmatrix.DomainMatrix.scc (page 2684)

sub(B)

Subtracts two *SDM* (page 2692) matrices

Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices.sdm import SDM
>>> A = SDM({0:{1: ZZ(2)}, 1:{0:ZZ(1)}}, (2, 2), ZZ)
>>> B  = SDM({0:{0: ZZ(3)}, 1:{1:ZZ(4)}}, (2, 2), ZZ)
>>> A.sub(B)
{0: {0: -3, 1: 2}, 1: {0: 1, 1: -4}}
```

to_ddm()

Convert a *SDM* (page 2692) object to a *DDM* (page 2690) object

Examples

```
>>> from sympy.polys.matrices.sdm import SDM
>>> from sympy import QQ
>>> A = SDM({0:{1:QQ(2)}, 1:{}}, (2, 2), QQ)
>>> A.to_ddm()
[[0, 2], [0, 0]]
```

to_list()

Converts a *SDM* (page 2692) object to a list

Examples

```
>>> from sympy.polys.matrices.sdm import SDM
>>> from sympy import QQ
>>> elemsdict = {0:{1:QQ(2)}, 1:{}}
>>> A = SDM(elemsdict, (2, 2), QQ)
>>> A.to_list()
[[0, 2], [0, 0]]
```

transpose()

Returns the transpose of a *SDM* (page 2692) matrix

Examples

```
>>> from sympy.polys.matrices.sdm import SDM
>>> from sympy import QQ
>>> A = SDM({0:{1:QQ(2)}, 1:{}}, (2, 2), QQ)
>>> A.transpose()
{1: {0: 2}}
```

vstack(**B*)

Vertically stacks *SDM* (page 2692) matrices.

Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices.sdm import SDM
```

```
>>> A = SDM({0: {0: ZZ(1), 1: ZZ(2)}, 1: {0: ZZ(3), 1: ZZ(4)}}, (2,␣
↪2), ZZ)
>>> B = SDM({0: {0: ZZ(5), 1: ZZ(6)}, 1: {0: ZZ(7), 1: ZZ(8)}}, (2,␣
↪2), ZZ)
>>> A.vstack(B)
{0: {0: 1, 1: 2}, 1: {0: 3, 1: 4}, 2: {0: 5, 1: 6}, 3: {0: 7, 1: 8}}
```

```
>>> C = SDM({0: {0: ZZ(9), 1: ZZ(10)}, 1: {0: ZZ(11), 1: ZZ(12)}}, (2,
 ↪ 2), ZZ)
>>> A.vstack(B, C)
{0: {0: 1, 1: 2}, 1: {0: 3, 1: 4}, 2: {0: 5, 1: 6}, 3: {0: 7, 1: 8},␣
 ↪4: {0: 9, 1: 10}, 5: {0: 11, 1: 12}}
```

classmethod zeros(*shape*, *domain*)

Returns a *SDM* (page 2692) of size shape, belonging to the specified domain

In the example below we declare a matrix A where,

$$A := \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

```
>>> from sympy.polys.matrices.sdm import SDM
>>> from sympy import QQ
>>> A = SDM.zeros((2, 3), QQ)
>>> A
{}
```

Normal Forms

sympy.polys.matrices.normalforms.**smith_normal_form**(*m*)

Return the Smith Normal Form of a matrix m over the ring $domain$. This will only work if the ring is a principal ideal domain.

Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy.polys.matrices.normalforms import smith_normal_form
>>> m = DomainMatrix([[ZZ(12), ZZ(6), ZZ(4)],
...                   [ZZ(3), ZZ(9), ZZ(6)],
...                   [ZZ(2), ZZ(16), ZZ(14)]], (3, 3), ZZ)
>>> print(smith_normal_form(m).to_Matrix())
Matrix([[1, 0, 0], [0, 10, 0], [0, 0, -30]])
```

sympy.polys.matrices.normalforms.**hermite_normal_form**(*A*, ***, *D=None*, *check_rank=False*)

Compute the Hermite Normal Form of *DomainMatrix* (page 2671) A over *ZZ* (page 2525).

Parameters

A : $m \times n$ DomainMatrix over *ZZ* (page 2525).

D : *ZZ* (page 2525), optional

Let W be the HNF of A. If known in advance, a positive integer D being any multiple of $\det(W)$ may be provided. In this case, if A also has rank m, then we may use an alternative algorithm that works mod D in order to prevent coefficient explosion.

check_rank : boolean, optional (default=False)

The basic assumption is that, if you pass a value for D, then you already believe that A has rank m, so we do not waste time checking it for you. If you do want this to be checked (and the ordinary, non-modulo D algorithm to be used if the check fails), then set *check_rank* to `True`.

Returns

DomainMatrix (page 2671)

The HNF of matrix A.

Raises

DMDomainError

If the domain of the matrix is not *ZZ* (page 2525), or if D is given but is not in *ZZ* (page 2525).

DMShapeError

If the mod D algorithm is used but the matrix has more rows than columns.

Examples

```
>>> from sympy import ZZ
>>> from sympy.polys.matrices import DomainMatrix
>>> from sympy.polys.matrices.normalforms import hermite_normal_form
>>> m = DomainMatrix([[ZZ(12), ZZ(6), ZZ(4)],
...                   [ZZ(3), ZZ(9), ZZ(6)],
...                   [ZZ(2), ZZ(16), ZZ(14)]], (3, 3), ZZ)
>>> print(hermite_normal_form(m).to_Matrix())
Matrix([[10, 0, 2], [0, 15, 3], [0, 0, 2]])
```

References

[R694]

Number Fields

Introduction

Like many other computations in algebraic number theory, the splitting of rational primes can be treated by *rational* methods only. This fact is very important if computation by automatic computing machinery is considered. Only the knowledge of the irreducible polynomial $f(x)$, a zero of which generates the field in question, is needed.

—Olga Taussky, 1953

Concepts like number fields and algebraic numbers are essential to our understanding of algebraic number theory, but to the computer the subject is all about polynomials: the ring $\mathbb{Q}[x]$ reduced modulo irreducible polynomials $f(x) \in \mathbb{Q}[x]$. It thus finds a natural home under the *polys* (page 2360) module in SymPy.

Various authors (such as Taussky, Zimmer, Pohst and Zassenhaus, or Cohen) have articulated the main goals of computational algebraic number theory in different ways, but invariably the list centers around a certain essential set of tasks. As a goal for the `numberfields` module in SymPy, we may set the following list, based on [Cohen93], Sec. 4.9.3.

For a number field $K = \mathbb{Q}(\theta)$, whose ring of algebraic integers is denoted $\mathbb{Z}_K$, compute:

1. an integral basis of $\mathbb{Z}_K$
2. the decomposition of rational primes in $\mathbb{Z}_K$
3. $\mathfrak{p}$-adic valuations for ideals and elements
4. the Galois group of the Galois closure of K
5. a system of fundamental units of K
6. the regulator $R(K)$
7. the class number
8. the structure of the class group $Cl(K)$
9. decide whether a given ideal is principal, and if so compute a generator.

As a foundation, and to support our basic ability to define and work with number fields and algebraic numbers, we also set the following problems, following [Cohen93], Sec. 4.5.

10. Given an algebraic number – expressed by radicals and rational operations, or even as a special value of a transcendental function – determine its minimal polynomial over $\mathbb{Q}$.
11. The Subfield Problem: Given two number fields $\mathbb{Q}(\alpha)$, $\mathbb{Q}(\beta)$ via the minimal polynomials for their generators α and β, decide whether one field is isomorphic to a subfield of the other, and if so exhibit an embedding.
12. The Field Membership Problem: Given two algebraic numbers α, β, decide whether $\alpha \in \mathbb{Q}(\beta)$, and if so write $\alpha = f(\beta)$ for some $f(x) \in \mathbb{Q}[x]$.
13. The Primitive Element Problem: Given several algebraic numbers $\alpha_1, \ldots, \alpha_m$, compute a single algebraic number θ such that $\mathbb{Q}(\alpha_1, \ldots, \alpha_m) = \mathbb{Q}(\theta)$.

At present only a subset of the tasks enumerated above is yet supported in SymPy, and if you are interested in expanding support, you are encouraged to contribute! An excellent source, providing solutions to all the remaining problems (as well as those already solved) is [Cohen93].

At time of writing, the existing solutions to the above problems are found in the following places:

Task	Implementation
(1) integral basis	*round_two()* (page 2703)
(2) prime decomposition	*prime_decomp()* (page 2705)
(3) p-adic valuation	*prime_valuation()* (page 2709)
(10) find minimal polynomial	*minimal_polynomial()* (page 2710)
(11) subfield	*field_isomorphism()* (page 2711)
(12) field membership	*to_number_field()* (page 2714)
(13) primitive element	*primitive_element()* (page 2712)

Solving the Main Problems

Integral Basis

sympy.polys.numberfields.basis.**round_two**(*T*, *radicals=None*)

Zassenhaus's "Round 2" algorithm.

Parameters

T : *Poly* (page 2378), *AlgebraicField* (page 2539)

Either (1) the irreducible monic polynomial over *ZZ* (page 2525) defining the number field, or (2) an *AlgebraicField* (page 2539) representing the number field itself.

radicals : dict, optional

This is a way for any p-radicals (if computed) to be returned by reference. If desired, pass an empty dictionary. If the algorithm reaches the point where it computes the nilradical mod p of the ring of integers Z_K, then an $\mathbb{F}_p$-basis for this ideal will be stored in this dictionary under the key `p`. This can be useful for other algorithms, such as prime decomposition.

Returns

Pair `(ZK, dK)`, where:

`ZK` is a *Submodule* (page 2726) representing the maximal order.

`dK` is the discriminant of the field $K = \mathbb{Q}[x]/(T(x))$.

Explanation

Carry out Zassenhaus's "Round 2" algorithm on a monic irreducible polynomial T over *ZZ* (page 2525). This computes an integral basis and the discriminant for the field $K = \mathbb{Q}[x]/(T(x))$.

Alternatively, you may pass an *AlgebraicField* (page 2539) instance, in place of the polynomial T, in which case the algorithm is applied to the minimal polynomial for the field's primitive element.

Ordinarily this function need not be called directly, as one can instead access the *maximal_order()* (page 2545), *integral_basis()* (page 2544), and *discriminant()* (page 2543) methods of an *AlgebraicField* (page 2539).

Examples

Working through an AlgebraicField:

```
>>> from sympy import Poly, QQ
>>> from sympy.abc import x
>>> T = Poly(x ** 3 + x ** 2 - 2 * x + 8)
>>> K = QQ.alg_field_from_poly(T, "theta")
>>> print(K.maximal_order())
Submodule[[2, 0, 0], [0, 2, 0], [0, 1, 1]]/2
>>> print(K.discriminant())
-503
>>> print(K.integral_basis(fmt='sympy'))
[1, theta, theta/2 + theta**2/2]
```

Calling directly:

```
>>> from sympy import Poly
>>> from sympy.abc import x
>>> from sympy.polys.numberfields.basis import round_two
>>> T = Poly(x ** 3 + x ** 2 - 2 * x + 8)
>>> print(round_two(T))
(Submodule[[2, 0, 0], [0, 2, 0], [0, 1, 1]]/2, -503)
```

The nilradicals mod p that are sometimes computed during the Round Two algorithm may be useful in further calculations. Pass a dictionary under $radicals$ to receive these:

```
>>> T = Poly(x**3 + 3*x**2 + 5)
>>> rad = {}
>>> ZK, dK = round_two(T, radicals=rad)
>>> print(rad)
{3: Submodule[[-1, 1, 0], [-1, 0, 1]]}
```

See also:

AlgebraicField.maximal_order (page 2545), *AlgebraicField.integral_basis* (page 2544), *AlgebraicField.discriminant* (page 2543)

References

[R713]

Prime Decomposition

sympy.polys.numberfields.primes.**prime_decomp**(*p*, *T=None*, *ZK=None*, *dK=None*, *radical=None*)

Compute the decomposition of rational prime p in a number field.

Parameters

p : int

The rational prime whose decomposition is desired.

T : *Poly* (page 2378), optional

Monic irreducible polynomial defining the number field K in which to factor. NOTE: at least one of T or ZK must be provided.

ZK : *Submodule* (page 2726), optional

The maximal order for K, if already known. NOTE: at least one of T or ZK must be provided.

dK : int, optional

The discriminant of the field K, if already known.

radical : *Submodule* (page 2726), optional

The nilradical mod p in the integers of K, if already known.

Returns

List of *PrimeIdeal* (page 2706) instances.

Explanation

Ordinarily this should be accessed through the *primes_above()* (page 2545) method of an *AlgebraicField* (page 2539).

Examples

```
>>> from sympy import Poly, QQ
>>> from sympy.abc import x, theta
>>> T = Poly(x ** 3 + x ** 2 - 2 * x + 8)
>>> K = QQ.algebraic_field((T, theta))
>>> print(K.primes_above(2))
[[ (2, x**2 + 1) e=1, f=1 ], [ (2, (x**2 + 3*x + 2)/2) e=1, f=1 ],
 [ (2, (3*x**2 + 3*x)/2) e=1, f=1 ]]
```

References

[R714]

class sympy.polys.numberfields.primes.**PrimeIdeal**(*ZK*, *p*, *alpha*, *f*, *e=None*)

A prime ideal in a ring of algebraic integers.

__init__(*ZK*, *p*, *alpha*, *f*, *e=None*)

Parameters

ZK : *Submodule* (page 2726)

The maximal order where this ideal lives.

p : int

The rational prime this ideal divides.

alpha : *PowerBasisElement* (page 2732)

Such that the ideal is equal to `p*ZK + alpha*ZK`.

f : int

The inertia degree.

e : int, `None`, optional

The ramification index, if already known. If `None`, we will compute it here.

__add__(*other*)

Convert to a *Submodule* (page 2726) and add to another *Submodule* (page 2726).

See also:

as_submodule (page 2706)

__mul__(*other*)

Convert to a *Submodule* (page 2726) and multiply by another *Submodule* (page 2726) or a rational number.

See also:

as_submodule (page 2706)

as_submodule()

Represent this prime ideal as a *Submodule* (page 2726).

Returns

Submodule (page 2726)

Will be equal to `self.p * self.ZK + self.alpha * self.ZK`.

Explanation

The *PrimeIdeal* (page 2706) class serves to bundle information about a prime ideal, such as its inertia degree, ramification index, and two-generator representation, as well as to offer helpful methods like *valuation()* (page 2709) and *test_factor()* (page 2709).

However, in order to be added and multiplied by other ideals or rational numbers, it must first be converted into a *Submodule* (page 2726), which is a class that supports these operations.

In many cases, the user need not perform this conversion deliberately, since it is automatically performed by the arithmetic operator methods *__add__()* (page 2706) and *__mul__()* (page 2706).

Raising a *PrimeIdeal* (page 2706) to a non-negative integer power is also supported.

Examples

```
>>> from sympy import Poly, cyclotomic_poly, prime_decomp
>>> T = Poly(cyclotomic_poly(7))
>>> P0 = prime_decomp(7, T)[0]
>>> print(P0**6 == 7*P0.ZK)
True
```

Note that, on both sides of the equation above, we had a *Submodule* (page 2726). In the next equation we recall that adding ideals yields their GCD. This time, we need a deliberate conversion to *Submodule* (page 2726) on the right:

```
>>> print(P0 + 7*P0.ZK == P0.as_submodule())
True
```

See also:

__add__ (page 2706), *__mul__* (page 2706)

property is_inert

Say whether the rational prime we divide is inert, i.e. stays prime in our ring of integers.

reduce_ANP(*a*)

Reduce an *ANP* (page 2568) to a “small representative” modulo this prime ideal.

Parameters

elt : *ANP* (page 2568)

The element to be reduced.

Returns

ANP (page 2568)

The reduced element.

See also:

reduce_element (page 2708), *reduce_alg_num* (page 2707), *Submodule.reduce_element* (page 2728)

reduce_alg_num(*a*)

Reduce an *AlgebraicNumber* (page 988) to a "small representative" modulo this prime ideal.

Parameters

elt : *AlgebraicNumber* (page 988)

The element to be reduced.

Returns

AlgebraicNumber (page 988)

The reduced element.

See also:

reduce_element (page 2708), *reduce_ANP* (page 2707), *Submodule.reduce_element* (page 2728)

reduce_element(*elt*)

Reduce a *PowerBasisElement* (page 2732) to a "small representative" modulo this prime ideal.

Parameters

elt : *PowerBasisElement* (page 2732)

The element to be reduced.

Returns

PowerBasisElement (page 2732)

The reduced element.

See also:

reduce_ANP (page 2707), *reduce_alg_num* (page 2707), *Submodule.reduce_element* (page 2728)

repr(*field_gen=None*, *just_gens=False*)

Print a representation of this prime ideal.

Parameters

field_gen : *Symbol* (page 976), None, optional (default=None)

The symbol to use for the generator of the field. This will appear in our representation of `self.alpha`. If `None`, we use the variable of the defining polynomial of `self.ZK`.

just_gens : bool, optional (default=False)

If `True`, just print the "(p, alpha)" part, showing "just the generators" of the prime ideal. Otherwise, print a string of the form "[(p, alpha) e=..., f=...]", giving the ramification index and inertia degree, along with the generators.

Examples

```
>>> from sympy import cyclotomic_poly, QQ
>>> from sympy.abc import x, zeta
>>> T = cyclotomic_poly(7, x)
>>> K = QQ.algebraic_field((T, zeta))
>>> P = K.primes_above(11)
>>> print(P[0].repr())
[ (11, x**3 + 5*x**2 + 4*x - 1) e=1, f=3 ]
>>> print(P[0].repr(field_gen=zeta))
[ (11, zeta**3 + 5*zeta**2 + 4*zeta - 1) e=1, f=3 ]
>>> print(P[0].repr(field_gen=zeta, just_gens=True))
(11, zeta**3 + 5*zeta**2 + 4*zeta - 1)
```

test_factor()

Compute a test factor for this prime ideal.

Explanation

Write $\mathfrak{p}$ for this prime ideal, p for the rational prime it divides. Then, for computing $\mathfrak{p}$-adic valuations it is useful to have a number $\beta \in \mathbb{Z}_K$ such that $p/\mathfrak{p} = p\mathbb{Z}_K + \beta\mathbb{Z}_K$.

Essentially, this is the same as the number Ψ (or the "reagent") from Kummer's 1847 paper (*Ueber die Zerlegung...*, Crelle vol. 35) in which ideal divisors were invented.

valuation(I)

Compute the $\mathfrak{p}$-adic valuation of integral ideal I at this prime ideal.

Parameters

I : *Submodule* (page 2726)

See also:

prime_valuation (page 2709)

p-adic Valuation

sympy.polys.numberfields.primes.**prime_valuation**(I, P)

Compute the P-adic valuation for an integral ideal I.

Parameters

I : *Submodule* (page 2726)

An integral ideal whose valuation is desired.

P : *PrimeIdeal* (page 2706)

The prime at which to compute the valuation.

Returns

int

Examples

```
>>> from sympy import QQ
>>> from sympy.polys.numberfields import prime_valuation
>>> K = QQ.cyclotomic_field(5)
>>> P = K.primes_above(5)
>>> ZK = K.maximal_order()
>>> print(prime_valuation(25*ZK, P[0]))
8
```

See also:

PrimeIdeal.valuation (page 2709)

References

[R715]

Finding Minimal Polynomials

sympy.polys.numberfields.minpoly.**minimal_polynomial**(*ex*, *x=None*, *compose=True*, *polys=False*, *domain=None*)

Computes the minimal polynomial of an algebraic element.

Parameters

ex : Expr

Element or expression whose minimal polynomial is to be calculated.

x : Symbol, optional

Independent variable of the minimal polynomial

compose : boolean, optional (default=True)

Method to use for computing minimal polynomial. If `compose=True` (default) then `_minpoly_compose` is used, if `compose=False` then groebner bases are used.

polys : boolean, optional (default=False)

If `True` returns a `Poly` object else an `Expr` object.

domain : Domain, optional

Ground domain

Notes

By default `compose=True`, the minimal polynomial of the subexpressions of `ex` are computed, then the arithmetic operations on them are performed using the resultant and factorization. If `compose=False`, a bottom-up algorithm is used with `groebner`. The default algorithm stalls less frequently.

If no ground domain is given, it will be generated automatically from the expression.

Examples

```
>>> from sympy import minimal_polynomial, sqrt, solve, QQ
>>> from sympy.abc import x, y
```

```
>>> minimal_polynomial(sqrt(2), x)
x**2 - 2
>>> minimal_polynomial(sqrt(2), x, domain=QQ.algebraic_field(sqrt(2)))
x - sqrt(2)
>>> minimal_polynomial(sqrt(2) + sqrt(3), x)
x**4 - 10*x**2 + 1
>>> minimal_polynomial(solve(x**3 + x + 3)[0], x)
x**3 + x + 3
>>> minimal_polynomial(sqrt(y), x)
x**2 - y
```

sympy.polys.numberfields.minpoly.**minpoly**(*ex*, *x=None*, *compose=True*, *polys=False*, *domain=None*)

This is a synonym for *minimal_polynomial()* (page 2710).

The Subfield Problem

Functions in `polys.numberfields.subfield` solve the "Subfield Problem" and allied problems, for algebraic number fields.

Following Cohen (see [Cohen93] Section 4.5), we can define the main problem as follows:

- **Subfield Problem:**

  Given two number fields $\mathbb{Q}(\alpha)$, $\mathbb{Q}(\beta)$ via the minimal polynomials for their generators α and β, decide whether one field is isomorphic to a subfield of the other.

From a solution to this problem flow solutions to the following problems as well:

- **Primitive Element Problem:**

  Given several algebraic numbers $\alpha_1, \ldots, \alpha_m$, compute a single algebraic number θ such that $\mathbb{Q}(\alpha_1, \ldots, \alpha_m) = \mathbb{Q}(\theta)$.

- **Field Isomorphism Problem:**

  Decide whether two number fields $\mathbb{Q}(\alpha)$, $\mathbb{Q}(\beta)$ are isomorphic.

- **Field Membership Problem:**

  Given two algebraic numbers α, β, decide whether $\alpha \in \mathbb{Q}(\beta)$, and if so write $\alpha = f(\beta)$ for some $f(x) \in \mathbb{Q}[x]$.

sympy.polys.numberfields.subfield.**field_isomorphism**(*a*, *b*, *, *fast=True*)

Find an embedding of one number field into another.

Parameters

a : *Expr* (page 947)

Any expression representing an algebraic number.

b : *Expr* (page 947)

Any expression representing an algebraic number.

fast : boolean, optional (default=True)

If `True`, we first attempt a potentially faster way of computing the isomorphism, falling back on a slower method if this fails. If `False`, we go directly to the slower method, which is guaranteed to return a result.

Returns

List of rational numbers, or None

If $\mathbb{Q}(a)$ is not isomorphic to some subfield of $\mathbb{Q}(b)$, then return `None`. Otherwise, return a list of rational numbers representing an element of $\mathbb{Q}(b)$ to which a may be mapped, in order to define a monomorphism, i.e. an isomorphism from $\mathbb{Q}(a)$ to some subfield of $\mathbb{Q}(b)$. The elements of the list are the coefficients of falling powers of b.

Explanation

This function looks for an isomorphism from $\mathbb{Q}(a)$ onto some subfield of $\mathbb{Q}(b)$. Thus, it solves the Subfield Problem.

Examples

```
>>> from sympy import sqrt, field_isomorphism, I
>>> print(field_isomorphism(3, sqrt(2)))
[3]
>>> print(field_isomorphism( I*sqrt(3), I*sqrt(3)/2))
[2, 0]
```

sympy.polys.numberfields.subfield.**primitive_element**(*extension*, *x=None*, *, *ex=False*, *polys=False*)

Find a single generator for a number field given by several generators.

Parameters

extension : list of *Expr* (page 947)

Each expression must represent an algebraic number α_i.

x : *Symbol* (page 976), optional (default=None)

The desired symbol to appear in the computed minimal polynomial for the primitive element θ. If `None`, we use a dummy symbol.

ex : boolean, optional (default=False)

If and only if `True`, compute the representation of each α_i as a $\mathbb{Q}$-linear combination over the powers of θ.

polys : boolean, optional (default=False)

If `True`, return the minimal polynomial as a *Poly* (page 2378). Otherwise return it as an *Expr* (page 947).

Returns

Pair (f, coeffs) or triple (f, coeffs, reps), where:

`f` is the minimal polynomial for the primitive element. `coeffs` gives the primitive element as a linear combination of the given generators. `reps` is present if and only if argument `ex=True` was passed, and is a list of lists of rational numbers. Each list gives the coefficients of falling powers of the primitive element, to recover one of the original, given generators.

Explanation

The basic problem is this: Given several algebraic numbers $\alpha_1, \alpha_2, \ldots, \alpha_n$, find a single algebraic number θ such that $\mathbb{Q}(\alpha_1, \alpha_2, \ldots, \alpha_n) = \mathbb{Q}(\theta)$.

This function actually guarantees that θ will be a linear combination of the α_i, with non-negative integer coefficients.

Furthermore, if desired, this function will tell you how to express each α_i as a $\mathbb{Q}$-linear combination of the powers of θ.

Examples

```
>>> from sympy import primitive_element, sqrt, S, minpoly, simplify
>>> from sympy.abc import x
>>> f, lincomb, reps = primitive_element([sqrt(2), sqrt(3)], x, ex=True)
```

Then `lincomb` tells us the primitive element as a linear combination of the given generators `sqrt(2)` and `sqrt(3)`.

```
>>> print(lincomb)
[1, 1]
```

This means the primtiive element is $\sqrt{2} + \sqrt{3}$. Meanwhile `f` is the minimal polynomial for this primitive element.

```
>>> print(f)
x**4 - 10*x**2 + 1
>>> print(minpoly(sqrt(2) + sqrt(3), x))
x**4 - 10*x**2 + 1
```

Finally, `reps` (which was returned only because we set keyword arg `ex=True`) tells us how to recover each of the generators $\sqrt{2}$ and $\sqrt{3}$ as $\mathbb{Q}$-linear combinations of the powers of the primitive element $\sqrt{2} + \sqrt{3}$.

```
>>> print([S(r) for r in reps[0]])
[1/2, 0, -9/2, 0]
>>> theta = sqrt(2) + sqrt(3)
>>> print(simplify(theta**3/2 - 9*theta/2))
sqrt(2)
>>> print([S(r) for r in reps[1]])
[-1/2, 0, 11/2, 0]
>>> print(simplify(-theta**3/2 + 11*theta/2))
sqrt(3)
```

sympy.polys.numberfields.subfield.**to_number_field**(*extension*, *theta=None*, *, *gen=None*, *alias=None*)

Express one algebraic number in the field generated by another.

Parameters

extension : *Expr* (page 947) or list of *Expr* (page 947)

Either the algebraic number that is to be expressed in the other field, or else a list of algebraic numbers, a primitive element for which is to be expressed in the other field.

theta : *Expr* (page 947), None, optional (default=None)

If an *Expr* (page 947) representing an algebraic number, behavior is as described under **Explanation**. If `None`, then this function reduces to a shorthand for calling *primitive_element()* (page 2712) on `extension` and turning the computed primitive element into an *AlgebraicNumber* (page 988).

gen : *Symbol* (page 976), None, optional (default=None)

If provided, this will be used as the generator symbol for the minimal polynomial in the returned *AlgebraicNumber* (page 988).

alias : str, *Symbol* (page 976), None, optional (default=None)

If provided, this will be used as the alias symbol for the returned *AlgebraicNumber* (page 988).

Returns

AlgebraicNumber

Belonging to $\mathbb{Q}(\theta)$ and equaling η.

Raises

IsomorphismFailed

If $\eta \notin \mathbb{Q}(\theta)$.

Explanation

Given two algebraic numbers η, θ, this function either expresses η as an element of $\mathbb{Q}(\theta)$, or else raises an exception if $\eta \notin \mathbb{Q}(\theta)$.

This function is essentially just a convenience, utilizing *field_isomorphism()* (page 2711) (our solution of the Subfield Problem) to solve this, the Field Membership Problem.

As an additional convenience, this function allows you to pass a list of algebraic numbers $\alpha_1, \alpha_2, \ldots, \alpha_n$ instead of η. It then computes η for you, as a solution of the Primitive Element Problem, using *primitive_element()* (page 2712) on the list of α_i.

Examples

```
>>> from sympy import sqrt, to_number_field
>>> eta = sqrt(2)
>>> theta = sqrt(2) + sqrt(3)
>>> a = to_number_field(eta, theta)
>>> print(type(a))
<class 'sympy.core.numbers.AlgebraicNumber'>
>>> a.root
sqrt(2) + sqrt(3)
>>> print(a)
sqrt(2)
>>> a.coeffs()
[1/2, 0, -9/2, 0]
```

We get an *AlgebraicNumber* (page 988), whose `.root` is θ, whose value is η, and whose `.coeffs()` show how to write η as a $\mathbb{Q}$-linear combination in falling powers of θ.

See also:

field_isomorphism (page 2711), *primitive_element* (page 2712)

Internals

Algebraic number fields

Algebraic number fields are represented in SymPy by the *AlgebraicField* (page 2539) class, which is a part of *the polynomial domains system* (page 2504).

Representing algebraic numbers

There are several different ways to represent algebraic numbers, and different forms may be preferable for different computational tasks. See [Cohen93], Sec. 4.2.

As number field elements

In SymPy, there is a distinction between number and expression classes defined in the *sympy.core.numbers* (page 981) module on the one hand, and domains and domain elements defined in the *polys* (page 2360) module on the other. This is explained in more detail *here* (page 2477).

When it comes to algebraic numbers, the *sympy.core.numbers* (page 981) module offers the *AlgebraicNumber* (page 988) class, while the *polys* (page 2360) module offers the *ANP* (page 2568) class. This is the type of domain elements belonging to the *AlgebraicField* (page 2539) domain.

As elements of finitely-generated modules

In computational algebraic number theory, finitely-generated $\mathbb{Z}$-modules are of central importance. For example, every order and every ideal is such a module.

In particular, the maximal order – or ring of integers – in a number field is a finitely-generated $\mathbb{Z}$-module, whose generators form an integral basis for the field.

Classes allowing us to represent such modules, and their elements, are provided in the *modules* (page 2716) module. Here, the *ModuleElement* (page 2729) class provides another way to represent algebraic numbers.

Finitely-generated modules

Modules in number fields.

The classes defined here allow us to work with finitely generated, free modules, whose generators are algebraic numbers.

There is an abstract base class called *Module* (page 2719), which has two concrete subclasses, *PowerBasis* (page 2725) and *Submodule* (page 2726).

Every module is defined by its basis, or set of generators:

- For a *PowerBasis* (page 2725), the generators are the first n powers (starting with the zeroth) of an algebraic integer θ of degree n. The *PowerBasis* (page 2725) is constructed by passing either the minimal polynomial of θ, or an *AlgebraicField* (page 2539) having θ as its primitive element.
- For a *Submodule* (page 2726), the generators are a set of $\mathbb{Q}$-linear combinations of the generators of another module. That other module is then the "parent" of the *Submodule* (page 2726). The coefficients of the $\mathbb{Q}$-linear combinations may be given by an integer matrix, and a positive integer denominator. Each column of the matrix defines a generator.

```
>>> from sympy.polys import Poly, cyclotomic_poly, ZZ
>>> from sympy.abc import x
>>> from sympy.polys.matrices import DomainMatrix, DM
>>> from sympy.polys.numberfields.modules import PowerBasis
>>> T = Poly(cyclotomic_poly(5, x))
>>> A = PowerBasis(T)
>>> print(A)
```

(continues on next page)