Sage Reference Manual: Matrices and Spaces of Matrices

Release 7.6

The Sage Development Team

Mar 25, 2017

CONTENTS

Sage provides native support for working with matrices over any commutative or noncommutative ring. The parent object for a matrix is a matrix space `MatrixSpace(R,n,m)` of all $n \times m$ matrices over a ring R.

To create a matrix, either use the `matrix(...)` function or create a matrix space using the `MatrixSpace` command and coerce an object into it.

Matrices also act on row vectors, which you create using the `vector(...)` command or by making a `VectorSpace` and coercing lists into it. The natural action of matrices on row vectors is from the right. Sage currently does not have a column vector class (on which matrices would act from the left), but this is planned.

In addition to native Sage matrices, Sage also includes the following additional ways to compute with matrices:

- Several math software systems included with Sage have their own native matrix support, which can be used from Sage. E.g., PARI, GAP, Maxima, and Singular all have a notion of matrices.
- The GSL C-library is included with Sage, and can be used via Cython.
- The `scipy` module provides support for *sparse* numerical linear algebra, among many other things.
- The `numpy` module, which you load by typing `import numpy` is included standard with Sage. It contains a very sophisticated and well developed array class, plus optimized support for *numerical linear algebra*. Sage's matrices over RDF and CDF (native floating-point real and complex numbers) use numpy.

Finally, this module contains some data-structures for matrix-like objects like operation tables (e.g. the multiplication table of a group).

CHAPTER

ONE

MATRIX SPACES

You can create any space $\mathrm{Mat}_{n\times m}(R)$ of either dense or sparse matrices with given number of rows and columns over any commutative or noncommutative ring.

EXAMPLES:

```
sage: MS = MatrixSpace(QQ,6,6,sparse=True); MS
Full MatrixSpace of 6 by 6 sparse matrices over Rational Field
sage: MS.base_ring()
Rational Field
sage: MS = MatrixSpace(ZZ,3,5,sparse=False); MS
Full MatrixSpace of 3 by 5 dense matrices over Integer Ring
```

class `sage.matrix.matrix_space.`**`MatrixSpace`**(*base_ring*, *nrows*, *ncols=None*, *sparse=False*, *implementation='flint'*)

Bases: `sage.structure.unique_representation.UniqueRepresentation`, `sage.structure.parent_gens.ParentWithGens`

The space of all nrows x ncols matrices over base_ring.

INPUT:

- `base_ring` - a ring
- `nrows` - int, the number of rows
- `ncols` - (default nrows) int, the number of columns
- `sparse` - (default false) whether or not matrices are given a sparse representation

EXAMPLES:

```
sage: MatrixSpace(ZZ,10,5)
Full MatrixSpace of 10 by 5 dense matrices over Integer Ring
sage: MatrixSpace(ZZ,10,5).category()
Category of infinite enumerated modules over
 (euclidean domains and infinite enumerated sets and metric spaces)
sage: MatrixSpace(ZZ,10,10).category()
Category of infinite enumerated algebras over
 (euclidean domains and infinite enumerated sets and metric spaces)
sage: MatrixSpace(QQ,10).category()
Category of infinite algebras over (quotient fields and metric spaces)
```

`base_extend`(*R*)

Return base extension of this matrix space to R.

INPUT:

- `R` - ring

OUTPUT: a matrix space

EXAMPLES:

```
sage: Mat(ZZ,3,5).base_extend(QQ)
Full MatrixSpace of 3 by 5 dense matrices over Rational Field
sage: Mat(QQ,3,5).base_extend(GF(7))
Traceback (most recent call last):
...
TypeError: no base extension defined
```

basis ( )

Returns a basis for this matrix space.

> **Warning:** This will of course compute every generator of this matrix space. So for large matrices, this could take a long time, waste a massive amount of memory (for dense matrices), and is likely not very useful. Don't use this on large matrix spaces.

EXAMPLES:

```
sage: Mat(ZZ,2,2).basis()
[
[1 0]  [0 1]  [0 0]  [0 0]
[0 0], [0 0], [1 0], [0 1]
]
```

cached_method (*f*, *name=None*, *key=None*, *do_pickle=None*)

A decorator for cached methods.

EXAMPLES:

In the following examples, one can see how a cached method works in application. Below, we demonstrate what is done behind the scenes:

```
sage: class C:
....:     @cached_method
....:     def __hash__(self):
....:         print("compute hash")
....:         return int(5)
....:     @cached_method
....:     def f(self, x):
....:         print("computing cached method")
....:         return x*2
sage: c = C()
sage: type(C.__hash__)
<type 'sage.misc.cachefunc.CachedMethodCallerNoArgs'>
sage: hash(c)
compute hash
5
```

When calling a cached method for the second time with the same arguments, the value is gotten from the cache, so that a new computation is not needed:

```
sage: hash(c)
5
sage: c.f(4)
computing cached method
```

```
8
sage: c.f(4) is c.f(4)
True
```

Different instances have distinct caches:

```
sage: d = C()
sage: d.f(4) is c.f(4)
computing cached method
False
sage: d.f.clear_cache()
sage: c.f(4)
8
sage: d.f(4)
computing cached method
8
```

Using cached methods for the hash and other special methods was implemented in trac ticket #12601, by means of `CachedSpecialMethod`. We show that it is used behind the scenes:

```
sage: cached_method(c.__hash__)
<sage.misc.cachefunc.CachedSpecialMethod object at ...>
sage: cached_method(c.f)
<sage.misc.cachefunc.CachedMethod object at ...>
```

The parameter `do_pickle` can be used if the contents of the cache should be stored in a pickle of the cached method. This can be dangerous with special methods such as `__hash__`:

```
sage: class C:
....:     @cached_method(do_pickle=True)
....:     def __hash__(self):
....:         return id(self)

sage: import __main__
sage: __main__.C = C
sage: c = C()
sage: hash(c) # random output
sage: d = loads(dumps(c))
sage: hash(d) == hash(c)
True
```

However, the contents of a method's cache are not pickled unless `do_pickle` is set:

```
sage: class C:
....:     @cached_method
....:     def __hash__(self):
....:         return id(self)

sage: __main__.C = C
sage: c = C()
sage: hash(c) # random output
sage: d = loads(dumps(c))
sage: hash(d) == hash(c)
False
```

cardinality ( )

Return the number of elements in self.

EXAMPLES:

```
sage: MatrixSpace(GF(3), 2, 3).cardinality()
729
sage: MatrixSpace(ZZ, 2).cardinality()
+Infinity
sage: MatrixSpace(ZZ, 0, 3).cardinality()
1
```

change_ring (*R*)

Return matrix space over R with otherwise same parameters as self.

INPUT:

- `R` - ring

OUTPUT: a matrix space

EXAMPLES:

```
sage: Mat(QQ,3,5).change_ring(GF(7))
Full MatrixSpace of 3 by 5 dense matrices over Finite Field of size 7
```

column_space ()

Return the module spanned by all columns of matrices in this matrix space. This is a free module of rank the number of columns. It will be sparse or dense as this matrix space is sparse or dense.

EXAMPLES:

```
sage: M = Mat(GF(9,'a'),20,5,sparse=True); M.column_space()
Sparse vector space of dimension 20 over Finite Field in a of size 3^2
```

construction ()

EXAMPLES:

```
sage: A = matrix(ZZ, 2, [1..4], sparse=True)
sage: A.parent().construction()
(MatrixFunctor, Integer Ring)
sage: A.parent().construction()[0](QQ['x'])
Full MatrixSpace of 2 by 2 sparse matrices over Univariate Polynomial Ring in␣
↪x over Rational Field
sage: parent(A/2)
Full MatrixSpace of 2 by 2 sparse matrices over Rational Field
```

dimension ()

Returns (m rows) * (n cols) of self as Integer

EXAMPLES:

```
sage: MS = MatrixSpace(ZZ,4,6)
sage: u = MS.dimension()
sage: u - 24 == 0
True
```

dims ()

Returns (m row, n col) representation of self dimension

EXAMPLES:

```
sage: MS = MatrixSpace(ZZ,4,6)
sage: MS.dims()
(4, 6)
```

full_category_initialisation ( )

Make full use of the category framework.

Note: It turns out that it causes a massive speed regression in computations with elliptic curves, if a full initialisation of the category framework of matrix spaces happens at initialisation: The elliptic curves code treats matrix spaces as containers, not as objects of a category. Therefore, making full use of the category framework is now provided by a separate method (see trac ticket #11900).

EXAMPLES:

```
sage: MS = MatrixSpace(QQ,8)
sage: TestSuite(MS).run()
sage: type(MS)
<class 'sage.matrix.matrix_space.MatrixSpace_with_category'>
sage: MS.full_category_initialisation()
doctest:...: DeprecationWarning: the full_category_initialization
 method does nothing, as a matrix space now has its category
 systematically fully initialized
See http://trac.sagemath.org/15801 for details.
```

gen (*n*)

Return the n-th generator of this matrix space.

This doesn't compute all basis matrices, so it is reasonably intelligent.

EXAMPLES:

```
sage: M = Mat(GF(7),10000,5); M.ngens()
50000
sage: a = M.10
sage: a[:4]
[0 0 0 0 0]
[0 0 0 0 0]
[1 0 0 0 0]
[0 0 0 0 0]
```

get_action_impl (*S*, *op*, *self_on_left*)

identity_matrix ( )

Returns the identity matrix in `self`.

`self` must be a space of square matrices. The returned matrix is immutable. Please use `copy` if you want a modified copy.

EXAMPLES:

```
sage: MS1 = MatrixSpace(ZZ,4)
sage: MS2 = MatrixSpace(QQ,3,4)
sage: I = MS1.identity_matrix()
sage: I
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
```

```
sage: Er = MS2.identity_matrix()
Traceback (most recent call last):
...
TypeError: identity matrix must be square
```

is_dense ()

Returns True if matrices in self are dense and False otherwise.

EXAMPLES:

```
sage: Mat(RDF,2,3).is_sparse()
False
sage: Mat(RR,123456,22,sparse=True).is_sparse()
True
```

is_finite ()

EXAMPLES:

```
sage: MatrixSpace(GF(101), 10000).is_finite()
True
sage: MatrixSpace(QQ, 2).is_finite()
False
```

is_sparse ()

Returns True if matrices in self are sparse and False otherwise.

EXAMPLES:

```
sage: Mat(GF(2011),10000).is_sparse()
False
sage: Mat(GF(2011),10000,sparse=True).is_sparse()
True
```

matrix (*x=0*, *coerce=True*, *copy=True*)

Create a matrix in `self`.

INPUT:

- `x` – (default: 0) data to construct a new matrix from. Can be one of the following:
  - 0, corresponding to the zero matrix;
  - 1, corresponding to the identity_matrix;
  - a matrix, whose dimensions must match `self` and whose base ring must be convertible to the base ring of `self`;
  - a list of entries corresponding to all elements of the new matrix;
  - a list of rows with each row given as an iterable;
- `coerce` – (default: `True`) whether to coerce `x` into self;
- `copy` – (default: `True`) whether to copy `x` during construction (makes a difference only if `x` is a matrix in `self`).

OUTPUT:

- a matrix in `self`.

EXAMPLES:

```
sage: M = MatrixSpace(ZZ, 2)
sage: M.matrix([[1,0],[0,-1]])
[ 1  0]
[ 0 -1]
sage: M.matrix([1,0,0,-1])
[ 1  0]
[ 0 -1]
sage: M.matrix([1,2,3,4])
[1 2]
[3 4]
```

Note that the last "flip" cannot be performed if `x` is a matrix, no matter what is `rows` (it used to be possible but was fixed by trac ticket #10793):

```
sage: projection = matrix(ZZ,[[1,0,0],[0,1,0]])
sage: projection
[1 0 0]
[0 1 0]
sage: projection.parent()
Full MatrixSpace of 2 by 3 dense matrices over Integer Ring
sage: M = MatrixSpace(ZZ, 3 , 2)
sage: M
Full MatrixSpace of 3 by 2 dense matrices over Integer Ring
sage: M(projection)
Traceback (most recent call last):
...
ValueError: a matrix from
Full MatrixSpace of 2 by 3 dense matrices over Integer Ring
cannot be converted to a matrix in
Full MatrixSpace of 3 by 2 dense matrices over Integer Ring!
```

If you really want to make from a matrix another matrix of different dimensions, use either transpose method or explicit conversion to a list:

```
sage: M(projection.list())
[1 0]
[0 0]
[1 0]
```

matrix_space (*nrows=None*, *ncols=None*, *sparse=False*)

Return the matrix space with given number of rows, columns and sparcity over the same base ring as self, and defaults the same as self.

EXAMPLES:

```
sage: M = Mat(GF(7),100,200)
sage: M.matrix_space(5000)
Full MatrixSpace of 5000 by 200 dense matrices over Finite Field of size 7
sage: M.matrix_space(ncols=5000)
Full MatrixSpace of 100 by 5000 dense matrices over Finite Field of size 7
sage: M.matrix_space(sparse=True)
Full MatrixSpace of 100 by 200 sparse matrices over Finite Field of size 7
```

ncols ( )

Return the number of columns of matrices in this space.

EXAMPLES:

```
sage: M = Mat(ZZ['x'],200000,500000,sparse=True)
sage: M.ncols()
500000
```

ngens ( )

Return the number of generators of this matrix space, which is the number of entries in the matrices in this space.

EXAMPLES:

```
sage: M = Mat(GF(7),100,200); M.ngens()
20000
```

nrows ( )

Return the number of rows of matrices in this space.

EXAMPLES:

```
sage: M = Mat(ZZ,200000,500000)
sage: M.nrows()
200000
```

one ( )

Returns the identity matrix in `self` .

`self` must be a space of square matrices. The returned matrix is immutable. Please use `copy` if you want a modified copy.

EXAMPLES:

```
sage: MS1 = MatrixSpace(ZZ,4)
sage: MS2 = MatrixSpace(QQ,3,4)
sage: I = MS1.identity_matrix()
sage: I
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
sage: Er = MS2.identity_matrix()
Traceback (most recent call last):
...
TypeError: identity matrix must be square
```

random_element (*density=None*, **args*, ***kwds*)

Returns a random element from this matrix space.

INPUT:

- `density` - `float` or `None` (default: `None`); rough measure of the proportion of nonzero entries in the random matrix; if set to `None` , all entries of the matrix are randomized, allowing for any element of the underlying ring, but if set to a `float` , a proportion of entries is selected and randomized to non-zero elements of the ring
- `*args, **kwds` - remaining parameters, which may be passed to the random_element function of the base ring. ("may be", since this function calls the `randomize` function on the zero matrix, which need not call the `random_element` function of the base ring at all in general.)

OUTPUT:

- Matrix

Note: This method will randomize a proportion of roughly `density` entries in a newly allocated zero matrix.

By default, if the user sets the value of `density` explicitly, this method will enforce that these entries are set to non-zero values. However, if the test for equality with zero in the base ring is too expensive, the user can override this behaviour by passing the argument `nonzero=False` to this method.

Otherwise, if the user does not set the value of `density`, the default value is taken to be 1, and the option `nonzero=False` is passed to the `randomize` method.

EXAMPLES:

```
sage: Mat(ZZ,2,5).random_element()
[ -8   2   0   0   1]
[ -1   2   1 -95  -1]
sage: Mat(QQ,2,5).random_element(density=0.5)
[ 2  0  0  0  1]
[ 0  0  0 -1  0]
sage: Mat(QQ,3,sparse=True).random_element()
[  -1   -1   -1]
[  -3 -1/3   -1]
[   0   -1    1]
sage: Mat(GF(9,'a'),3,sparse=True).random_element()
[    a   2*a     1]
[    2     1 a + 2]
[  2*a     2     2]
```

row_space ( )

Return the module spanned by all rows of matrices in this matrix space. This is a free module of rank the number of rows. It will be sparse or dense as this matrix space is sparse or dense.

EXAMPLES:

```
sage: M = Mat(ZZ,20,5,sparse=False); M.row_space()
Ambient free module of rank 5 over the principal ideal domain Integer Ring
```

some_elements ( )

Return some elements of this matrix space.

See `TestSuite` for a typical use case.

OUTPUT:

An iterator.

EXAMPLES:

```
sage: M = MatrixSpace(ZZ, 2, 2)
sage: tuple(M.some_elements())
(
[1 0]  [1 1]  [ 0  1]  [-2  3]  [-4  5]  [-6  7]  [-8  9]  [-10  11]
[0 0], [1 1], [-1  2], [-3  4], [-5  6], [-7  8], [-9 10], [-11  12],

[-12  13]  [-14  15]  [-16  17]  [-18  19]  [-20  21]  [-22  23]
[-13  14], [-15  16], [-17  18], [-19  20], [-21  22], [-23  24],

[-24  25]  [-26  27]  [-28  29]  [-30  31]  [-32  33]  [-34  35]
[-25  26], [-27  28], [-29  30], [-31  32], [-33  34], [-35  36],
```

```
[-36  37]  [-38  39]  [-40  41]  [-42  43]  [-44  45]  [-46  47]
[-37  38], [-39  40], [-41  42], [-43  44], [-45  46], [-47  48],

[-48  49]
[-49  50]
)

sage: M = MatrixSpace(QQ, 2, 3)
sage: tuple(M.some_elements())
(
[1 0 0]  [1/2 1/2 1/2]  [ 1/2 -1/2    2]  [  -1   42  2/3]
[0 0 0], [1/2 1/2 1/2], [  -2    0    1], [-2/3  3/2 -3/2],

[ 4/5 -4/5  5/4]  [ 7/6 -7/6  8/9]  [ 10/11 -10/11  11/10]
[-5/4  6/7 -6/7], [-8/9  9/8 -9/8], [-11/10  12/13 -12/13],

[ 13/12 -13/12  14/15]  [ 16/17 -16/17  17/16]
[-14/15  15/14 -15/14], [-17/16  18/19 -18/19],

[  19/18  -19/18  20/441]  [ 22/529 -22/529  529/22]
[-20/441  441/20 -441/20], [-529/22  24/625 -24/625],

[ 625/24 -625/24  26/729]  [ 28/841 -28/841  841/28]
[-26/729  729/26 -729/26], [-841/28  30/961 -30/961],

[  961/30  -961/30  32/1089]  [ 34/1225 -34/1225  1225/34]
[-32/1089  1089/32 -1089/32], [-1225/34  36/1369 -36/1369],

[ 1369/36 -1369/36  38/1521]  [ 40/68921 -40/68921  68921/40]
[-38/1521  1521/38 -1521/38], [-68921/40  42/79507 -42/79507],

[ 79507/42 -79507/42  44/91125]
[-44/91125  91125/44 -91125/44]
)

sage: M = MatrixSpace(SR, 2, 2)
sage: tuple(M.some_elements())
(
[1 0]  [some_variable some_variable]
[0 0], [some_variable some_variable]
)
```

zero ( )

Returns the zero matrix in `self`.

`self` must be a space of square matrices. The returned matrix is immutable. Please use `copy` if you want a modified copy.

EXAMPLES:

```
sage: z = MatrixSpace(GF(7),2,4).zero_matrix(); z
[0 0 0 0]
[0 0 0 0]
sage: z.is_mutable()
False
```

zero_matrix ( )

Returns the zero matrix in `self`.

`self` must be a space of square matrices. The returned matrix is immutable. Please use `copy` if you want a modified copy.

EXAMPLES:

```
sage: z = MatrixSpace(GF(7),2,4).zero_matrix(); z
[0 0 0 0]
[0 0 0 0]
sage: z.is_mutable()
False
```

`sage.matrix.matrix_space.`**`dict_to_list`**`(`*entries*, *nrows*, *ncols*`)`

Given a dictionary of coordinate tuples, return the list given by reading off the nrows*ncols matrix in row order.

EXAMPLES:

```
sage: from sage.matrix.matrix_space import dict_to_list
sage: d = {}
sage: d[(0,0)] = 1
sage: d[(1,1)] = 2
sage: dict_to_list(d, 2, 2)
[1, 0, 0, 2]
sage: dict_to_list(d, 2, 3)
[1, 0, 0, 0, 2, 0]
```

`sage.matrix.matrix_space.`**`is_MatrixSpace`**`(`*x*`)`

Returns True if self is an instance of MatrixSpace returns false if self is not an instance of MatrixSpace

EXAMPLES:

```
sage: from sage.matrix.matrix_space import is_MatrixSpace
sage: MS = MatrixSpace(QQ,2)
sage: A = MS.random_element()
sage: is_MatrixSpace(MS)
True
sage: is_MatrixSpace(A)
False
sage: is_MatrixSpace(5)
False
```

`sage.matrix.matrix_space.`**`list_to_dict`**`(`*entries*, *nrows*, *ncols*, *rows=True*`)`

Given a list of entries, create a dictionary whose keys are coordinate tuples and values are the entries.

EXAMPLES:

```
sage: from sage.matrix.matrix_space import list_to_dict
sage: d = list_to_dict([1,2,3,4],2,2)
sage: d[(0,1)]
2
sage: d = list_to_dict([1,2,3,4],2,2,rows=False)
sage: d[(0,1)]
3
```

`sage.matrix.matrix_space.`**`test_trivial_matrices_inverse`**`(`*ring*, *sparse=True*, *check-rank=True*`)`

Tests inversion, determinant and is_invertible for trivial matrices.

This function is a helper to check that the inversion of trivial matrices (of size 0x0, nx0, 0xn or 1x1) is handled consistently by the various implementation of matrices. The coherency is checked through a bunch of assertions. If an inconsistency is found, an AssertionError is raised which should make clear what is the problem.

INPUT:

- `ring` - a ring
- `sparse` - a boolean
- `checkrank` - a boolean

OUTPUT:

- nothing if everything is correct, otherwise raise an AssertionError

The methods determinant, is_invertible, rank and inverse are checked for

- the 0x0 empty identity matrix
- the 0x3 and 3x0 matrices
- the 1x1 null matrix [0]
- the 1x1 identity matrix [1]

If `checkrank` is `False` then the rank is not checked. This is used the check matrix over ring where echelon form is not implemented.

Todo

This must be adapted to category check framework when ready (see trac ticket #5274).

CHAPTER

TWO

GENERAL MATRIX CONSTRUCTOR

class `sage.matrix.constructor.` **`MatrixFactory`**

Bases: `object`

Create a matrix.

This implements the `matrix` constructor:

```
sage: matrix([[1,2],[3,4]])
[1 2]
[3 4]
```

It also contains methods to create special types of matrices, see `matrix.[tab]` for more options. For example:

```
sage: matrix.identity(2)
[1 0]
[0 1]
```

INPUT:

The matrix command takes the entries of a matrix, optionally preceded by a ring and the dimensions of the matrix, and returns a matrix.

The entries of a matrix can be specified as a flat list of elements, a list of lists (i.e., a list of rows), a list of Sage vectors, a callable object, or a dictionary having positions as keys and matrix entries as values (see the examples). If you pass in a callable object, then you must specify the number of rows and columns. You can create a matrix of zeros by passing an empty list or the integer zero for the entries. To construct a multiple of the identity (cI), you can specify square dimensions and pass in c. Calling matrix() with a Sage object may return something that makes sense. Calling matrix() with a NumPy array will convert the array to a matrix.

The ring, number of rows, and number of columns of the matrix can be specified by setting the `ring`, `nrows`, or `ncols` keyword parameters or by passing them as the first arguments to the function in specified order. The ring defaults to `ZZ` if it is not specified and cannot be determined from the entries. If the number of rows and columns are not specified and cannot be determined, then an empty 0x0 matrix is returned.

INPUT:

- `ring` – the base ring for the entries of the matrix.
- `nrows` – the number of rows in the matrix.
- `ncols` – the number of columns in the matrix.
- `sparse` – create a sparse matrix. This defaults to `True` when the entries are given as a dictionary, otherwise defaults to `False`.
- `entries` – see examples below.

OUTPUT:

a matrix

EXAMPLES:

```
sage: m = matrix(2); m; m.parent()
[0 0]
[0 0]
Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
```

```
sage: m = matrix(2,3); m; m.parent()
[0 0 0]
[0 0 0]
Full MatrixSpace of 2 by 3 dense matrices over Integer Ring
```

```
sage: m = matrix(QQ,[[1,2,3],[4,5,6]]); m; m.parent()
[1 2 3]
[4 5 6]
Full MatrixSpace of 2 by 3 dense matrices over Rational Field
```

```
sage: m = matrix(QQ, 3, 3, lambda i, j: i+j); m
[0 1 2]
[1 2 3]
[2 3 4]
sage: m = matrix(3, lambda i,j: i-j); m
[ 0 -1 -2]
[ 1  0 -1]
[ 2  1  0]
```

```
sage: matrix(QQ, 2, 3, lambda x, y: x+y)
[0 1 2]
[1 2 3]
sage: matrix(QQ, 5, 5, lambda x, y: (x+1) / (y+1))
[  1 1/2 1/3 1/4 1/5]
[  2   1 2/3 1/2 2/5]
[  3 3/2   1 3/4 3/5]
[  4   2 4/3   1 4/5]
[  5 5/2 5/3 5/4   1]
```

```
sage: v1=vector((1,2,3))
sage: v2=vector((4,5,6))
sage: m = matrix([v1,v2]); m; m.parent()
[1 2 3]
[4 5 6]
Full MatrixSpace of 2 by 3 dense matrices over Integer Ring
```

```
sage: m = matrix(QQ,2,[1,2,3,4,5,6]); m; m.parent()
[1 2 3]
[4 5 6]
Full MatrixSpace of 2 by 3 dense matrices over Rational Field
```

```
sage: m = matrix(QQ,2,3,[1,2,3,4,5,6]); m; m.parent()
[1 2 3]
[4 5 6]
Full MatrixSpace of 2 by 3 dense matrices over Rational Field
```

```
sage: m = matrix({(0,1): 2, (1,1):2/5}); m; m.parent()
[  0   2]
[  0 2/5]
Full MatrixSpace of 2 by 2 sparse matrices over Rational Field
```

```
sage: m = matrix(QQ,2,3,{(1,1): 2}); m; m.parent()
[0 0 0]
[0 2 0]
Full MatrixSpace of 2 by 3 sparse matrices over Rational Field
```

```
sage: import numpy
sage: n = numpy.array([[1,2],[3,4]],float)
sage: m = matrix(n); m; m.parent()
[1.0 2.0]
[3.0 4.0]
Full MatrixSpace of 2 by 2 dense matrices over Real Double Field
```

```
sage: v = vector(ZZ, [1, 10, 100])
sage: m = matrix(v); m; m.parent()
[  1  10 100]
Full MatrixSpace of 1 by 3 dense matrices over Integer Ring
sage: m = matrix(GF(7), v); m; m.parent()
[1 3 2]
Full MatrixSpace of 1 by 3 dense matrices over Finite Field of size 7
```

```
sage: g = graphs.PetersenGraph()
sage: m = matrix(g); m; m.parent()
[0 1 0 0 1 1 0 0 0 0]
[1 0 1 0 0 0 1 0 0 0]
[0 1 0 1 0 0 0 1 0 0]
[0 0 1 0 1 0 0 0 1 0]
[1 0 0 1 0 0 0 0 0 1]
[1 0 0 0 0 0 0 1 1 0]
[0 1 0 0 0 0 0 0 1 1]
[0 0 1 0 0 1 0 0 0 1]
[0 0 0 1 0 1 1 0 0 0]
[0 0 0 0 1 0 1 1 0 0]
Full MatrixSpace of 10 by 10 dense matrices over Integer Ring
```

```
sage: matrix(ZZ, 10, 10, range(100), sparse=True).parent()
Full MatrixSpace of 10 by 10 sparse matrices over Integer Ring
```

```
sage: R = PolynomialRing(QQ, 9, 'x')
sage: A = matrix(R, 3, 3, R.gens()); A
[x0 x1 x2]
[x3 x4 x5]
[x6 x7 x8]
sage: det(A)
-x2*x4*x6 + x1*x5*x6 + x2*x3*x7 - x0*x5*x7 - x1*x3*x8 + x0*x4*x8
```

AUTHORS:

- William Stein: Initial implementation
- Jason Grout (2008-03): almost a complete rewrite, with bits and pieces from the original implementation
- Jeroen Demeyer (2016-02-05): major clean up, see trac ticket #20015 and trac ticket #20016

sage.matrix.constructor.**ncols_from_dict**(*d*)

Given a dictionary that defines a sparse matrix, return the number of columns that matrix should have.

This is for internal use by the matrix function.

INPUT:

- `d` - dict

OUTPUT:

integer

EXAMPLES:

```
sage: sage.matrix.constructor.ncols_from_dict({})
0
```

Here the answer is 301 not 300, since there is a 0-th row.

```
sage: sage.matrix.constructor.ncols_from_dict({(4,300):10})
301
```

sage.matrix.constructor.**nrows_from_dict**(*d*)

Given a dictionary that defines a sparse matrix, return the number of rows that matrix should have.

This is for internal use by the matrix function.

INPUT:

- `d` - dict

OUTPUT:

integer

EXAMPLES:

```
sage: sage.matrix.constructor.nrows_from_dict({})
0
```

Here the answer is 301 not 300, since there is a 0-th row.

```
sage: sage.matrix.constructor.nrows_from_dict({(300,4):10})
301
```

sage.matrix.constructor.**prepare**(*w*)

Given a list w of numbers, find a common ring that they all canonically map to, and return the list of images of the elements of w in that ring along with the ring.

This is for internal use by the matrix function.

INPUT:

- `w` - list

OUTPUT:

list, ring

EXAMPLES:

```
sage: sage.matrix.constructor.prepare([-2, Mod(1,7)])
([5, 1], Ring of integers modulo 7)
```

Notice that the elements must all canonically coerce to a common ring (since Sequence is called):

```
sage: sage.matrix.constructor.prepare([2/1, Mod(1,7)])
Traceback (most recent call last):
...
TypeError: unable to find a common ring for all elements
```

sage.matrix.constructor.**prepare_dict**(*w*)

Given a dictionary w of numbers, find a common ring that they all canonically map to, and return the dictionary of images of the elements of w in that ring along with the ring.

This is for internal use by the matrix function.

INPUT:

- `w` - dict

OUTPUT:

dict, ring

EXAMPLES:

```
sage: sage.matrix.constructor.prepare_dict({(0,1):2, (4,10):Mod(1,7)})
({(0, 1): 2, (4, 10): 1}, Ring of integers modulo 7)
```

CHAPTER

THREE

MATRICES OVER AN ARBITRARY RING

AUTHORS:

- William Stein
- Martin Albrecht: conversion to Pyrex
- Jaap Spies: various functions
- Gary Zablackis: fixed a sign bug in generic determinant.
- William Stein and Robert Bradshaw - complete restructuring.
- Rob Beezer - refactor kernel functions.

Elements of matrix spaces are of class `Matrix` (or a class derived from Matrix). They can be either sparse or dense, and can be defined over any base ring.

EXAMPLES:

We create the 2×3 matrix

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}$$

as an element of a matrix space over $\mathbf{Q}$:

```
sage: M = MatrixSpace(QQ,2,3)
sage: A = M([1,2,3, 4,5,6]); A
[1 2 3]
[4 5 6]
sage: A.parent()
Full MatrixSpace of 2 by 3 dense matrices over Rational Field
```

Alternatively, we could create A more directly as follows (which would completely avoid having to create the matrix space):

```
sage: A = matrix(QQ, 2, [1,2,3, 4,5,6]); A
[1 2 3]
[4 5 6]
```

We next change the top-right entry of A. Note that matrix indexing is 0-based in Sage, so the top right entry is $(0, 2)$, which should be thought of as “row number 0, column number 2”.

```
sage: A[0,2] = 389
sage: A
[  1   2 389]
[  4   5   6]
```

Also notice how matrices print. All columns have the same width and entries in a given column are right justified. Next we compute the reduced row echelon form of A.

```
sage: A.rref()
[      1       0 -1933/3]
[      0       1  1550/3]
```

3.1 Indexing

Sage has quite flexible ways of extracting elements or submatrices from a matrix:

```
sage: m=[(1, -2, -1, -1,9), (1, 8, 6, 2,2), (1, 1, -1, 1,4), (-1, 2, -2, -1,4)] ; M =␣
↪matrix(m)
sage: M
[ 1 -2 -1 -1  9]
[ 1  8  6  2  2]
[ 1  1 -1  1  4]
[-1  2 -2 -1  4]
```

Get the 2 x 2 submatrix of M, starting at row index and column index 1:

```
sage: M[1:3,1:3]
[ 8  6]
[ 1 -1]
```

Get the 2 x 3 submatrix of M starting at row index and column index 1:

```
sage: M[1:3,[1..3]]
[ 8  6  2]
[ 1 -1  1]
```

Get the second column of M:

```
sage: M[:,1]
[-2]
[ 8]
[ 1]
[ 2]
```

Get the first row of M:

```
sage: M[0,:]
[ 1 -2 -1 -1  9]
```

Get the last row of M (negative numbers count from the end):

```
sage: M[-1,:]
[-1  2 -2 -1  4]
```

More examples:

```
sage: M[range(2),:]
[ 1 -2 -1 -1  9]
[ 1  8  6  2  2]
sage: M[range(2),4]
[9]
```

```
[2]
sage: M[range(3),range(5)]
[ 1 -2 -1 -1  9]
[ 1  8  6  2  2]
[ 1  1 -1  1  4]

sage: M[3,range(5)]
[-1  2 -2 -1  4]
sage: M[3,:]
[-1  2 -2 -1  4]
sage: M[3,4]
4

sage: M[-1,:]
[-1  2 -2 -1  4]

sage: A = matrix(ZZ,3,4, [3, 2, -5, 0, 1, -1, 1, -4, 1, 0, 1, -3]); A
[ 3  2 -5  0]
[ 1 -1  1 -4]
[ 1  0  1 -3]
```

A series of three numbers, separated by colons, like `n:m:s`, means numbers from `n` up to (but not including) `m`, in steps of `s`. So `0:5:2` means the sequence `[0,2,4]`:

```
sage: A[:,0:4:2]
[ 3 -5]
[ 1  1]
[ 1  1]

sage: A[1:,0:4:2]
[1 1]
[1 1]

sage: A[2::-1,:]
[ 1  0  1 -3]
[ 1 -1  1 -4]
[ 3  2 -5  0]

sage: A[1:,3::-1]
[-4  1 -1  1]
[-3  1  0  1]

sage: A[1:,3::-2]
[-4 -1]
[-3  0]

sage: A[2::-1,3:1:-1]
[-3  1]
[-4  1]
[ 0 -5]
```

We can also change submatrices using these indexing features:

```
sage: M=matrix([(1, -2, -1, -1,9), (1, 8, 6, 2,2), (1, 1, -1, 1,4), (-1, 2, -2, -
↪1,4)]); M
[ 1 -2 -1 -1  9]
[ 1  8  6  2  2]
[ 1  1 -1  1  4]
```

```
[-1  2 -2 -1  4]
```

Set the 2 x 2 submatrix of M, starting at row index and column index 1:

```
sage: M[1:3,1:3] = [[1,0],[0,1]]; M
[ 1 -2 -1 -1  9]
[ 1  1  0  2  2]
[ 1  0  1  1  4]
[-1  2 -2 -1  4]
```

Set the 2 x 3 submatrix of M starting at row index and column index 1:

```
sage: M[1:3,[1..3]] = M[2:4,0:3]; M
[ 1 -2 -1 -1  9]
[ 1  1  0  1  2]
[ 1 -1  2 -2  4]
[-1  2 -2 -1  4]
```

Set part of the first column of M:

```
sage: M[1:,0]=[[2],[3],[4]]; M
[ 1 -2 -1 -1  9]
[ 2  1  0  1  2]
[ 3 -1  2 -2  4]
[ 4  2 -2 -1  4]
```

Or do a similar thing with a vector:

```
sage: M[1:,0]=vector([-2,-3,-4]); M
[ 1 -2 -1 -1  9]
[-2  1  0  1  2]
[-3 -1  2 -2  4]
[-4  2 -2 -1  4]
```

Or a constant:

```
sage: M[1:,0]=30; M
[ 1 -2 -1 -1  9]
[30  1  0  1  2]
[30 -1  2 -2  4]
[30  2 -2 -1  4]
```

Set the first row of M:

```
sage: M[0,:]=[[20,21,22,23,24]]; M
[20 21 22 23 24]
[30  1  0  1  2]
[30 -1  2 -2  4]
[30  2 -2 -1  4]
sage: M[0,:]=vector([0,1,2,3,4]); M
[ 0  1  2  3  4]
[30  1  0  1  2]
[30 -1  2 -2  4]
[30  2 -2 -1  4]
sage: M[0,:]=-3; M
[-3 -3 -3 -3 -3]
[30  1  0  1  2]
[30 -1  2 -2  4]
```

```
[30  2 -2 -1  4]


sage: A = matrix(ZZ,3,4, [3, 2, -5, 0, 1, -1, 1, -4, 1, 0, 1, -3]); A
[ 3  2 -5  0]
[ 1 -1  1 -4]
[ 1  0  1 -3]
```

We can use the step feature of slices to set every other column:

```
sage: A[:,0:3:2] = 5; A
[ 5  2  5  0]
[ 5 -1  5 -4]
[ 5  0  5 -3]

sage: A[1:,0:4:2] = [[100,200],[300,400]]; A
[  5   2   5   0]
[100  -1 200  -4]
[300   0 400  -3]
```

We can also count backwards to flip the matrix upside down:

```
sage: A[::-1,:]=A; A
[300   0 400  -3]
[100  -1 200  -4]
[  5   2   5   0]


sage: A[1:,3::-1]=[[2,3,0,1],[9,8,7,6]]; A
[300   0 400  -3]
[  1   0   3   2]
[  6   7   8   9]

sage: A[1:,::-2] = A[1:,::2]; A
[300   0 400  -3]
[  1   3   3   1]
[  6   8   8   6]

sage: A[::-1,3:1:-1] = [[4,3],[1,2],[-1,-2]]; A
[300   0  -2  -1]
[  1   3   2   1]
[  6   8   3   4]
```

We save and load a matrix:

```
sage: A = matrix(Integers(8),3,range(9))
sage: loads(dumps(A)) == A
True
```

MUTABILITY: Matrices are either immutable or not. When initially created, matrices are typically mutable, so one can change their entries. Once a matrix A is made immutable using `A.set_immutable()` the entries of A cannot be changed, and A can never be made mutable again. However, properties of A such as its rank, characteristic polynomial, etc., are all cached so computations involving A may be more efficient. Once A is made immutable it cannot be changed back. However, one can obtain a mutable copy of A using `copy(A)` .

EXAMPLES:

```
sage: A = matrix(RR,2,[1,10,3.5,2])
sage: A.set_immutable()
sage: copy(A) is A
False
```

The echelon form method always returns immutable matrices with known rank.

EXAMPLES:

```
sage: A = matrix(Integers(8),3,range(9))
sage: A.determinant()
0
sage: A[0,0] = 5
sage: A.determinant()
1
sage: A.set_immutable()
sage: A[0,0] = 5
Traceback (most recent call last):
...
ValueError: matrix is immutable; please change a copy instead (i.e., use copy(M) to
↪change a copy of M).
```

3.1.1 Implementation and Design

Class Diagram (an x means that class is currently supported):

```
x Matrix
x   Matrix_sparse
x     Matrix_generic_sparse
x     Matrix_integer_sparse
x     Matrix_rational_sparse
      Matrix_cyclo_sparse
x     Matrix_modn_sparse
      Matrix_RR_sparse
      Matrix_CC_sparse
      Matrix_RDF_sparse
      Matrix_CDF_sparse

x  Matrix_dense
x     Matrix_generic_dense
x     Matrix_integer_dense
x     Matrix_rational_dense
      Matrix_cyclo_dense    -- idea: restrict scalars to QQ, compute charpoly there,
↪then factor
x     Matrix_modn_dense
      Matrix_RR_dense
      Matrix_CC_dense
x     Matrix_real_double_dense
x     Matrix_complex_double_dense
x     Matrix_complex_ball_dense
```

The corresponding files in the sage/matrix library code directory are named

```
[matrix] [base ring] [dense or sparse].
```

```
New matrices types can only be implemented in Cython.

*********** LEVEL 1  **********
NON-OPTIONAL
For each base field it is *absolutely* essential to completely
implement the following functionality for that base ring:

   * __cinit__     -- should use sig_malloc from ext/stdsage.pxi (only
                      needed if allocate memory)
   * __init__      -- this signature: 'def __init__(self, parent, entries, copy,␣
↪coerce)'
   * __dealloc__   -- use sig_free (only needed if allocate memory)
   * set_unsafe(self, size_t i, size_t j, x) -- doesn't do bounds or any other checks;
↪ assumes x is in self._base_ring
   * get_unsafe(self, size_t i, size_t j) -- doesn't do checks
   * __richcmp__    -- always the same (I don't know why its needed -- bug in PYREX).

Note that the __init__ function must construct the all zero matrix if ``entries ==␣
↪None``.

*********** LEVEL 2  **********

IMPORTANT (and *highly* recommended):

After getting the special class with all level 1 functionality to
work, implement all of the following (they should not change
functionality, except speed (always faster!) in any way):

   * def _pickle(self):
          return data, version
   * def _unpickle(self, data, int version)
          reconstruct matrix from given data and version; may assume _parent, _nrows,␣
↪and _ncols are set.
          Use version numbers >= 0 so if you change the pickle strategy then
          old objects still unpickle.
   * cdef _list -- list of underlying elements (need not be a copy)
   * cdef _dict -- sparse dictionary of underlying elements
   * cdef _add_ -- add two matrices with identical parents
   * _matrix_times_matrix_c_impl -- multiply two matrices with compatible dimensions␣
↪and
                                    identical base rings (both sparse or both dense)
   * cpdef _cmp_ -- compare two matrices with identical parents
   * cdef _lmul_c_impl -- multiply this matrix on the right by a scalar, i.e., self *␣
↪scalar
   * cdef _rmul_c_impl -- multiply this matrix on the left by a scalar, i.e., scalar␣
↪* self
   * __copy__
   * __neg__

The list and dict returned by _list and _dict will *not* be changed
by any internal algorithms and are not accessible to the user.

*********** LEVEL 3  **********
OPTIONAL:

   * cdef _sub_
   * __invert__
   * _multiply_classical
```

```
   * __deepcopy__

Further special support:
   * Matrix windows -- to support Strassen multiplication for a given base ring.
   * Other functions, e.g., transpose, for which knowing the
     specific representation can be helpful.

.. note::

   - For caching, use self.fetch and self.cache.

   - Any method that can change the matrix should call
     ``check_mutability()`` first.  There are also many fast cdef'd bounds checking␣
↪methods.

   - Kernels of matrices
     Implement only a left_kernel() or right_kernel() method, whichever requires
     the least overhead (usually meaning little or no transposing).  Let the
     methods in the matrix2 class handle left, right, generic kernel distinctions.
```

CHAPTER FOUR

MISCELLANEOUS MATRIX FUNCTIONS

`sage.matrix.matrix_misc.`**`permanental_minor_polynomial`**(*A*, *permanent_only=False*, *var='t'*, *prec=None*)

Return the polynomial of the sums of permanental minors of `A`.

INPUT:

- A – a matrix
- $permanent_only$ – if True, return only the permanent of A
- var – name of the polynomial variable
- $prec$ – if prec is not None, truncate the polynomial at precision $prec$

The polynomial of the sums of permanental minors is

$$\sum_{i=0}^{min(nrows,ncols)} p_i(A)x^i$$

where $p_i(A)$ is the i-th permanental minor of A (that can also be obtained through the method `permanental_minor()` via `A.permanental_minor(i)`).

The algorithm implemented by that function has been developed by P. Butera and M. Pernici, see [BP2015]. Its complexity is $O(2^n m^2 n)$ where m and n are the number of rows and columns of A. Moreover, if A is a banded matrix with width w, that is $A_{ij} = 0$ for $|i - j| > w$ and $w < n/2$, then the complexity of the algorithm is $O(4^w (w + 1)n^2)$.

INPUT:

- `A` – matrix
- `permanent_only` – optional boolean. If `True`, only the permanent is computed (might be faster).
- `var` – a variable name

EXAMPLES:

```
sage: from sage.matrix.matrix_misc import permanental_minor_polynomial
sage: m = matrix([[1,1],[1,2]])
sage: permanental_minor_polynomial(m)
3*t^2 + 5*t + 1
sage: permanental_minor_polynomial(m, permanent_only=True)
3
sage: permanental_minor_polynomial(m, prec=2)
5*t + 1
```

```
sage: M = MatrixSpace(ZZ,4,4)
sage: A = M([1,0,1,0,1,0,1,0,1,0,10,10,1,0,1,1])
sage: permanental_minor_polynomial(A)
84*t^3 + 114*t^2 + 28*t + 1
sage: [A.permanental_minor(i) for i in range(5)]
[1, 28, 114, 84, 0]
```

An example over **Q**:

```
sage: M = MatrixSpace(QQ,2,2)
sage: A = M([1/5,2/7,3/2,4/5])
sage: permanental_minor_polynomial(A, True)
103/175
```

An example with polynomial coefficients:

```
sage: R.<a> = PolynomialRing(ZZ)
sage: A = MatrixSpace(R,2)([[a,1], [a,a+1]])
sage: permanental_minor_polynomial(A, True)
a^2 + 2*a
```

A usage of the `var` argument:

```
sage: m = matrix(ZZ,4,[0,1,2,3,1,2,3,0,2,3,0,1,3,0,1,2])
sage: permanental_minor_polynomial(m, var='x')
164*x^4 + 384*x^3 + 172*x^2 + 24*x + 1
```

ALGORITHM:

The permanent $perm(A)$ of a $n \times n$ matrix A is the coefficient of the $x_1 x_2 \ldots x_n$ monomial in

$$\prod_{i=1}^{n}\left(\sum_{j=1}^{n} A_{ij} x_j\right)$$

Evaluating this product one can neglect x_i^2, that is x_i can be considered to be nilpotent of order 2.

To formalize this procedure, consider the algebra $R = K[\eta_1, \eta_2, \ldots, \eta_n]$ where the η_i are commuting, nilpotent of order 2 (i.e. $\eta_i^2 = 0$). Formally it is the quotient ring of the polynomial ring in $\eta_1, \eta_2, \ldots, \eta_n$ quotiented by the ideal generated by the η_i^2.

We will mostly consider the ring $R[t]$ of polynomials over R. We denote a generic element of $R[t]$ by $p(\eta_1, \ldots, \eta_n)$ or $p(\eta_{i_1}, \ldots, \eta_{i_k})$ if we want to emphasize that some monomials in the η_i are missing.

Introduce an "integration" operation $\langle p \rangle$ over R and $R[t]$ consisting in the sum of the coefficients of the non-vanishing monomials in η_i (i.e. the result of setting all variables η_i to 1). Let us emphasize that this is *not* a morphism of algebras as $\langle \eta_1 \rangle^2 = 1$ while $\langle \eta_1^2 \rangle = 0$!

Let us consider an example of computation. Let $p_1 = 1 + t\eta_1 + t\eta_2$ and $p_2 = 1 + t\eta_1 + t\eta_3$. Then

$$p_1 p_2 = 1 + 2t\eta_1 + t(\eta_2 + \eta_3) + t^2(\eta_1\eta_2 + \eta_1\eta_3 + \eta_2\eta_3)$$

and

$$\langle p_1 p_2 \rangle = 1 + 4t + 3t^2$$

In this formalism, the permanent is just

$$perm(A) = \langle \prod_{i=1}^{n} \sum_{j=1}^{n} A_{ij}\eta_j \rangle$$

A useful property of $\langle . \rangle$ which makes this algorithm efficient for band matrices is the following: let $p_1(\eta_1, \ldots, \eta_n)$ and $p_2(\eta_j, \ldots, \eta_n)$ be polynomials in $R[t]$ where $j \geq 1$. Then one has

$$\langle p_1(\eta_1, \ldots, \eta_n) p_2 \rangle = \langle p_1(1, \ldots, 1, \eta_j, \ldots, \eta_n) p_2 \rangle$$

where $\eta_1, .., \eta_{j-1}$ are replaced by 1 in p_1. Informally, we can "integrate" these variables *before* performing the product. More generally, if a monomial η_i is missing in one of the terms of a product of two terms, then it can be integrated in the other term.

Now let us consider an $m \times n$ matrix with $m \leq n$. The *sum of permanental 'k'-minors of 'A'* is

$$perm(A, k) = \sum_{r,c} perm(A_{r,c})$$

where the sum is over the k-subsets r of rows and k-subsets c of columns and $A_{r,c}$ is the submatrix obtained from A by keeping only the rows r and columns c. Of course $perm(A, \min(m, n)) = perm(A)$ and note that $perm(A, 1)$ is just the sum of all entries of the matrix.

The generating function of these sums of permanental minors is

$$g(t) = \left\langle \prod_{i=1}^{m} \left(1 + t \sum_{j=1}^{n} A_{ij} \eta_j \right) \right\rangle$$

In fact the t^k coefficient of $g(t)$ corresponds to choosing k rows of A; η_i is associated to the i-th column; nilpotency avoids having twice the same column in a product of A's.

For more details, see the article [BP2015].

From a technical point of view, the product in $K[\eta_1, \ldots, \eta_n][t]$ is implemented as a subroutine in `prm_mul()` . The indices of the rows and columns actually start at 0, so the variables are $\eta_0, \ldots, \eta_{n-1}$. Polynomials are represented in dictionary form: to a variable η_i is associated the key 2^i (or in Python `1 << i`). The keys associated to products are obtained by considering the development in base 2: to the monomial $\eta_{i_1} \ldots \eta_{i_k}$ is associated the key $2^{i_1} + \ldots + 2^{i_k}$. So the product $\eta_1 \eta_2$ corresponds to the key $6 = (110)_2$ while $\eta_0 \eta_3$ has key $9 = (1001)_2$. In particular all operations on monomials are implemented via bitwise operations on the keys.

`sage.matrix.matrix_misc.`**`prm_mul`**`(`*p1*, *p2*, *mask_free*, *prec*`)`

Return the product of `p1` and `p2` , putting free variables in `mask_free` to 1.

This function is mainly use as a subroutine of `permanental_minor_polynomial()` .

INPUT:

- $p1, p2$ – polynomials as dictionaries
- $mask_free$ – an integer mask that give the list of free variables (the i-th variable is free if the i-th bit of `mask_free` is 1)
- $prec$ – if $prec$ is not None, truncate the product at precision $prec$

EXAMPLES:

```
sage: from sage.matrix.matrix_misc import prm_mul
sage: t = polygen(ZZ, 't')
sage: p1 = {0: 1, 1: t, 4: t}
sage: p2 = {0: 1, 1: t, 2: t}
sage: prm_mul(p1, p2, 1, None)
{0: 2*t + 1, 2: t^2 + t, 4: t^2 + t, 6: t^2}
```

`sage.matrix.matrix_misc.`**`row_iterator`**`(`*A*`)` sage.matrix.matrix_misc. **row_reduced_form** (*M*, *transformation=False*)

This function computes a row reduced form of a matrix over a rational function field $k(x)$, for k a field.

INPUT:

- M - a matrix over $k(x)$ or $k[x]$ for k a field.
- $transformation$ - A boolean (default: $False$). If this boolean is set to $True$ a second matrix is output (see OUTPUT).

OUTPUT:

If $transformation$ is $False$, the output is W, a row reduced form of M.

If $transformation$ is $True$, this function will output a pair (W, N) consisting of two matrices over $k(x)$:

1. W - a row reduced form of M.
2. N - an invertible matrix over $k(x)$ satisfying $NW = M$.

EXAMPLES:

The function expects matrices over the rational function field, but other examples below show how one can provide matrices over the ring of polynomials (whose quotient field is the rational function field).

```
sage: R.<t> = GF(3)['t']
sage: K = FractionField(R)
sage: import sage.matrix.matrix_misc
sage: sage.matrix.matrix_misc.row_reduced_form(matrix([[(t-1)^2/t],[(t-1)]]))
doctest:...: DeprecationWarning: Row reduced form will soon be supported only for
↪matrices of polynomials.
See http://trac.sagemath.org/21024 for details.
[        0]
[(t + 2)/t]
```

The last example shows the usage of the transformation parameter.

:: sage: Fq.<a> = GF(2^3) sage: Fx.<x> = Fq[] sage: A = matrix(Fx,[[x^2+a,x^4+a],[x^3,a*x^4]]) sage: from sage.matrix.matrix_misc import row_reduced_form sage: row_reduced_form(A,transformation=True) ([x^2 + a x^4 + a] [1 0] [x^3 + a*x^2 + a^2 a^2], [a 1])

NOTES:

See docstring for row_reduced_form method of matrices for more information.

CHAPTER

FIVE

ABSTRACT BASE CLASS FOR MATRICES

For design documentation see matrix/docs.py.

class sage.matrix.matrix.**Matrix**

Bases: *sage.matrix.matrix2.Matrix*

sage.matrix.matrix.**is_Matrix**(*x*)

EXAMPLES:

```
sage: from sage.matrix.matrix import is_Matrix
sage: is_Matrix(0)
False
sage: is_Matrix(matrix([[1,2],[3,4]]))
True
```

CHAPTER

SIX

BASE CLASS FOR MATRICES, PART 0

Note: For design documentation see matrix/docs.py.

EXAMPLES:

```
sage: matrix(2,[1,2,3,4])
[1 2]
[3 4]
```

class `sage.matrix.matrix0.`**`Matrix`**

Bases: `sage.structure.element.Matrix`

A generic matrix.

The `Matrix` class is the base class for all matrix classes. To create a `Matrix`, first create a `MatrixSpace`, then coerce a list of elements into the `MatrixSpace`. See the documentation of `MatrixSpace` for more details.

EXAMPLES:

We illustrate matrices and matrix spaces. Note that no actual matrix that you make should have class Matrix; the class should always be derived from Matrix.

```
sage: M = MatrixSpace(CDF,2,3); M
Full MatrixSpace of 2 by 3 dense matrices over Complex Double Field
sage: a = M([1,2,3,  4,5,6]); a
[1.0 2.0 3.0]
[4.0 5.0 6.0]
sage: type(a)
<type 'sage.matrix.matrix_complex_double_dense.Matrix_complex_double_dense'>
sage: parent(a)
Full MatrixSpace of 2 by 3 dense matrices over Complex Double Field
```

```
sage: matrix(CDF, 2,3, [1,2,3, 4,5,6])
[1.0 2.0 3.0]
[4.0 5.0 6.0]
sage: Mat(CDF,2,3)(range(1,7))
[1.0 2.0 3.0]
[4.0 5.0 6.0]
```

```
sage: Q.<i,j,k> = QuaternionAlgebra(QQ, -1,-1)
sage: matrix(Q,2,1,[1,2])
[1]
[2]
```

act_on_polynomial (*f*)
Returns the polynomial f(self*x).

INPUT:

- `self` - an nxn matrix
- `f` - a polynomial in n variables x=(x1,...,xn)

OUTPUT: The polynomial f(self*x).

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: x, y = R.gens()
sage: f = x**2 - y**2
sage: M = MatrixSpace(QQ, 2)
sage: A = M([1,2,3,4])
sage: A.act_on_polynomial(f)
-8*x^2 - 20*x*y - 12*y^2
```

add_multiple_of_column (*i, j, s, start_row=0*)
Add s times column j to column i.

EXAMPLES: We add -1 times the third column to the second column of an integer matrix, remembering to start numbering cols at zero:

```
sage: a = matrix(ZZ,2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: a.add_multiple_of_column(1,2,-1)
sage: a
[ 0 -1  2]
[ 3 -1  5]
```

To add a rational multiple, we first need to change the base ring:

```
sage: a = a.change_ring(QQ)
sage: a.add_multiple_of_column(1,0,1/3)
sage: a
[ 0 -1  2]
[ 3  0  5]
```

If not, we get an error message:

```
sage: a.add_multiple_of_column(1,0,i)
Traceback (most recent call last):
...
TypeError: Multiplying column by Symbolic Ring element cannot be done over␣
↪Rational Field, use change_ring or with_added_multiple_of_column instead.
```

add_multiple_of_row (*i, j, s, start_col=0*)
Add s times row j to row i.

EXAMPLES: We add -3 times the first row to the second row of an integer matrix, remembering to start numbering rows at zero:

```
sage: a = matrix(ZZ,2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: a.add_multiple_of_row(1,0,-3)
```

```
sage: a
[ 0  1  2]
[ 3  1 -1]
```

To add a rational multiple, we first need to change the base ring:

```
sage: a = a.change_ring(QQ)
sage: a.add_multiple_of_row(1,0,1/3)
sage: a
[   0    1    2]
[   3  4/3 -1/3]
```

If not, we get an error message:

```
sage: a.add_multiple_of_row(1,0,i)
Traceback (most recent call last):
...
TypeError: Multiplying row by Symbolic Ring element cannot be done over␣
↪Rational Field, use change_ring or with_added_multiple_of_row instead.
```

anticommutator (*other*)

Return the anticommutator `self` and `other`.

The *anticommutator* of two $n \times n$ matrices A and B is defined as $\{A, B\} := AB + BA$ (sometimes this is written as $[A, B]_+$).

EXAMPLES:

```
sage: A = Matrix(ZZ, 2, 2, range(4))
sage: B = Matrix(ZZ, 2, 2, [0, 1, 0, 0])
sage: A.anticommutator(B)
[2 3]
[0 2]
sage: A.anticommutator(B) == B.anticommutator(A)
True
sage: A.commutator(B) + B.anticommutator(A) == 2*A*B
True
```

base_ring ( )

Returns the base ring of the matrix.

EXAMPLES:

```
sage: m=matrix(QQ,2,[1,2,3,4])
sage: m.base_ring()
Rational Field
```

change_ring (*ring*)

Return the matrix obtained by coercing the entries of this matrix into the given ring.

Always returns a copy (unless self is immutable, in which case returns self).

EXAMPLES:

```
sage: A = Matrix(QQ, 2, 2, [1/2, 1/3, 1/3, 1/4])
sage: A.parent()
 Full MatrixSpace of 2 by 2 dense matrices over Rational Field
sage: A.change_ring(GF(25,'a'))
[3 2]
```

```
[2 4]
sage: A.change_ring(GF(25,'a')).parent()
 Full MatrixSpace of 2 by 2 dense matrices over Finite Field in a of size 5^2
sage: A.change_ring(ZZ)
Traceback (most recent call last):
...
TypeError: matrix has denominators so can't change to ZZ.
```

Changing rings preserves subdivisions:

```
sage: A.subdivide([1], []); A
[1/2 1/3]
[-------]
[1/3 1/4]
sage: A.change_ring(GF(25,'a'))
[3 2]
[---]
[2 4]
```

commutator (*other*)

Return the commutator self*other - other*self.

EXAMPLES:

```
sage: A = Matrix(ZZ, 2, 2, range(4))
sage: B = Matrix(ZZ, 2, 2, [0, 1, 0, 0])
sage: A.commutator(B)
[-2 -3]
[ 0  2]
sage: A.commutator(B) == -B.commutator(A)
True
```

dict ( )

Dictionary of the elements of self with keys pairs (i,j) and values the nonzero entries of self.

It is safe to change the returned dictionary.

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: a = matrix(R,2,[x,y,0, 0,0,2*x+y]); a
[      x       y       0]
[      0       0 2*x + y]
sage: d = a.dict(); d
{(0, 0): x, (0, 1): y, (1, 2): 2*x + y}
```

Notice that changing the returned list does not change a (the list is a copy):

```
sage: d[0,0] = 25
sage: a
[      x       y       0]
[      0       0 2*x + y]
```

dimensions ( )

Returns the dimensions of this matrix as the tuple (nrows, ncols).

EXAMPLES:

```
sage: M = matrix([[1,2,3],[4,5,6]])
sage: N = M.transpose()
sage: M.dimensions()
(2, 3)
sage: N.dimensions()
(3, 2)
```

AUTHORS:

•Benjamin Lundell (2012-02-09): examples

is_alternating ( )

Return `True` if `self` is an alternating matrix.

Here, "alternating matrix" means a square matrix A satisfying $A^T = -A$ and such that the diagonal entries of A are 0. Notice that the condition that the diagonal entries be 0 is not redundant for matrices over arbitrary ground rings (but it is redundant when 2 is invertible in the ground ring). A square matrix A only required to satisfy $A^T = -A$ is said to be "skew-symmetric", and this property is checked by the *is_skew_symmetric()* method.

EXAMPLES:

```
sage: m = matrix(QQ, [[0,2], [-2,0]])
sage: m.is_alternating()
True
sage: m = matrix(QQ, [[1,2], [2,1]])
sage: m.is_alternating()
False
```

In contrast to the property of being skew-symmetric, the property of being alternating does not tolerate nonzero entries on the diagonal even if they are their own negatives:

```
sage: n = matrix(Zmod(4), [[0, 1], [-1, 2]])
sage: n.is_alternating()
False
```

is_dense ( )

Returns True if this is a dense matrix.

In Sage, being dense is a property of the underlying representation, not the number of nonzero entries.

EXAMPLES:

```
sage: matrix(QQ,2,2,range(4)).is_dense()
True
sage: matrix(QQ,2,2,range(4),sparse=True).is_dense()
False
```

is_hermitian ( )

Returns `True` if the matrix is equal to its conjugate-transpose.

OUTPUT:

`True` if the matrix is square and equal to the transpose with every entry conjugated, and `False` otherwise.

Note that if conjugation has no effect on elements of the base ring (such as for integers), then the *is_symmetric()* method is equivalent and faster.

This routine is for matrices over exact rings and so may not work properly for matrices over `RR` or `CC` . For matrices with approximate entries, the rings of double-precision floating-point numbers, `RDF` and `CDF` , are a better choice since the *sage.matrix.matrix_double_dense.Matrix_double_dense.is_hermitian()* method has a tolerance parameter. This provides control over allowing for minor discrepancies between entries when checking equality.

The result is cached.

EXAMPLES:

```
sage: A = matrix(QQbar, [[ 1 + I,  1 - 6*I, -1 - I],
....:                    [-3 - I,     -4*I,     -2],
....:                    [-1 + I, -2 - 8*I,  2 + I]])
sage: A.is_hermitian()
False
sage: B = A*A.conjugate_transpose()
sage: B.is_hermitian()
True
```

Sage has several fields besides the entire complex numbers where conjugation is non-trivial.

```
sage: F.<b> = QuadraticField(-7)
sage: C = matrix(F, [[-2*b - 3,  7*b - 6, -b + 3],
....:                [-2*b - 3, -3*b + 2,   -2*b],
....:                [   b + 1,        0,     -2]])
sage: C.is_hermitian()
False
sage: C = C*C.conjugate_transpose()
sage: C.is_hermitian()
True
```

A matrix that is nearly Hermitian, but for a non-real diagonal entry.

```
sage: A = matrix(QQbar, [[    2,   2-I, 1+4*I],
....:                    [  2+I,   3+I, 2-6*I],
....:                    [1-4*I, 2+6*I,     5]])
sage: A.is_hermitian()
False
sage: A[1,1] = 132
sage: A.is_hermitian()
True
```

Rectangular matrices are never Hermitian.

```
sage: A = matrix(QQbar, 3, 4)
sage: A.is_hermitian()
False
```

A square, empty matrix is trivially Hermitian.

```
sage: A = matrix(QQ, 0, 0)
sage: A.is_hermitian()
True
```

is_immutable ( )

Return True if this matrix is immutable.

See the documentation for self.set_immutable for more details about mutability.

EXAMPLES:

```
sage: A = Matrix(QQ['t','s'], 2, 2, range(4))
sage: A.is_immutable()
False
sage: A.set_immutable()
sage: A.is_immutable()
True
```

is_invertible ( )

Return True if this matrix is invertible.

EXAMPLES: The following matrix is invertible over $\mathbf{Q}$ but not over $\mathbf{Z}$.

```
sage: A = MatrixSpace(ZZ, 2)(range(4))
sage: A.is_invertible()
False
sage: A.matrix_over_field().is_invertible()
True
```

The inverse function is a constructor for matrices over the fraction field, so it can work even if A is not invertible.

```
sage: ~A    # inverse of A
[-3/2  1/2]
[   1    0]
```

The next matrix is invertible over $\mathbf{Z}$.

```
sage: A = MatrixSpace(IntegerRing(),2)([1,10,0,-1])
sage: A.is_invertible()
True
sage: ~A                    # compute the inverse
[ 1 10]
[ 0 -1]
```

The following nontrivial matrix is invertible over $\mathbf{Z}[x]$.

```
sage: R.<x> = PolynomialRing(IntegerRing())
sage: A = MatrixSpace(R,2)([1,x,0,-1])
sage: A.is_invertible()
True
sage: ~A
[ 1  x]
[ 0 -1]
```

is_mutable ( )

Return True if this matrix is mutable.

See the documentation for self.set_immutable for more details about mutability.

EXAMPLES:

```
sage: A = Matrix(QQ['t','s'], 2, 2, range(4))
sage: A.is_mutable()
True
sage: A.set_immutable()
sage: A.is_mutable()
False
```

is_singular ( )

Returns `True` if `self` is singular.

OUTPUT:

A square matrix is singular if it has a zero determinant and this method will return `True` in exactly this case. When the entries of the matrix come from a field, this is equivalent to having a nontrivial kernel, or lacking an inverse, or having linearly dependent rows, or having linearly dependent columns.

For square matrices over a field the methods `is_invertible()` and `is_singular()` are logical opposites. However, it is an error to apply `is_singular()` to a matrix that is not square, while `is_invertible()` will always return `False` for a matrix that is not square.

EXAMPLES:

A singular matrix over the field `QQ`.

```
sage: A = matrix(QQ, 4, [-1,2,-3,6,0,-1,-1,0,-1,1,-5,7,-1,6,5,2])
sage: A.is_singular()
True
sage: A.right_kernel().dimension()
1
```

A matrix that is not singular, i.e. nonsingular, over a field.

```
sage: B = matrix(QQ, 4, [1,-3,-1,-5,2,-5,-2,-7,-2,5,3,4,-1,4,2,6])
sage: B.is_singular()
False
sage: B.left_kernel().dimension()
0
```

For *rectangular* matrices, invertibility is always `False`, but asking about singularity will give an error.

```
sage: C = matrix(QQ, 5, range(30))
sage: C.is_invertible()
False
sage: C.is_singular()
Traceback (most recent call last):
...
ValueError: self must be a square matrix
```

When the base ring is not a field, then a matrix may be both not invertible and not singular.

```
sage: D = matrix(ZZ, 4, [2,0,-4,8,2,1,-2,7,2,5,7,0,0,1,4,-6])
sage: D.is_invertible()
False
sage: D.is_singular()
False
sage: d = D.determinant(); d
2
sage: d.is_unit()
False
```

is_skew_symmetric ( )

Return `True` if `self` is a skew-symmetric matrix.

Here, "skew-symmetric matrix" means a square matrix A satisfying $A^T = -A$. It does not require that the diagonal entries of A are 0 (although this automatically follows from $A^T = -A$ when 2 is invertible in the ground ring over which the matrix is considered). Skew-symmetric matrices A whose diagonal entries are 0 are said to be "alternating", and this property is checked by the `is_alternating()` method.

EXAMPLES:

```
sage: m = matrix(QQ, [[0,2], [-2,0]])
sage: m.is_skew_symmetric()
True
sage: m = matrix(QQ, [[1,2], [2,1]])
sage: m.is_skew_symmetric()
False
```

Skew-symmetric is not the same as alternating when 2 is a zero-divisor in the ground ring:

```
sage: n = matrix(Zmod(4), [[0, 1], [-1, 2]])
sage: n.is_skew_symmetric()
True
```

but yet the diagonal cannot be completely arbitrary in this case:

```
sage: n = matrix(Zmod(4), [[0, 1], [-1, 3]])
sage: n.is_skew_symmetric()
False
```

is_skew_symmetrizable (*return_diag=False*, *positive=True*)

This function takes a square matrix over an *ordered integral domain* and checks if it is skew-symmetrizable. A matrix B is skew-symmetrizable iff there exists an invertible diagonal matrix D such that DB is skew-symmetric.

> **Warning:** Expects `self` to be a matrix over an *ordered integral domain*.

INPUT:

- `return_diag` – bool(default:False) if True and `self` is skew-symmetrizable the diagonal entries of the matrix D are returned.
- `positive` – bool(default:True) if True, the condition that D has positive entries is added.

OUTPUT:

- True – if `self` is skew-symmetrizable and `return_diag` is False
- the diagonal entries of a matrix D such that DB is skew-symmetric – iff `self` is skew-symmetrizable and `return_diag` is True
- False – iff `self` is not skew-symmetrizable

EXAMPLES:

```
sage: matrix([[0,6],[3,0]]).is_skew_symmetrizable(positive=False)
True
sage: matrix([[0,6],[3,0]]).is_skew_symmetrizable(positive=True)
False

sage: M = matrix(4,[0,1,0,0,-1,0,-1,0,0,2,0,1,0,0,-1,0]); M
[ 0  1  0  0]
[-1  0 -1  0]
[ 0  2  0  1]
[ 0  0 -1  0]

sage: M.is_skew_symmetrizable(return_diag=True)
[1, 1, 1/2, 1/2]
```

```
sage: M2 = diagonal_matrix([1,1,1/2,1/2])*M; M2
[   0    1    0    0]
[  -1    0   -1    0]
[   0    1    0  1/2]
[   0    0 -1/2    0]

sage: M2.is_skew_symmetric()
True
```

REFERENCES:

•[FZ2001] S. Fomin, A. Zelevinsky. Cluster Algebras 1: Foundations, arXiv:math/0104151 (2001).

is_sparse ( )

Return True if this is a sparse matrix.

In Sage, being sparse is a property of the underlying representation, not the number of nonzero entries.

EXAMPLES:

```
sage: matrix(QQ,2,2,range(4)).is_sparse()
False
sage: matrix(QQ,2,2,range(4),sparse=True).is_sparse()
True
```

is_square ( )

Return True precisely if this matrix is square, i.e., has the same number of rows and columns.

EXAMPLES:

```
sage: matrix(QQ,2,2,range(4)).is_square()
True
sage: matrix(QQ,2,3,range(6)).is_square()
False
```

is_symmetric ( )

Returns True if this is a symmetric matrix.

A symmetric matrix is necessarily square.

EXAMPLES:

```
sage: m=Matrix(QQ,2,range(0,4))
sage: m.is_symmetric()
False

sage: m=Matrix(QQ,2,(1,1,1,1,1,1))
sage: m.is_symmetric()
False

sage: m=Matrix(QQ,1,(2,))
sage: m.is_symmetric()
True
```

is_symmetrizable (*return_diag=False*, *positive=True*)

This function takes a square matrix over an *ordered integral domain* and checks if it is symmetrizable. A matrix B is symmetrizable iff there exists an invertible diagonal matrix D such that DB is symmetric.

> **Warning:** Expects `self` to be a matrix over an *ordered integral domain*.

INPUT:

- `return_diag` – bool(default:False) if True and `self` is symmetrizable the diagonal entries of the matrix D are returned.
- `positive` – bool(default:True) if True, the condition that D has positive entries is added.

OUTPUT:

- True – if `self` is symmetrizable and `return_diag` is False
- the diagonal entries of a matrix D such that DB is symmetric – iff `self` is symmetrizable and `return_diag` is True
- False – iff `self` is not symmetrizable

EXAMPLES:

```
sage: matrix([[0,6],[3,0]]).is_symmetrizable(positive=False)
True

sage: matrix([[0,6],[3,0]]).is_symmetrizable(positive=True)
True

sage: matrix([[0,6],[0,0]]).is_symmetrizable(return_diag=True)
False

sage: matrix([2]).is_symmetrizable(positive=True)
True

sage: matrix([[1,2],[3,4]]).is_symmetrizable(return_diag=true)
[1, 2/3]
```

REFERENCES:

- [FZ2001] S. Fomin, A. Zelevinsky. Cluster Algebras 1: Foundations, arXiv:math/0104151 (2001).

is_unit ( )

Return True if this matrix is invertible.

EXAMPLES: The following matrix is invertible over **Q** but not over **Z**.

```
sage: A = MatrixSpace(ZZ, 2)(range(4))
sage: A.is_invertible()
False
sage: A.matrix_over_field().is_invertible()
True
```

The inverse function is a constructor for matrices over the fraction field, so it can work even if A is not invertible.

```
sage: ~A   # inverse of A
[-3/2  1/2]
[   1    0]
```

The next matrix is invertible over **Z**.

```
sage: A = MatrixSpace(IntegerRing(),2)([1,10,0,-1])
sage: A.is_invertible()
True
sage: ~A                    # compute the inverse
[ 1 10]
[ 0 -1]
```

The following nontrivial matrix is invertible over $\mathbf{Z}[x]$.

```
sage: R.<x> = PolynomialRing(IntegerRing())
sage: A = MatrixSpace(R,2)([1,x,0,-1])
sage: A.is_invertible()
True
sage: ~A
[ 1  x]
[ 0 -1]
```

iterates (*v*, *n*, *rows=True*)

Let A be this matrix and v be a free module element. If rows is True, return a matrix whose rows are the entries of the following vectors:

$$v, vA, vA^2, \ldots, vA^{n-1}.$$

If rows is False, return a matrix whose columns are the entries of the following vectors:

$$v, Av, A^2v, \ldots, A^{n-1}v.$$

INPUT:

- `v` - free module element
- `n` - nonnegative integer

EXAMPLES:

```
sage: A = matrix(ZZ,2, [1,1,3,5]); A
[1 1]
[3 5]
sage: v = vector([1,0])
sage: A.iterates(v,0)
[]
sage: A.iterates(v,5)
[  1   0]
[  1   1]
[  4   6]
[ 22  34]
[124 192]
```

Another example:

```
sage: a = matrix(ZZ,3,range(9)); a
[0 1 2]
[3 4 5]
[6 7 8]
sage: v = vector([1,0,0])
sage: a.iterates(v,4)
[  1   0   0]
[  0   1   2]
[ 15  18  21]
```

```
[180 234 288]
sage: a.iterates(v,4,rows=False)
[  1   0  15 180]
[  0   3  42 558]
[  0   6  69 936]
```

linear_combination_of_columns (*v*)

Return the linear combination of the columns of `self` given by the coefficients in the list `v` .

INPUT:

- `v` - a list of scalars. The length can be less than the number of columns of `self` but not greater.

OUTPUT:

The vector (or free module element) that is a linear combination of the columns of `self` . If the list of scalars has fewer entries than the number of columns, additional zeros are appended to the list until it has as many entries as the number of columns.

EXAMPLES:

```
sage: a = matrix(ZZ,2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: a.linear_combination_of_columns([1,1,1])
(3, 12)

sage: a.linear_combination_of_columns([0,0,0])
(0, 0)

sage: a.linear_combination_of_columns([1/2,2/3,3/4])
(13/6, 95/12)
```

The list `v` can be anything that is iterable. Perhaps most naturally, a vector may be used.

```
sage: v = vector(ZZ, [1,2,3])
sage: a.linear_combination_of_columns(v)
(8, 26)
```

We check that a matrix with no columns behaves properly.

```
sage: matrix(QQ,2,0).linear_combination_of_columns([])
(0, 0)
```

The object returned is a vector, or a free module element.

```
sage: B = matrix(ZZ, 4, 3, range(12))
sage: w = B.linear_combination_of_columns([-1,2,-3])
sage: w
(-4, -10, -16, -22)
sage: w.parent()
Ambient free module of rank 4 over the principal ideal domain Integer Ring
sage: x = B.linear_combination_of_columns([1/2,1/3,1/4])
sage: x
(5/6, 49/12, 22/3, 127/12)
sage: x.parent()
Vector space of dimension 4 over Rational Field
```

The length of v can be less than the number of columns, but not greater.

```
sage: A = matrix(QQ,3,5, range(15))
sage: A.linear_combination_of_columns([1,-2,3,-4])
(-8, -18, -28)
sage: A.linear_combination_of_columns([1,2,3,4,5,6])
Traceback (most recent call last):
...
ValueError: length of v must be at most the number of columns of self
```

linear_combination_of_rows (*v*)

Return the linear combination of the rows of `self` given by the coefficients in the list `v` .

INPUT:

- `v` - a list of scalars. The length can be less than the number of rows of `self` but not greater.

OUTPUT:

The vector (or free module element) that is a linear combination of the rows of `self` . If the list of scalars has fewer entries than the number of rows, additional zeros are appended to the list until it has as many entries as the number of rows.

EXAMPLES:

```
sage: a = matrix(ZZ,2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: a.linear_combination_of_rows([1,2])
(6, 9, 12)

sage: a.linear_combination_of_rows([0,0])
(0, 0, 0)

sage: a.linear_combination_of_rows([1/2,2/3])
(2, 19/6, 13/3)
```

The list `v` can be anything that is iterable. Perhaps most naturally, a vector may be used.

```
sage: v = vector(ZZ, [1,2])
sage: a.linear_combination_of_rows(v)
(6, 9, 12)
```

We check that a matrix with no rows behaves properly.

```
sage: matrix(QQ,0,2).linear_combination_of_rows([])
(0, 0)
```

The object returned is a vector, or a free module element.

```
sage: B = matrix(ZZ, 4, 3, range(12))
sage: w = B.linear_combination_of_rows([-1,2,-3,4])
sage: w
(24, 26, 28)
sage: w.parent()
Ambient free module of rank 3 over the principal ideal domain Integer Ring
sage: x = B.linear_combination_of_rows([1/2,1/3,1/4,1/5])
sage: x
(43/10, 67/12, 103/15)
sage: x.parent()
Vector space of dimension 3 over Rational Field
```

The length of v can be less than the number of rows, but not greater.

```
sage: A = matrix(QQ,3,4,range(12))
sage: A.linear_combination_of_rows([2,3])
(12, 17, 22, 27)
sage: A.linear_combination_of_rows([1,2,3,4])
Traceback (most recent call last):
...
ValueError: length of v must be at most the number of rows of self
```

list ( )

List of the elements of `self` ordered by elements in each row. It is safe to change the returned list.

> **Warning:** This function returns a list of the entries in the matrix `self`. It does not return a list of the rows of `self`, so it is different than the output of `list(self)`, which returns `[self[0],self[1],...]`.

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: a = matrix(R,2,[x,y,x*y, y,x,2*x+y]); a
[      x       y     x*y]
[      y       x 2*x + y]
sage: v = a.list(); v
[x, y, x*y, y, x, 2*x + y]
```

Note that list(a) is different than a.list():

```
sage: a.list()
[x, y, x*y, y, x, 2*x + y]
sage: list(a)
[(x, y, x*y), (y, x, 2*x + y)]
```

Notice that changing the returned list does not change a (the list is a copy):

```
sage: v[0] = 25
sage: a
[      x       y     x*y]
[      y       x 2*x + y]
```

mod (*p*)

Return matrix mod p, over the reduced ring.

EXAMPLES:

```
sage: M = matrix(ZZ, 2, 2, [5, 9, 13, 15])
sage: M.mod(7)
[5 2]
[6 1]
sage: parent(M.mod(7))
Full MatrixSpace of 2 by 2 dense matrices over Ring of integers modulo 7
```

multiplicative_order ( )

Return the multiplicative order of this matrix, which must therefore be invertible.

EXAMPLES:

```
sage: A = matrix(GF(59),3,[10,56,39,53,56,33,58,24,55])
sage: A.multiplicative_order()
580
sage: (A^580).is_one()
True
```

```
sage: B = matrix(GF(10007^3,'b'),0)
sage: B.multiplicative_order()
1
```

```
sage: C = matrix(GF(2^10,'c'),2,3,[1]*6)
sage: C.multiplicative_order()
Traceback (most recent call last):
...
ArithmeticError: self must be invertible ...
```

```
sage: D = matrix(IntegerModRing(6),3,[5,5,3,0,2,5,5,4,0])
sage: D.multiplicative_order()
Traceback (most recent call last):
...
NotImplementedError: ... only ... over finite fields
```

```
sage: E = MatrixSpace(GF(11^2,'e'),5).random_element()
sage: (E^E.multiplicative_order()).is_one()
True
```

REFERENCES:

- Frank Celler and C. R. Leedham-Green, "Calculating the Order of an Invertible Matrix", 1997

mutate (*k*)

Mutates `self` at row and column index `k`.

> **Warning:** Only makes sense if `self` is skew-symmetrizable.

INPUT:

- `k` – integer at which row/column `self` is mutated.

EXAMPLES:

Mutation of the B-matrix of the quiver of type A_3:

```
sage: M = matrix(ZZ,3,[0,1,0,-1,0,-1,0,1,0]); M
[ 0  1  0]
[-1  0 -1]
[ 0  1  0]

sage: M.mutate(0); M
[ 0 -1  0]
[ 1  0 -1]
[ 0  1  0]

sage: M.mutate(1); M
[ 0  1 -1]
[-1  0  1]
```

```
[ 1 -1  0]

sage: M = matrix(ZZ,6,[0,1,0,-1,0,-1,0,1,0,1,0,0,0,1,0,0,0,1]); M
[ 0  1  0]
[-1  0 -1]
[ 0  1  0]
[ 1  0  0]
[ 0  1  0]
[ 0  0  1]

sage: M.mutate(0); M
[ 0 -1  0]
[ 1  0 -1]
[ 0  1  0]
[-1  1  0]
[ 0  1  0]
[ 0  0  1]
```

REFERENCES:

•[FZ2001] S. Fomin, A. Zelevinsky. Cluster Algebras 1: Foundations, arXiv:math/0104151 (2001).

ncols ( )

Return the number of columns of this matrix.

EXAMPLES:

```
sage: M = MatrixSpace(QQ, 2, 3)
sage: A = M([1,2,3, 4,5,6])
sage: A
[1 2 3]
[4 5 6]
sage: A.ncols()
3
sage: A.nrows()
2
```

AUTHORS:

•Naqi Jaffery (2006-01-24): examples

nonpivots ( )

Return the list of i such that the i-th column of self is NOT a pivot column of the reduced row echelon form of self.

OUTPUT: sorted tuple of (Python) integers

EXAMPLES:

```
sage: a = matrix(QQ,3,3,range(9)); a
[0 1 2]
[3 4 5]
[6 7 8]
sage: a.echelon_form()
[ 1  0 -1]
[ 0  1  2]
[ 0  0  0]
sage: a.nonpivots()
(2,)
```

nonzero_positions (*copy=True*, *column_order=False*)

Returns the sorted list of pairs (i,j) such that self[i,j] != 0.

INPUT:

- `copy` - (default: True) It is safe to change the resulting list (unless you give the option copy=False).
- `column_order` - (default: False) If true, returns the list of pairs (i,j) such that self[i,j] != 0, but sorted by columns, i.e., column j=0 entries occur first, then column j=1 entries, etc.

EXAMPLES:

```
sage: a = matrix(QQ, 2,3, [1,2,0,2,0,0]); a
[1 2 0]
[2 0 0]
sage: a.nonzero_positions()
[(0, 0), (0, 1), (1, 0)]
sage: a.nonzero_positions(copy=False)
[(0, 0), (0, 1), (1, 0)]
sage: a.nonzero_positions(column_order=True)
[(0, 0), (1, 0), (0, 1)]
sage: a = matrix(QQ, 2,3, [1,2,0,2,0,0], sparse=True); a
[1 2 0]
[2 0 0]
sage: a.nonzero_positions()
[(0, 0), (0, 1), (1, 0)]
sage: a.nonzero_positions(copy=False)
[(0, 0), (0, 1), (1, 0)]
sage: a.nonzero_positions(column_order=True)
[(0, 0), (1, 0), (0, 1)]
```

nonzero_positions_in_column (*i*)

Return a sorted list of the integers j such that self[j,i] is nonzero, i.e., such that the j-th position of the i-th column is nonzero.

INPUT:

- `i` - an integer

OUTPUT: list

EXAMPLES:

```
sage: a = matrix(QQ, 3,2, [1,2,0,2,0,0]); a
[1 2]
[0 2]
[0 0]
sage: a.nonzero_positions_in_column(0)
[0]
sage: a.nonzero_positions_in_column(1)
[0, 1]
```

You'll get an IndexError, if you select an invalid column:

```
sage: a.nonzero_positions_in_column(2)
Traceback (most recent call last):
...
IndexError: matrix column index out of range
```

nonzero_positions_in_row (*i*)

Return the integers j such that self[i,j] is nonzero, i.e., such that the j-th position of the i-th row is nonzero.

INPUT:

- `i` - an integer

OUTPUT: list

EXAMPLES:

```
sage: a = matrix(QQ, 3,2, [1,2,0,2,0,0]); a
[1 2]
[0 2]
[0 0]
sage: a.nonzero_positions_in_row(0)
[0, 1]
sage: a.nonzero_positions_in_row(1)
[1]
sage: a.nonzero_positions_in_row(2)
[]
```

nrows ( )

Return the number of rows of this matrix.

EXAMPLES:

```
sage: M = MatrixSpace(QQ,6,7)
sage: A = M([1,2,3,4,5,6,7, 22,3/4,34,11,7,5,3, 99,65,1/2,2/3,3/5,4/5,5/6,␣
↪9,8/9, 9/8,7/6,6/7,76,4, 0,9,8,7,6,5,4, 123,99,91,28,6,1024,1])
sage: A
[   1    2    3    4    5    6    7]
[  22  3/4   34   11    7    5    3]
[  99   65  1/2  2/3  3/5  4/5  5/6]
[   9  8/9  9/8  7/6  6/7   76    4]
[   0    9    8    7    6    5    4]
[ 123   99   91   28    6 1024    1]
sage: A.ncols()
7
sage: A.nrows()
6
```

AUTHORS:

- Naqi Jaffery (2006-01-24): examples

permute_columns (*permutation*)

Permute the columns of `self` by applying the permutation group element `permutation`.

As a permutation group element acts on integers $\{1, \dots, n\}$ the columns are considered as being numbered from 1 for this operation.

INPUT:

- `permutation` – a `PermutationGroupElement`.

EXAMPLES: We create a matrix:

```
sage: M =␣
↪matrix(ZZ,[[1,0,0,0,0],[0,2,0,0,0],[0,0,3,0,0],[0,0,0,4,0],[0,0,0,0,5]])
sage: M
[1 0 0 0 0]
[0 2 0 0 0]
[0 0 3 0 0]
```

```
[0 0 0 4 0]
[0 0 0 0 5]
```

Next of all, create a permutation group element and act on `M` with it:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: sigma, tau = G.gens()
sage: sigma
(1,2,3)(4,5)
sage: M.permute_columns(sigma)
sage: M
[0 0 1 0 0]
[2 0 0 0 0]
[0 3 0 0 0]
[0 0 0 0 4]
[0 0 0 5 0]
```

permute_rows (*permutation*)

Permute the rows of `self` by applying the permutation group element `permutation`.

As a permutation group element acts on integers $\{1, \ldots, n\}$ the rows are considered as being numbered from 1 for this operation.

INPUT:

- `permutation` – a `PermutationGroupElement`

EXAMPLES: We create a matrix:

```
sage: M = 
↪matrix(ZZ,[[1,0,0,0,0],[0,2,0,0,0],[0,0,3,0,0],[0,0,0,4,0],[0,0,0,0,5]])
sage: M
[1 0 0 0 0]
[0 2 0 0 0]
[0 0 3 0 0]
[0 0 0 4 0]
[0 0 0 0 5]
```

Next of all, create a permutation group element and act on `M`:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: sigma, tau = G.gens()
sage: sigma
(1,2,3)(4,5)
sage: M.permute_rows(sigma)
sage: M
[0 2 0 0 0]
[0 0 3 0 0]
[1 0 0 0 0]
[0 0 0 0 5]
[0 0 0 4 0]
```

permute_rows_and_columns (*row_permutation*, *column_permutation*)

Permute the rows and columns of `self` by applying the permutation group elements `row_permutation` and `column_permutation` respectively.

As a permutation group element acts on integers $\{1, \ldots, n\}$ the rows and columns are considered as being numbered from 1 for this operation.

INPUT:

- row_permutation – a PermutationGroupElement
- column_permutation – a PermutationGroupElement

OUTPUT:

- A matrix.

EXAMPLES: We create a matrix:

```
sage: M =␣
↪matrix(ZZ,[[1,0,0,0,0],[0,2,0,0,0],[0,0,3,0,0],[0,0,0,4,0],[0,0,0,0,5]])
sage: M
[1 0 0 0 0]
[0 2 0 0 0]
[0 0 3 0 0]
[0 0 0 4 0]
[0 0 0 0 5]
```

Next of all, create a permutation group element and act on M :

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: sigma, tau = G.gens()
sage: sigma
(1,2,3)(4,5)
sage: M.permute_rows_and_columns(sigma,tau)
sage: M
[2 0 0 0 0]
[0 3 0 0 0]
[0 0 0 0 1]
[0 0 0 5 0]
[0 0 4 0 0]
```

pivots ( )

Return the pivot column positions of this matrix.

OUTPUT: a tuple of Python integers: the position of the first nonzero entry in each row of the echelon form.

This returns a tuple so it is immutable; see trac ticket #10752.

EXAMPLES:

```
sage: A = matrix(QQ, 2, 2, range(4))
sage: A.pivots()
(0, 1)
```

rank ( )

Return the rank of this matrix.

EXAMPLES:

```
sage: m = matrix(GF(7),5,range(25))
sage: m.rank()
2
```

Rank is not implemented over the integers modulo a composite yet.:

```
sage: m = matrix(Integers(4), 2, [2,2,2,2])
sage: m.rank()
Traceback (most recent call last):
```

```
...
NotImplementedError: Echelon form not implemented over 'Ring of integers␣
↪modulo 4'.
```

rescale_col (*i*, *s*, *start_row=0*)

Replace i-th col of self by s times i-th col of self.

INPUT:

- `i` - ith column
- `s` - scalar
- `start_row` - only rescale entries at this row and lower

EXAMPLES: We rescale the last column of a matrix over the rational numbers:

```
sage: a = matrix(QQ,2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: a.rescale_col(2,1/2); a
[  0   1   1]
[  3   4 5/2]
sage: R.<x> = QQ[]
```

We rescale the last column of a matrix over a polynomial ring:

```
sage: a = matrix(R,2,3,[1,x,x^2,x^3,x^4,x^5]); a
[  1   x x^2]
[x^3 x^4 x^5]
sage: a.rescale_col(2,1/2); a
[      1         x 1/2*x^2]
[    x^3       x^4 1/2*x^5]
```

We try and fail to rescale a matrix over the integers by a non-integer:

```
sage: a = matrix(ZZ,2,3,[0,1,2, 3,4,4]); a
[0 1 2]
[3 4 4]
sage: a.rescale_col(2,1/2)
Traceback (most recent call last):
...
TypeError: Rescaling column by Rational Field element cannot be done over␣
↪Integer Ring, use change_ring or with_rescaled_col instead.
```

To rescale the matrix by 1/2, you must change the base ring to the rationals:

```
sage: a = a.change_ring(QQ); a
[0 1 2]
[3 4 4]
sage: a.rescale_col(2,1/2); a
[0 1 1]
[3 4 2]
```

rescale_row (*i*, *s*, *start_col=0*)

Replace i-th row of self by s times i-th row of self.

INPUT:

- `i` - ith row

- `s` - scalar
- `start_col` - only rescale entries at this column and to the right

EXAMPLES: We rescale the second row of a matrix over the rational numbers:

```
sage: a = matrix(QQ,3,range(6)); a
[0 1]
[2 3]
[4 5]
sage: a.rescale_row(1,1/2); a
[ 0   1]
[ 1 3/2]
[ 4   5]
```

We rescale the second row of a matrix over a polynomial ring:

```
sage: R.<x> = QQ[]
sage: a = matrix(R,3,[1,x,x^2,x^3,x^4,x^5]);a
[  1   x]
[x^2 x^3]
[x^4 x^5]
sage: a.rescale_row(1,1/2); a
[      1       x]
[1/2*x^2 1/2*x^3]
[    x^4     x^5]
```

We try and fail to rescale a matrix over the integers by a non-integer:

```
sage: a = matrix(ZZ,2,3,[0,1,2, 3,4,4]); a
[0 1 2]
[3 4 4]
sage: a.rescale_row(1,1/2)
Traceback (most recent call last):
...
TypeError: Rescaling row by Rational Field element cannot be done over␣
↪Integer Ring, use change_ring or with_rescaled_row instead.
```

To rescale the matrix by 1/2, you must change the base ring to the rationals:

```
sage: a = a.change_ring(QQ); a
[0 1 2]
[3 4 4]
sage: a.rescale_col(1,1/2); a
[  0 1/2   2]
[  3   2   4]
```

set_col_to_multiple_of_col (*i, j, s*)

Set column i equal to s times column j.

EXAMPLES: We change the second column to -3 times the first column.

```
sage: a = matrix(ZZ,2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: a.set_col_to_multiple_of_col(1,0,-3)
sage: a
[ 0  0  2]
[ 3 -9  5]
```

If we try to multiply a column by a rational number, we get an error message:

```
sage: a.set_col_to_multiple_of_col(1,0,1/2)
Traceback (most recent call last):
...
TypeError: Multiplying column by Rational Field element cannot be done over␣
↪Integer Ring, use change_ring or with_col_set_to_multiple_of_col instead.
```

set_immutable ( )

Call this function to set the matrix as immutable.

Matrices are always mutable by default, i.e., you can change their entries using `A[i,j] = x`. However, mutable matrices aren't hashable, so can't be used as keys in dictionaries, etc. Also, often when implementing a class, you might compute a matrix associated to it, e.g., the matrix of a Hecke operator. If you return this matrix to the user you're really returning a reference and the user could then change an entry; this could be confusing. Thus you should set such a matrix immutable.

EXAMPLES:

```
sage: A = Matrix(QQ, 2, 2, range(4))
sage: A.is_mutable()
True
sage: A[0,0] = 10
sage: A
[10  1]
[ 2  3]
```

Mutable matrices are not hashable, so can't be used as keys for dictionaries:

```
sage: hash(A)
Traceback (most recent call last):
...
TypeError: mutable matrices are unhashable
sage: v = {A:1}
Traceback (most recent call last):
...
TypeError: mutable matrices are unhashable
```

If we make A immutable it suddenly is hashable.

```
sage: A.set_immutable()
sage: A.is_mutable()
False
sage: A[0,0] = 10
Traceback (most recent call last):
...
ValueError: matrix is immutable; please change a copy instead (i.e., use␣
↪copy(M) to change a copy of M).
sage: hash(A) #random
12
sage: v = {A:1}; v
{[10  1]
 [ 2  3]: 1}
```

set_row_to_multiple_of_row (*i, j, s*)

Set row i equal to s times row j.

EXAMPLES: We change the second row to -3 times the first row:

```
sage: a = matrix(ZZ,2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: a.set_row_to_multiple_of_row(1,0,-3)
sage: a
[ 0  1  2]
[ 0 -3 -6]
```

If we try to multiply a row by a rational number, we get an error message:

```
sage: a.set_row_to_multiple_of_row(1,0,1/2)
Traceback (most recent call last):
...
TypeError: Multiplying row by Rational Field element cannot be done over
↪Integer Ring, use change_ring or with_row_set_to_multiple_of_row instead.
```

str (*rep_mapping=None*, *zero=None*, *plus_one=None*, *minus_one=None*, *unicode=False*, *shape=None*)

Return a nice string representation of the matrix.

INPUT:

- `rep_mapping` - a dictionary or callable used to override the usual representation of elements.

  If `rep_mapping` is a dictionary then keys should be elements of the base ring and values the desired string representation. Values sent in via the other keyword arguments will override values in the dictionary. Use of a dictionary can potentially take a very long time due to the need to hash entries of the matrix. Matrices with entries from `QQbar` are one example.

  If `rep_mapping` is callable then it will be called with elements of the matrix and must return a string. Simply call `repr()` on elements which should have the default representation.

- `zero` - string (default: `None`); if not `None` use the value of `zero` as the representation of the zero element.

- `plus_one` - string (default: `None`); if not `None` use the value of `plus_one` as the representation of the one element.

- `minus_one` - string (default: `None`); if not `None` use the value of `minus_one` as the representation of the negative of the one element.

- `unicode` - boolean (default: `False`). Whether to use Unicode symbols instead of ASCII symbols for brackets and subdivision lines.

- `shape` - one of `"square"` or `"round"` (default: `None`). Switches between round and square brackets. The default depends on the setting of the `unicode` keyword argument. For Unicode symbols, the default is round brackets in accordance with the TeX rendering, while the ASCII rendering defaults to square brackets.

EXAMPLES:

```
sage: R = PolynomialRing(QQ,6,'z')
sage: a = matrix(2,3, R.gens())
sage: a.__repr__()
'[z0 z1 z2]\n[z3 z4 z5]'

sage: M = matrix([[1,0],[2,-1]])
sage: M.str()
'[ 1  0]\n[ 2 -1]'
sage: M.str(plus_one='+',minus_one='-',zero='.')
'[+ .]\n[2 -]'
```

```
sage: M.str({1:"not this one",2:"II"},minus_one="*",plus_one="I")
'[ I  0]\n[II  *]'

sage: def print_entry(x):
....:   if x>0:
....:       return '+'
....:   elif x<0:
....:       return '-'
....:   else: return '.'
...
sage: M.str(print_entry)
'[+ .]\n[+ -]'
sage: M.str(repr)
'[ 1  0]\n[ 2 -1]'

sage: M = matrix([[1,2,3],[4,5,6],[7,8,9]])
sage: M.subdivide(None, 2)
sage: print(M.str(unicode=True))
⎛1 2│3⎞
⎜4 5│6⎟
⎝7 8│9⎠
sage: M.subdivide([0,1,1,3], [0,2,3,3])
sage: print(M.str(unicode=True, shape="square"))
⎡+---+-++⎤
⎢|1 2|3||⎥
⎢+---+-++⎥
⎢+---+-++⎥
⎢|4 5|6||⎥
⎢|7 8|9||⎥
⎣+---+-++⎦
```

swap_columns (*c1*, *c2*)

Swap columns c1 and c2 of self.

EXAMPLES: We create a rational matrix:

```
sage: M = MatrixSpace(QQ,3,3)
sage: A = M([1,9,-7,4/5,4,3,6,4,3])
sage: A
[  1   9  -7]
[4/5   4   3]
[  6   4   3]
```

Since the first column is numbered zero, this swaps the second and third columns:

```
sage: A.swap_columns(1,2); A
[  1  -7   9]
[4/5   3   4]
[  6   3   4]
```

swap_rows (*r1*, *r2*)

Swap rows r1 and r2 of self.

EXAMPLES: We create a rational matrix:

```
sage: M = MatrixSpace(QQ,3,3)
sage: A = M([1,9,-7,4/5,4,3,6,4,3])
sage: A
[  1   9  -7]
[4/5   4   3]
[  6   4   3]
```

Since the first row is numbered zero, this swaps the first and third rows:

```
sage: A.swap_rows(0,2); A
[  6   4   3]
[4/5   4   3]
[  1   9  -7]
```

with_added_multiple_of_column (*i*, *j*, *s*, *start_row=0*)

Add s times column j to column i, returning new matrix.

EXAMPLES: We add -1 times the third column to the second column of an integer matrix, remembering to start numbering cols at zero:

```
sage: a = matrix(ZZ,2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: b = a.with_added_multiple_of_column(1,2,-1); b
[ 0 -1  2]
[ 3 -1  5]
```

The original matrix is unchanged:

```
sage: a
[0 1 2]
[3 4 5]
```

Adding a rational multiple is okay, and reassigning a variable is okay:

```
sage: a = a.with_added_multiple_of_column(0,1,1/3); a
[ 1/3    1    2]
[13/3    4    5]
```

with_added_multiple_of_row (*i*, *j*, *s*, *start_col=0*)

Add s times row j to row i, returning new matrix.

EXAMPLES: We add -3 times the first row to the second row of an integer matrix, remembering to start numbering rows at zero:

```
sage: a = matrix(ZZ,2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: b = a.with_added_multiple_of_row(1,0,-3); b
[ 0  1  2]
[ 3  1 -1]
```

The original matrix is unchanged:

```
sage: a
[0 1 2]
[3 4 5]
```

Adding a rational multiple is okay, and reassigning a variable is okay:

```
sage: a = a.with_added_multiple_of_row(0,1,1/3); a
[    1  7/3 11/3]
[    3    4    5]
```

with_col_set_to_multiple_of_col (*i*, *j*, *s*)

Set column i equal to s times column j, returning a new matrix.

EXAMPLES: We change the second column to -3 times the first column.

```
sage: a = matrix(ZZ,2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: b = a.with_col_set_to_multiple_of_col(1,0,-3); b
[ 0  0  2]
[ 3 -9  5]
```

Note that the original matrix is unchanged:

```
sage: a
[0 1 2]
[3 4 5]
```

Adding a rational multiple is okay, and reassigning a variable is okay:

```
sage: a = a.with_col_set_to_multiple_of_col(1,0,1/2); a
[   0   0   2]
[   3 3/2   5]
```

with_permuted_columns (*permutation*)

Return the matrix obtained from permuting the columns of `self` by applying the permutation group element `permutation`.

As a permutation group element acts on integers $\{1, \ldots, n\}$ the columns are considered as being numbered from 1 for this operation.

INPUT:

- `permutation`, a `PermutationGroupElement`

OUTPUT:

- A matrix.

EXAMPLES: We create some matrix:

```
sage: M = matrix(ZZ,[[1,0,0,0,0],[0,2,0,0,0],[0,0,3,0,0],[0,0,0,4,0],[0,0,0,0,5]])
sage: M
[1 0 0 0 0]
[0 2 0 0 0]
[0 0 3 0 0]
[0 0 0 4 0]
[0 0 0 0 5]
```

Next of all, create a permutation group element and act on `M`:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: sigma, tau = G.gens()
sage: sigma
```

```
(1,2,3)(4,5)
sage: M.with_permuted_columns(sigma)
[0 0 1 0 0]
[2 0 0 0 0]
[0 3 0 0 0]
[0 0 0 0 4]
[0 0 0 5 0]
```

with_permuted_rows (*permutation*)

Return the matrix obtained from permuting the rows of `self` by applying the permutation group element `permutation`.

As a permutation group element acts on integers $\{1, \ldots, n\}$ the rows are considered as being numbered from 1 for this operation.

INPUT:

- `permutation` – a `PermutationGroupElement`

OUTPUT:

- A matrix.

EXAMPLES: We create a matrix:

```
sage: M =␣
↪matrix(ZZ,[[1,0,0,0,0],[0,2,0,0,0],[0,0,3,0,0],[0,0,0,4,0],[0,0,0,0,5]])
sage: M
[1 0 0 0 0]
[0 2 0 0 0]
[0 0 3 0 0]
[0 0 0 4 0]
[0 0 0 0 5]
```

Next of all, create a permutation group element and act on `M`:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: sigma, tau = G.gens()
sage: sigma
(1,2,3)(4,5)
sage: M.with_permuted_rows(sigma)
[0 2 0 0 0]
[0 0 3 0 0]
[1 0 0 0 0]
[0 0 0 0 5]
[0 0 0 4 0]
```

with_permuted_rows_and_columns (*row_permutation, column_permutation*)

Return the matrix obtained from permuting the rows and columns of `self` by applying the permutation group elements `row_permutation` and `column_permutation`.

As a permutation group element acts on integers $\{1, \ldots, n\}$ the rows are considered as being numbered from 1 for this operation.

INPUT:

- `row_permutation` – a `PermutationGroupElement`
- `column_permutation` – a `PermutationGroupElement`

OUTPUT:

•A matrix.

EXAMPLES: We create a matrix:

```
sage: M =␣
↪matrix(ZZ,[[1,0,0,0,0],[0,2,0,0,0],[0,0,3,0,0],[0,0,0,4,0],[0,0,0,0,5]])
sage: M
[1 0 0 0 0]
[0 2 0 0 0]
[0 0 3 0 0]
[0 0 0 4 0]
[0 0 0 0 5]
```

Next of all, create a permutation group element and act on `M`:

```
sage: G = PermutationGroup(['(1,2,3)(4,5)', '(1,2,3,4,5)'])
sage: sigma, tau = G.gens()
sage: sigma
(1,2,3)(4,5)
sage: M.with_permuted_rows_and_columns(sigma,tau)
[2 0 0 0 0]
[0 3 0 0 0]
[0 0 0 0 1]
[0 0 0 5 0]
[0 0 4 0 0]
```

with_rescaled_col (*i*, *s*, *start_row=0*)

Replaces i-th col of self by s times i-th col of self, returning new matrix.

EXAMPLES: We rescale the last column of a matrix over the integers:

```
sage: a = matrix(ZZ,2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: b = a.with_rescaled_col(2,-2); b
[  0   1  -4]
[  3   4 -10]
```

The original matrix is unchanged:

```
sage: a
[0 1 2]
[3 4 5]
```

Adding a rational multiple is okay, and reassigning a variable is okay:

```
sage: a = a.with_rescaled_col(1,1/3); a
[  0 1/3   2]
[  3 4/3   5]
```

with_rescaled_row (*i*, *s*, *start_col=0*)

Replaces i-th row of self by s times i-th row of self, returning new matrix.

EXAMPLES: We rescale the second row of a matrix over the integers:

```
sage: a = matrix(ZZ,3,2,range(6)); a
[0 1]
[2 3]
[4 5]
```

```
sage: b = a.with_rescaled_row(1,-2); b
[ 0  1]
[-4 -6]
[ 4  5]
```

The original matrix is unchanged:

```
sage: a
[0 1]
[2 3]
[4 5]
```

Adding a rational multiple is okay, and reassigning a variable is okay:

```
sage: a = a.with_rescaled_row(2,1/3); a
[  0   1]
[  2   3]
[4/3 5/3]
```

with_row_set_to_multiple_of_row (*i*, *j*, *s*)

Set row i equal to s times row j, returning a new matrix.

EXAMPLES: We change the second row to -3 times the first row:

```
sage: a = matrix(ZZ,2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: b = a.with_row_set_to_multiple_of_row(1,0,-3); b
[ 0  1  2]
[ 0 -3 -6]
```

Note that the original matrix is unchanged:

```
sage: a
[0 1 2]
[3 4 5]
```

Adding a rational multiple is okay, and reassigning a variable is okay:

```
sage: a = a.with_row_set_to_multiple_of_row(1,0,1/2); a
[  0   1   2]
[  0 1/2   1]
```

with_swapped_columns (*c1*, *c2*)

Swap columns `c1` and `c2` of `self` and return a new matrix.

INPUT:

- `c1`, `c2` - integers specifying columns of `self` to interchange

OUTPUT:

A new matrix, identical to `self` except that columns `c1` and `c2` are swapped.

EXAMPLES:

Remember that columns are numbered starting from zero.

```
sage: A = matrix(QQ, 4, range(20))
sage: A.with_swapped_columns(1, 2)
```

```
[ 0  2  1  3  4]
[ 5  7  6  8  9]
[10 12 11 13 14]
[15 17 16 18 19]
```

Trying to swap a column with itself will succeed, but still return a new matrix.

```
sage: A = matrix(QQ, 4, range(20))
sage: B = A.with_swapped_columns(2, 2)
sage: A == B
True
sage: A is B
False
```

The column specifications are checked.

```
sage: A = matrix(4, range(20))
sage: A.with_swapped_columns(-1, 2)
Traceback (most recent call last):
...
IndexError: matrix column index out of range

sage: A.with_swapped_columns(2, 5)
Traceback (most recent call last):
...
IndexError: matrix column index out of range
```

with_swapped_rows (*r1*, *r2*)

Swap rows `r1` and `r2` of `self` and return a new matrix.

INPUT:

- `r1`, `r2` - integers specifying rows of `self` to interchange

OUTPUT:

A new matrix, identical to `self` except that rows `r1` and `r2` are swapped.

EXAMPLES:

Remember that rows are numbered starting from zero.

```
sage: A = matrix(QQ, 4, range(20))
sage: A.with_swapped_rows(1, 2)
[ 0  1  2  3  4]
[10 11 12 13 14]
[ 5  6  7  8  9]
[15 16 17 18 19]
```

Trying to swap a row with itself will succeed, but still return a new matrix.

```
sage: A = matrix(QQ, 4, range(20))
sage: B = A.with_swapped_rows(2, 2)
sage: A == B
True
sage: A is B
False
```

The row specifications are checked.

```
sage: A = matrix(4, range(20))
sage: A.with_swapped_rows(-1, 2)
Traceback (most recent call last):
...
IndexError: matrix row index out of range

sage: A.with_swapped_rows(2, 5)
Traceback (most recent call last):
...
IndexError: matrix row index out of range
```

sage.matrix.matrix0.**set_max_cols**(*n*)

Sets the global variable max_cols (which is used in deciding how to output a matrix).

EXAMPLES:

```
sage: from sage.matrix.matrix0 import set_max_cols
sage: set_max_cols(50)
```

sage.matrix.matrix0.**set_max_rows**(*n*)

Sets the global variable max_rows (which is used in deciding how to output a matrix).

EXAMPLES:

```
sage: from sage.matrix.matrix0 import set_max_rows
sage: set_max_rows(20)
```

sage.matrix.matrix0.**unpickle**(*cls*, *parent*, *immutability*, *cache*, *data*, *version*)

Unpickle a matrix. This is only used internally by Sage. Users should never call this function directly.

EXAMPLES: We illustrating saving and loading several different types of matrices.

OVER **Z**:

```
sage: A = matrix(ZZ,2,range(4))
sage: loads(dumps(A)) # indirect doctest
[0 1]
[2 3]
```

Sparse OVER **Q**:

Dense over $\mathbf{Q}[x, y]$:

Dense over finite field.

CHAPTER

SEVEN

BASE CLASS FOR MATRICES, PART 1

For design documentation see `sage.matrix.docs` .

class sage.matrix.matrix1. **Matrix**

Bases: `sage.matrix.matrix0.Matrix`

augment (*right*, *subdivide=False*)

Returns a new matrix formed by appending the matrix (or vector) `right` on the right side of `self` .

INPUT:

- `right` - a matrix, vector or free module element, whose dimensions are compatible with `self` .
- `subdivide` - default: `False` - request the resulting matrix to have a new subdivision, separating `self` from `right` .

OUTPUT:

A new matrix formed by appending `right` onto the right side of `self` . If `right` is a vector (or free module element) then in this context it is appropriate to consider it as a column vector. (The code first converts a vector to a 1-column matrix.)

If `subdivide` is `True` then any column subdivisions for the two matrices are preserved, and a new subdivision is added between `self` and `right` . If the row divisions are identical, then they are preserved, otherwise they are discarded. When `subdivide` is `False` there is no subdivision information in the result.

> **Warning:** If `subdivide` is `True` then unequal row subdivisions will be discarded, since it would be ambiguous how to interpret them. If the subdivision behavior is not what you need, you can manage subdivisions yourself with methods like `get_subdivisions()` and `subdivide()` . You might also find `block_matrix()` or `block_diagonal_matrix()` useful and simpler in some instances.

EXAMPLES:

Augmenting with a matrix.

```
sage: A = matrix(QQ, 3, range(12))
sage: B = matrix(QQ, 3, range(9))
sage: A.augment(B)
[ 0  1  2  3  0  1  2]
[ 4  5  6  7  3  4  5]
[ 8  9 10 11  6  7  8]
```

Augmenting with a vector.

```
sage: A = matrix(QQ, 2, [0, 2, 4, 6, 8, 10])
sage: v = vector(QQ, 2, [100, 200])
sage: A.augment(v)
[  0   2   4 100]
[  6   8  10 200]
```

Errors are raised if the sizes are incompatible.

```
sage: A = matrix(RR, [[1, 2],[3, 4]])
sage: B = matrix(RR, [[10, 20], [30, 40], [50, 60]])
sage: A.augment(B)
Traceback (most recent call last):
...
TypeError: number of rows must be the same, 2 != 3

sage: v = vector(RR, [100, 200, 300])
sage: A.augment(v)
Traceback (most recent call last):
...
TypeError: number of rows must be the same, 2 != 3
```

Setting `subdivide` to `True` will, in its simplest form, add a subdivision between `self` and `right`.

```
sage: A = matrix(QQ, 3, range(12))
sage: B = matrix(QQ, 3, range(15))
sage: A.augment(B, subdivide=True)
[ 0  1  2  3| 0  1  2  3  4]
[ 4  5  6  7| 5  6  7  8  9]
[ 8  9 10 11|10 11 12 13 14]
```

Column subdivisions are preserved by augmentation, and enriched, if subdivisions are requested. (So multiple augmentations can be recorded.)

```
sage: A = matrix(QQ, 3, range(6))
sage: A.subdivide(None, [1])
sage: B = matrix(QQ, 3, range(9))
sage: B.subdivide(None, [2])
sage: A.augment(B, subdivide=True)
[0|1|0 1|2]
[2|3|3 4|5]
[4|5|6 7|8]
```

Row subdivisions can be preserved, but only if they are identical. Otherwise, this information is discarded and must be managed separately.

```
sage: A = matrix(QQ, 3, range(6))
sage: A.subdivide([1,3], None)
sage: B = matrix(QQ, 3, range(9))
sage: B.subdivide([1,3], None)
sage: A.augment(B, subdivide=True)
[0 1|0 1 2]
[---+-----]
[2 3|3 4 5]
[4 5|6 7 8]
[---+-----]

sage: A.subdivide([1,2], None)
sage: A.augment(B, subdivide=True)
```

```
[0 1|0 1 2]
[2 3|3 4 5]
[4 5|6 7 8]
```

The result retains the base ring of `self` by coercing the elements of `right` into the base ring of `self`.

```
sage: A = matrix(QQ, 2, [1,2])
sage: B = matrix(RR, 2, [sin(1.1), sin(2.2)])
sage: C = A.augment(B); C
[                  1 183017397/205358938]
[                  2 106580492/131825561]
sage: C.parent()
Full MatrixSpace of 2 by 2 dense matrices over Rational Field

sage: D = B.augment(A); D
[0.89120736006...  1.00000000000000]
[0.80849640381...  2.00000000000000]
sage: D.parent()
Full MatrixSpace of 2 by 2 dense matrices over Real Field with 53 bits of␣
↪precision
```

Sometimes it is not possible to coerce into the base ring of `self`. A solution is to change the base ring of `self` to a more expansive ring. Here we mix the rationals with a ring of polynomials with rational coefficients.

```
sage: R = PolynomialRing(QQ, 'y')
sage: A = matrix(QQ, 1, [1,2])
sage: B = matrix(R, 1, ['y', 'y^2'])

sage: C = B.augment(A); C
[  y y^2   1   2]
sage: C.parent()
Full MatrixSpace of 1 by 4 dense matrices over Univariate Polynomial Ring in␣
↪y over Rational Field

sage: D = A.augment(B)
Traceback (most recent call last):
...
TypeError: not a constant polynomial

sage: E = A.change_ring(R)
sage: F = E.augment(B); F
[  1   2   y y^2]
sage: F.parent()
Full MatrixSpace of 1 by 4 dense matrices over Univariate Polynomial Ring in␣
↪y over Rational Field
```

AUTHORS:

- Naqi Jaffery (2006-01-24): examples
- Rob Beezer (2010-12-07): vector argument, docstring, subdivisions

block_sum (*other*)

Return the block matrix that has self and other on the diagonal:

```
[ self     0 ]
[    0 other ]
```

EXAMPLES:

```
sage: A = matrix(QQ[['t']], 2, range(1, 5))
sage: A.block_sum(100*A)
[  1   2   0   0]
[  3   4   0   0]
[  0   0 100 200]
[  0   0 300 400]
```

column (*i*, *from_list=False*)

Return the `i` 'th column of this matrix as a vector.

This column is a dense vector if and only if the matrix is a dense matrix.

INPUT:

- `i` - integer
- `from_list` - bool (default: False); if true, returns the `i` 'th element of `self.columns()` (see *columns()*), which may be faster, but requires building a list of all columns the first time it is called after an entry of the matrix is changed.

EXAMPLES:

```
sage: a = matrix(2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: a.column(1)
(1, 4)
```

If the column is negative, it wraps around, just like with list indexing, e.g., -1 gives the right-most column:

```
sage: a.column(-1)
(2, 5)
```

columns (*copy=True*)

Return a list of the columns of self.

INPUT:

- `copy` - (default: True) if True, return a copy of the list of columns which is safe to change.

If `self` is a sparse matrix, columns are returned as sparse vectors, otherwise returned vectors are dense.

EXAMPLES:

```
sage: matrix(3, [1..9]).columns()
[(1, 4, 7), (2, 5, 8), (3, 6, 9)]
sage: matrix(RR, 2, [sqrt(2), pi, exp(1), 0]).columns()
[(1.41421356237310, 2.71828182845905), (3.14159265358979, 0.000000000000000)]
sage: matrix(RR, 0, 2, []).columns()
[(), ()]
sage: matrix(RR, 2, 0, []).columns()
[]
sage: m = matrix(RR, 3, 3, {(1,2): pi, (2, 2): -1, (0,1): sqrt(2)})
sage: parent(m.columns()[0])
Sparse vector space of dimension 3 over Real Field with 53 bits of precision
```

Sparse matrices produce sparse columns.

```
sage: A = matrix(QQ, 2, range(4), sparse=True)
sage: v = A.columns()[0]
sage: v.is_sparse()
True
```

delete_columns (*dcols*, *check=True*)

Return the matrix constructed from deleting the columns with indices in the `dcols` list.

INPUT:

- `dcols` - list of indices of columns to be deleted from self.
- `check` - checks whether any index in `dcols` is out of range. Defaults to `True`.

SEE ALSO: The methods *delete_rows()* and *matrix_from_columns()* are related.

EXAMPLES:

```
sage: A = Matrix(3,4,range(12)); A
[ 0  1  2  3]
[ 4  5  6  7]
[ 8  9 10 11]
sage: A.delete_columns([0,2])
[ 1  3]
[ 5  7]
[ 9 11]
```

`dcols` can be a tuple. But only the underlying set of indices matters.

```
sage: A.delete_columns((2,0,2))
[ 1  3]
[ 5  7]
[ 9 11]
```

The default is to check whether any index in `dcols` is out of range.

```
sage: A.delete_columns([-1,2,4])
Traceback (most recent call last):
...
IndexError: [4, -1] contains invalid indices.
sage: A.delete_columns([-1,2,4], check=False)
[ 0  1  3]
[ 4  5  7]
[ 8  9 11]
```

AUTHORS:

- Wai Yan Pong (2012-03-05)

delete_rows (*drows*, *check=True*)

Return the matrix constructed from deleting the rows with indices in the `drows` list.

INPUT:

- `drows` - list of indices of rows to be deleted from self.
- `check` - checks whether any index in `drows` is out of range. Defaults to `True`.

SEE ALSO: The methods *delete_columns()* and *matrix_from_rows()* are related.

EXAMPLES:

```
sage: A = Matrix(4,3,range(12)); A
[ 0  1  2]
[ 3  4  5]
[ 6  7  8]
[ 9 10 11]
sage: A.delete_rows([0,2])
[ 3  4  5]
[ 9 10 11]
```

`drows` can be a tuple. But only the underlying set of indices matters.

```
sage: A.delete_rows((2,0,2))
[ 3  4  5]
[ 9 10 11]
```

The default is to check whether the any index in `drows` is out of range.

```
sage: A.delete_rows([-1,2,4])
Traceback (most recent call last):
...
IndexError: [4, -1] contains invalid indices.
sage: A.delete_rows([-1,2,4], check=False)
[ 0  1  2]
[ 3  4  5]
[ 9 10 11]
```

AUTHORS:

- Wai Yan Pong (2012-03-05)

dense_columns (*copy=True*)

Return list of the dense columns of self.

INPUT:

- `copy` - (default: True) if True, return a copy so you can modify it safely

EXAMPLES:

An example over the integers:

```
sage: a = matrix(3,3,range(9)); a
[0 1 2]
[3 4 5]
[6 7 8]
sage: a.dense_columns()
[(0, 3, 6), (1, 4, 7), (2, 5, 8)]
```

We do an example over a polynomial ring:

```
sage: R.<x> = QQ[ ]
sage: a = matrix(R, 2, [x,x^2, 2/3*x,1+x^5]); a
[      x     x^2]
[  2/3*x x^5 + 1]
sage: a.dense_columns()
[(x, 2/3*x), (x^2, x^5 + 1)]
sage: a = matrix(R, 2, [x,x^2, 2/3*x,1+x^5], sparse=True)
```

```
sage: c = a.dense_columns(); c
[(x, 2/3*x), (x^2, x^5 + 1)]
sage: parent(c[1])
Ambient free module of rank 2 over the principal ideal domain Univariate␣
↪Polynomial Ring in x over Rational Field
```

dense_matrix ( )

If this matrix is sparse, return a dense matrix with the same entries. If this matrix is dense, return this matrix (not a copy).

Note: The definition of "dense" and "sparse" in Sage have nothing to do with the number of nonzero entries. Sparse and dense are properties of the underlying representation of the matrix.

EXAMPLES:

```
sage: A = MatrixSpace(QQ,2, sparse=True)([1,2,0,1])
sage: A.is_sparse()
True
sage: B = A.dense_matrix()
sage: B.is_sparse()
False
sage: A*B
[1 4]
[0 1]
sage: A.parent()
Full MatrixSpace of 2 by 2 sparse matrices over Rational Field
sage: B.parent()
Full MatrixSpace of 2 by 2 dense matrices over Rational Field
```

In Sage, the product of a sparse and a dense matrix is always dense:

```
sage: (A*B).parent()
Full MatrixSpace of 2 by 2 dense matrices over Rational Field
sage: (B*A).parent()
Full MatrixSpace of 2 by 2 dense matrices over Rational Field
```

dense_rows (*copy=True*)

Return list of the dense rows of self.

INPUT:

- `copy` - (default: True) if True, return a copy so you can modify it safely (note that the individual vectors in the copy should not be modified since they are mutable!)

EXAMPLES:

```
sage: m = matrix(3, range(9)); m
[0 1 2]
[3 4 5]
[6 7 8]
sage: v = m.dense_rows(); v
[(0, 1, 2), (3, 4, 5), (6, 7, 8)]
sage: v is m.dense_rows()
False
sage: m.dense_rows(copy=False) is m.dense_rows(copy=False)
True
sage: m[0,0] = 10
```

```
sage: m.dense_rows()
[(10, 1, 2), (3, 4, 5), (6, 7, 8)]
```

lift ( )

Return lift of self to the covering ring of the base ring R, which is by definition the ring returned by calling cover_ring() on R, or just R itself if the cover_ring method is not defined.

EXAMPLES:

```
sage: M = Matrix(Integers(7), 2, 2, [5, 9, 13, 15]) ; M
[5 2]
[6 1]
sage: M.lift()
[5 2]
[6 1]
sage: parent(M.lift())
Full MatrixSpace of 2 by 2 dense matrices over Integer Ring
```

The field QQ doesn't have a cover_ring method:

```
sage: hasattr(QQ, 'cover_ring')
False
```

So lifting a matrix over QQ gives back the same exact matrix.

```
sage: B = matrix(QQ, 2, [1..4])
sage: B.lift()
[1 2]
[3 4]
sage: B.lift() is B
True
```

lift_centered ( )

Apply the lift_centered method to every entry of self.

OUTPUT:

If self is a matrix over the Integers mod n, this method returns the unique matrix m such that m is congruent to self mod n and for every entry $m[i,j]$ we have $-n/2 < m[i,j] \leq n/2$. If the coefficient ring does not have a cover_ring method, return self.

EXAMPLES:

```
sage: M = Matrix(Integers(8), 2, 4, range(8)) ; M
[0 1 2 3]
[4 5 6 7]
sage: L = M.lift_centered(); L
[ 0  1  2  3]
[ 4 -3 -2 -1]
sage: parent(L)
Full MatrixSpace of 2 by 4 dense matrices over Integer Ring
```

The returned matrix is congruent to M modulo 8.:

```
sage: L.mod(8)
[0 1 2 3]
[4 5 6 7]
```

The field QQ doesn't have a cover_ring method:

```
sage: hasattr(QQ, 'cover_ring')
False
```

So lifting a matrix over QQ gives back the same exact matrix.

```
sage: B = matrix(QQ, 2, [1..4])
sage: B.lift_centered()
[1 2]
[3 4]
sage: B.lift_centered() is B
True
```

matrix_from_columns (*columns*)

Return the matrix constructed from self using columns with indices in the columns list.

EXAMPLES:

```
sage: M = MatrixSpace(Integers(8),3,3)
sage: A = M(range(9)); A
[0 1 2]
[3 4 5]
[6 7 0]
sage: A.matrix_from_columns([2,1])
[2 1]
[5 4]
[0 7]
```

matrix_from_rows (*rows*)

Return the matrix constructed from self using rows with indices in the rows list.

EXAMPLES:

```
sage: M = MatrixSpace(Integers(8),3,3)
sage: A = M(range(9)); A
[0 1 2]
[3 4 5]
[6 7 0]
sage: A.matrix_from_rows([2,1])
[6 7 0]
[3 4 5]
```

matrix_from_rows_and_columns (*rows*, *columns*)

Return the matrix constructed from self from the given rows and columns.

EXAMPLES:

```
sage: M = MatrixSpace(Integers(8),3,3)
sage: A = M(range(9)); A
[0 1 2]
[3 4 5]
[6 7 0]
sage: A.matrix_from_rows_and_columns([1], [0,2])
[3 5]
sage: A.matrix_from_rows_and_columns([1,2], [1,2])
[4 5]
[7 0]
```

Note that row and column indices can be reordered or repeated:

```
sage: A.matrix_from_rows_and_columns([2,1], [2,1])
[0 7]
[5 4]
```

For example here we take from row 1 columns 2 then 0 twice, and do this 3 times.

```
sage: A.matrix_from_rows_and_columns([1,1,1],[2,0,0])
[5 3 3]
[5 3 3]
[5 3 3]
```

AUTHORS:

- Jaap Spies (2006-02-18)
- Didier Deshommes: some Pyrex speedups implemented

matrix_over_field ()

Return copy of this matrix, but with entries viewed as elements of the fraction field of the base ring (assuming it is defined).

EXAMPLES:

```
sage: A = MatrixSpace(IntegerRing(),2)([1,2,3,4])
sage: B = A.matrix_over_field()
sage: B
[1 2]
[3 4]
sage: B.parent()
Full MatrixSpace of 2 by 2 dense matrices over Rational Field
```

matrix_space (*nrows=None*, *ncols=None*, *sparse=None*)

Return the ambient matrix space of self.

INPUT:

- `nrows`, `ncols` - (optional) number of rows and columns in returned matrix space.
- `sparse` - whether the returned matrix space uses sparse or dense matrices.

EXAMPLES:

```
sage: m = matrix(3, [1..9])
sage: m.matrix_space()
Full MatrixSpace of 3 by 3 dense matrices over Integer Ring
sage: m.matrix_space(ncols=2)
Full MatrixSpace of 3 by 2 dense matrices over Integer Ring
sage: m.matrix_space(1)
Full MatrixSpace of 1 by 3 dense matrices over Integer Ring
sage: m.matrix_space(1, 2, True)
Full MatrixSpace of 1 by 2 sparse matrices over Integer Ring
```

new_matrix (*nrows=None*, *ncols=None*, *entries=None*, *coerce=True*, *copy=True*, *sparse=None*)

Create a matrix in the parent of this matrix with the given number of rows, columns, etc. The default parameters are the same as for self.

INPUT:

These three variables get sent to `matrix_space()`:

- `nrows`, `ncols` - number of rows and columns in returned matrix. If not specified, defaults to `None` and will give a matrix of the same size as self.
- `sparse` - whether returned matrix is sparse or not. Defaults to same value as self.

The remaining three variables (`coerce`, `entries`, and `copy`) are used by `sage.matrix.matrix_space.MatrixSpace()` to construct the new matrix.

> **Warning:** This function called with no arguments returns the zero matrix of the same dimension and sparseness of self.

EXAMPLES:

```
sage: A = matrix(ZZ,2,2,[1,2,3,4]); A
[1 2]
[3 4]
sage: A.new_matrix()
[0 0]
[0 0]
sage: A.new_matrix(1,1)
[0]
sage: A.new_matrix(3,3).parent()
Full MatrixSpace of 3 by 3 dense matrices over Integer Ring
```

```
sage: A = matrix(RR,2,3,[1.1,2.2,3.3,4.4,5.5,6.6]); A
[1.10000000000000 2.20000000000000 3.30000000000000]
[4.40000000000000 5.50000000000000 6.60000000000000]
sage: A.new_matrix()
[0.000000000000000 0.000000000000000 0.000000000000000]
[0.000000000000000 0.000000000000000 0.000000000000000]
sage: A.new_matrix().parent()
Full MatrixSpace of 2 by 3 dense matrices over Real Field with 53 bits of
↪precision
```

numpy (*dtype=None*)

Return the Numpy matrix associated to this matrix.

INPUT:

- `dtype` - The desired data-type for the array. If not given, then the type will be determined as the minimum type required to hold the objects in the sequence.

EXAMPLES:

```
sage: a = matrix(3,range(12))
sage: a.numpy()
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
sage: a.numpy('f')
array([[  0.,   1.,   2.,   3.],
       [  4.,   5.,   6.,   7.],
       [  8.,   9.,  10.,  11.]], dtype=float32)
sage: a.numpy('d')
array([[  0.,   1.,   2.,   3.],
       [  4.,   5.,   6.,   7.],
       [  8.,   9.,  10.,  11.]])
sage: a.numpy('B')
```

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]], dtype=uint8)
```

Type `numpy.typecodes` for a list of the possible typecodes:

```
sage: import numpy
sage: sorted(numpy.typecodes.items())
[('All', '?bhilqpBHILQPefdgFDGSUVOMm'), ('AllFloat', 'efdgFDG'), ('AllInteger
↪', 'bBhHiIlLqQpP'), ('Character', 'c'), ('Complex', 'FDG'), ('Datetime', 'Mm
↪'), ('Float', 'efdg'), ('Integer', 'bhilqp'), ('UnsignedInteger', 'BHILQP')]
```

Alternatively, numpy automatically calls this function (via the magic `__array__()` method) to convert Sage matrices to numpy arrays:

```
sage: import numpy
sage: b=numpy.array(a); b
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
sage: b.dtype
dtype('int32')  # 32-bit
dtype('int64')  # 64-bit
sage: b.shape
(3, 4)
```

row (*i*, *from_list=False*)

Return the `i` 'th row of this matrix as a vector.

This row is a dense vector if and only if the matrix is a dense matrix.

INPUT:

- `i` - integer
- `from_list` - bool (default: False); if true, returns the `i` 'th element of `self.rows()` (see *rows()*), which may be faster, but requires building a list of all rows the first time it is called after an entry of the matrix is changed.

EXAMPLES:

```
sage: a = matrix(2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: a.row(0)
(0, 1, 2)
sage: a.row(1)
(3, 4, 5)
sage: a.row(-1)  # last row
(3, 4, 5)
```

rows (*copy=True*)

Return a list of the rows of self.

INPUT:

- `copy` - (default: True) if True, return a copy of the list of rows which is safe to change.

If `self` is a sparse matrix, rows are returned as sparse vectors, otherwise returned vectors are dense.

EXAMPLES:

```
sage: matrix(3, [1..9]).rows()
[(1, 2, 3), (4, 5, 6), (7, 8, 9)]
sage: matrix(RR, 2, [sqrt(2), pi, exp(1), 0]).rows()
[(1.41421356237310, 3.14159265358979), (2.71828182845905, 0.000000000000000)]
sage: matrix(RR, 0, 2, []).rows()
[]
sage: matrix(RR, 2, 0, []).rows()
[(), ()]
sage: m = matrix(RR, 3, 3, {(1,2): pi, (2, 2): -1, (0,1): sqrt(2)})
sage: parent(m.rows()[0])
Sparse vector space of dimension 3 over Real Field with 53 bits of precision
```

Sparse matrices produce sparse rows.

```
sage: A = matrix(QQ, 2, range(4), sparse=True)
sage: v = A.rows()[0]
sage: v.is_sparse()
True
```

set_column (*col*, *v*)

Sets the entries of column `col` to the entries of `v` .

INPUT:

- `col` - index of column to be set.
- `v` - a list or vector of the new entries.

OUTPUT:

Changes the matrix in-place, so there is no output.

EXAMPLES:

New entries may be contained in a vector.:

```
sage: A = matrix(QQ, 5, range(25))
sage: u = vector(QQ, [0, -1, -2, -3, -4])
sage: A.set_column(2, u)
sage: A
[ 0  1  0  3  4]
[ 5  6 -1  8  9]
[10 11 -2 13 14]
[15 16 -3 18 19]
[20 21 -4 23 24]
```

New entries may be in any sort of list.:

```
sage: A = matrix([[1, 2], [3, 4]]); A
[1 2]
[3 4]
sage: A.set_column(0, [0, 0]); A
[0 2]
[0 4]
sage: A.set_column(1, (0, 0)); A
[0 0]
[0 0]
```

set_row (*row*, *v*)

Sets the entries of row `row` to the entries of `v` .

INPUT:

- `row` - index of row to be set.
- `v` - a list or vector of the new entries.

OUTPUT:

Changes the matrix in-place, so there is no output.

EXAMPLES:

New entries may be contained in a vector.:

```
sage: A = matrix(QQ, 5, range(25))
sage: u = vector(QQ, [0, -1, -2, -3, -4])
sage: A.set_row(2, u)
sage: A
[ 0  1  2  3  4]
[ 5  6  7  8  9]
[ 0 -1 -2 -3 -4]
[15 16 17 18 19]
[20 21 22 23 24]
```

New entries may be in any sort of list.:

```
sage: A = matrix([[1, 2], [3, 4]]); A
[1 2]
[3 4]
sage: A.set_row(0, [0, 0]); A
[0 0]
[3 4]
sage: A.set_row(1, (0, 0)); A
[0 0]
[0 0]
```

sparse_columns (*copy=True*)

Return a list of the columns of `self` as sparse vectors (or free module elements).

INPUT:

- **copy - (default: True) if True, return a copy so you can** modify it safely

EXAMPLES:

```
sage: a = matrix(2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: v = a.sparse_columns(); v
[(0, 3), (1, 4), (2, 5)]
sage: v[1].is_sparse()
True
```

sparse_matrix ()

If this matrix is dense, return a sparse matrix with the same entries. If this matrix is sparse, return this matrix (not a copy).

Note: The definition of “dense” and “sparse” in Sage have nothing to do with the number of nonzero entries. Sparse and dense are properties of the underlying representation of the matrix.

EXAMPLES:

```
sage: A = MatrixSpace(QQ,2, sparse=False)([1,2,0,1])
sage: A.is_sparse()
False
sage: B = A.sparse_matrix()
sage: B.is_sparse()
True
sage: A
[1 2]
[0 1]
sage: B
[1 2]
[0 1]
sage: A*B
[1 4]
[0 1]
sage: A.parent()
Full MatrixSpace of 2 by 2 dense matrices over Rational Field
sage: B.parent()
Full MatrixSpace of 2 by 2 sparse matrices over Rational Field
sage: (A*B).parent()
Full MatrixSpace of 2 by 2 dense matrices over Rational Field
sage: (B*A).parent()
Full MatrixSpace of 2 by 2 dense matrices over Rational Field
```

sparse_rows (*copy=True*)

Return a list of the rows of `self` as sparse vectors (or free module elements).

INPUT:

- **copy - (default: True) if True, return a copy so you can** modify it safely

EXAMPLES:

```
sage: m = Mat(ZZ,3,3,sparse=True)(range(9)); m
[0 1 2]
[3 4 5]
[6 7 8]
sage: v = m.sparse_rows(); v
[(0, 1, 2), (3, 4, 5), (6, 7, 8)]
sage: m.sparse_rows(copy=False) is m.sparse_rows(copy=False)
True
sage: v[1].is_sparse()
True
sage: m[0,0] = 10
sage: m.sparse_rows()
[(10, 1, 2), (3, 4, 5), (6, 7, 8)]
```

stack (*bottom*, *subdivide=False*)

Return a new matrix formed by appending the matrix (or vector) `bottom` below `self`:

```
[  self   ]
[ bottom ]
```

INPUT:

- `bottom` - a matrix, vector or free module element, whose dimensions are compatible with `self`.

•`subdivide` - default: `False` - request the resulting matrix to have a new subdivision, separating `self` from `bottom`.

OUTPUT:

A new matrix formed by appending `bottom` beneath `self`. If `bottom` is a vector (or free module element) then in this context it is appropriate to consider it as a row vector. (The code first converts a vector to a 1-row matrix.)

If `subdivide` is `True` then any row subdivisions for the two matrices are preserved, and a new subdivision is added between `self` and `bottom`. If the column divisions are identical, then they are preserved, otherwise they are discarded. When `subdivide` is `False` there is no subdivision information in the result.

> **Warning:** If `subdivide` is `True` then unequal column subdivisions will be discarded, since it would be ambiguous how to interpret them. If the subdivision behavior is not what you need, you can manage subdivisions yourself with methods like *`subdivisions()`* and *`subdivide()`*. You might also find `block_matrix()` or `block_diagonal_matrix()` useful and simpler in some instances.

EXAMPLES:

Stacking with a matrix.

```
sage: A = matrix(QQ, 4, 3, range(12))
sage: B = matrix(QQ, 3, 3, range(9))
sage: A.stack(B)
[ 0  1  2]
[ 3  4  5]
[ 6  7  8]
[ 9 10 11]
[ 0  1  2]
[ 3  4  5]
[ 6  7  8]
```

Stacking with a vector.

```
sage: A = matrix(QQ, 3, 2, [0, 2, 4, 6, 8, 10])
sage: v = vector(QQ, 2, [100, 200])
sage: A.stack(v)
[  0   2]
[  4   6]
[  8  10]
[100 200]
```

Errors are raised if the sizes are incompatible.

```
sage: A = matrix(RR, [[1, 2],[3, 4]])
sage: B = matrix(RR, [[10, 20, 30], [40, 50, 60]])
sage: A.stack(B)
Traceback (most recent call last):
...
TypeError: number of columns must be the same, not 2 and 3

sage: v = vector(RR, [100, 200, 300])
sage: A.stack(v)
Traceback (most recent call last):
```

```
...
TypeError: number of columns must be the same, not 2 and 3
```

Setting `subdivide` to `True` will, in its simplest form, add a subdivision between `self` and `bottom`.

```
sage: A = matrix(QQ, 2, 5, range(10))
sage: B = matrix(QQ, 3, 5, range(15))
sage: A.stack(B, subdivide=True)
[ 0  1  2  3  4]
[ 5  6  7  8  9]
[--------------]
[ 0  1  2  3  4]
[ 5  6  7  8  9]
[10 11 12 13 14]
```

Row subdivisions are preserved by stacking, and enriched, if subdivisions are requested. (So multiple stackings can be recorded.)

```
sage: A = matrix(QQ, 2, 4, range(8))
sage: A.subdivide([1], None)
sage: B = matrix(QQ, 3, 4, range(12))
sage: B.subdivide([2], None)
sage: A.stack(B, subdivide=True)
[ 0  1  2  3]
[-----------]
[ 4  5  6  7]
[-----------]
[ 0  1  2  3]
[ 4  5  6  7]
[-----------]
[ 8  9 10 11]
```

Column subdivisions can be preserved, but only if they are identical. Otherwise, this information is discarded and must be managed separately.

```
sage: A = matrix(QQ, 2, 5, range(10))
sage: A.subdivide(None, [2,4])
sage: B = matrix(QQ, 3, 5, range(15))
sage: B.subdivide(None, [2,4])
sage: A.stack(B, subdivide=True)
[ 0  1| 2  3| 4]
[ 5  6| 7  8| 9]
[-----+-----+--]
[ 0  1| 2  3| 4]
[ 5  6| 7  8| 9]
[10 11|12 13|14]

sage: A.subdivide(None, [1,2])
sage: A.stack(B, subdivide=True)
[ 0  1  2  3  4]
[ 5  6  7  8  9]
[--------------]
[ 0  1  2  3  4]
[ 5  6  7  8  9]
[10 11 12 13 14]
```

The base ring of the result is the common parent for the base rings of `self` and `bottom`. In particular, the parent for `A.stack(B)` and `B.stack(A)` should be equal:

```
sage: A = matrix(QQ, 1, 2, [1,2])
sage: B = matrix(RR, 1, 2, [sin(1.1), sin(2.2)])
sage: C = A.stack(B); C
[ 1.00000000000000  2.00000000000000]
[0.891207360061435 0.808496403819590]
sage: C.parent()
Full MatrixSpace of 2 by 2 dense matrices over Real Field with 53 bits of␣
↪precision

sage: D = B.stack(A); D
[0.891207360061435 0.808496403819590]
[ 1.00000000000000  2.00000000000000]
sage: D.parent()
Full MatrixSpace of 2 by 2 dense matrices over Real Field with 53 bits of␣
↪precision
```

```
sage: R.<y> = PolynomialRing(ZZ)
sage: A = matrix(QQ, 1, 2, [1, 2/3])
sage: B = matrix(R, 1, 2, [y, y^2])

sage: C = A.stack(B); C
[  1 2/3]
[  y y^2]
sage: C.parent()
Full MatrixSpace of 2 by 2 dense matrices over Univariate Polynomial Ring in␣
↪y over Rational Field
```

Stacking a dense matrix atop a sparse one returns a sparse matrix:

```
sage: M = Matrix(ZZ, 2, 3, range(6), sparse=False)
sage: N = diagonal_matrix([10,11,12], sparse=True)
sage: P = M.stack(N); P
[ 0  1  2]
[ 3  4  5]
[10  0  0]
[ 0 11  0]
[ 0  0 12]
sage: P.is_sparse()
True
sage: P = N.stack(M); P
[10  0  0]
[ 0 11  0]
[ 0  0 12]
[ 0  1  2]
[ 3  4  5]
sage: P.is_sparse()
True
```

One can stack matrices over different rings (trac ticket #16399).

```
sage: M = Matrix(ZZ, 2, 3, range(6))
sage: N = Matrix(QQ, 1, 3, [10,11,12])
sage: M.stack(N)
[ 0  1  2]
[ 3  4  5]
[10 11 12]
```

```
sage: N.stack(M)
[10 11 12]
[ 0  1  2]
[ 3  4  5]
```

AUTHORS:

- Rob Beezer (2011-03-19): rewritten to mirror code for `augment()`
- Jeroen Demeyer (2015-01-06): refactor, see trac ticket #16399. Put all boilerplate in one place (here) and put the actual type-dependent implementation in `_stack_impl`.

submatrix (*row=0*, *col=0*, *nrows=-1*, *ncols=-1*)

Return the matrix constructed from self using the specified range of rows and columns.

INPUT:

- `row`, `col` - index of the starting row and column. Indices start at zero.
- `nrows`, `ncols` - (optional) number of rows and columns to take. If not provided, take all rows below and all columns to the right of the starting entry.

SEE ALSO:

The functions `matrix_from_rows()`, `matrix_from_columns()`, and `matrix_from_rows_and_columns()` allow one to select arbitrary subsets of rows and/or columns.

EXAMPLES:

Take the 3×3 submatrix starting from entry (1,1) in a 4×4 matrix:

```
sage: m = matrix(4, [1..16])
sage: m.submatrix(1, 1)
[ 6  7  8]
[10 11 12]
[14 15 16]
```

Same thing, except take only two rows:

```
sage: m.submatrix(1, 1, 2)
[ 6  7  8]
[10 11 12]
```

And now take only one column:

```
sage: m.submatrix(1, 1, 2, 1)
[ 6]
[10]
```

You can take zero rows or columns if you want:

```
sage: m.submatrix(1, 1, 0)
[]
sage: parent(m.submatrix(1, 1, 0))
Full MatrixSpace of 0 by 3 dense matrices over Integer Ring
```

CHAPTER

EIGHT

BASE CLASS FOR MATRICES, PART 2

For design documentation see matrix/docs.py.

AUTHORS:

- William Stein: initial version
- Miguel Marco (2010-06-19): modified eigenvalues and eigenvectors functions to allow the option extend=False
- Rob Beezer (2011-02-05): refactored all of the matrix kernel routines

class `sage.matrix.matrix2.`**`Matrix`**

Bases: *`sage.matrix.matrix1.Matrix`*

`C`

Returns the conjugate matrix.

EXAMPLES:

```
sage: A = matrix(QQbar, [[     -3,  5 - 3*I, 7 - 4*I],
....:                    [7 + 3*I, -1 + 6*I, 3 + 5*I],
....:                    [3 + 3*I, -3 + 6*I, 5 +   I]])
sage: A.C
[      -3  5 + 3*I  7 + 4*I]
[ 7 - 3*I -1 - 6*I  3 - 5*I]
[ 3 - 3*I -3 - 6*I  5 - 1*I]
```

`H`

Returns the conjugate-transpose (Hermitian) matrix.

EXAMPLES:

```
sage: A = matrix(QQbar, [[     -3,  5 - 3*I, 7 - 4*I],
....:                    [7 + 3*I, -1 + 6*I, 3 + 5*I],
....:                    [3 + 3*I, -3 + 6*I, 5 +   I]])
sage: A.H
[      -3  7 - 3*I  3 - 3*I]
[ 5 + 3*I -1 - 6*I -3 - 6*I]
[ 7 + 4*I  3 - 5*I  5 - 1*I]
```

`I`

Returns the inverse of the matrix, if it exists.

EXAMPLES:

```
sage: A = matrix(QQ, [[-5, -3, -1, -7],
....:                 [ 1,  1,  1,  0],
....:                 [-1, -2, -2,  0],
```

```
....:                      [-2, -1,  0, -4]])
sage: A.I
doctest:...: DeprecationWarning: The I property on matrices has been␣
↪deprecated. Please use the inverse() method instead.
See http://trac.sagemath.org/20904 for details.
[ 0  2  1  0]
[-4 -8 -2  7]
[ 4  7  1 -7]
[ 1  1  0 -2]

sage: B = matrix(QQ, [[-11, -5, 18,  -6],
....:                 [  1,  2, -6,   8],
....:                 [ -4, -2,  7,  -3],
....:                 [  1, -2,  5, -11]])
sage: B.I
Traceback (most recent call last):
...
ZeroDivisionError: input matrix must be nonsingular
```

LU (*pivot=None*, *format='plu'*)

Finds a decomposition into a lower-triangular matrix and an upper-triangular matrix.

INPUT:

- `pivot` - pivoting strategy
  - 'auto' (default) - see if the matrix entries are ordered (i.e. if they have an absolute value method), and if so, use a the partial pivoting strategy. Otherwise, fall back to the nonzero strategy. This is the best choice for general routines that may call this for matrix entries of a variety of types.
  - 'partial' - each column is examined for the element with the largest absolute value and the row containing this element is swapped into place.
  - 'nonzero' - the first nonzero element in a column is located and the row with this element is used.
- `format` - contents of output, see more discussion below about output.
  - 'plu' (default) - a triple; matrices P, L and U such that A = P*L*U.
  - 'compact' - a pair; row permutation as a tuple, and the matrices L and U combined into one matrix.

OUTPUT:

Suppose that A is an $m \times n$ matrix, then an LU decomposition is a lower-triangular $m \times m$ matrix L with every diagonal element equal to 1, and an upper-triangular $m \times n$ matrix, U such that the product LU, after a permutation of the rows, is then equal to A. For the 'plu' format the permutation is returned as an $m \times m$ permutation matrix P such that

$$A = PLU$$

It is more common to place the permutation matrix just to the left of A. If you desire this version, then use the inverse of P which is computed most efficiently as its transpose.

If the 'partial' pivoting strategy is used, then the non-diagonal entries of L will be less than or equal to 1 in absolute value. The 'nonzero' pivot strategy may be faster, but the growth of data structures for elements of the decomposition might counteract the advantage.

By necessity, returned matrices have a base ring equal to the fraction field of the base ring of the original matrix.

In the 'compact' format, the first returned value is a tuple that is a permutation of the rows of LU that yields A. See the doctest for how you might employ this permutation. Then the matrices L and U are merged into one matrix – remove the diagonal of ones in L and the remaining nonzero entries can replace the entries of U beneath the diagonal.

The results are cached, only in the compact format, separately for each pivot strategy called. Repeated requests for the 'plu' format will require just a small amount of overhead in each call to bust out the compact format to the three matrices. Since only the compact format is cached, the components of the compact format are immutable, while the components of the 'plu' format are regenerated, and hence are mutable.

Notice that while U is similar to row-echelon form and the rows of U span the row space of A, the rows of U are not generally linearly independent. Nor are the pivot columns (or rank) immediately obvious. However for rings without specialized echelon form routines, this method is about twice as fast as the generic echelon form routine since it only acts "below the diagonal", as would be predicted from a theoretical analysis of the algorithms.

Note: This is an exact computation, so limited to exact rings. If you need numerical results, convert the base ring to the field of real double numbers, `RDF` or the field of complex double numbers, `CDF` , which will use a faster routine that is careful about numerical subtleties.

ALGORITHM:

"Gaussian Elimination with Partial Pivoting," Algorithm 21.1 of [TB1997].

EXAMPLES:

Notice the difference in the L matrix as a result of different pivoting strategies. With partial pivoting, every entry of L has absolute value 1 or less.

```
sage: A = matrix(QQ, [[1, -1,  0,  2,  4,  7, -1],
....:                 [2, -1,  0,  6,  4,  8, -2],
....:                 [2,  0,  1,  4,  2,  6,  0],
....:                 [1,  0, -1,  8, -1, -1, -3],
....:                 [1,  1,  2, -2, -1,  1,  3]])
sage: P, L, U = A.LU(pivot='partial')
sage: P
[0 0 0 0 1]
[1 0 0 0 0]
[0 0 0 1 0]
[0 0 1 0 0]
[0 1 0 0 0]
sage: L
[   1    0    0    0    0]
[ 1/2    1    0    0    0]
[ 1/2  1/3    1    0    0]
[   1  2/3  1/5    1    0]
[ 1/2 -1/3 -2/5    0    1]
sage: U
[    2    -1     0     6     4     8    -2]
[    0   3/2     2    -5    -3    -3     4]
[    0     0  -5/3  20/3    -2    -4 -10/3]
[    0     0     0     0   2/5   4/5     0]
[    0     0     0     0   1/5   2/5     0]
sage: A == P*L*U
True
sage: P, L, U = A.LU(pivot='nonzero')
sage: P
```

```
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
[0 0 0 0 1]
sage: L
[ 1  0  0  0  0]
[ 2  1  0  0  0]
[ 2  2  1  0  0]
[ 1  1 -1  1  0]
[ 1  2  2  0  1]
sage: U
[ 1 -1  0  2  4  7 -1]
[ 0  1  0  2 -4 -6  0]
[ 0  0  1 -4  2  4  2]
[ 0  0  0  0  1  2  0]
[ 0  0  0  0 -1 -2  0]
sage: A == P*L*U
True
```

An example of the compact format.

```
sage: B = matrix(QQ, [[ 1,  3,  5,  5],
....:                 [ 1,  4,  7,  8],
....:                 [-1, -4, -6, -6],
....:                 [ 0, -2, -5, -8],
....:                 [-2, -6, -6, -2]])
sage: perm, M = B.LU(format='compact')
sage: perm
(4, 3, 0, 1, 2)
sage: M
[  -2   -6   -6   -2]
[   0   -2   -5   -8]
[-1/2    0    2    4]
[-1/2 -1/2  3/4    0]
[ 1/2  1/2 -1/4    0]
```

We can easily illustrate the relationships between the two formats with a square matrix.

```
sage: C = matrix(QQ, [[-2,  3, -2, -5],
....:                 [ 1, -2,  1,  3],
....:                 [-4,  7, -3, -8],
....:                 [-3,  8, -1, -5]])
sage: P, L, U = C.LU(format='plu')
sage: perm, M = C.LU(format='compact')
sage: (L - identity_matrix(4)) + U == M
True
sage: p = [perm[i]+1 for i in range(len(perm))]
sage: PP = Permutation(p).to_matrix()
sage: PP == P
True
```

For a nonsingular matrix, and the 'nonzero' pivot strategy there is no need to permute rows, so the permutation matrix will be the identity. Furthermore, it can be shown that then the L and U matrices are uniquely determined by requiring L to have ones on the diagonal.

```
sage: D = matrix(QQ, [[ 1,  0,  2,  0, -2, -1],
....:                 [ 3, -2,  3, -1,  0,  6],
```

```
....:                 [-4,  2, -3,  1, -1, -8],
....:                 [-2,  2, -3,  2,  1,  0],
....:                 [ 0, -1, -1,  0,  2,  5],
....:                 [-1,  2, -4, -1,  5, -3]])
sage: P, L, U = D.LU(pivot='nonzero')
sage: P
[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: L
[   1    0    0    0    0    0]
[   3    1    0    0    0    0]
[  -4   -1    1    0    0    0]
[  -2   -1   -1    1    0    0]
[   0  1/2  1/4  1/2    1    0]
[  -1   -1 -5/2   -2   -6    1]
sage: U
[   1    0    2    0   -2   -1]
[   0   -2   -3   -1    6    9]
[   0    0    2    0   -3   -3]
[   0    0    0    1    0    4]
[   0    0    0    0 -1/4 -3/4]
[   0    0    0    0    0    1]
sage: D == L*U
True
```

The base ring of the matrix may be any field, or a ring which has a fraction field implemented in Sage. The ring needs to be exact (there is a numerical LU decomposition for matrices over `RDF` and `CDF`). Matrices returned are over the original field, or the fraction field of the ring. If the field is not ordered (i.e. the absolute value function is not implemented), then the pivot strategy needs to be 'nonzero'.

```
sage: A = matrix(RealField(100), 3, 3, range(9))
sage: P, L, U = A.LU()
Traceback (most recent call last):
...
TypeError: base ring of the matrix must be exact, not Real Field with 100␣
↪bits of precision

sage: A = matrix(Integers(6), 3, 2, range(6))
sage: A.LU()
Traceback (most recent call last):
...
TypeError: base ring of the matrix needs a field of fractions, not Ring of␣
↪integers modulo 6

sage: R.<y> = PolynomialRing(QQ, 'y')
sage: B = matrix(R, [[y+1, y^2+y], [y^2, y^3]])
sage: P, L, U = B.LU(pivot='partial')
Traceback (most recent call last):
...
TypeError: cannot take absolute value of matrix entries, try 'pivot=nonzero'
sage: P, L, U = B.LU(pivot='nonzero')
sage: P
[1 0]
[0 1]
```

```
sage: L
[          1           0]
[y^2/(y + 1)           1]
sage: U
[   y + 1 y^2 + y]
[       0       0]
sage: L.base_ring()
Fraction Field of Univariate Polynomial Ring in y over Rational Field
sage: B == P*L*U
True

sage: F.<a> = FiniteField(5^2)
sage: C = matrix(F, [[a + 3, 4*a + 4, 2, 4*a + 2],
....:                [3, 2*a + 4, 2*a + 4, 2*a + 1],
....:                [3*a + 1, a + 3, 2*a + 4, 4*a + 3],
....:                [a, 3, 3*a + 1, a]])
sage: P, L, U = C.LU(pivot='nonzero')
sage: P
[1 0 0 0]
[0 1 0 0]
[0 0 1 0]
[0 0 0 1]
sage: L
[      1       0       0       0]
[3*a + 3       1       0       0]
[    2*a 4*a + 2       1       0]
[2*a + 3       2 2*a + 4       1]
sage: U
[  a + 3 4*a + 4       2 4*a + 2]
[      0   a + 1   a + 3 2*a + 4]
[      0       0       1 4*a + 2]
[      0       0       0       0]
sage: L.base_ring()
Finite Field in a of size 5^2
sage: C == P*L*U
True
```

With no pivoting strategy given (i.e. `pivot=None`) the routine will try to use partial pivoting, but then fall back to the nonzero strategy. For the nonsingular matrix below, we see evidence of pivoting when viewed over the rationals, and no pivoting over the integers mod 29.

```
sage: entries = [3, 20, 11, 7, 16, 28, 5, 15, 21, 23, 22, 18, 8, 23, 15, 2]
sage: A = matrix(Integers(29), 4, 4, entries)
sage: perm, _ = A.LU(format='compact'); perm
(0, 1, 2, 3)
sage: B = matrix(QQ, 4, 4, entries)
sage: perm, _ = B.LU(format='compact'); perm
(2, 0, 1, 3)
```

The U matrix is only guaranteed to be upper-triangular. The rows are not necessarily linearly independent, nor are the pivots columns or rank in evidence.

```
sage: A = matrix(QQ, [[ 1, -4,  1,  0, -2,  1, 3,  3,  2],
....:                 [-1,  4,  0, -4,  0, -4, 5, -7, -7],
....:                 [ 0,  0,  1, -4, -1, -3, 6, -5, -6],
....:                 [-2,  8, -1, -4,  2, -4, 1, -8, -7],
....:                 [ 1, -4,  2, -4, -3,  2, 5,  6,  4]])
sage: P, L, U = A.LU()
```

```
sage: U
[   -2      8     -1     -4      2     -4      1     -8     -7]
[    0      0    1/2     -2     -1     -2    9/2     -3   -7/2]
[    0      0    3/2     -6     -2      0   11/2      2    1/2]
[    0      0      0      0   -1/3     -1    5/3   -5/3   -5/3]
[    0      0      0      0    1/3     -3    7/3  -19/3  -19/3]
sage: A.rref()
[ 1 -4  0  4  0  0 -1 -1 -1]
[ 0  0  1 -4  0  0  1  0 -1]
[ 0  0  0  0  1  0 -2 -1 -1]
[ 0  0  0  0  0  1 -1  2  2]
[ 0  0  0  0  0  0  0  0  0]
sage: A.pivots()
(0, 2, 4, 5)
```

AUTHOR:

- Rob Beezer (2011-04-26)

QR (*full=True*)

Returns a factorization of `self` as a unitary matrix and an upper-triangular matrix.

INPUT:

- `full` - default: `True` - if `True` then the returned matrices have dimensions as described below. If `False` the `R` matrix has no zero rows and the columns of `Q` are a basis for the column space of `self`.

OUTPUT:

If `self` is an $m \times n$ matrix and `full=True` then this method returns a pair of matrices: Q is an $m \times m$ unitary matrix (meaning its inverse is its conjugate-transpose) and R is an $m \times n$ upper-triangular matrix with non-negative entries on the diagonal. For a matrix of full rank this factorization is unique (due to the restriction to positive entries on the diagonal).

If `full=False` then Q has m rows and the columns form an orthonormal basis for the column space of `self`. So, in particular, the conjugate-transpose of Q times Q will be an identity matrix. The matrix R will still be upper-triangular but will also have full rank, in particular it will lack the zero rows present in a full factorization of a rank-deficient matrix.

The results obtained when `full=True` are cached, hence Q and R are immutable matrices in this case.

Note: This is an exact computation, so limited to exact rings. Also the base ring needs to have a fraction field implemented in Sage and this field must contain square roots. One example is the field of algebraic numbers, `QQbar`, as used in the examples below. If you need numerical results, convert the base ring to the field of complex double numbers, `CDF`, which will use a faster routine that is careful about numerical subtleties.

ALGORITHM:

"Modified Gram-Schmidt," Algorithm 8.1 of [TB1997].

EXAMPLES:

For a nonsingular matrix, the QR decomposition is unique.

```
sage: A = matrix(QQbar, [[-2, 0, -4, -1, -1],
....:                    [-2, 1, -6, -3, -1],
....:                    [1, 1, 7, 4, 5],
```

```
....:                     [3, 0, 8, 3, 3],
....:                     [-1, 1, -6, -6, 5]])
sage: Q, R = A.QR()
sage: Q
[ -0.4588314677411235?  -0.1260506983326509?   0.3812120831224489?   -0.
↪394573711338418?     -0.6874400625964?]
[ -0.4588314677411235?   0.4726901187474409? -0.05198346588033394?   0.
↪7172941251646595?     -0.2209628772631?]
[  0.2294157338705618?   0.6617661662464172?   0.6619227988762521?  -0.
↪1808720937375480?      0.1964114464561?]
[  0.6882472016116853?   0.1890760474989764?  -0.2044682991293135?   0.
↪0966302966543065?     -0.6628886317894?]
[ -0.2294157338705618?   0.5357154679137663?   -0.609939332995919?   -0.
↪536422031427112?      0.0245514308070?]
sage: R
[  4.358898943540674? -0.4588314677411235?   13.07669683062202?   6.
↪194224814505168?   2.982404540317303?]
[                   0    1.670171752907625?  0.5987408170800917?  -1.
↪292019657909672?   6.207996892883057?]
[                   0                    0   5.444401659866974?   5.
↪468660610611130? -0.6827161852283857?]
[                   0                    0                    0   1.
↪027626039419836?  -3.619300149686620?]
[                   0                    0                    0                   ␣
↪      0   0.024551430807012?]
sage: Q.conjugate_transpose()*Q
[1.000000000000000?            0.?e-18            0.?e-17            0.?e-16 ␣
↪            0.?e-13]
[           0.?e-18 1.000000000000000?            0.?e-17            0.?e-16 ␣
↪            0.?e-13]
[           0.?e-17            0.?e-17 1.000000000000000?            0.?e-16 ␣
↪            0.?e-13]
[           0.?e-16            0.?e-16            0.?e-16 1.000000000000000? ␣
↪            0.?e-13]
[           0.?e-13            0.?e-13            0.?e-13            0.?e-13 ␣
↪ 1.0000000000000?]
sage: Q*R == A
True
```

An example with complex numbers in `QQbar` , the field of algebraic numbers.

```
sage: A = matrix(QQbar, [[-8, 4*I + 1, -I + 2, 2*I + 1],
....:                     [1, -2*I - 1, -I + 3, -I + 1],
....:                     [I + 7, 2*I + 1, -2*I + 7, -I + 1],
....:                     [I + 2, 0, I + 12, -1]])
sage: Q, R = A.QR()
sage: Q
[                        -0.7302967433402215?    0.2070566455055649? + 0.
↪5383472783144687?*I    0.2463049809998642? - 0.0764456358723292?*I    0.
↪2381617683194332? - 0.1036596032779695?*I]
[                         0.0912870929175277?   -0.2070566455055649? - 0.
↪3778783780476559?*I    0.3786559533863033? - 0.1952221495524667?*I     0.
↪701244450214469? - 0.3643711650986595?*I]
[   0.6390096504226938? + 0.0912870929175277?*I    0.1708217325420910? + 0.
↪6677576817554466?*I -0.03411475806452072? + 0.04090198741767143?*I    0.
↪3140171085506764? - 0.0825191718705412?*I]
[   0.1825741858350554? + 0.0912870929175277?*I  -0.03623491296347385? + 0.
↪0724698259269477?*I   0.8632284069415110? + 0.06322839976356195?*I   -0.
↪4499694867611521? - 0.0116119181208918?*I]
```

```
sage: R
[                        10.95445115010333?              0.?e-18 - 1.
↪917028951268082?*I    5.385938482134133? - 2.190890230020665?*I  -0.
↪2738612787525831? - 2.190890230020665?*I]
[                                         0              4.829596256417300?
↪ + 0.?e-18*I   -0.8696379111233373? - 5.864879483945125?*I   0.
↪993871898426712? - 0.3054085521207082?*I]
[                                         0                                 ␣
↪           0              12.00160760935814? + 0.?e-16*I -0.
↪2709533402297273? + 0.4420629644486323?*I]
[                                         0                                 ␣
↪           0                                         0              1.
↪942963944258992? + 0.?e-16*I]
sage: Q.conjugate_transpose()*Q
[1.000000000000000? + 0.?e-19*I            0.?e-18 + 0.?e-17*I            0.?
↪e-17 + 0.?e-17*I            0.?e-16 + 0.?e-16*I]
[           0.?e-18 + 0.?e-17*I 1.000000000000000? + 0.?e-17*I            0.?
↪e-17 + 0.?e-17*I            0.?e-16 + 0.?e-16*I]
[           0.?e-17 + 0.?e-17*I            0.?e-17 + 0.?e-17*I 1.
↪000000000000000? + 0.?e-17*I            0.?e-16 + 0.?e-16*I]
[           0.?e-16 + 0.?e-16*I            0.?e-16 + 0.?e-16*I            0.?
↪e-16 + 0.?e-16*I 1.000000000000000? + 0.?e-16*I]
sage: Q*R - A
[             0.?e-17 0.?e-17 + 0.?e-17*I 0.?e-16 + 0.?e-16*I 0.?e-16 + 0.?e-
↪16*I]
[             0.?e-18 0.?e-17 + 0.?e-17*I 0.?e-16 + 0.?e-16*I 0.?e-16 + 0.?e-
↪16*I]
[0.?e-17 + 0.?e-18*I 0.?e-17 + 0.?e-17*I 0.?e-16 + 0.?e-16*I 0.?e-16 + 0.?e-
↪16*I]
[0.?e-18 + 0.?e-18*I 0.?e-18 + 0.?e-18*I 0.?e-16 + 0.?e-16*I 0.?e-16 + 0.?e-
↪16*I]
```

A rank-deficient rectangular matrix, with both values of the `full` keyword.

```
sage: A = matrix(QQbar, [[2, -3, 3],
....:                    [-1, 1, -1],
....:                    [-1, 3, -3],
....:                    [-5, 1, -1]])
sage: Q, R = A.QR()
sage: Q
[  0.3592106040535498?  -0.5693261797050169?   0.7239227659930268?   0.
↪1509015305256380?]
[ -0.1796053020267749?   0.1445907757980996?                     0   0.
↪9730546968377341?]
[ -0.1796053020267749?   0.7048800320157352?    0.672213996993525?  -0.
↪1378927778941174?]
[ -0.8980265101338745?  -0.3976246334447737?   0.1551263069985058? -0.
↪1066717157846818?]
sage: R
[ 5.567764362830022? -2.694079530401624?  2.694079530401624?]
[                  0  3.569584777515583? -3.569584777515583?]
[                  0                   0                   0]
[                  0                   0                   0]
sage: Q.conjugate_transpose()*Q
[                 1            0.?e-18            0.?e-18            0.?e-18]
[           0.?e-18                  1            0.?e-18            0.?e-18]
[           0.?e-18            0.?e-18 1.000000000000000?            0.?e-18]
[           0.?e-18            0.?e-18            0.?e-18 1.000000000000000?]
```

```
sage: Q, R = A.QR(full=False)
sage: Q
[ 0.3592106040535498? -0.5693261797050169?]
[-0.1796053020267749?  0.1445907757980996?]
[-0.1796053020267749?  0.7048800320157352?]
[-0.8980265101338745? -0.3976246334447737?]
sage: R
[ 5.567764362830022? -2.694079530401624?  2.694079530401624?]
[                  0  3.569584777515583? -3.569584777515583?]
sage: Q.conjugate_transpose()*Q
[      1 0.?e-18]
[0.?e-18       1]
```

Another rank-deficient rectangular matrix, with complex entries, as a reduced decomposition.

```
sage: A = matrix(QQbar, [[-3*I - 3, I - 3, -12*I + 1, -2],
....:                    [-I - 1, -2, 5*I - 1, -I - 2],
....:                    [-4*I - 4, I - 5, -7*I, -I - 4]])
sage: Q, R = A.QR(full=False)
sage: Q
[ -0.4160251471689219? - 0.4160251471689219?*I   0.5370861555295747? + 0.
↪1790287185098583?*I]
[ -0.1386750490563073? - 0.1386750490563073?*I  -0.7519206177414046? - 0.
↪2506402059138015?*I]
[ -0.5547001962252291? - 0.5547001962252291?*I -0.2148344622118299? - 0.
↪07161148740394329?*I]
sage: R
[                        7.211102550927979?  3.328201177351375? - 5.
↪269651864139676?*I   7.904477796209515? + 8.45917799243475?*I   4.
↪021576422632911? - 2.634825932069838?*I]
[                                          0                         1.
↪074172311059150?  -1.611258466588724? - 9.13046464400277?*I 1.
↪611258466588724? + 0.5370861555295747?*I]
sage: Q.conjugate_transpose()*Q
[1 0]
[0 1]
sage: Q*R-A
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
```

Results of full decompositions are cached and thus returned immutable.

```
sage: A = random_matrix(QQbar, 2, 2)
sage: Q, R = A.QR()
sage: Q.is_mutable()
False
sage: R.is_mutable()
False
```

Trivial cases return trivial results of the correct size, and we check Q itself in one case.

```
sage: A = zero_matrix(QQbar, 0, 10)
sage: Q, R = A.QR()
sage: Q.nrows(), Q.ncols()
(0, 0)
sage: R.nrows(), R.ncols()
```

```
(0, 10)
sage: A = zero_matrix(QQbar, 3, 0)
sage: Q, R = A.QR()
sage: Q.nrows(), Q.ncols()
(3, 3)
sage: R.nrows(), R.ncols()
(3, 0)
sage: Q
[1 0 0]
[0 1 0]
[0 0 1]
```

AUTHOR:

- Rob Beezer (2011-02-17)

T

Returns the transpose of a matrix.

EXAMPLES:

```
sage: A = matrix(QQ, 5, range(25))
sage: A.T
[ 0  5 10 15 20]
[ 1  6 11 16 21]
[ 2  7 12 17 22]
[ 3  8 13 18 23]
[ 4  9 14 19 24]
```

adjoint ()

Returns the adjoint matrix of self (matrix of cofactors).

OUTPUT:

- `N` - the adjoint matrix, such that N * M = M * N = M.parent(M.det())

ALGORITHM:

Use PARI whenever the method `self._adjoint` is included to do so in an inheriting class. Otherwise, use a generic division-free algorithm to compute the characteristic polynomial and hence the adjoint.

The result is cached.

EXAMPLES:

```
sage: M = Matrix(ZZ,2,2,[5,2,3,4]) ; M
[5 2]
[3 4]
sage: N = M.adjoint() ; N
[ 4 -2]
[-3  5]
sage: M * N
[14  0]
[ 0 14]
sage: N * M
[14  0]
[ 0 14]
sage: M = Matrix(QQ,2,2,[5/3,2/56,33/13,41/10]) ; M
[  5/3  1/28]
[33/13 41/10]
sage: N = M.adjoint() ; N
```

```
[ 41/10  -1/28]
[-33/13    5/3]
sage: M * N
[7363/1092         0]
[        0 7363/1092]
```

AUTHORS:

- Unknown: No author specified in the file from 2009-06-25
- Sebastian Pancratz (2009-06-25): Reflecting the change that `_adjoint` is now implemented in this class

apply_map (*phi*, *R=None*, *sparse=None*)

Apply the given map phi (an arbitrary Python function or callable object) to this dense matrix. If R is not given, automatically determine the base ring of the resulting matrix.

INPUT:

- `sparse` – True to make the output a sparse matrix; default False
- `phi` - arbitrary Python function or callable object
- `R` - (optional) ring

OUTPUT: a matrix over R

EXAMPLES:

```
sage: m = matrix(ZZ, 3, 3, range(9))
sage: k.<a> = GF(9)
sage: f = lambda x: k(x)
sage: n = m.apply_map(f); n
[0 1 2]
[0 1 2]
[0 1 2]
sage: n.parent()
Full MatrixSpace of 3 by 3 dense matrices over Finite Field in a of size 3^2
```

In this example, we explicitly specify the codomain.

```
sage: s = GF(3)
sage: f = lambda x: s(x)
sage: n = m.apply_map(f, k); n
[0 1 2]
[0 1 2]
[0 1 2]
sage: n.parent()
Full MatrixSpace of 3 by 3 dense matrices over Finite Field in a of size 3^2
```

If self is subdivided, the result will be as well:

```
sage: m = matrix(2, 2, srange(4))
sage: m.subdivide(None, 1); m
[0|1]
[2|3]
sage: m.apply_map(lambda x: x*x)
[0|1]
[4|9]
```

If the matrix is sparse, the result will be as well:

```
sage: m = matrix(ZZ,100,100,sparse=True)
sage: m[18,32] = -6
sage: m[1,83] = 19
sage: n = m.apply_map(abs, R=ZZ)
sage: n.dict()
{(1, 83): 19, (18, 32): 6}
sage: n.is_sparse()
True
```

If the map sends most of the matrix to zero, then it may be useful to get the result as a sparse matrix.

```
sage: m = matrix(ZZ, 3, 3, range(1, 10))
sage: n = m.apply_map(lambda x: 1//x, sparse=True); n
[1 0 0]
[0 0 0]
[0 0 0]
sage: n.parent()
Full MatrixSpace of 3 by 3 sparse matrices over Integer Ring
```

apply_morphism (*phi*)

Apply the morphism phi to the coefficients of this dense matrix.

The resulting matrix is over the codomain of phi.

INPUT:

- `phi` - a morphism, so phi is callable and phi.domain() and phi.codomain() are defined. The codomain must be a ring.

OUTPUT: a matrix over the codomain of phi

EXAMPLES:

```
sage: m = matrix(ZZ, 3, 3, range(9))
sage: phi = ZZ.hom(GF(5))
sage: m.apply_morphism(phi)
[0 1 2]
[3 4 0]
[1 2 3]
sage: parent(m.apply_morphism(phi))
Full MatrixSpace of 3 by 3 dense matrices over Finite Field of size 5
```

We apply a morphism to a matrix over a polynomial ring:

```
sage: R.<x,y> = QQ[]
sage: m = matrix(2, [x,x^2 + y, 2/3*y^2-x, x]); m
[          x     x^2 + y]
[2/3*y^2 - x           x]
sage: phi = R.hom([y,x])
sage: m.apply_morphism(phi)
[          y     y^2 + x]
[2/3*x^2 - y           y]
```

as_bipartite_graph ()

Construct a bipartite graph `B` representing the matrix uniquely.

Vertices are labeled 1 to `nrows` on the left and `nrows` + 1 to `nrows` + `ncols` on the right, representing rows and columns correspondingly. Each row is connected to each column with an edge weighted by the value of the corresponding matrix entry.

This graph is a helper for calculating automorphisms of a matrix under row and column permutations. See `automorphisms_of_rows_and_columns()`.

OUTPUT:

•A bipartite graph.

EXAMPLES:

```
sage: M = matrix(QQ, [[1/3, 7], [6, 1/4], [8, -5]])
sage: M
[1/3   7]
[  6 1/4]
[  8  -5]

sage: B = M.as_bipartite_graph()
sage: B
Bipartite graph on 5 vertices
sage: B.edges()
[(1, 4, 1/3), (1, 5, 7), (2, 4, 6), (2, 5, 1/4), (3, 4, 8), (3, 5, -5)]
sage: len(B.left) == M.nrows()
True
sage: len(B.right) == M.ncols()
True
```

as_sum_of_permutations()

Returns the current matrix as a sum of permutation matrices

According to the Birkhoff-von Neumann Theorem, any bistochastic matrix can be written as a positive sum of permutation matrices, which also means that the polytope of bistochastic matrices is integer.

As a non-bistochastic matrix can obviously not be written as a sum of permutations, this theorem is an equivalence.

This function, given a bistochastic matrix, returns the corresponding decomposition.

See also:

•`bistochastic_as_sum_of_permutations` – for more information on this method.

•`Birkhoff_polytope()`

EXAMPLES:

We create a bistochastic matrix from a convex sum of permutations, then try to deduce the decomposition from the matrix

```
sage: L = []
sage: L.append((9,Permutation([4, 1, 3, 5, 2])))
sage: L.append((6,Permutation([5, 3, 4, 1, 2])))
sage: L.append((3,Permutation([3, 1, 4, 2, 5])))
sage: L.append((2,Permutation([1, 4, 2, 3, 5])))
sage: M = sum([c * p.to_matrix() for (c,p) in L])
sage: decomp = sage.combinat.permutation.bistochastic_as_sum_of_
↪permutations(M)
sage: print(decomp)
2*B[[1, 4, 2, 3, 5]] + 3*B[[3, 1, 4, 2, 5]] + 9*B[[4, 1, 3, 5, 2]] + 6*B[[5,␣
↪3, 4, 1, 2]]
```

An exception is raised when the matrix is not bistochastic:

```
sage: M = Matrix([[2,3],[2,2]])
sage: decomp = sage.combinat.permutation.bistochastic_as_sum_of_
↪permutations(M)
Traceback (most recent call last):
...
ValueError: The matrix is not bistochastic
```

automorphisms_of_rows_and_columns ( )

Return the automorphisms of `self` under permutations of rows and columns as a list of pairs of `PermutationGroupElement` objects.

EXAMPLES:

```
sage: M = matrix(ZZ,[[1,0],[1,0],[0,1]])
sage: M
[1 0]
[1 0]
[0 1]
sage: A = M.automorphisms_of_rows_and_columns()
sage: A
[((), ()), ((1,2), ())]
sage: M = matrix(ZZ,[[1,1,1,1],[1,1,1,1]])
sage: A = M.automorphisms_of_rows_and_columns()
sage: len(A)
48
```

One can now apply these automorphisms to `M` to show that it leaves it invariant:

```
sage: all(M.with_permuted_rows_and_columns(*i) == M for i in A)
True
```

characteristic_polynomial (**args*, ***kwds*)

Synonym for self.charpoly(...).

EXAMPLES:

```
sage: a = matrix(QQ, 2,2, [1,2,3,4]); a
[1 2]
[3 4]
sage: a.characteristic_polynomial('T')
T^2 - 5*T - 2
```

charpoly (*var='x'*, *algorithm=None*)

Returns the characteristic polynomial of self, as a polynomial over the base ring.

ALGORITHM:

In the generic case of matrices over a ring (commutative and with unity), there is a division-free algorithm, which can be accessed using `"df"` , with complexity $O(n^4)$. Alternatively, by specifying `"hessenberg"` , this method computes the Hessenberg form of the matrix and then reads off the characteristic polynomial. Moreover, for matrices over number fields, this method can use PARI's charpoly implementation instead.

The method's logic is as follows: If no algorithm is specified, first check if the base ring is a number field (and then use PARI), otherwise check if the base ring is the ring of integers modulo n (in which case compute the characteristic polynomial of a lift of the matrix to the integers, and then coerce back to the base), next check if the base ring is an exact field (and then use the Hessenberg form), or otherwise, use the generic division-free algorithm. If an algorithm is specified explicitly, if `algorithm == "hessenberg"` , use the Hessenberg form, or otherwise use the generic division-free algorithm.

The result is cached.

INPUT:

- `var` - a variable name (default: 'x')
- **`algorithm` - string:**
  - `"df"` - Generic $O(n^4)$ division-free algorithm
  - `"hessenberg"` - Use the Hessenberg form of the matrix

EXAMPLES:

First a matrix over $\mathbf{Z}$:

```
sage: A = MatrixSpace(ZZ,2)( [1,2,  3,4] )
sage: f = A.charpoly('x')
sage: f
x^2 - 5*x - 2
sage: f.parent()
Univariate Polynomial Ring in x over Integer Ring
sage: f(A)
[0 0]
[0 0]
```

An example over $\mathbf{Q}$:

```
sage: A = MatrixSpace(QQ,3)(range(9))
sage: A.charpoly('x')
x^3 - 12*x^2 - 18*x
sage: A.trace()
12
sage: A.determinant()
0
```

We compute the characteristic polynomial of a matrix over the polynomial ring $\mathbf{Z}[a]$:

```
sage: R.<a> = PolynomialRing(ZZ)
sage: M = MatrixSpace(R,2)([a,1,  a,a+1]); M
[    a     1]
[    a a + 1]
sage: f = M.charpoly('x'); f
x^2 + (-2*a - 1)*x + a^2
sage: f.parent()
Univariate Polynomial Ring in x over Univariate Polynomial Ring in a over
→Integer Ring
sage: M.trace()
2*a + 1
sage: M.determinant()
a^2
```

We compute the characteristic polynomial of a matrix over the multi-variate polynomial ring $\mathbf{Z}[x, y]$:

```
sage: R.<x,y> = PolynomialRing(ZZ,2)
sage: A = MatrixSpace(R,2)([x, y, x^2, y^2])
sage: f = A.charpoly('x'); f
x^2 + (-y^2 - x)*x - x^2*y + x*y^2
```

It's a little difficult to distinguish the variables. To fix this, we temporarily view the indeterminate as Z:

```
sage: with localvars(f.parent(), 'Z'): print(f)
Z^2 + (-y^2 - x)*Z - x^2*y + x*y^2
```

We could also compute f in terms of Z from the start:

```
sage: A.charpoly('Z')
Z^2 + (-y^2 - x)*Z - x^2*y + x*y^2
```

Here is an example over a number field:

```
sage: x = QQ['x'].gen()
sage: K.<a> = NumberField(x^2 - 2)
sage: m = matrix(K, [[a-1, 2], [a, a+1]])
sage: m.charpoly('Z')
Z^2 - 2*a*Z - 2*a + 1
sage: m.charpoly('a')(m) == 0
True
```

Over integers modulo n with composite n:

```
sage: A = Mat(Integers(6),3,3)(range(9))
sage: A.charpoly()
x^3
```

Here is an example over a general commutative ring, that is to say, as of version 4.0.2, SAGE does not even positively determine that `S` in the following example is an integral domain. But the computation of the characteristic polynomial succeeds as follows:

```
sage: R.<a,b> = QQ[]
sage: S.<x,y> = R.quo((b^3))
sage: A = matrix(S, [[x*y^2,2*x],[2,x^10*y]])
sage: A
[ x*y^2     2*x]
[     2 x^10*y]
sage: A.charpoly('T')
T^2 + (-x^10*y - x*y^2)*T - 4*x
```

AUTHORS:

- Unknown: No author specified in the file from 2009-06-25
- Sebastian Pancratz (2009-06-25): Include the division-free algorithm

cholesky ( )

Returns the Cholesky decomposition of a symmetric or Hermitian matrix.

INPUT:

A square matrix that is real, symmetric and positive definite. Or a square matrix that is complex, Hermitian and positive definite. Generally, the base ring for the entries of the matrix needs to be a subfield of the algebraic numbers (`QQbar`). Examples include the rational numbers (`QQ`), some number fields, and real algebraic numbers and the algebraic numbers themselves.

OUTPUT:

For a matrix A the routine returns a lower triangular matrix L such that,

$$A = LL^*$$

where L^* is the conjugate-transpose in the complex case, and just the transpose in the real case. If the matrix fails to be positive definite (perhaps because it is not symmetric or Hermitian), then a `ValueError` results.

ALGORITHM:

Whether or not the matrix is positive definite is checked first in every case. This is accomplished with an indefinite factorization (see `indefinite_factorization()`) which caches its result. This algorithm is of an order $n^3/3$. If the matrix is positive definite, this computation always succeeds, using just field operations. The transition to a Cholesky decomposition "only" requires computing square roots of the positive (real) entries of the diagonal matrix produced in the indefinite factorization. Hence, there is no real penalty in the positive definite check (here, or prior to calling this routine), but a field extension with square roots may not be implemented in all reasonable cases.

EXAMPLES:

This simple example has a result with entries that remain in the field of rational numbers.

```
sage: A = matrix(QQ, [[ 4, -2,  4,  2],
....:                 [-2, 10, -2, -7],
....:                 [ 4, -2,  8,  4],
....:                 [ 2, -7,  4,  7]])
sage: A.is_symmetric()
True
sage: L = A.cholesky()
sage: L
[ 2  0  0  0]
[-1  3  0  0]
[ 2  0  2  0]
[ 1 -2  1  1]
sage: L.parent()
Full MatrixSpace of 4 by 4 dense matrices over Rational Field
sage: L*L.transpose() == A
True
```

This seemingly simple example requires first moving to the rational numbers for field operations, and then square roots necessitate that the result has entries in the field of algebraic numbers.

```
sage: A = matrix(ZZ, [[ 78, -30, -37,  -2],
....:                 [-30, 102, 179, -18],
....:                 [-37, 179, 326, -38],
....:                 [ -2, -18, -38,  15]])
sage: A.is_symmetric()
True
sage: L = A.cholesky()
sage: L
[   8.83176086632785?                   0                   0                   ␣
↪     0]
[ -3.396831102433787?    9.51112708681461?                   0                   ␣
↪     0]
[ -4.189425026335004?   17.32383862241232?   2.886751345948129?                   ␣
↪     0]
[-0.2264554068289192?  -1.973397116652010?  -1.649572197684645?   2.
↪886751345948129?]
sage: L.parent()
Full MatrixSpace of 4 by 4 dense matrices over Algebraic Real Field
sage: L*L.transpose() == A
True
```

Some subfields of the complex numbers, such as this number field of complex numbers with rational real and imaginary parts, allow for this computation.

```
sage: C.<I> = QuadraticField(-1)
sage: A = matrix(C, [[        23,  17*I + 3,  24*I + 25,     21*I],
....:                [ -17*I + 3,        38, -69*I + 89, 7*I + 15],
....:                [-24*I + 25, 69*I + 89,        976, 24*I + 6],
....:                [     -21*I, -7*I + 15,  -24*I + 6,       28]])
sage: A.is_hermitian()
True
sage: L = A.cholesky()
sage: L
[               4.79...?                        0                        0 ␣
↪      0]
[   0.62...? - 3.54...?*I                 5.00...?                        0 ␣
↪      0]
[   5.21...? - 5.00...?*I   13.58...? + 10.72...?*I                24.98...? ␣
↪      0]
[             -4.37...?*I   -0.10...? -  0.85...?*I  -0.21...? + 0.37...?*I 2.
↪81...?]
sage: L.parent()
Full MatrixSpace of 4 by 4 dense matrices over Algebraic Field
sage: (L*L.conjugate_transpose() - A.change_ring(QQbar)).norm() < 10^-10
True
```

The field of algebraic numbers is an ideal setting for this computation.

```
sage: A = matrix(QQbar, [[        2,    4 + 2*I,    6 - 4*I],
....:                    [ -2*I + 4,         11, 10 - 12*I],
....:                    [  4*I + 6, 10 + 12*I,        37]])
sage: A.is_hermitian()
True
sage: L = A.cholesky()
sage: L
[                      1.414213562373095?          0                     0]
[2.828427124746190? - 1.414213562373095?*I          1                     0]
[4.242640687119285? + 2.828427124746190?*I   -2*I + 2    1.732050807568878?]
sage: L.parent()
Full MatrixSpace of 3 by 3 dense matrices over Algebraic Field
sage: (L*L.conjugate_transpose() - A.change_ring(QQbar)).norm() < 10^-10
True
```

Results are cached, hence immutable. Use the `copy` function if you need to make a change.

```
sage: A = matrix(QQ, [[ 4, -2,   4,   2],
....:                 [-2, 10, -2, -7],
....:                 [ 4, -2,   8,   4],
....:                 [ 2, -7,   4,   7]])
sage: L = A.cholesky()
sage: L.is_immutable()
True

sage: from copy import copy
sage: LC = copy(L)
sage: LC[0,0] = 1000
sage: LC
[1000    0    0    0]
[  -1    3    0    0]
[   2    0    2    0]
```

```
[   1   -2    1    1]
```

There are a variety of situations which will prevent the computation of a Cholesky decomposition.

The base ring must be exact. For numerical work, create a matrix with a base ring of RDF or CDF and use the *cholesky()* method for matrices of that type.

```
sage: F = RealField(100)
sage: A = matrix(F, [[1.0, 3.0], [3.0, -6.0]])
sage: A.cholesky()
Traceback (most recent call last):
...
TypeError: base ring of the matrix must be exact, not Real Field with 100␣
↪bits of precision
```

The base ring may not have a fraction field.

```
sage: A = matrix(Integers(6), [[2, 0], [0, 4]])
sage: A.cholesky()
Traceback (most recent call last):
...
ValueError: Could not see Ring of integers modulo 6 as a subring of
the real or complex numbers
```

The base field may not have elements that are comparable to zero.

```
sage: F.<a> = FiniteField(5^4)
sage: A = matrix(F, [[2+a^3, 3], [3, 3]])
sage: A.cholesky()
Traceback (most recent call last):
...
ValueError: Could not see Finite Field in a of size 5^4 as a subring
of the real or complex numbers
```

The algebraic closure of the fraction field of the base ring may not be implemented.

```
sage: F = Integers(7)
sage: A = matrix(F, [[4, 0], [0, 3]])
sage: A.cholesky()
Traceback (most recent call last):
...
ValueError: Could not see Ring of integers modulo 7 as a subring of
the real or complex numbers
```

The matrix may not be positive definite.

```
sage: C.<I> = QuadraticField(-1)
sage: B = matrix(C, [[      2, 4 - 2*I, 2 + 2*I],
....:                [4 + 2*I,       8,    10*I],
....:                [2 - 2*I,   -10*I,      -3]])
sage: B.is_positive_definite()
False
sage: B.cholesky()
Traceback (most recent call last):
...
ValueError: matrix is not positive definite,
so cannot compute Cholesky decomposition
```

The matrix could be positive semi-definite, and thus lack a Cholesky decomposition.

```
sage: A = matrix(QQ, [[21, 15, 12, -3],
....:                 [15, 12,  9,  12],
....:                 [12,  9,  7,  3],
....:                 [-3,  12,  3,  8]])
sage: A.is_positive_definite()
False
sage: [A[:i,:i].determinant() for i in range(1,A.nrows()+1)]
[21, 27, 0, 0]
sage: A.cholesky()
Traceback (most recent call last):
...
ValueError: matrix is not positive definite,
so cannot compute Cholesky decomposition
```

AUTHOR:

- Rob Beezer (2012-05-27)

column_module ( )

Return the free module over the base ring spanned by the columns of this matrix.

EXAMPLES:

```
sage: t = matrix(QQ, 3, 3, range(9)); t
[0 1 2]
[3 4 5]
[6 7 8]
sage: t.column_module()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]
```

column_space ( )

Return the vector space over the base ring spanned by the columns of this matrix.

EXAMPLES:

```
sage: M = MatrixSpace(QQ,3,3)
sage: A = M([1,9,-7,4/5,4,3,6,4,3])
sage: A.column_space()
Vector space of degree 3 and dimension 3 over Rational Field
Basis matrix:
[1 0 0]
[0 1 0]
[0 0 1]
sage: W = MatrixSpace(CC,2,2)
sage: B = W([1, 2+3*I,4+5*I,9]); B
[                     1.00000000000000 2.00000000000000 + 3.00000000000000*I]
[4.00000000000000 + 5.00000000000000*I                      9.00000000000000]
sage: B.column_space()
Vector space of degree 2 and dimension 2 over Complex Field with 53 bits of
precision
Basis matrix:
[ 1.00000000000000 0.000000000000000]
[0.000000000000000  1.00000000000000]
```

conjugate ( )

Return the conjugate of self, i.e. the matrix whose entries are the conjugates of the entries of self.

EXAMPLES:

```
sage: A = matrix(CDF, [[1+I,1],[0,2*I]])
sage: A.conjugate()
[1.0 - 1.0*I          1.0]
[        0.0       -2.0*I]
```

A matrix over a not-totally-real number field:

```
sage: K.<j> = NumberField(x^2+5)
sage: M = matrix(K, [[1+j,1], [0,2*j]])
sage: M.conjugate()
[-j + 1      1]
[     0   -2*j]
```

There is a shortcut for the conjugate:

```
sage: M.C
[-j + 1      1]
[     0   -2*j]
```

There is also a shortcut for the conjugate transpose, or "Hermitian transpose":

```
sage: M.H
[-j + 1      0]
[     1   -2*j]
```

Conjugates work (trivially) for matrices over rings that embed canonically into the real numbers:

```
sage: M = random_matrix(ZZ, 2)
sage: M == M.conjugate()
True
sage: M = random_matrix(QQ, 3)
sage: M == M.conjugate()
True
sage: M = random_matrix(RR, 2)
sage: M == M.conjugate()
True
```

conjugate_transpose ( )

Returns the transpose of `self` after each entry has been converted to its complex conjugate.

Note: This function is sometimes known as the "adjoint" of a matrix, though there is substantial variation and some confusion with the use of that term.

OUTPUT:

A matrix formed by taking the complex conjugate of every entry of `self` and then transposing the resulting matrix.

Complex conjugation is implemented for many subfields of the complex numbers. See the examples below, or more at `conjugate()`.

EXAMPLES:

```
sage: M = matrix(SR, 2, 2, [[2-I, 3+4*I], [9-6*I, 5*I]])
sage: M.base_ring()
Symbolic Ring
sage: M.conjugate_transpose()
 [   I + 2  6*I + 9]
[-4*I + 3     -5*I]

sage: P = matrix(CC, 3, 2, [0.95-0.63*I, 0.84+0.13*I, 0.94+0.23*I, 0.23+0.
↪59*I, 0.52-0.41*I, -0.50+0.90*I])
sage: P.base_ring()
Complex Field with 53 bits of precision
sage: P.conjugate_transpose()
[ 0.950... + 0.630...*I  0.940... - 0.230...*I  0.520... + 0.410...*I]
[ 0.840... - 0.130...*I  0.230... - 0.590...*I -0.500... - 0.900...*I]
```

There is also a shortcut for the conjugate transpose, or "Hermitian transpose":

```
sage: M.H
 [   I + 2  6*I + 9]
[-4*I + 3     -5*I]
```

Matrices over base rings that can be embedded in the real numbers will behave as expected.

```
sage: P = random_matrix(QQ, 3, 4)
sage: P.conjugate_transpose() == P.transpose()
True
```

The conjugate of a matrix is formed by taking conjugates of all the entries. Some specialized subfields of the complex numbers are implemented in Sage and complex conjugation can be applied. (Matrices over quadratic number fields are another class of examples.)

```
sage: C = CyclotomicField(5)
sage: a = C.gen(); a
zeta5
sage: CC(a)
0.309016994374947 + 0.951056516295154*I
sage: M = matrix(C, 1, 2, [a^2, a+a^3])
sage: M.conjugate_transpose()
[             zeta5^3]
[-zeta5^3 - zeta5 - 1]
```

Conjugation does not make sense over rings not containing complex numbers.

```
sage: N = matrix(GF(5), 2, [0,1,2,3])
sage: N.conjugate_transpose()
Traceback (most recent call last):
...
AttributeError: 'sage.rings.finite_rings.integer_mod.IntegerMod_int' object␣
↪has no attribute 'conjugate'
```

AUTHOR:

Rob Beezer (2010-12-13)

cyclic_subspace (*v*, *var=None*, *basis='echelon'*)

Create a cyclic subspace for a vector, and optionally, a minimal polynomial for the iterated powers.

These subspaces are also known as Krylov subspaces. They are spanned by the vectors

$$\{v, Av, A^2v, A^3v, \dots\}$$

INPUT:

- `self` - a square matrix with entries from a field.
- `v` - a vector with a degree equal to the size of the matrix and entries compatible with the entries of the matrix.
- `var` - default: `None` - if specified as a string or a generator of a polynomial ring, then this will be used to construct a polynomial reflecting a relation of linear dependence on the powers A^iv *and* this will cause the polynomial to be returned along with the subspace. A generator must create polynomials with coefficients from the same field as the matrix entries.
- `basis` - default: `echelon` - the basis for the subspace is "echelonized" by default, but the keyword 'iterates' will return a subspace with a user basis equal to the largest linearly independent set $\{v, Av, A^2v, A^3v, \dots, A^{k-1}v\}$.

OUTPUT:

Suppose k is the smallest power such that $\{v, Av, A^2v, A^3v, \dots, A^kv\}$ is linearly dependent. Then the subspace returned will have dimension k and be spanned by the powers 0 through $k-1$.

If a polynomial is requested through the use of the `var` keyword, then a pair is returned, with the polynomial first and the subspace second. The polynomial is the unique monic polynomial whose coefficients provide a relation of linear dependence on the first k powers.

For less convenient, but more flexible output, see the helper method "_cyclic_subspace" in this module.

EXAMPLES:

```
sage: A = matrix(QQ, [[5,4,2,1],[0,1,-1,-1],[-1,-1,3,0],[1,1,-1,2]])
sage: v = vector(QQ, [0,1,0,0])
sage: E = A.cyclic_subspace(v); E
Vector space of degree 4 and dimension 3 over Rational Field
Basis matrix:
[ 1  0  0  0]
[ 0  1  0  0]
[ 0  0  1 -1]
sage: F = A.cyclic_subspace(v, basis='iterates'); F
Vector space of degree 4 and dimension 3 over Rational Field
User basis matrix:
[ 0  1  0  0]
[ 4  1 -1  1]
[23  1 -8  8]
sage: E == F
True
sage: p, S = A.cyclic_subspace(v, var='T'); p
T^3 - 9*T^2 + 24*T - 16
sage: gen = polygen(QQ, 'z')
sage: p, S = A.cyclic_subspace(v, var=gen); p
z^3 - 9*z^2 + 24*z - 16
sage: p.degree() == E.dimension()
True
```

The polynomial has coefficients that yield a non-trivial relation of linear dependence on the iterates. Or, equivalently, evaluating the polynomial with the matrix will create a matrix that annihilates the vector.

```
sage: A = matrix(QQ, [[15, 37/3, -16, -104/3, -29, -7/3, 35, 2/3, -29/3, -1/
↪3],
....:                 [ 2, 9, -1, -6, -6, 0, 7, 0, -2, 0],
....:                 [24, 74/3, -29, -208/3, -58, -14/3, 70, 4/3, -58/3, -2/
↪3],
....:                 [-6, -19, 3, 21, 19, 0, -21, 0, 6, 0],
....:                 [2, 6, -1, -6, -3, 0, 7, 0, -2, 0],
....:                 [-96, -296/3, 128, 832/3, 232, 65/3, -279, -16/3, 232/
↪3, 8/3],
....:                 [0, 0, 0, 0, 0, 0, 3, 0, 0, 0],
....:                 [20, 26/3, -30, -199/3, -42, -14/3, 70, 13/3, -55/3, -2/
↪3],
....:                 [18, 57, -9, -54, -57, 0, 63, 0, -15, 0],
....:                 [0, 0, 0, 0, 0, 0, 0, 0, 0, 3]])
sage: u = zero_vector(QQ, 10); u[0] = 1
sage: p, S = A.cyclic_subspace(u, var='t', basis='iterates')
sage: S
Vector space of degree 10 and dimension 3 over Rational Field
User basis matrix:
[   1    0    0    0    0    0    0    0    0    0]
[  15    2   24   -6    2  -96    0   20   18    0]
[  79   12  140  -36   12 -560    0  116  108    0]
sage: p
t^3 - 9*t^2 + 27*t - 27
sage: k = p.degree()
sage: coeffs = p.list()
sage: iterates = S.basis() + [A^k*u]
sage: sum(coeffs[i]*iterates[i] for i in range(k+1))
(0, 0, 0, 0, 0, 0, 0, 0, 0, 0)
sage: u in p(A).right_kernel()
True
```

AUTHOR:

- Rob Beezer (2011-05-20)

decomposition (*algorithm='spin'*, *is_diagonalizable=False*, *dual=False*)

Returns the decomposition of the free module on which this matrix A acts from the right (i.e., the action is x goes to x A), along with whether this matrix acts irreducibly on each factor. The factors are guaranteed to be sorted in the same way as the corresponding factors of the characteristic polynomial.

Let A be the matrix acting from the on the vector space V of column vectors. Assume that A is square. This function computes maximal subspaces W_1, ..., W_n corresponding to Galois conjugacy classes of eigenvalues of A. More precisely, let $f(X)$ be the characteristic polynomial of A. This function computes the subspace $W_i = ker(g_(A)^n)$, where $g_i(X)$ is an irreducible factor of $f(X)$ and $g_i(X)$ exactly divides $f(X)$. If the optional parameter is_diagonalizable is True, then we let $W_i = ker(g(A))$, since then we know that $ker(g(A)) = ker(g(A)^n)$.

INPUT:

- `self` - a matrix
- `algorithm` - 'spin' (default): algorithm involves iterating the action of self on a vector. 'kernel': naively just compute $ker(f_i(A))$ for each factor f_i.
- `dual` - bool (default: False): If True, also returns the corresponding decomposition of V under the action of the transpose of A. The factors are guaranteed to correspond.
- `is_diagonalizable` - if the matrix is known to be diagonalizable, set this to True, which might speed up the algorithm in some cases.

Note: If the base ring is not a field, the kernel algorithm is used.

OUTPUT:

- `Sequence` - list of pairs (V,t), where V is a vector spaces and t is a bool, and t is True exactly when the charpoly of self on V is irreducible.
- (optional) list - list of pairs (W,t), where W is a vector space and t is a bool, and t is True exactly when the charpoly of the transpose of self on W is irreducible.

EXAMPLES:

```
sage: A = matrix(ZZ, 4, [3,4,5,6,7,3,8,10,14,5,6,7,2,2,10,9])
sage: B = matrix(QQ, 6, 6, range(36))
sage: B*11
[  0  11  22  33  44  55]
[ 66  77  88  99 110 121]
[132 143 154 165 176 187]
[198 209 220 231 242 253]
[264 275 286 297 308 319]
[330 341 352 363 374 385]
sage: A.decomposition()
[
(Ambient free module of rank 4 over the principal ideal domain Integer Ring,␣
↪True)
]
sage: B.decomposition()
[
(Vector space of degree 6 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1 -2 -3 -4]
[ 0  1  2  3  4  5], True),
(Vector space of degree 6 and dimension 4 over Rational Field
Basis matrix:
[ 1  0  0  0 -5  4]
[ 0  1  0  0 -4  3]
[ 0  0  1  0 -3  2]
[ 0  0  0  1 -2  1], False)
]
```

decomposition_of_subspace (*M*, *check_restrict=True*, ***kwds*)

Suppose the right action of self on M leaves M invariant. Return the decomposition of M as a list of pairs (W, is_irred) where is_irred is True if the charpoly of self acting on the factor W is irreducible.

Additional inputs besides M are passed onto the decomposition command.

INPUT:

- M – A subspace of the free module `self` acts on.
- **`check_restrict` – A boolean (default: `True`); Call restrict** with or without check.
- `kwds` – Keywords that will be forwarded to `decomposition()` .

EXAMPLES:

```
sage: t = matrix(QQ, 3, [3, 0, -2, 0, -2, 0, 0, 0, 0]); t
[ 3  0 -2]
[ 0 -2  0]
[ 0  0  0]
```

```
sage: t.fcp('X')    # factored charpoly
(X - 3) * X * (X + 2)
sage: v = kernel(t*(t+2)); v    # an invariant subspace
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[0 1 0]
[0 0 1]
sage: D = t.decomposition_of_subspace(v); D
[
(Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[0 0 1], True),
(Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[0 1 0], True)
]
sage: t.restrict(D[0][0])
[0]
sage: t.restrict(D[1][0])
[-2]
```

We do a decomposition over ZZ:

```
sage: a = matrix(ZZ,6,[0, 0, -2, 0, 2, 0, 2, -4, -2, 0, 2, 0, 0, 0, -2, -2,␣
↪0, 0, 2, 0, -2, -4, 2, -2, 0, 2, 0, -2, -2, 0, 0, 2, 0, -2, 0, 0])
sage: a.decomposition_of_subspace(ZZ^6)
[
(Free module of degree 6 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1  0  1 -1  1 -1]
[ 0  1  0 -1  2 -1], False),
(Free module of degree 6 and rank 4 over Integer Ring
Echelon basis matrix:
[ 1  0 -1  0  1  0]
[ 0  1  0  0  0  0]
[ 0  0  0  1  0  0]
[ 0  0  0  0  0  1], False)
]
```

denominator ( )

Return the least common multiple of the denominators of the elements of self.

If there is no denominator function for the base field, or no LCM function for the denominators, raise a TypeError.

EXAMPLES:

```
sage: A = MatrixSpace(QQ,2)(['1/2', '1/3', '1/5', '1/7'])
sage: A.denominator()
210
```

A trivial example:

```
sage: A = matrix(QQ, 0,2)
sage: A.denominator()
1
```

Denominators are not defined for real numbers:

```
sage: A = MatrixSpace(RealField(),2)([1,2,3,4])
sage: A.denominator()
Traceback (most recent call last):
...
TypeError: denominator not defined for elements of the base ring
```

We can even compute the denominator of matrix over the fraction field of $\mathbf{Z}[x]$.

```
sage: K.<x> = Frac(ZZ['x'])
sage: A = MatrixSpace(K,2)([1/x, 2/(x+1), 1, 5/(x^3)])
sage: A.denominator()
x^4 + x^3
```

Here's an example involving a cyclotomic field:

```
sage: K.<z> = CyclotomicField(3)
sage: M = MatrixSpace(K,3,sparse=True)
sage: A = M([(1+z)/3,(2+z)/3,z/3,1,1+z,-2,1,5,-1+z])
sage: print(A)
[1/3*z + 1/3 1/3*z + 2/3       1/3*z]
[          1       z + 1          -2]
[          1           5       z - 1]
sage: print(A.denominator())
3
```

density ( )

Return the density of the matrix.

By density we understand the ratio of the number of nonzero positions and the self.nrows() * self.ncols(), i.e. the number of possible nonzero positions.

EXAMPLES:

First, note that the density parameter does not ensure the density of a matrix, it is only an upper bound.

```
sage: A = random_matrix(GF(127),200,200,density=0.3)
sage: A.density()
5211/20000
```

```
sage: A = matrix(QQ,3,3,[0,1,2,3,0,0,6,7,8])
sage: A.density()
2/3
```

```
sage: a = matrix([[],[],[],[]])
sage: a.density()
0
```

derivative (**args*)

Derivative with respect to variables supplied in args.

Multiple variables and iteration counts may be supplied; see documentation for the global derivative() function for more details.

EXAMPLES:

```
sage: v = vector([1,x,x^2])
sage: v.derivative(x)
(0, 1, 2*x)
```

```
sage: type(v.derivative(x)) == type(v)
True
sage: v = vector([1,x,x^2], sparse=True)
sage: v.derivative(x)
(0, 1, 2*x)
sage: type(v.derivative(x)) == type(v)
True
sage: v.derivative(x,x)
(0, 0, 2)
```

det (**args*, ***kwds*)

Synonym for self.determinant(...).

EXAMPLES:

```
sage: A = MatrixSpace(Integers(8),3)([1,7,3, 1,1,1, 3,4,5])
sage: A.det()
6
```

determinant (*algorithm=None*)

Returns the determinant of self.

ALGORITHM:

For small matrices (n less than 4), this is computed using the naive formula. In the specific case of matrices over the integers modulo a non-prime, the determinant of a lift is computed over the integers. In general, the characteristic polynomial is computed either using the Hessenberg form (specified by `"hessenberg"`) or the generic division-free algorithm (specified by `"df"`). When the base ring is an exact field, the default choice is `"hessenberg"` , otherwise it is `"df"` . Note that for matrices over most rings, more sophisticated algorithms can be used. (Type `A.determinant?` to see what is done for a specific matrix `A` .)

INPUT:

- **`algorithm`** **- string:**
  - `"df"` - Generic O(n^4) division-free algorithm
  - `"hessenberg"` - Use the Hessenberg form of the matrix

EXAMPLES:

```
sage: A = MatrixSpace(Integers(8),3)([1,7,3, 1,1,1, 3,4,5])
sage: A.determinant()
6
sage: A.determinant() is A.determinant()
True
sage: A[0,0] = 10
sage: A.determinant()
7
```

We compute the determinant of the arbitrary 3x3 matrix:

```
sage: R = PolynomialRing(QQ,9,'x')
sage: A = matrix(R,3,R.gens())
sage: A
[x0 x1 x2]
[x3 x4 x5]
[x6 x7 x8]
```

```
sage: A.determinant()
-x2*x4*x6 + x1*x5*x6 + x2*x3*x7 - x0*x5*x7 - x1*x3*x8 + x0*x4*x8
```

We create a matrix over $\mathbf{Z}[x,y]$ and compute its determinant.

```
sage: R.<x,y> = PolynomialRing(IntegerRing(),2)
sage: A = MatrixSpace(R,2)([x, y, x**2, y**2])
sage: A.determinant()
-x^2*y + x*y^2
```

A matrix over a non-domain:

```
sage: m = matrix(Integers(4), 2, [1,2,2,3])
sage: m.determinant()
3
```

AUTHORS:

- Unknown: No author specified in the file from 2009-06-25
- Sebastian Pancratz (2009-06-25): Use the division-free algorithm for charpoly
- Thierry Monteil (2010-10-05): Bugfix for trac ticket #10063, so that the determinant is computed even for rings for which the is_field method is not implemented.

diagonal ( )

Return the diagonal entries of `self`.

OUTPUT:

A list containing the entries of the matrix that have equal row and column indices, in order of the indices. Behavior is not limited to square matrices.

EXAMPLES:

```
sage: A = matrix([[2,5],[3,7]]); A
[2 5]
[3 7]
sage: A.diagonal()
[2, 7]
```

Two rectangular matrices.

```
sage: B = matrix(3, 7, range(21)); B
[ 0  1  2  3  4  5  6]
[ 7  8  9 10 11 12 13]
[14 15 16 17 18 19 20]
sage: B.diagonal()
[0, 8, 16]

sage: C = matrix(3, 2, range(6)); C
[0 1]
[2 3]
[4 5]
sage: C.diagonal()
[0, 3]
```

Empty matrices behave properly.

```
sage: E = matrix(0, 5, []); E
[]
sage: E.diagonal()
[]
```

echelon_form (*algorithm='default'*, *cutoff=0*, ***kwds*)
Return the echelon form of self.

Note: This row reduction does not use division if the matrix is not over a field (e.g., if the matrix is over the integers). If you want to calculate the echelon form using division, then use *rref()* , which assumes that the matrix entries are in a field (specifically, the field of fractions of the base ring of the matrix).

INPUT:

- `algorithm` – string. Which algorithm to use. Choices are
  - `'default'` : Let Sage choose an algorithm (default).
  - `'classical'` : Gauss elimination.
  - `'strassen'` : use a Strassen divide and conquer algorithm (if available)
- `cutoff` – integer. Only used if the Strassen algorithm is selected.
- `transformation` – boolean. Whether to also return the transformation matrix. Some matrix backends do not provide this information, in which case this option is ignored.

OUTPUT:

The reduced row echelon form of `self` , as an immutable matrix. Note that self is *not* changed by this command. Use *echelonize()* to change `self` in place.

If the optional parameter `transformation=True` is specified, the output consists of a pair (E, T) of matrices where E is the echelon form of `self` and T is the transformation matrix.

EXAMPLES:

```
sage: MS = MatrixSpace(GF(19),2,3)
sage: C = MS.matrix([1,2,3,4,5,6])
sage: C.rank()
2
sage: C.nullity()
0
sage: C.echelon_form()
[ 1  0 18]
[ 0  1  2]
```

The matrix library used for $\mathbf{Z}/p$-matrices does not return the transformation matrix, so the `transformation` option is ignored:

```
sage: C.echelon_form(transformation=True)
[ 1  0 18]
[ 0  1  2]

sage: D = matrix(ZZ, 2, 3, [1,2,3,4,5,6])
sage: D.echelon_form(transformation=True)
(
[1 2 3]  [ 1  0]
[0 3 6], [ 4 -1]
```

```
)
sage: E, T = D.echelon_form(transformation=True)
sage: T*D == E
True
```

echelonize (*algorithm='default'*, *cutoff=0*, ***kwds*)

Transform `self` into a matrix in echelon form over the same base ring as self.

Note: This row reduction does not use division if the matrix is not over a field (e.g., if the matrix is over the integers). If you want to calculate the echelon form using division, then use *rref()* , which assumes that the matrix entries are in a field (specifically, the field of fractions of the base ring of the matrix).

INPUT:

- `algorithm` – string. Which algorithm to use. Choices are
  - `'default'` : Let Sage choose an algorithm (default).
  - `'classical'` : Gauss elimination.
  - `'strassen'` : use a Strassen divide and conquer algorithm (if available)
- `cutoff` – integer. Only used if the Strassen algorithm is selected.
- `transformation` – boolean. Whether to also return the transformation matrix. Some matrix backends do not provide this information, in which case this option is ignored.

OUTPUT:

The matrix `self` is put into echelon form. Nothing is returned unless the keyword option `transformation=True` is specified, in which case the transformation matrix is returned.

EXAMPLES:

```
sage: a = matrix(QQ,3,3,range(9)); a
[0 1 2]
[3 4 5]
[6 7 8]
sage: a.echelonize()
sage: a
[ 1  0 -1]
[ 0  1  2]
[ 0  0  0]
```

An immutable matrix cannot be transformed into echelon form. Use `self.echelon_form()` instead:

```
sage: a = matrix(QQ,3,3,range(9)); a.set_immutable()
sage: a.echelonize()
Traceback (most recent call last):
...
ValueError: matrix is immutable; please change a copy instead
(i.e., use copy(M) to change a copy of M).
sage: a.echelon_form()
[ 1  0 -1]
[ 0  1  2]
[ 0  0  0]
```

Echelon form over the integers is what is also classically often known as Hermite normal form:

```
sage: a = matrix(ZZ,3,3,range(9))
sage: a.echelonize(); a
[ 3  0 -3]
[ 0  1  2]
[ 0  0  0]
```

We compute an echelon form both over a domain and fraction field:

```
sage: R.<x,y> = QQ[]
sage: a = matrix(R, 2, [x,y,x,y])
sage: a.echelon_form()               # not very useful? -- why two copies of␣
↪the same row?
[x y]
[x y]
```

```
sage: b = a.change_ring(R.fraction_field())
sage: b.echelon_form()               # potentially useful
[  1 y/x]
[  0   0]
```

Echelon form is not defined over arbitrary rings:

```
sage: a = matrix(Integers(9),3,3,range(9))
sage: a.echelon_form()
Traceback (most recent call last):
...
NotImplementedError: Echelon form not implemented over 'Ring of integers␣
↪modulo 9'.
```

Involving a sparse matrix:

```
sage: m = matrix(3,[1, 1, 1, 1, 0, 2, 1, 2, 0], sparse=True); m
[1 1 1]
[1 0 2]
[1 2 0]
sage: m.echelon_form()
[ 1  0  2]
[ 0  1 -1]
[ 0  0  0]
sage: m.echelonize(); m
[ 1  0  2]
[ 0  1 -1]
[ 0  0  0]
```

The transformation matrix is optionally returned:

```
sage: m_original = m
sage: transformation_matrix = m.echelonize(transformation=True)
sage: m == transformation_matrix * m_original
True
```

eigenmatrix_left ( )

Return matrices D and P, where D is a diagonal matrix of eigenvalues and P is the corresponding matrix where the rows are corresponding eigenvectors (or zero vectors) so that P*self = D*P.

EXAMPLES:

```
sage: A = matrix(QQ,3,3,range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: D, P = A.eigenmatrix_left()
sage: D
[                  0                   0                   0]
[                  0 -1.348469228349535?                   0]
[                  0                   0  13.34846922834954?]
sage: P
[                   1                   -2                    1]
[                   1  0.3101020514433644? -0.3797958971132713?]
[                   1   1.289897948556636?   1.579795897113272?]
sage: P*A == D*P
True
```

Because P is invertible, A is diagonalizable.

```
sage: A == (~P)*D*P
True
```

The matrix P may contain zero rows corresponding to eigenvalues for which the algebraic multiplicity is greater than the geometric multiplicity. In these cases, the matrix is not diagonalizable.

```
sage: A = jordan_block(2,3); A
[2 1 0]
[0 2 1]
[0 0 2]
sage: A = jordan_block(2,3)
sage: D, P = A.eigenmatrix_left()
sage: D
[2 0 0]
[0 2 0]
[0 0 2]
sage: P
[0 0 1]
[0 0 0]
[0 0 0]
sage: P*A == D*P
True
```

eigenmatrix_right ()

Return matrices D and P, where D is a diagonal matrix of eigenvalues and P is the corresponding matrix where the columns are corresponding eigenvectors (or zero vectors) so that self*P = P*D.

EXAMPLES:

```
sage: A = matrix(QQ,3,3,range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: D, P = A.eigenmatrix_right()
sage: D
[                  0                   0                   0]
[                  0 -1.348469228349535?                   0]
[                  0                   0  13.34846922834954?]
sage: P
[                   1                    1                    1]
```

```
[                  -2  0.1303061543300932?   3.069693845669907?]
[                   1 -0.7393876913398137?   5.139387691339814?]
sage: A*P == P*D
True
```

Because P is invertible, A is diagonalizable.

```
sage: A == P*D*(~P)
True
```

The matrix P may contain zero columns corresponding to eigenvalues for which the algebraic multiplicity is greater than the geometric multiplicity. In these cases, the matrix is not diagonalizable.

```
sage: A = jordan_block(2,3); A
[2 1 0]
[0 2 1]
[0 0 2]
sage: A = jordan_block(2,3)
sage: D, P = A.eigenmatrix_right()
sage: D
[2 0 0]
[0 2 0]
[0 0 2]
sage: P
[1 0 0]
[0 0 0]
[0 0 0]
sage: A*P == P*D
True
```

eigenspaces_left (*format='all'*, *var='a'*, *algebraic_multiplicity=False*)

Compute the left eigenspaces of a matrix.

Note that `eigenspaces_left()` and `left_eigenspaces()` are identical methods. Here "left" refers to the eigenvectors being placed to the left of the matrix.

INPUT:

- `self` - a square matrix over an exact field. For inexact matrices consult the numerical or symbolic matrix classes.
- `format` - default: `None`
  - `'all'` - attempts to create every eigenspace. This will always be possible for matrices with rational entries.
  - `'galois'` - for each irreducible factor of the characteristic polynomial, a single eigenspace will be output for a single root/eigenvalue for the irreducible factor.
  - `None` - Uses the 'all' format if the base ring is contained in an algebraically closed field which is implemented. Otherwise, uses the 'galois' format.
- `var` - default: 'a' - variable name used to represent elements of the root field of each irreducible factor of the characteristic polynomial. If var='a', then the root fields will be in terms of a0, a1, a2, ..., where the numbering runs across all the irreducible factors of the characteristic polynomial, even for linear factors.
- `algebraic_multiplicity` - default: False - whether or not to include the algebraic multiplicity of each eigenvalue in the output. See the discussion below.

OUTPUT:

If algebraic_multiplicity=False, return a list of pairs (e, V) where e is an eigenvalue of the matrix, and V is the corresponding left eigenspace. For Galois conjugates of eigenvalues, there may be just one representative eigenspace, depending on the `format` keyword.

If algebraic_multiplicity=True, return a list of triples (e, V, n) where e and V are as above and n is the algebraic multiplicity of the eigenvalue.

Warning: Uses a somewhat naive algorithm (simply factors the characteristic polynomial and computes kernels directly over the extension field).

EXAMPLES:

We compute the left eigenspaces of a 3×3 rational matrix. First, we request *all* of the eigenvalues, so the results are in the field of algebraic numbers, *QQbar*. Then we request just one eigenspace per irreducible factor of the characteristic polynomial with the *galois* keyword.

```
sage: A = matrix(QQ,3,3,range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: es = A.eigenspaces_left(format='all'); es
[
(0, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1 -2  1]),
(-1.348469228349535?, Vector space of degree 3 and dimension 1 over Algebraic␣
↪Field
User basis matrix:
[                   1   0.3101020514433644? -0.3797958971132713?]),
(13.34846922834954?, Vector space of degree 3 and dimension 1 over Algebraic␣
↪Field
User basis matrix:
[                 1 1.289897948556636? 1.579795897113272?])
]

sage: es = A.eigenspaces_left(format='galois'); es
[
(0, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1 -2  1]),
(a1, Vector space of degree 3 and dimension 1 over Number Field in a1 with␣
↪defining polynomial x^2 - 12*x - 18
User basis matrix:
[            1 1/15*a1 + 2/5 2/15*a1 - 1/5])
]
sage: es = A.eigenspaces_left(format='galois', algebraic_multiplicity=True);␣
↪es
[
(0, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1 -2  1], 1),
(a1, Vector space of degree 3 and dimension 1 over Number Field in a1 with␣
↪defining polynomial x^2 - 12*x - 18
User basis matrix:
[            1 1/15*a1 + 2/5 2/15*a1 - 1/5], 1)
```

```
]
sage: e, v, n = es[0]; v = v.basis()[0]
sage: delta = e*v - v*A
sage: abs(abs(delta)) < 1e-10
True
```

The same computation, but with implicit base change to a field.

```
sage: A = matrix(ZZ,3,3,range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: A.eigenspaces_left(format='galois')
[
(0, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1 -2  1]),
(a1, Vector space of degree 3 and dimension 1 over Number Field in a1 with
↪defining polynomial x^2 - 12*x - 18
User basis matrix:
[            1 1/15*a1 + 2/5 2/15*a1 - 1/5])
]
```

We compute the left eigenspaces of the matrix of the Hecke operator T_2 on level 43 modular symbols, both with all eigenvalues (the default) and with one subspace per factor.

```
sage: A = ModularSymbols(43).T(2).matrix(); A
[ 3  0  0  0  0  0 -1]
[ 0 -2  1  0  0  0  0]
[ 0 -1  1  1  0 -1  0]
[ 0 -1  0 -1  2 -1  1]
[ 0 -1  0  1  1 -1  1]
[ 0  0 -2  0  2 -2  1]
[ 0  0 -1  0  1  0 -1]
sage: A.base_ring()
Rational Field
sage: f = A.charpoly(); f
x^7 + x^6 - 12*x^5 - 16*x^4 + 36*x^3 + 52*x^2 - 32*x - 48
sage: factor(f)
(x - 3) * (x + 2)^2 * (x^2 - 2)^2
sage: A.eigenspaces_left(algebraic_multiplicity=True)
[
(3, Vector space of degree 7 and dimension 1 over Rational Field
User basis matrix:
[   1    0  1/7    0 -1/7    0 -2/7], 1),
(-2, Vector space of degree 7 and dimension 2 over Rational Field
User basis matrix:
[ 0  1  0  1 -1  1 -1]
[ 0  0  1  0 -1  2 -1], 2),
(-1.414213562373095?, Vector space of degree 7 and dimension 2 over Algebraic
↪Field
User basis matrix:
[                  0                   1                   0
↪-1 0.4142135623730951?                   1                  -1]
[                  0                   0                   1
↪ 0                  -1                   0  2.414213562373095?], 2),
(1.414213562373095?, Vector space of degree 7 and dimension 2 over Algebraic
↪Field
```

```
User basis matrix:
[                    0                    1                    0                    
↪    -1  -2.414213562373095?                    1                   -1]
[                    0                    0                    1                    
↪     0                   -1                    0 -0.4142135623730951?], 2)
]
sage: A.eigenspaces_left(format='galois', algebraic_multiplicity=True)
[
(3, Vector space of degree 7 and dimension 1 over Rational Field
User basis matrix:
[   1    0  1/7    0 -1/7    0 -2/7], 1),
(-2, Vector space of degree 7 and dimension 2 over Rational Field
User basis matrix:
[ 0  1  0  1 -1  1 -1]
[ 0  0  1  0 -1  2 -1], 2),
(a2, Vector space of degree 7 and dimension 2 over Number Field in a2 with
↪defining polynomial x^2 - 2
User basis matrix:
[      0       1       0      -1 -a2 - 1       1      -1]
[      0       0       1       0      -1       0 -a2 + 1], 2)
]
```

Next we compute the left eigenspaces over the finite field of order 11.

```
sage: A = ModularSymbols(43, base_ring=GF(11), sign=1).T(2).matrix(); A
[ 3  9  0  0]
[ 0  9  0  1]
[ 0 10  9  2]
[ 0  9  0  2]
sage: A.base_ring()
Finite Field of size 11
sage: A.charpoly()
x^4 + 10*x^3 + 3*x^2 + 2*x + 1
sage: A.eigenspaces_left(format='galois', var = 'beta')
[
(9, Vector space of degree 4 and dimension 1 over Finite Field of size 11
User basis matrix:
[0 0 1 5]),
(3, Vector space of degree 4 and dimension 1 over Finite Field of size 11
User basis matrix:
[1 6 0 6]),
(beta2, Vector space of degree 4 and dimension 1 over Univariate Quotient
↪Polynomial Ring in beta2 over Finite Field of size 11 with modulus x^2 + 9
User basis matrix:
[           0            1            0 5*beta2 + 10])
]
```

This method is only applicable to exact matrices. The "eigenmatrix" routines for matrices with double-precision floating-point entries (RDF, CDF) are the best alternative. (Since some platforms return eigenvectors that are the negatives of those given here, this one example is not tested here.) There are also "eigenmatrix" routines for matrices with symbolic entries.

```
sage: A = matrix(QQ, 3, 3, range(9))
sage: A.change_ring(RR).eigenspaces_left()
Traceback (most recent call last):
...
NotImplementedError: eigenspaces cannot be computed reliably for inexact
↪rings such as Real Field with 53 bits of precision,
```

```
consult numerical or symbolic matrix classes for other options

sage: em = A.change_ring(RDF).eigenmatrix_left()
sage: eigenvalues = em[0]; eigenvalues.dense_matrix() # abs tol 1e-13
[13.348469228349522                0.0                 0.0]
[               0.0 -1.348469228349534                 0.0]
[               0.0                0.0                 0.0]
sage: eigenvectors = em[1]; eigenvectors # not tested
[ 0.440242867...  0.567868371...  0.695493875...]
[ 0.897878732...  0.278434036... -0.341010658...]
[ 0.408248290... -0.816496580...  0.408248290...]

sage: x, y = var('x y')
sage: S = matrix([[x, y], [y, 3*x^2]])
sage: em = S.eigenmatrix_left()
sage: eigenvalues = em[0]; eigenvalues
[3/2*x^2 + 1/2*x - 1/2*sqrt(9*x^4 - 6*x^3 + x^2 + 4*y^2)                                                       ␣
↪                                                        0]
[                                                        0 3/2*x^2 + 1/2*x + 1/
↪2*sqrt(9*x^4 - 6*x^3 + x^2 + 4*y^2)]
sage: eigenvectors = em[1]; eigenvectors
[                                                     1 1/2*(3*x^2 - x -␣
↪sqrt(9*x^4 - 6*x^3 + x^2 + 4*y^2))/y]
[                                                     1 1/2*(3*x^2 - x +␣
↪sqrt(9*x^4 - 6*x^3 + x^2 + 4*y^2))/y]
```

A request for `'all'` the eigenvalues, when it is not possible, will raise an error. Using the `'galois'` format option is more likely to be successful.

```
sage: F.<b> = FiniteField(11^2)
sage: A = matrix(F, [[b + 1, b + 1], [10*b + 4, 5*b + 4]])
sage: A.eigenspaces_left(format='all')
Traceback (most recent call last):
...
NotImplementedError: unable to construct eigenspaces for eigenvalues outside␣
↪the base field,
try the keyword option: format='galois'

sage: A.eigenspaces_left(format='galois')
[
(a0, Vector space of degree 2 and dimension 1 over Univariate Quotient␣
↪Polynomial Ring in a0 over Finite Field in b of size 11^2 with modulus x^2␣
↪+ (5*b + 6)*x + 8*b + 10
User basis matrix:
[               1 6*b*a0 + 3*b + 1])
]
```

eigenspaces_right (*format='all'*, *var='a'*, *algebraic_multiplicity=False*)

Compute the right eigenspaces of a matrix.

Note that `eigenspaces_right()` and `right_eigenspaces()` are identical methods. Here "right" refers to the eigenvectors being placed to the right of the matrix.

INPUT:

- `self` - a square matrix over an exact field. For inexact matrices consult the numerical or symbolic matrix classes.
- `format` - default: `None`

    –'all' - attempts to create every eigenspace. This will always be possible for matrices with rational entries.

    –'galois' - for each irreducible factor of the characteristic polynomial, a single eigenspace will be output for a single root/eigenvalue for the irreducible factor.

    –None - Uses the 'all' format if the base ring is contained in an algebraically closed field which is implemented. Otherwise, uses the 'galois' format.

- var - default: 'a' - variable name used to represent elements of the root field of each irreducible factor of the characteristic polynomial. If var='a', then the root fields will be in terms of a0, a1, a2, ..., where the numbering runs across all the irreducible factors of the characteristic polynomial, even for linear factors.

- algebraic_multiplicity - default: False - whether or not to include the algebraic multiplicity of each eigenvalue in the output. See the discussion below.

OUTPUT:

If algebraic_multiplicity=False, return a list of pairs (e, V) where e is an eigenvalue of the matrix, and V is the corresponding left eigenspace. For Galois conjugates of eigenvalues, there may be just one representative eigenspace, depending on the format keyword.

If algebraic_multiplicity=True, return a list of triples (e, V, n) where e and V are as above and n is the algebraic multiplicity of the eigenvalue.

> **Warning:** Uses a somewhat naive algorithm (simply factors the characteristic polynomial and computes kernels directly over the extension field).

EXAMPLES:

Right eigenspaces are computed from the left eigenspaces of the transpose of the matrix. As such, there is a greater collection of illustrative examples at the *eigenspaces_left()* .

We compute the right eigenspaces of a 3×3 rational matrix.

```
sage: A = matrix(QQ, 3 ,3, range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: A.eigenspaces_right()
[
(0, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1 -2  1]),
(-1.348469228349535?, Vector space of degree 3 and dimension 1 over Algebraic␣
↪Field
User basis matrix:
[                    1   0.1303061543300932? -0.7393876913398137?]),
(13.34846922834954?, Vector space of degree 3 and dimension 1 over Algebraic␣
↪Field
User basis matrix:
[                  1 3.069693845669907? 5.139387691339814?])
]
sage: es = A.eigenspaces_right(format='galois'); es
[
(0, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1 -2  1]),
```

```
(a1, Vector space of degree 3 and dimension 1 over Number Field in a1 with␣
↪defining polynomial x^2 - 12*x - 18
User basis matrix:
[           1 1/5*a1 + 2/5 2/5*a1 - 1/5])
]
sage: es = A.eigenspaces_right(format='galois', algebraic_multiplicity=True);␣
↪es
[
(0, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1 -2  1], 1),
(a1, Vector space of degree 3 and dimension 1 over Number Field in a1 with␣
↪defining polynomial x^2 - 12*x - 18
User basis matrix:
[           1 1/5*a1 + 2/5 2/5*a1 - 1/5], 1)
]
sage: e, v, n = es[0]; v = v.basis()[0]
sage: delta = v*e - A*v
sage: abs(abs(delta)) < 1e-10
True
```

The same computation, but with implicit base change to a field:

```
sage: A = matrix(ZZ, 3, range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: A.eigenspaces_right(format='galois')
[
(0, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1 -2  1]),
(a1, Vector space of degree 3 and dimension 1 over Number Field in a1 with␣
↪defining polynomial x^2 - 12*x - 18
User basis matrix:
[           1 1/5*a1 + 2/5 2/5*a1 - 1/5])
]
```

This method is only applicable to exact matrices. The "eigenmatrix" routines for matrices with double-precision floating-point entries (`RDF` , `CDF`) are the best alternative. (Since some platforms return eigenvectors that are the negatives of those given here, this one example is not tested here.) There are also "eigenmatrix" routines for matrices with symbolic entries.

```
sage: B = matrix(RR, 3, 3, range(9))
sage: B.eigenspaces_right()
Traceback (most recent call last):
...
NotImplementedError: eigenspaces cannot be computed reliably for inexact␣
↪rings such as Real Field with 53 bits of precision,
consult numerical or symbolic matrix classes for other options

sage: em = B.change_ring(RDF).eigenmatrix_right()
sage: eigenvalues = em[0]; eigenvalues.dense_matrix() # abs tol 1e-13
[13.348469228349522                0.0                0.0]
[               0.0 -1.348469228349534                0.0]
[               0.0                0.0                0.0]
sage: eigenvectors = em[1]; eigenvectors # not tested
[ 0.164763817...  0.799699663...  0.408248290...]
```

```
[ 0.505774475...  0.104205787... -0.816496580...]
[ 0.846785134... -0.591288087...  0.408248290...]

sage: x, y = var('x y')
sage: S = matrix([[x, y], [y, 3*x^2]])
sage: em = S.eigenmatrix_right()
sage: eigenvalues = em[0]; eigenvalues
[3/2*x^2 + 1/2*x - 1/2*sqrt(9*x^4 - 6*x^3 + x^2 + 4*y^2)                                   ␣
↪                                  0]
[                                                      0 3/2*x^2 + 1/2*x + 1/
↪2*sqrt(9*x^4 - 6*x^3 + x^2 + 4*y^2)]
sage: eigenvectors = em[1]; eigenvectors
[                                                    1                                   ␣
↪                              1]
[1/2*(3*x^2 - x - sqrt(9*x^4 - 6*x^3 + x^2 + 4*y^2))/y 1/2*(3*x^2 - x +␣
↪sqrt(9*x^4 - 6*x^3 + x^2 + 4*y^2))/y]
```

eigenvalues (*extend=True*)

Return a sequence of the eigenvalues of a matrix, with multiplicity. If the eigenvalues are roots of polynomials in QQ, then QQbar elements are returned that represent each separate root.

If the option extend is set to False, only eigenvalues in the base ring are considered.

EXAMPLES:

```
sage: a = matrix(ZZ, 4, range(16)); a
[ 0  1  2  3]
[ 4  5  6  7]
[ 8  9 10 11]
[12 13 14 15]
sage: sorted(a.eigenvalues(), reverse=True)
[32.46424919657298?, 0, 0, -2.464249196572981?]
```

```
sage: a=matrix([(1, 9, -1, -1), (-2, 0, -10, 2), (-1, 0, 15, -2), (0, 1, 0, -
↪1)])
sage: a.eigenvalues()
[-0.9386318578049146?, 15.50655435353258?, 0.2160387521361705? - 4.
↪713151979747493?*I, 0.2160387521361705? + 4.713151979747493?*I]
```

A symmetric matrix a+a.transpose() should have real eigenvalues

```
sage: b=a+a.transpose()
sage: ev = b.eigenvalues(); ev
[-8.35066086057957?, -1.107247901349379?, 5.718651326708515?, 33.
↪73925743522043?]
```

The eigenvalues are elements of QQbar, so they really represent exact roots of polynomials, not just approximations.

```
sage: e = ev[0]; e
-8.35066086057957?
sage: p = e.minpoly(); p
x^4 - 30*x^3 - 171*x^2 + 1460*x + 1784
sage: p(e) == 0
True
```

To perform computations on the eigenvalue as an element of a number field, you can always convert back to a number field element.

```
sage: e.as_number_field_element()
(Number Field in a with defining polynomial y^4 - 2*y^3 - 507*y^2 - 3972*y -
→4264,
a + 7,
Ring morphism:
  From: Number Field in a with defining polynomial y^4 - 2*y^3 - 507*y^2 -
→3972*y - 4264
  To:   Algebraic Real Field
  Defn: a |--> -15.35066086057957?)
```

Notice the effect of the extend option.

```
sage: M=matrix(QQ,[[0,-1,0],[1,0,0],[0,0,2]])
sage: M.eigenvalues()
[2, -1*I, 1*I]
sage: M.eigenvalues(extend=False)
[2]
```

The method also works for matrices over finite fields:

```
sage: M = matrix(GF(3), [[0,1,1],[1,2,0],[2,0,1]])
sage: ev = sorted(M.eigenvalues()); ev
[2*z3, 2*z3 + 1, 2*z3 + 2]
```

Similarly as in the case of QQbar, the eigenvalues belong to some algebraic closure but they can be converted to elements of a finite field:

```
sage: e = ev[0]
sage: e.parent()
Algebraic closure of Finite Field of size 3
sage: e.as_finite_field_element()
(Finite Field in z3 of size 3^3, 2*z3, Ring morphism:
  From: Finite Field in z3 of size 3^3
  To:   Algebraic closure of Finite Field of size 3
  Defn: z3 |--> z3)
```

eigenvectors_left (*extend=True*)

Compute the left eigenvectors of a matrix.

For each distinct eigenvalue, returns a list of the form (e,V,n) where e is the eigenvalue, V is a list of eigenvectors forming a basis for the corresponding left eigenspace, and n is the algebraic multiplicity of the eigenvalue.

If the option extend is set to False, then only the eigenvalues that live in the base ring are considered.

EXAMPLES: We compute the left eigenvectors of a 3×3 rational matrix.

```
sage: A = matrix(QQ,3,3,range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: es = A.eigenvectors_left(); es
[(0, [
(1, -2, 1)
], 1),
(-1.348469228349535?, [(1, 0.3101020514433644?, -0.3797958971132713?)], 1),
(13.34846922834954?, [(1, 1.289897948556636?, 1.579795897113272?)], 1)]
sage: eval, [evec], mult = es[0]
sage: delta = eval*evec - evec*A
```

```
sage: abs(abs(delta)) < 1e-10
True
```

Notice the difference between considering ring extensions or not.

```
sage: M=matrix(QQ,[[0,-1,0],[1,0,0],[0,0,2]])
sage: M.eigenvectors_left()
[(2, [
(0, 0, 1)
], 1), (-1*I, [(1, -1*I, 0)], 1), (1*I, [(1, 1*I, 0)], 1)]
sage: M.eigenvectors_left(extend=False)
[(2, [
(0, 0, 1)
], 1)]
```

eigenvectors_right (*extend=True*)

Compute the right eigenvectors of a matrix.

For each distinct eigenvalue, returns a list of the form (e,V,n) where e is the eigenvalue, V is a list of eigenvectors forming a basis for the corresponding right eigenspace, and n is the algebraic multiplicity of the eigenvalue. If `extend = True` (the default), this will return eigenspaces over the algebraic closure of the base field where this is implemented; otherwise it will restrict to eigenvalues in the base field.

EXAMPLES: We compute the right eigenvectors of a 3×3 rational matrix.

```
sage: A = matrix(QQ,3,3,range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: es = A.eigenvectors_right(); es
[(0, [
(1, -2, 1)
], 1),
(-1.348469228349535?, [(1, 0.1303061543300932?, -0.7393876913398137?)], 1),
(13.34846922834954?, [(1, 3.069693845669907?, 5.139387691339814?)], 1)]
sage: A.eigenvectors_right(extend=False)
[(0, [
(1, -2, 1)
], 1)]
sage: eval, [evec], mult = es[0]
sage: delta = eval*evec - A*evec
sage: abs(abs(delta)) < 1e-10
True
```

elementary_divisors ( )

If self is a matrix over a principal ideal domain R, return elements d_i for $1 \leq i \leq k = \min(r, s)$ where r and s are the number of rows and columns of self, such that the cokernel of self is isomorphic to

$$R/(d_1) \oplus R/(d_2) \oplus R/(d_k)$$

with $d_i \mid d_{i+1}$ for all i. These are the diagonal entries of the Smith form of self (see *smith_form()*).

EXAMPLES:

```
sage: OE.<w> = EquationOrder(x^2 - x + 2)
sage: m = Matrix([ [1, w],[w,7]])
sage: m.elementary_divisors()
[1, -w + 9]
```

See also:

smith_form()

elementwise_product (*right*)

Returns the elementwise product of two matrices of the same size (also known as the Hadamard product).

INPUT:

- `right` - the right operand of the product. A matrix of the same size as `self` such that multiplication of elements of the base rings of `self` and `right` is defined, once Sage's coercion model is applied. If the matrices have different sizes, or if multiplication of individual entries cannot be achieved, a `TypeError` will result.

OUTPUT:

A matrix of the same size as `self` and `right`. The entry in location (i, j) of the output is the product of the two entries in location (i, j) of `self` and `right` (in that order).

The parent of the result is determined by Sage's coercion model. If the base rings are identical, then the result is dense or sparse according to this property for the left operand. If the base rings must be adjusted for one, or both, matrices then the result will be sparse only if both operands are sparse. No subdivisions are present in the result.

If the type of the result is not to your liking, or the ring could be "tighter," adjust the operands with *change_ring()*. Adjust sparse versus dense inputs with the methods *sparse_matrix()* and *dense_matrix()*.

EXAMPLES:

```
sage: A = matrix(ZZ, 2, 3, range(6))
sage: B = matrix(QQ, 2, 3, [5, 1/3, 2/7, 11/2, -3/2, 8])
sage: C = A.elementwise_product(B)
sage: C
[   0  1/3  4/7]
[33/2   -6   40]
sage: C.parent()
Full MatrixSpace of 2 by 3 dense matrices over Rational Field
```

Notice the base ring of the results in the next two examples.

```
sage: D = matrix(ZZ['x'],2,[1+x^2,2,3,4-x])
sage: E = matrix(QQ,2,[1,2,3,4])
sage: F = D.elementwise_product(E)
sage: F
[  x^2 + 1          4]
[        9 -4*x + 16]
sage: F.parent()
Full MatrixSpace of 2 by 2 dense matrices over Univariate Polynomial Ring in
x over Rational Field
```

```
sage: G = matrix(GF(3),2,[0,1,2,2])
sage: H = matrix(ZZ,2,[1,2,3,4])
sage: J = G.elementwise_product(H)
sage: J
[0 2]
[0 2]
sage: J.parent()
Full MatrixSpace of 2 by 2 dense matrices over Finite Field of size 3
```

Non-commutative rings behave as expected. These are the usual quaternions.

```
sage: R.<i,j,k> = QuaternionAlgebra(-1, -1)
sage: A = matrix(R, 2, [1,i,j,k])
sage: B = matrix(R, 2, [i,i,i,i])
sage: A.elementwise_product(B)
[ i -1]
[-k  j]
sage: B.elementwise_product(A)
[ i -1]
[ k -j]
```

Input that is not a matrix will raise an error.

```
sage: A = random_matrix(ZZ,5,10,x=20)
sage: A.elementwise_product(vector(ZZ, [1,2,3,4]))
Traceback (most recent call last):
...
TypeError: operand must be a matrix, not an element of Ambient free module of␣
↪rank 4 over the principal ideal domain Integer Ring
```

Matrices of different sizes for operands will raise an error.

```
sage: A = random_matrix(ZZ,5,10,x=20)
sage: B = random_matrix(ZZ,10,5,x=40)
sage: A.elementwise_product(B)
Traceback (most recent call last):
...
TypeError: incompatible sizes for matrices from: Full MatrixSpace of 5 by 10␣
↪dense matrices over Integer Ring and Full MatrixSpace of 10 by 5 dense␣
↪matrices over Integer Ring
```

Some pairs of rings do not have a common parent where multiplication makes sense. This will raise an error.

```
sage: A = matrix(QQ, 3, 2, range(6))
sage: B = matrix(GF(3), 3, [2]*6)
sage: A.elementwise_product(B)
Traceback (most recent call last):
...
TypeError: no common canonical parent for objects with parents: 'Full␣
↪MatrixSpace of 3 by 2 dense matrices over Rational Field' and 'Full␣
↪MatrixSpace of 3 by 2 dense matrices over Finite Field of size 3'
```

We illustrate various combinations of sparse and dense matrices. Notice how if base rings are unequal, both operands must be sparse to get a sparse result.

```
sage: A = matrix(ZZ, 5, 6, range(30), sparse=False)
sage: B = matrix(ZZ, 5, 6, range(30), sparse=True)
sage: C = matrix(QQ, 5, 6, range(30), sparse=True)
sage: A.elementwise_product(C).is_dense()
True
sage: B.elementwise_product(C).is_sparse()
True
sage: A.elementwise_product(B).is_dense()
True
sage: B.elementwise_product(A).is_dense()
True
```

AUTHOR:

- Rob Beezer (2009-07-13)

exp ( )

Calculate the exponential of this matrix X, which is the matrix

$$e^X = \sum_{k=0}^{\infty} \frac{X^k}{k!}.$$

This function depends on maxima's matrix exponentiation function, which does not deal well with floating point numbers. If the matrix has floating point numbers, they will be rounded automatically to rational numbers during the computation. If you want approximations to the exponential that are calculated numerically, you may get better results by first converting your matrix to RDF or CDF, as shown in the last example.

EXAMPLES:

```
sage: a=matrix([[1,2],[3,4]])
sage: a.exp()
[-1/22*((sqrt(33) - 11)*e^sqrt(33) - sqrt(33) - 11)*e^(-1/2*sqrt(33) + 5/2)
↪           2/33*(sqrt(33)*e^sqrt(33) - sqrt(33))*e^(-1/2*sqrt(33) + 5/2)]
[            1/11*(sqrt(33)*e^sqrt(33) - sqrt(33))*e^(-1/2*sqrt(33) + 5/2)
↪1/22*((sqrt(33) + 11)*e^sqrt(33) - sqrt(33) + 11)*e^(-1/2*sqrt(33) + 5/2)]

sage: type(a.exp())
<type 'sage.matrix.matrix_symbolic_dense.Matrix_symbolic_dense'>

sage: a=matrix([[1/2,2/3],[3/4,4/5]])
sage: a.exp()
[-1/418*((3*sqrt(209) - 209)*e^(1/10*sqrt(209)) - 3*sqrt(209) - 209)*e^(-1/
↪20*sqrt(209) + 13/20)                   20/627*(sqrt(209)*e^(1/
↪10*sqrt(209)) - sqrt(209))*e^(-1/20*sqrt(209) + 13/20)]
[             15/418*(sqrt(209)*e^(1/10*sqrt(209)) - sqrt(209))*e^(-1/
↪20*sqrt(209) + 13/20)  1/418*((3*sqrt(209) + 209)*e^(1/10*sqrt(209)) -
↪3*sqrt(209) + 209)*e^(-1/20*sqrt(209) + 13/20)]

sage: a=matrix(RR,[[1,pi.n()],[1e2,1e-2]])
sage: a.exp()
[ 1/11882424341266*((11*sqrt(227345670387496707609) + 5941212170633)*e^(3/
↪1275529100*sqrt(227345670387496707609)) - 11*sqrt(227345670387496707609) +
↪5941212170633)*e^(-3/2551058200*sqrt(227345670387496707609) + 101/200)
↪                        445243650/
↪75781890129165569203*(sqrt(227345670387496707609)*e^(3/
↪1275529100*sqrt(227345670387496707609)) - sqrt(227345670387496707609))*e^(-
↪3/2551058200*sqrt(227345670387496707609) + 101/200)]
[                                   10000/
↪53470909535697*(sqrt(227345670387496707609)*e^(3/
↪1275529100*sqrt(227345670387496707609)) - sqrt(227345670387496707609))*e^(-
↪3/2551058200*sqrt(227345670387496707609) + 101/200) -1/
↪11882424341266*((11*sqrt(227345670387496707609) - 5941212170633)*e^(3/
↪1275529100*sqrt(227345670387496707609)) - 11*sqrt(227345670387496707609) -
↪5941212170633)*e^(-3/2551058200*sqrt(227345670387496707609) + 101/200)]
sage: a.change_ring(RDF).exp()  # rel tol 1e-14
[42748127.31532951 7368259.244159399]
[234538976.1381042 40426191.45156228]
```

extended_echelon_form (*subdivide=False*, ***kwds*)

Returns the echelon form of `self` augmented with an identity matrix.

INPUT:

- `subdivide` - default: `False` - determines if the returned matrix is subdivided. See the description of the (output) below for details.
- `kwds` - additional keywords that can be passed to the method that computes the echelon form.

OUTPUT:

If A is an $m \times n$ matrix, add the m columns of an $m \times m$ identity matrix to the right of `self`. Then row-reduce this $m \times (n + m)$ matrix. This matrix is returned as an immutable matrix.

If `subdivide` is `True` then the returned matrix has a single division among the columns and a single division among the rows. The column subdivision has n columns to the left and m columns to the right. The row division separates the non-zero rows from the zero rows, when restricted to the first n columns.

For a nonsingular matrix the final m columns of the extended echelon form are the inverse of `self`. For a matrix of any size, the final m columns provide a matrix that transforms `self` to echelon form when it multiplies `self` from the left. When the base ring is a field, the uniqueness of reduced row-echelon form implies that this transformation matrix can be taken as the coefficients giving a canonical set of linear combinations of the rows of `self` that yield reduced row-echelon form.

When subdivided as described above, and again over a field, the parts of the subdivision in the upper-left corner and lower-right corner satisfy several interesting relationships with the row space, column space, left kernel and right kernel of `self`. See the examples below.

Note: This method returns an echelon form. If the base ring is not a field, no atttempt is made to move to the fraction field. See an example below where the base ring is changed manually.

EXAMPLES:

The four relationships at the end of this example hold in general.

```
sage: A = matrix(QQ, [[2, -1, 7, -1, 0, -3],
....:                 [-1, 1, -5, 3, 4, 4],
....:                 [2, -1, 7, 0, 2, -2],
....:                 [2, 0, 4, 3, 6, 1],
....:                 [2, -1, 7, 0, 2, -2]])
sage: E = A.extended_echelon_form(subdivide=True); E
[   1    0    2    0    0   -1|   0   -1    0    1   -1]
[   0    1   -3    0   -2    0|   0   -2    0    2   -3]
[   0    0    0    1    2    1|   0  2/3    0 -1/3  2/3]
[-----------------------------+------------------------]
[   0    0    0    0    0    0|   1  2/3    0 -1/3 -1/3]
[   0    0    0    0    0    0|   0    0    1    0   -1]
sage: J = E.matrix_from_columns(range(6,11)); J
[   0   -1    0    1   -1]
[   0   -2    0    2   -3]
[   0  2/3    0 -1/3  2/3]
[   1  2/3    0 -1/3 -1/3]
[   0    0    1    0   -1]
sage: J*A == A.rref()
True
sage: C = E.subdivision(0,0); C
[ 1  0  2  0  0 -1]
[ 0  1 -3  0 -2  0]
[ 0  0  0  1  2  1]
sage: L = E.subdivision(1,1); L
[   1  2/3    0 -1/3 -1/3]
```

```
[    0     0     1     0    -1]
sage: A.right_kernel() == C.right_kernel()
True
sage: A.row_space() == C.row_space()
True
sage: A.column_space() == L.right_kernel()
True
sage: A.left_kernel() == L.row_space()
True
```

For a nonsingular matrix, the right half of the extended echelon form is the inverse matrix.

```
sage: B = matrix(QQ, [[1,3,4], [1,4,4], [0,-2,-1]])
sage: E = B.extended_echelon_form()
sage: J = E.matrix_from_columns(range(3,6)); J
[-4  5  4]
[-1  1  0]
[ 2 -2 -1]
sage: J == B.inverse()
True
```

The result is in echelon form, so if the base ring is not a field, the leading entry of each row may not be 1. But you can easily change to the fraction field if necessary.

```
sage: A = matrix(ZZ, [[16, 20, 4, 5, -4, 13, 5],
....:                 [10, 13, 3, -3, 7, 11, 6],
....:                 [-12, -15, -3, -3, 2, -10, -4],
....:                 [10, 13, 3, 3, -1, 9, 4],
....:                 [4, 5, 1, 8, -10, 1, -1]])
sage: E = A.extended_echelon_form(subdivide=True); E
[ 2  0 -2  2 -9 -3 -4| 0  4 -3 -9  4]
[ 0  1  1  2  0  1  1| 0  1  2  1  1]
[ 0  0  0  3 -4 -1 -1| 0  3  1 -3  3]
[--------------------+--------------]
[ 0  0  0  0  0  0  0| 1  6  3 -6  5]
[ 0  0  0  0  0  0  0| 0  7  2 -7  6]
sage: J = E.matrix_from_columns(range(7,12)); J
[ 0  4 -3 -9  4]
[ 0  1  2  1  1]
[ 0  3  1 -3  3]
[ 1  6  3 -6  5]
[ 0  7  2 -7  6]
sage: J*A == A.echelon_form()
True
sage: B = A.change_ring(QQ)
sage: B.extended_echelon_form(subdivide=True)
[     1      0     -1      0  -19/6   -7/6   -5/3|     0      0 -89/42   -5/2␣
↪   1/7]
[     0      1      1      0    8/3    5/3    5/3|     0      0  34/21      2␣
↪  -1/7]
[     0      0      0      1   -4/3   -1/3   -1/3|     0      0   1/21      0␣
↪   1/7]
[------------------------------------------------+---------------------------
↪------]
[     0      0      0      0      0      0      0|     1      0    9/7      0␣
↪  -1/7]
[     0      0      0      0      0      0      0|     0      1    2/7     -1␣
↪   6/7]
```

Subdivided, or not, the result is immutable, so make a copy if you want to make changes.

```
sage: A = matrix(FiniteField(7), [[2,0,3], [5,5,3], [5,6,5]])
sage: E = A.extended_echelon_form()
sage: E.is_mutable()
False
sage: F = A.extended_echelon_form(subdivide=True)
sage: F
[1 0 0|0 4 6]
[0 1 0|4 2 2]
[0 0 1|5 2 3]
[-----+-----]
sage: F.is_mutable()
False
sage: G = copy(F)
sage: G.subdivide([],[]); G
[1 0 0 0 4 6]
[0 1 0 4 2 2]
[0 0 1 5 2 3]
```

If you want to determine exactly which algorithm is used to compute the echelon form, you can add additional keywords to pass on to the `echelon_form()` routine employed on the augmented matrix. Sending the flag `include_zero_rows` is a bit silly, since the extended echelon form will never have any zero rows.

```
sage: A = matrix(ZZ, [[1,2], [5,0], [5,9]])
sage: E = A.extended_echelon_form(algorithm='padic', include_zero_rows=False)
sage: E
[  1   0  36   1  -8]
[  0   1   5   0  -1]
[  0   0  45   1 -10]
```

AUTHOR:

•Rob Beezer (2011-02-02)

fcp (*var='x'*)

Return the factorization of the characteristic polynomial of self.

INPUT:

•`var` - (default: 'x') name of variable of charpoly

EXAMPLES:

```
sage: M = MatrixSpace(QQ,3,3)
sage: A = M([1,9,-7,4/5,4,3,6,4,3])
sage: A.fcp()
x^3 - 8*x^2 + 209/5*x - 286
sage: A = M([3, 0, -2, 0, -2, 0, 0, 0, 0])
sage: A.fcp('T')
(T - 3) * T * (T + 2)
```

find (*f*, *indices=False*)

Find elements in this matrix satisfying the constraints in the function f. The function is evaluated on each element of the matrix .

INPUT:

•`f` - a function that is evaluated on each element of this matrix.

- `indices` - whether or not to return the indices and elements of this matrix that satisfy the function.

OUTPUT: If `indices` is not specified, return a matrix with 1 where f is satisfied and 0 where it is not. If `indices` is specified, return a dictionary containing the elements of this matrix satisfying f.

EXAMPLES:

```
sage: M = matrix(4,3,[1, -1/2, -1, 1, -1, -1/2, -1, 0, 0, 2, 0, 1])
sage: M.find(lambda entry:entry==0)
[0 0 0]
[0 0 0]
[0 1 1]
[0 1 0]
```

```
sage: M.find(lambda u:u<0)
[0 1 1]
[0 1 1]
[1 0 0]
[0 0 0]
```

```
sage: M = matrix(4,3,[1, -1/2, -1, 1, -1, -1/2, -1, 0, 0, 2, 0, 1])
sage: len(M.find(lambda u:u<1 and u>-1,indices=True))
5
```

```
sage: M.find(lambda u:u!=1/2)
[1 1 1]
[1 1 1]
[1 1 1]
[1 1 1]
```

```
sage: M.find(lambda u:u>1.2)
[0 0 0]
[0 0 0]
[0 0 0]
[1 0 0]
```

```
sage: sorted(M.find(lambda u:u!=0,indices=True).keys()) == M.nonzero_
↪positions()
True
```

get_subdivisions ( )

Returns the current subdivision of self.

EXAMPLES:

```
sage: M = matrix(5, 5, range(25))
sage: M.subdivisions()
([], [])
sage: M.subdivide(2,3)
sage: M.subdivisions()
([2], [3])
sage: N = M.parent()(1)
sage: N.subdivide(M.subdivisions()); N
[1 0 0|0 0]
[0 1 0|0 0]
[-----+---]
[0 0 1|0 0]
```

```
[0 0 0|1 0]
[0 0 0|0 1]
```

gram_schmidt (*orthonormal=False*)

Performs Gram-Schmidt orthogonalization on the rows of the matrix, returning a new matrix and a matrix accomplishing the transformation.

INPUT:

- `self` - a matrix whose rows are to be orthogonalized.
- `orthonormal` - default: `False` - if `True` the returned orthogonal vectors are unit vectors. This keyword is ignored if the matrix is over `RDF` or `CDF` and the results are always orthonormal.

OUTPUT:

A pair of matrices, `G` and `M` such that if `A` represents `self`, where the parenthetical properties occur when `orthonormal = True`:

- `A = M*G`
- The rows of `G` are an orthogonal (resp. orthonormal) set of vectors.
- `G` times the conjugate-transpose of `G` is a diagonal (resp. identity) matrix.
- The row space of `G` equals the row space of `A`.
- `M` is a full-rank matrix with zeros above the diagonal.

For exact rings, any zero vectors produced (when the original vectors are linearly dependent) are not output, thus the orthonormal set is linearly independent, and thus a basis for the row space of the original matrix.

Any notion of a Gram-Schmidt procedure requires that the base ring of the matrix has a fraction field implemented. In order to arrive at an orthonormal set, it must be possible to construct square roots of the elements of the base field. In Sage, your best option is the field of algebraic numbers, `QQbar`, which properly contains the rationals and number fields.

If you have an approximate numerical matrix, then this routine requires that your base field be the real and complex double-precision floating point numbers, `RDF` and `CDF`. In this case, the matrix is treated as having full rank, as no attempt is made to recognize linear dependence with approximate calculations.

EXAMPLES:

Inexact Rings, Numerical Matrices:

First, the inexact rings, `CDF` and `RDF`.

```
sage: A = matrix(CDF, [[ 0.6454 + 0.7491*I, -0.8662 + 0.1489*I,  0.7656 - 0.
↪00344*I],
....:                  [-0.2913 + 0.8057*I,  0.8321 + 0.8170*I, -0.6744 + 0.
↪9248*I],
....:                  [ 0.2554 + 0.3517*I, -0.4454 - 0.1715*I,  0.8325 - 0.
↪6282*I]])
sage: G, M = A.gram_schmidt()
sage: G.round(6)  # random signs
[-0.422243 - 0.490087*I  0.566698 - 0.097416*I -0.500882 + 0.002251*I]
[-0.057002 - 0.495035*I  -0.35059 - 0.625323*I  0.255514 - 0.415284*I]
[ 0.394105 - 0.421778*I -0.392266 - 0.039345*I  -0.352905 + 0.62195*I]
sage: M.round(6)  # random
[             -1.528503                    0.0                    0.0]
[  0.459974 - 0.40061*I              -1.741233                    0.0]
[-0.934304 + 0.148868*I   0.54833 + 0.073202*I              -0.550725]
```

```
sage: (A - M*G).zero_at(10^-12)
[0.0 0.0 0.0]
[0.0 0.0 0.0]
[0.0 0.0 0.0]
sage: (G*G.conjugate_transpose())  # random
[0.9999999999999999                0.0                0.0]
[               0.0 0.9999999999999997                0.0]
[               0.0                0.0                1.0]
```

A rectangular matrix. Note that the `orthonormal` keyword is ignored in these cases.

```
sage: A = matrix(RDF, [[-0.978325, -0.751994, 0.925305, -0.200512, 0.420458],
....:                  [-0.474877, -0.983403, 0.089836,  0.132218, 0.672965]])
sage: G, M = A.gram_schmidt(orthonormal=False)
sage: G.round(6).zero_at(10^-6)
[-0.607223 -0.466745  0.574315 -0.124453  0.260968]
[ 0.123203 -0.617909 -0.530578  0.289773  0.487368]
sage: M.round(6).zero_at(10^-6)
[1.611147      0.0]
[0.958116 0.867778]
sage: (A-M*G).zero_at(10^-12)
[0.0 0.0 0.0 0.0 0.0]
[0.0 0.0 0.0 0.0 0.0]
sage: (G*G.transpose()).round(6).zero_at(10^-6)
[1.0 0.0]
[0.0 1.0]
```

Even though a set of vectors may be linearly dependent, no effort is made to decide when a zero vector is really the result of a relation of linear dependence. So in this regard, input matrices are treated as being of full rank. Try one of the base rings that provide exact results if you need exact results.

```
sage: entries = [[1,1,2], [2,1,3], [3,1,4]]
sage: A = matrix(QQ, entries)
sage: A.rank()
2
sage: B = matrix(RDF, entries)
sage: G, M = B.gram_schmidt()
sage: G.round(6)  # random signs
[-0.408248 -0.408248 -0.816497]
[ 0.707107 -0.707107      -0.0]
[ -0.57735  -0.57735   0.57735]
sage: M.round(10)  # random
[-2.4494897428          0.0          0.0]
[-3.6742346142  0.7071067812          0.0]
[-4.8989794856  1.4142135624          0.0]
sage: (A - M*G).zero_at(1e-14)
[0.0 0.0 0.0]
[0.0 0.0 0.0]
[0.0 0.0 0.0]
sage: (G*G.transpose())  # abs tol 1e-14
[0.9999999999999997                0.0                0.0]
[               0.0 0.9999999999999998                0.0]
[               0.0                0.0                1.0]
```

Exact Rings, Orthonormalization:

To scale a vector to unit length requires taking a square root, which often takes us outside the base ring. For the integers and the rationals, the field of algebraic numbers (`QQbar`) is big enough to contain what we need, but the price is that the computations are very slow, hence mostly of value for small cases or instruction. Now we need to use the `orthonormal` keyword.

```
sage: A = matrix(QQbar, [[6, -8,  1],
....:                    [4,  1,  3],
....:                    [6,  3,  3],
....:                    [7,  1, -5],
....:                    [7, -3,  5]])
sage: G, M = A.gram_schmidt(orthonormal=True)
sage: G
[ 0.5970223141259934? -0.7960297521679913? 0.09950371902099891?]
[ 0.6063218341690895?  0.5289635311888953?  0.5937772444966257?]
[ 0.5252981913594170?  0.2941669871612735?  -0.798453250866314?]
sage: M
[ 10.04987562112089?                    0                    0]
[ 1.890570661398980?  4.735582601355131?                    0]
[ 1.492555785314984?  7.006153332071100?  1.638930357041381?]
[ 2.885607851608969?  1.804330147889395?  7.963520581008761?]
[ 7.064764050490923?  5.626248468100069? -1.197679876299471?]
sage: M*G-A
[0 0 0]
[0 0 0]
[0 0 0]
[0 0 0]
[0 0 0]
sage: (G*G.transpose()-identity_matrix(3)).norm() < 10^-10
True
sage: G.row_space() == A.row_space()
True
```

After trac ticket #14047, the matrix can also be over the algebraic reals `AA` :

```
sage: A = matrix(AA, [[6, -8,  1],
....:                 [4,  1,  3],
....:                 [6,  3,  3],
....:                 [7,  1, -5],
....:                 [7, -3,  5]])
sage: G, M = A.gram_schmidt(orthonormal=True)
sage: G
[ 0.5970223141259934? -0.7960297521679913? 0.09950371902099891?]
[ 0.6063218341690895?  0.5289635311888953?  0.5937772444966257?]
[ 0.5252981913594170?  0.2941669871612735?  -0.798453250866314?]
sage: M
[ 10.04987562112089?                    0                    0]
[ 1.890570661398980?  4.735582601355131?                    0]
[ 1.492555785314984?  7.006153332071100?  1.638930357041381?]
[ 2.885607851608969?  1.804330147889395?  7.963520581008761?]
[ 7.064764050490923?  5.626248468100069? -1.197679876299471?]
```

Starting with complex numbers with rational real and imaginary parts. Note the use of the conjugate-transpose when checking the orthonormality.

```
sage: A = matrix(QQbar, [[  -2,    -I - 1, 4*I + 2,       -1],
....:                    [-4*I, -2*I + 17,       0,  9*I + 1],
....:                    [   1,  -2*I - 6, -I + 11, -5*I + 1]])
sage: G, M = A.gram_schmidt(orthonormal=True)
sage: (M*G-A).norm() < 10^-10
True
```

```
sage: id3 = G*G.conjugate().transpose()
sage: (id3 - identity_matrix(3)).norm() < 10^-10
True
sage: G.row_space() == A.row_space()   # long time
True
```

A square matrix with small rank. The zero vectors produced as a result of linear dependence get eliminated, so the rows of `G` are a basis for the row space of `A`.

```
sage: A = matrix(QQbar, [[2, -6, 3, 8],
....:                    [1, -3, 2, 5],
....:                    [0,  0, 2, 4],
....:                    [2, -6, 3, 8]])
sage: A.change_ring(QQ).rank()
2
sage: G, M = A.gram_schmidt(orthonormal=True)
sage: G
[ 0.1881441736767195? -0.5644325210301583?  0.2822162605150792?  0.
↪7525766947068779?]
[-0.2502818123591464?   0.750845437077439?  0.3688363550555841?  0.
↪4873908977520218?]
sage: M
[10.630145812734649?                   0]
[ 6.208757731331742? 0.6718090752798139?]
[ 3.574739299857670?  2.687236301119256?]
[10.630145812734649?                   0]
sage: M*G-A
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
sage: (G*G.transpose()-identity_matrix(2)).norm() < 10^-10
True
sage: G.row_space() == A.row_space()
True
```

Exact Rings, Orthogonalization:

If we forego scaling orthogonal vectors to unit vectors, we can apply Gram-Schmidt to a much greater variety of rings. Use the `orthonormal=False` keyword (or assume it as the default). Note that now the orthogonality check creates a diagonal matrix whose diagonal entries are the squares of the lengths of the vectors.

First, in the rationals, without involving `QQbar`.

```
sage: A = matrix(QQ, [[-1,  3,  2,  2],
....:                 [-1,  0, -1,  0],
....:                 [-1, -2, -3, -1],
....:                 [ 1,  1,  2,  0]])
sage: A.rank()
3
sage: G, M = A.gram_schmidt()
sage: G
[    -1      3      2      2]
[-19/18    1/6   -8/9    1/9]
[  2/35  -4/35  -2/35   9/35]
sage: M
[     1      0      0]
```

```
[ -1/18      1      0]
[-13/18  59/35      1]
[   1/3 -48/35     -2]
sage: M*G-A
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
sage: G*G.transpose()
[   18      0      0]
[    0 35/18      0]
[    0      0   3/35]
sage: G.row_space() == A.row_space()
True
```

A complex subfield of the complex numbers.

```
sage: C.<z> = CyclotomicField(5)
sage: A = matrix(C, [[              -z^3 - 2*z,             -z^3 - 1, 2*z^3 -␣
↪2*z^2 + 2*z,            1],
....:                [         z^3 - 2*z^2 + 1, -z^3 + 2*z^2 - z - 1,              ␣
↪          -1,       z^2 + z],
....:                [-1/2*z^3 - 2*z^2 + z + 1,         -z^3 + z - 2,    -2*z^
↪3 + 1/2*z^2, 2*z^2 - z + 2]])
sage: G, M = A.gram_schmidt(orthonormal=False)
sage: G
[                                                      -z^3 - 2*z                  ␣
↪                                             -z^3 - 1                   ␣
↪                         2*z^3 - 2*z^2 + 2*z                            ␣
↪                                1]
[                   155/139*z^3 - 161/139*z^2 + 31/139*z + 13/139                   ␣
↪    -175/139*z^3 + 180/139*z^2 - 125/139*z - 142/139                  ␣
↪230/139*z^3 + 124/139*z^2 + 6/139*z + 19/139                      -14/139*z^
↪3 + 92/139*z^2 - 6/139*z - 95/139]
[-10359/19841*z^3 - 36739/39682*z^2 + 24961/39682*z - 11879/39682  -28209/
↪39682*z^3 - 3671/19841*z^2 + 51549/39682*z - 38613/39682    -42769/39682*z^
↪3 - 615/39682*z^2 - 1252/19841*z - 14392/19841   4895/19841*z^3 + 57885/
↪39682*z^2 - 46094/19841*z + 65747/39682]
sage: M
[                                                              1                  ␣
↪                                                0                   ␣
↪                               0]
[                 14/139*z^3 + 47/139*z^2 + 145/139*z + 95/139                   ␣
↪                                                1                   ␣
↪                               0]
[              -7/278*z^3 + 199/278*z^2 + 183/139*z + 175/278 -3785/39682*z^3␣
↪+ 3346/19841*z^2 - 3990/19841*z + 2039/19841                             ␣
↪                               1]
sage: M*G - A
[0 0 0 0]
[0 0 0 0]
[0 0 0 0]
sage: G*G.conjugate().transpose()
[                               15*z^3 + 15*z^2 + 28                                ␣
↪                          0                                             ␣
↪ 0]
[                                                    0               463/139*z^
↪3 + 463/139*z^2 + 1971/139                                             ␣
↪ 0]
```

```
[                                                   0
↪                         0 230983/19841*z^3 + 230983/19841*z^2 + 1003433/
↪39682]
sage: G.row_space() == A.row_space()
True
```

A slightly edited legacy example.

```
sage: A = matrix(ZZ, 3, [-1, 2, 5, -11, 1, 1, 1, -1, -3]); A
[ -1   2   5]
[-11   1   1]
[  1  -1  -3]
sage: G, mu = A.gram_schmidt()
sage: G
[      -1        2        5]
[   -52/5     -1/5       -2]
[   2/187   36/187 -14/187]
sage: mu
[      1       0       0]
[    3/5       1       0]
[   -3/5 -7/187       1]
sage: G.row(0) * G.row(1)
0
sage: G.row(0) * G.row(2)
0
sage: G.row(1) * G.row(2)
0
```

The relation between mu and A is as follows.

```
sage: mu*G == A
True
```

hadamard_bound ( )

Return an int n such that the absolute value of the determinant of this matrix is at most 10^n.

This is got using both the row norms and the column norms.

This function only makes sense when the base field can be coerced to the real double field RDF or the MPFR Real Field with 53-bits precision.

EXAMPLES:

```
sage: a = matrix(ZZ, 3, [1,2,5,7,-3,4,2,1,123])
sage: a.hadamard_bound()
4
sage: a.det()
-2014
sage: 10^4
10000
```

In this example the Hadamard bound has to be computed (automatically) using MPFR instead of doubles, since doubles overflow:

```
sage: a = matrix(ZZ, 2, [2^10000,3^10000,2^50,3^19292])
sage: a.hadamard_bound()
12215
sage: len(str(a.det()))
12215
```

hermite_form (*include_zero_rows=True*, *transformation=False*)

Return the Hermite form of self, if it is defined.

INPUT:

- `include_zero_rows` – bool (default: True); if False the zero rows in the output matrix are deleted.
- `transformation` – bool (default: False) a matrix U such that U*self == H.

OUTPUT:

- matrix H
- (optional) transformation matrix U such that U*self == H, possibly with zero rows deleted...

EXAMPLES:

```
sage: M = FunctionField(GF(7),'x').maximal_order()
sage: K.<x> = FunctionField(GF(7)); M = K.maximal_order()
sage: A = matrix(M, 2, 3, [x, 1, 2*x, x, 1+x, 2])
sage: A.hermite_form()
[      x       1     2*x]
[      0       x 5*x + 2]
sage: A.hermite_form(transformation=True)
(
[      x       1     2*x]  [1 0]
[      0       x 5*x + 2], [6 1]
)
sage: A = matrix(M, 2, 3, [x, 1, 2*x, 2*x, 2, 4*x])
sage: A.hermite_form(transformation=True, include_zero_rows=False)
([  x   1 2*x], [1 0])
sage: H, U = A.hermite_form(transformation=True, include_zero_rows=True); H, U
(
[  x   1 2*x]  [1 0]
[  0   0   0], [5 1]
)
sage: U*A == H
True
sage: H, U = A.hermite_form(transformation=True, include_zero_rows=False)
sage: U*A
[  x   1 2*x]
sage: U*A == H
True
```

hessenberg_form ( )

Return Hessenberg form of self.

If the base ring is merely an integral domain (and not a field), the Hessenberg form will (in general) only be defined over the fraction field of the base ring.

EXAMPLES:

```
sage: A = matrix(ZZ,4,[2, 1, 1, -2, 2, 2, -1, -1, -1,1,2,3,4,5,6,7])
sage: h = A.hessenberg_form(); h
[    2  -7/2 -19/5    -2]
[    2   1/2 -17/5    -1]
[    0  25/4  15/2   5/2]
[    0     0  58/5     3]
sage: parent(h)
```

```
Full MatrixSpace of 4 by 4 dense matrices over Rational Field
sage: A.hessenbergize()
Traceback (most recent call last):
...
TypeError: Hessenbergize only possible for matrices over a field
```

hessenbergize ( )

Transform self to Hessenberg form.

The hessenberg form of a matrix A is a matrix that is similar to A, so has the same characteristic polynomial as A, and is upper triangular except possible for entries right below the diagonal.

ALGORITHM: See Henri Cohen's first book.

EXAMPLES:

```
sage: A = matrix(QQ,3, [2, 1, 1, -2, 2, 2, -1, -1, -1])
sage: A.hessenbergize(); A
[  2 3/2   1]
[ -2   3   2]
[  0  -3  -2]
```

```
sage: A = matrix(QQ,4, [2, 1, 1, -2, 2, 2, -1, -1, -1,1,2,3,4,5,6,7])
sage: A.hessenbergize(); A
[    2  -7/2 -19/5    -2]
[    2   1/2 -17/5    -1]
[    0  25/4  15/2   5/2]
[    0     0  58/5     3]
```

You can't Hessenbergize an immutable matrix:

```
sage: A = matrix(QQ, 3, [1..9])
sage: A.set_immutable()
sage: A.hessenbergize()
Traceback (most recent call last):
...
ValueError: matrix is immutable; please change a copy instead (i.e., use␣
↪copy(M) to change a copy of M).
```

image ( )

Return the image of the homomorphism on rows defined by this matrix.

EXAMPLES:

```
sage: MS1 = MatrixSpace(ZZ,4)
sage: MS2 = MatrixSpace(QQ,6)
sage: A = MS1.matrix([3,4,5,6,7,3,8,10,14,5,6,7,2,2,10,9])
sage: B = MS2.random_element()
```

```
sage: image(A)
Free module of degree 4 and rank 4 over Integer Ring
Echelon basis matrix:
[  1   0   0 426]
[  0   1   0 518]
[  0   0   1 293]
[  0   0   0 687]
```

```
sage: image(B) == B.row_module()
True
```

indefinite_factorization (*algorithm='symmetric'*, *check=True*)

Decomposes a symmetric or Hermitian matrix into a lower triangular matrix and a diagonal matrix.

INPUT:

- `self` - a square matrix over a ring. The base ring must have an implemented fraction field.
- `algorithm` - default: `'symmetric'`. Either `'symmetric'` or `'hermitian'`, according to if the input matrix is symmetric or hermitian.
- `check` - default: `True` - if `True` then performs the check that the matrix is consistent with the `algorithm` keyword.

OUTPUT:

A lower triangular matrix L with each diagonal element equal to 1 and a vector of entries that form a diagonal matrix D. The vector of diagonal entries can be easily used to form the matrix, as demonstrated below in the examples.

For a symmetric matrix, A, these will be related by

$$A = LDL^T$$

If A is Hermitian matrix, then the transpose of L should be replaced by the conjugate-transpose of L.

If any leading principal submatrix (a square submatrix in the upper-left corner) is singular then this method will fail with a `ValueError`.

ALGORITHM:

The algorithm employed only uses field operations, but the computation of each diagonal entry has the potential for division by zero. The number of operations is of order $n^3/3$, which is half the count for an LU decomposition. This makes it an appropriate candidate for solving systems with symmetric (or Hermitian) coefficient matrices.

EXAMPLES:

There is no requirement that a matrix be positive definite, as indicated by the negative entries in the resulting diagonal matrix. The default is that the input matrix is symmetric.

```
sage: A = matrix(QQ, [[ 3,  -6,   9,   6,  -9],
....:                 [-6,  11, -16, -11,  17],
....:                 [ 9, -16,  28,  16, -40],
....:                 [ 6, -11,  16,   9, -19],
....:                 [-9,  17, -40, -19,  68]])
sage: A.is_symmetric()
True
sage: L, d = A.indefinite_factorization()
sage: D = diagonal_matrix(d)
sage: L
[ 1  0  0  0  0]
[-2  1  0  0  0]
[ 3 -2  1  0  0]
[ 2 -1  0  1  0]
[-3  1 -3  1  1]
sage: D
[ 3  0  0  0  0]
[ 0 -1  0  0  0]
```

```
[ 0  0  5  0  0]
[ 0  0  0 -2  0]
[ 0  0  0  0 -1]
sage: A == L*D*L.transpose()
True
```

Optionally, Hermitian matrices can be factored and the result has a similar property (but not identical). Here, the field is all complex numbers with rational real and imaginary parts. As theory predicts, the diagonal entries will be real numbers.

```
sage: C.<I> = QuadraticField(-1)
sage: B = matrix(C, [[      2, 4 - 2*I, 2 + 2*I],
....:                [4 + 2*I,       8,    10*I],
....:                [2 - 2*I,   -10*I,      -3]])
sage: B.is_hermitian()
True
sage: L, d = B.indefinite_factorization(algorithm='hermitian')
sage: D = diagonal_matrix(d)
sage: L
[      1       0       0]
[  I + 2       1       0]
[ -I + 1 2*I + 1       1]
sage: D
[ 2  0  0]
[ 0 -2  0]
[ 0  0  3]
sage: B == L*D*L.conjugate_transpose()
True
```

If a leading principal submatrix has zero determinant, this algorithm will fail. This will never happen with a positive definite matrix.

```
sage: A = matrix(QQ, [[21, 15, 12, -2],
....:                 [15, 12,  9,  6],
....:                 [12,  9,  7,  3],
....:                 [-2,  6,  3,  8]])
sage: A.is_symmetric()
True
sage: A[0:3,0:3].det() == 0
True
sage: A.indefinite_factorization()
Traceback (most recent call last):
...
ValueError: 3x3 leading principal submatrix is singular,
so cannot create indefinite factorization
```

This algorithm only depends on field operations, so outside of the singular submatrix situation, any matrix may be factored. This provides a reasonable alternative to the Cholesky decomposition.

```
sage: F.<a> = FiniteField(5^3)
sage: A = matrix(F,
....:     [[      a^2 + 2*a, 4*a^2 + 3*a + 4,       3*a^2 + a, 2*a^2 + 2*a +
↪1],
....:      [4*a^2 + 3*a + 4,       4*a^2 + 2,             3*a, 2*a^2 + 4*a +
↪2],
....:      [      3*a^2 + a,             3*a,       3*a^2 + 2, 3*a^2 + 2*a +
↪3],
....:      [2*a^2 + 2*a + 1, 2*a^2 + 4*a + 2, 3*a^2 + 2*a + 3, 3*a^2 + 2*a +
↪4]])
```

```
sage: A.is_symmetric()
True
sage: L, d = A.indefinite_factorization()
sage: D = diagonal_matrix(d)
sage: L
[              1               0               0               0]
[4*a^2 + 4*a + 3               1               0               0]
[              3   4*a^2 + a + 2               1               0]
[      4*a^2 + 4 2*a^2 + 3*a + 3 2*a^2 + 3*a + 1               1]
sage: D
[      a^2 + 2*a               0               0               0]
[              0 2*a^2 + 2*a + 4               0               0]
[              0               0 3*a^2 + 4*a + 3               0]
[              0               0               0       a^2 + 3*a]
sage: A == L*D*L.transpose()
True
```

AUTHOR:

- Rob Beezer (2012-05-24)

integer_kernel (*ring='ZZ'*)

Return the kernel of this matrix over the given ring (which should be either the base ring, or a PID whose fraction field is the base ring).

Assume that the base field of this matrix has a numerator and denominator functions for its elements, e.g., it is the rational numbers or a fraction field. This function computes a basis over the integers for the kernel of self.

If the matrix is not coercible into QQ, then the PID itself should be given as a second argument, as in the third example below.

EXAMPLES:

```
sage: A = MatrixSpace(QQ, 4)(range(16))
sage: A.integer_kernel()
Free module of degree 4 and rank 2 over Integer Ring
Echelon basis matrix:
[ 1  0 -3  2]
[ 0  1 -2  1]
```

The integer kernel even makes sense for matrices with fractional entries:

```
sage: A = MatrixSpace(QQ, 2)(['1/2',0,  0, 0])
sage: A.integer_kernel()
Free module of degree 2 and rank 1 over Integer Ring
Echelon basis matrix:
[0 1]
```

An example over a bigger ring:

```
sage: L.<w> = NumberField(x^2 - x + 2)
sage: OL = L.ring_of_integers()
sage: A = matrix(L, 2, [1, w/2])
sage: A.integer_kernel(OL)
Free module of degree 2 and rank 1 over Maximal Order in Number Field in w␣
↪with defining polynomial x^2 - x + 2
Echelon basis matrix:
[    -1 -w + 1]
```

inverse ( )

Returns the inverse of self, without changing self.

Note that one can use the Python inverse operator to obtain the inverse as well.

EXAMPLES:

```
sage: m = matrix([[1,2],[3,4]])
sage: m^(-1)
[  -2    1]
[ 3/2 -1/2]
sage: m.inverse()
[  -2    1]
[ 3/2 -1/2]
sage: ~m
[  -2    1]
[ 3/2 -1/2]
```

```
sage: m = matrix([[1,2],[3,4]], sparse=True)
sage: m^(-1)
[  -2    1]
[ 3/2 -1/2]
sage: m.inverse()
[  -2    1]
[ 3/2 -1/2]
sage: ~m
[  -2    1]
[ 3/2 -1/2]
```

is_bistochastic (*normalized=True*)

Returns `True` if this matrix is bistochastic.

A matrix is said to be bistochastic if both the sums of the entries of each row and the sum of the entries of each column are equal to 1 and all entries are nonnegative.

INPUT:

- `normalized` – if set to `True` (default), checks that the sums are equal to 1. When set to `False`, checks that the row sums and column sums are all equal to some constant possibly different from 1.

EXAMPLES:

The identity matrix is clearly bistochastic:

```
sage: Matrix(5,5,1).is_bistochastic()
True
```

The same matrix, multiplied by 2, is not bistochastic anymore, though is verifies the constraints of `normalized == False`:

```
sage: (2 * Matrix(5,5,1)).is_bistochastic()
False
sage: (2 * Matrix(5,5,1)).is_bistochastic(normalized = False)
True
```

Here is a matrix whose row and column sums is 1, but not all entries are nonnegative:

```
sage: m = matrix([[-1,2],[2,-1]])
sage: m.is_bistochastic()
False
```

is_diagonalizable (*base_field=None*)

Determines if the matrix is similar to a diagonal matrix.

INPUT:

- `base_field` - a new field to use for entries of the matrix.

OUTPUT:

If `self` is the matrix A, then it is diagonalizable if there is an invertible matrix S and a diagonal matrix D such that

$$S^{-1}AS = D$$

This routine returns `True` if `self` is diagonalizable. The diagonal entries of the matrix D are the eigenvalues of A. It may be necessary to "increase" the base field to contain all of the eigenvalues. Over the rationals, the field of algebraic numbers, `sage.rings.qqbar` is a good choice.

To obtain the matrices S and D use the *`jordan_form()`* method with the `transformation=True` keyword.

ALGORITHM:

For each eigenvalue, this routine checks that the algebraic multiplicity (number of occurences as a root of the characteristic polynomial) is equal to the geometric multiplicity (dimension of the eigenspace), which is sufficient to ensure a basis of eigenvectors for the columns of S.

EXAMPLES:

A matrix that is diagonalizable over the rationals, as evidenced by its Jordan form.

```
sage: A = matrix(QQ, [[-7, 16, 12,  0,    6],
....:                  [-9, 15,  0, 12, -27],
....:                  [ 9, -8, 11, -12,  51],
....:                  [ 3, -4,  0,  -1,   9],
....:                  [-1,  0, -4,   4, -12]])
sage: A.jordan_form(subdivide=False)
[ 2  0  0  0  0]
[ 0  3  0  0  0]
[ 0  0  3  0  0]
[ 0  0  0 -1  0]
[ 0  0  0  0 -1]
sage: A.is_diagonalizable()
True
```

A matrix that is not diagonalizable over the rationals, as evidenced by its Jordan form.

```
sage: A = matrix(QQ, [[-3, -14, 2, -1, 15],
....:                  [4, 6, -2, 3, -8],
....:                  [-2, -14, 0, 0, 10],
....:                  [3, 13, -2, 0, -11],
....:                  [-1, 6, 1, -3, 1]])
sage: A.jordan_form(subdivide=False)
[-1  1  0  0  0]
[ 0 -1  0  0  0]
[ 0  0  2  1  0]
[ 0  0  0  2  1]
[ 0  0  0  0  2]
sage: A.is_diagonalizable()
False
```

If any eigenvalue of a matrix is outside the base ring, then this routine raises an error. However, the ring can be "expanded" to contain the eigenvalues.

```
sage: A = matrix(QQ, [[1,  0,  1,  1, -1],
....:                 [0,  1,  0,  4,  8],
....:                 [2,  1,  3,  5,  1],
....:                 [2, -1,  1,  0, -2],
....:                 [0, -1, -1, -5, -8]])

sage: [e in QQ for e in A.eigenvalues()]
[False, False, False, False, False]
sage: A.is_diagonalizable()
Traceback (most recent call last):
...
RuntimeError: an eigenvalue of the matrix is not contained in Rational Field

sage: [e in QQbar for e in A.eigenvalues()]
[True, True, True, True, True]
sage: A.is_diagonalizable(base_field=QQbar)
True
```

Other exact fields may be employed, though it will not always be possible to expand their base fields to contain all the eigenvalues.

```
sage: F.<b> = FiniteField(5^2)
sage: A = matrix(F, [[      4, 3*b + 2, 3*b + 1, 3*b + 4],
....:                [2*b + 1,     4*b,       0,       2],
....:                [    4*b,   b + 2, 2*b + 3,       3],
....:                [    2*b,     3*b, 4*b + 4, 3*b + 3]])
sage: A.jordan_form()
[      4       1|      0       0]
[      0       4|      0       0]
[---------------+---------------]
[      0       0|2*b + 1       1]
[      0       0|      0 2*b + 1]
sage: A.is_diagonalizable()
False

sage: F.<c> = QuadraticField(-7)
sage: A = matrix(F, [[   c + 3,   2*c - 2,   -2*c + 2,     c - 1],
....:                [2*c + 10, 13*c + 15, -13*c - 17, 11*c + 31],
....:                [2*c + 10, 14*c + 10, -14*c - 12, 12*c + 30],
....:                [       0,   2*c - 2,   -2*c + 2,   2*c + 2]])
sage: A.jordan_form(subdivide=False)
[    4     0     0     0]
[    0    -2     0     0]
[    0     0 c + 3     0]
[    0     0     0 c + 3]
sage: A.is_diagonalizable()
True
```

A trivial matrix is diagonalizable, trivially.

```
sage: A = matrix(QQ, 0, 0)
sage: A.is_diagonalizable()
True
```

A matrix must be square to be diagonalizable.

```
sage: A = matrix(QQ, 3, 4)
sage: A.is_diagonalizable()
False
```

The matrix must have entries from a field, and it must be an exact field.

```
sage: A = matrix(ZZ, 4, range(16))
sage: A.is_diagonalizable()
Traceback (most recent call last):
...
ValueError: matrix entries must be from a field, not Integer Ring

sage: A = matrix(RDF, 4, range(16))
sage: A.is_diagonalizable()
Traceback (most recent call last):
...
ValueError: base field must be exact, not Real Double Field
```

AUTHOR:

- Rob Beezer (2011-04-01)

is_normal ()

Returns `True` if the matrix commutes with its conjugate-transpose.

OUTPUT:

`True` if the matrix is square and commutes with its conjugate-transpose, and `False` otherwise.

Normal matrices are precisely those that can be diagonalized by a unitary matrix.

This routine is for matrices over exact rings and so may not work properly for matrices over `RR` or `CC`. For matrices with approximate entries, the rings of double-precision floating-point numbers, `RDF` and `CDF`, are a better choice since the *sage.matrix.matrix_double_dense.Matrix_double_dense.is_normal()* method has a tolerance parameter. This provides control over allowing for minor discrepancies between entries when checking equality.

The result is cached.

EXAMPLES:

Hermitian matrices are normal.

```
sage: A = matrix(QQ, 5, 5, range(25)) + I*matrix(QQ, 5, 5, range(0, 50, 2))
sage: B = A*A.conjugate_transpose()
sage: B.is_hermitian()
True
sage: B.is_normal()
True
```

Circulant matrices are normal.

```
sage: G = graphs.CirculantGraph(20, [3, 7])
sage: D = digraphs.Circuit(20)
sage: A = 3*D.adjacency_matrix() - 5*G.adjacency_matrix()
sage: A.is_normal()
True
```

Skew-symmetric matrices are normal.

```
sage: A = matrix(QQ, 5, 5, range(25))
sage: B = A - A.transpose()
sage: B.is_skew_symmetric()
True
sage: B.is_normal()
True
```

A small matrix that does not fit into any of the usual categories of normal matrices.

```
sage: A = matrix(ZZ, [[1, -1],
....:                 [1,  1]])
sage: A.is_normal()
True
sage: not A.is_hermitian() and not A.is_skew_symmetric()
True
```

Sage has several fields besides the entire complex numbers where conjugation is non-trivial.

```
sage: F.<b> = QuadraticField(-7)
sage: C = matrix(F, [[-2*b - 3,  7*b - 6, -b + 3],
....:                [-2*b - 3, -3*b + 2,   -2*b],
....:                [   b + 1,        0,     -2]])
sage: C = C*C.conjugate_transpose()
sage: C.is_normal()
True
```

A matrix that is nearly normal, but for a non-real diagonal entry.

```
sage: A = matrix(QQbar, [[    2,   2-I, 1+4*I],
....:                    [  2+I,   3+I, 2-6*I],
....:                    [1-4*I, 2+6*I,     5]])
sage: A.is_normal()
False
sage: A[1,1] = 132
sage: A.is_normal()
True
```

Rectangular matrices are never normal.

```
sage: A = matrix(QQbar, 3, 4)
sage: A.is_normal()
False
```

A square, empty matrix is trivially normal.

```
sage: A = matrix(QQ, 0, 0)
sage: A.is_normal()
True
```

AUTHOR:

•Rob Beezer (2011-03-31)

is_one ( )

Return True if this matrix is the identity matrix.

EXAMPLES:

```
sage: m = matrix(QQ,2,2,range(4))
sage: m.is_one()
False
sage: m = matrix(QQ,2,[5,0,0,5])
sage: m.is_one()
False
sage: m = matrix(QQ,2,[1,0,0,1])
sage: m.is_one()
True
sage: m = matrix(QQ,2,[1,1,1,1])
sage: m.is_one()
False
```

is_permutation_of (*N*, *check=False*)

Return `True` if there exists a permutation of rows and columns sending `self` to `N` and `False` otherwise.

INPUT:

- `N` – a matrix.
- **`check` – boolean (default: `False`). If `False`** return Boolean indicating whether there exists a permutation of rows and columns sending `self` to `N` and False otherwise. If `True` return a tuple of a Boolean and a permutation mapping `self` to `N` if such a permutation exists, and (`False` , `None`) if it does not.

OUTPUT:

A Boolean or a tuple of a Boolean and a permutation.

EXAMPLES:

```
sage: M = matrix(ZZ,[[1,2,3],[3,5,3],[2,6,4]])
sage: M
[1 2 3]
[3 5 3]
[2 6 4]
sage: N = matrix(ZZ,[[1,2,3],[2,6,4],[3,5,3]])
sage: N
[1 2 3]
[2 6 4]
[3 5 3]
sage: M.is_permutation_of(N)
True
```

Some examples that are not permutations of each other:

```
sage: N = matrix(ZZ,[[1,2,3],[4,5,6],[7,8,9]])
sage: N
[1 2 3]
[4 5 6]
[7 8 9]
sage: M.is_permutation_of(N)
False
sage: N = matrix(ZZ,[[1,2],[3,4]])
sage: N
[1 2]
[3 4]
sage: M.is_permutation_of(N)
False
```

And for when `check` is True:

```
sage: N = matrix(ZZ,[[3,5,3],[2,6,4],[1,2,3]])
sage: N
[3 5 3]
[2 6 4]
[1 2 3]
sage: r = M.is_permutation_of(N, check=True)
sage: r
(True, ((1,2,3), ()))
sage: p = r[1]
sage: M.with_permuted_rows_and_columns(*p) == N
True
```

is_positive_definite ()

Determines if a real or symmetric matrix is positive definite.

A square matrix A is positive definite if it is symmetric with real entries or Hermitan with complex entries, and for every non-zero vector $\vec{x}$

$$\vec{x}^* A\vec{x} > 0$$

Here $\vec{x}^*$ is the conjugate-transpose, which can be simplified to just the transpose in the real case.

ALGORITHM:

A matrix is positive definite if and only if the diagonal entries from the indefinite factorization are all positive (see *indefinite_factorization()*). So this algorithm is of order `n^3/3` and may be applied to matrices with elements of any ring that has a fraction field contained within the reals or complexes.

INPUT:

Any square matrix.

OUTPUT:

This routine will return `True` if the matrix is square, symmetric or Hermitian, and meets the condition above for the quadratic form.

The base ring for the elements of the matrix needs to have a fraction field implemented and the computations that result from the indefinite factorization must be convertable to real numbers that are comparable to zero.

EXAMPLES:

A real symmetric matrix that is positive definite, as evidenced by the positive entries for the diagonal matrix of the indefinite factorization and the positive determinants of the leading principal submatrices.

```
sage: A = matrix(QQ, [[ 4, -2,  4,  2],
....:                 [-2, 10, -2, -7],
....:                 [ 4, -2,  8,  4],
....:                 [ 2, -7,  4,  7]])
sage: A.is_positive_definite()
True
sage: _, d = A.indefinite_factorization(algorithm='symmetric')
sage: d
(4, 9, 4, 1)
sage: [A[:i,:i].determinant() for i in range(1,A.nrows()+1)]
[4, 36, 144, 144]
```

A real symmetric matrix which is not positive definite, along with a vector that makes the quadratic form negative.

```
sage: A = matrix(QQ, [[ 3,  -6,   9,   6,  -9],
....:                 [-6,  11, -16, -11,  17],
....:                 [ 9, -16,  28,  16, -40],
....:                 [ 6, -11,  16,   9, -19],
....:                 [-9,  17, -40, -19,  68]])
sage: A.is_positive_definite()
False
sage: _, d = A.indefinite_factorization(algorithm='symmetric')
sage: d
(3, -1, 5, -2, -1)
sage: [A[:i,:i].determinant() for i in range(1,A.nrows()+1)]
[3, -3, -15, 30, -30]
sage: u = vector(QQ, [2, 2, 0, 1, 0])
sage: u.row()*A*u
(-3)
```

A real symmetric matrix with a singular leading principal submatrix, that is therefore not positive definite. The vector `u` makes the quadratic form zero.

```
sage: A = matrix(QQ, [[21, 15, 12, -2],
....:                 [15, 12,  9,  6],
....:                 [12,  9,  7,  3],
....:                 [-2,  6,  3,  8]])
sage: A.is_positive_definite()
False
sage: [A[:i,:i].determinant() for i in range(1,A.nrows()+1)]
[21, 27, 0, -75]
sage: u = vector(QQ, [1,1,-3,0])
sage: u.row()*A*u
(0)
```

An Hermitian matrix that is positive definite.

```
sage: C.<I> = NumberField(x^2 + 1, embedding=CC(0,1))
sage: A = matrix(C, [[        23,  17*I + 3,  24*I + 25,     21*I],
....:               [ -17*I + 3,        38, -69*I + 89, 7*I + 15],
....:               [-24*I + 25, 69*I + 89,        976, 24*I + 6],
....:               [     -21*I, -7*I + 15,  -24*I + 6,       28]])
sage: A.is_positive_definite()
True
sage: _, d = A.indefinite_factorization(algorithm='hermitian')
sage: d
(23, 576/23, 89885/144, 142130/17977)
sage: [A[:i,:i].determinant() for i in range(1,A.nrows()+1)]
[23, 576, 359540, 2842600]
```

An Hermitian matrix that is not positive definite. The vector `u` makes the quadratic form negative.

```
sage: C.<I> = QuadraticField(-1)
sage: B = matrix(C, [[      2, 4 - 2*I, 2 + 2*I],
....:               [4 + 2*I,       8,    10*I],
....:               [2 - 2*I,   -10*I,      -3]])
sage: B.is_positive_definite()
False
sage: _, d = B.indefinite_factorization(algorithm='hermitian')
```

```
sage: d
(2, -2, 3)
sage: [B[:i,:i].determinant() for i in range(1,B.nrows()+1)]
[2, -4, -12]
sage: u = vector(C, [-5 + 10*I, 4 - 3*I, 0])
sage: u.row().conjugate()*B*u
(-50)
```

A positive definite matrix over an algebraically closed field.

```
sage: A = matrix(QQbar, [[        2,    4 + 2*I,    6 - 4*I],
....:                    [ -2*I + 4,         11, 10 - 12*I],
....:                    [  4*I + 6, 10 + 12*I,        37]])
sage: A.is_positive_definite()
True
sage: [A[:i,:i].determinant() for i in range(1,A.nrows()+1)]
[2, 2, 6]
```

AUTHOR:

- Rob Beezer (2012-05-24)

is_scalar (*a=None*)

Return True if this matrix is a scalar matrix.

INPUT:

- base_ring element a, which is chosen as self[0][0] if a = None

OUTPUT:

- whether self is a scalar matrix (in fact the scalar matrix aI if a is input)

EXAMPLES:

```
sage: m = matrix(QQ,2,2,range(4))
sage: m.is_scalar(5)
False
sage: m = matrix(QQ,2,[5,0,0,5])
sage: m.is_scalar(5)
True
sage: m = matrix(QQ,2,[1,0,0,1])
sage: m.is_scalar(1)
True
sage: m = matrix(QQ,2,[1,1,1,1])
sage: m.is_scalar(1)
False
```

is_similar (*other*, *transformation=False*)

Return `True` if `self` and `other` are similar, i.e. related by a change-of-basis matrix.

INPUT:

- `other` – a matrix, which should be square, and of the same size as `self`.
- `transformation` – default: `False` - if `True`, the output may include the change-of-basis matrix (also known as the similarity transformation). See below for an exact description.

OUTPUT:

Two matrices, A and B are similar if they are square matrices of the same size and there is an invertible matrix S such that $A = S^{-1}BS$. S can be interpreted as a change-of-basis matrix if A and B are viewed as matrix representations of the same linear transformation from a vector space to itself.

When `transformation=False` this method will return `True` if such a matrix S exists, otherwise it will return `False`. When `transformation=True` the method returns a pair. The first part of the pair is `True` or `False` depending on if the matrices are similar. The second part of the pair is the change-of-basis matrix when the matrices are similar and `None` when the matrices are not similar.

When a similarity transformation matrix `S` is requested, it will satisfy `self = S.inverse()*other*S`.

rings and coefficients

Inexact rings are not supported. Only rings having a fraction field can be used as coefficients.

The base rings for the matrices are promoted to a common field for the similarity check using rational form over this field.

If the fraction fields of both matrices are the same, this field is used. Otherwise, if the fraction fields are only related by a canonical coercion, the common coercion field is used.

In all cases, the result is about similarity over a common field.

similarity transformation

For computation of the similarity transformation, the matrices are first checked to be similar over their common field.

In this case, a similarity transformation is then searched for over the common field. If this fails, the matrices are promoted to the algebraic closure of their common field (whenever it is available) and a similarity transformation is looked for over the algebraic closure.

For example, matrices over the rationals may be promoted to the field of algebraic numbers (`QQbar`) for computation of the similarity transformation.

> **Warning:** When the two matrices are similar, this routine may fail to find the similarity transformation. A technical explanation follows.

The similarity check is accomplished with rational form, which will be successful for any pair of matrices over the same field. However, the computation of rational form does not provide a transformation. So we instead compute Jordan form, which does provide a transformation. But Jordan form will require that the eigenvalues of the matrix can be represented within Sage, requiring the existence of the appropriate extension field. When this is not possible, a `RuntimeError` is raised, as demonstrated in an example below.

EXAMPLES:

The two matrices in this example were constructed to be similar. The computations happen in the field of algebraic numbers, but we are able to convert the change-of-basis matrix back to the rationals (which may not always be possible).

```
sage: A = matrix(ZZ, [[-5, 2, -11],
....:                 [-6, 7, -42],
....:                 [0, 1, -6]])
```

```
sage: B = matrix(ZZ, [[ 1, 12,  3],
....:                 [-1, -6, -1],
....:                 [ 0,  6,  1]])
sage: A.is_similar(B)
True
sage: _, T = A.is_similar(B, transformation=True)
sage: T
[ 1.000000000000000? + 0.?e-14*I            0.?e-14 + 0.?e-14*I            0.?
↪e-14 + 0.?e-14*I]
[-0.66666666666667? + 0.?e-15*I 0.166666666666667? + 0.?e-15*I -0.
↪83333333333334? + 0.?e-14*I]
[ 0.66666666666667? + 0.?e-14*I            0.?e-14 + 0.?e-14*I -0.
↪33333333333333? + 0.?e-14*I]
sage: T.change_ring(QQ)
[   1    0    0]
[-2/3  1/6 -5/6]
[ 2/3    0 -1/3]
sage: A == T.inverse()*B*T
True
```

Other exact fields are supported.

```
sage: F.<a> = FiniteField(7^2)
sage: A = matrix(F,[[2*a + 5, 6*a + 6,   a + 3],
....:               [  a + 3, 2*a + 2, 4*a + 2],
....:               [2*a + 6, 5*a + 5,     3*a]])
sage: B = matrix(F,[[5*a + 5, 6*a + 4,   a + 1],
....:               [  a + 5, 4*a + 3, 3*a + 3],
....:               [3*a + 5,   a + 4, 5*a + 6]])
sage: A.is_similar(B)
True
sage: B.is_similar(A)
True
sage: _, T = A.is_similar(B, transformation=True)
sage: T
[      1       0       0]
[6*a + 1 4*a + 3 4*a + 2]
[6*a + 3 3*a + 5 3*a + 6]
sage: A == T.inverse()*B*T
True
```

Two matrices with different sets of eigenvalues, so they cannot possibly be similar.

```
sage: A = matrix(QQ, [[ 2,  3, -3, -6],
....:                 [ 0,  1, -2, -8],
....:                 [-3, -3,  4,  3],
....:                 [-1, -2,  2,  6]])
sage: B = matrix(QQ, [[ 1,  1,  2,  4],
....:                 [-1,  2, -3, -7],
....:                 [-2,  3, -4, -7],
....:                 [ 0, -1,  0,  0]])
sage: A.eigenvalues() == B.eigenvalues()
False
sage: A.is_similar(B, transformation=True)
(False, None)
```

Similarity is an equivalence relation, so this routine computes a representative of the equivalence class for each matrix, the rational form, as provided by `rational_form()` . The matrices below have identical eigenvalues (as evidenced by equal characteristic polynomials), but slightly different rational forms, and hence are not similar.

```
sage: A = matrix(QQ, [[ 19, -7, -29],
....:                 [-16, 11,  30],
....:                 [ 15, -7, -25]])
sage: B = matrix(QQ, [[-38, -63,  42],
....:                 [ 14,  25, -14],
....:                 [-14, -21,  18]])
sage: A.charpoly() == B.charpoly()
True
sage: A.rational_form()
[  0   0 -48]
[  1   0   8]
[  0   1   5]
sage: B.rational_form()
[ 4| 0  0]
[--+-----]
[ 0| 0 12]
[ 0| 1  1]
sage: A.is_similar(B)
False
```

Obtaining the transformation between two similar matrices requires the Jordan form, which requires computing the eigenvalues of the matrix, which may not lie in the field used for entries of the matrix. In this unfortunate case, the computation of the transformation may fail with a `RuntimeError` , EVEN when the matrices are similar. This is not the case for matrices over the integers, rationals or algebraic numbers, since the computations are done in the algebraically closed field of algebraic numbers. Here is an example where the similarity is obvious by design, but we are not able to resurrect a similarity transformation.

```
sage: F.<a> = FiniteField(7^2)
sage: C = matrix(F,[[   a + 2, 5*a + 4],
....:               [6*a + 6, 6*a + 4]])
sage: S = matrix(F, [[0, 1],
....:                 [1, 0]])
sage: D = S.inverse()*C*S
sage: C.is_similar(D)
True
sage: C.is_similar(D, transformation=True)
Traceback (most recent call last):
...
RuntimeError: unable to compute transformation for similar matrices
sage: C.jordan_form()
Traceback (most recent call last):
...
RuntimeError: Some eigenvalue does not exist in
Finite Field in a of size 7^2.
```

An example over a finite field of prime order, which uses the algebraic closure of this field to find the change-of-basis matrix:

```
sage: cox = posets.TamariLattice(3).coxeter_transformation()
sage: M = cox.change_ring(GF(3))
sage: M.is_similar(M**3, True)   # long time
(
      [1 0 0 0 0]
      [0 1 1 0 2]
      [0 0 0 0 1]
```

```
      [1 2 0 2 1]
True, [0 0 1 0 0]
)
```

Inexact rings and fields are not supported.

```
sage: A = matrix(CDF, 2, 2, range(4))
sage: B = copy(A)
sage: A.is_similar(B)
Traceback (most recent call last):
...
TypeError: matrix entries must come from an exact field,
not Complex Double Field
```

Base rings for the matrices need to have a fraction field. So in particular, the ring needs to be at least an integral domain.

```
sage: Z6 = Integers(6)
sage: A = matrix(Z6, 2, 2, range(4))
sage: A.is_similar(A)
Traceback (most recent call last):
...
ValueError: base ring of a matrix needs a fraction field,
maybe the ring is not an integral domain
```

If the fraction fields of the entries are unequal and do not coerce in a common field, it is an error.

```
sage: A = matrix(GF(3), 2, 2, range(4))
sage: B = matrix(GF(2), 2, 2, range(4))
sage: A.is_similar(B, transformation=True)
Traceback (most recent call last):
...
TypeError: no common canonical parent for objects with parents:
'Full MatrixSpace of 2 by 2 dense matrices over Finite Field
of size 3' and
'Full MatrixSpace of 2 by 2 dense matrices over Finite Field
of size 2'
```

A matrix over the integers and a matrix over the algebraic numbers will be compared over the algebraic numbers (by coercion of `QQ` in `QQbar`).

```
sage: A = matrix(ZZ, 2, 2, range(4))
sage: B = matrix(QQbar, 2, 2, range(4))
sage: A.is_similar(B)
True
```

AUTHOR:

- Rob Beezer (2011-03-15, 2015-05-25)

is_unitary ()

Returns `True` if the columns of the matrix are an orthonormal basis.

For a matrix with real entries this determines if a matrix is "orthogonal" and for a matrix with complex entries this determines if the matrix is "unitary."

OUTPUT:

`True` if the matrix is square and its conjugate-transpose is its inverse, and `False` otherwise. In other words, a matrix is orthogonal or unitary if the product of its conjugate-transpose times the matrix is the identity matrix.

For numerical matrices a specialized routine available over `RDF` and `CDF` is a good choice.

EXAMPLES:

```
sage: A = matrix(QQbar, [[(1/sqrt(5))*(1+i), (1/sqrt(55))*(3+2*I), (1/
↪sqrt(22))*(2+2*I)],
....:                    [(1/sqrt(5))*(1-i), (1/sqrt(55))*(2+2*I),  (1/
↪sqrt(22))*(-3+I)],
....:                    [    (1/sqrt(5))*I, (1/sqrt(55))*(3-5*I),    (1/
↪sqrt(22))*(-2)]])
sage: A.is_unitary()
True
```

A permutation matrix is always orthogonal.

```
sage: sigma = Permutation([1,3,4,5,2])
sage: P = sigma.to_matrix(); P
[1 0 0 0 0]
[0 0 0 0 1]
[0 1 0 0 0]
[0 0 1 0 0]
[0 0 0 1 0]
sage: P.is_unitary()
True
sage: P.change_ring(GF(3)).is_unitary()
True
sage: P.change_ring(GF(3)).is_unitary()
True
```

A square matrix far from unitary.

```
sage: A = matrix(QQ, 4, range(16))
sage: A.is_unitary()
False
```

Rectangular matrices are never unitary.

```
sage: A = matrix(QQbar, 3, 4)
sage: A.is_unitary()
False
```

jordan_form (*base_ring=None*, *sparse=False*, *subdivide=True*, *transformation=False*, *eigenvalues=None*, *check_input=True*)

Compute the Jordan normal form of this square matrix A, if it exists.

This computation is performed in a naive way using the ranks of powers of $A-xI$, where x is an eigenvalue of the matrix A. If desired, a transformation matrix P can be returned, which is such that the Jordan canonical form is given by $P^{-1}AP$; if the matrix is diagonalizable, this equals to *eigendecomposition* or *spectral decomposition*.

INPUT:

- `base_ring` - Ring in which to compute the Jordan form.
- `sparse` - (default `False`) If `sparse=True` , return a sparse matrix.
- `subdivide` - (default `True`) If `subdivide=True` , the subdivisions for the Jordan blocks in the matrix are shown.

- `transformation` - (default `False`) If `transformation=True` , computes also the transformation matrix.
- `eigenvalues` - (default `None`) A complete set of roots, with multiplicity, of the characteristic polynomial of A, encoded as a list of pairs, each having the form (r, m) with r a root and m its multiplicity. If this is `None` , then Sage computes this list itself, but this is only possible over base rings in whose quotient fields polynomial factorization is implemented. Over all other rings, providing this list manually is the only way to compute Jordan normal forms.
- `check_input` - (default `True`) A Boolean specifying whether the list `eigenvalues` (if provided) has to be checked for correctness. Set this to `False` for a speedup if the eigenvalues are known to be correct.

NOTES:

Currently, the Jordan normal form is not computed over inexact rings in any but the trivial cases when the matrix is either 0×0 or 1×1.

In the case of exact rings, this method does not compute any generalized form of the Jordan normal form, but is only able to compute the result if the characteristic polynomial of the matrix splits over the specific base ring.

Note that the base ring must be a field or a ring with an implemented fraction field.

EXAMPLES:

```
sage: a = matrix(ZZ,4,[1, 0, 0, 0, 0, 1, 0, 0, 1, \
-1, 1, 0, 1, -1, 1, 2]); a
[ 1  0  0  0]
[ 0  1  0  0]
[ 1 -1  1  0]
[ 1 -1  1  2]
sage: a.jordan_form()
[2|0 0|0]
[-+---+-]
[0|1 1|0]
[0|0 1|0]
[-+---+-]
[0|0 0|1]
sage: a.jordan_form(subdivide=False)
[2 0 0 0]
[0 1 1 0]
[0 0 1 0]
[0 0 0 1]
sage: b = matrix(ZZ,3,3,range(9)); b
[0 1 2]
[3 4 5]
[6 7 8]
sage: b.jordan_form()
Traceback (most recent call last):
...
RuntimeError: Some eigenvalue does not exist in Rational Field.
sage: b.jordan_form(RealField(15))
Traceback (most recent call last):
...
ValueError: Jordan normal form not implemented over inexact rings.
```

Here we need to specify a field, since the eigenvalues are not defined in the smallest ring containing the matrix entries (trac ticket #14508):

```
sage: c = matrix([[0,1,0],[0,0,1],[1,0,0]]);
sage: c.jordan_form(CyclotomicField(3))
[         1|         0|         0]
[----------+----------+----------]
[         0|     zeta3|         0]
[----------+----------+----------]
[         0|         0|-zeta3 - 1]
```

If you need the transformation matrix as well as the Jordan form of `self` , then pass the option `transformation=True` . For example:

```
sage: m = matrix([[5,4,2,1],[0,1,-1,-1],[-1,-1,3,0],[1,1,-1,2]]); m
[ 5  4  2  1]
[ 0  1 -1 -1]
[-1 -1  3  0]
[ 1  1 -1  2]
sage: jf, p = m.jordan_form(transformation=True)
sage: jf
[2|0|0 0]
[-+-+---]
[0|1|0 0]
[-+-+---]
[0|0|4 1]
[0|0|0 4]
sage: ~p * m * p
[2 0 0 0]
[0 1 0 0]
[0 0 4 1]
[0 0 0 4]
```

Note that for matrices over inexact rings, we do not attempt to compute the Jordan normal form, since it is not numerically stable:

```
sage: b = matrix(ZZ,3,3,range(9))
sage: jf, p = b.jordan_form(RealField(15), transformation=True)
Traceback (most recent call last):
...
ValueError: Jordan normal form not implemented over inexact rings.
```

```
sage: evals = [(i,i) for i in range(1,6)]
sage: n = sum(range(1,6))
sage: jf = block_diagonal_matrix([jordan_block(ev,size) for ev,size in evals])
sage: p = random_matrix(ZZ,n,n)
sage: while p.rank() != n: p = random_matrix(ZZ,n,n)
sage: m = p * jf * ~p
sage: mjf, mp = m.jordan_form(transformation=True)
sage: mjf == jf
True
sage: m = diagonal_matrix([1,1,0,0])
sage: jf,P = m.jordan_form(transformation=True)
sage: jf == ~P*m*P
True
```

We verify that the bug from trac ticket #6942 is fixed:

```
sage: M =␣
↪Matrix(GF(2),[[1,0,1,0,0,0,1],[1,0,0,1,1,1,0],[1,1,0,1,1,1,1],[1,1,1,0,1,1,1],[1,1,1,0,0,1
```

```
sage: J, T = M.jordan_form(transformation=True)
sage: J
[1 1|0 0|0 0|0]
[0 1|0 0|0 0|0]
[---+---+---+-]
[0 0|1 1|0 0|0]
[0 0|0 1|0 0|0]
[---+---+---+-]
[0 0|0 0|1 1|0]
[0 0|0 0|0 1|0]
[---+---+---+-]
[0 0|0 0|0 0|1]
sage: M * T == T * J
True
sage: T.rank()
7
sage: M.rank()
7
```

We verify that the bug from trac ticket #6932 is fixed:

```
sage: M=Matrix(1,1,[1])
sage: M.jordan_form(transformation=True)
([1], [1])
```

We now go through three 10×10 matrices to exhibit cases where there are multiple blocks of the same size:

```
sage: A = matrix(QQ, [[15, 37/3, -16, -104/3, -29, -7/3, 0, 2/3, -29/3, -1/
↪3], [2, 9, -1, -6, -6, 0, 0, 0, -2, 0], [24, 74/3, -41, -208/3, -58, -23/3,␣
↪0, 4/3, -58/3, -2/3], [-6, -19, 3, 21, 19, 0, 0, 0, 6, 0], [2, 6, 3, -6, -
↪3, 1, 0, 0, -2, 0], [-96, -296/3, 176, 832/3, 232, 101/3, 0, -16/3, 232/3,␣
↪8/3], [-4, -2/3, 21, 16/3, 4, 14/3, 3, -1/3, 4/3, -25/3], [20, 26/3, -66, -
↪199/3, -42, -41/3, 0, 13/3, -55/3, -2/3], [18, 57, -9, -54, -57, 0, 0, 0, -
↪15, 0], [0, 0, 0, 0, 0, 0, 0, 0, 0, 3]]); A
[    15   37/3    -16 -104/3    -29   -7/3      0    2/3  -29/3   -1/3]
[     2      9     -1     -6     -6      0      0      0     -2      0]
[    24   74/3    -41 -208/3    -58  -23/3      0    4/3  -58/3   -2/3]
[    -6    -19      3     21     19      0      0      0      6      0]
[     2      6      3     -6     -3      1      0      0     -2      0]
[   -96 -296/3    176  832/3    232  101/3      0  -16/3  232/3    8/3]
[    -4   -2/3     21   16/3      4   14/3      3   -1/3    4/3  -25/3]
[    20   26/3    -66 -199/3    -42  -41/3      0   13/3  -55/3   -2/3]
[    18     57     -9    -54    -57      0      0      0    -15      0]
[     0      0      0      0      0      0      0      0      0      3]
sage: J, T = A.jordan_form(transformation=True); J
[3 1 0|0 0 0|0 0 0|0]
[0 3 1|0 0 0|0 0 0|0]
[0 0 3|0 0 0|0 0 0|0]
[-----+-----+-----+-]
[0 0 0|3 1 0|0 0 0|0]
[0 0 0|0 3 1|0 0 0|0]
[0 0 0|0 0 3|0 0 0|0]
[-----+-----+-----+-]
[0 0 0|0 0 0|3 1 0|0]
[0 0 0|0 0 0|0 3 1|0]
[0 0 0|0 0 0|0 0 3|0]
[-----+-----+-----+-]
```

```
[0 0 0|0 0 0|0 0 0|3]
sage: T * J * T**(-1) == A
True
sage: T.rank()
10
```

```
sage: A = matrix(QQ, [[15, 37/3, -16, -14/3, -29, -7/3, 0, 2/3, 1/3, 44/3],␣
↪[2, 9, -1, 0, -6, 0, 0, 0, 0, 3], [24, 74/3, -41, -28/3, -58, -23/3, 0, 4/
↪3, 2/3, 88/3], [-6, -19, 3, 3, 19, 0, 0, 0, 0, -9], [2, 6, 3, 0, -3, 1, 0,␣
↪0, 0, 3], [-96, -296/3, 176, 112/3, 232, 101/3, 0, -16/3, -8/3, -352/3], [-
↪4, -2/3, 21, 16/3, 4, 14/3, 3, -1/3, 4/3, -25/3], [20, 26/3, -66, -28/3, -
↪42, -41/3, 0, 13/3, 2/3, 82/3], [18, 57, -9, 0, -57, 0, 0, 0, 3, 28], [0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 3]]); A
[    15   37/3    -16  -14/3    -29   -7/3      0    2/3    1/3   44/3]
[     2      9     -1      0     -6      0      0      0      0      3]
[    24   74/3    -41  -28/3    -58  -23/3      0    4/3    2/3   88/3]
[    -6    -19      3      3     19      0      0      0      0     -9]
[     2      6      3      0     -3      1      0      0      0      3]
[   -96 -296/3    176  112/3    232  101/3      0  -16/3   -8/3 -352/3]
[    -4   -2/3     21   16/3      4   14/3      3   -1/3    4/3  -25/3]
[    20   26/3    -66  -28/3    -42  -41/3      0   13/3    2/3   82/3]
[    18     57     -9      0    -57      0      0      0      3     28]
[     0      0      0      0      0      0      0      0      0      3]
sage: J, T = A.jordan_form(transformation=True); J
[3 1 0|0 0 0|0 0|0 0]
[0 3 1|0 0 0|0 0|0 0]
[0 0 3|0 0 0|0 0|0 0]
[-----+-----+---+---]
[0 0 0|3 1 0|0 0|0 0]
[0 0 0|0 3 1|0 0|0 0]
[0 0 0|0 0 3|0 0|0 0]
[-----+-----+---+---]
[0 0 0|0 0 0|3 1|0 0]
[0 0 0|0 0 0|0 3|0 0]
[-----+-----+---+---]
[0 0 0|0 0 0|0 0|3 1]
[0 0 0|0 0 0|0 0|0 3]
sage: T * J * T**(-1) == A
True
sage: T.rank()
10
```

```
sage: A = matrix(QQ, [[15, 37/3, -16, -104/3, -29, -7/3, 35, 2/3, -29/3, -1/
↪3], [2, 9, -1, -6, -6, 0, 7, 0, -2, 0], [24, 74/3, -29, -208/3, -58, -14/3,␣
↪70, 4/3, -58/3, -2/3], [-6, -19, 3, 21, 19, 0, -21, 0, 6, 0], [2, 6, -1, -
↪6, -3, 0, 7, 0, -2, 0], [-96, -296/3, 128, 832/3, 232, 65/3, -279, -16/3,␣
↪232/3, 8/3], [0, 0, 0, 0, 0, 0, 3, 0, 0, 0], [20, 26/3, -30, -199/3, -42, -
↪14/3, 70, 13/3, -55/3, -2/3], [18, 57, -9, -54, -57, 0, 63, 0, -15, 0], [0,␣
↪0, 0, 0, 0, 0, 0, 0, 0, 3]]); A
[    15   37/3    -16 -104/3    -29   -7/3     35    2/3  -29/3   -1/3]
[     2      9     -1     -6     -6      0      7      0     -2      0]
[    24   74/3    -29 -208/3    -58  -14/3     70    4/3  -58/3   -2/3]
[    -6    -19      3     21     19      0    -21      0      6      0]
[     2      6     -1     -6     -3      0      7      0     -2      0]
[   -96 -296/3    128  832/3    232   65/3   -279  -16/3  232/3    8/3]
[     0      0      0      0      0      0      3      0      0      0]
[    20   26/3    -30 -199/3    -42  -14/3     70   13/3  -55/3   -2/3]
```

```
[    18     57     -9    -54    -57      0     63      0    -15      0]
[     0      0      0      0      0      0      0      0      0      3]
sage: J, T = A.jordan_form(transformation=True); J
[3 1 0|0 0|0 0|0 0|0]
[0 3 1|0 0|0 0|0 0|0]
[0 0 3|0 0|0 0|0 0|0]
[-----+---+---+---+-]
[0 0 0|3 1|0 0|0 0|0]
[0 0 0|0 3|0 0|0 0|0]
[-----+---+---+---+-]
[0 0 0|0 0|3 1|0 0|0]
[0 0 0|0 0|0 3|0 0|0]
[-----+---+---+---+-]
[0 0 0|0 0|0 0|3 1|0]
[0 0 0|0 0|0 0|0 3|0]
[-----+---+---+---+-]
[0 0 0|0 0|0 0|0 0|3]
sage: T * J * T**(-1) == A
True
sage: T.rank()
10
```

Verify that we smoothly move to QQ from ZZ (trac ticket #12693), i.e. we work in the vector space over the field:

```
sage: M = matrix(((2,2,2),(0,0,0),(-2,-2,-2)))
sage: J, P = M.jordan_form(transformation=True)
sage: J; P
[0 1|0]
[0 0|0]
[---+-]
[0 0|0]
[ 2  1  0]
[ 0  0  1]
[-2  0 -1]
sage: J - ~P * M * P
[0 0 0]
[0 0 0]
[0 0 0]
sage: parent(M)
Full MatrixSpace of 3 by 3 dense matrices over Integer Ring
sage: parent(J) == parent(P) == MatrixSpace(QQ, 3)
True
sage: M.jordan_form(transformation=True) == (M/1).jordan_
↪form(transformation=True)
True
```

By providing eigenvalues ourselves, we can compute the Jordan form even lacking a polynomial factorization algorithm.

```
sage: Qx = PolynomialRing(QQ, 'x11, x12, x13, x21, x22, x23, x31, x32, x33')
sage: x11, x12, x13, x21, x22, x23, x31, x32, x33 = Qx.gens()
sage: M = matrix(Qx, [[0, 0, x31], [0, 0, x21], [0, 0, 0]])    # This is a␣
↪nilpotent matrix.
sage: M.jordan_form(eigenvalues=[(0, 3)])
[0 1|0]
[0 0|0]
[---+-]
```

```
[0 0|0]
sage: M.jordan_form(eigenvalues=[(0, 2)])
Traceback (most recent call last):
...
ValueError: The provided list of eigenvalues is not correct.
sage: M.jordan_form(transformation=True, eigenvalues=[(0, 3)])
(
[0 1|0]
[0 0|0]  [x31   0   1]
[---+-]  [x21   0   0]
[0 0|0], [  0   1   0]
)
```

The base ring for the matrix needs to have a fraction field and it needs to be implemented.

```
sage: A = matrix(Integers(6), 2, 2, range(4))
sage: A.jordan_form()
Traceback (most recent call last):
...
ValueError: Matrix entries must be from a field, not Ring of integers modulo 6
```

kernel (**args*, ***kwds*)

Returns the left kernel of this matrix, as a vector space or free module. This is the set of vectors `x` such that `x*self = 0`.

Note: For the right kernel, use *right_kernel()*. The method *kernel()* is exactly equal to *left_kernel()*.

INPUT:

- `algorithm` - default: 'default' - a keyword that selects the algorithm employed. Allowable values are:
  - 'default' - allows the algorithm to be chosen automatically
  - 'generic' - naive algorithm usable for matrices over any field
  - 'flint' - FLINT library code for matrices over the rationals or the integers
  - 'pari' - PARI library code for matrices over number fields or the integers
  - 'padic' - padic algorithm from IML library for matrices over the rationals and integers
  - 'pluq' - PLUQ matrix factorization for matrices mod 2
- `basis` - default: 'echelon' - a keyword that describes the format of the basis used to construct the left kernel. Allowable values are:
  - 'echelon': the basis matrix is returned in echelon form
  - 'pivot' : each basis vector is computed from the reduced row-echelon form of `self` by placing a single one in a non-pivot column and zeros in the remaining non-pivot columns. Only available for matrices over fields.
  - 'LLL': an LLL-reduced basis. Only available for matrices over the integers.

OUTPUT:

A vector space or free module whose degree equals the number of rows in `self` and which contains all the vectors `x` such that `x*self = 0`.

If `self` has 0 rows, the kernel has dimension 0, while if `self` has 0 columns the kernel is the entire ambient vector space.

The result is cached. Requesting the left kernel a second time, but with a different basis format, will return the cached result with the format from the first computation.

Note: For much more detailed documentation of the various options see `right_kernel()`, since this method just computes the right kernel of the transpose of `self`.

EXAMPLES:

Over the rationals with a basis matrix in echelon form.

```
sage: A = matrix(QQ, [[1, 2, 4, -7, 4],
....:                 [1, 1, 0, 2, -1],
....:                 [1, 0, 3, -3, 1],
....:                 [0, -1, -1, 3, -2],
....:                 [0, 0, -1, 2, -1]])
sage: A.left_kernel()
Vector space of degree 5 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1  2 -1]
[ 0  1 -1  1 -4]
```

Over a finite field, with a basis matrix in "pivot" format.

```
sage: A = matrix(FiniteField(7), [[5, 0, 5, 2, 4],
....:                             [1, 3, 2, 3, 6],
....:                             [1, 1, 6, 5, 3],
....:                             [2, 5, 6, 0, 0]])
sage: A.kernel(basis='pivot')
Vector space of degree 4 and dimension 2 over Finite Field of size 7
User basis matrix:
[5 2 1 0]
[6 3 0 1]
```

The left kernel of a zero matrix is the entire ambient vector space whose degree equals the number of rows of `self` (i.e. everything).

```
sage: A = MatrixSpace(QQ, 3, 4)(0)
sage: A.kernel()
Vector space of degree 3 and dimension 3 over Rational Field
Basis matrix:
[1 0 0]
[0 1 0]
[0 0 1]
```

We test matrices with no rows or columns.

```
sage: A = matrix(QQ, 2, 0)
sage: A.left_kernel()
Vector space of degree 2 and dimension 2 over Rational Field
Basis matrix:
[1 0]
[0 1]
sage: A = matrix(QQ, 0, 2)
sage: A.left_kernel()
```

```
Vector space of degree 0 and dimension 0 over Rational Field
Basis matrix:
[]
```

The results are cached. Note that requesting a new format for the basis is ignored and the cached copy is returned. Work with a copy if you need a new left kernel, or perhaps investigate the `right_kernel_matrix()` method on the transpose, which does not cache its results and is more flexible.

```
sage: A = matrix(QQ, [[1,1],[2,2]])
sage: K1 = A.left_kernel()
sage: K1
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[   1 -1/2]
sage: K2 = A.left_kernel()
sage: K1 is K2
True
sage: K3 = A.left_kernel(basis='pivot')
sage: K3
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[   1 -1/2]
sage: B = copy(A)
sage: K3 = B.left_kernel(basis='pivot')
sage: K3
Vector space of degree 2 and dimension 1 over Rational Field
User basis matrix:
[-2  1]
sage: K3 is K1
False
sage: K3 == K1
True
```

kernel_on (*V*, *poly=None*, *check=True*)

Return the kernel of self restricted to the invariant subspace V. The result is a vector subspace of V, which is also a subspace of the ambient space.

INPUT:

- `V` - vector subspace
- `check` - (optional) default: True; whether to check that V is invariant under the action of self.
- `poly` - (optional) default: None; if not None, compute instead the kernel of poly(self) on V.

OUTPUT:

- a subspace

Warning: This function does *not* check that V is in fact invariant under self if check is False. With check False this function is much faster.

EXAMPLES:

```
sage: t = matrix(QQ, 4, [39, -10, 0, -12, 0, 2, 0, -1, 0, 1, -2, 0, 0, 2, 0, -
↪2]); t
[ 39 -10   0 -12]
```

```
[  0   2   0  -1]
[  0   1  -2   0]
[  0   2   0  -2]
sage: t.fcp()
(x - 39) * (x + 2) * (x^2 - 2)
sage: s = (t-39)*(t^2-2)
sage: V = s.kernel(); V
Vector space of degree 4 and dimension 3 over Rational Field
Basis matrix:
[1 0 0 0]
[0 1 0 0]
[0 0 0 1]
sage: s.restrict(V)
[0 0 0]
[0 0 0]
[0 0 0]
sage: s.kernel_on(V)
Vector space of degree 4 and dimension 3 over Rational Field
Basis matrix:
[1 0 0 0]
[0 1 0 0]
[0 0 0 1]
sage: k = t-39
sage: k.restrict(V)
[  0 -10 -12]
[  0 -37  -1]
[  0   2 -41]
sage: ker = k.kernel_on(V); ker
Vector space of degree 4 and dimension 1 over Rational Field
Basis matrix:
[   1 -2/7    0 -2/7]
sage: ker.0 * k
(0, 0, 0, 0)
```

Test that trac ticket #9425 is fixed.

```
sage: V = span([[1/7,0,0] ,[0,1,0]], ZZ); V
Free module of degree 3 and rank 2 over Integer Ring
Echelon basis matrix:
[1/7   0   0]
[  0   1   0]
sage: T = matrix(ZZ,3,[1,0,0,0,0,0,0,0,0]); T
[1 0 0]
[0 0 0]
[0 0 0]
sage: W = T.kernel_on(V); W.basis()
[
(0, 1, 0)
]
sage: W.is_submodule(V)
True
```

left_eigenmatrix ( )

Return matrices D and P, where D is a diagonal matrix of eigenvalues and P is the corresponding matrix where the rows are corresponding eigenvectors (or zero vectors) so that P*self = D*P.

EXAMPLES:

```
sage: A = matrix(QQ,3,3,range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: D, P = A.eigenmatrix_left()
sage: D
[                 0                  0                  0]
[                 0 -1.348469228349535?                  0]
[                 0                  0  13.34846922834954?]
sage: P
[                  1                  -2                   1]
[                  1  0.3101020514433644? -0.3797958971132713?]
[                  1   1.289897948556636?   1.579795897113272?]
sage: P*A == D*P
True
```

Because P is invertible, A is diagonalizable.

```
sage: A == (~P)*D*P
True
```

The matrix P may contain zero rows corresponding to eigenvalues for which the algebraic multiplicity is greater than the geometric multiplicity. In these cases, the matrix is not diagonalizable.

```
sage: A = jordan_block(2,3); A
[2 1 0]
[0 2 1]
[0 0 2]
sage: A = jordan_block(2,3)
sage: D, P = A.eigenmatrix_left()
sage: D
[2 0 0]
[0 2 0]
[0 0 2]
sage: P
[0 0 1]
[0 0 0]
[0 0 0]
sage: P*A == D*P
True
```

left_eigenspaces (*format='all'*, *var='a'*, *algebraic_multiplicity=False*)

Compute the left eigenspaces of a matrix.

Note that `eigenspaces_left()` and `left_eigenspaces()` are identical methods. Here "left" refers to the eigenvectors being placed to the left of the matrix.

INPUT:

- `self` - a square matrix over an exact field. For inexact matrices consult the numerical or symbolic matrix classes.
- `format` - default: `None`
  - `'all'` - attempts to create every eigenspace. This will always be possible for matrices with rational entries.
  - `'galois'` - for each irreducible factor of the characteristic polynomial, a single eigenspace will be output for a single root/eigenvalue for the irreducible factor.

   – `None` - Uses the 'all' format if the base ring is contained in an algebraically closed field which is implemented. Otherwise, uses the 'galois' format.

- `var` - default: 'a' - variable name used to represent elements of the root field of each irreducible factor of the characteristic polynomial. If var='a', then the root fields will be in terms of a0, a1, a2, ..., where the numbering runs across all the irreducible factors of the characteristic polynomial, even for linear factors.
- `algebraic_multiplicity` - default: False - whether or not to include the algebraic multiplicity of each eigenvalue in the output. See the discussion below.

OUTPUT:

If algebraic_multiplicity=False, return a list of pairs (e, V) where e is an eigenvalue of the matrix, and V is the corresponding left eigenspace. For Galois conjugates of eigenvalues, there may be just one representative eigenspace, depending on the `format` keyword.

If algebraic_multiplicity=True, return a list of triples (e, V, n) where e and V are as above and n is the algebraic multiplicity of the eigenvalue.

> **Warning:** Uses a somewhat naive algorithm (simply factors the characteristic polynomial and computes kernels directly over the extension field).

EXAMPLES:

We compute the left eigenspaces of a 3×3 rational matrix. First, we request all of the eigenvalues, so the results are in the field of algebraic numbers, $QQbar$. Then we request just one eigenspace per irreducible factor of the characteristic polynomial with the $galois$ keyword.

```
sage: A = matrix(QQ,3,3,range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: es = A.eigenspaces_left(format='all'); es
[
(0, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1 -2  1]),
(-1.348469228349535?, Vector space of degree 3 and dimension 1 over Algebraic␣
↪Field
User basis matrix:
[                   1   0.3101020514433644? -0.3797958971132713?]),
(13.34846922834954?, Vector space of degree 3 and dimension 1 over Algebraic␣
↪Field
User basis matrix:
[                 1 1.289897948556636? 1.579795897113272?])
]

sage: es = A.eigenspaces_left(format='galois'); es
[
(0, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1 -2  1]),
(a1, Vector space of degree 3 and dimension 1 over Number Field in a1 with␣
↪defining polynomial x^2 - 12*x - 18
User basis matrix:
[            1 1/15*a1 + 2/5 2/15*a1 - 1/5])
]
```

```
sage: es = A.eigenspaces_left(format='galois', algebraic_multiplicity=True);␣
↪es
[
(0, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1 -2  1], 1),
(a1, Vector space of degree 3 and dimension 1 over Number Field in a1 with␣
↪defining polynomial x^2 - 12*x - 18
User basis matrix:
[            1 1/15*a1 + 2/5 2/15*a1 - 1/5], 1)
]
sage: e, v, n = es[0]; v = v.basis()[0]
sage: delta = e*v - v*A
sage: abs(abs(delta)) < 1e-10
True
```

The same computation, but with implicit base change to a field.

```
sage: A = matrix(ZZ,3,3,range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: A.eigenspaces_left(format='galois')
[
(0, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1 -2  1]),
(a1, Vector space of degree 3 and dimension 1 over Number Field in a1 with␣
↪defining polynomial x^2 - 12*x - 18
User basis matrix:
[            1 1/15*a1 + 2/5 2/15*a1 - 1/5])
]
```

We compute the left eigenspaces of the matrix of the Hecke operator T_2 on level 43 modular symbols, both with all eigenvalues (the default) and with one subspace per factor.

```
sage: A = ModularSymbols(43).T(2).matrix(); A
[ 3  0  0  0  0  0 -1]
[ 0 -2  1  0  0  0  0]
[ 0 -1  1  1  0 -1  0]
[ 0 -1  0 -1  2 -1  1]
[ 0 -1  0  1  1 -1  1]
[ 0  0 -2  0  2 -2  1]
[ 0  0 -1  0  1  0 -1]
sage: A.base_ring()
Rational Field
sage: f = A.charpoly(); f
x^7 + x^6 - 12*x^5 - 16*x^4 + 36*x^3 + 52*x^2 - 32*x - 48
sage: factor(f)
(x - 3) * (x + 2)^2 * (x^2 - 2)^2
sage: A.eigenspaces_left(algebraic_multiplicity=True)
[
(3, Vector space of degree 7 and dimension 1 over Rational Field
User basis matrix:
[   1    0  1/7    0 -1/7    0 -2/7], 1),
(-2, Vector space of degree 7 and dimension 2 over Rational Field
User basis matrix:
[ 0  1  0  1 -1  1 -1]
```

```
[ 0  0  1  0 -1  2 -1], 2),
(-1.414213562373095?, Vector space of degree 7 and dimension 2 over Algebraic␣
↪Field
User basis matrix:
[                  0                   1                   0                  ␣
↪-1 0.4142135623730951?                   1                  -1]
[                  0                   0                   1                  ␣
↪ 0                  -1                   0  2.414213562373095?], 2),
(1.414213562373095?, Vector space of degree 7 and dimension 2 over Algebraic␣
↪Field
User basis matrix:
[                   0                    1                    0               ␣
↪    -1   -2.414213562373095?                    1                   -1]
[                   0                    0                    1               ␣
↪     0                   -1                    0 -0.4142135623730951?], 2)
]
sage: A.eigenspaces_left(format='galois', algebraic_multiplicity=True)
[
(3, Vector space of degree 7 and dimension 1 over Rational Field
User basis matrix:
[   1    0  1/7    0 -1/7    0 -2/7], 1),
(-2, Vector space of degree 7 and dimension 2 over Rational Field
User basis matrix:
[ 0  1  0  1 -1  1 -1]
[ 0  0  1  0 -1  2 -1], 2),
(a2, Vector space of degree 7 and dimension 2 over Number Field in a2 with␣
↪defining polynomial x^2 - 2
User basis matrix:
[      0       1       0      -1 -a2 - 1       1      -1]
[      0       0       1       0      -1       0 -a2 + 1], 2)
]
```

Next we compute the left eigenspaces over the finite field of order 11.

```
sage: A = ModularSymbols(43, base_ring=GF(11), sign=1).T(2).matrix(); A
[ 3  9  0  0]
[ 0  9  0  1]
[ 0 10  9  2]
[ 0  9  0  2]
sage: A.base_ring()
Finite Field of size 11
sage: A.charpoly()
x^4 + 10*x^3 + 3*x^2 + 2*x + 1
sage: A.eigenspaces_left(format='galois', var = 'beta')
[
(9, Vector space of degree 4 and dimension 1 over Finite Field of size 11
User basis matrix:
[0 0 1 5]),
(3, Vector space of degree 4 and dimension 1 over Finite Field of size 11
User basis matrix:
[1 6 0 6]),
(beta2, Vector space of degree 4 and dimension 1 over Univariate Quotient␣
↪Polynomial Ring in beta2 over Finite Field of size 11 with modulus x^2 + 9
User basis matrix:
[           0            1            0 5*beta2 + 10])
]
```

This method is only applicable to exact matrices. The “eigenmatrix” routines for matrices with double-precision floating-point entries (RDF , CDF) are the best alternative. (Since some platforms return eigenvectors that are the negatives of those given here, this one example is not tested here.) There are also "eigenmatrix" routines for matrices with symbolic entries.

```
sage: A = matrix(QQ, 3, 3, range(9))
sage: A.change_ring(RR).eigenspaces_left()
Traceback (most recent call last):
...
NotImplementedError: eigenspaces cannot be computed reliably for inexact␣
↪rings such as Real Field with 53 bits of precision,
consult numerical or symbolic matrix classes for other options

sage: em = A.change_ring(RDF).eigenmatrix_left()
sage: eigenvalues = em[0]; eigenvalues.dense_matrix() # abs tol 1e-13
[13.348469228349522                0.0                0.0]
[               0.0 -1.348469228349534                0.0]
[               0.0                0.0                0.0]
sage: eigenvectors = em[1]; eigenvectors # not tested
[ 0.440242867...  0.567868371...  0.695493875...]
[ 0.897878732...  0.278434036... -0.341010658...]
[ 0.408248290... -0.816496580...  0.408248290...]

sage: x, y = var('x y')
sage: S = matrix([[x, y], [y, 3*x^2]])
sage: em = S.eigenmatrix_left()
sage: eigenvalues = em[0]; eigenvalues
[3/2*x^2 + 1/2*x - 1/2*sqrt(9*x^4 - 6*x^3 + x^2 + 4*y^2)                                                                ␣
↪                                0]
[                                                       0 3/2*x^2 + 1/2*x + 1/
↪2*sqrt(9*x^4 - 6*x^3 + x^2 + 4*y^2)]
sage: eigenvectors = em[1]; eigenvectors
[                                                    1 1/2*(3*x^2 - x -␣
↪sqrt(9*x^4 - 6*x^3 + x^2 + 4*y^2))/y]
[                                                    1 1/2*(3*x^2 - x +␣
↪sqrt(9*x^4 - 6*x^3 + x^2 + 4*y^2))/y]
```

A request for 'all' the eigenvalues, when it is not possible, will raise an error. Using the 'galois' format option is more likely to be successful.

```
sage: F.<b> = FiniteField(11^2)
sage: A = matrix(F, [[b + 1, b + 1], [10*b + 4, 5*b + 4]])
sage: A.eigenspaces_left(format='all')
Traceback (most recent call last):
...
NotImplementedError: unable to construct eigenspaces for eigenvalues outside␣
↪the base field,
try the keyword option: format='galois'

sage: A.eigenspaces_left(format='galois')
[
(a0, Vector space of degree 2 and dimension 1 over Univariate Quotient␣
↪Polynomial Ring in a0 over Finite Field in b of size 11^2 with modulus x^2␣
↪+ (5*b + 6)*x + 8*b + 10
User basis matrix:
[               1 6*b*a0 + 3*b + 1])
]
```

left_eigenvectors (*extend=True*)

Compute the left eigenvectors of a matrix.

For each distinct eigenvalue, returns a list of the form (e,V,n) where e is the eigenvalue, V is a list of eigenvectors forming a basis for the corresponding left eigenspace, and n is the algebraic multiplicity of the eigenvalue.

If the option extend is set to False, then only the eigenvalues that live in the base ring are considered.

EXAMPLES: We compute the left eigenvectors of a 3×3 rational matrix.

```
sage: A = matrix(QQ,3,3,range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: es = A.eigenvectors_left(); es
[(0, [
(1, -2, 1)
], 1),
(-1.348469228349535?, [(1, 0.3101020514433644?, -0.3797958971132713?)], 1),
(13.34846922834954?, [(1, 1.289897948556636?, 1.579795897113272?)], 1)]
sage: eval, [evec], mult = es[0]
sage: delta = eval*evec - evec*A
sage: abs(abs(delta)) < 1e-10
True
```

Notice the difference between considering ring extensions or not.

```
sage: M=matrix(QQ,[[0,-1,0],[1,0,0],[0,0,2]])
sage: M.eigenvectors_left()
[(2, [
(0, 0, 1)
], 1), (-1*I, [(1, -1*I, 0)], 1), (1*I, [(1, 1*I, 0)], 1)]
sage: M.eigenvectors_left(extend=False)
[(2, [
(0, 0, 1)
], 1)]
```

left_kernel (**args*, ***kwds*)

Returns the left kernel of this matrix, as a vector space or free module. This is the set of vectors `x` such that `x*self = 0`.

Note: For the right kernel, use `right_kernel()`. The method `kernel()` is exactly equal to `left_kernel()`.

INPUT:

- `algorithm` - default: 'default' - a keyword that selects the algorithm employed. Allowable values are:
  - 'default' - allows the algorithm to be chosen automatically
  - 'generic' - naive algorithm usable for matrices over any field
  - 'flint' - FLINT library code for matrices over the rationals or the integers
  - 'pari' - PARI library code for matrices over number fields or the integers
  - 'padic' - padic algorithm from IML library for matrices over the rationals and integers
  - 'pluq' - PLUQ matrix factorization for matrices mod 2

- `basis` - default: 'echelon' - a keyword that describes the format of the basis used to construct the left kernel. Allowable values are:
  - 'echelon': the basis matrix is returned in echelon form
  - 'pivot' : each basis vector is computed from the reduced row-echelon form of `self` by placing a single one in a non-pivot column and zeros in the remaining non-pivot columns. Only available for matrices over fields.
  - 'LLL': an LLL-reduced basis. Only available for matrices over the integers.

OUTPUT:

A vector space or free module whose degree equals the number of rows in `self` and which contains all the vectors `x` such that `x*self = 0`.

If `self` has 0 rows, the kernel has dimension 0, while if `self` has 0 columns the kernel is the entire ambient vector space.

The result is cached. Requesting the left kernel a second time, but with a different basis format, will return the cached result with the format from the first computation.

Note: For much more detailed documentation of the various options see *`right_kernel()`*, since this method just computes the right kernel of the transpose of `self`.

EXAMPLES:

Over the rationals with a basis matrix in echelon form.

```
sage: A = matrix(QQ, [[1, 2, 4, -7, 4],
....:                 [1, 1, 0, 2, -1],
....:                 [1, 0, 3, -3, 1],
....:                 [0, -1, -1, 3, -2],
....:                 [0, 0, -1, 2, -1]])
sage: A.left_kernel()
Vector space of degree 5 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1  2 -1]
[ 0  1 -1  1 -4]
```

Over a finite field, with a basis matrix in "pivot" format.

```
sage: A = matrix(FiniteField(7), [[5, 0, 5, 2, 4],
....:                             [1, 3, 2, 3, 6],
....:                             [1, 1, 6, 5, 3],
....:                             [2, 5, 6, 0, 0]])
sage: A.kernel(basis='pivot')
Vector space of degree 4 and dimension 2 over Finite Field of size 7
User basis matrix:
[5 2 1 0]
[6 3 0 1]
```

The left kernel of a zero matrix is the entire ambient vector space whose degree equals the number of rows of `self` (i.e. everything).

```
sage: A = MatrixSpace(QQ, 3, 4)(0)
sage: A.kernel()
Vector space of degree 3 and dimension 3 over Rational Field
Basis matrix:
```

```
[1 0 0]
[0 1 0]
[0 0 1]
```

We test matrices with no rows or columns.

```
sage: A = matrix(QQ, 2, 0)
sage: A.left_kernel()
Vector space of degree 2 and dimension 2 over Rational Field
Basis matrix:
[1 0]
[0 1]
sage: A = matrix(QQ, 0, 2)
sage: A.left_kernel()
Vector space of degree 0 and dimension 0 over Rational Field
Basis matrix:
[]
```

The results are cached. Note that requesting a new format for the basis is ignored and the cached copy is returned. Work with a copy if you need a new left kernel, or perhaps investigate the *right_kernel_matrix()* method on the transpose, which does not cache its results and is more flexible.

```
sage: A = matrix(QQ, [[1,1],[2,2]])
sage: K1 = A.left_kernel()
sage: K1
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[   1 -1/2]
sage: K2 = A.left_kernel()
sage: K1 is K2
True
sage: K3 = A.left_kernel(basis='pivot')
sage: K3
Vector space of degree 2 and dimension 1 over Rational Field
Basis matrix:
[   1 -1/2]
sage: B = copy(A)
sage: K3 = B.left_kernel(basis='pivot')
sage: K3
Vector space of degree 2 and dimension 1 over Rational Field
User basis matrix:
[-2  1]
sage: K3 is K1
False
sage: K3 == K1
True
```

left_nullity()

Return the (left) nullity of this matrix, which is the dimension of the (left) kernel of this matrix acting from the right on row vectors.

EXAMPLES:

```
sage: M = Matrix(QQ,[[1,0,0,1],[0,1,1,0],[1,1,1,0]])
sage: M.nullity()
0
```

```
sage: M.left_nullity()
0
```

```
sage: A = M.transpose()
sage: A.nullity()
1
sage: A.left_nullity()
1
```

```
sage: M = M.change_ring(ZZ)
sage: M.nullity()
0
sage: A = M.transpose()
sage: A.nullity()
1
```

matrix_window (*row=0*, *col=0*, *nrows=-1*, *ncols=-1*, *check=1*)

Return the requested matrix window.

EXAMPLES:

```
sage: A = matrix(QQ, 3, 3, range(9))
sage: A.matrix_window(1,1, 2, 1)
Matrix window of size 2 x 1 at (1,1):
[0 1 2]
[3 4 5]
[6 7 8]
```

We test the optional check flag.

```
sage: matrix([1]).matrix_window(0,1,1,1, check=False)
Matrix window of size 1 x 1 at (0,1):
[1]
sage: matrix([1]).matrix_window(0,1,1,1)
Traceback (most recent call last):
...
IndexError: matrix window index out of range
```

Another test of bounds checking:

```
sage: matrix([1]).matrix_window(1,1,1,1)
Traceback (most recent call last):
...
IndexError: matrix window index out of range
```

maxspin (*v*)

Computes the largest integer n such that the list of vectors $S = [v, v*A, ..., v*A^n]$ are linearly independent, and returns that list.

INPUT:

- `self` - Matrix
- `v` - Vector

OUTPUT:

- `list` - list of Vectors

ALGORITHM: The current implementation just adds vectors to a vector space until the dimension doesn't grow. This could be optimized by directly using matrices and doing an efficient Echelon form. Also, when the base is Q, maybe we could simultaneously keep track of what is going on in the reduction modulo p, which might make things much faster.

EXAMPLES:

```
sage: t = matrix(QQ, 3, 3, range(9)); t
[0 1 2]
[3 4 5]
[6 7 8]
sage: v = (QQ^3).0
sage: t.maxspin(v)
[(1, 0, 0), (0, 1, 2), (15, 18, 21)]
sage: k = t.kernel(); k
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 -2  1]
sage: t.maxspin(k.0)
[(1, -2, 1)]
```

minimal_polynomial (*var='x'*, ***kwds*)

This is a synonym for `self.minpoly`

EXAMPLES:

```
sage: a = matrix(QQ, 4, 4, range(16))
sage: a.minimal_polynomial('z')
z^3 - 30*z^2 - 80*z
sage: a.minpoly()
x^3 - 30*x^2 - 80*x
```

minors (*k*)

Return the list of all $k \times k$ minors of self.

Let A be an $m \times n$ matrix and k an integer with $0 \leq k$, $k \leq m$ and $k \leq n$. A $k \times k$ minor of A is the determinant of a $k \times k$ matrix obtained from A by deleting $m - k$ rows and $n - k$ columns. There are no $k \times k$ minors of A if k is larger than either m or n.

The returned list is sorted in lexicographical row major ordering, e.g., if A is a 3×3 matrix then the minors returned are with these rows/columns: [[0, 1]x[0, 1], [0, 1]x[0, 2], [0, 1]x[1, 2], [0, 2]x[0, 1], [0, 2]x[0, 2], [0, 2]x[1, 2], [1, 2]x[0, 1], [1, 2]x[0, 2], [1, 2]x[1, 2]].

INPUT:

- `k` – integer

EXAMPLES:

```
sage: A = Matrix(ZZ,2,3,[1,2,3,4,5,6]); A
[1 2 3]
[4 5 6]
sage: A.minors(2)
[-3, -6, -3]
sage: A.minors(1)
[1, 2, 3, 4, 5, 6]
sage: A.minors(0)
[1]
sage: A.minors(5)
[]
```

```
sage: k = GF(37)
sage: P.<x0,x1,x2> = PolynomialRing(k)
sage: A = Matrix(P,2,3,[x0*x1, x0, x1, x2, x2 + 16, x2 + 5*x1 ])
sage: A.minors(2)
[x0*x1*x2 + 16*x0*x1 - x0*x2, 5*x0*x1^2 + x0*x1*x2 - x1*x2, 5*x0*x1 + x0*x2 -␣
↪x1*x2 - 16*x1]
```

This test addresses an issue raised at trac ticket #20512:

```
sage: A.minors(0)[0].parent() == P
True
```

minpoly (*var='x'*, ***kwds*)

Return the minimal polynomial of self.

This uses a simplistic - and potentially very very slow - algorithm that involves computing kernels to determine the powers of the factors of the charpoly that divide the minpoly.

EXAMPLES:

```
sage: A = matrix(GF(9,'c'), 4, [1, 1, 0,0, 0,1,0,0, 0,0,5,0, 0,0,0,5])
sage: factor(A.minpoly())
(x + 1) * (x + 2)^2
sage: A.minpoly()(A) == 0
True
sage: factor(A.charpoly())
(x + 1)^2 * (x + 2)^2
```

The default variable name is x, but you can specify another name:

```
sage: factor(A.minpoly('y'))
(y + 1) * (y + 2)^2
```

norm (*p=2*)

Return the p-norm of this matrix, where p can be 1, 2, inf, or the Frobenius norm.

INPUT:

- `self` - a matrix whose entries are coercible into CDF
- `p` - one of the following options:
- `1` - the largest column-sum norm
- `2 (default)` - the Euclidean norm
- `Infinity` - the largest row-sum norm
- `'frob'` - the Frobenius (sum of squares) norm

OUTPUT: RDF number

See also:

- `sage.misc.functional.norm()`

EXAMPLES:

```
sage: A = matrix(ZZ, [[1,2,4,3],[-1,0,3,-10]])
sage: A.norm(1)
13.0
```

```
sage: A.norm(Infinity)
14.0
sage: B = random_matrix(QQ, 20, 21)
sage: B.norm(Infinity) == (B.transpose()).norm(1)
True
```

```
sage: Id = identity_matrix(12)
sage: Id.norm(2)
1.0
sage: A = matrix(RR, 2, 2, [13,-4,-4,7])
sage: A.norm()  # rel tol 2e-16
14.999999999999998
```

Norms of numerical matrices over high-precision reals are computed by this routine. Faster routines for double precision entries from RDF or CDF are provided by the *Matrix_double_dense* class.

```
sage: A = matrix(CC, 2, 3, [3*I,4,1-I,1,2,0])
sage: A.norm('frob')
5.656854249492381
sage: A.norm(2)
5.470684443210...
sage: A.norm(1)
6.0
sage: A.norm(Infinity)
8.414213562373096
sage: a = matrix([[],[],[],[]])
sage: a.norm()
0.0
sage: a.norm(Infinity) == a.norm(1)
True
```

nullity ()

Return the (left) nullity of this matrix, which is the dimension of the (left) kernel of this matrix acting from the right on row vectors.

EXAMPLES:

```
sage: M = Matrix(QQ,[[1,0,0,1],[0,1,1,0],[1,1,1,0]])
sage: M.nullity()
0
sage: M.left_nullity()
0
```

```
sage: A = M.transpose()
sage: A.nullity()
1
sage: A.left_nullity()
1
```

```
sage: M = M.change_ring(ZZ)
sage: M.nullity()
0
sage: A = M.transpose()
sage: A.nullity()
1
```

numerical_approx (*prec=None*, *digits=None*, *algorithm=None*)

Return a numerical approximation of `self` with `prec` bits (or decimal `digits`) of precision.

INPUT:

- `prec` – precision in bits
- `digits` – precision in decimal digits (only used if `prec` is not given)
- `algorithm` – ignored for matrices

OUTPUT: A matrix converted to a real or complex field

EXAMPLES:

```
sage: d = matrix([[3, 0],[0,sqrt(2)]]) ;
sage: b = matrix([[1, -1], [2, 2]]) ; e = b * d * b.inverse();e
[ 1/2*sqrt(2) + 3/2 -1/4*sqrt(2) + 3/4]
[      -sqrt(2) + 3  1/2*sqrt(2) + 3/2]
```

```
sage: e.numerical_approx(53)
[ 2.20710678118655 0.396446609406726]
[ 1.58578643762690  2.20710678118655]
```

```
sage: e.numerical_approx(20)
[ 2.2071 0.39645]
[ 1.5858  2.2071]
```

```
sage: (e-I).numerical_approx(20)
[2.2071 - 1.0000*I           0.39645]
[           1.5858 2.2071 - 1.0000*I]
```

```
sage: M=matrix(QQ,4,[i/(i+1) for i in range(12)]);M
[    0   1/2   2/3]
[  3/4   4/5   5/6]
[  6/7   7/8   8/9]
[ 9/10 10/11 11/12]
```

```
sage: M.numerical_approx()
[0.000000000000000 0.500000000000000 0.666666666666667]
[0.750000000000000 0.800000000000000 0.833333333333333]
[0.857142857142857 0.875000000000000 0.888888888888889]
[0.900000000000000 0.909090909090909 0.916666666666667]
```

```
sage: matrix(SR, 2, 2, range(4)).n()
[0.000000000000000  1.00000000000000]
[ 2.00000000000000  3.00000000000000]
```

```
sage: numerical_approx(M)
[0.000000000000000 0.500000000000000 0.666666666666667]
[0.750000000000000 0.800000000000000 0.833333333333333]
[0.857142857142857 0.875000000000000 0.888888888888889]
[0.900000000000000 0.909090909090909 0.916666666666667]
```

permanent (*algorithm='Ryser'*)

Return the permanent of this matrix.

Let $A = (a_{i,j})$ be an $m \times n$ matrix over any commutative ring with $m \le n$. The permanent of A is

$$\mathrm{per}(A) = \sum_{\pi} a_{1,\pi(1)} a_{2,\pi(2)} \cdots a_{m,\pi(m)}$$

where the summation extends over all one-to-one functions π from $\{1, \ldots, m\}$ to $\{1, \ldots, n\}$.

The product $a_{1,\pi(1)} a_{2,\pi(2)} \cdots a_{m,\pi(m)}$ is called *diagonal product*. So the permanent of an $m \times n$ matrix A is the sum of all the diagonal products of A.

By default, this method uses Ryser's algorithm, but setting `algorithm` to "ButeraPernici" you can use the algorithm of Butera and Pernici (which is well suited for band matrices, i.e. matrices whose entries are concentrated near the diagonal).

INPUT:

- `A` – matrix of size $m \times n$ with $m \le n$
- `algorithm` – either "Ryser" (default) or "ButeraPernici". The Butera-Pernici algorithm takes advantage of presence of zeros and is very well suited for sparse matrices.

ALGORITHM:

The Ryser algorithm is implemented in the method `_permanent_ryser()`. It is a modification of theorem 7.1.1. from Brualdi and Ryser: Combinatorial Matrix Theory. Instead of deleting columns from A, we choose columns from A and calculate the product of the row sums of the selected submatrix.

The Butera-Pernici algorithm is implemented in the function `permanental_minor_polynomial()`. It takes advantage of cancellations that may occur in the computations.

EXAMPLES:

```
sage: A = ones_matrix(4,4)
sage: A.permanent()
24

sage: A = matrix(3,6,[1,1,1,1,0,0,0,1,1,1,1,0,0,0,1,1,1,1])
sage: A.permanent()
36
sage: B = A.change_ring(RR)
sage: B.permanent()
36.0000000000000
```

The permanent above is directed to the Sloane's sequence OEIS sequence A079908 ("The Dancing School Problems") for which the third term is 36:

```
sage: oeis(79908)                           # optional -- internet
A079908: Solution to the Dancing School Problem with 3 girls and n+3 boys:␣
↪f(3,n).
sage: _(3)                                  # optional -- internet
36
```

```
sage: A = matrix(4,5,[1,1,0,1,1,0,1,1,1,1,1,0,1,0,1,1,1,0,1,0])
sage: A.permanent()
32
```

A huge permanent that can not be reasonably computed with the Ryser algorithm (a 50×50 band matrix with width 5):

```
sage: n, w = 50, 5
sage: A = matrix(ZZ, n, n, lambda i,j: (i+j)%5 + 1 if abs(i-j) <= w else 0)
sage: A.permanent(algorithm="ButeraPernici")
57766972735511097036962481710892268404670105604676932908
```

See Minc: Permanents, Example 2.1, p. 5.

```
sage: A = matrix(QQ,2,2,[1/5,2/7,3/2,4/5])
sage: A.permanent()
103/175
```

```
sage: R.<a> = PolynomialRing(ZZ)
sage: A = matrix(R,2,2,[a,1,a,a+1])
sage: A.permanent()
a^2 + 2*a
```

```
sage: R.<x,y> = PolynomialRing(ZZ,2)
sage: A = matrix(R,2,2,[x, y, x^2, y^2])
sage: A.permanent()
x^2*y + x*y^2
```

AUTHORS:

- Jaap Spies (2006-02-16 and 2006-02-21)

permanental_minor (*k*, *algorithm='Ryser'*)

Return the permanental k-minor of this matrix.

The *permanental* k*-minor* of a matrix A is the sum of the permanents of all possible k by k submatrices of A. Note that the maximal permanental minor is just the permanent.

For a (0,1)-matrix A the permanental k-minor counts the number of different selections of k 1's of A with no two of the 1's on the same row and no two of the 1's on the same column.

See Brualdi and Ryser: Combinatorial Matrix Theory, p. 203. Note the typo $p_0(A) = 0$ in that reference! For applications see Theorem 7.2.1 and Theorem 7.2.4.

See also:

The method `rook_vector()` returns the list of all permanental minors.

INPUT:

- `k` – the size of the minor
- `algorithm` – either "Ryser" (default) or "ButeraPernici". The Butera-Pernici algorithm is well suited for band matrices.

EXAMPLES:

```
sage: A = matrix(4,[1,0,1,0,1,0,1,0,1,0,10,10,1,0,1,1])
sage: A.permanental_minor(2)
114
```

```
sage: A = matrix(3,6,[1,1,1,1,0,0,0,1,1,1,1,0,0,0,1,1,1,1])
sage: A.permanental_minor(0)
1
sage: A.permanental_minor(1)
12
sage: A.permanental_minor(2)
```

```
40
sage: A.permanental_minor(3)
36
```

Note that if $k = m = n$, the permanental k-minor equals $\mathrm{per}(A)$:

```
sage: A.permanent()
36
```

The permanental minors of the “complement” matrix of A is related to the permanent of A:

```
sage: m, n = 3, 6
sage: C = matrix(m, n, lambda i,j: 1 - A[i,j])
sage: sum((-1)^k * C.permanental_minor(k)*factorial(n-k)/factorial(n-m) for k␣
↪in range(m+1))
36
```

See Theorem 7.2.1 of Brualdi and Ryser: Combinatorial Matrix Theory: per(A)

AUTHORS:

- Jaap Spies (2006-02-19)

permutation_normal_form (*check=False*)

Take the set of matrices that are `self` permuted by any row and column permutation, and return the maximal one of the set where matrices are ordered lexicographically going along each row.

INPUT:

- **check – (default: False) If True return a tuple of** the maximal matrix and the permutations taking taking `self` to the maximal matrix. If `False` , return only the maximal matrix.

OUTPUT:

The maximal matrix.

EXAMPLES:

```
sage: M = matrix(ZZ, [[0, 0, 1], [1, 0, 2], [0, 0, 0]])
sage: M
[0 0 1]
[1 0 2]
[0 0 0]

sage: M.permutation_normal_form()
[2 1 0]
[1 0 0]
[0 0 0]

sage: M = matrix(ZZ, [[-1, 3], [-1, 5], [2, 4]])
sage: M
[-1  3]
[-1  5]
[ 2  4]

sage: M.permutation_normal_form(check=True)
(
[ 5 -1]
[ 4  2]
[ 3 -1],
```

```
((1,2,3), (1,2))
)
```

pfaffian (*algorithm=None*, *check=True*)

Return the Pfaffian of `self`, assuming that `self` is an alternating matrix.

INPUT:

- `algorithm` – string, the algorithm to use; currently the following algorithms have been implemented:
  - `'definition'` - using the definition given by perfect matchings
- `check` (default: `True`) – Boolean determining whether to check `self` for alternatingness and squareness. This has to be set to `False` if `self` is defined over a non-discrete ring.

The Pfaffian of an alternating matrix is defined as follows:

Let A be an alternating $k \times k$ matrix over a commutative ring. (Here, "alternating" means that $A^T = -A$ and that the diagonal entries of A are zero.) If k is odd, then the Pfaffian of the matrix A is defined to be 0. Let us now define it when k is even. In this case, set $n = k/2$ (this is an integer). For every i and j, we denote the (i, j)-th entry of A by $a_{i,j}$. Let M denote the set of all perfect matchings of the set $\{1, 2, \ldots, 2n\}$ (see `sage.combinat.perfect_matching.PerfectMatchings`). For every matching $m \in M$, define the sign $\operatorname{sign}(m)$ of m by writing m as $\{\{i_1, j_1\}, \{i_2, j_2\}, \ldots, \{i_n, j_n\}\}$ with $i_k < j_k$ for all k, and setting $\operatorname{sign}(m)$ to be the sign of the permutation $(i_1, j_1, i_2, j_2, \ldots, i_n, j_n)$ (written here in one-line notation). For every matching $m \in M$, define the weight $w(m)$ of m by writing m as $\{\{i_1, j_1\}, \{i_2, j_2\}, \ldots, \{i_n, j_n\}\}$ with $i_k < j_k$ for all k, and setting $w(m) = a_{i_1,j_1} a_{i_2,j_2} \cdots a_{i_n,j_n}$. Now, the Pfaffian of the matrix A is defined to be the sum

$$\sum_{m \in M} \operatorname{sign}(m) w(m).$$

The Pfaffian of A is commonly denoted by $\operatorname{Pf}(A)$. It is well-known that $(\operatorname{Pf}(A))^2 = \det A$ for every alternating matrix A, and that $\operatorname{Pf}(U^T A U) = \det U \cdot \operatorname{Pf}(A)$ for any $n \times n$ matrix U and any alternating $n \times n$ matrix A.

See [Knu1995], [DW1995] and [Rot2001], just to name three sources, for further properties of Pfaffians.

ALGORITHM:

The current implementation uses the definition given above. It checks alternatingness of the matrix `self` only if `check` is `True` (this is important because even if `self` is alternating, a non-discrete base ring might prevent Sage from being able to check this).

Todo

Implement faster algorithms, including a division-free one. Does [Rot2001], section 3.3 give one?

Check the implementation of the matchings used here for performance?

EXAMPLES:

A 3×3 alternating matrix has Pfaffian 0 independently of its entries:

```
sage: MSp = MatrixSpace(Integers(27), 3)
sage: A = MSp([0, 2, -3,  -2, 0, 8,  3, -8, 0])
sage: A.pfaffian()
0
sage: parent(A.pfaffian())
Ring of integers modulo 27
```

The Pfaffian of a 2×2 alternating matrix is just its northeast entry:

```
sage: MSp = MatrixSpace(QQ, 2)
sage: A = MSp([0, 4,  -4, 0])
sage: A.pfaffian()
4
sage: parent(A.pfaffian())
Rational Field
```

The Pfaffian of a 0×0 alternating matrix is 1:

```
sage: MSp = MatrixSpace(ZZ, 0)
sage: A = MSp([])
sage: A.pfaffian()
1
sage: parent(A.pfaffian())
Integer Ring
```

Let us compute the Pfaffian of a generic 4×4 alternating matrix:

```
sage: R = PolynomialRing(QQ, 'x12,x13,x14,x23,x24,x34')
sage: x12, x13, x14, x23, x24, x34 = R.gens()
sage: A = matrix(R, [[   0,  x12,  x13,  x14],
....:                [-x12,    0,  x23,  x24],
....:                [-x13, -x23,    0,  x34],
....:                [-x14, -x24, -x34,    0]])
sage: A.pfaffian()
x14*x23 - x13*x24 + x12*x34
sage: parent(A.pfaffian())
Multivariate Polynomial Ring in x12, x13, x14, x23, x24, x34 over Rational␣
↪Field
```

The Pfaffian of an alternating matrix squares to its determinant:

```
sage: A = [[0] * 6 for i in range(6)]
sage: for i in range(6):
....:     for j in range(i):
....:         u = floor(random() * 10)
....:         A[i][j] = u
....:         A[j][i] = -u
....:     A[i][i] = 0
sage: AA = Matrix(ZZ, A)
sage: AA.pfaffian() ** 2 == AA.det()
True
```

AUTHORS:

- Darij Grinberg (1 Oct 2013): first (slow) implementation.

pivot_rows ( )

Return the pivot row positions for this matrix, which are a topmost subset of the rows that span the row space and are linearly independent.

OUTPUT: a tuple of integers

EXAMPLES:

```
sage: A = matrix(QQ,3,3, [0,0,0,1,2,3,2,4,6]); A
[0 0 0]
[1 2 3]
```

```
[2 4 6]
sage: A.pivot_rows()
(1,)
sage: A.pivot_rows() # testing cached value
(1,)
```

plot (**args*, ***kwds*)

A plot of this matrix.

Each (ith, jth) matrix element is given a different color value depending on its relative size compared to the other elements in the matrix.

The tick marks drawn on the frame axes denote the (ith, jth) element of the matrix.

This method just calls `matrix_plot`. `*args` and `**kwds` are passed to `matrix_plot`.

EXAMPLES:

A matrix over ZZ colored with different grey levels:

```
sage: A = matrix([[1,3,5,1],[2,4,5,6],[1,3,5,7]])
sage: A.plot()
Graphics object consisting of 1 graphics primitive
```

Here we make a random matrix over RR and use cmap='hsv' to color the matrix elements different RGB colors (see documentation for `matrix_plot` for more information on cmaps):

```
sage: A = random_matrix(RDF, 50)
sage: plot(A, cmap='hsv')
Graphics object consisting of 1 graphics primitive
```

Another random plot, but over GF(389):

```
sage: A = random_matrix(GF(389), 10)
sage: A.plot(cmap='Oranges')
Graphics object consisting of 1 graphics primitive
```

prod_of_row_sums (*cols*)

Calculate the product of all row sums of a submatrix of A for a list of selected columns `cols`.

EXAMPLES:

```
sage: a = matrix(QQ, 2,2, [1,2,3,2]); a
[1 2]
[3 2]
sage: a.prod_of_row_sums([0,1])
15
```

Another example:

```
sage: a = matrix(QQ, 2,3, [1,2,3,2,5,6]); a
[1 2 3]
[2 5 6]
sage: a.prod_of_row_sums([1,2])
55
```

AUTHORS:

- Jaap Spies (2006-02-18)

pseudoinverse (*algorithm=None*)

Return the Moore-Penrose pseudoinverse of this matrix.

INPUT:

- `algorithm` (default: guess) – one of the following:
  - `"numpy"` – Use numpy's `linalg.pinv()` which is suitable over real or complex fields.
  - `"exact"` – Use a simple algorithm which is not numerically stable but useful over exact fields. Assume that no conjugation is needed, that the conjugate transpose is just the transpose.
  - `"exactconj"` – Like `exact` but use the conjugate transpose.

OUTPUT: a matrix

EXAMPLES:

```
sage: M = diagonal_matrix(CDF, [0, I, 1+I])
sage: M
[        0.0          0.0          0.0]
[        0.0        1.0*I          0.0]
[        0.0          0.0 1.0 + 1.0*I]
sage: M.pseudoinverse()   # tol 1e-15
[        0.0          0.0          0.0]
[        0.0       -1.0*I          0.0]
[        0.0          0.0 0.5 - 0.5*I]
```

We check the properties of the pseudoinverse over an exact field:

```
sage: M = random_matrix(QQ, 6, 3) * random_matrix(QQ, 3, 5)
sage: Mx = M.pseudoinverse()
sage: M * Mx * M == M
True
sage: Mx * M * Mx == Mx
True
sage: (M * Mx).is_symmetric()
True
sage: (Mx * M).is_symmetric()
True
```

Beware that the `exact` algorithm is not numerically stable, but the default `numpy` algorithm is:

```
sage: M = matrix(RR, 3, 3, [1,2,3,1/3,2/3,3/3,1/5,2/5,3/5])
sage: M.pseudoinverse()   # tol 1e-15
[0.0620518477661335 0.0206839492553778 0.0124103695532267]
[ 0.124103695532267 0.0413678985107557 0.0248207391064534]
[ 0.186155543298400 0.0620518477661335 0.0372311086596801]
sage: M.pseudoinverse(algorithm="numpy")   # tol 1e-15
[0.0620518477661335 0.0206839492553778 0.0124103695532267]
[ 0.124103695532267 0.0413678985107557 0.0248207391064534]
[ 0.186155543298400 0.0620518477661335 0.0372311086596801]
sage: M.pseudoinverse(algorithm="exact")
[ 0.125000000000000 0.0625000000000000 0.0312500000000000]
[ 0.250000000000000  0.125000000000000 0.0625000000000000]
[ 0.000000000000000  0.000000000000000 0.0625000000000000]
```

When multiplying the given matrix with the pseudoinverse, the result is symmetric for the `exact` algorithm or hermitian for the `exactconj` algorithm:

```
sage: M = matrix(QQbar, 2, 2, [1, sqrt(-3), -sqrt(-3), 3])
sage: M * M.pseudoinverse()
[   0.2500000000000000?  0.4330127018922193?*I]
[-0.4330127018922193?*I     0.750000000000000?]
sage: M * M.pseudoinverse(algorithm="exactconj")
[                    1/4  0.4330127018922193?*I]
[-0.4330127018922193?*I                    3/4]
sage: M * M.pseudoinverse(algorithm="exact")
[                 -1/2 0.866025403784439?*I]
[0.866025403784439?*I                  3/2]
```

For an invertible matrix, the pseudoinverse is just the inverse:

```
sage: M = matrix([[1,2], [3,4]])
sage: ~M
[  -2    1]
[ 3/2 -1/2]
sage: M.pseudoinverse()
[  -2    1]
[ 3/2 -1/2]
```

Numpy gives a strange answer due to rounding errors:

```
sage: M.pseudoinverse(algorithm="numpy")  # random
[-1286742750677287/643371375338643 1000799917193445/1000799917193444]
[  519646110850445/346430740566963  -300239975158034/600479950316067]
```

randomize (*density=1*, *nonzero=False*, **args*, ***kwds*)

Replace a proportion of the entries of a matrix by random elements, leaving the remaining entries unchanged.

Note: The locations of the entries of the matrix to change are determined randomly, with the total number of locations determined by the `density` keyword. These locations are not guaranteed to be distinct. So it is possible that the same position can be chosen multiple times, especially for a very small matrix. The exception is when `density = 1`, in which case every entry of the matrix will be changed.

INPUT:

- `density` - `float` (default: `1`); upper bound for the proportion of entries that are changed
- `nonzero` - Bool (default: `False`); if `True` , then new entries will be nonzero
- `*args,**kwds` - Remaining parameters may be passed to the `random_element` function of the base ring

EXAMPLES:

We construct the zero matrix over a polynomial ring.

```
sage: a = matrix(QQ['x'], 3); a
[0 0 0]
[0 0 0]
[0 0 0]
```

We then randomize roughly half the entries:

```
sage: a.randomize(0.5)
sage: a
```

```
[      1/2*x^2 - x - 12 1/2*x^2 - 1/95*x - 1/2                     0]
[-5/2*x^2 + 2/3*x - 1/4                      0                     0]
[          -x^2 + 2/3*x                      0                     0]
```

Now we randomize all the entries of the resulting matrix:

```
sage: a.randomize()
sage: a
[     1/3*x^2 - x + 1              -x^2 + 1              x^2 - x]
[ -1/14*x^2 - x - 1/4           -4*x - 1/5 -1/4*x^2 - 1/2*x + 4]
[ 1/9*x^2 + 5/2*x - 3      -x^2 + 3/2*x + 1    -2/7*x^2 - x - 1/2]
```

We create the zero matrix over the integers:

```
sage: a = matrix(ZZ, 2); a
[0 0]
[0 0]
```

Then we randomize it; the `x` and `y` keywords, which determine the size of the random elements, are passed on to the `random_element` method for `ZZ`.

```
sage: a.randomize(x=-2^64, y=2^64)
sage: a
[-12401200298100116246   1709403521783430739]
[ -4417091203680573707  17094769731745295000]
```

rational_form (*format='right'*, *subdivide=True*)

Returns the rational canonical form, also known as Frobenius form.

INPUT:

- `self` - a square matrix with entries from an exact field.
- `format` - default: 'right' - one of 'right', 'bottom', 'left', 'top' or 'invariants'. The first four will cause a matrix to be returned with companion matrices dictated by the keyword. The value 'invariants' will cause a list of lists to be returned, where each list contains coefficients of a polynomial associated with a companion matrix.
- `subdivide` - default: 'True' - if 'True' and a matrix is returned, then it contains subdivisions delineating the companion matrices along the diagonal.

OUTPUT:

The rational form of a matrix is a similar matrix composed of submatrices ("blocks") placed on the main diagonal. Each block is a companion matrix. Associated with each companion matrix is a polynomial. In rational form, the polynomial of one block will divide the polynomial of the next block (and thus, the polynomials of all subsequent blocks).

Rational form, also known as Frobenius form, is a canonical form. In other words, two matrices are similar if and only if their rational canonical forms are equal. The algorithm used does not provide the similarity transformation matrix (also known as the change-of-basis matrix).

Companion matrices may be written in one of four styles, and any such style may be selected with the `format` keyword. See the companion matrix constructor, `sage.matrix.constructor.companion_matrix()`, for more information about companion matrices.

If the 'invariants' value is used for the `format` keyword, then the return value is a list of lists, where each list is the coefficients of the polynomial associated with one of the companion matrices on the diagonal. These coefficients include the leading one of the monic polynomial and are ready to be coerced into any polynomial ring over the same field (see examples of this below). This return value is intended to be the most compact representation and the easiest to use for testing equality of rational forms.

Because the minimal and characteristic polynomials of a companion matrix are the associated polynomial, it is easy to see that the product of the polynomials of the blocks will be the characteristic polynomial and the final polynomial will be the minimal polynomial of the entire matrix.

ALGORITHM:

We begin with ZigZag form, which is due to Arne Storjohann and is documented at *zigzag_form()* . Then we eliminate ''corner'' entries enroute to rational form via an additional algorithm of Storjohann's [Sto2011].

EXAMPLES:

The lists of coefficients returned with the `invariants` keyword are designed to easily convert to the polynomials associated with the companion matrices. This is illustrated by the construction below of the `polys` list. Then we can test the divisibility condition on the list of polynomials. Also the minimal and characteristic polynomials are easy to determine from this list.

```
sage: A = matrix(QQ, [[ 11,  14, -15,  -4, -38, -29,  1,  23,  14, -63,  17,
↪24,  36,  32],
....:                 [ 18,   6, -17, -11, -31, -43, 12,  26,   0, -69,  11,
↪13,  17,  24],
....:                 [ 11,  16, -22,  -8, -48, -34,  0,  31,  16, -82,  26,
↪31,  39,  37],
....:                 [ -8, -18,  22,  10,  46,  33,  3, -27, -12,  70, -19, -
↪20, -42, -31],
....:                 [-13, -21,  16,  10,  52,  43,  4, -28, -25,  89, -37, -
↪20, -53, -62],
....:                 [ -2,  -6,   0,   0,   6,  10,  1,   1,  -7,  14, -11,
↪-3, -10, -18],
....:                 [ -9, -19,  -3,   4,  23,  30,  8,  -3, -27,  55, -40,
↪-5, -40, -69],
....:                 [  4,  -8,  -1,  -1,   5,  -4,  9,   5, -11,   4, -14,
↪-2, -13, -17],
....:                 [  1,  -2,  16,  -1,  19,  -2, -1, -17,   2,  19,   5, -
↪25,  -7,  14],
....:                 [  7,   7, -13,  -4, -26,  -21, 3,  18,   5, -40,   7,
↪15,  20,  14],
....:                 [ -6,  -7, -12,   4,  -1,  18,  3,   8, -11,  15, -18,
↪17, -15, -41],
....:                 [  5,  11, -11,  -3, -26, -19, -1,  14,  10, -42,  14,
↪17,  25,  23],
....:                 [-16, -15,   3,  10,  29,  45, -1, -13, -19,  71, -35,
↪-2, -35, -65],
....:                 [  4,   2,   3,  -2,  -2, -10,  1,   0,   3, -11,   6,
↪-4,   6,  17]])
sage: A.rational_form()
[   0   -4|   0    0    0    0|   0    0    0    0    0    0    0    0]
[   1    4|   0    0    0    0|   0    0    0    0    0    0    0    0]
[---------+-------------------+---------------------------------------]
[   0    0|   0    0    0   12|   0    0    0    0    0    0    0    0]
[   0    0|   1    0    0   -4|   0    0    0    0    0    0    0    0]
[   0    0|   0    1    0   -9|   0    0    0    0    0    0    0    0]
[   0    0|   0    0    1    6|   0    0    0    0    0    0    0    0]
[---------+-------------------+---------------------------------------]
[   0    0|   0    0    0    0|   0    0    0    0    0    0    0 -216]
[   0    0|   0    0    0    0|   1    0    0    0    0    0    0  108]
[   0    0|   0    0    0    0|   0    1    0    0    0    0    0  306]
```

```
[   0    0|   0    0    0    0|   0    0    1    0    0    0    0 -271]
[   0    0|   0    0    0    0|   0    0    0    1    0    0    0  -41]
[   0    0|   0    0    0    0|   0    0    0    0    1    0    0  134]
[   0    0|   0    0    0    0|   0    0    0    0    0    1    0  -64]
[   0    0|   0    0    0    0|   0    0    0    0    0    0    1   13]

sage: R = PolynomialRing(QQ, 'x')
sage: invariants = A.rational_form(format='invariants')
sage: invariants
[[4, -4, 1], [-12, 4, 9, -6, 1], [216, -108, -306, 271, 41, -134, 64, -13, 1]]
sage: polys = [R(p) for p in invariants]
sage: [p.factor() for p in polys]
[(x - 2)^2, (x - 3) * (x + 1) * (x - 2)^2, (x + 1)^2 * (x - 3)^3 * (x - 2)^3]
sage: all(polys[i].divides(polys[i+1]) for i in range(len(polys)-1))
True
sage: polys[-1] == A.minimal_polynomial(var='x')
True
sage: prod(polys) == A.characteristic_polynomial(var='x')
True
```

Rational form is a canonical form. Any two matrices are similar if and only if their rational forms are equal. By starting with Jordan canonical forms, the matrices `C` and `D` below were built as similar matrices, while `E` was built to be just slightly different. All three matrices have equal characteristic polynomials though `E` 's minimal polynomial differs.

```
sage: C = matrix(QQ, [[2,  31, -10,  -9, -125,  13,  62, -12],
....:                 [0,  48, -16, -16, -188,  20,  92, -16],
....:                 [0,   9,  -1,   2,  -33,   5,  18,   0],
....:                 [0,  15,  -5,   0,  -59,   7,  30,  -4],
....:                 [0, -21,   7,   2,   84, -10, -42,   5],
....:                 [0, -42,  14,   8,  167, -17, -84,  13],
....:                 [0, -50,  17,  10,  199, -23, -98,  14],
....:                 [0,  15,  -5,  -2,  -59,   7,  30, -2]])
sage: C.minimal_polynomial().factor()
(x - 2)^2
sage: C.characteristic_polynomial().factor()
(x - 2)^8
sage: C.rational_form()
[ 0 -4| 0  0| 0  0| 0  0]
[ 1  4| 0  0| 0  0| 0  0]
[-----+-----+-----+-----]
[ 0  0| 0 -4| 0  0| 0  0]
[ 0  0| 1  4| 0  0| 0  0]
[-----+-----+-----+-----]
[ 0  0| 0  0| 0 -4| 0  0]
[ 0  0| 0  0| 1  4| 0  0]
[-----+-----+-----+-----]
[ 0  0| 0  0| 0  0| 0 -4]
[ 0  0| 0  0| 0  0| 1  4]

sage: D = matrix(QQ, [[ -4,   3,    7,   2,  -4,   5,    7,   -3],
....:                 [ -6,   5,    7,   2,  -4,   5,    7,   -3],
....:                 [ 21, -12,   89,  25,   8,  27,   98,  -95],
....:                 [ -9,   5,  -44, -11,  -3, -13,  -48,   47],
....:                 [ 23, -13,   74,  21,  12,  22,   85,  -84],
....:                 [ 31, -18,  135,  38,  12,  47,  155, -147],
....:                 [-33,  19, -138, -39, -13, -45, -156,  151],
....:                 [ -7,   4,  -29,  -8,  -3, -10,  -34,   34]])
```

```
sage: D.minimal_polynomial().factor()
(x - 2)^2
sage: D.characteristic_polynomial().factor()
(x - 2)^8
sage: D.rational_form()
[ 0 -4| 0  0| 0  0| 0  0]
[ 1  4| 0  0| 0  0| 0  0]
[-----+-----+-----+-----]
[ 0  0| 0 -4| 0  0| 0  0]
[ 0  0| 1  4| 0  0| 0  0]
[-----+-----+-----+-----]
[ 0  0| 0  0| 0 -4| 0  0]
[ 0  0| 0  0| 1  4| 0  0]
[-----+-----+-----+-----]
[ 0  0| 0  0| 0  0| 0 -4]
[ 0  0| 0  0| 0  0| 1  4]

sage: E = matrix(QQ, [[ 0, -8,   4, -6, -2,   5, -3,  11],
....:                 [-2, -4,   2, -4, -2,   4, -2,   6],
....:                 [ 5, 14,  -7, 12,  3,  -8,  6, -27],
....:                 [-3, -8,   7, -5,  0,   2, -6,  17],
....:                 [ 0,  5,   0,  2,  4,  -4,  1,   2],
....:                 [-3, -7,   5, -6, -1,   5, -4,  14],
....:                 [ 6, 18, -10, 14,  4, -10, 10, -28],
....:                 [-2, -6,   4, -5, -1,   3,  -3, 13]])
sage: E.minimal_polynomial().factor()
(x - 2)^3
sage: E.characteristic_polynomial().factor()
(x - 2)^8
sage: E.rational_form()
[  2|  0   0|  0   0|  0   0   0]
[---+-------+-------+-----------]
[  0|  0  -4|  0   0|  0   0   0]
[  0|  1   4|  0   0|  0   0   0]
[---+-------+-------+-----------]
[  0|  0   0|  0  -4|  0   0   0]
[  0|  0   0|  1   4|  0   0   0]
[---+-------+-------+-----------]
[  0|  0   0|  0   0|  0   0   8]
[  0|  0   0|  0   0|  1   0 -12]
[  0|  0   0|  0   0|  0   1   6]
```

The principal feature of rational canonical form is that it can be computed over any field using only field operations. Other forms, such as Jordan canonical form, are complicated by the need to determine the eigenvalues of the matrix, which can lie outside the field. The following matrix has all of its eigenvalues outside the rationals - some are irrational ($\pm\sqrt{2}$) and the rest are complex ($-1 \pm 2i$).

```
sage: A = matrix(QQ,
....: [[-154,   -3,  -54,   44,   48, -244,  -19,   67, -326,   85,   355,
↪581],
....:  [ 504,   25,  156, -145, -171,  793,   99, -213, 1036, -247, -1152, -
↪1865],
....:  [ 294,   -1,  112,  -89,  -90,  469,   36, -128,  634, -160,  -695, -
↪1126],
....:  [ -49,  -32,   25,    7,   37,  -64,  -58,   12,  -42,  -14,    72,
↪106],
....:  [-261, -123,   65,   47,  169, -358, -254,   70, -309,  -29,   454,
↪673],
```

```
....: [-448, -123, -10,  109,  227, -668, -262,  163, -721,   95,   896, ␣
↪1410],
....: [  38,    7,   8,  -14,  -17,   66,    6,  -23,   73,  -29,   -78,  -
↪143],
....: [ -96,   10, -55,   37,   24, -168,   17,   56, -231,   88,   237, ␣
↪412],
....: [ 310,   67,  31,  -81, -143,  473,  143, -122,  538,  -98,  -641, -
↪1029],
....: [ 139,  -35,  99,  -49,  -18,  236,  -41,  -70,  370, -118,  -377,  -
↪619],
....: [ 243,    9,  81,  -72,  -81,  386,   43, -105,  508, -124,  -564,  -
↪911],
....: [-155,   -3, -55,   45,   50, -245,  -27,   65, -328,   77,   365, ␣
↪583]])
sage: A.characteristic_polynomial().factor()
(x^2 - 2)^2 * (x^2 + 2*x + 5)^4
sage: A.eigenvalues(extend=False)
[]
sage: A.rational_form()
[  0  -5|  0   0   0   0|  0   0   0   0   0   0]
[  1  -2|  0   0   0   0|  0   0   0   0   0   0]
[-------+---------------+-----------------------]
[  0   0|  0   0   0  10|  0   0   0   0   0   0]
[  0   0|  1   0   0   4|  0   0   0   0   0   0]
[  0   0|  0   1   0  -3|  0   0   0   0   0   0]
[  0   0|  0   0   1  -2|  0   0   0   0   0   0]
[-------+---------------+-----------------------]
[  0   0|  0   0   0   0|  0   0   0   0   0  50]
[  0   0|  0   0   0   0|  1   0   0   0   0  40]
[  0   0|  0   0   0   0|  0   1   0   0   0   3]
[  0   0|  0   0   0   0|  0   0   1   0   0 -12]
[  0   0|  0   0   0   0|  0   0   0   1   0 -12]
[  0   0|  0   0   0   0|  0   0   0   0   1  -4]
sage: F.<x> = QQ[]
sage: polys = A.rational_form(format='invariants')
sage: [F(p).factor() for p in polys]
[x^2 + 2*x + 5, (x^2 - 2) * (x^2 + 2*x + 5), (x^2 - 2) * (x^2 + 2*x + 5)^2]
```

Rational form may be computed over any field. The matrix below is an example where the eigenvalues lie outside the field.

```
sage: F.<a> = FiniteField(7^2)
sage: A = matrix(F,
....: [[5*a + 3, 4*a + 1, 6*a + 2, 2*a + 5,       6, 4*a + 5, 4*a + 5,       ␣
↪5,   a + 6,       5,  4*a + 4],
....:  [6*a + 3, 2*a + 4,       0,       6, 5*a + 5,     2*a, 5*a + 1,       ␣
↪1, 5*a + 2,     4*a, 5*a + 6],
....:  [3*a + 1, 6*a + 6,   a + 6,       2,       0, 3*a + 6, 5*a + 4, 5*a +␣
↪6, 5*a + 2,       3, 4*a + 2],
....:  [    3*a,     6*a,     3*a,     4*a, 4*a + 4, 3*a + 6,     6*a,       ␣
↪4, 3*a + 4, 6*a + 2,     4*a],
....:  [4*a + 5,   a + 1, 4*a + 3, 6*a + 5, 5*a + 2, 5*a + 2,     6*a, 4*a +␣
↪6, 6*a + 4, 5*a + 3, 3*a + 1],
....:  [    3*a,     6*a, 4*a + 1, 6*a + 2, 2*a + 5, 4*a + 6,       2,   a +␣
↪5, 2*a + 4, 2*a + 1, 2*a + 1],
....:  [4*a + 5, 3*a + 3,       6, 4*a + 1, 4*a + 3, 6*a + 3,       6, 3*a +␣
↪3,       3,   a + 3,       0],
....:  [6*a + 6,   a + 4, 2*a + 6, 3*a + 5, 4*a + 3,       2,       a, 3*a +␣
↪4,     5*a, 2*a + 5, 4*a + 3],
```

```
....:  [3*a + 5, 6*a + 2,     4*a,    a + 5,        0,     5*a, 6*a + 5, 2*a +
↪1, 3*a + 1, 3*a + 5, 4*a + 2],
....:  [3*a + 2,    a + 3, 3*a + 6,        a, 3*a + 5, 5*a + 1, 3*a + 2,    a +
↪3,    a + 2, 6*a + 1, 3*a + 3],
....:  [6*a + 6, 5*a + 1,     4*a,        2, 5*a + 5, 3*a + 5, 3*a + 1,
↪2*a,     2*a, 2*a + 4, 4*a + 2]])
sage: A.rational_form()
[  a + 2|       0       0       0|       0       0       0       0       0
↪ 0       0]
[-------+-----------------------+-------------------------------------------
↪----------]
[      0|       0       0   a + 6|       0       0       0       0       0
↪ 0       0]
[      0|       1       0 6*a + 4|       0       0       0       0       0
↪ 0       0]
[      0|       0       1 6*a + 4|       0       0       0       0       0
↪ 0       0]
[-------+-----------------------+-------------------------------------------
↪----------]
[      0|       0       0       0|       0       0       0       0       0
↪ 0     2*a]
[      0|       0       0       0|       1       0       0       0       0
↪ 0 6*a + 3]
[      0|       0       0       0|       0       1       0       0       0
↪ 0 6*a + 1]
[      0|       0       0       0|       0       0       1       0       0
↪ 0   a + 2]
[      0|       0       0       0|       0       0       0       1       0
↪ 0   a + 6]
[      0|       0       0       0|       0       0       0       0       1
↪ 0 2*a + 1]
[      0|       0       0       0|       0       0       0       0       0
↪ 1 2*a + 1]
sage: invariants = A.rational_form(format='invariants')
sage: invariants
[[6*a + 5, 1], [6*a + 1, a + 3, a + 3, 1], [5*a, a + 4, a + 6, 6*a + 5, 6*a +
↪1, 5*a + 6, 5*a + 6, 1]]
sage: R.<x> = F[]
sage: polys = [R(p) for p in invariants]
sage: [p.factor() for p in polys]
[x + 6*a + 5, (x + 6*a + 5) * (x^2 + (2*a + 5)*x + 5*a), (x + 6*a + 5) * (x^2
↪+ (2*a + 5)*x + 5*a)^3]
sage: polys[-1] == A.minimal_polynomial()
True
sage: prod(polys) == A.characteristic_polynomial()
True
sage: A.eigenvalues()
Traceback (most recent call last):
...
NotImplementedError: algebraic closures of finite fields are only implemented
↪for prime fields
```

Companion matrices may be selected as any one of four different types. See the documentation for the companion matrix constructor, `sage.matrix.constructor.companion_matrix()`, for more information.

```
sage: A = matrix(QQ, [[35, -18, -2, -45],
....:                 [22, -22, 12, -16],
```

```
....:                  [ 5, -12, 12,   4],
....:                  [16,  -6, -4, -23]])
sage: A.rational_form(format='right')
[ 2| 0  0  0]
[--+--------]
[ 0| 0  0 10]
[ 0| 1  0 -1]
[ 0| 0  1  0]
sage: A.rational_form(format='bottom')
[ 2| 0  0  0]
[--+--------]
[ 0| 0  1  0]
[ 0| 0  0  1]
[ 0|10 -1  0]
sage: A.rational_form(format='left')
[ 2| 0  0  0]
[--+--------]
[ 0| 0  1  0]
[ 0|-1  0  1]
[ 0|10  0  0]
sage: A.rational_form(format='top')
[ 2| 0  0  0]
[--+--------]
[ 0| 0 -1 10]
[ 0| 1  0  0]
[ 0| 0  1  0]
```

AUTHOR:

- Rob Beezer (2011-06-09)

restrict (*V*, *check=True*)

Returns the matrix that defines the action of self on the chosen basis for the invariant subspace V. If V is an ambient, returns self (not a copy of self).

INPUT:

- `V` - vector subspace
- `check` - (optional) default: True; if False may not check that V is invariant (hence can be faster).

OUTPUT: a matrix

> **Warning:** This function returns an nxn matrix, where V has dimension n. It does *not* check that V is in fact invariant under self, unless check is True.

EXAMPLES:

```
sage: V = VectorSpace(QQ, 3)
sage: M = MatrixSpace(QQ, 3)
sage: A = M([1,2,0, 3,4,0, 0,0,0])
sage: W = V.subspace([[1,0,0], [0,1,0]])
sage: A.restrict(W)
[1 2]
[3 4]
sage: A.restrict(W, check=True)
[1 2]
[3 4]
```

We illustrate the warning about invariance not being checked by default, by giving a non-invariant subspace. With the default check=False this function returns the 'restriction' matrix, which is meaningless as check=True reveals.

```
sage: W2 = V.subspace([[1,0,0], [0,1,1]])
sage: A.restrict(W2, check=False)
[1 2]
[3 4]
sage: A.restrict(W2, check=True)
Traceback (most recent call last):
...
ArithmeticError: subspace is not invariant under matrix
```

restrict_codomain (*V*)

Suppose that self defines a linear map from some domain to a codomain that contains V and that the image of self is contained in V. This function returns a new matrix A that represents this linear map but as a map to V, in the sense that if x is in the domain, then xA is the linear combination of the elements of the basis of V that equals v*self.

INPUT:

- `V` - vector space (space of degree `self.ncols()`) that contains the image of self.

See also:

restrict() , *restrict_domain()*

EXAMPLES:

```
sage: A = matrix(QQ,3,[1..9])
sage: V = (QQ^3).span([[1,2,3], [7,8,9]]); V
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]
sage: z = vector(QQ,[1,2,5])
sage: B = A.restrict_codomain(V); B
[1 2]
[4 5]
[7 8]
sage: z*B
(44, 52)
sage: z*A
(44, 52, 60)
sage: 44*V.0 + 52*V.1
(44, 52, 60)
```

restrict_domain (*V*)

Compute the matrix relative to the basis for V on the domain obtained by restricting self to V, but not changing the codomain of the matrix. This is the matrix whose rows are the images of the basis for V.

INPUT:

- `V` - vector space (subspace of ambient space on which self acts)

See also:

restrict()

EXAMPLES:

```
sage: V = QQ^3
sage: A = matrix(QQ,3,[1,2,0, 3,4,0, 0,0,0])
sage: W = V.subspace([[1,0,0], [1,2,3]])
sage: A.restrict_domain(W)
[1 2 0]
[3 4 0]
sage: W2 = V.subspace_with_basis([[1,0,0], [1,2,3]])
sage: A.restrict_domain(W2)
[ 1  2  0]
[ 7 10  0]
```

right_eigenmatrix ()

Return matrices D and P, where D is a diagonal matrix of eigenvalues and P is the corresponding matrix where the columns are corresponding eigenvectors (or zero vectors) so that self*P = P*D.

EXAMPLES:

```
sage: A = matrix(QQ,3,3,range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: D, P = A.eigenmatrix_right()
sage: D
[                  0                   0                   0]
[                  0 -1.348469228349535?                   0]
[                  0                   0  13.34846922834954?]
sage: P
[                   1                    1                    1]
[                  -2   0.1303061543300932?   3.069693845669907?]
[                   1 -0.7393876913398137?   5.139387691339814?]
sage: A*P == P*D
True
```

Because P is invertible, A is diagonalizable.

```
sage: A == P*D*(~P)
True
```

The matrix P may contain zero columns corresponding to eigenvalues for which the algebraic multiplicity is greater than the geometric multiplicity. In these cases, the matrix is not diagonalizable.

```
sage: A = jordan_block(2,3); A
[2 1 0]
[0 2 1]
[0 0 2]
sage: A = jordan_block(2,3)
sage: D, P = A.eigenmatrix_right()
sage: D
[2 0 0]
[0 2 0]
[0 0 2]
sage: P
[1 0 0]
[0 0 0]
[0 0 0]
sage: A*P == P*D
True
```

right_eigenspaces (*format='all'*, *var='a'*, *algebraic_multiplicity=False*)

Compute the right eigenspaces of a matrix.

Note that `eigenspaces_right()` and `right_eigenspaces()` are identical methods. Here "right" refers to the eigenvectors being placed to the right of the matrix.

INPUT:

- `self` - a square matrix over an exact field. For inexact matrices consult the numerical or symbolic matrix classes.
- `format` - default: `None`
  - `'all'` - attempts to create every eigenspace. This will always be possible for matrices with rational entries.
  - `'galois'` - for each irreducible factor of the characteristic polynomial, a single eigenspace will be output for a single root/eigenvalue for the irreducible factor.
  - `None` - Uses the 'all' format if the base ring is contained in an algebraically closed field which is implemented. Otherwise, uses the 'galois' format.
- `var` - default: 'a' - variable name used to represent elements of the root field of each irreducible factor of the characteristic polynomial. If var='a', then the root fields will be in terms of a0, a1, a2, ..., where the numbering runs across all the irreducible factors of the characteristic polynomial, even for linear factors.
- `algebraic_multiplicity` - default: False - whether or not to include the algebraic multiplicity of each eigenvalue in the output. See the discussion below.

OUTPUT:

If algebraic_multiplicity=False, return a list of pairs (e, V) where e is an eigenvalue of the matrix, and V is the corresponding left eigenspace. For Galois conjugates of eigenvalues, there may be just one representative eigenspace, depending on the `format` keyword.

If algebraic_multiplicity=True, return a list of triples (e, V, n) where e and V are as above and n is the algebraic multiplicity of the eigenvalue.

> **Warning:** Uses a somewhat naive algorithm (simply factors the characteristic polynomial and computes kernels directly over the extension field).

EXAMPLES:

Right eigenspaces are computed from the left eigenspaces of the transpose of the matrix. As such, there is a greater collection of illustrative examples at the `eigenspaces_left()`.

We compute the right eigenspaces of a 3×3 rational matrix.

```
sage: A = matrix(QQ, 3 ,3, range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: A.eigenspaces_right()
[
(0, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1 -2  1]),
(-1.348469228349535?, Vector space of degree 3 and dimension 1 over Algebraic␣
↪Field
User basis matrix:
```

```
[                   1  0.1303061543300932? -0.7393876913398137?]),
(13.34846922834954?, Vector space of degree 3 and dimension 1 over Algebraic␣
↪Field
User basis matrix:
[                 1 3.069693845669907? 5.139387691339814?])
]
sage: es = A.eigenspaces_right(format='galois'); es
[
(0, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1 -2  1]),
(a1, Vector space of degree 3 and dimension 1 over Number Field in a1 with␣
↪defining polynomial x^2 - 12*x - 18
User basis matrix:
[           1 1/5*a1 + 2/5 2/5*a1 - 1/5])
]
sage: es = A.eigenspaces_right(format='galois', algebraic_multiplicity=True);␣
↪es
[
(0, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1 -2  1], 1),
(a1, Vector space of degree 3 and dimension 1 over Number Field in a1 with␣
↪defining polynomial x^2 - 12*x - 18
User basis matrix:
[           1 1/5*a1 + 2/5 2/5*a1 - 1/5], 1)
]
sage: e, v, n = es[0]; v = v.basis()[0]
sage: delta = v*e - A*v
sage: abs(abs(delta)) < 1e-10
True
```

The same computation, but with implicit base change to a field:

```
sage: A = matrix(ZZ, 3, range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: A.eigenspaces_right(format='galois')
[
(0, Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1 -2  1]),
(a1, Vector space of degree 3 and dimension 1 over Number Field in a1 with␣
↪defining polynomial x^2 - 12*x - 18
User basis matrix:
[           1 1/5*a1 + 2/5 2/5*a1 - 1/5])
]
```

This method is only applicable to exact matrices. The "eigenmatrix" routines for matrices with double-precision floating-point entries (`RDF` , `CDF`) are the best alternative. (Since some platforms return eigenvectors that are the negatives of those given here, this one example is not tested here.) There are also "eigenmatrix" routines for matrices with symbolic entries.

```
sage: B = matrix(RR, 3, 3, range(9))
sage: B.eigenspaces_right()
Traceback (most recent call last):
...
```

```
NotImplementedError: eigenspaces cannot be computed reliably for inexact␣
↪rings such as Real Field with 53 bits of precision,
consult numerical or symbolic matrix classes for other options

sage: em = B.change_ring(RDF).eigenmatrix_right()
sage: eigenvalues = em[0]; eigenvalues.dense_matrix() # abs tol 1e-13
[13.348469228349522                0.0                0.0]
[               0.0 -1.348469228349534                0.0]
[               0.0                0.0                0.0]
sage: eigenvectors = em[1]; eigenvectors # not tested
[ 0.164763817...  0.799699663...  0.408248290...]
[ 0.505774475...  0.104205787... -0.816496580...]
[ 0.846785134... -0.591288087...  0.408248290...]

sage: x, y = var('x y')
sage: S = matrix([[x, y], [y, 3*x^2]])
sage: em = S.eigenmatrix_right()
sage: eigenvalues = em[0]; eigenvalues
[3/2*x^2 + 1/2*x - 1/2*sqrt(9*x^4 - 6*x^3 + x^2 + 4*y^2)                                                       ␣
↪                                  0]
[                                                       0 3/2*x^2 + 1/2*x + 1/
↪2*sqrt(9*x^4 - 6*x^3 + x^2 + 4*y^2)]
sage: eigenvectors = em[1]; eigenvectors
[                                                    1                                                       ␣
↪                            1]
[1/2*(3*x^2 - x - sqrt(9*x^4 - 6*x^3 + x^2 + 4*y^2))/y 1/2*(3*x^2 - x +␣
↪sqrt(9*x^4 - 6*x^3 + x^2 + 4*y^2))/y]
```

right_eigenvectors (*extend=True*)

Compute the right eigenvectors of a matrix.

For each distinct eigenvalue, returns a list of the form (e,V,n) where e is the eigenvalue, V is a list of eigenvectors forming a basis for the corresponding right eigenspace, and n is the algebraic multiplicity of the eigenvalue. If `extend = True` (the default), this will return eigenspaces over the algebraic closure of the base field where this is implemented; otherwise it will restrict to eigenvalues in the base field.

EXAMPLES: We compute the right eigenvectors of a 3×3 rational matrix.

```
sage: A = matrix(QQ,3,3,range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: es = A.eigenvectors_right(); es
[(0, [
(1, -2, 1)
], 1),
(-1.348469228349535?, [(1, 0.1303061543300932?, -0.7393876913398137?)], 1),
(13.34846922834954?, [(1, 3.069693845669907?, 5.139387691339814?)], 1)]
sage: A.eigenvectors_right(extend=False)
[(0, [
(1, -2, 1)
], 1)]
sage: eval, [evec], mult = es[0]
sage: delta = eval*evec - A*evec
sage: abs(abs(delta)) < 1e-10
True
```

right_kernel (**args*, ***kwds*)

Returns the right kernel of this matrix, as a vector space or free module. This is the set of vectors `x` such that `self*x = 0`.

Note: For the left kernel, use *left_kernel()*. The method *kernel()* is exactly equal to *left_kernel()*.

INPUT:

- `algorithm` - default: 'default' - a keyword that selects the algorithm employed. Allowable values are:
  - 'default' - allows the algorithm to be chosen automatically
  - 'generic' - naive algorithm usable for matrices over any field
  - 'flint' - FLINT library code for matrices over the rationals or the integers
  - 'pari' - PARI library code for matrices over number fields or the integers
  - 'padic' - padic algorithm from IML library for matrices over the rationals and integers
  - 'pluq' - PLUQ matrix factorization for matrices mod 2
- `basis` - default: 'echelon' - a keyword that describes the format of the basis used to construct the right kernel. Allowable values are:
  - 'echelon': the basis matrix is returned in echelon form
  - 'pivot' : each basis vector is computed from the reduced row-echelon form of `self` by placing a single one in a non-pivot column and zeros in the remaining non-pivot columns. Only available for matrices over fields.
  - 'LLL': an LLL-reduced basis. Only available for matrices over the integers.

OUTPUT:

A vector space or free module whose degree equals the number of columns in `self` and which contains all the vectors `x` such that `self*x = 0`.

If `self` has 0 columns, the kernel has dimension 0, while if `self` has 0 rows the kernel is the entire ambient vector space.

The result is cached. Requesting the right kernel a second time, but with a different basis format, will return the cached result with the format from the first computation.

Note: For more detailed documentation on the selection of algorithms used and a more flexible method for computing a basis matrix for a right kernel (rather than computing a vector space), see *right_kernel_matrix()*, which powers the computations for this method.

EXAMPLES:

```
sage: A = matrix(QQ, [[0, 0, 1, 2, 2, -5, 3],
....:                 [-1, 5, 2, 2, 1, -7, 5],
....:                 [0, 0, -2, -3, -3, 8, -5],
....:                 [-1, 5, 0, -1, -2, 1, 0]])
sage: K = A.right_kernel(); K
Vector space of degree 7 and dimension 4 over Rational Field
Basis matrix:
[ 1  0  0  0 -1 -1 -1]
[ 0  1  0  0  5  5  5]
```

```
[ 0  0  1  0 -1 -2 -3]
[ 0  0  0  1  0  1  1]
sage: A*K.basis_matrix().transpose() == zero_matrix(QQ, 4, 4)
True
```

The default is basis vectors that form a matrix in echelon form. A “pivot basis” instead has a basis matrix where the columns of an identity matrix are in the locations of the non-pivot columns of the original matrix. This alternate format is available whenever the base ring is a field.

```
sage: A = matrix(QQ, [[0, 0, 1, 2, 2, -5, 3],
....:                 [-1, 5, 2, 2, 1, -7, 5],
....:                 [0, 0, -2, -3, -3, 8, -5],
....:                 [-1, 5, 0, -1, -2, 1, 0]])
sage: A.rref()
[ 1 -5  0  0  1  1 -1]
[ 0  0  1  0  0 -1  1]
[ 0  0  0  1  1 -2  1]
[ 0  0  0  0  0  0  0]
sage: A.nonpivots()
(1, 4, 5, 6)
sage: K = A.right_kernel(basis='pivot'); K
Vector space of degree 7 and dimension 4 over Rational Field
User basis matrix:
[ 5  1  0  0  0  0  0]
[-1  0  0 -1  1  0  0]
[-1  0  1  2  0  1  0]
[ 1  0 -1 -1  0  0  1]
sage: A*K.basis_matrix().transpose() == zero_matrix(QQ, 4, 4)
True
```

Matrices may have any field as a base ring. Number fields are computed by PARI library code, matrices over $GF(2)$ are computed by the M4RI library, and matrices over the rationals are computed by the IML library. For any of these specialized cases, general-purpose code can be called instead with the keyword setting `algorithm='generic'`.

Over an arbitrary field, with two basis formats. Same vector space, different bases.

```
sage: F.<a> = FiniteField(5^2)
sage: A = matrix(F, 3, 4, [[  1,   a,      1+a,   a^3+a^5],
....:                      [  a, a^4,    a+a^4,   a^4+a^8],
....:                      [a^2, a^6, a^2+a^6, a^5+a^10]])
sage: K = A.right_kernel(); K
Vector space of degree 4 and dimension 2 over Finite Field in a of size 5^2
Basis matrix:
[      1       0 3*a + 4 2*a + 2]
[      0       1     2*a 3*a + 3]
sage: A*K.basis_matrix().transpose() == zero_matrix(F, 3, 2)
True
```

In the following test, we have to force usage of *Matrix_generic_dense*, since the option `basis = 'pivot'` would simply yield the same result as the previous test, if the optional meataxe package is installed.

```
sage: from sage.matrix.matrix_generic_dense import Matrix_generic_dense
sage: B = Matrix_generic_dense(A.parent(), A.list(), False, False)
sage: P = B.right_kernel(basis = 'pivot'); P
Vector space of degree 4 and dimension 2 over Finite Field in a of size 5^2
User basis matrix:
```

```
[        4         4         1         0]
[   a + 2 3*a + 3         0         1]
```

If the optional meataxe package is installed, we again have to make sure to work with a copy of B that has the same type as `P.basis_matrix()`:

```
sage: B.parent()(B.list())*P.basis_matrix().transpose() == zero_matrix(F, 3,␣
↪2)
True
sage: K == P
True
```

Over number fields, PARI is used by default, but general-purpose code can be requested. Same vector space, same bases, different code.:

```
sage: Q = QuadraticField(-7)
sage: a = Q.gen(0)
sage: A = matrix(Q, [[  2, 5-a,      15-a, 16+4*a],
....:                 [2+a,   a, -7 + 5*a, -3+3*a]])
sage: K = A.right_kernel(algorithm='default'); K
Vector space of degree 4 and dimension 2 over Number Field in a with defining␣
↪polynomial x^2 + 7
Basis matrix:
[                1                 0     7/88*a + 3/88 -3/176*a - 39/176]
[                0                 1   -1/88*a - 13/88  13/176*a - 7/176]
sage: A*K.basis_matrix().transpose() == zero_matrix(Q, 2, 2)
True
sage: B = copy(A)
sage: G = A.right_kernel(algorithm='generic'); G
Vector space of degree 4 and dimension 2 over Number Field in a with defining␣
↪polynomial x^2 + 7
Basis matrix:
[                1                 0     7/88*a + 3/88 -3/176*a - 39/176]
[                0                 1   -1/88*a - 13/88  13/176*a - 7/176]
sage: B*G.basis_matrix().transpose() == zero_matrix(Q, 2, 2)
True
sage: K == G
True
```

For matrices over the integers, several options are possible. The basis can be an LLL-reduced basis or an echelon basis. The pivot basis isnot available. A heuristic will decide whether to use a p-adic algorithm from the IML library or an algorithm from the PARI library. Note how specifying the algorithm can mildly influence the LLL basis.

```
sage: A = matrix(ZZ, [[0, -1, -1, 2, 9, 4, -4],
....:                  [-1, 1, 0, -2, -7, -1, 6],
....:                  [2, 0, 1, 0, 1, -5, -2],
....:                  [-1, -1, -1, 3, 10, 10, -9],
....:                  [-1, 2, 0, -3, -7, 1, 6]])
sage: A.right_kernel(basis='echelon')
Free module of degree 7 and rank 2 over Integer Ring
Echelon basis matrix:
[  1   5  -8   3  -1  -1  -1]
[  0  11 -19   5  -2  -3  -3]
sage: B = copy(A)
sage: B.right_kernel(basis='LLL')
Free module of degree 7 and rank 2 over Integer Ring
```

```
User basis matrix:
[ 2 -1  3  1  0  1  1]
[-5 -3  2 -5  1 -1 -1]
sage: C = copy(A)
sage: C.right_kernel(basis='pivot')
Traceback (most recent call last):
...
ValueError: pivot basis only available over a field, not over Integer Ring
sage: D = copy(A)
sage: D.right_kernel(algorithm='pari')
Free module of degree 7 and rank 2 over Integer Ring
Echelon basis matrix:
[  1   5  -8   3  -1  -1  -1]
[  0  11 -19   5  -2  -3  -3]
sage: E = copy(A)
sage: E.right_kernel(algorithm='padic', basis='LLL')
Free module of degree 7 and rank 2 over Integer Ring
User basis matrix:
[-2  1 -3 -1  0 -1 -1]
[ 5  3 -2  5 -1  1  1]
```

Besides the integers, rings may be as general as principal ideal domains. Results are then free modules.

```
sage: R.<y> = QQ[]
sage: A = matrix(R, [[  1,   y, 1+y^2],
....:                [y^3, y^2, 2*y^3]])
sage: K = A.right_kernel(algorithm='default', basis='echelon'); K
Free module of degree 3 and rank 1 over Univariate Polynomial Ring in y over
→Rational Field
Echelon basis matrix:
[-1 -y  1]
sage: A*K.basis_matrix().transpose() == zero_matrix(ZZ, 2, 1)
True
```

It is possible to compute a kernel for a matrix over an integral domain which is not a PID, but usually this will fail.

```
sage: D.<x> = ZZ[]
sage: A = matrix(D, 2, 2, [[x^2 - x, -x + 5],
....:                      [x^2 - 8, -x + 2]])
sage: A.right_kernel()
Traceback (most recent call last):
...
ArithmeticError: Ideal Ideal (x^2 - x, x^2 - 8) of Univariate Polynomial Ring
→in x over Integer Ring not principal
```

Matrices over non-commutative rings are not a good idea either. These are the “usual” quaternions.

```
sage: Q.<i,j,k> = QuaternionAlgebra(-1,-1)
sage: A = matrix(Q, 2, [i,j,-1,k])
sage: A.right_kernel()
Traceback (most recent call last):
...
NotImplementedError: Cannot compute a matrix kernel over Quaternion Algebra (-
→1, -1) with base ring Rational Field
```

Sparse matrices, over the rationals and the integers, use the same routines as the dense versions.

```
sage: A = matrix(ZZ, [[0, -1, 1, 1, 2],
....:                 [1, -2, 0, 1, 3],
....:                 [-1, 2, 0, -1, -3]],
....:             sparse=True)
sage: A.right_kernel()
Free module of degree 5 and rank 3 over Integer Ring
Echelon basis matrix:
[ 1  0  0  2 -1]
[ 0  1  0 -1  1]
[ 0  0  1 -3  1]
sage: B = A.change_ring(QQ)
sage: B.is_sparse()
True
sage: B.right_kernel()
Vector space of degree 5 and dimension 3 over Rational Field
Basis matrix:
[ 1  0  0  2 -1]
[ 0  1  0 -1  1]
[ 0  0  1 -3  1]
```

With no columns, the kernel can only have dimension zero. With no rows, every possible vector is in the kernel.

```
sage: A = matrix(QQ, 2, 0)
sage: A.right_kernel()
Vector space of degree 0 and dimension 0 over Rational Field
Basis matrix:
[]
sage: A = matrix(QQ, 0, 2)
sage: A.right_kernel()
Vector space of degree 2 and dimension 2 over Rational Field
Basis matrix:
[1 0]
[0 1]
```

Every vector is in the kernel of a zero matrix, the dimension is the number of columns.

```
sage: A = zero_matrix(QQ, 10, 20)
sage: A.right_kernel()
Vector space of degree 20 and dimension 20 over Rational Field
Basis matrix:
20 x 20 dense matrix over Rational Field
```

Results are cached as the right kernel of the matrix. Subsequent requests for the right kernel will return the cached result, without regard for new values of the algorithm or format keyword. Work with a copy if you need a new right kernel, or perhaps investigate the `right_kernel_matrix()` method, which does not cache its results and is more flexible.

```
sage: A = matrix(QQ, 3, 3, range(9))
sage: K1 = A.right_kernel(basis='echelon')
sage: K1
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 -2  1]
sage: K2 = A.right_kernel(basis='pivot')
sage: K2
Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
```

```
[ 1 -2  1]
sage: K1 is K2
True
sage: B = copy(A)
sage: K3 = B.kernel(basis='pivot')
sage: K3
Vector space of degree 3 and dimension 1 over Rational Field
User basis matrix:
[ 1 -2  1]
sage: K3 is K1
False
sage: K3 == K1
True
```

right_kernel_matrix (**args*, ***kwds*)

Returns a matrix whose rows form a basis for the right kernel of `self` .

INPUT:

- `algorithm` - default: 'default' - a keyword that selects the algorithm employed. Allowable values are:
  - 'default' - allows the algorithm to be chosen automatically
  - 'generic' - naive algorithm usable for matrices over any field
  - 'flint' - FLINT library code for matrices over the rationals or the integers
  - 'pari' - PARI library code for matrices over number fields or the integers
  - 'padic' - padic algorithm from IML library for matrices over the rationals and integers
  - 'pluq' - PLUQ matrix factorization for matrices mod 2
- `basis` - default: 'echelon' - a keyword that describes the format of the basis returned. Allowable values are:
  - 'echelon': the basis matrix is returned in echelon form
  - 'pivot' : each basis vector is computed from the reduced row-echelon form of `self` by placing a single one in a non-pivot column and zeros in the remaining non-pivot columns. Only available for matrices over fields.
  - 'LLL': an LLL-reduced basis. Only available for matrices over the integers.
  - 'computed': no work is done to transform the basis, it is returned exactly as provided by whichever routine actually computed the basis. Request this for the least possible computation possible, but with no guarantees about the format of the basis.

OUTPUT:

A matrix `X` whose rows are an independent set spanning the right kernel of `self` . So `self*X.transpose()` is a zero matrix.

The output varies depending on the choice of `algorithm` and the format chosen by `basis` .

The results of this routine are not cached, so you can call it again with different options to get possibly different output (like the basis format). Conversely, repeated calls on the same matrix will always start from scratch.

Note: If you want to get the most basic description of a kernel, with a minimum of overhead, then ask for the right kernel matrix with the basis format requested as 'computed'. You are then free to work with the output for whatever purpose. For a left kernel, call this method on the transpose of your matrix.

For greater convenience, plus cached results, request an actual vector space or free module with `right_kernel()` or `left_kernel()`.

EXAMPLES:

Over the Rational Numbers:

Kernels are computed by the IML library in `_right_kernel_matrix()`. Setting the *algorithm* keyword to 'default', 'padic' or unspecified will yield the same result, as there is no optional behavior. The 'computed' format of the basis vectors are exactly the negatives of the vectors in the 'pivot' format.

```
sage: A = matrix(QQ, [[1, 0, 1, -3, 1],
....:                 [-5, 1, 0, 7, -3],
....:                 [0, -1, -4, 6, -2],
....:                 [4, -1, 0, -6, 2]])
sage: C = A.right_kernel_matrix(algorithm='default', basis='computed'); C
[-1  2 -2 -1  0]
[ 1  2  0  0 -1]
sage: A*C.transpose() == zero_matrix(QQ, 4, 2)
True
sage: P = A.right_kernel_matrix(algorithm='padic', basis='pivot'); P
[ 1 -2  2  1  0]
[-1 -2  0  0  1]
sage: A*P.transpose() == zero_matrix(QQ, 4, 2)
True
sage: C == -P
True
sage: E = A.right_kernel_matrix(algorithm='default', basis='echelon'); E
[   1    0    1  1/2 -1/2]
[   0    1 -1/2 -1/4 -1/4]
sage: A*E.transpose() == zero_matrix(QQ, 4, 2)
True
```

Since the rationals are a field, we can call the general code available for any field by using the 'generic' keyword.

```
sage: A = matrix(QQ, [[1, 0, 1, -3, 1],
....:                 [-5, 1, 0, 7, -3],
....:                 [0, -1, -4, 6, -2],
....:                 [4, -1, 0, -6, 2]])
sage: G = A.right_kernel_matrix(algorithm='generic', basis='echelon'); G
[   1    0    1  1/2 -1/2]
[   0    1 -1/2 -1/4 -1/4]
sage: A*G.transpose() == zero_matrix(QQ, 4, 2)
True
```

We verify that the rational matrix code is called for both dense and sparse rational matrices, with equal result.

```
sage: A = matrix(QQ, [[1, 0, 1, -3, 1],
....:                 [-5, 1, 0, 7, -3],
....:                 [0, -1, -4, 6, -2],
....:                 [4, -1, 0, -6, 2]],
....:            sparse=False)
sage: B = copy(A).sparse_matrix()
sage: set_verbose(1)
```

```
sage: D = A.right_kernel(); D
verbose 1 (<module>) computing a right kernel for 4x5 matrix over Rational␣
↪Field
...
Vector space of degree 5 and dimension 2 over Rational Field
Basis matrix:
[   1    0    1  1/2 -1/2]
[   0    1 -1/2 -1/4 -1/4]
sage: S = B.right_kernel(); S
verbose 1 (<module>) computing a right kernel for 4x5 matrix over Rational␣
↪Field
...
Vector space of degree 5 and dimension 2 over Rational Field
Basis matrix:
[   1    0    1  1/2 -1/2]
[   0    1 -1/2 -1/4 -1/4]
sage: set_verbose(0)
sage: D == S
True
```

Over Number Fields:

Kernels are by default computed by PARI, (except for exceptions like the rationals themselves). The raw results from PARI are a pivot basis, so the *basis* keywords 'computed' and 'pivot' will return the same results.

```
sage: Q = QuadraticField(-7)
sage: a = Q.gen(0)
sage: A = matrix(Q, [[2, 5-a, 15-a, 16+4*a],
....:                [2+a, a, -7 + 5*a, -3+3*a]])
sage: C = A.right_kernel_matrix(algorithm='default', basis='computed'); C
[     -a      -3       1       0]
[     -2 -a - 1       0       1]
sage: A*C.transpose() == zero_matrix(Q, 2, 2)
True
sage: P = A.right_kernel_matrix(algorithm='pari', basis='pivot'); P
[     -a      -3       1       0]
[     -2 -a - 1       0       1]
sage: A*P.transpose() == zero_matrix(Q, 2, 2)
True
sage: E = A.right_kernel_matrix(algorithm='default', basis='echelon'); E
[                 1                  0    7/88*a + 3/88 -3/176*a - 39/176]
[                 0                  1   -1/88*a - 13/88  13/176*a - 7/176]
sage: A*E.transpose() == zero_matrix(Q, 2, 2)
True
```

We can bypass using PARI for number fields and use Sage's general code for matrices over any field. The basis vectors as computed are in pivot format.

```
sage: Q = QuadraticField(-7)
sage: a = Q.gen(0)
sage: A = matrix(Q, [[2, 5-a, 15-a, 16+4*a],[2+a, a, -7 + 5*a, -3+3*a]])
sage: G = A.right_kernel_matrix(algorithm='generic', basis='computed'); G
[     -a      -3       1       0]
[     -2 -a - 1       0       1]
sage: A*G.transpose() == zero_matrix(Q, 2, 2)
True
```

We check that number fields are handled by the right routine as part of typical right kernel computation.

```
sage: Q = QuadraticField(-7)
sage: a = Q.gen(0)
sage: A = matrix(Q, [[2, 5-a, 15-a, 16+4*a],[2+a, a, -7 + 5*a, -3+3*a]])
sage: set_verbose(1)
sage: A.right_kernel(algorithm='default')
verbose ...
verbose 1 (<module>) computing right kernel matrix over a number field for␣
↪2x4 matrix
verbose 1 (<module>) done computing right kernel matrix over a number field␣
↪for 2x4 matrix
...
Vector space of degree 4 and dimension 2 over Number Field in a with defining␣
↪polynomial x^2 + 7
Basis matrix:
[                1                 0     7/88*a + 3/88 -3/176*a - 39/176]
[                0                 1   -1/88*a - 13/88  13/176*a - 7/176]
sage: set_verbose(0)
```

Over the Finite Field of Order 2:

Kernels are computed by the M4RI library using PLUQ matrix decomposition in the `_right_kernel_matrix()` method. There are no options for the algorithm used.

```
sage: A = matrix(GF(2),[[0, 1, 1, 0, 0, 0],
....:                   [1, 0, 0, 0, 1, 1,],
....:                   [1, 0, 0, 0, 1, 1]])
sage: E = A.right_kernel_matrix(algorithm='default', format='echelon'); E
[1 0 0 0 0 1]
[0 1 1 0 0 0]
[0 0 0 1 0 0]
[0 0 0 0 1 1]
sage: A*E.transpose() == zero_matrix(GF(2), 3, 4)
True
```

Since GF(2) is a field we can route this computation to the generic code and obtain the 'pivot' form of the basis. The `algorithm` keywords, 'pluq', 'default' and unspecified, all have the same effect as there is no optional behavior.

```
sage: A = matrix(GF(2),[[0, 1, 1, 0, 0, 0],
....:                   [1, 0, 0, 0, 1, 1,],
....:                   [1, 0, 0, 0, 1, 1]])
sage: P = A.right_kernel_matrix(algorithm='generic', basis='pivot'); P
[0 1 1 0 0 0]
[0 0 0 1 0 0]
[1 0 0 0 1 0]
[1 0 0 0 0 1]
sage: A*P.transpose() == zero_matrix(GF(2), 3, 4)
True
sage: DP = A.right_kernel_matrix(algorithm='default', basis='pivot'); DP
[0 1 1 0 0 0]
[0 0 0 1 0 0]
[1 0 0 0 1 0]
[1 0 0 0 0 1]
sage: A*DP.transpose() == zero_matrix(GF(2), 3, 4)
True
sage: A.right_kernel_matrix(algorithm='pluq', basis='echelon')
[1 0 0 0 0 1]
```

```
[0 1 1 0 0 0]
[0 0 0 1 0 0]
[0 0 0 0 1 1]
```

We test that the mod 2 code is called for matrices over GF(2).

```
sage: A = matrix(GF(2),[[0, 1, 1, 0, 0, 0],
....:                   [1, 0, 0, 0, 1, 1,],
....:                   [1, 0, 0, 0, 1, 1]])
sage: set_verbose(1)
sage: A.right_kernel(algorithm='default')
verbose ...
verbose 1 (<module>) computing right kernel matrix over integers mod 2 for␣
↪3x6 matrix
verbose 1 (<module>) done computing right kernel matrix over integers mod 2␣
↪for 3x6 matrix
...
Vector space of degree 6 and dimension 4 over Finite Field of size 2
Basis matrix:
[1 0 0 0 0 1]
[0 1 1 0 0 0]
[0 0 0 1 0 0]
[0 0 0 0 1 1]
sage: set_verbose(0)
```

Over Arbitrary Fields:

For kernels over fields not listed above, totally general code will compute a set of basis vectors in the pivot format. These could be returned as a basis in echelon form.

```
sage: F.<a> = FiniteField(5^2)
sage: A = matrix(F, 3, 4, [[  1,   a,     1+a,  a^3+a^5],
....:                      [  a, a^4,   a+a^4,  a^4+a^8],
....:                      [a^2, a^6, a^2+a^6, a^5+a^10]])
sage: P = A.right_kernel_matrix(algorithm='default', basis='pivot'); P
[      4       4       1       0]
[  a + 2 3*a + 3       0       1]
sage: A*P.transpose() == zero_matrix(F, 3, 2)
True
sage: E = A.right_kernel_matrix(algorithm='default', basis='echelon'); E
[      1       0 3*a + 4 2*a + 2]
[      0       1     2*a 3*a + 3]
sage: A*E.transpose() == zero_matrix(F, 3, 2)
True
```

This general code can be requested for matrices over any field with the `algorithm` keyword 'generic'. Normally, matrices over the rationals would be handled by specific routines from the IML library. The default format is an echelon basis, but a pivot basis may be requested, which is identical to the computed basis.

```
sage: A = matrix(QQ, 3, 4, [[1,3,-2,4],
....:                       [2,0,2,2],
....:                       [-1,1,-2,0]])
sage: G = A.right_kernel_matrix(algorithm='generic'); G
[   1    0 -1/2 -1/2]
[   0    1  1/2 -1/2]
sage: A*G.transpose() == zero_matrix(QQ, 3, 2)
True
```

```
sage: C = A.right_kernel_matrix(algorithm='generic', basis='computed'); C
[-1  1  1  0]
[-1 -1  0  1]
sage: A*C.transpose() == zero_matrix(QQ, 3, 2)
True
```

We test that the generic code is called for matrices over fields, lacking any more specific routine.

```
sage: F.<a> = FiniteField(5^2)
sage: A = matrix(F, 3, 4, [[  1,    a,      1+a,   a^3+a^5],
....:                      [  a, a^4,    a+a^4,   a^4+a^8],
....:                      [a^2, a^6, a^2+a^6, a^5+a^10]])
sage: set_verbose(1)
sage: A.right_kernel(algorithm='default')
verbose ...
verbose 1 (<module>) computing right kernel matrix over an arbitrary field␣
↪for 3x4 matrix
...
Vector space of degree 4 and dimension 2 over Finite Field in a of size 5^2
Basis matrix:
[      1       0 3*a + 4 2*a + 2]
[      0       1     2*a 3*a + 3]
sage: set_verbose(0)
```

Over the Integers:

Either the IML or PARI libraries are used to provide a set of basis vectors. The `algorithm` keyword can be used to select either, or when set to 'default' a heuristic will choose between the two. Results can be returned in the 'compute' format, straight out of the libraries. Unique to the integers, the basis vectors can be returned as an LLL basis. Note the similarities and differences in the results. The 'pivot' format is not available, since the integers are not a field.

```
sage: A = matrix(ZZ, [[8, 0, 7, 1, 3, 4, 6],
....:                 [4, 0, 3, 4, 2, 7, 7],
....:                 [1, 4, 6, 1, 2, 8, 5],
....:                 [0, 3, 1, 2, 3, 6, 2]])

sage: X = A.right_kernel_matrix(algorithm='default', basis='echelon'); X
[  1  12   3  14  -3 -10   1]
[  0  35   0  25  -1 -31  17]
[  0   0   7  12  -3  -1  -8]
sage: A*X.transpose() == zero_matrix(ZZ, 4, 3)
True

sage: X = A.right_kernel_matrix(algorithm='padic', basis='LLL'); X
[ -3  -1   5   7   2  -3  -2]
[  3   1   2   5  -5   2  -6]
[ -4 -13   2  -7   5   7  -3]
sage: A*X.transpose() == zero_matrix(ZZ, 4, 3)
True

sage: X = A.right_kernel_matrix(algorithm='pari', basis='computed'); X
[ -3  -1   5   7   2  -3  -2]
[  3   1   2   5  -5   2  -6]
[ -4 -13   2  -7   5   7  -3]
sage: A*X.transpose() == zero_matrix(ZZ, 4, 3)
True
```

```
sage: X = A.right_kernel_matrix(algorithm='padic', basis='computed'); X
[ 265  345 -178   17 -297    0    0]
[-242 -314  163  -14  271   -1    0]
[ -36  -47   25   -1   40    0   -1]
sage: A*X.transpose() == zero_matrix(ZZ, 4, 3)
True
```

We test that the code for integer matrices is called for matrices defined over the integers, both dense and sparse, with equal result.

```
sage: A = matrix(ZZ, [[8, 0, 7, 1, 3, 4, 6],
....:                 [4, 0, 3, 4, 2, 7, 7],
....:                 [1, 4, 6, 1, 2, 8, 5],
....:                 [0, 3, 1, 2, 3, 6, 2]],
....:            sparse=False)
sage: B = copy(A).sparse_matrix()
sage: set_verbose(1)
sage: D = A.right_kernel(); D
verbose 1 (<module>) computing a right kernel for 4x7 matrix over Integer Ring
verbose 1 (<module>) computing right kernel matrix over the integers for 4x7␣
↪matrix
...
verbose 1 (<module>) done computing right kernel matrix over the integers for␣
↪4x7 matrix
...
Free module of degree 7 and rank 3 over Integer Ring
Echelon basis matrix:
[  1  12   3  14  -3 -10   1]
[  0  35   0  25  -1 -31  17]
[  0   0   7  12  -3  -1  -8]
sage: S = B.right_kernel(); S
verbose 1 (<module>) computing a right kernel for 4x7 matrix over Integer Ring
verbose 1 (<module>) computing right kernel matrix over the integers for 4x7␣
↪matrix
...
verbose 1 (<module>) done computing right kernel matrix over the integers for␣
↪4x7 matrix
...
Free module of degree 7 and rank 3 over Integer Ring
Echelon basis matrix:
[  1  12   3  14  -3 -10   1]
[  0  35   0  25  -1 -31  17]
[  0   0   7  12  -3  -1  -8]
sage: set_verbose(0)
sage: D == S
True
```

Over Principal Ideal Domains:

Kernels can be computed using Smith normal form. Only the default algorithm is available, and the 'pivot' basis format is not available.

```
sage: R.<y> = QQ[]
sage: A = matrix(R, [[  1,   y, 1+y^2],
....:                [y^3, y^2, 2*y^3]])
sage: E = A.right_kernel_matrix(algorithm='default', basis='echelon'); E
[-1 -y  1]
sage: A*E.transpose() == zero_matrix(ZZ, 2, 1)
```

```
True
```

It can be computationally expensive to determine if an integral domain is a principal ideal domain. The Smith normal form routine can fail for non-PIDs, as in this example.

```
sage: D.<x> = ZZ[]
sage: A = matrix(D, 2, 2, [[x^2 - x, -x + 5],
....:                      [x^2 - 8, -x + 2]])
sage: A.right_kernel_matrix()
Traceback (most recent call last):
...
ArithmeticError: Ideal Ideal (x^2 - x, x^2 - 8) of Univariate Polynomial Ring␣
↪in x over Integer Ring not principal
```

We test that the domain code is called for domains that lack any extra structure.

```
sage: R.<y> = QQ[]
sage: A = matrix(R, [[  1,   y, 1+y^2],
....:                [y^3, y^2, 2*y^3]])
sage: set_verbose(1)
sage: A.right_kernel(algorithm='default', basis='echelon')
verbose ...
verbose 1 (<module>) computing right kernel matrix over a domain for 2x3␣
↪matrix
verbose 1 (<module>) done computing right kernel matrix over a domain for 2x3␣
↪matrix
...
Free module of degree 3 and rank 1 over Univariate Polynomial Ring in y over␣
↪Rational Field
Echelon basis matrix:
[-1 -y  1]
sage: set_verbose(0)
```

Trivial Cases:

We test two trivial cases. Any possible values for the keywords (`algorithm` , `basis`) will return identical results.

```
sage: A = matrix(ZZ, 0, 2)
sage: A.right_kernel_matrix()
[1 0]
[0 1]
sage: A = matrix(FiniteField(7), 2, 0)
sage: A.right_kernel_matrix().parent()
Full MatrixSpace of 0 by 0 dense matrices over Finite Field of size 7
```

AUTHOR:

- Rob Beezer (2011-02-05)

right_nullity ( )

Return the right nullity of this matrix, which is the dimension of the right kernel.

EXAMPLES:

```
sage: A = MatrixSpace(QQ,3,2)(range(6))
sage: A.right_nullity()
0
```

```
sage: A = matrix(ZZ,3,3,range(9))
sage: A.right_nullity()
1
```

rook_vector (*algorithm='ButeraPernici'*, *complement=False*, *use_complement=None*)
Return the rook vector of this matrix.

Let A be an m by n (0,1)-matrix. We identify A with a chessboard where rooks can be placed on the fields (i, j) with $A_{i,j} = 1$. The number $r_k = p_k(A)$ (the permanental k-minor) counts the number of ways to place k rooks on this board so that no rook can attack another.

The *rook vector* of the matrix A is the list consisting of $r_0, r_1, \ldots, r_h$, where $h = min(m, n)$. The *rook polynomial* is defined by $r(x) = \sum_{k=0}^{h} r_k x^k$.

The rook vector can be generalized to matrices defined over any rings using permanental minors. Among the available algorithms, only "Godsil" needs the condition on the entries to be either 0 or 1.

See Wikipedia article Rook_polynomial for more information and also the method `permanental_minor()` to compute individual permanental minor.

See also `sage.matrix.matrix2.permanental_minor_polynomial` and the graph method `matching_polynomial`.

INPUT:

- `self` – an m by n matrix
- `algorithm` – a string which must be either "Ryser" or "ButeraPernici" (default) or "Godsil"; Ryser one might be faster on simple and small instances. Godsil only accepts input in 0,1.
- `complement` – boolean (default: `False`) whether we consider the rook vector of the complement matrix. If set to `True` then the matrix must have entries in {0, 1} and the complement matrix is the one for which the 0's are replaced by 1's and 1's by 0's.
- `use_complement` – Boolean (default: `None`) whether to compute the rook vector of a (0,1)-matrix from its complement. By default this is determined by the density of ones in the matrix.

EXAMPLES:

The standard chessboard is an 8 by 8 grid in which any positions is allowed. In that case one gets that the number of ways to position 4 non-attacking rooks is 117600 while for 8 rooks it is 40320:

```
sage: ones_matrix(8,8).rook_vector()
[1, 64, 1568, 18816, 117600, 376320, 564480, 322560, 40320]
```

These numbers are the coefficients of a modified Laguerre polynomial:

```
sage: x = polygen(QQ)
sage: factorial(8) * laguerre(8,-x)
x^8 + 64*x^7 + 1568*x^6 + 18816*x^5 + 117600*x^4 + 376320*x^3 +
564480*x^2 + 322560*x + 40320
```

The number of derangements of length n is the permanent of a matrix with 0 on the diagonal and 1 elsewhere; for $n = 21$ it is 18795307255050944540 (see OEIS sequence A000166):

sage: A = identity_matrix(21) sage: A.rook_vector(complement=True)[-1] 18795307255050944540 sage: Derangements(21).cardinality() 18795307255050944540

An other example that we convert into a rook polynomial:

```
sage: A = matrix(3,6, [1,1,1,1,0,0,0,1,1,1,1,0,0,0,1,1,1,1])
sage: A
[1 1 1 1 0 0]
[0 1 1 1 1 0]
[0 0 1 1 1 1]
sage: A.rook_vector()
[1, 12, 40, 36]

sage: R = PolynomialRing(ZZ, 'x')
sage: R(A.rook_vector())
36*x^3 + 40*x^2 + 12*x + 1
```

Different algorithms are available:

```
sage: A = matrix([[1,0,0,1],[0,1,1,0],[0,1,1,0],[1,0,0,1]])
sage: A.rook_vector(algorithm="ButeraPernici")
[1, 8, 20, 16, 4]
sage: A.rook_vector(algorithm="Ryser")
[1, 8, 20, 16, 4]
sage: A.rook_vector(algorithm="Godsil")
[1, 8, 20, 16, 4]
```

When the matrix A has more ones then zeroes it is usually faster to compute the rook polynomial of the complementary matrix, with zeroes and ones interchanged, and use the inclusion-exclusion theorem, giving for a $m \times n$ matrix A with complementary matrix B

$$r_k(A) = \sum_{j=0}^{k} (-1)^j \binom{m-j}{k-j} \binom{n-j}{k-j} (k-j)! r_j(B)$$

see [Rio1958] or the introductory text [AS2011]. This can be done setting the argument `use_complement` to `True`.

An example with an exotic matrix (for which only Butera-Pernici and Ryser algorithms are available):

```
sage: R.<x,y> = PolynomialRing(GF(5))
sage: A = matrix(R,[[1,x,y],[x*y,x**2+y,0]])
sage: A.rook_vector(algorithm="ButeraPernici")
[1, x^2 + x*y + x + 2*y + 1, 2*x^2*y + x*y^2 + x^2 + y^2 + y]
sage: A.rook_vector(algorithm="Ryser")
[1, x^2 + x*y + x + 2*y + 1, 2*x^2*y + x*y^2 + x^2 + y^2 + y]
sage: A.rook_vector(algorithm="Godsil")
Traceback (most recent call last):
...
ValueError: coefficients must be zero or one, but we have 'x' in position␣
↪(0,1).
sage: B = A.transpose()
sage: B.rook_vector(algorithm="ButeraPernici")
[1, x^2 + x*y + x + 2*y + 1, 2*x^2*y + x*y^2 + x^2 + y^2 + y]
sage: B.rook_vector(algorithm="Ryser")
[1, x^2 + x*y + x + 2*y + 1, 2*x^2*y + x*y^2 + x^2 + y^2 + y]
```

AUTHORS:

- Jaap Spies (2006-02-24)
- Mario Pernici (2014-07-01)

row_module (*base_ring=None*)

Return the free module over the base ring spanned by the rows of self.

EXAMPLES:

```
sage: A = MatrixSpace(IntegerRing(), 2)([1,2,3,4])
sage: A.row_module()
Free module of degree 2 and rank 2 over Integer Ring
Echelon basis matrix:
[1 0]
[0 2]
```

row_reduced_form (*transformation=None*)

This function computes a row reduced form of a matrix over a rational function field $k(x)$, where k is a field.

A matrix M over $k(x)$ is row reduced if the (row-wise) leading term matrix of dM has the same rank as M, where $d \in k[x]$ is a minimal degree polynomial such that dM is in $k[x]$. The (row-wise) leading term matrix of a polynomial matrix M_0 is matrix over k whose (i, j)'th entry is the x^{d_i} coefficient of $M_0[i, j]$, where d_i is the greatest degree among polynomials in the i'th row of M_0.

INPUT:

- `transformation` – A boolean (default: `False`). If this is set to `True` , the transformation matrix U will be returned as well: this is an invertible matrix over $k(x)$ such that `self` equals UW, where W is the output matrix.

OUTPUT:

- W - a matrix over the same ring as $self$ (i.e. either $k(x)$ or $k[x]$ for a field k) giving a row reduced form of `self` .

EXAMPLES:

The routine expects matrices over the rational function field. One can also provide matrices over the ring of polynomials (whose quotient field is the rational function field).

```
sage: R.<t> = GF(3)['t']
sage: K = FractionField(R)
sage: M = matrix([[(t-1)^2/t],[(t-1)]])
sage: M.row_reduced_form()
doctest:...: DeprecationWarning: Row reduced form will soon be supported only
↪for matrices of polynomials.
See http://trac.sagemath.org/21024 for details.
[        0]
[(t + 2)/t]
```

If `self` is an $n \times 1$ matrix with at least one non-zero entry, W has a single non-zero entry and that entry is a scalar multiple of the greatest-common-divisor of the entries of `self` .

```
sage: M1 = matrix([[t*(t-1)*(t+1)],[t*(t-2)*(t+2)],[t]])
sage: output1 = M1.row_reduced_form()
sage: output1
[0]
[0]
[t]
```

We check that the output is the same for a matrix M if its entries are rational functions instead of polynomials. We also check that the type of the output follows the documentation. See trac ticket #9063

```
sage: M2 = M1.change_ring(K)
sage: output2 = M2.row_reduced_form()
sage: output1 == output2
```

```
True
sage: output1.base_ring() is R
True
sage: output2.base_ring() is K
True
```

The following is the first half of example 5 in [Hes2002] *except* that we have transposed `self`; [Hes2002] uses column operations and we use row.

```
sage: R.<t> = QQ['t']
sage: M = matrix([[t^3 - t,t^2 - 2],[0,t]]).transpose()
sage: M.row_reduced_form(transformation=False)
[      t    -t^2]
[t^2 - 2       t]
```

The next example demonstrates what happens when `self` is a zero matrix.

```
sage: R.<t> = GF(5)['t']
sage: K = FractionField(R)
sage: M = matrix([[K(0),K(0)],[K(0),K(0)]])
sage: M.row_reduced_form()
[0 0]
[0 0]
```

In the following example, `self` has more rows than columns.

```
sage: R.<t> = QQ['t']
sage: M = matrix([[t,t,t],[0,0,t]])
sage: M.row_reduced_form()
[ t  t  t]
[-t -t  0]
```

The next example shows that M must be a matrix with coefficients in $k(t)$ or in $k[t]$ for some field k.

```
sage: M = matrix([[1,0],[1,1]])
sage: M.row_reduced_form()
Traceback (most recent call last):
...
TypeError: the coefficients of M must lie in a univariate polynomial ring␣
↪over a field

sage: PZ.<y> = ZZ[]
sage: M = matrix([[y,0],[1,y]])
sage: M.row_reduced_form()
Traceback (most recent call last):
...
TypeError: the coefficients of M must lie in a univariate polynomial ring␣
↪over a field
```

The last example shows the usage of the transformation parameter.

```
sage: Fq.<a> = GF(2^3)
sage: Fx.<x> = Fq[]
sage: A = matrix(Fx,[[x^2+a,x^4+a],[x^3,a*x^4]])
sage: W,U = A.row_reduced_form(transformation=True);
sage: W,U
(
[          x^2 + a           x^4 + a]  [1 0]
```

```
[x^3 + a*x^2 + a^2                 a^2], [a 1]
)
sage: U*W == A
True
sage: U.is_invertible()
True
```

NOTES:

- For consistency with LLL and other algorithms in Sage, we have opted for row operations; however, some references e.g. [Hes2002] transpose and use column operations.

REFERENCES:

- [Hes2002]
- [Kal1980]

row_space (*base_ring=None*)

Return the row space of this matrix. (Synonym for self.row_module().)

EXAMPLES:

```
sage: t = matrix(QQ, 3, 3, range(9)); t
[0 1 2]
[3 4 5]
[6 7 8]
sage: t.row_space()
Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2]
```

```
sage: m = Matrix(Integers(5),2,2,[2,2,2,2]);
sage: m.row_space()
Vector space of degree 2 and dimension 1 over Ring of integers modulo 5
Basis matrix:
[1 1]
```

rref (**args*, ***kwds*)

Return the reduced row echelon form of the matrix, considered as a matrix over a field.

If the matrix is over a ring, then an equivalent matrix is constructed over the fraction field, and then row reduced.

All arguments are passed on to `echelon_form()` .

Note: Because the matrix is viewed as a matrix over a field, every leading coefficient of the returned matrix will be one and will be the only nonzero entry in its column.

EXAMPLES:

```
sage: A=matrix(3,range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: A.rref()
[ 1  0 -1]
```

```
[ 0  1  2]
[ 0  0  0]
```

Note that there is a difference between `rref()` and `echelon_form()` when the matrix is not over a field (in this case, the integers instead of the rational numbers):

```
sage: A.base_ring()
Integer Ring
sage: A.echelon_form()
[ 3  0 -3]
[ 0  1  2]
[ 0  0  0]

sage: B=random_matrix(QQ,3,num_bound=10); B
[ -4  -3   6]
[  5  -5 9/2]
[3/2  -4  -7]
sage: B.rref()
[1 0 0]
[0 1 0]
[0 0 1]
```

In this case, since `B` is a matrix over a field (the rational numbers), `rref()` and `echelon_form()` are exactly the same:

```
sage: B.echelon_form()
[1 0 0]
[0 1 0]
[0 0 1]
sage: B.echelon_form() is B.rref()
True
```

Since `echelon_form()` is not implemented for every ring, sometimes behavior varies, as here:

```
sage: R.<x>=ZZ[]
sage: C = matrix(3,[2,x,x^2,x+1,3-x,-1,3,2,1])
sage: C.rref()
[1 0 0]
[0 1 0]
[0 0 1]
sage: C.base_ring()
Univariate Polynomial Ring in x over Integer Ring
sage: C.echelon_form()
Traceback (most recent call last):
...
NotImplementedError: Ideal Ideal (2, x + 1) of Univariate Polynomial Ring in␣
↪x over Integer Ring not principal
Echelon form not implemented over 'Univariate Polynomial Ring in x over␣
↪Integer Ring'.
sage: C = matrix(3,[2,x,x^2,x+1,3-x,-1,3,2,1/2])
sage: C.echelon_form()
[                               2                                x                          ␣
↪                    x^2]
[                               0                                1                          ␣
↪15*x^2 - 3/2*x - 31/2]
[                               0                                0 5/2*x^3 -␣
↪15/4*x^2 - 9/4*x + 7/2]
sage: C.rref()
```

```
[1 0 0]
[0 1 0]
[0 0 1]
sage: C = matrix(3,[2,x,x^2,x+1,3-x,-1/x,3,2,1/2])
sage: C.echelon_form()
[1 0 0]
[0 1 0]
[0 0 1]
```

set_block (*row*, *col*, *block*)

Sets the sub-matrix of self, with upper left corner given by row, col to block.

EXAMPLES:

```
sage: A = matrix(QQ, 3, 3, range(9))/2
sage: B = matrix(ZZ, 2, 1, [100,200])
sage: A.set_block(0, 1, B)
sage: A
[  0 100   1]
[3/2 200 5/2]
[  3 7/2   4]
```

We test that an exception is raised when the block is out of bounds:

```
sage: matrix([1]).set_block(0,1,matrix([1]))
Traceback (most recent call last):
...
IndexError: matrix window index out of range
```

smith_form ( )

If self is a matrix over a principal ideal domain R, return matrices D, U, V over R such that D = U * self * V, U and V have unit determinant, and D is diagonal with diagonal entries the ordered elementary divisors of self, ordered so that $D_i \mid D_{i+1}$. Note that U and V are not uniquely defined in general, and D is defined only up to units.

INPUT:

- `self` - a matrix over an integral domain. If the base ring is not a PID, the routine might work, or else it will fail having found an example of a non-principal ideal. Note that we do not call any methods to check whether or not the base ring is a PID, since this might be quite expensive (e.g. for rings of integers of number fields of large degree).

ALGORITHM: Lifted wholesale from http://en.wikipedia.org/wiki/Smith_normal_form

See also:

elementary_divisors()

AUTHORS:

- David Loeffler (2008-12-05)

EXAMPLES:

An example over the ring of integers of a number field (of class number 1):

```
sage: OE.<w> = EquationOrder(x^2 - x + 2)
sage: m = Matrix([ [1, w],[w,7]])
sage: d, u, v = m.smith_form()
sage: (d, u, v)
(
```

```
[      1       0]   [ 1   0]   [ 1 -w]
[      0 -w + 9],  [-w   1],  [ 0   1]
)
sage: u * m * v == d
True
sage: u.base_ring() == v.base_ring() == d.base_ring() == OE
True
sage: u.det().is_unit() and v.det().is_unit()
True
```

An example over the polynomial ring QQ[x]:

```
sage: R.<x> = QQ[]; m=x*matrix(R,2,2,1) - matrix(R, 2,2,[3,-4,1,-1]); m.smith_
↪form()
(
[            1              0]  [     0     -1]  [     1 x + 1]
[            0 x^2 - 2*x + 1],  [     1 x - 3],  [     0      1]
)
```

An example over a field:

```
sage: m = matrix( GF(17), 3, 3, [11,5,1,3,6,8,1,16,0]); d,u,v = m.smith_form()
sage: d
[1 0 0]
[0 1 0]
[0 0 0]
sage: u*m*v == d
True
```

Some examples over non-PID's work anyway:

```
sage: R.<s> = EquationOrder(x^2 + 5) # class number 2
sage: A = matrix(R, 2, 2, [s-1,-s,-s,2*s+1])
sage: D, U, V = A.smith_form()
sage: D, U, V
(
[      1       0]   [     4 s + 4]   [        1 -5*s + 6]
[      0 -s - 6],  [     s s - 1],  [        0        1]
)
sage: D == U*A*V
True
```

Others don't, but they fail quite constructively:

```
sage: matrix(R,2,2,[s-1,-s-2,-2*s,-s-2]).smith_form()
Traceback (most recent call last):
...
ArithmeticError: Ideal Fractional ideal (2, s + 1) not principal
```

Empty matrices are handled safely:

```
sage: m = MatrixSpace(OE, 2,0)(0); d,u,v=m.smith_form(); u*m*v == d
True
sage: m = MatrixSpace(OE, 0,2)(0); d,u,v=m.smith_form(); u*m*v == d
True
sage: m = MatrixSpace(OE, 0,0)(0); d,u,v=m.smith_form(); u*m*v == d
True
```

Some pathological cases that crashed earlier versions:

```
sage: m = Matrix(OE, [[2*w,2*w-1,-w+1],[2*w+2,-2*w-1,w-1],[-2*w-1,-2*w-2,2*w-
↪1]]); d, u, v = m.smith_form(); u * m * v == d
True
sage: m = matrix(OE, 3, 3, [-5*w-1,-2*w-2,4*w-10,8*w,-w,w-1,-1,1,-8]); d,u,v␣
↪= m.smith_form(); u*m*v == d
True
```

solve_left (*B*, *check=True*)

If self is a matrix A, then this function returns a vector or matrix X such that $XA = B$. If B is a vector then X is a vector and if B is a matrix, then X is a matrix.

INPUT:

- `B` - a matrix
- `check` - bool (default: True) - if False and self is nonsquare, may not raise an error message even if there is no solution. This is faster but more dangerous.

EXAMPLES:

```
sage: A = matrix(QQ,4,2, [0, -1, 1, 0, -2, 2, 1, 0])
sage: B = matrix(QQ,2,2, [1, 0, 1, -1])
sage: X = A.solve_left(B)
sage: X*A == B
True
```

solve_right (*B*, *check=True*)

If self is a matrix A, then this function returns a vector or matrix X such that $AX = B$. If B is a vector then X is a vector and if B is a matrix, then X is a matrix.

Note: In Sage one can also write `A \backslash B` for `A.solve_right(B)`, i.e., Sage implements the "the MATLAB/Octave backslash operator".

INPUT:

- `B` - a matrix or vector
- `check` - bool (default: True) - if False and self is nonsquare, may not raise an error message even if there is no solution. This is faster but more dangerous.

OUTPUT: a matrix or vector

See also:

solve_left()

EXAMPLES:

```
sage: A = matrix(QQ, 3, [1,2,3,-1,2,5,2,3,1])
sage: b = vector(QQ,[1,2,3])
sage: x = A \ b; x
(-13/12, 23/12, -7/12)
sage: A * x
(1, 2, 3)
```

We solve with A nonsquare:

```
sage: A = matrix(QQ,2,4, [0, -1, 1, 0, -2, 2, 1, 0]); B = matrix(QQ,2,2, [1,␣
↪0, 1, -1])
sage: X = A.solve_right(B); X
[-3/2  1/2]
[  -1    0]
[   0    0]
[   0    0]
sage: A*X == B
True
```

Another nonsingular example:

```
sage: A = matrix(QQ,2,3, [1,2,3,2,4,6]); v = vector([-1/2,-1])
sage: x = A \ v; x
(-1/2, 0, 0)
sage: A*x == v
True
```

Same example but over **Z**:

```
sage: A = matrix(ZZ,2,3, [1,2,3,2,4,6]); v = vector([-1,-2])
sage: A \ v
(-1, 0, 0)
```

An example in which there is no solution:

```
sage: A = matrix(QQ,2,3, [1,2,3,2,4,6]); v = vector([1,1])
sage: A \ v
Traceback (most recent call last):
...
ValueError: matrix equation has no solutions
```

A ValueError is raised if the input is invalid:

```
sage: A = matrix(QQ,4,2, [0, -1, 1, 0, -2, 2, 1, 0])
sage: B = matrix(QQ,2,2, [1, 0, 1, -1])
sage: X = A.solve_right(B)
Traceback (most recent call last):
...
ValueError: number of rows of self must equal number of rows of B
```

We solve with A singular:

```
sage: A = matrix(QQ,2,3, [1,2,3,2,4,6]); B = matrix(QQ,2,2, [6, -6, 12, -12])
sage: X = A.solve_right(B); X
[ 6 -6]
[ 0  0]
[ 0  0]
sage: A*X == B
True
```

We illustrate left associativity, etc., of the backslash operator.

```
sage: A = matrix(QQ, 2, [1,2,3,4])
sage: A \ A
[1 0]
[0 1]
sage: A \ A \ A
```

```
[1 2]
[3 4]
sage: A.parent()(1) \ A
[1 2]
[3 4]
sage: A \ (A \ A)
[  -2    1]
[ 3/2 -1/2]
sage: X = A \ (A - 2); X
[ 5 -2]
[-3  2]
sage: A * X
[-1  2]
[ 3  2]
```

Solving over a polynomial ring:

```
sage: x = polygen(QQ, 'x')
sage: A = matrix(2, [x,2*x,-5*x^2+1,3])
sage: v = vector([3,4*x - 2])
sage: X = A \ v
sage: X
((-8*x^2 + 4*x + 9)/(10*x^3 + x), (19*x^2 - 2*x - 3)/(10*x^3 + x))
sage: A * X == v
True
```

Solving some systems over $\mathbf{Z}/n\mathbf{Z}$:

```
sage: A = Matrix(Zmod(6), 3, 2, [1,2,3,4,5,6])
sage: B = vector(Zmod(6), [1,1,1])
sage: A.solve_right(B)
(5, 1)
sage: B = vector(Zmod(6), [5,1,1])
sage: A.solve_right(B)
Traceback (most recent call last):
...
ValueError: matrix equation has no solutions
sage: A = Matrix(Zmod(128), 2, 3, [23,11,22,4,1,0])
sage: B = Matrix(Zmod(128), 2, 1, [1,0])
sage: A.solve_right(B)
[  1]
[124]
[  1]
sage: B = B.column(0)
sage: A.solve_right(B)
(1, 124, 1)
```

Solving a system over the p-adics:

```
sage: k = Qp(5,4)
sage: a = matrix(k, 3, [1,7,3,2,5,4,1,1,2]); a
[    1 + O(5^4) 2 + 5 + O(5^4)     3 + O(5^4)]
[    2 + O(5^4)     5 + O(5^5)     4 + O(5^4)]
[    1 + O(5^4)     1 + O(5^4)     2 + O(5^4)]
sage: v = vector(k, 3, [1,2,3])
sage: x = a \ v; x
(4 + 5 + 5^2 + 3*5^3 + O(5^4), 2 + 5 + 3*5^2 + 5^3 + O(5^4), 1 + 5 + O(5^4))
sage: a * x == v
```

```
True
```

Solving a system of linear equation symbolically using symbolic matrices:

```
sage: var('a,b,c,d,x,y')
(a, b, c, d, x, y)
sage: A=matrix(SR,2,[a,b,c,d]); A
[a b]
[c d]
sage: result=vector(SR,[3,5]); result
(3, 5)
sage: soln=A.solve_right(result)
sage: soln
(-b*(3*c/a - 5)/(a*(b*c/a - d)) + 3/a, (3*c/a - 5)/(b*c/a - d))
sage: (a*x+b*y).subs(x=soln[0],y=soln[1]).simplify_full()
3
sage: (c*x+d*y).subs(x=soln[0],y=soln[1]).simplify_full()
5
sage: (A*soln).apply_map(lambda x: x.simplify_full())
(3, 5)
```

subdivide (*row_lines=None*, *col_lines=None*)

Divides self into logical submatrices which can then be queried and extracted. If a subdivision already exists, this method forgets the previous subdivision and flushes the cache.

INPUT:

- `row_lines` - None, an integer, or a list of integers (lines at which self must be split).
- `col_lines` - None, an integer, or a list of integers (columns at which self must be split).

OUTPUT: changes self

Note: One may also pass a tuple into the first argument which will be interpreted as (row_lines, col_lines)

EXAMPLES:

```
sage: M = matrix(5, 5, prime_range(100))
sage: M.subdivide(2,3); M
[ 2  3  5| 7 11]
[13 17 19|23 29]
[--------+-----]
[31 37 41|43 47]
[53 59 61|67 71]
[73 79 83|89 97]
sage: M.subdivision(0,0)
[ 2  3  5]
[13 17 19]
sage: M.subdivision(1,0)
[31 37 41]
[53 59 61]
[73 79 83]
sage: M.subdivision_entry(1,0,0,0)
31
sage: M.subdivisions()
([2], [3])
sage: M.subdivide(None, [1,3]); M
[ 2| 3  5| 7 11]
```

```
[13|17 19|23 29]
[31|37 41|43 47]
[53|59 61|67 71]
[73|79 83|89 97]
```

Degenerate cases work too:

```
sage: M.subdivide([2,5], [0,1,3]); M
[| 2| 3  5| 7 11]
[|13|17 19|23 29]
[+--+-----+-----]
[|31|37 41|43 47]
[|53|59 61|67 71]
[|73|79 83|89 97]
[+--+-----+-----]
sage: M.subdivision(0,0)
[]
sage: M.subdivision(0,1)
[ 2]
[13]
sage: M.subdivide([2,2,3], [0,0,1,1]); M
[|| 2|| 3  5  7 11]
[||13||17 19 23 29]
[++--++-----------]
[++--++-----------]
[||31||37 41 43 47]
[++--++-----------]
[||53||59 61 67 71]
[||73||79 83 89 97]
sage: M.subdivision(0,0)
[]
sage: M.subdivision(2,4)
[37 41 43 47]
```

Indices do not need to be in the right order (trac ticket #14064):

```
sage: M.subdivide([4, 2], [3, 1]); M
[ 2| 3  5| 7 11]
[13|17 19|23 29]
[--+-----+-----]
[31|37 41|43 47]
[53|59 61|67 71]
[--+-----+-----]
[73|79 83|89 97]
```

AUTHORS:

- Robert Bradshaw (2007-06-14)

subdivision (i, j)

Returns an immutable copy of the (i,j)th submatrix of self, according to a previously set subdivision.

Before a subdivision is set, the only valid arguments are (0,0) which returns self.

EXAMPLES:

```
sage: M = matrix(3, 4, range(12))
sage: M.subdivide(1,2); M
[ 0  1| 2  3]
```

```
[-----+-----]
[ 4  5| 6  7]
[ 8  9|10 11]
sage: M.subdivision(0,0)
[0 1]
sage: M.subdivision(0,1)
[2 3]
sage: M.subdivision(1,0)
[4 5]
[8 9]
```

It handles size-zero subdivisions as well.

```
sage: M = matrix(3, 4, range(12))
sage: M.subdivide([0],[0,2,2,4]); M
[+-----++-----+]
[| 0  1|| 2  3|]
[| 4  5|| 6  7|]
[| 8  9||10 11|]
sage: M.subdivision(0,0)
[]
sage: M.subdivision(1,1)
[0 1]
[4 5]
[8 9]
sage: M.subdivision(1,2)
[]
sage: M.subdivision(1,0)
[]
sage: M.subdivision(0,1)
[]
```

subdivision_entry (*i*, *j*, *x*, *y*)

Returns the x,y entry of the i,j submatrix of self.

EXAMPLES:

```
sage: M = matrix(5, 5, range(25))
sage: M.subdivide(3,3); M
[ 0  1  2| 3  4]
[ 5  6  7| 8  9]
[10 11 12|13 14]
[--------+-----]
[15 16 17|18 19]
[20 21 22|23 24]
sage: M.subdivision_entry(0,0,1,2)
7
sage: M.subdivision(0,0)[1,2]
7
sage: M.subdivision_entry(0,1,0,0)
3
sage: M.subdivision_entry(1,0,0,0)
15
sage: M.subdivision_entry(1,1,1,1)
24
```

Even though this entry exists in the matrix, the index is invalid for the submatrix.

```
sage: M.subdivision_entry(0,0,4,0)
Traceback (most recent call last):
...
IndexError: Submatrix 0,0 has no entry 4,0
```

subdivisions ( )

Returns the current subdivision of self.

EXAMPLES:

```
sage: M = matrix(5, 5, range(25))
sage: M.subdivisions()
([], [])
sage: M.subdivide(2,3)
sage: M.subdivisions()
([2], [3])
sage: N = M.parent()(1)
sage: N.subdivide(M.subdivisions()); N
[1 0 0|0 0]
[0 1 0|0 0]
[-----+---]
[0 0 1|0 0]
[0 0 0|1 0]
[0 0 0|0 1]
```

subs (**args*, ***kwds*)

Substitute values to the variables in that matrix.

All the arguments are transmitted unchanged to the method `subs` of the coefficients.

EXAMPLES:

```
sage: var('a,b,d,e')
(a, b, d, e)
sage: m = matrix([[a,b], [d,e]])
sage: m.substitute(a=1)
[1 b]
[d e]
sage: m.subs(a=b, b=d)
[b d]
[d e]
sage: m.subs({a: 3, b:2, d:1, e:-1})
[ 3  2]
[ 1 -1]
```

The parent of the newly created matrix might be different from the initial one. It depends on what the method `.subs` does on coefficients (see trac ticket #19045):

```
sage: x = polygen(ZZ)
sage: m = matrix([[x]])
sage: m2 = m.subs(x=2)
sage: m2.parent()
Full MatrixSpace of 1 by 1 dense matrices over Integer Ring
sage: m1 = m.subs(x=RDF(1))
sage: m1.parent()
Full MatrixSpace of 1 by 1 dense matrices over Real Double Field
```

However, sparse matrices remain sparse:

```
sage: m = matrix({(3,2): -x, (59,38): x^2+2}, nrows=1000, ncols=1000)
sage: m1 = m.subs(x=1)
sage: m1.is_sparse()
True
```

symplectic_form ( )

Find a symplectic form for self if self is an anti-symmetric, alternating matrix defined over a field.

Returns a pair (F, C) such that the rows of C form a symplectic basis for self and F = C * self * C.transpose().

Raises a ValueError if not over a field, or self is not anti-symmetric, or self is not alternating.

Anti-symmetric means that $M = -M^t$. Alternating means that the diagonal of M is identically zero.

A symplectic basis is a basis of the form $e_1, \ldots, e_j, f_1, \ldots f_j, z_1, \ldots, z_k$ such that

- $z_i M v^t$ = 0 for all vectors v
- $e_i M e_j{}^t = 0$ for all i, j
- $f_i M f_j{}^t = 0$ for all i, j
- $e_i M f_i{}^t = 1$ for all i
- $e_i M f_j{}^t = 0$ for all i not equal j.

See the example for a pictorial description of such a basis.

EXAMPLES:

```
sage: E = matrix(QQ, 8, 8, [0, -1/2, -2, 1/2, 2, 0, -2, 1, 1/2, 0, -1, -3, 0,␣
↪2, 5/2, -3, 2, 1, 0, 3/2, -1, 0, -1, -2, -1/2, 3, -3/2, 0, 1, 3/2, -1/2, -1/
↪2, -2, 0, 1, -1, 0, 0, 1, -1, 0, -2, 0, -3/2, 0, 0, 1/2, -2, 2, -5/2, 1, 1/
↪2, -1, -1/2, 0, -1, -1, 3, 2, 1/2, 1, 2, 1, 0]); E
[   0 -1/2   -2  1/2    2    0   -2    1]
[ 1/2    0   -1   -3    0    2  5/2   -3]
[   2    1    0  3/2   -1    0   -1   -2]
[-1/2    3 -3/2    0    1  3/2 -1/2 -1/2]
[  -2    0    1   -1    0    0    1   -1]
[   0   -2    0 -3/2    0    0  1/2   -2]
[   2 -5/2    1  1/2   -1 -1/2    0   -1]
[  -1    3    2  1/2    1    2    1    0]
sage: F, C = E.symplectic_form(); F
[ 0  0  0  0  1  0  0  0]
[ 0  0  0  0  0  1  0  0]
[ 0  0  0  0  0  0  1  0]
[ 0  0  0  0  0  0  0  1]
[-1  0  0  0  0  0  0  0]
[ 0 -1  0  0  0  0  0  0]
[ 0  0 -1  0  0  0  0  0]
[ 0  0  0 -1  0  0  0  0]
sage: F == C * E * C.transpose()
True
```

tensor_product (*A*, *subdivide=True*)

Returns the tensor product of two matrices.

INPUT:

- `A` - a matrix
- `subdivide` - default: True - whether or not to return natural subdivisions with the matrix

OUTPUT:

Replace each element of `self` by a copy of `A`, but first create a scalar multiple of `A` by the element it replaces. So if `self` is an $m \times n$ matrix and `A` is a $p \times q$ matrix, then the tensor product is an $mp \times nq$ matrix. By default, the matrix will be subdivided into submatrices of size $p \times q$.

EXAMPLES:

```
sage: M1=Matrix(QQ,[[-1,0],[-1/2,-1]])
sage: M2=Matrix(ZZ,[[1,-1,2],[-2,4,8]])
sage: M1.tensor_product(M2)
[  -1    1   -2|   0    0    0]
[   2   -4   -8|   0    0    0]
[--------------+--------------]
[-1/2  1/2   -1|  -1    1   -2]
[   1   -2   -4|   2   -4   -8]
sage: M2.tensor_product(M1)
[  -1    0|   1    0|  -2    0]
[-1/2   -1| 1/2    1|  -1   -2]
[---------+---------+---------]
[   2    0|  -4    0|  -8    0]
[   1    2|  -2   -4|  -4   -8]
```

Subdivisions can be optionally suppressed.

```
sage: M1.tensor_product(M2, subdivide=False)
[  -1    1   -2    0    0    0]
[   2   -4   -8    0    0    0]
[-1/2  1/2   -1   -1    1   -2]
[   1   -2   -4    2   -4   -8]
```

Different base rings are handled sensibly.

```
sage: A = matrix(ZZ, 2, 3, range(6))
sage: B = matrix(FiniteField(23), 3, 4, range(12))
sage: C = matrix(FiniteField(29), 4, 5, range(20))
sage: D = A.tensor_product(B)
sage: D.parent()
Full MatrixSpace of 6 by 12 dense matrices over Finite Field of size 23
sage: E = C.tensor_product(B)
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for *: 'Finite Field of size 29' and
↪'Full MatrixSpace of 3 by 4 dense matrices over Finite Field of size 23'
```

The input is checked to be sure it is a matrix.

```
sage: A = matrix(QQ, 2, 2, range(4))
sage: A.tensor_product('junk')
Traceback (most recent call last):
...
TypeError: tensor product requires a second matrix, not junk
```

trace ()

Return the trace of self, which is the sum of the diagonal entries of self.

INPUT:

- `self` - a square matrix

OUTPUT: element of the base ring of self

EXAMPLES:

```
sage: a = matrix(3,3,range(9)); a
[0 1 2]
[3 4 5]
[6 7 8]
sage: a.trace()
12
sage: a = matrix({(1,1):10, (2,1):-3, (2,2):4/3}); a
[   0    0    0]
[   0   10    0]
[   0   -3  4/3]
sage: a.trace()
34/3
```

trace_of_product (*other*)

Returns the trace of self * other without computing the entire product.

EXAMPLES:

```
sage: M = random_matrix(ZZ, 10, 20)
sage: N = random_matrix(ZZ, 20, 10)
sage: M.trace_of_product(N)
-1629
sage: (M*N).trace()
-1629
```

visualize_structure (*maxsize=512*)

Visualize the non-zero entries

White pixels are put at positions with zero entries. If 'maxsize' is given, then the maximal dimension in either x or y direction is set to 'maxsize' depending on which is bigger. If the image is scaled, the darkness of the pixel reflects how many of the represented entries are nonzero. So if e.g. one image pixel actually represents a 2x2 submatrix, the dot is darker the more of the four values are nonzero.

INPUT:

- `maxsize` - integer (default: `512`). Maximal dimension in either x or y direction of the resulting image. If `None` or a maxsize larger than `max(self.nrows(),self.ncols())` is given the image will have the same pixelsize as the matrix dimensions.

OUTPUT:

Bitmap image as an instance of `Image` .

EXAMPLES:

```
sage: M = random_matrix(CC, 5, 7)
sage: for i in range(5):  M[i,i] = 0
sage: M[4, 0] = M[0, 6] = M[4, 6] = 0
sage: img = M.visualize_structure();  img
7x5px 24-bit RGB image
```

You can use `save()` to save the resulting image:

```
sage: filename = tmp_filename(ext='.png')
sage: img.save(filename)
sage: open(filename).read().startswith('\x89PNG')
True
```

wiedemann (*i*, *t=0*)

Application of Wiedemann's algorithm to the i-th standard basis vector.

INPUT:

- `i` - an integer
- `t` - an integer (default: 0) if t is nonzero, use only the first t linear recurrence relations.

IMPLEMENTATION: This is a toy implementation.

EXAMPLES:

```
sage: t = matrix(QQ, 3, 3, range(9)); t
[0 1 2]
[3 4 5]
[6 7 8]
sage: t.wiedemann(0)
x^2 - 12*x - 18
sage: t.charpoly()
x^3 - 12*x^2 - 18*x
```

zigzag_form (*subdivide=True*, *transformation=False*)

Find a matrix in ZigZag form that is similar to `self` .

INPUT:

- `self` - a square matrix with entries from an exact field.
- `transformation` - default: False - if `True` return a change-of-basis matrix relating the matrix and its ZigZag form.
- `subdivide` - default: `True` - if `True` the ZigZag form matrix is subdivided according to the companion matrices described in the output section below.

OUTPUT:

A matrix in ZigZag form has blocks on the main diagonal that are companion matrices. The first companion matrix has ones just below the main diagonal. The last column has the negatives of coefficients of a monic polynomial, but not the leading one. Low degree monomials have their coefficients in the earlier rows. The second companion matrix is like the first only transposed. The third is like the first. The fourth is like the second. And so on.

These blocks on the main diagonal define blocks just off the diagonal. To the right of the first companion matrix, and above the second companion matrix is a block that is totally zero, except the entry of the first row and first column may be a one. Below the second block and to the left of the third block is a block that is totally zero, except the entry of the first row and first column may be one. This alternating pattern continues. It may now be apparent how this form gets its name. Any other entry of the matrix is zero. So this form is reminiscent of rational canonical form and is a good precursor to that form.

If `transformation` is `True` , then the output is a pair of matrices. The first is the form `Z` and the second is an invertible matrix `U` such that `U.inverse()*self*U` equals `Z` . In other words, the representation of `self` with respect to the columns of `U` will be `Z` .

If subdivide is `True` then the matrix returned as the form is partitioned according to the companion matrices and these may be manipulated by several different matrix methods.

For output that may be more useful as input to other routines, see the helper method `_zigzag_form()` .

Note: An effort has been made to optimize computation of the form, but no such work has been done for the computation of the transformation matrix, so for fastest results do not request the transformation matrix.

ALGORITHM:

ZigZag form, and its computation, are due to Arne Storjohann and are described in [Sto2000] and [Sto1998], where the former is more representative of the code here.

EXAMPLES:

Two examples that illustrate ZigZag form well. Notice that this is *not* a canonical form. The two matrices below are similar, since they have equal Jordan canonical forms, yet their ZigZag forms are quite different. In other words, while the computation of the form is deterministic, the final result, when viewed as a property of a linear transformation, is dependent on the basis used for the matrix representation.

```
sage: A = matrix(QQ, [[-68,   69, -27, -11, -65,   9, -181, -32],
....:                 [-52,   52, -27,  -8, -52, -16, -133, -14],
....:                 [ 92,  -97,  47,  14,  90,  32,  241,  18],
....:                 [139, -144,  60,  18, 148, -10,  362,  77],
....:                 [ 40,  -41,  12,   6,  45, -24,  105,  42],
....:                 [-46,   48, -20,  -7, -47,   0, -122, -22],
....:                 [-26,   27, -13,  -4, -29,  -6,  -66, -14],
....:                 [-33,   34, -13,  -5, -35,   7,  -87, -23]])
sage: Z, U = A.zigzag_form(transformation=True)
sage: Z
[  0   0   0  40|  1   0|  0   0]
[  1   0   0  52|  0   0|  0   0]
[  0   1   0  18|  0   0|  0   0]
[  0   0   1  -1|  0   0|  0   0]
[---------------+-------+-------]
[  0   0   0   0|  0   1|  0   0]
[  0   0   0   0|-25  10|  0   0]
[---------------+-------+-------]
[  0   0   0   0|  1   0|  0  -4]
[  0   0   0   0|  0   0|  1  -4]
sage: U.inverse()*A*U == Z
True

sage: B = matrix(QQ, [[ 16,  69, -13,   2, -52,  143,   90,  -3],
....:                 [ 26,  54,   6,  -5, -28,   73,   73, -48],
....:                 [-16, -79,  12, -10,  64, -142, -115,  41],
....:                 [ 27,  -7,  21, -33,  39,  -20,  -42,  43],
....:                 [  8, -75,  34, -32,  86, -156, -130,  42],
....:                 [  2, -17,   7,  -8,  20,  -33,  -31,  16],
....:                 [-24, -80,   7,  -3,  56, -136, -112,  42],
....:                 [ -6, -19,   0,  -1,  13,  -28,  -27,  15]])
sage: Z, U = B.zigzag_form(transformation=True)
sage: Z
[   0    0    0    0    0 1000|   0|   0]
[   1    0    0    0    0  900|   0|   0]
[   0    1    0    0    0  -30|   0|   0]
[   0    0    1    0    0 -153|   0|   0]
[   0    0    0    1    0    3|   0|   0]
[   0    0    0    0    1    9|   0|   0]
[------------------------------+----+----]
[   0    0    0    0    0    0|  -2|   0]
[------------------------------+----+----]
[   0    0    0    0    0    0|   1|  -2]
sage: U.inverse()*B*U == Z
True
```

```
sage: A.jordan_form() == B.jordan_form()
True
```

Two more examples, illustrating the two extremes of the zig-zag nature of this form. The first has a one in each of the off-diagonal blocks, the second has all zeros in each off-diagonal block. Notice again that the two matrices are similar, since their Jordan canonical forms are equal.

```
sage: C = matrix(QQ, [[2,  31, -10,  -9, -125,  13,  62, -12],
....:                 [0,  48, -16, -16, -188,  20,  92, -16],
....:                 [0,   9,  -1,   2,  -33,   5,  18,   0],
....:                 [0,  15,  -5,   0,  -59,   7,  30,  -4],
....:                 [0, -21,   7,   2,   84, -10, -42,   5],
....:                 [0, -42,  14,   8,  167, -17, -84,  13],
....:                 [0, -50,  17,  10,  199, -23, -98,  14],
....:                 [0,  15,  -5,  -2,  -59,   7,  30, -2]])
sage: Z, U = C.zigzag_form(transformation=True)
sage: Z
[2|1|0|0|0|0|0|0]
[-+-+-+-+-+-+-+-]
[0|2|0|0|0|0|0|0]
[-+-+-+-+-+-+-+-]
[0|1|2|1|0|0|0|0]
[-+-+-+-+-+-+-+-]
[0|0|0|2|0|0|0|0]
[-+-+-+-+-+-+-+-]
[0|0|0|1|2|1|0|0]
[-+-+-+-+-+-+-+-]
[0|0|0|0|0|2|0|0]
[-+-+-+-+-+-+-+-]
[0|0|0|0|0|1|2|1]
[-+-+-+-+-+-+-+-]
[0|0|0|0|0|0|0|2]
sage: U.inverse()*C*U == Z
True

sage: D = matrix(QQ, [[ -4,   3,    7,   2,  -4,   5,    7,   -3],
....:                 [ -6,   5,    7,   2,  -4,   5,    7,   -3],
....:                 [ 21, -12,   89,  25,   8,  27,   98,  -95],
....:                 [ -9,   5,  -44, -11,  -3, -13,  -48,   47],
....:                 [ 23, -13,   74,  21,  12,  22,   85,  -84],
....:                 [ 31, -18,  135,  38,  12,  47,  155, -147],
....:                 [-33,  19, -138, -39, -13, -45, -156,  151],
....:                 [ -7,   4,  -29,  -8,  -3, -10,  -34,   34]])
sage: Z, U = D.zigzag_form(transformation=True)
sage: Z
[ 0 -4| 0  0| 0  0| 0  0]
[ 1  4| 0  0| 0  0| 0  0]
[-----+-----+-----+-----]
[ 0  0| 0  1| 0  0| 0  0]
[ 0  0|-4  4| 0  0| 0  0]
[-----+-----+-----+-----]
[ 0  0| 0  0| 0 -4| 0  0]
[ 0  0| 0  0| 1  4| 0  0]
[-----+-----+-----+-----]
[ 0  0| 0  0| 0  0| 0  1]
[ 0  0| 0  0| 0  0|-4  4]
sage: U.inverse()*D*U == Z
```

```
True

sage: C.jordan_form() == D.jordan_form()
True
```

ZigZag form is achieved entirely with the operations of the field, so while the eigenvalues may lie outside the field, this does not impede the computation of the form.

```
sage: F.<a> = GF(5^4)
sage: A = matrix(F, [[      a,       0,   0, a + 3],
....:                [      0,a^2 + 1,   0,      0],
....:                [      0,       0,a^3,      0],
....:                [a^2 +4 ,       0,   0,a + 2]])
sage: A.zigzag_form()
[                    0 a^3 + 2*a^2 + 2*a + 2|                    0|                    ␣
↪          0]
[                    1                2*a + 2|                    0|                    ␣
↪          0]
[-------------------------------------------+---------------------+-----------
↪----------]
[                    0                     0|                  a^3|                    ␣
↪          0]
[-------------------------------------------+---------------------+-----------
↪----------]
[                    0                     0|                    0|                    ␣
↪    a^2 + 1]
sage: A.eigenvalues()
Traceback (most recent call last):
...
NotImplementedError: algebraic closures of finite fields are only implemented␣
↪for prime fields
```

Subdivisions are optional.

```
sage: F.<a> = GF(5^4)
sage: A = matrix(F, [[      a,       0,   0, a + 3],
....:                [      0,a^2 + 1,   0,      0],
....:                [      0,       0,a^3,      0],
....:                [a^2 +4 ,       0,   0,a + 2]])
sage: A.zigzag_form(subdivide=False)
[                    0 a^3 + 2*a^2 + 2*a + 2                    0                    ␣
↪          0]
[                    1                2*a + 2                    0                    ␣
↪          0]
[                    0                      0                  a^3                    ␣
↪          0]
[                    0                      0                    0                    ␣
↪    a^2 + 1]
```

AUTHOR:

- Rob Beezer (2011-06-09)

`sage.matrix.matrix2.`**`cmp_pivots`**`(x, y)`

Compare two sequences of pivot columns.

- If x is shorter than y, return -1, i.e., x < y, “not as good”.
- If x is longer than y, x > y, “better”.

- If the length is the same then x is better, i.e., x > y if the entries of x are correspondingly >= those of y with one being greater.

`sage.matrix.matrix2.` **decomp_seq** (*v*)

This function is used internally be the decomposition matrix method. It takes a list of tuples and produces a sequence that is correctly sorted and prints with carriage returns.

EXAMPLES:

```
sage: from sage.matrix.matrix2 import decomp_seq
sage: V = [(QQ^3, 2), (QQ^2, 1)]
sage: decomp_seq(V)
[
(Vector space of dimension 2 over Rational Field, 1),
(Vector space of dimension 3 over Rational Field, 2)
]
```

CHAPTER

NINE

GENERIC ASYMPTOTICALLY FAST STRASSEN ALGORITHMS

Sage implements asymptotically fast echelon form and matrix multiplication algorithms.

class sage.matrix.strassen.**int_range**(*indices=None*, *range=None*)

Represent a list of integers as a list of integer intervals.

Note: Repetitions are not considered.

Useful class for dealing with pivots in the strassen echelon, could have much more general application

INPUT:

It can be one of the following:

- indices - integer, start of the unique interval
- range - integer, length of the unique interval

OR

- indices - list of integers, the integers to wrap into intervals

OR

- indices - None (default), shortcut for an empty list

OUTPUT:

An instance of int_range, i.e. a list of pairs (start,length).

EXAMPLES:

From a pair of integers:

```
sage: from sage.matrix.strassen import int_range
sage: int_range(2, 4)
[(2, 4)]
```

Default:

```
sage: int_range()
[]
```

From a list of integers:

```
sage: int_range([1,2,3,4])
[(1, 4)]
sage: int_range([1,2,3,4,6,7,8])
[(1, 4), (6, 3)]
```

```
sage: int_range([1,2,3,4,100,101,102])
[(1, 4), (100, 3)]
sage: int_range([1,1000,2,101,3,4,100,102])
[(1, 4), (100, 3), (1000, 1)]
```

Repetitions are not considered:

```
sage: int_range([1,2,3])
[(1, 3)]
sage: int_range([1,1,1,1,2,2,2,3])
[(1, 3)]
```

AUTHORS:

- Robert Bradshaw

intervals ( )

Return the list of intervals.

OUTPUT:

A list of pairs of integers.

EXAMPLES:

```
sage: from sage.matrix.strassen import int_range
sage: I = int_range([4,5,6,20,21,22,23])
sage: I.intervals()
[(4, 3), (20, 4)]
sage: type(I.intervals())
<... 'list'>
```

to_list ( )

Return the (sorted) list of integers represented by this object.

OUTPUT:

A list of integers.

EXAMPLES:

```
sage: from sage.matrix.strassen import int_range
sage: I = int_range([6,20,21,4,5,22,23])
sage: I.to_list()
[4, 5, 6, 20, 21, 22, 23]
```

```
sage: I = int_range(34, 9)
sage: I.to_list()
[34, 35, 36, 37, 38, 39, 40, 41, 42]
```

Repetitions are not considered:

```
sage: I = int_range([1,1,1,1,2,2,2,3])
sage: I.to_list()
[1, 2, 3]
```

sage.matrix.strassen.**strassen_echelon** (*A*, *cutoff*)

Compute echelon form, in place. Internal function, call with M.echelonize(algorithm="strassen") Based on work of Robert Bradshaw and David Harvey at MSRI workshop in 2006.

INPUT:

- A - matrix window
- cutoff - size at which algorithm reverts to naive Gaussian elimination and multiplication must be at least 1.

OUTPUT: The list of pivot columns

EXAMPLES:

```
sage: A = matrix(QQ, 7, [5, 0, 0, 0, 0, 0, -1, 0, 0, 1, 0, 0, 0, 0, 0, -1, 3, 1,
↪0, -1, 0, 0, -1, 0, 1, 2, -1, 1, 0, -1, 0, 1, 3, -1, 1, 0, 0, -2, 0, 2, 0, 1,
↪0, 0, -1, 0, 1, 0, 1])
sage: B = A.__copy__(); B._echelon_strassen(1); B
[ 1  0  0  0  0  0  0]
[ 0  1  0 -1  0  1  0]
[ 0  0  1  0  0  0  0]
[ 0  0  0  0  1  0  0]
[ 0  0  0  0  0  0  1]
[ 0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0]
sage: C = A.__copy__(); C._echelon_strassen(2); C == B
True
sage: C = A.__copy__(); C._echelon_strassen(4); C == B
True
```

```
sage: n = 32; A = matrix(Integers(389),n,range(n^2))
sage: B = A.__copy__(); B._echelon_in_place_classical()
sage: C = A.__copy__(); C._echelon_strassen(2)
sage: B == C
True
```

```
sage: A = matrix(Integers(7), 4, 4, [1,0,5,0,2,0,3,6,5,1,2,6,4,6,1,1])
sage: B = A.__copy__(); B._echelon_in_place_classical()
sage: C = A.__copy__(); C._echelon_strassen(2)    #indirect doctest
sage: B == C
True
```

AUTHORS:

- Robert Bradshaw

sage.matrix.strassen. **strassen_window_multiply** (*C*, *A*, *B*, *cutoff*)

Multiplies the submatrices specified by A and B, places result in C. Assumes that A and B have compatible dimensions to be multiplied, and that C is the correct size to receive the product, and that they are all defined over the same ring.

Uses strassen multiplication at high levels and then uses MatrixWindow methods at low levels. EXAMPLES: The following matrix dimensions are chosen especially to exercise the eight possible parity combinations that could occur while subdividing the matrix in the strassen recursion. The base case in both cases will be a (4x5) matrix times a (5x6) matrix.

```
sage: A = MatrixSpace(Integers(2^65), 64, 83).random_element()
sage: B = MatrixSpace(Integers(2^65), 83, 101).random_element()
sage: A._multiply_classical(B) == A._multiply_strassen(B, 3) #indirect doctest
True
```

AUTHORS:

- David Harvey

- Simon King (2011-07): Improve memory efficiency; trac ticket #11610

`sage.matrix.strassen.`**`test`**`(` *n*, *m*, *R*, *c=2* `)`

INPUT:

- `n` - integer
- `m` - integer
- `R` - ring
- `c` - integer (optional, default:2)

EXAMPLES:

```
sage: from sage.matrix.strassen import test
sage: for n in range(5):
....:     print("{} {}".format(n, test(2*n,n,Frac(QQ['x']),2)))
0 True
1 True
2 True
3 True
4 True
```

CHAPTER

TEN

MINIMAL POLYNOMIALS OF LINEAR RECURRENCE SEQUENCES

AUTHORS:

- William Stein

`sage.matrix.berlekamp_massey.`**`berlekamp_massey`**`(`*a*`)`

Use the Berlekamp-Massey algorithm to find the minimal polynomial of a linearly recurrence sequence a.

The minimal polynomial of a linear recurrence $\{a_r\}$ is by definition the unique monic polynomial g, such that if $\{a_r\}$ satisfies a linear recurrence $a_{j+k} + b_{j-1}a_{j-1+k} + \cdots + b_0 a_k = 0$ (for all $k \geq 0$), then g divides the polynomial $x^j + \sum_{i=0}^{j-1} b_i x^i$.

INPUT:

- `a` - a list of even length of elements of a field (or domain)

OUTPUT:

- `Polynomial` - the minimal polynomial of the sequence (as a polynomial over the field in which the entries of a live)

EXAMPLES:

```
sage: berlekamp_massey([1,2,1,2,1,2])
x^2 - 1
sage: berlekamp_massey([GF(7)(1),19,1,19])
x^2 + 6
sage: berlekamp_massey([2,2,1,2,1,191,393,132])
x^4 - 36727/11711*x^3 + 34213/5019*x^2 + 7024942/35133*x - 335813/1673
sage: berlekamp_massey(prime_range(2,38))
x^6 - 14/9*x^5 - 7/9*x^4 + 157/54*x^3 - 25/27*x^2 - 73/18*x + 37/9
```

CHAPTER

ELEVEN

BASE CLASS FOR DENSE MATRICES

class sage.matrix.matrix_dense.**Matrix_dense**

Bases: *sage.matrix.matrix.Matrix*

antitranspose()

Returns the antitranspose of self, without changing self.

EXAMPLES:

```
sage: A = matrix(2,3,range(6)); A
[0 1 2]
[3 4 5]
sage: A.antitranspose()
[5 2]
[4 1]
[3 0]
```

```
sage: A.subdivide(1,2); A
[0 1|2]
[---+-]
[3 4|5]
sage: A.antitranspose()
[5|2]
[-+-]
[4|1]
[3|0]
```

transpose()

Returns the transpose of self, without changing self.

EXAMPLES: We create a matrix, compute its transpose, and note that the original matrix is not changed.

```
sage: M = MatrixSpace(QQ,  2)
sage: A = M([1,2,3,4])
sage: B = A.transpose()
sage: print(B)
[1 3]
[2 4]
sage: print(A)
[1 2]
[3 4]
```

`.T` is a convenient shortcut for the transpose:

```
sage: A.T
[1 3]
[2 4]
```

```
sage: A.subdivide(None, 1); A
[1|2]
[3|4]
sage: A.transpose()
[1 3]
[---]
[2 4]
```

CHAPTER

TWELVE

BASE CLASS FOR SPARSE MATRICES

class sage.matrix.matrix_sparse.**Matrix_sparse**

Bases: *sage.matrix.matrix.Matrix*

antitranspose ()

apply_map (*phi*, *R=None*, *sparse=True*)

Apply the given map phi (an arbitrary Python function or callable object) to this matrix. If R is not given, automatically determine the base ring of the resulting matrix.

INPUT: sparse – False to make the output a dense matrix; default True

- phi - arbitrary Python function or callable object
- R - (optional) ring

OUTPUT: a matrix over R

EXAMPLES:

```
sage: m = matrix(ZZ, 10000, {(1,2): 17}, sparse=True)
sage: k.<a> = GF(9)
sage: f = lambda x: k(x)
sage: n = m.apply_map(f)
sage: n.parent()
Full MatrixSpace of 10000 by 10000 sparse matrices over Finite Field in a of␣
↪size 3^2
sage: n[1,2]
2
```

An example where the codomain is explicitly specified.

```
sage: n = m.apply_map(lambda x:x%3, GF(3))
sage: n.parent()
Full MatrixSpace of 10000 by 10000 sparse matrices over Finite Field of size 3
sage: n[1,2]
2
```

If we didn't specify the codomain, the resulting matrix in the above case ends up over ZZ again:

```
sage: n = m.apply_map(lambda x:x%3)
sage: n.parent()
Full MatrixSpace of 10000 by 10000 sparse matrices over Integer Ring
sage: n[1,2]
2
```

If self is subdivided, the result will be as well:

```
sage: m = matrix(2, 2, [0, 0, 3, 0])
sage: m.subdivide(None, 1); m
[0|0]
[3|0]
sage: m.apply_map(lambda x: x*x)
[0|0]
[9|0]
```

If the map sends zero to a non-zero value, then it may be useful to get the result as a dense matrix.

```
sage: m = matrix(ZZ, 3, 3, [0] * 7 + [1,2], sparse=True); m
[0 0 0]
[0 0 0]
[0 1 2]
sage: parent(m)
Full MatrixSpace of 3 by 3 sparse matrices over Integer Ring
sage: n = m.apply_map(lambda x: x+polygen(QQ), sparse=False); n
[    x     x     x]
[    x     x     x]
[    x x + 1 x + 2]
sage: parent(n)
Full MatrixSpace of 3 by 3 dense matrices over Univariate Polynomial Ring in␣
↪x over Rational Field
```

apply_morphism (*phi*)

Apply the morphism phi to the coefficients of this sparse matrix.

The resulting matrix is over the codomain of phi.

INPUT:

- `phi` - a morphism, so phi is callable and phi.domain() and phi.codomain() are defined. The codomain must be a ring.

OUTPUT: a matrix over the codomain of phi

EXAMPLES:

```
sage: m = matrix(ZZ, 3, range(9), sparse=True)
sage: phi = ZZ.hom(GF(5))
sage: m.apply_morphism(phi)
[0 1 2]
[3 4 0]
[1 2 3]
sage: m.apply_morphism(phi).parent()
Full MatrixSpace of 3 by 3 sparse matrices over Finite Field of size 5
```

augment (*right*, *subdivide=False*)

Return the augmented matrix of the form:

```
[self | right].
```

EXAMPLES:

```
sage: M = MatrixSpace(QQ, 2, 2, sparse=True)
sage: A = M([1,2, 3,4])
sage: A
[1 2]
```

```
[3 4]
sage: N = MatrixSpace(QQ, 2, 1, sparse=True)
sage: B = N([9,8])
sage: B
[9]
[8]
sage: A.augment(B)
[1 2 9]
[3 4 8]
sage: B.augment(A)
[9 1 2]
[8 3 4]
```

A vector may be augmented to a matrix.

```
sage: A = matrix(QQ, 3, 4, range(12), sparse=True)
sage: v = vector(QQ, 3, range(3), sparse=True)
sage: A.augment(v)
[ 0  1  2  3  0]
[ 4  5  6  7  1]
[ 8  9 10 11  2]
```

The `subdivide` option will add a natural subdivision between `self` and `right` . For more details about how subdivisions are managed when augmenting, see *sage.matrix.matrix1.Matrix.augment()*.

```
sage: A = matrix(QQ, 3, 5, range(15), sparse=True)
sage: B = matrix(QQ, 3, 3, range(9), sparse=True)
sage: A.augment(B, subdivide=True)
[ 0  1  2  3  4| 0  1  2]
[ 5  6  7  8  9| 3  4  5]
[10 11 12 13 14| 6  7  8]
```

change_ring (*ring*)

Return the matrix obtained by coercing the entries of this matrix into the given ring.

Always returns a copy (unless self is immutable, in which case returns self).

EXAMPLES:

```
sage: A = matrix(QQ['x,y'], 2, [0,-1,2*x,-2], sparse=True); A
[  0  -1]
[2*x  -2]
sage: A.change_ring(QQ['x,y,z'])
[  0  -1]
[2*x  -2]
```

Subdivisions are preserved when changing rings:

```
sage: A.subdivide([2],[]); A
[  0  -1]
[2*x  -2]
[-------]
sage: A.change_ring(RR['x,y'])
[                 0  -1.00000000000000]
[2.00000000000000*x  -2.00000000000000]
[-------------------------------------]
```

charpoly (*var='x'*, ***kwds*)

Return the characteristic polynomial of this matrix.

Note - the generic sparse charpoly implementation in Sage is to just compute the charpoly of the corresponding dense matrix, so this could use a lot of memory. In particular, for this matrix, the charpoly will be computed using a dense algorithm.

EXAMPLES:

```
sage: A = matrix(ZZ, 4, range(16), sparse=True)
sage: A.charpoly()
x^4 - 30*x^3 - 80*x^2
sage: A.charpoly('y')
y^4 - 30*y^3 - 80*y^2
sage: A.charpoly()
x^4 - 30*x^3 - 80*x^2
```

density ( )

Return the density of the matrix.

By density we understand the ratio of the number of nonzero positions and the self.nrows() * self.ncols(), i.e. the number of possible nonzero positions.

EXAMPLES:

```
sage: a = matrix([[],[],[],[]], sparse=True); a.density()
0
sage: a = matrix(5000,5000,{(1,2): 1}); a.density()
1/25000000
```

determinant (***kwds*)

Return the determinant of this matrix.

Note: the generic sparse determinant implementation in Sage is to just compute the determinant of the corresponding dense matrix, so this could use a lot of memory. In particular, for this matrix, the determinant will be computed using a dense algorithm.

EXAMPLES:

```
sage: A = matrix(ZZ, 4, range(16), sparse=True)
sage: B = A + identity_matrix(ZZ, 4, sparse=True)
sage: B.det()
-49
```

matrix_from_rows_and_columns (*rows*, *columns*)

Return the matrix constructed from self from the given rows and columns.

EXAMPLES:

```
sage: M = MatrixSpace(Integers(8),3,3, sparse=True)
sage: A = M(range(9)); A
[0 1 2]
[3 4 5]
[6 7 0]
sage: A.matrix_from_rows_and_columns([1], [0,2])
[3 5]
sage: A.matrix_from_rows_and_columns([1,2], [1,2])
```

```
[4 5]
[7 0]
```

Note that row and column indices can be reordered or repeated:

```
sage: A.matrix_from_rows_and_columns([2,1], [2,1])
[0 7]
[5 4]
```

For example here we take from row 1 columns 2 then 0 twice, and do this 3 times.

```
sage: A.matrix_from_rows_and_columns([1,1,1],[2,0,0])
[5 3 3]
[5 3 3]
[5 3 3]
```

We can efficiently extract large submatrices:

```
sage: A = random_matrix(ZZ, 100000, density=.00005, sparse=True)  # long time␣
↪(4s on sage.math, 2012)
sage: B = A[50000:,:50000]        # long time
sage: len(B.nonzero_positions())  # long time
17550               # 32-bit
125449              # 64-bit
```

We must pass in a list of indices:

```
sage: A=random_matrix(ZZ,100,density=.02,sparse=True)
sage: A.matrix_from_rows_and_columns(1,[2,3])
Traceback (most recent call last):
...
TypeError: rows must be a list of integers
sage: A.matrix_from_rows_and_columns([1,2],3)
Traceback (most recent call last):
...
TypeError: columns must be a list of integers
```

AUTHORS:

- Jaap Spies (2006-02-18)
- Didier Deshommes: some Pyrex speedups implemented
- Jason Grout: sparse matrix optimizations

transpose ()

Returns the transpose of self, without changing self.

EXAMPLES: We create a matrix, compute its transpose, and note that the original matrix is not changed.

```
sage: M = MatrixSpace(QQ,  2, sparse=True)
sage: A = M([1,2,3,4])
sage: B = A.transpose()
sage: print(B)
[1 3]
[2 4]
sage: print(A)
[1 2]
[3 4]
```

`.T` is a convenient shortcut for the transpose:

```
sage: A.T
[1 3]
[2 4]
```

CHAPTER

THIRTEEN

DENSE MATRICES OVER A GENERAL RING

class sage.matrix.matrix_generic_dense.**Matrix_generic_dense**

Bases: *sage.matrix.matrix_dense.Matrix_dense*

The Matrix_generic_dense class derives from Matrix, and defines functionality for dense matrices over any base ring. Matrices are represented by a list of elements in the base ring, and element access operations are implemented in this class.

EXAMPLES:

```
sage: A = random_matrix(Integers(25)['x'],2); A
[     x^2 + 12*x + 2    4*x^2 + 13*x + 8]
[ 22*x^2 + 2*x + 17 19*x^2 + 22*x + 14]
sage: type(A)
<type 'sage.matrix.matrix_generic_dense.Matrix_generic_dense'>
sage: TestSuite(A).run()
```

Test comparisons:

```
sage: A = random_matrix(Integers(25)['x'],2)
sage: A == A
True
sage: A < A + 1
True
sage: A+1 < A
False
```

CHAPTER

FOURTEEN

SPARSE MATRICES OVER A GENERAL RING

EXAMPLES:

```
sage: R.<x> = PolynomialRing(QQ)
sage: M = MatrixSpace(QQ['x'],2,3,sparse=True); M
Full MatrixSpace of 2 by 3 sparse matrices over Univariate Polynomial Ring in x over
↪Rational Field
sage: a = M(range(6)); a
[0 1 2]
[3 4 5]
sage: b = M([x^n for n in range(6)]); b
[  1    x x^2]
[x^3 x^4 x^5]
sage: a * b.transpose()
[            2*x^2 + x           2*x^5 + x^4]
[      5*x^2 + 4*x + 3 5*x^5 + 4*x^4 + 3*x^3]
sage: pari(a)*pari(b.transpose())
[2*x^2 + x, 2*x^5 + x^4; 5*x^2 + 4*x + 3, 5*x^5 + 4*x^4 + 3*x^3]
sage: c = copy(b); c
[  1    x x^2]
[x^3 x^4 x^5]
sage: c[0,0] = 5; c
[  5    x x^2]
[x^3 x^4 x^5]
sage: b[0,0]
1
sage: c.dict()
{(0, 0): 5, (0, 1): x, (0, 2): x^2, (1, 0): x^3, (1, 1): x^4, (1, 2): x^5}
sage: c.list()
[5, x, x^2, x^3, x^4, x^5]
sage: c.rows()
[(5, x, x^2), (x^3, x^4, x^5)]
sage: TestSuite(c).run()
sage: d = c.change_ring(CC['x']); d
[5.00000000000000                x               x^2]
[             x^3              x^4               x^5]
sage: latex(c)
\left(\begin{array}{rrr}
5 & x & x^{2} \\
x^{3} & x^{4} & x^{5}
\end{array}\right)
sage: c.sparse_rows()
[(5, x, x^2), (x^3, x^4, x^5)]
sage: d = c.dense_matrix(); d
[  5    x x^2]
[x^3 x^4 x^5]
```

```
sage: parent(d)
Full MatrixSpace of 2 by 3 dense matrices over Univariate Polynomial Ring in x over␣
↪Rational Field
sage: c.sparse_matrix() is c
True
sage: c.is_sparse()
True
```

class sage.matrix.matrix_generic_sparse.**Matrix_generic_sparse**

Bases: *sage.matrix.matrix_sparse.Matrix_sparse*

Generic sparse matrix.

The Matrix_generic_sparse class derives from *Matrix_sparse*, and defines functionality for sparse matrices over any base ring. A generic sparse matrix is represented using a dictionary whose keys are pairs of integers (i, j) and values in the base ring. The values of the dictionary must never be zero.

EXAMPLES:

```
sage: R.<a,b> = PolynomialRing(ZZ,'a,b')
sage: M = MatrixSpace(R,5,5,sparse=True)
sage: M({(0,0):5*a+2*b, (3,4): -a})
[5*a + 2*b         0         0         0         0]
[        0         0         0         0         0]
[        0         0         0         0         0]
[        0         0         0         0        -a]
[        0         0         0         0         0]
sage: M(3)
[3 0 0 0 0]
[0 3 0 0 0]
[0 0 3 0 0]
[0 0 0 3 0]
[0 0 0 0 3]
sage: V = FreeModule(ZZ, 5,sparse=True)
sage: m = M([V({0:3}), V({2:2, 4:-1}), V(0), V(0), V({1:2})])
sage: m
[ 3  0  0  0  0]
[ 0  0  2  0 -1]
[ 0  0  0  0  0]
[ 0  0  0  0  0]
[ 0  2  0  0  0]
```

Note: The datastructure can potentially be optimized. Firstly, as noticed in trac ticket #17663, we lose time in using 2-tuples to store indices. Secondly, there is no fast way to access non-zero elements in a given row/column.

sage.matrix.matrix_generic_sparse.**Matrix_sparse_from_rows** (*X*)

INPUT:

- X - nonempty list of SparseVector rows

OUTPUT: Sparse_matrix with those rows.

EXAMPLES:

```
sage: V = VectorSpace(QQ,20,sparse=True)
sage: v = V(0)
sage: v[9] = 4
sage: from sage.matrix.matrix_generic_sparse import Matrix_sparse_from_rows
```

```
sage: Matrix_sparse_from_rows([v])
[0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0]
sage: Matrix_sparse_from_rows([v, v, v, V(0)])
[0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 4 0 0 0 0 0 0 0 0 0 0]
[0 0 0 0 0 0 0 0 0 4 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]
```

CHAPTER

FIFTEEN

SPARSE MATRICES OVER $\mathbb{Z}/N\mathbb{Z}$ FOR N SMALL

This is a compiled implementation of sparse matrices over $\mathbf{Z}/n\mathbf{Z}$ for n small.

TODO: - move vectors into a Cython vector class - add _add_ and _mul_ methods.

EXAMPLES:

```
sage: a = matrix(Integers(37),3,3,range(9),sparse=True); a
[0 1 2]
[3 4 5]
[6 7 8]
sage: type(a)
<type 'sage.matrix.matrix_modn_sparse.Matrix_modn_sparse'>
sage: parent(a)
Full MatrixSpace of 3 by 3 sparse matrices over Ring of integers modulo 37
sage: a^2
[15 18 21]
[ 5 17 29]
[32 16  0]
sage: a+a
[ 0  2  4]
[ 6  8 10]
[12 14 16]
sage: b = a.new_matrix(2,3,range(6)); b
[0 1 2]
[3 4 5]
sage: a*b
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for *: 'Full MatrixSpace of 3 by 3 sparse␣
↪matrices over Ring of integers modulo 37' and 'Full MatrixSpace of 2 by 3 sparse␣
↪matrices over Ring of integers modulo 37'
sage: b*a
[15 18 21]
[ 5 17 29]
```

```
sage: TestSuite(a).run()
sage: TestSuite(b).run()
```

```
sage: a.echelonize(); a
[ 1  0 36]
[ 0  1  2]
[ 0  0  0]
sage: b.echelonize(); b
[ 1  0 36]
```

```
[ 0  1  2]
sage: a.pivots()
(0, 1)
sage: b.pivots()
(0, 1)
sage: a.rank()
2
sage: b.rank()
2
sage: a[2,2] = 5
sage: a.rank()
3
```

class sage.matrix.matrix_modn_sparse.**Matrix_modn_sparse**

Bases: *sage.matrix.matrix_sparse.Matrix_sparse*

Create a sparse matrix over the integers modulo `n`.

INPUT:

- `parent` – a matrix space
- `entries` – can be one of the following:
  - a Python dictionary whose items have the form `(i, j): x`, where `0 <= i < nrows`, `0 <= j < ncols`, and `x` is coercible to an element of the integers modulo `n`. The `i, j` entry of `self` is set to `x`. The `x` 's can be `0`.
  - Alternatively, entries can be a list of *all* the entries of the sparse matrix, read row-by-row from top to bottom (so they would be mostly 0).
- `copy` – ignored
- `coerce` – ignored

density()

Return the density of self, i.e., the ratio of the number of nonzero entries of self to the total size of self.

EXAMPLES:

```
sage: A = matrix(QQ,3,3,[0,1,2,3,0,0,6,7,8],sparse=True)
sage: A.density()
2/3
```

Notice that the density parameter does not ensure the density of a matrix; it is only an upper bound.

```
sage: A = random_matrix(GF(127),200,200,density=0.3, sparse=True)
sage: A.density()
2073/8000
```

lift()

Return lift of this matrix to a sparse matrix over the integers.

EXAMPLES: sage: a = matrix(GF(7),2,3,[1..6], sparse=True) sage: a.lift() [1 2 3] [4 5 6] sage: a.lift().parent() Full MatrixSpace of 2 by 3 sparse matrices over Integer Ring

Subdivisions are preserved when lifting:

```
sage: a.subdivide([], [1,1]); a
[1||2 3]
[4||5 6]
sage: a.lift()
```

```
[1||2 3]
[4||5 6]
```

matrix_from_columns (*cols*)

Return the matrix constructed from self using columns with indices in the columns list.

EXAMPLES:

```
sage: M = MatrixSpace(GF(127),3,3,sparse=True)
sage: A = M(range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: A.matrix_from_columns([2,1])
[2 1]
[5 4]
[8 7]
```

matrix_from_rows (*rows*)

Return the matrix constructed from self using rows with indices in the rows list.

INPUT:

- `rows` - list or tuple of row indices

EXAMPLES:

```
sage: M = MatrixSpace(GF(127),3,3,sparse=True)
sage: A = M(range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: A.matrix_from_rows([2,1])
[6 7 8]
[3 4 5]
```

p

rank (*gauss=False*)

Compute the rank of self.

INPUT:

- `gauss` - if True LinBox' Gaussian elimination is used. If False 'Symbolic Reordering' as implemented in LinBox is used. If 'native' the native Sage implementation is used. (default: False)

EXAMPLES:

```
sage: A = random_matrix(GF(127),200,200,density=0.01,sparse=True)
sage: r1 = A.rank(gauss=False)
sage: r2 = A.rank(gauss=True)
sage: r3 = A.rank(gauss='native')
sage: r1 == r2 == r3
True
sage: r1
155
```

ALGORITHM: Uses LinBox or native implementation.

REFERENCES:

- Jean-Guillaume Dumas and Gilles Villars. 'Computing the Rank of Large Sparse Matrices over Finite Fields'. Proc. CASC'2002, The Fifth International Workshop on Computer Algebra in Scientific Computing, Big Yalta, Crimea, Ukraine, 22-27 sept. 2002, Springer-Verlag, http://perso.ens-lyon.fr/gilles.villard/BIBLIOGRAPHIE/POSTSCRIPT/rankjgd.ps

Note: For very sparse matrices Gaussian elimination is faster because it barly has anything to do. If the fill in needs to be considered, 'Symbolic Reordering' is usually much faster.

swap_rows (*r1*, *r2*)

transpose ( )

Return the transpose of self.

EXAMPLES:

```
sage: A = matrix(GF(127),3,3,[0,1,0,2,0,0,3,0,0],sparse=True)
sage: A
[0 1 0]
[2 0 0]
[3 0 0]
sage: A.transpose()
[0 2 3]
[1 0 0]
[0 0 0]
```

`.T` is a convenient shortcut for the transpose:

```
sage: A.T
[0 2 3]
[1 0 0]
[0 0 0]
```

CHAPTER SIXTEEN

SYMBOLIC MATRICES

Matrices with symbolic entries. The underlying representation is a pointer to a Maxima object.

EXAMPLES:

```
sage: matrix(SR, 2, 2, range(4))
[0 1]
[2 3]
sage: matrix(SR, 2, 2, var('t'))
[t 0]
[0 t]
```

Arithmetic:

```
sage: -matrix(SR, 2, range(4))
[ 0 -1]
[-2 -3]
sage: m = matrix(SR, 2, [1..4]); sqrt(2)*m
[  sqrt(2) 2*sqrt(2)]
[3*sqrt(2) 4*sqrt(2)]
sage: m = matrix(SR, 4, [1..4^2])
sage: m * m
[ 90 100 110 120]
[202 228 254 280]
[314 356 398 440]
[426 484 542 600]

sage: m = matrix(SR, 3, [1, 2, 3]); m
[1]
[2]
[3]
sage: m.transpose() * m
[14]
```

Computing inverses:

```
sage: M = matrix(SR, 2, var('a,b,c,d'))
sage: ~M
[1/a - b*c/(a^2*(b*c/a - d))           b/(a*(b*c/a - d))]
[          c/(a*(b*c/a - d))              -1/(b*c/a - d)]
sage: (~M*M).simplify_rational()
[1 0]
[0 1]
sage: M = matrix(SR, 3, 3, range(9)) - var('t')
sage: (~M * M).simplify_rational()
[1 0 0]
```

```
[0 1 0]
[0 0 1]

sage: matrix(SR, 1, 1, 1).inverse()
[1]
sage: matrix(SR, 0, 0).inverse()
[]
sage: matrix(SR, 3, 0).inverse()
Traceback (most recent call last):
...
ArithmeticError: self must be a square matrix
```

Transposition:

```
sage: m = matrix(SR, 2, [sqrt(2), -1, pi, e^2])
sage: m.transpose()
[sqrt(2)      pi]
[     -1     e^2]
```

`.T` is a convenient shortcut for the transpose:

```
sage: m.T
[sqrt(2)      pi]
[     -1     e^2]
```

Test pickling:

```
sage: m = matrix(SR, 2, [sqrt(2), 3, pi, e]); m
[sqrt(2)       3]
[     pi       e]
sage: TestSuite(m).run()
```

Comparison:

```
sage: m = matrix(SR, 2, [sqrt(2), 3, pi, e])
sage: m == m
True
sage: m != 3
True
sage: m = matrix(SR,2,[1..4]); n = m^2
sage: (exp(m+n) - exp(m)*exp(n)).simplify_rational() == 0       # indirect test
True
```

Determinant:

```
sage: M = matrix(SR, 2, 2, [x,2,3,4])
sage: M.determinant()
4*x - 6
sage: M = matrix(SR, 3,3,range(9))
sage: M.det()
0
sage: t = var('t')
sage: M = matrix(SR, 2, 2, [cos(t), sin(t), -sin(t), cos(t)])
sage: M.det()
cos(t)^2 + sin(t)^2
sage: M = matrix([[sqrt(x),0,0,0], [0,1,0,0], [0,0,1,0], [0,0,0,1]])
sage: det(M)
sqrt(x)
```

Permanents:

```
sage: M = matrix(SR, 2, 2, [x,2,3,4])
sage: M.permanent()
4*x + 6
```

Rank:

```
sage: M = matrix(SR, 5, 5, range(25))
sage: M.rank()
2
sage: M = matrix(SR, 5, 5, range(25)) - var('t')
sage: M.rank()
5

.. warning::

    :meth:`rank` may return the wrong answer if it cannot determine that a
    matrix element that is equivalent to zero is indeed so.
```

Copying symbolic matrices:

```
sage: m = matrix(SR, 2, [sqrt(2), 3, pi, e])
sage: n = copy(m)
sage: n[0,0] = sin(1)
sage: m
[sqrt(2)       3]
[     pi       e]
sage: n
[sin(1)      3]
[    pi      e]
```

Conversion to Maxima:

```
sage: m = matrix(SR, 2, [sqrt(2), 3, pi, e])
sage: m._maxima_()
matrix([sqrt(2),3],[%pi,%e])
```

class `sage.matrix.matrix_symbolic_dense.` **`Matrix_symbolic_dense`**

Bases: *`sage.matrix.matrix_generic_dense.Matrix_generic_dense`*

`arguments` ( )

Returns a tuple of the arguments that self can take.

EXAMPLES:

```
sage: var('x,y,z')
(x, y, z)
sage: M = MatrixSpace(SR,2,2)
sage: M(x).arguments()
(x,)
sage: M(x+sin(x)).arguments()
(x,)
```

`canonicalize_radical` ( )

Choose a canonical branch of each entrie of `self` by calling `Expression.canonicalize_radical()` componentwise.

EXAMPLES:

```
sage: var('x','y')
(x, y)
sage: l1 = [sqrt(2)*sqrt(3)*sqrt(6) , log(x*y)]
sage: l2 = [sin(x/(x^2 + x)) , 1]
sage: m = matrix([l1, l2])
sage: m
[sqrt(6)*sqrt(3)*sqrt(2)                log(x*y)]
[       sin(x/(x^2 + x))                       1]
sage: m.canonicalize_radical()
[              6 log(x) + log(y)]
[ sin(1/(x + 1))                1]
```

charpoly (*var='x'*, *algorithm=None*)

Compute the characteristic polynomial of self, using maxima.

EXAMPLES:

```
sage: M = matrix(SR, 2, 2, var('a,b,c,d'))
sage: M.charpoly('t')
t^2 + (-a - d)*t - b*c + a*d
sage: matrix(SR, 5, [1..5^2]).charpoly()
x^5 - 65*x^4 - 250*x^3
```

eigenvalues ( )

Compute the eigenvalues by solving the characteristic polynomial in maxima

EXAMPLES:

```
sage: a=matrix(SR,[[1,2],[3,4]])
sage: a.eigenvalues()
[-1/2*sqrt(33) + 5/2, 1/2*sqrt(33) + 5/2]
```

eigenvectors_left ( )

Compute the left eigenvectors of a matrix.

For each distinct eigenvalue, returns a list of the form (e,V,n) where e is the eigenvalue, V is a list of eigenvectors forming a basis for the corresponding left eigenspace, and n is the algebraic multiplicity of the eigenvalue.

EXAMPLES:

```
sage: A = matrix(SR,3,3,range(9)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: es = A.eigenvectors_left(); es
[(-3*sqrt(6) + 6, [(1, -1/5*sqrt(6) + 4/5, -2/5*sqrt(6) + 3/5)], 1),
 (3*sqrt(6) + 6, [(1, 1/5*sqrt(6) + 4/5, 2/5*sqrt(6) + 3/5)], 1),
 (0, [(1, -2, 1)], 1)]
sage: eval, [evec], mult = es[0]
sage: delta = eval*evec - evec*A
sage: abs(abs(delta)) < 1e-10
sqrt(9/25*((2*sqrt(6) - 3)*(sqrt(6) - 2) + 7*sqrt(6) - 18)^2 + 9/25*((sqrt(6)␣
↪- 2)*(sqrt(6) - 4) + 6*sqrt(6) - 14)^2) < (1.00000000000000e-10)
sage: abs(abs(delta)).n() < 1e-10
True
```

```
sage: A = matrix(SR, 2, 2, var('a,b,c,d'))
sage: A.eigenvectors_left()
[(1/2*a + 1/2*d - 1/2*sqrt(a^2 + 4*b*c - 2*a*d + d^2), [(1, -1/2*(a - d +␣
↪sqrt(a^2 + 4*b*c - 2*a*d + d^2))/c)], 1), (1/2*a + 1/2*d + 1/2*sqrt(a^2 +␣
↪4*b*c - 2*a*d + d^2), [(1, -1/2*(a - d - sqrt(a^2 + 4*b*c - 2*a*d + d^2))/
↪c)], 1)]
sage: es = A.eigenvectors_left(); es
[(1/2*a + 1/2*d - 1/2*sqrt(a^2 + 4*b*c - 2*a*d + d^2), [(1, -1/2*(a - d +␣
↪sqrt(a^2 + 4*b*c - 2*a*d + d^2))/c)], 1), (1/2*a + 1/2*d + 1/2*sqrt(a^2 +␣
↪4*b*c - 2*a*d + d^2), [(1, -1/2*(a - d - sqrt(a^2 + 4*b*c - 2*a*d + d^2))/
↪c)], 1)]
sage: eval, [evec], mult = es[0]
sage: delta = eval*evec - evec*A
sage: delta.apply_map(lambda x: x.full_simplify())
(0, 0)
```

This routine calls Maxima and can struggle with even small matrices with a few variables, such as a 3×3 matrix with three variables. However, if the entries are integers or rationals it can produce exact values in a reasonable time. These examples create 0-1 matrices from the adjacency matrices of graphs and illustrate how the format and type of the results differ when the base ring changes. First for matrices over the rational numbers, then the same matrix but viewed as a symbolic matrix.

```
sage: G=graphs.CycleGraph(5)
sage: am = G.adjacency_matrix()
sage: spectrum = am.eigenvectors_left()
sage: qqbar_evalue = spectrum[2][0]
sage: type(qqbar_evalue)
<class 'sage.rings.qqbar.AlgebraicNumber'>
sage: qqbar_evalue
0.618033988749895?

sage: am = G.adjacency_matrix().change_ring(SR)
sage: spectrum = am.eigenvectors_left()
sage: symbolic_evalue = spectrum[2][0]
sage: type(symbolic_evalue)
<type 'sage.symbolic.expression.Expression'>
sage: symbolic_evalue
1/2*sqrt(5) - 1/2

sage: bool(qqbar_evalue == symbolic_evalue)
True
```

A slightly larger matrix with a "nice" spectrum.

```
sage: G=graphs.CycleGraph(6)
sage: am = G.adjacency_matrix().change_ring(SR)
sage: am.eigenvectors_left()
[(-1, [(1, 0, -1, 1, 0, -1), (0, 1, -1, 0, 1, -1)], 2), (1, [(1, 0, -1, -1,␣
↪0, 1), (0, 1, 1, 0, -1, -1)], 2), (-2, [(1, -1, 1, -1, 1, -1)], 1), (2,␣
↪[(1, 1, 1, 1, 1, 1)], 1)]
```

eigenvectors_right ( )

Compute the right eigenvectors of a matrix.

For each distinct eigenvalue, returns a list of the form (e,V,n) where e is the eigenvalue, V is a list of eigenvectors forming a basis for the corresponding right eigenspace, and n is the algebraic multiplicity of the eigenvalue.

EXAMPLES:

```
sage: A = matrix(SR,2,2,range(4)); A
[0 1]
[2 3]
sage: right = A.eigenvectors_right(); right
[(-1/2*sqrt(17) + 3/2, [(1, -1/2*sqrt(17) + 3/2)], 1), (1/2*sqrt(17) + 3/2,␣
↪[(1, 1/2*sqrt(17) + 3/2)], 1)]
```

The right eigenvectors are nothing but the left eigenvectors of the transpose matrix:

```
sage: left  = A.transpose().eigenvectors_left(); left
[(-1/2*sqrt(17) + 3/2, [(1, -1/2*sqrt(17) + 3/2)], 1), (1/2*sqrt(17) + 3/2,␣
↪[(1, 1/2*sqrt(17) + 3/2)], 1)]
sage: right[0][1] == left[0][1]
True
```

exp ( )

Return the matrix exponential of this matrix X, which is the matrix

$$e^X = \sum_{k=0}^{\infty} \frac{X^k}{k!}.$$

This function depends on maxima's matrix exponentiation function, which does not deal well with floating point numbers. If the matrix has floating point numbers, they will be rounded automatically to rational numbers during the computation.

EXAMPLES:

```
sage: m = matrix(SR,2, [0,x,x,0]); m
[0 x]
[x 0]
sage: m.exp()
[1/2*(e^(2*x) + 1)*e^(-x) 1/2*(e^(2*x) - 1)*e^(-x)]
[1/2*(e^(2*x) - 1)*e^(-x) 1/2*(e^(2*x) + 1)*e^(-x)]
sage: exp(m)
[1/2*(e^(2*x) + 1)*e^(-x) 1/2*(e^(2*x) - 1)*e^(-x)]
[1/2*(e^(2*x) - 1)*e^(-x) 1/2*(e^(2*x) + 1)*e^(-x)]
```

Exp works on 0x0 and 1x1 matrices:

```
sage: m = matrix(SR,0,[]); m
[]
sage: m.exp()
[]
sage: m = matrix(SR,1,[2]); m
[2]
sage: m.exp()
[e^2]
```

Commuting matrices m, n have the property that $e^{m+n} = e^m e^n$ (but non-commuting matrices need not):

```
sage: m = matrix(SR,2,[1..4]); n = m^2
sage: m*n
[ 37  54]
[ 81 118]
sage: n*m
[ 37  54]
```

```
[ 81 118]

sage: a = exp(m+n) - exp(m)*exp(n)
sage: a.simplify_rational() == 0
True
```

The input matrix must be square:

```
sage: m = matrix(SR,2,3,[1..6]); exp(m)
Traceback (most recent call last):
...
ValueError: exp only defined on square matrices
```

In this example we take the symbolic answer and make it numerical at the end:

```
sage: exp(matrix(SR, [[1.2, 5.6], [3,4]])).change_ring(RDF)  # rel tol 1e-15
[ 346.5574872980695  661.7345909344504]
[354.50067371488416  677.4247827652946]
```

Another example involving the reversed identity matrix, which we clumsily create:

```
sage: m = identity_matrix(SR,4); m = matrix(list(reversed(m.rows()))) * x
sage: exp(m)
[1/2*(e^(2*x) + 1)*e^(-x)                         0                         0 1/
↪2*(e^(2*x) - 1)*e^(-x)]
[                        0 1/2*(e^(2*x) + 1)*e^(-x) 1/2*(e^(2*x) - 1)*e^(-x)   ␣
↪                       0]
[                        0 1/2*(e^(2*x) - 1)*e^(-x) 1/2*(e^(2*x) + 1)*e^(-x)   ␣
↪                       0]
[1/2*(e^(2*x) - 1)*e^(-x)                         0                         0 1/
↪2*(e^(2*x) + 1)*e^(-x)]
```

expand ( )

Operates point-wise on each element.

EXAMPLES:

```
sage: M = matrix(2, 2, range(4)) - var('x')
sage: M*M
[      x^2 + 2       -2*x + 3]
[     -4*x + 6 (x - 3)^2 + 2]
sage: (M*M).expand()
[       x^2 + 2          -2*x + 3]
[      -4*x + 6 x^2 - 6*x + 11]
```

factor ( )

Operates point-wise on each element.

EXAMPLES:

```
sage: M = matrix(SR, 2, 2, x^2 - 2*x + 1); M
[x^2 - 2*x + 1             0]
[            0 x^2 - 2*x + 1]
sage: M.factor()
[(x - 1)^2         0]
[        0 (x - 1)^2]
```

fcp (*var='x'*)

Return the factorization of the characteristic polynomial of self.

INPUT:

- `var` - (default: 'x') name of variable of charpoly

EXAMPLES:

```
sage: a = matrix(SR,[[1,2],[3,4]])
sage: a.fcp()
x^2 - 5*x - 2
sage: [i for i in a.fcp()]
[(x^2 - 5*x - 2, 1)]
sage: a = matrix(SR,[[1,0],[0,2]])
sage: a.fcp()
(x - 2) * (x - 1)
sage: [i for i in a.fcp()]
[(x - 2, 1), (x - 1, 1)]
sage: a = matrix(SR, 5, [1..5^2])
sage: a.fcp()
(x^2 - 65*x - 250) * x^3
sage: list(a.fcp())
[(x^2 - 65*x - 250, 1), (x, 3)]
```

jordan_form (*subdivide=True*, *transformation=False*)

Return a Jordan normal form of `self` .

INPUT:

- `self` – a square matrix
- `subdivide` – boolean (default: `True`)
- `transformation` – boolean (default: `False`)

OUTPUT:

If `transformation` is `False` , only a Jordan normal form (unique up to the ordering of the Jordan blocks) is returned. Otherwise, a pair `(J,P)` is returned, where `J` is a Jordan normal form and `P` is an invertible matrix such that `self` equals `P * J * P^(-1)` .

If `subdivide` is `True` , the Jordan blocks in the returned matrix `J` are indicated by a subdivision in the sense of `subdivide()` .

EXAMPLES:

We start with some examples of diagonalisable matrices:

```
sage: a,b,c,d = var('a,b,c,d')
sage: matrix([a]).jordan_form()
[a]
sage: matrix([[a, 0], [1, d]]).jordan_form(subdivide=True)
[d|0]
[-+-]
[0|a]
sage: matrix([[a, 0], [1, d]]).jordan_form(subdivide=False)
[d 0]
[0 a]
sage: matrix([[a, x, x], [0, b, x], [0, 0, c]]).jordan_form()
[c|0|0]
[-+-+-]
[0|b|0]
[-+-+-]
[0|0|a]
```

In the following examples, we compute Jordan forms of some non-diagonalisable matrices:

```
sage: matrix([[a, a], [0, a]]).jordan_form()
[a 1]
[0 a]
sage: matrix([[a, 0, b], [0, c, 0], [0, 0, a]]).jordan_form()
[c|0 0]
[-+---]
[0|a 1]
[0|0 a]
```

The following examples illustrate the `transformation` flag. Note that symbolic expressions may need to be simplified to make consistency checks succeed:

```
sage: A = matrix([[x - a*c, a^2], [-c^2, x + a*c]])
sage: J, P = A.jordan_form(transformation=True)
sage: J, P
(
[x 1]  [-a*c    1]
[0 x], [-c^2    0]
)
sage: A1 = P * J * ~P; A1
[             -a*c + x (a*c - x)*a/c + a*x/c]
[                 -c^2               a*c + x]
sage: A1.simplify_rational() == A
True

sage: B = matrix([[a, b, c], [0, a, d], [0, 0, a]])
sage: J, T = B.jordan_form(transformation=True)
sage: J, T
(
[a 1 0]  [b*d   c   0]
[0 a 1]  [  0   d   0]
[0 0 a], [  0   0   1]
)
sage: (B * T).simplify_rational() == T * J
True
```

Finally, some examples involving square roots:

```
sage: matrix([[a, -b], [b, a]]).jordan_form()
[a - I*b|      0]
[-------+-------]
[      0|a + I*b]
sage: matrix([[a, b], [c, d]]).jordan_form(subdivide=False)
[1/2*a + 1/2*d - 1/2*sqrt(a^2 + 4*b*c - 2*a*d + d^2)
↪                          0]
[                                                  0 1/2*a + 1/2*d + 1/
↪2*sqrt(a^2 + 4*b*c - 2*a*d + d^2)]
```

minpoly (*var='x'*)

Return the minimal polynomial of `self`.

EXAMPLES:

```
sage: M = Matrix.identity(SR, 2)
sage: M.minpoly()
x - 1
```

```
sage: t = var('t')
sage: m = matrix(2, [1, 2, 4, t])
sage: m.minimal_polynomial()
x^2 + (-t - 1)*x + t - 8
```

number_of_arguments ( )

Returns the number of arguments that self can take.

EXAMPLES:

```
sage: var('a,b,c,x,y')
(a, b, c, x, y)
sage: m = matrix([[a, (x+y)/(x+y)], [x^2, y^2+2]]); m
[      a       1]
[    x^2 y^2 + 2]
sage: m.number_of_arguments()
3
```

simplify ( )

Simplifies self.

EXAMPLES:

```
sage: var('x,y,z')
(x, y, z)
sage: m = matrix([[z, (x+y)/(x+y)], [x^2, y^2+2]]); m
[      z       1]
[    x^2 y^2 + 2]
sage: m.simplify()
[      z       1]
[    x^2 y^2 + 2]
```

simplify_full ( )

Simplify a symbolic matrix by calling `Expression.simplify_full()` componentwise.

INPUT:

- `self` - The matrix whose entries we should simplify.

OUTPUT:

A copy of `self` with all of its entries simplified.

EXAMPLES:

Symbolic matrices will have their entries simplified:

```
sage: a,n,k = SR.var('a,n,k')
sage: f1 = sin(x)^2 + cos(x)^2
sage: f2 = sin(x/(x^2 + x))
sage: f3 = binomial(n,k)*factorial(k)*factorial(n-k)
sage: f4 = x*sin(2)/(x^a)
sage: A = matrix(SR, [[f1,f2],[f3,f4]])
sage: A.simplify_full()
[             1     sin(1/(x + 1))]
[     factorial(n) x^(-a + 1)*sin(2)]
```

simplify_rational ( )

EXAMPLES:

```
sage: M = matrix(SR, 3, 3, range(9)) - var('t')
sage: (~M*M)[0,0]
t*(3*(2/t + (6/t + 7)/((t - 3/t - 4)*t))*(2/t + (6/t + 5)/((t - 3/t
- 4)*t))/(t - (6/t + 7)*(6/t + 5)/(t - 3/t - 4) - 12/t - 8) + 1/t +
3/((t - 3/t - 4)*t^2)) - 6*(2/t + (6/t + 5)/((t - 3/t - 4)*t))/(t -
(6/t + 7)*(6/t + 5)/(t - 3/t - 4) - 12/t - 8) - 3*(6/t + 7)*(2/t +
(6/t + 5)/((t - 3/t - 4)*t))/((t - (6/t + 7)*(6/t + 5)/(t - 3/t -
4) - 12/t - 8)*(t - 3/t - 4)) - 3/((t - 3/t - 4)*t)
sage: expand((~M*M)[0,0])
1
sage: (~M * M).simplify_rational()
[1 0 0]
[0 1 0]
[0 0 1]
```

simplify_trig ()

EXAMPLES:

```
sage: theta = var('theta')
sage: M = matrix(SR, 2, 2, [cos(theta), sin(theta), -sin(theta), cos(theta)])
sage: ~M
[1/cos(theta) - sin(theta)^2/((sin(theta)^2/cos(theta) +␣
↪cos(theta))*cos(theta)^2)                      -sin(theta)/((sin(theta)^2/
↪cos(theta) + cos(theta))*cos(theta))]
[                    sin(theta)/((sin(theta)^2/cos(theta) +␣
↪cos(theta))*cos(theta))                                           1/
↪(sin(theta)^2/cos(theta) + cos(theta))]
sage: (~M).simplify_trig()
[ cos(theta) -sin(theta)]
[ sin(theta)  cos(theta)]
```

variables ()

Returns the variables of self.

EXAMPLES:

```
sage: var('a,b,c,x,y')
(a, b, c, x, y)
sage: m = matrix([[x, x+2], [x^2, x^2+2]]); m
[      x   x + 2]
[    x^2 x^2 + 2]
sage: m.variables()
(x,)
sage: m = matrix([[a, b+c], [x^2, y^2+2]]); m
[      a   b + c]
[    x^2 y^2 + 2]
sage: m.variables()
(a, b, c, x, y)
```

CHAPTER

SEVENTEEN

DENSE MATRICES OVER THE INTEGER RING

AUTHORS:

- William Stein
- Robert Bradshaw
- Marc Masdeu (August 2014). Implemented using FLINT, see trac ticket #16803.
- Jeroen Demeyer (October 2014): lots of fixes, see trac ticket #17090 and trac ticket #17094.
- Vincent Delecroix (February 2015): make it faster, see trac ticket #17822.

EXAMPLES:

```
sage: a = matrix(ZZ, 3,3, range(9)); a
[0 1 2]
[3 4 5]
[6 7 8]
sage: a.det()
0
sage: a[0,0] = 10; a.det()
-30
sage: a.charpoly()
x^3 - 22*x^2 + 102*x + 30
sage: b = -3*a
sage: a == b
False
sage: b < a
True
```

class sage.matrix.matrix_integer_dense. **Matrix_integer_dense**

Bases: *sage.matrix.matrix_dense.Matrix_dense*

Matrix over the integers, implemented using FLINT.

On a 32-bit machine, they can have at most $2^{32} - 1$ rows or columns. On a 64-bit machine, matrices can have at most $2^{64} - 1$ rows or columns.

EXAMPLES:

```
sage: a = MatrixSpace(ZZ,3)(2); a
[2 0 0]
[0 2 0]
[0 0 2]
sage: a = matrix(ZZ,1,3, [1,2,-3]); a
[ 1  2 -3]
sage: a = MatrixSpace(ZZ,2,4)(2); a
Traceback (most recent call last):
```

```
...
TypeError: nonzero scalar matrix must be square
```

BKZ (*delta=None*, *algorithm='fpLLL'*, *fp=None*, *block_size=10*, *prune=0*, *use_givens=False*, *precision=0*, *proof=None*, ***kwds*)

Block Korkin-Zolotarev reduction.

INPUT:

- `delta` – (default: `0.99`) LLL parameter
- `algorithm` – (default: `"fpLLL"`) `"fpLLL"` or `"NTL"`
- `fp` – floating point number implementation
  - `None` – NTL's exact reduction or fpLLL's wrapper (default)
  - `'fp'` – double precision: NTL's FP or fpLLL's double
  - `'ld'` – long doubles (fpLLL only)
  - `'qd'` – NTL's QP
  - **`'qd1'` – quad doubles: Uses `quad_float` precision** to compute Gram-Schmidt, but uses double precision in the search phase of the block reduction algorithm. This seems adequate for most purposes, and is faster than `'qd'`, which uses quad_float precision uniformly throughout (NTL only).
  - `'xd'` – extended exponent: NTL's XD or fpLLL's dpe
  - `'rr'` – arbitrary precision: NTL'RR or fpLLL's MPFR
- `block_size` – (default: `10`) Specifies the size of the blocks in the reduction. High values yield shorter vectors, but the running time increases double exponentially with `block_size`. `block_size` should be between 2 and the number of rows of `self`.
- `proof` – (default: same as `proof.linear_algebra()`) Insist on full BKZ reduction. If disabled and fplll is called, reduction is much faster but the result is not fully BKZ reduced.

NLT SPECIFIC INPUT:

- `prune` – (default: `0`) The optional parameter `prune` can be set to any positive number to invoke the Volume Heuristic from [SH1995]. This can significantly reduce the running time, and hence allow much bigger block size, but the quality of the reduction is of course not as good in general. Higher values of `prune` mean better quality, and slower running time. When `prune` is `0`, pruning is disabled. Recommended usage: for `block_size==30`, set `10 <= prune <=15`.
- `use_givens` – Use Given's orthogonalization. This is a bit slower, but generally much more stable, and is really the preferred orthogonalization strategy. For a nice description of this, see Chapter 5 of [GL1996].

fpLLL SPECIFIC INPUT:

- `precision` – (default: `0` for automatic choice) bit precision to use if `fp='rr'` is set
- `**kwds` – kwds are passed through to fpylll. See $fpylll.fplll.BKZ.Param$ for details.

Also, if the verbose level is at least 2, some output is printed during the computation.

EXAMPLES:

```
sage: A = Matrix(ZZ,3,3,range(1,10))
sage: A.BKZ()
[ 0  0  0]
[ 2  1  0]
[-1  1  3]

sage: A = Matrix(ZZ,3,3,range(1,10))
sage: A.BKZ(use_givens=True)
[ 0  0  0]
[ 2  1  0]
[-1  1  3]

sage: A = Matrix(ZZ,3,3,range(1,10))
sage: A.BKZ(fp="fp")
[ 0  0  0]
[ 2  1  0]
[-1  1  3]
```

ALGORITHM:

Calls either NTL or fpLLL.

LLL (*delta=None*, *eta=None*, *algorithm='fpLLL:wrapper'*, *fp=None*, *prec=0*, *early_red=False*, *use_givens=False*, *use_siegel=False*, ***kwds*)

Return LLL reduced or approximated LLL reduced lattice R for this matrix interpreted as a lattice.

A lattice $(b_1, b_2, ..., b_d)$ is (δ, η)-LLL-reduced if the two following conditions hold:

- For any $i > j$, we have $|\mu_{i,j}| \leq \eta$.
- For any $i < d$, we have $\delta |b_i^*|^2 \leq |b_{i+1}^* + \mu_{i+1,i} b_i^*|^2$,

where $\mu_{i,j} = \langle b_i, b_j^* \rangle / \langle b_j^*, b_j^* \rangle$ and b_i^* is the i-th vector of the Gram-Schmidt orthogonalisation of $(b_1, b_2, ..., b_d)$.

The default reduction parameters are $\delta = 3/4$ and $\eta = 0.501$. The parameters δ and η must satisfy: $0.25 < \delta \leq 1.0$ and $0.5 \leq \eta < \sqrt{\delta}$. Polynomial time complexity is only guaranteed for $\delta < 1$. Not every algorithm admits the case $\delta = 1$.

The lattice is returned as a matrix. Also the rank (and the determinant) of `self` are cached if those are computed during the reduction. Note that in general this only happens when `self.rank() == self.ncols()` and the exact algorithm is used.

INPUT:

- `delta` – (default: `0.99`) δ parameter as described above
- `eta` – (default: `0.501`) η parameter as described above, ignored by NTL
- `algorithm` – string one of the algorithms listed below (default: `"fpLLL:wrapper"`).
- `fp` – floating point number implementation:
  - `None` – NTL's exact reduction or fpLLL's wrapper
  - `'fp'` – double precision: NTL's FP or fpLLL's double
  - `'ld'` – long doubles (fpLLL only)
  - `'qd'` – NTL's QP
  - `'xd'` – extended exponent: NTL's XD or fpLLL's dpe
  - `'rr'` – arbitrary precision: NTL's RR or fpLLL's MPFR

- `prec` – (default: auto choose) precision, ignored by NTL
- `early_red` – (default: `False`) perform early reduction, ignored by NTL
- `use_givens` – (default: `False`) use Givens orthogonalization only applicable to approximate reductions and NTL; this is more stable but slower
- `use_siegel` – (default: `False`) use Siegel's condition instead of Lovasz's condition, ignored by NTL
- `**kwds` – kwds are passed through to fpylll. See $fpylll.fplll.LLL.reduction$ for details.

Also, if the verbose level is at least 2, some output is printed during the computation.

AVAILABLE ALGORITHMS:

- `NTL:LLL` - NTL's LLL + choice of `fp` .
- `fpLLL:heuristic` - fpLLL's heuristic + choice of `fp` .
- `fpLLL:fast` - fpLLL's fast + choice of `fp` .
- `fpLLL:proved` - fpLLL's proved + choice of `fp` .
- `fpLLL:wrapper` - fpLLL's automatic choice (default).

OUTPUT:

A matrix over the integers.

EXAMPLES:

```
sage: A = Matrix(ZZ,3,3,range(1,10))
sage: A.LLL()
[ 0  0  0]
[ 2  1  0]
[-1  1  3]
```

We compute the extended GCD of a list of integers using LLL, this example is from the Magma handbook:

```
sage: Q = [ 67015143, 248934363018, 109210, 25590011055, 74631449,
....:       10230248, 709487, 68965012139, 972065, 864972271 ]
sage: n = len(Q)
sage: S = 100
sage: X = Matrix(ZZ, n, n + 1)
sage: for i in range(n):
....:     X[i, i + 1] = 1
sage: for i in range(n):
....:     X[i, 0] = S * Q[i]
sage: L = X.LLL()
sage: M = L.row(n-1).list()[1:]
sage: M
[-3, -1, 13, -1, -4, 2, 3, 4, 5, -1]
sage: add(Q[i]*M[i] for i in range(n))
-1
```

The case $\delta = 1$ is not always supported:

```
sage: L = X.LLL(delta=2)
Traceback (most recent call last):
...
TypeError: delta must be <= 1
sage: L = X.LLL(delta=1)     # not tested, will eat lots of ram
```

```
Traceback (most recent call last):
...
RuntimeError: infinite loop in LLL
sage: L = X.LLL(delta=1, algorithm='NTL:LLL')
sage: L[-1]
(-100, -3, -1, 13, -1, -4, 2, 3, 4, 5, -1)
```

Note: See `ntl.mat_ZZ` or `fpylll.fplll.lll` for details on the used algorithms.

Albeit LLL is a deterministic algorithm, the output for different implementations and on CPUs (32-bit vs. 64-bit) may vary, while still being correct.

LLL_gram ( )

LLL reduction of the lattice whose gram matrix is `self`.

INPUT:

- `M` – gram matrix of a definite quadratic form

OUTPUT:

`U` - unimodular transformation matrix such that `U.T * M * U` is LLL-reduced.

ALGORITHM: Use PARI

EXAMPLES:

```
sage: M = Matrix(ZZ, 2, 2, [5,3,3,2]) ; M
[5 3]
[3 2]
sage: U = M.LLL_gram(); U
[-1  1]
[ 1 -2]
sage: U.transpose() * M * U
[1 0]
[0 1]
```

Semidefinite and indefinite forms no longer raise a `ValueError`:

```
sage: Matrix(ZZ,2,2,[2,6,6,3]).LLL_gram()
[-3 -1]
[ 1  0]
sage: Matrix(ZZ,2,2,[1,0,0,-1]).LLL_gram()
[ 0 -1]
[ 1  0]
```

antitranspose ( )

Returns the antitranspose of self, without changing self.

EXAMPLES:

```
sage: A = matrix(2,3,range(6))
sage: type(A)
<type 'sage.matrix.matrix_integer_dense.Matrix_integer_dense'>
sage: A.antitranspose()
[5 2]
[4 1]
[3 0]
sage: A
```

```
[0 1 2]
[3 4 5]

sage: A.subdivide(1,2); A
[0 1|2]
[---+-]
[3 4|5]
sage: A.antitranspose()
[5|2]
[-+-]
[4|1]
[3|0]
```

augment (*right*, *subdivide=False*)

Returns a new matrix formed by appending the matrix (or vector) `right` on the right side of `self`.

INPUT:

- `right` - a matrix, vector or free module element, whose dimensions are compatible with `self`.
- `subdivide` - default: `False` - request the resulting matrix to have a new subdivision, separating `self` from `right`.

OUTPUT:

A new matrix formed by appending `right` onto the right side of `self`. If `right` is a vector (or free module element) then in this context it is appropriate to consider it as a column vector. (The code first converts a vector to a 1-column matrix.)

EXAMPLES:

```
sage: A = matrix(ZZ, 4, 5, range(20))
sage: B = matrix(ZZ, 4, 3, range(12))
sage: A.augment(B)
[ 0  1  2  3  4  0  1  2]
[ 5  6  7  8  9  3  4  5]
[10 11 12 13 14  6  7  8]
[15 16 17 18 19  9 10 11]
```

A vector may be augmented to a matrix.

```
sage: A = matrix(ZZ, 3, 5, range(15))
sage: v = vector(ZZ, 3, range(3))
sage: A.augment(v)
[ 0  1  2  3  4  0]
[ 5  6  7  8  9  1]
[10 11 12 13 14  2]
```

The `subdivide` option will add a natural subdivision between `self` and `right`. For more details about how subdivisions are managed when augmenting, see *`sage.matrix.matrix1.Matrix.augment()`*.

```
sage: A = matrix(ZZ, 3, 5, range(15))
sage: B = matrix(ZZ, 3, 3, range(9))
sage: A.augment(B, subdivide=True)
[ 0  1  2  3  4| 0  1  2]
[ 5  6  7  8  9| 3  4  5]
[10 11 12 13 14| 6  7  8]
```

Errors are raised if the sizes are incompatible.

```
sage: A = matrix(ZZ, [[1, 2],[3, 4]])
sage: B = matrix(ZZ, [[10, 20], [30, 40], [50, 60]])
sage: A.augment(B)
Traceback (most recent call last):
...
TypeError: number of rows must be the same, not 2 != 3
```

charpoly (*var='x'*, *algorithm='generic'*)

INPUT:

- `var` - a variable name
- `algorithm` - 'generic' (default), 'flint' or 'linbox'

EXAMPLES:

```
sage: A = matrix(ZZ,6, range(36))
sage: f = A.charpoly(); f
x^6 - 105*x^5 - 630*x^4
sage: f(A) == 0
True
sage: g = A.charpoly(algorithm='flint')
sage: f == g
True
sage: n=20; A = Mat(ZZ,n)(range(n^2))
sage: A.charpoly()
x^20 - 3990*x^19 - 266000*x^18
sage: A.minpoly()
x^3 - 3990*x^2 - 266000*x
```

decomposition (***kwds*)

Returns the decomposition of the free module on which this matrix A acts from the right (i.e., the action is x goes to x A), along with whether this matrix acts irreducibly on each factor. The factors are guaranteed to be sorted in the same way as the corresponding factors of the characteristic polynomial, and are saturated as ZZ modules.

INPUT:

- `self` - a matrix over the integers
- `**kwds` - these are passed onto to the decomposition over QQ command.

EXAMPLES:

```
sage: t = ModularSymbols(11,sign=1).hecke_matrix(2)
sage: w = t.change_ring(ZZ)
sage: w.list()
[3, -1, 0, -2]
```

determinant (*algorithm='default'*, *proof=None*, *stabilize=2*)

Return the determinant of this matrix.

INPUT:

- `algorithm`
  - `'default'` – use `flint`
  - `'flint'` – let flint do the determinant
  - `'padic'` - uses a p-adic / multimodular algorithm that relies on code in IML and linbox

– `'linbox'` - calls linbox det (you *must* set proof=False to use this!)

– `'ntl'` - calls NTL's det function

– `'pari'` - uses PARI

•`proof` - bool or None; if None use proof.linear_algebra(); only relevant for the padic algorithm.

Note: It would be *VERY VERY* hard for det to fail even with proof=False.

•`stabilize` - if proof is False, require det to be the same for this many CRT primes in a row. Ignored if proof is True.

ALGORITHM: The p-adic algorithm works by first finding a random vector v, then solving A*x = v and taking the denominator d. This gives a divisor of the determinant. Then we compute $\det(A)/d$ using a multimodular algorithm and the Hadamard bound, skipping primes that divide d.

EXAMPLES:

```
sage: A = matrix(ZZ,8,8,[3..66])
sage: A.determinant()
0
```

```
sage: A = random_matrix(ZZ,20,20)
sage: D1 = A.determinant()
sage: A._clear_cache()
sage: D2 = A.determinant(algorithm='ntl')
sage: D1 == D2
True
```

We have a special-case algorithm for 4 x 4 determinants:

```
sage: A = matrix(ZZ,4,[1,2,3,4,4,3,2,1,0,5,0,1,9,1,2,3])
sage: A.determinant()
270
```

Next we try the Linbox det. Note that we must have proof=False.

```
sage: A = matrix(ZZ,5,[1,2,3,4,5,4,6,3,2,1,7,9,7,5,2,1,4,6,7,8,3,2,4,6,7])
sage: A.determinant(algorithm='linbox')
Traceback (most recent call last):
...
RuntimeError: you must pass the proof=False option to the determinant command␣
↪to use LinBox's det algorithm
sage: A.determinant(algorithm='linbox',proof=False)
-21
sage: A._clear_cache()
sage: A.determinant()
-21
```

A bigger example:

```
sage: A = random_matrix(ZZ,30)
sage: d = A.determinant()
sage: A._clear_cache()
sage: A.determinant(algorithm='linbox',proof=False) == d
True
```

echelon_form (*algorithm='default'*, *proof=None*, *include_zero_rows=True*, *transformation=False*, *D=None*)

Return the echelon form of this matrix over the integers, also known as the hermite normal form (HNF).

INPUT:

- `algorithm` – String. The algorithm to use. Valid options are:
  - `'default'` – Let Sage pick an algorithm (default). Up to 75 rows or columns with no transformation matrix, use pari with flag 0; otherwise, use flint.
  - `'flint'` - use flint
  - `'ntl'` - use NTL (only works for square matrices of full rank!)
  - `'padic'` - an asymptotically fast p-adic modular algorithm, If your matrix has large coefficients and is small, you may also want to try this.
  - `'pari'` - use PARI with flag 1
  - `'pari0'` - use PARI with flag 0
  - `'pari1'` - use PARI with flag 1
  - `'pari4'` - use PARI with flag 4 (use heuristic LLL)
- `proof` - (default: True); if proof=False certain determinants are computed using a randomized hybrid p-adic multimodular strategy until it stabilizes twice (instead of up to the Hadamard bound). It is *incredibly* unlikely that one would ever get an incorrect result with proof=False.
- `include_zero_rows` - (default: True) if False, don't include zero rows
- `transformation` - if given, also compute transformation matrix; only valid for flint and padic algorithm
- `D` - (default: None) if given and the algorithm is 'ntl', then D must be a multiple of the determinant and this function will use that fact.

OUTPUT:

The Hermite normal form (=echelon form over $\mathbf{Z}$) of self as an immutable matrix.

EXAMPLES:

```
sage: A = MatrixSpace(ZZ,2)([1,2,3,4])
sage: A.echelon_form()
[1 0]
[0 2]
sage: A = MatrixSpace(ZZ,5)(range(25))
sage: A.echelon_form()
[  5   0  -5 -10 -15]
[  0   1   2   3   4]
[  0   0   0   0   0]
[  0   0   0   0   0]
[  0   0   0   0   0]
```

Getting a transformation matrix in the nonsquare case:

```
sage: A = matrix(ZZ,5,3,[1..15])
sage: H, U = A.hermite_form(transformation=True, include_zero_rows=False)
sage: H
[1 2 3]
[0 3 6]
sage: U
```

```
[  0   0   0   4  -3]
[  0   0   0  13 -10]
sage: U*A == H
True
```

```
sage: m = matrix(ZZ,0,0,[])
sage: m.echelon_form()
[]
```

Note: If 'ntl' is chosen for a non square matrix this function raises a ValueError.

Special cases: 0 or 1 rows:

```
sage: a = matrix(ZZ, 1,2,[0,-1])
sage: a.hermite_form()
[0 1]
sage: a.pivots()
(1,)
sage: a = matrix(ZZ, 1,2,[0,0])
sage: a.hermite_form()
[0 0]
sage: a.pivots()
()
sage: a = matrix(ZZ,1,3); a
[0 0 0]
sage: a.echelon_form(include_zero_rows=False)
[]
sage: a.echelon_form(include_zero_rows=True)
[0 0 0]
```

Illustrate using various algorithms.:

```
sage: matrix(ZZ,3,[1..9]).hermite_form(algorithm='pari')
[1 2 3]
[0 3 6]
[0 0 0]
sage: matrix(ZZ,3,[1..9]).hermite_form(algorithm='pari0')
[1 2 3]
[0 3 6]
[0 0 0]
sage: matrix(ZZ,3,[1..9]).hermite_form(algorithm='pari4')
[1 2 3]
[0 3 6]
[0 0 0]
sage: matrix(ZZ,3,[1..9]).hermite_form(algorithm='padic')
[1 2 3]
[0 3 6]
[0 0 0]
sage: matrix(ZZ,3,[1..9]).hermite_form(algorithm='default')
[1 2 3]
[0 3 6]
[0 0 0]
```

The 'ntl' algorithm doesn't work on matrices that do not have full rank.:

```
sage: matrix(ZZ,3,[1..9]).hermite_form(algorithm='ntl')
Traceback (most recent call last):
...
ValueError: ntl only computes HNF for square matrices of full rank.
sage: matrix(ZZ,3,[0] +[2..9]).hermite_form(algorithm='ntl')
[1 0 0]
[0 1 0]
[0 0 3]
```

elementary_divisors (*algorithm='pari'*)

Return the elementary divisors of self, in order.

Warning: This is MUCH faster than the smith_form function.

The elementary divisors are the invariants of the finite abelian group that is the cokernel of *left* multiplication of this matrix. They are ordered in reverse by divisibility.

INPUT:

- `self` - matrix
- `algorithm` - (default: 'pari')
  - `'pari'` : works robustly, but is slower.
  - `'linbox'` - use linbox (currently off, broken)

OUTPUT: list of integers

Note: These are the invariants of the cokernel of *left* multiplication:

```
sage: M = Matrix([[3,0,1],[0,1,0]])
sage: M
[3 0 1]
[0 1 0]
sage: M.elementary_divisors()
[1, 1]
sage: M.transpose().elementary_divisors()
[1, 1, 0]
```

EXAMPLES:

```
sage: matrix(3, range(9)).elementary_divisors()
[1, 3, 0]
sage: matrix(3, range(9)).elementary_divisors(algorithm='pari')
[1, 3, 0]
sage: C = MatrixSpace(ZZ,4)([3,4,5,6,7,3,8,10,14,5,6,7,2,2,10,9])
sage: C.elementary_divisors()
[1, 1, 1, 687]
```

```
sage: M = matrix(ZZ, 3, [1,5,7, 3,6,9, 0,1,2])
sage: M.elementary_divisors()
[1, 1, 6]
```

This returns a copy, which is safe to change:

```
sage: edivs = M.elementary_divisors()
sage: edivs.pop()
6
sage: M.elementary_divisors()
[1, 1, 6]
```

See also:

smith_form()

frobenius (*flag=0*, *var='x'*)

Return the Frobenius form (rational canonical form) of this matrix.

INPUT:

- `flag` – 0 (default), 1 or 2 as follows:
  - `0` – (default) return the Frobenius form of this matrix.
  - `1` – return only the elementary divisor polynomials, as polynomials in var.
  - `2` – return a two-components vector [F,B] where F is the Frobenius form and B is the basis change so that $M = B^{-1}FB$.
- `var` – a string (default: 'x')

ALGORITHM: uses PARI's matfrobenius()

EXAMPLES:

```
sage: A = MatrixSpace(ZZ, 3)(range(9))
sage: A.frobenius(0)
[ 0  0  0]
[ 1  0 18]
[ 0  1 12]
sage: A.frobenius(1)
[x^3 - 12*x^2 - 18*x]
sage: A.frobenius(1, var='y')
[y^3 - 12*y^2 - 18*y]
sage: F, B = A.frobenius(2)
sage: A == B^(-1)*F*B
True
sage: a=matrix([])
sage: a.frobenius(2)
([], [])
sage: a.frobenius(0)
[]
sage: a.frobenius(1)
[]
sage: B = random_matrix(ZZ,2,3)
sage: B.frobenius()
Traceback (most recent call last):
...
ArithmeticError: frobenius matrix of non-square matrix not defined.
```

AUTHORS:

- Martin Albrect (2006-04-02)

TODO: - move this to work for more general matrices than just over Z. This will require fixing how PARI polynomials are coerced to Sage polynomials.

gcd ( )

Return the gcd of all entries of self; very fast.

EXAMPLES:

```
sage: a = matrix(ZZ,2, [6,15,-6,150])
sage: a.gcd()
3
```

height ( )

Return the height of this matrix, i.e., the max absolute value of the entries of the matrix.

OUTPUT: A nonnegative integer.

EXAMPLES:

```
sage: a = Mat(ZZ,3)(range(9))
sage: a.height()
8
sage: a = Mat(ZZ,2,3)([-17,3,-389,15,-1,0]); a
[ -17    3 -389]
[  15   -1    0]
sage: a.height()
389
```

hermite_form (*algorithm='default'*, *proof=None*, *include_zero_rows=True*, *transformation=False*, *D=None*)

Return the echelon form of this matrix over the integers, also known as the hermite normal form (HNF).

INPUT:

- `algorithm` – String. The algorithm to use. Valid options are:
  - `'default'` – Let Sage pick an algorithm (default). Up to 75 rows or columns with no transformation matrix, use pari with flag 0; otherwise, use flint.
  - `'flint'` - use flint
  - `'ntl'` - use NTL (only works for square matrices of full rank!)
  - `'padic'` - an asymptotically fast p-adic modular algorithm, If your matrix has large coefficients and is small, you may also want to try this.
  - `'pari'` - use PARI with flag 1
  - `'pari0'` - use PARI with flag 0
  - `'pari1'` - use PARI with flag 1
  - `'pari4'` - use PARI with flag 4 (use heuristic LLL)
- `proof` - (default: True); if proof=False certain determinants are computed using a randomized hybrid p-adic multimodular strategy until it stabilizes twice (instead of up to the Hadamard bound). It is *incredibly* unlikely that one would ever get an incorrect result with proof=False.
- `include_zero_rows` - (default: True) if False, don't include zero rows
- `transformation` - if given, also compute transformation matrix; only valid for flint and padic algorithm
- `D` - (default: None) if given and the algorithm is 'ntl', then D must be a multiple of the determinant and this function will use that fact.

OUTPUT:

The Hermite normal form (=echelon form over **Z**) of self as an immutable matrix.

EXAMPLES:

```
sage: A = MatrixSpace(ZZ,2)([1,2,3,4])
sage: A.echelon_form()
[1 0]
[0 2]
sage: A = MatrixSpace(ZZ,5)(range(25))
sage: A.echelon_form()
[  5   0  -5 -10 -15]
[  0   1   2   3   4]
[  0   0   0   0   0]
[  0   0   0   0   0]
[  0   0   0   0   0]
```

Getting a transformation matrix in the nonsquare case:

```
sage: A = matrix(ZZ,5,3,[1..15])
sage: H, U = A.hermite_form(transformation=True, include_zero_rows=False)
sage: H
[1 2 3]
[0 3 6]
sage: U
[  0   0   0   4  -3]
[  0   0   0  13 -10]
sage: U*A == H
True
```

```
sage: m = matrix(ZZ,0,0,[])
sage: m.echelon_form()
[]
```

Note: If 'ntl' is chosen for a non square matrix this function raises a ValueError.

Special cases: 0 or 1 rows:

```
sage: a = matrix(ZZ, 1,2,[0,-1])
sage: a.hermite_form()
[0 1]
sage: a.pivots()
(1,)
sage: a = matrix(ZZ, 1,2,[0,0])
sage: a.hermite_form()
[0 0]
sage: a.pivots()
()
sage: a = matrix(ZZ,1,3); a
[0 0 0]
sage: a.echelon_form(include_zero_rows=False)
[]
sage: a.echelon_form(include_zero_rows=True)
[0 0 0]
```

Illustrate using various algorithms.:

```
sage: matrix(ZZ,3,[1..9]).hermite_form(algorithm='pari')
[1 2 3]
[0 3 6]
[0 0 0]
sage: matrix(ZZ,3,[1..9]).hermite_form(algorithm='pari0')
[1 2 3]
[0 3 6]
[0 0 0]
sage: matrix(ZZ,3,[1..9]).hermite_form(algorithm='pari4')
[1 2 3]
[0 3 6]
[0 0 0]
sage: matrix(ZZ,3,[1..9]).hermite_form(algorithm='padic')
[1 2 3]
[0 3 6]
[0 0 0]
sage: matrix(ZZ,3,[1..9]).hermite_form(algorithm='default')
[1 2 3]
[0 3 6]
[0 0 0]
```

The 'ntl' algorithm doesn't work on matrices that do not have full rank.:

```
sage: matrix(ZZ,3,[1..9]).hermite_form(algorithm='ntl')
Traceback (most recent call last):
...
ValueError: ntl only computes HNF for square matrices of full rank.
sage: matrix(ZZ,3,[0] +[2..9]).hermite_form(algorithm='ntl')
[1 0 0]
[0 1 0]
[0 0 3]
```

index_in_saturation (*proof=None*)

Return the index of self in its saturation.

INPUT:

- `proof` - (default: use proof.linear_algebra()); if False, the determinant calculations are done with proof=False.

OUTPUT:

- `positive integer` - the index of the row span of this matrix in its saturation

ALGORITHM: Use Hermite normal form twice to find an invertible matrix whose inverse transforms a matrix with the same row span as self to its saturation, then compute the determinant of that matrix.

EXAMPLES:

```
sage: A = matrix(ZZ, 2,3, [1..6]); A
[1 2 3]
[4 5 6]
sage: A.index_in_saturation()
3
sage: A.saturation()
[1 2 3]
[1 1 1]
```

insert_row (*index*, *row*)

Create a new matrix from self with.

INPUT:

- `index` - integer
- `row` - a vector

EXAMPLES:

```
sage: X = matrix(ZZ,3,range(9)); X
[0 1 2]
[3 4 5]
[6 7 8]
sage: X.insert_row(1, [1,5,-10])
[  0   1   2]
[  1   5 -10]
[  3   4   5]
[  6   7   8]
sage: X.insert_row(0, [1,5,-10])
[  1   5 -10]
[  0   1   2]
[  3   4   5]
[  6   7   8]
sage: X.insert_row(3, [1,5,-10])
[  0   1   2]
[  3   4   5]
[  6   7   8]
[  1   5 -10]
```

integer_valued_polynomials_generators ( )

Determine the generators of the ring of integer valued polynomials on this matrix.

OUTPUT:

A pair `(mu_B,P)` where `P` is a list of polynomials in $\mathbf{Q}[X]$ such that

$$\{f \in \mathbf{Q}[X] \mid f(B) \in M_n(\mathbf{Z})\} = \mu_B \mathbf{Q}[X] + \sum_{g \in P} g\mathbf{Z}[X]$$

where B is this matrix.

EXAMPLES:

```
sage: B = matrix(ZZ, [[1, 0, 1], [1, -2, -1], [10, 0, 0]])
sage: B.integer_valued_polynomials_generators()
(x^3 + x^2 - 12*x - 20, [1, 1/4*x^2 + 3/4*x + 1/2])
```

See also:

compute_J_ideal, *integer_valued_polynomials_generators()*

is_LLL_reduced (*delta=None*, *eta=None*)

Return `True` if this lattice is (δ, η)-LLL reduced. See `self.LLL` for a definition of LLL reduction.

INPUT:

- `delta` – (default: 0.99) parameter δ as described above
- `eta` – (default: 0.501) parameter η as described above

EXAMPLES:

```
sage: A = random_matrix(ZZ, 10, 10)
sage: L = A.LLL()
sage: A.is_LLL_reduced()
False
sage: L.is_LLL_reduced()
True
```

is_one ( )

Tests whether self is the identity matrix.

EXAMPLES:

```
sage: matrix(2, [1,0,0,1]).is_one()
True
sage: matrix(2, [1,1,0,1]).is_one()
False
sage: matrix(2, 3, [1,0,0,0,1,0]).is_one()
False
```

minpoly (*var='x'*, *algorithm='linbox'*)

INPUT:

- `var` - a variable name
- `algorithm` - 'linbox' (default) 'generic'

Note: Linbox charpoly disabled on 64-bit machines, since it hangs in many cases.

EXAMPLES:

```
sage: A = matrix(ZZ,6, range(36))
sage: A.minpoly()
x^3 - 105*x^2 - 630*x
sage: n=6; A = Mat(ZZ,n)([k^2 for k in range(n^2)])
sage: A.minpoly()
x^4 - 2695*x^3 - 257964*x^2 + 1693440*x
```

null_ideal (*b=0*)

Return the (b)-ideal of this matrix.

Let B be a $n \times n$ matrix. The *null ideal* modulo b, or (b)-ideal, is

$$N_{(b)}(B) = \{f \in \mathbf{Z}[X] \mid f(B) \in M_n(b\mathbf{Z})\}.$$

INPUT:

- `b` – an element of $\mathbf{Z}$ (default: 0)

OUTPUT:

An ideal in $\mathbf{Z}[X]$.

EXAMPLES:

```
sage: B = matrix(ZZ, [[1, 0, 1], [1, -2, -1], [10, 0, 0]])
sage: B.null_ideal()
Principal ideal (x^3 + x^2 - 12*x - 20) of
    Univariate Polynomial Ring in x over Integer Ring
sage: B.null_ideal(8)
```

```
Ideal (8, x^3 + x^2 - 12*x - 20, 2*x^2 + 6*x + 4) of
    Univariate Polynomial Ring in x over Integer Ring
sage: B.null_ideal(6)
Ideal (6, 2*x^3 + 2*x^2 - 24*x - 40, 3*x^2 + 3*x) of
    Univariate Polynomial Ring in x over Integer Ring
```

See also:

compute_J_ideal, *null_ideal()*

p_minimal_polynomials (*p*, *s_max=None*)

Compute (p^s)-minimal polynomials ν_s of this matrix.

For $s \geq 0$, a (p^s)-minimal polynomial of a matrix B is a monic polynomial $f \in \mathbf{Z}[X]$ of minimal degree such that all entries of $f(B)$ are divisible by p^s.

Compute a finite subset $\mathcal{S}$ of the positive integers and (p^s)-minimal polynomials ν_s for $s \in \mathcal{S}$.

For $0 < t \leq \max \mathcal{S}$, a (p^t)-minimal polynomial is given by ν_s where $s = \min\{r \in \mathcal{S} \mid r \geq t\}$. For $t > \max \mathcal{S}$, the minimal polynomial of B is also a (p^t)-minimal polynomial.

INPUT:

- `p` – a prime in $\mathbf{Z}$
- `s_max` – a positive integer (default: `None`); if set, only (p^s)-minimal polynomials for `s <= s_max` are computed (see below for details)

OUTPUT:

A dictionary. Keys are the finite set $\mathcal{S}$, the values are the associated (p^s)-minimal polynomials ν_s, $s \in \mathcal{S}$.

Setting `s_max` only affects the output if `s_max` is at most $\max \mathcal{S}$ where $\mathcal{S}$ denotes the full set. In that case, only those ν_s with `s <= s_max` are returned where `s_max` is always included even if it is not included in the full set $\mathcal{S}$.

EXAMPLES:

```
sage: B = matrix(ZZ, [[1, 0, 1], [1, -2, -1], [10, 0, 0]])
sage: B.p_minimal_polynomials(2)
{2: x^2 + 3*x + 2}
```

See also:

compute_J_ideal, *p_minimal_polynomials()*

pivots ( )

Return the pivot column positions of this matrix.

OUTPUT: a tuple of Python integers: the position of the first nonzero entry in each row of the echelon form.

EXAMPLES:

```
sage: n = 3; A = matrix(ZZ,n,range(n^2)); A
[0 1 2]
[3 4 5]
[6 7 8]
sage: A.pivots()
(0, 1)
sage: A.echelon_form()
[ 3  0 -3]
```

```
[ 0  1  2]
[ 0  0  0]
```

prod_of_row_sums (*cols*)

Return the product of the sums of the entries in the submatrix of `self` with given columns.

INPUT:

•`cols` – a list (or set) of integers representing columns of `self`

OUTPUT: an integer

EXAMPLES:

```
sage: a = matrix(ZZ,2,3,[1..6]); a
[1 2 3]
[4 5 6]
sage: a.prod_of_row_sums([0,2])
40
sage: (1+3)*(4+6)
40
sage: a.prod_of_row_sums(set([0,2]))
40
```

randomize (*density=1*, *x=None*, *y=None*, *distribution=None*, *nonzero=False*)

Randomize `density` proportion of the entries of this matrix, leaving the rest unchanged.

The parameters are the same as the ones for the integer ring's `random_element` function.

If `x` and `y` are given, randomized entries of this matrix have to be between `x` and `y` and have density 1.

INPUT:

•`self` - a mutable matrix over ZZ

•`density` - a float between 0 and 1

•`x, y` - if not `None`, these are passed to the `ZZ.random_element` function as the upper and lower endpoints in the uniform distribution

•`distribution` - would also be passed into `ZZ.random_element` if given

•`nonzero` - bool (default: `False`); whether the new entries are guaranteed to be zero

OUTPUT:

•None, the matrix is modified in-place

EXAMPLES:

```
sage: A = matrix(ZZ, 2,3, [1..6]); A
[1 2 3]
[4 5 6]
sage: A.randomize()
sage: A
[-8  2  0]
[ 0  1 -1]
sage: A.randomize(x=-30,y=30)
sage: A
[  5 -19  24]
[ 24  23  -9]
```

rank (*algorithm='modp'*)

Return the rank of this matrix.

INPUT:

- `algorithm` – either `'modp'` (default) or `'flint'` or `'linbox'`

OUTPUT:

- a nonnegative integer – the rank

Note: The rank is cached.

ALGORITHM:

If set to `'modp'`, first check if the matrix has maximum possible rank by working modulo one random prime. If not, call LinBox's rank function.

EXAMPLES:

```
sage: a = matrix(ZZ,2,3,[1..6]); a
[1 2 3]
[4 5 6]
sage: a.rank()
2
sage: a = matrix(ZZ,3,3,[1..9]); a
[1 2 3]
[4 5 6]
[7 8 9]
sage: a.rank()
2
```

Here is a bigger example - the rank is of course still 2:

```
sage: a = matrix(ZZ,100,[1..100^2]); a.rank()
2
```

rational_reconstruction (*N*)

Use rational reconstruction to lift self to a matrix over the rational numbers (if possible), where we view self as a matrix modulo N.

INPUT:

- `N` - an integer

OUTPUT:

- `matrix` - over QQ or raise a ValueError

EXAMPLES: We create a random 4x4 matrix over ZZ.

```
sage: A = matrix(ZZ, 4, [4, -4, 7, 1, -1, 1, -1, -12, -1, -1, 1, -1, -3, 1,␣
↪5, -1])
```

There isn't a unique rational reconstruction of it:

```
sage: A.rational_reconstruction(11)
Traceback (most recent call last):
...
ValueError: rational reconstruction does not exist
```

We throw in a denominator and reduce the matrix modulo 389 - it does rationally reconstruct.

```
sage: B = (A/3 % 389).change_ring(ZZ)
sage: B.rational_reconstruction(389) == A/3
True
```

saturation (*p=0*, *proof=None*, *max_dets=5*)

Return a saturation matrix of self, which is a matrix whose rows span the saturation of the row span of self. This is not unique.

The saturation of a $\mathbf{Z}$ module M embedded in $\mathbf{Z}^n$ is the a module S that contains M with finite index such that $\mathbf{Z}^n/S$ is torsion free. This function takes the row span M of self, and finds another matrix of full rank with row span the saturation of M.

INPUT:

- `p` - (default: 0); if nonzero given, saturate only at the prime p, i.e., return a matrix whose row span is a $\mathbf{Z}$-module S that contains self and such that the index of S in its saturation is coprime to p. If p is None, return full saturation of self.
- `proof` - (default: use proof.linear_algebra()); if False, the determinant calculations are done with proof=False.
- `max_dets` - (default: 5); technical parameter - max number of determinant to compute when bounding prime divisor of self in its saturation.

OUTPUT:

- `matrix` - a matrix over ZZ

Note: The result is *not* cached.

ALGORITHM: 1. Replace input by a matrix of full rank got from a subset of the rows. 2. Divide out any common factors from rows. 3. Check max_dets random dets of submatrices to see if their GCD (with p) is 1 - if so matrix is saturated and we're done. 4. Finally, use that if A is a matrix of full rank, then $hnf(transpose(A))^{-1} * A$ is a saturation of A.

EXAMPLES:

```
sage: A = matrix(ZZ, 3, 5, [-51, -1509, -71, -109, -593, -19, -341, 4, 86,␣
↪98, 0, -246, -11, 65, 217])
sage: A.echelon_form()
[      1       5    2262   20364   56576]
[      0       6   35653  320873  891313]
[      0       0   42993  386937 1074825]
sage: S = A.saturation(); S
[  -51 -1509   -71  -109  -593]
[  -19  -341     4    86    98]
[   35   994    43    51   347]
```

Notice that the saturation spans a different module than A.

```
sage: S.echelon_form()
[ 1  2  0  8 32]
[ 0  3  0 -2 -6]
[ 0  0  1  9 25]
sage: V = A.row_space(); W = S.row_space()
sage: V.is_submodule(W)
True
```

```
sage: V.index_in(W)
85986
sage: V.index_in_saturation()
85986
```

We illustrate each option:

```
sage: S = A.saturation(p=2)
sage: S = A.saturation(proof=False)
sage: S = A.saturation(max_dets=2)
```

smith_form ( )

Returns matrices S, U, and V such that S = U*self*V, and S is in Smith normal form. Thus S is diagonal with diagonal entries the ordered elementary divisors of S.

> **Warning:** The elementary_divisors function, which returns the diagonal entries of S, is VASTLY faster than this function.

The elementary divisors are the invariants of the finite abelian group that is the cokernel of this matrix. They are ordered in reverse by divisibility.

EXAMPLES:

```
sage: A = MatrixSpace(IntegerRing(), 3)(range(9))
sage: D, U, V = A.smith_form()
sage: D
[1 0 0]
[0 3 0]
[0 0 0]
sage: U
[ 0  1  0]
[ 0 -1  1]
[-1  2 -1]
sage: V
[-1  4  1]
[ 1 -3 -2]
[ 0  0  1]
sage: U*A*V
[1 0 0]
[0 3 0]
[0 0 0]
```

It also makes sense for nonsquare matrices:

```
sage: A = Matrix(ZZ,3,2,range(6))
sage: D, U, V = A.smith_form()
sage: D
[1 0]
[0 2]
[0 0]
sage: U
[ 0  1  0]
[ 0 -1  1]
[-1  2 -1]
sage: V
[-1  3]
```

```
[ 1 -2]
sage: U * A * V
[1 0]
[0 2]
[0 0]
```

Empty matrices are handled sensibly (see trac ticket #3068):

```
sage: m = MatrixSpace(ZZ, 2,0)(0); d,u,v = m.smith_form(); u*m*v == d
True
sage: m = MatrixSpace(ZZ, 0,2)(0); d,u,v = m.smith_form(); u*m*v == d
True
sage: m = MatrixSpace(ZZ, 0,0)(0); d,u,v = m.smith_form(); u*m*v == d
True
```

See also:

elementary_divisors()

symplectic_form ( )

Find a symplectic basis for self if self is an anti-symmetric, alternating matrix.

Returns a pair (F, C) such that the rows of C form a symplectic basis for self and F = C * self * C.transpose().

Raises a ValueError if self is not anti-symmetric, or self is not alternating.

Anti-symmetric means that $M = -M^t$. Alternating means that the diagonal of M is identically zero.

A symplectic basis is a basis of the form $e_1, \ldots, e_j, f_1, \ldots f_j, z_1, \ldots, z_k$ such that

- $z_i M v^t$ = 0 for all vectors v
- $e_i M {e_j}^t = 0$ for all i, j
- $f_i M {f_j}^t = 0$ for all i, j
- $e_i M {f_i}^t = 1$ for all i
- $e_i M {f_j}^t = 0$ **for all** i **not equal** j.

The ordering for the factors $d_i | d_{i+1}$ and for the placement of zeroes was chosen to agree with the output of `smith_form`.

See the example for a pictorial description of such a basis.

EXAMPLES:

```
sage: E = matrix(ZZ, 5, 5, [0, 14, 0, -8, -2, -14, 0, -3, -11, 4, 0, 3, 0, 0,␣
↪0, 8, 11, 0, 0, 8, 2, -4, 0, -8, 0]); E
[  0  14   0  -8  -2]
[-14   0  -3 -11   4]
[  0   3   0   0   0]
[  8  11   0   0   8]
[  2  -4   0  -8   0]
sage: F, C = E.symplectic_form()
sage: F
[ 0  0  1  0  0]
[ 0  0  0  2  0]
[-1  0  0  0  0]
[ 0 -2  0  0  0]
[ 0  0  0  0  0]
sage: F == C * E * C.transpose()
```

```
True
sage: E.smith_form()[0]
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 2 0 0]
[0 0 0 2 0]
[0 0 0 0 0]
```

transpose ()

Returns the transpose of self, without changing self.

EXAMPLES:

We create a matrix, compute its transpose, and note that the original matrix is not changed.

```
sage: A = matrix(ZZ, 2, 3, range(6))
sage: type(A)
<type 'sage.matrix.matrix_integer_dense.Matrix_integer_dense'>
sage: B = A.transpose()
sage: print(B)
[0 3]
[1 4]
[2 5]
sage: print(A)
[0 1 2]
[3 4 5]
```

`.T` is a convenient shortcut for the transpose:

```
sage: A.T
[0 3]
[1 4]
[2 5]
```

```
sage: A.subdivide(None, 1); A
[0|1 2]
[3|4 5]
sage: A.transpose()
[0 3]
[---]
[1 4]
[2 5]
```

CHAPTER

EIGHTEEN

DENSE MATRICES OVER THE RATIONAL FIELD

EXAMPLES:

We create a 3x3 matrix with rational entries and do some operations with it.

```
sage: a = matrix(QQ, 3,3, [1,2/3, -4/5, 1,1,1, 8,2, -3/19]); a
[    1   2/3  -4/5]
[    1     1     1]
[    8     2 -3/19]
sage: a.det()
2303/285
sage: a.charpoly()
x^3 - 35/19*x^2 + 1259/285*x - 2303/285
sage: b = a^(-1); b
[ -615/2303  -426/2303   418/2303]
[ 2325/2303  1779/2303  -513/2303]
[-1710/2303   950/2303    95/2303]
sage: b.det()
285/2303
sage: a == b
False
sage: a < b
False
sage: b < a
True
sage: a > b
True
sage: a*b
[1 0 0]
[0 1 0]
[0 0 1]
```

class `sage.matrix.matrix_rational_dense.`**`MatrixWindow`**

Bases: `object`

class `sage.matrix.matrix_rational_dense.`**`Matrix_rational_dense`**

Bases: *`sage.matrix.matrix_dense.Matrix_dense`*

`LLL` (**args*, ***kwargs*)

Return an LLL reduced or approximated LLL reduced lattice for `self` interpreted as a lattice.

For details on input parameters, see *`sage.matrix.matrix_integer_dense.Matrix_integer_dense.LLL()`*.

EXAMPLES:

```
sage: A = Matrix(QQ, 3, 3, [1/n for n in range(1, 10)])
sage: A.LLL()
[ 1/28 -1/40 -1/18]
[ 1/28 -1/40  1/18]
[    0 -3/40     0]
```

antitranspose ( )

Returns the antitranspose of self, without changing self.

EXAMPLES:

```
sage: A = matrix(QQ,2,3,range(6))
sage: type(A)
<type 'sage.matrix.matrix_rational_dense.Matrix_rational_dense'>
sage: A.antitranspose()
[5 2]
[4 1]
[3 0]
sage: A
[0 1 2]
[3 4 5]

sage: A.subdivide(1,2); A
[0 1|2]
[---+-]
[3 4|5]
sage: A.antitranspose()
[5|2]
[-+-]
[4|1]
[3|0]
```

change_ring (*R*)

Create the matrix over R with entries the entries of self coerced into R.

EXAMPLES:

```
sage: a = matrix(QQ,2,[1/2,-1,2,3])
sage: a.change_ring(GF(3))
[2 2]
[2 0]
sage: a.change_ring(ZZ)
Traceback (most recent call last):
...
TypeError: matrix has denominators so can't change to ZZ.
sage: b = a.change_ring(QQ['x']); b
[1/2  -1]
[  2   3]
sage: b.parent()
Full MatrixSpace of 2 by 2 dense matrices over Univariate Polynomial Ring in␣
↪x over Rational Field
```

charpoly (*var='x'*, *algorithm='linbox'*)

Return the characteristic polynomial of this matrix.

INPUT:

- `var` - 'x' (string)
- `algorithm` - 'linbox' (default) or 'generic'

OUTPUT: a polynomial over the rational numbers.

EXAMPLES:

```
sage: a = matrix(QQ, 3, [4/3, 2/5, 1/5, 4, -3/2, 0, 0, -2/3, 3/4])
sage: f = a.charpoly(); f
x^3 - 7/12*x^2 - 149/40*x + 97/30
sage: f(a)
[0 0 0]
[0 0 0]
[0 0 0]
```

column (*i*, *from_list=False*)

Return the i-th column of this matrix as a dense vector.

INPUT:

- `i` - integer
- `from_list` - ignored

EXAMPLES:

```
sage: matrix(QQ,2,[1/5,-2/3,3/4,4/9]).column(1)
(-2/3, 4/9)
sage: matrix(QQ,2,[1/5,-2/3,3/4,4/9]).column(1,from_list=True)
(-2/3, 4/9)
sage: matrix(QQ,2,[1/5,-2/3,3/4,4/9]).column(-1)
(-2/3, 4/9)
sage: matrix(QQ,2,[1/5,-2/3,3/4,4/9]).column(-2)
(1/5, 3/4)
```

decomposition (*is_diagonalizable=False*, *dual=False*, *algorithm='default'*, *height_guess=None*, *proof=None*)

Returns the decomposition of the free module on which this matrix A acts from the right (i.e., the action is x goes to x A), along with whether this matrix acts irreducibly on each factor. The factors are guaranteed to be sorted in the same way as the corresponding factors of the characteristic polynomial.

Let A be the matrix acting from the on the vector space V of column vectors. Assume that A is square. This function computes maximal subspaces W_1, ..., W_n corresponding to Galois conjugacy classes of eigenvalues of A. More precisely, let f(X) be the characteristic polynomial of A. This function computes the subspace $W_i = ker(g_(A)^n)$, where g_i(X) is an irreducible factor of f(X) and g_i(X) exactly divides f(X). If the optional parameter is_diagonalizable is True, then we let W_i = ker(g(A)), since then we know that ker(g(A)) = $ker(g(A)^n)$.

If dual is True, also returns the corresponding decomposition of V under the action of the transpose of A. The factors are guaranteed to correspond.

INPUT:

- `is_diagonalizable` - ignored
- `dual` - whether to also return decompositions for the dual
- `algorithm`
  - 'default': use default algorithm for computing Echelon forms
  - 'multimodular': much better if the answers factors have small height
- `height_guess` - positive integer; only used by the multimodular algorithm

- `proof` - bool or None (default: None, see proof.linear_algebra or sage.structure.proof); only used by the multimodular algorithm. Note that the Sage global default is proof=True.

Note: IMPORTANT: If you expect that the subspaces in the answer are spanned by vectors with small height coordinates, use algorithm='multimodular' and height_guess=1; this is potentially much faster than the default. If you know for a fact the answer will be very small, use algorithm='multimodular', height_guess=bound on height, proof=False.

You can get very very fast decomposition with proof=False.

EXAMPLES:

```
sage: a = matrix(QQ,3,[1..9])
sage: a.decomposition()
[
(Vector space of degree 3 and dimension 1 over Rational Field
Basis matrix:
[ 1 -2  1], True),
(Vector space of degree 3 and dimension 2 over Rational Field
Basis matrix:
[ 1  0 -1]
[ 0  1  2], True)
]
```

denominator ( )

Return the denominator of this matrix.

OUTPUT: a Sage Integer

EXAMPLES:

```
sage: b = matrix(QQ,2,range(6)); b[0,0]=-5007/293; b
[-5007/293         1         2]
[        3         4         5]
sage: b.denominator()
293
```

determinant (*algorithm='default'*, *proof=None*)

Return the determinant of this matrix.

INPUT:

- `proof` - bool or None; if None use proof.linear_algebra(); only relevant for the padic algorithm.
- `algorithm`:

  "default" – use PARI for up to 7 rows, then use integer

  "pari" – use PARI

  "integer" – clear denominators and call det on integer matrix

Note: It would be *VERY VERY* hard for det to fail even with proof=False.

ALGORITHM: Clear denominators and call the integer determinant function.

EXAMPLES:

```
sage: m = matrix(QQ,3,[1,2/3,4/5, 2,2,2, 5,3,2/5])
sage: m.determinant()
-34/15
sage: m.charpoly()
x^3 - 17/5*x^2 - 122/15*x + 34/15
```

echelon_form (*algorithm='default'*, *height_guess=None*, *proof=None*, ***kwds*)

INPUT:

- `algorithm`
  - 'default' (default): use heuristic choice
  - 'padic': an algorithm based on the IML p-adic solver.
  - 'multimodular': uses a multimodular algorithm the uses linbox modulo many primes.
  - 'classical': just clear each column using Gauss elimination
- `height_guess`, `**kwds` - all passed to the multimodular algorithm; ignored by the p-adic algorithm.
- `proof` - bool or None (default: None, see proof.linear_algebra or sage.structure.proof). Passed to the multimodular algorithm. Note that the Sage global default is proof=True.

OUTPUT: the reduced row echelon form of self.

EXAMPLES:

```
sage: a = matrix(QQ, 4, range(16)); a[0,0] = 1/19; a[0,1] = 1/5; a
[1/19  1/5    2    3]
[   4    5    6    7]
[   8    9   10   11]
[  12   13   14   15]
sage: a.echelon_form()
[      1       0       0 -76/157]
[      0       1       0  -5/157]
[      0       0       1 238/157]
[      0       0       0       0]
sage: a.echelon_form(algorithm='multimodular')
[      1       0       0 -76/157]
[      0       1       0  -5/157]
[      0       0       1 238/157]
[      0       0       0       0]
```

The result is an immutable matrix, so if you want to modify the result then you need to make a copy. This checks that trac ticket #10543 is fixed.

```
sage: A = matrix(QQ, 2, range(6))
sage: E = A.echelon_form()
sage: E.is_mutable()
False
sage: F = copy(E)
sage: F[0,0] = 50
sage: F
[50  0 -1]
[ 0  1  2]
```

echelonize (*algorithm='default'*, *height_guess=None*, *proof=None*, ***kwds*)

Transform the matrix `self` into reduced row echelon form in place.

INPUT:

- `algorithm`:
  - `'default'` (default): use heuristic choice
  - `'padic'` : an algorithm based on the IML p-adic solver.
  - `'multimodular'` : uses a multimodular algorithm the uses linbox modulo many primes.
  - `'classical'` : just clear each column using Gauss elimination
- `height_guess`, `**kwds` - all passed to the multimodular algorithm; ignored by the p-adic algorithm.
- `proof` - bool or None (default: None, see proof.linear_algebra or sage.structure.proof). Passed to the multimodular algorithm. Note that the Sage global default is `proof=True`.

OUTPUT:

Nothing. The matrix `self` is transformed into reduced row echelon form in place.

EXAMPLES:

```
sage: a = matrix(QQ, 4, range(16)); a[0,0] = 1/19; a[0,1] = 1/5; a
[1/19  1/5    2    3]
[   4    5    6    7]
[   8    9   10   11]
[  12   13   14   15]
sage: a.echelonize(); a
[      1       0       0 -76/157]
[      0       1       0  -5/157]
[      0       0       1 238/157]
[      0       0       0       0]
```

```
sage: a = matrix(QQ, 4, range(16)); a[0,0] = 1/19; a[0,1] = 1/5
sage: a.echelonize(algorithm='multimodular'); a
[      1       0       0 -76/157]
[      0       1       0  -5/157]
[      0       0       1 238/157]
[      0       0       0       0]
```

height ()

Return the height of this matrix, which is the maximum of the absolute values of all numerators and denominators of entries in this matrix.

OUTPUT: an Integer

EXAMPLES:

```
sage: b = matrix(QQ,2,range(6)); b[0,0]=-5007/293; b
[-5007/293         1         2]
[        3         4         5]
sage: b.height()
5007
```

minpoly (*var='x'*, *algorithm='linbox'*)

Return the minimal polynomial of this matrix.

INPUT:

- `var` - 'x' (string)
- `algorithm` - 'linbox' (default) or 'generic'

OUTPUT: a polynomial over the rational numbers.

EXAMPLES:

```
sage: a = matrix(QQ, 3, [4/3, 2/5, 1/5, 4, -3/2, 0, 0, -2/3, 3/4])
sage: f = a.minpoly(); f
x^3 - 7/12*x^2 - 149/40*x + 97/30
sage: a = Mat(ZZ,4)(range(16))
sage: f = a.minpoly(); f.factor()
x * (x^2 - 30*x - 80)
sage: f(a) == 0
True
```

```
sage: a = matrix(QQ, 4, [1..4^2])
sage: factor(a.minpoly())
x * (x^2 - 34*x - 80)
sage: factor(a.minpoly('y'))
y * (y^2 - 34*y - 80)
sage: factor(a.charpoly())
x^2 * (x^2 - 34*x - 80)
sage: b = matrix(QQ, 4, [-1, 2, 2, 0, 0, 4, 2, 2, 0, 0, -1, -2, 0, -4, 0, 4])
sage: a = matrix(QQ, 4, [1, 1, 0,0, 0,1,0,0, 0,0,5,0, 0,0,0,5])
sage: c = b^(-1)*a*b
sage: factor(c.minpoly())
(x - 5) * (x - 1)^2
sage: factor(c.charpoly())
(x - 5)^2 * (x - 1)^2
```

prod_of_row_sums (*cols*)

randomize (*density=1*, *num_bound=2*, *den_bound=2*, *distribution=None*, *nonzero=False*)

Randomize `density` proportion of the entries of this matrix, leaving the rest unchanged.

If `x` and `y` are given, randomized entries of this matrix have numerators and denominators bounded by `x` and `y` and have density 1.

INPUT:

- `density` - number between 0 and 1 (default: 1)
- `num_bound` - numerator bound (default: 2)
- `den_bound` - denominator bound (default: 2)
- `distribution` - `None` or '1/n' (default: `None`); if '1/n' then `num_bound` , `den_bound` are ignored and numbers are chosen using the GMP function `mpq_randomize_entry_recip_uniform`
- `nonzero` - Bool (default: `False`); whether the new entries are forced to be non-zero

OUTPUT:

- None, the matrix is modified in-space

EXAMPLES:

```
sage: a = matrix(QQ,2,4); a.randomize(); a
[ 0 -1  2 -2]
[ 1 -1  2  1]
```

```
sage: a = matrix(QQ,2,4); a.randomize(density=0.5); a
[ -1  -2   0   0]
[  0   0 1/2   0]
sage: a = matrix(QQ,2,4); a.randomize(num_bound=100, den_bound=100); a
[ 14/27  21/25  43/42 -48/67]
[-19/55  64/67 -11/51     76]
sage: a = matrix(QQ,2,4); a.randomize(distribution='1/n'); a
[      3     1/9     1/2     1/4]
[      1    1/39       2 -1955/2]
```

rank ( )

Return the rank of this matrix.

EXAMPLES:: sage: matrix(QQ,3,[1..9]).rank() 2 sage: matrix(QQ,100,[1..100^2]).rank() 2

row (*i*, *from_list=False*)

Return the i-th row of this matrix as a dense vector.

INPUT:

- `i` - integer
- `from_list` - ignored

EXAMPLES:

```
sage: matrix(QQ,2,[1/5,-2/3,3/4,4/9]).row(1)
(3/4, 4/9)
sage: matrix(QQ,2,[1/5,-2/3,3/4,4/9]).row(1,from_list=True)
(3/4, 4/9)
sage: matrix(QQ,2,[1/5,-2/3,3/4,4/9]).row(-2)
(1/5, -2/3)
```

set_row_to_multiple_of_row (*i*, *j*, *s*)

Set row i equal to s times row j.

EXAMPLES:

```
sage: a = matrix(QQ,2,3,range(6)); a
[0 1 2]
[3 4 5]
sage: a.set_row_to_multiple_of_row(1,0,-3)
sage: a
[ 0  1  2]
[ 0 -3 -6]
```

transpose ( )

Returns the transpose of self, without changing self.

EXAMPLES:

We create a matrix, compute its transpose, and note that the original matrix is not changed.

```
sage: A = matrix(QQ, 2, 3, range(6))
sage: type(A)
<type 'sage.matrix.matrix_rational_dense.Matrix_rational_dense'>
sage: B = A.transpose()
sage: print(B)
[0 3]
[1 4]
[2 5]
```

```
sage: print(A)
[0 1 2]
[3 4 5]
```

.T is a convenient shortcut for the transpose:

```
sage: print(A.T)
[0 3]
[1 4]
[2 5]
```

```
sage: A.subdivide(None, 1); A
[0|1 2]
[3|4 5]
sage: A.transpose()
[0 3]
[---]
[1 4]
[2 5]
```

CHAPTER

NINETEEN

DENSE MATRICES USING A NUMPY BACKEND.

This serves as a base class for dense matrices over Real Double Field and Complex Double Field.

AUTHORS:

- Jason Grout, Sep 2008: switch to NumPy backend, factored out the Matrix_double_dense class
- Josh Kantor
- William Stein: many bug fixes and touch ups.

EXAMPLES:

```
sage: b=Mat(RDF,2,3).basis()
sage: b[0]
[1.0 0.0 0.0]
[0.0 0.0 0.0]
```

We deal with the case of zero rows or zero columns:

```
sage: m = MatrixSpace(RDF,0,3)
sage: m.zero_matrix()
[]
```

class sage.matrix.matrix_double_dense.**Matrix_double_dense**

Bases: *sage.matrix.matrix_dense.Matrix_dense*

Base class for matrices over the Real Double Field and the Complex Double Field. These are supposed to be fast matrix operations using C doubles. Most operations are implemented using numpy which will call the underlying BLAS on the system.

This class cannot be instantiated on its own. The numpy matrix creation depends on several variables that are set in the subclasses.

EXAMPLES:

```
sage: m = Matrix(RDF, [[1,2],[3,4]])
sage: m**2
[ 7.0 10.0]
[15.0 22.0]
sage: m^(-1)        # rel tol 1e-15
[-1.9999999999999996  0.9999999999999998]
[ 1.4999999999999998 -0.4999999999999999]
```

LU ( )

Returns a decomposition of the (row-permuted) matrix as a product of a lower-triangular matrix ("L") and an upper-triangular matrix ("U").

OUTPUT:

For an $m \times n$ matrix `A` this method returns a triple of immutable matrices `P,L,U` such that

- `P*A = L*U`
- `P` is a square permutation matrix, of size $m \times m$, so is all zeroes, but with exactly a single one in each row and each column.
- `L` is lower-triangular, square of size $m \times m$, with every diagonal entry equal to one.
- `U` is upper-triangular with size $m \times n$, i.e. entries below the "diagonal" are all zero.

The computed decomposition is cached and returned on subsequent calls, thus requiring the results to be immutable.

Effectively, `P` permutes the rows of `A`. Then `L` can be viewed as a sequence of row operations on this matrix, where each operation is adding a multiple of a row to a subsequent row. There is no scaling (thus 1's on the diagonal of `L`) and no row-swapping (`P` does that). As a result `U` is close to being the result of Gaussian-elimination. However, round-off errors can make it hard to determine the zero entries of `U`.

Note: Sometimes this decomposition is written as `A=P*L*U`, where `P` represents the inverse permutation and is the matrix inverse of the `P` returned by this method. The computation of this matrix inverse can be accomplished quickly with just a transpose as the matrix is orthogonal/unitary.

EXAMPLES:

```
sage: m = matrix(RDF,4,range(16))
sage: P,L,U = m.LU()
sage: P*m
[12.0 13.0 14.0 15.0]
[ 0.0  1.0  2.0  3.0]
[ 8.0  9.0 10.0 11.0]
[ 4.0  5.0  6.0  7.0]
sage: L*U # rel tol 2e-16
[12.0 13.0 14.0 15.0]
[ 0.0  1.0  2.0  3.0]
[ 8.0  9.0 10.0 11.0]
[ 4.0  5.0  6.0  7.0]
```

trac ticket #10839 made this routine available for rectangular matrices.

```
sage: A = matrix(RDF, 5, 6, range(30)); A
[ 0.0  1.0  2.0  3.0  4.0  5.0]
[ 6.0  7.0  8.0  9.0 10.0 11.0]
[12.0 13.0 14.0 15.0 16.0 17.0]
[18.0 19.0 20.0 21.0 22.0 23.0]
[24.0 25.0 26.0 27.0 28.0 29.0]
sage: P, L, U = A.LU()
sage: P
[0.0 0.0 0.0 0.0 1.0]
[1.0 0.0 0.0 0.0 0.0]
[0.0 0.0 1.0 0.0 0.0]
[0.0 0.0 0.0 1.0 0.0]
[0.0 1.0 0.0 0.0 0.0]
sage: L.zero_at(0)   # Use zero_at(0) to get rid of signed zeros
[ 1.0  0.0  0.0  0.0  0.0]
[ 0.0  1.0  0.0  0.0  0.0]
[ 0.5  0.5  1.0  0.0  0.0]
```

```
[0.75 0.25  0.0  1.0  0.0]
[0.25 0.75  0.0  0.0  1.0]
sage: U.zero_at(0)   # Use zero_at(0) to get rid of signed zeros
[24.0 25.0 26.0 27.0 28.0 29.0]
[ 0.0  1.0  2.0  3.0  4.0  5.0]
[ 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]
sage: P*A-L*U
[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 0.0 0.0 0.0]
sage: P.transpose()*L*U
[ 0.0  1.0  2.0  3.0  4.0  5.0]
[ 6.0  7.0  8.0  9.0 10.0 11.0]
[12.0 13.0 14.0 15.0 16.0 17.0]
[18.0 19.0 20.0 21.0 22.0 23.0]
[24.0 25.0 26.0 27.0 28.0 29.0]
```

Trivial cases return matrices of the right size and characteristics.

```
sage: A = matrix(RDF, 5, 0)
sage: P, L, U = A.LU()
sage: P.parent()
Full MatrixSpace of 5 by 5 dense matrices over Real Double Field
sage: L.parent()
Full MatrixSpace of 5 by 5 dense matrices over Real Double Field
sage: U.parent()
Full MatrixSpace of 5 by 0 dense matrices over Real Double Field
sage: P*A-L*U
[]
```

The results are immutable since they are cached.

```
sage: P, L, U = matrix(RDF, 2, 2, range(4)).LU()
sage: L[0,0] = 0
Traceback (most recent call last):
    ...
ValueError: matrix is immutable; please change a copy instead (i.e., use␣
↪copy(M) to change a copy of M).
sage: P[0,0] = 0
Traceback (most recent call last):
    ...
ValueError: matrix is immutable; please change a copy instead (i.e., use␣
↪copy(M) to change a copy of M).
sage: U[0,0] = 0
Traceback (most recent call last):
    ...
ValueError: matrix is immutable; please change a copy instead (i.e., use␣
↪copy(M) to change a copy of M).
```

LU_valid()

Returns `True` if the LU form of this matrix has already been computed.

EXAMPLES:

```
sage: A = random_matrix(RDF,3) ; A.LU_valid()
False
sage: P, L, U = A.LU()
sage: A.LU_valid()
True
```

QR ( )

Returns a factorization into a unitary matrix and an upper-triangular matrix.

INPUT:

Any matrix over `RDF` or `CDF`.

OUTPUT:

`Q`, `R` – a pair of matrices such that if A is the original matrix, then

$$A = QR, \quad Q^*Q = I$$

where R is upper-triangular. Q^* is the conjugate-transpose in the complex case, and just the transpose in the real case. So Q is a unitary matrix (or rather, orthogonal, in the real case), or equivalently Q has orthogonal columns. For a matrix of full rank this factorization is unique up to adjustments via multiples of rows and columns by multiples with scalars having modulus 1. So in the full-rank case, R is unique if the diagonal entries are required to be positive real numbers.

The resulting decomposition is cached.

ALGORITHM:

Calls "linalg.qr" from SciPy, which is in turn an interface to LAPACK routines.

EXAMPLES:

Over the reals, the inverse of `Q` is its transpose, since including a conjugate has no effect. In the real case, we say `Q` is orthogonal.

```
sage: A = matrix(RDF, [[-2, 0, -4, -1, -1],
....:                   [-2, 1, -6, -3, -1],
....:                   [1, 1, 7, 4, 5],
....:                   [3, 0, 8, 3, 3],
....:                   [-1, 1, -6, -6, 5]])
sage: Q, R = A.QR()
```

At this point, `Q` is only well-defined up to the signs of its columns, and similarly for `R` and its rows, so we normalize them:

```
sage: Qnorm = Q._normalize_columns()
sage: Rnorm = R._normalize_rows()
sage: Qnorm.round(6).zero_at(10^-6)
[ 0.458831  0.126051  0.381212  0.394574   0.68744]
[ 0.458831  -0.47269 -0.051983 -0.717294  0.220963]
[-0.229416 -0.661766  0.661923  0.180872 -0.196411]
[-0.688247 -0.189076 -0.204468  -0.09663  0.662889]
[ 0.229416 -0.535715 -0.609939  0.536422 -0.024551]
sage: Rnorm.round(6).zero_at(10^-6)
[ 4.358899 -0.458831 13.076697  6.194225  2.982405]
[      0.0  1.670172  0.598741  -1.29202  6.207997]
[      0.0       0.0  5.444402  5.468661 -0.682716]
[      0.0       0.0       0.0  1.027626   -3.6193]
[      0.0       0.0       0.0       0.0  0.024551]
```

```
sage: (Q*Q.transpose())  # tol 1e-14
[0.9999999999999994                0.0                0.0                0.0 ␣
↪                0.0]
[                0.0                1.0                0.0                0.0 ␣
↪                0.0]
[                0.0                0.0 0.9999999999999999                0.0 ␣
↪                0.0]
[                0.0                0.0                0.0 0.9999999999999998 ␣
↪                0.0]
[                0.0                0.0                0.0                0.0␣
↪1.0000000000000002]
sage: (Q*R - A).zero_at(10^-14)
[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]
```

Now over the complex numbers, demonstrating that the SciPy libraries are (properly) using the Hermitian inner product, so that `Q` is a unitary matrix (its inverse is the conjugate-transpose).

```
sage: A = matrix(CDF, [[-8, 4*I + 1, -I + 2, 2*I + 1],
....:                 [1, -2*I - 1, -I + 3, -I + 1],
....:                 [I + 7, 2*I + 1, -2*I + 7, -I + 1],
....:                 [I + 2, 0, I + 12, -1]])
sage: Q, R = A.QR()
sage: Q._normalize_columns()  # tol 1e-6
[                            0.7302967433402214    0.20705664550556482 + 0.
↪5383472783144685*I   0.24630498099986423 - 0.07644563587232917*I   0.
↪23816176831943323 - 0.10365960327796941*I]
[                          -0.09128709291752768  -0.20705664550556482 - 0.
↪3778783780476558 4*I   0.3786559533863031 5 - 0.19522214955246678*I    0.
↪7012444502144682 - 0.3643711650986594 7*I]
[  -0.6390096504226938 - 0.09128709291752768*I    0.17082173254209104 + 0.
↪6677576817554466*I -0.03411475806452064 + 0.040901987417671426*I   0.
↪31401710855067644 - 0.08251917187054114*I]
[ -0.18257418583505536 - 0.09128709291752768*I  -0.03623491296347384 + 0.
↪07246982592694771*I    0.8632284069415112 + 0.06322839976356195*I  -0.
↪44996948676115206 - 0.01161191812089182*I]
sage: R._normalize_rows().zero_at(1e-15)  # tol 1e-6
[                        10.954451150103322                             -1.
↪9170289512680814*I   5.385938482134133 - 2.1908902300206643*I -0.
↪2738612787525829 - 2.1908902300206643*I]
[                                      0.0                              4.
↪8295962564173  -0.8696379111233719 - 5.864879483945123*I  0.993871898426711␣
↪- 0.30540855212070794*I]
[                                      0.0                                   ␣
↪      0.0                12.00160760935814 -0.2709533402297273 + 0.
↪4420629644486325*I]
[                                      0.0                                   ␣
↪      0.0                                     0.0                          ␣
↪1.9429639442589917]
sage: (Q.conjugate().transpose()*Q).zero_at(1e-15)  # tol 1e-15
[                1.0                0.0                0.0                0.0]
[                0.0 0.9999999999999994                0.0                0.0]
[                0.0                0.0 1.0000000000000002                0.0]
[                0.0                0.0                0.0 1.0000000000000004]
sage: (Q*R - A).zero_at(10^-14)
```

```
[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]
```

An example of a rectangular matrix that is also rank-deficient. If you run this example yourself, you may see a very small, nonzero entries in the third row, in the third column, even though the exact version of the matrix has rank 2. The final two columns of `Q` span the left kernel of `A` (as evidenced by the two zero rows of `R`). Different platforms will compute different bases for this left kernel, so we do not exhibit the actual matrix.

```
sage: Arat = matrix(QQ, [[2, -3, 3],
....:                     [-1, 1, -1],
....:                     [-1, 3, -3],
....:                     [-5, 1, -1]])
sage: Arat.rank()
2
sage: A = Arat.change_ring(CDF)
sage: Q, R = A.QR()
sage: R._normalize_rows()   # abs tol 1e-14
[     5.567764362830022    -2.6940795304016243     2.6940795304016243]
[                   0.0     3.5695847775155825    -3.5695847775155825]
[                   0.0                    0.0 2.4444034681064287e-16]
[                   0.0                    0.0                    0.0]
sage: (Q.conjugate_transpose()*Q)  # abs tol 1e-14
[     1.0000000000000002  -5.185196889911925e-17 -4.1457180570414476e-17  -2.
↪909388767229071e-17]
[ -5.185196889911925e-17      1.0000000000000002  -9.286869233696149e-17 -1.
↪1035822863186828e-16]
[-4.1457180570414476e-17  -9.286869233696149e-17                     1.0  4.
↪4159215672155694e-17]
[ -2.909388767229071e-17 -1.1035822863186828e-16  4.4159215672155694e-17
↪                  1.0]
```

Results are cached, meaning they are immutable matrices. Make a copy if you need to manipulate a result.

```
sage: A = random_matrix(CDF, 2, 2)
sage: Q, R = A.QR()
sage: Q.is_mutable()
False
sage: R.is_mutable()
False
sage: Q[0,0] = 0
Traceback (most recent call last):
...
ValueError: matrix is immutable; please change a copy instead (i.e., use
↪copy(M) to change a copy of M).
sage: Qcopy = copy(Q)
sage: Qcopy[0,0] = 679
sage: Qcopy[0,0]
679.0
```

SVD ( )

Return the singular value decomposition of this matrix.

The U and V matrices are not unique and may be returned with different values in the future or on different systems. The S matrix is unique and contains the singular values in descending order.

The computed decomposition is cached and returned on subsequent calls.

INPUT:

- A – a matrix

OUTPUT:

- U, S, V – immutable matrices such that $A = U * S * V.conj().transpose()$ where U and V are orthogonal and S is zero off of the diagonal.

Note that if self is m-by-n, then the dimensions of the matrices that this returns are (m,m), (m,n), and (n, n).

Note: If all you need is the singular values of the matrix, see the more convenient *singular_values()* .

EXAMPLES:

```
sage: m = matrix(RDF,4,range(1,17))
sage: U,S,V = m.SVD()
sage: U*S*V.transpose()   # tol 1e-14
[0.9999999999999993 1.9999999999999987  3.000000000000001  4.000000000000002]
[ 4.999999999999998  5.999999999999998  6.999999999999998                8.0]
[ 8.999999999999998  9.999999999999996 10.999999999999998               12.0]
[12.999999999999998               14.0               15.0               16.0]
```

A non-square example:

```
sage: m = matrix(RDF, 2, range(1,7)); m
[1.0 2.0 3.0]
[4.0 5.0 6.0]
sage: U, S, V = m.SVD()
sage: U*S*V.transpose()   # tol 1e-14
[0.9999999999999994 1.9999999999999998  2.999999999999999]
[ 4.000000000000001  5.000000000000002  6.000000000000001]
```

S contains the singular values:

```
sage: S.round(4)
[ 9.508    0.0    0.0]
[   0.0 0.7729    0.0]
sage: [round(sqrt(abs(x)),4) for x in (S*S.transpose()).eigenvalues()]
[9.508, 0.7729]
```

U and V are orthogonal matrices:

```
sage: U # random, SVD is not unique
[-0.386317703119 -0.922365780077]
[-0.922365780077  0.386317703119]
[-0.274721127897 -0.961523947641]
[-0.961523947641  0.274721127897]
sage: (U*U.transpose())   # tol 1e-15
[               1.0                0.0]
[               0.0 1.0000000000000004]
sage: V # random, SVD is not unique
[-0.428667133549  0.805963908589  0.408248290464]
[-0.566306918848  0.112382414097 -0.816496580928]
[-0.703946704147 -0.581199080396  0.408248290464]
```

```
sage: (V*V.transpose())   # tol 1e-15
[0.9999999999999999                0.0                0.0]
[               0.0                1.0                0.0]
[               0.0                0.0 0.9999999999999999]
```

cholesky ( )

Returns the Cholesky factorization of a matrix that is real symmetric, or complex Hermitian.

INPUT:

Any square matrix with entries from `RDF` that is symmetric, or with entries from `CDF` that is Hermitian. The matrix must be positive definite for the Cholesky decomposition to exist.

OUTPUT:

For a matrix A the routine returns a lower triangular matrix L such that,

$$A = LL^*$$

where L^* is the conjugate-transpose in the complex case, and just the transpose in the real case. If the matrix fails to be positive definite (perhaps because it is not symmetric or Hermitian), then this function raises a `ValueError` .

IMPLEMENTATION:

The existence of a Cholesky decomposition and the positive definite property are equivalent. So this method and the *`is_positive_definite()`* method compute and cache both the Cholesky decomposition and the positive-definiteness. So the *`is_positive_definite()`* method or catching a `ValueError` from the *`cholesky()`* method are equally expensive computationally and if the decomposition exists, it is cached as a side-effect of either routine.

EXAMPLES:

A real matrix that is symmetric and positive definite.

```
sage: M = matrix(RDF,[[ 1,   1,    1,     1,     1],
....:                 [ 1,   5,   31,   121,   341],
....:                 [ 1,  31,  341,  1555,  4681],
....:                 [ 1,121, 1555,  7381, 22621],
....:                 [ 1,341, 4681, 22621, 69905]])
sage: M.is_symmetric()
True
sage: L = M.cholesky()
sage: L.round(6).zero_at(10^-10)
[   1.0     0.0         0.0        0.0      0.0]
[   1.0     2.0         0.0        0.0      0.0]
[   1.0    15.0   10.723805        0.0      0.0]
[   1.0    60.0   60.985814   7.792973      0.0]
[   1.0   170.0  198.623524  39.366567   1.7231]
sage: (L*L.transpose()).round(6).zero_at(10^-10)
[ 1.0     1.0     1.0     1.0     1.0]
[ 1.0     5.0    31.0   121.0   341.0]
[ 1.0    31.0   341.0  1555.0  4681.0]
[ 1.0   121.0  1555.0  7381.0 22621.0]
[ 1.0   341.0  4681.0 22621.0 69905.0]
```

A complex matrix that is Hermitian and positive definite.

```
sage: A = matrix(CDF, [[        23,  17*I + 3,  24*I + 25,     21*I],
....:                  [ -17*I + 3,        38, -69*I + 89, 7*I + 15],
```

```
....:                      [-24*I + 25, 69*I + 89,         976, 24*I + 6],
....:                      [     -21*I, -7*I + 15,  -24*I + 6,        28]])
sage: A.is_hermitian()
True
sage: L = A.cholesky()
sage: L.round(6).zero_at(10^-10)
[               4.795832                      0.0                      0.0        ␣
↪0.0]
[  0.625543 - 3.544745*I                 5.004346                      0.0        ␣
↪0.0]
[   5.21286 - 5.004346*I 13.588189 + 10.721116*I                24.984023        ␣
↪0.0]
[             -4.378803*I  -0.104257 - 0.851434*I  -0.21486 + 0.371348*I  2.
↪811799]
sage: (L*L.conjugate_transpose()).round(6).zero_at(10^-10)
[         23.0   3.0 + 17.0*I 25.0 + 24.0*I        21.0*I]
[ 3.0 - 17.0*I           38.0 89.0 - 69.0*I  15.0 + 7.0*I]
[25.0 - 24.0*I 89.0 + 69.0*I          976.0  6.0 + 24.0*I]
[      -21.0*I  15.0 - 7.0*I   6.0 - 24.0*I          28.0]
```

This routine will recognize when the input matrix is not positive definite. The negative eigenvalues are an equivalent indicator. (Eigenvalues of a Hermitian matrix must be real, so there is no loss in ignoring the imprecise imaginary parts).

```
sage: A = matrix(RDF, [[ 3,   -6,    9,    6,   -9],
....:                      [-6,   11, -16, -11,   17],
....:                      [ 9, -16,   28,   16, -40],
....:                      [ 6, -11,   16,    9, -19],
....:                      [-9,   17, -40, -19,   68]])
sage: A.is_symmetric()
True
sage: A.eigenvalues()
[108.07..., 13.02..., -0.02..., -0.70..., -1.37...]
sage: A.cholesky()
Traceback (most recent call last):
...
ValueError: matrix is not positive definite

sage: B = matrix(CDF, [[      2, 4 - 2*I, 2 + 2*I],
....:                      [4 + 2*I,       8,    10*I],
....:                      [2 - 2*I,   -10*I,      -3]])
sage: B.is_hermitian()
True
sage: [ev.real() for ev in B.eigenvalues()]
[15.88..., 0.08..., -8.97...]
sage: B.cholesky()
Traceback (most recent call last):
...
ValueError: matrix is not positive definite
```

condition (*p='frob'*)

Returns the condition number of a square nonsingular matrix.

Roughly speaking, this is a measure of how sensitive the matrix is to round-off errors in numerical computations. The minimum possible value is 1.0, and larger numbers indicate greater sensitivity.

INPUT:

- `p` - default: 'frob' - controls which norm is used to compute the condition number, allowable values are 'frob' (for the Frobenius norm), integers -2, -1, 1, 2, positive and negative infinity. See output discussion for specifics.

OUTPUT:

The condition number of a matrix is the product of a norm of the matrix times the norm of the inverse of the matrix. This requires that the matrix be square and invertible (nonsingular, full rank).

Returned value is a double precision floating point value in `RDF` , or `Infinity` . Row and column sums described below are sums of the absolute values of the entries, where the absolute value of the complex number $a + bi$ is $\sqrt{a^2 + b^2}$. Singular values are the "diagonal" entries of the "S" matrix in the singular value decomposition.

- `p = 'frob'` : the default norm employed in computing the condition number, the Frobenius norm, which for a matrix $A = (a_{ij})$ computes

$$\left(\sum_{i,j} |a_{i,j}|^2\right)^{1/2}$$

- `p = 'sv'` : the quotient of the maximal and minimal singular value.
- `p = Infinity` or `p = oo` : the maximum row sum.
- `p = -Infinity` or `p = -oo` : the minimum column sum.
- `p = 1` : the maximum column sum.
- `p = -1` : the minimum column sum.
- `p = 2` : the 2-norm, equal to the maximum singular value.
- `p = -2` : the minimum singular value.

ALGORITHM:

Computation is performed by the `cond()` function of the SciPy/NumPy library.

EXAMPLES:

First over the reals.

```
sage: A = matrix(RDF, 4, [(1/4)*x^3 for x in range(16)]); A
[   0.0   0.25    2.0   6.75]
[  16.0  31.25   54.0  85.75]
[ 128.0 182.25  250.0 332.75]
[ 432.0 549.25  686.0 843.75]
sage: A.condition()
9923.88955...
sage: A.condition(p='frob')
9923.88955...
sage: A.condition(p=Infinity)  # tol 2e-14
22738.50000000045
sage: A.condition(p=-Infinity)  # tol 2e-14
17.50000000000028
sage: A.condition(p=1)
12139.21...
sage: A.condition(p=-1)  # tol 2e-14
550.0000000000093
sage: A.condition(p=2)
9897.8088...
sage: A.condition(p=-2)
0.000101032462...
```

And over the complex numbers.

```
sage: B = matrix(CDF, 3, [x + x^2*I for x in range(9)]); B
[          0.0   1.0 + 1.0*I   2.0 + 4.0*I]
[ 3.0 + 9.0*I 4.0 + 16.0*I 5.0 + 25.0*I]
[6.0 + 36.0*I 7.0 + 49.0*I 8.0 + 64.0*I]
sage: B.condition()
203.851798...
sage: B.condition(p='frob')
203.851798...
sage: B.condition(p=Infinity)
369.55630...
sage: B.condition(p=-Infinity)
5.46112969...
sage: B.condition(p=1)
289.251481...
sage: B.condition(p=-1)
20.4566639...
sage: B.condition(p=2)
202.653543...
sage: B.condition(p=-2)
0.00493453005...
```

Hilbert matrices are famously ill-conditioned, while an identity matrix can hit the minimum with the right norm.

```
sage: A = matrix(RDF, 10, [1/(i+j+1) for i in range(10) for j in range(10)])
sage: A.condition()   # tol 2e-4
16332197709146.014
sage: id = identity_matrix(CDF, 10)
sage: id.condition(p=1)
1.0
```

Return values are in RDF.

```
sage: A = matrix(CDF, 2, range(1,5))
sage: A.condition() in RDF
True
```

Rectangular and singular matrices raise errors if p is not 'sv'.

```
sage: A = matrix(RDF, 2, 3, range(6))
sage: A.condition()
Traceback (most recent call last):
...
TypeError: matrix must be square if p is not 'sv', not 2 x 3

sage: A.condition('sv')
7.34...

sage: A = matrix(QQ, 5, range(25))
sage: A.is_singular()
True
sage: B = A.change_ring(CDF)
sage: B.condition()
Traceback (most recent call last):
...
LinAlgError: Singular matrix
```

Improper values of `p` are caught.

```
sage: A = matrix(CDF, 2, range(1,5))
sage: A.condition(p='bogus')
Traceback (most recent call last):
...
ValueError: condition number 'p' must be +/- infinity, 'frob', 'sv' or an␣
↪integer, not bogus
sage: A.condition(p=632)
Traceback (most recent call last):
...
ValueError: condition number integer values of 'p' must be -2, -1, 1 or 2,␣
↪not 632
```

determinant()

Return the determinant of self.

ALGORITHM:

Use numpy

EXAMPLES:

```
sage: m = matrix(RDF,2,range(4)); m.det()
-2.0
sage: m = matrix(RDF,0,[]); m.det()
1.0
sage: m = matrix(RDF, 2, range(6)); m.det()
Traceback (most recent call last):
...
ValueError: self must be a square matrix
```

eigenvalues(*algorithm='default'*, *tol=None*)

Returns a list of eigenvalues.

INPUT:

- `self` - a square matrix
- `algorithm` - default: `'default'`
  - `'default'` - applicable to any matrix with double-precision floating point entries. Uses the `eigvals()` method from SciPy.
  - `'symmetric'` - converts the matrix into a real matrix (i.e. with entries from `RDF`), then applies the algorithm for Hermitian matrices. This algorithm can be significantly faster than the `'default'` algorithm.
  - `'hermitian'` - uses the `eigh()` method from SciPy, which applies only to real symmetric or complex Hermitian matrices. Since Hermitian is defined as a matrix equaling its conjugate-transpose, for a matrix with real entries this property is equivalent to being symmetric. This algorithm can be significantly faster than the `'default'` algorithm.
- `'tol'` - default: `None` - if set to a value other than `None` this is interpreted as a small real number used to aid in grouping eigenvalues that are numerically similar. See the output description for more information.

Warning: When using the `'symmetric'` or `'hermitian'` algorithms, no check is made on the input matrix, and only the entries below, and on, the main diagonal are employed in the computation.

Methods such as *is_symmetric()* and *is_hermitian()* could be used to verify this beforehand.

OUTPUT:

Default output for a square matrix of size n is a list of n eigenvalues from the complex double field, `CDF` . If the `'symmetric'` or `'hermitian'` algorithms are chosen, the returned eigenvalues are from the real double field, `RDF` .

If a tolerance is specified, an attempt is made to group eigenvalues that are numerically similar. The return is then a list of pairs, where each pair is an eigenvalue followed by its multiplicity. The eigenvalue reported is the mean of the eigenvalues computed, and these eigenvalues are contained in an interval (or disk) whose radius is less than `5*tol` for $n < 10,000$ in the worst case.

More precisely, for an $n \times n$ matrix, the diameter of the interval containing similar eigenvalues could be as large as sum of the reciprocals of the first n integers times `tol` .

Warning: Use caution when using the `tol` parameter to group eigenvalues. See the examples below to see how this can go wrong.

EXAMPLES:

```
sage: m = matrix(RDF, 2, 2, [1,2,3,4])
sage: ev = m.eigenvalues(); ev
[-0.372281323..., 5.37228132...]
sage: ev[0].parent()
Complex Double Field

sage: m = matrix(RDF, 2, 2, [0,1,-1,0])
sage: m.eigenvalues(algorithm='default')
[1.0*I, -1.0*I]

sage: m = matrix(CDF, 2, 2, [I,1,-I,0])
sage: m.eigenvalues()
[-0.624810533... + 1.30024259...*I, 0.624810533... - 0.30024259...*I]
```

The adjacency matrix of a graph will be symmetric, and the eigenvalues will be real.

```
sage: A = graphs.PetersenGraph().adjacency_matrix()
sage: A = A.change_ring(RDF)
sage: ev = A.eigenvalues(algorithm='symmetric'); ev  # tol 1e-14
[-2.0000000000000004, -1.9999999999999998, -1.9999999999999998, -1.
↪9999999999999993, 0.9999999999999994, 0.9999999999999997, 1.0, 1.
↪0000000000000002, 1.0000000000000004, 2.9999999999999996]
sage: ev[0].parent()
Real Double Field
```

The matrix `A` is "random", but the construction of `B` provides a positive-definite Hermitian matrix. Note that the eigenvalues of a Hermitian matrix are real, and the eigenvalues of a positive-definite matrix will be positive.

```
sage: A = matrix([[ 4*I + 5,  8*I + 1,  7*I + 5, 3*I + 5],
....:             [ 7*I - 2, -4*I + 7, -2*I + 4, 8*I + 8],
....:             [-2*I + 1,  6*I + 6,  5*I + 5,  -I - 4],
....:             [ 5*I + 1,  6*I + 2,    I - 4, -I + 3]])
```

```
sage: B = (A*A.conjugate_transpose()).change_ring(CDF)
sage: ev = B.eigenvalues(algorithm='hermitian'); ev
[2.68144025..., 49.5167998..., 274.086188..., 390.71557...]
sage: ev[0].parent()
Real Double Field
```

A tolerance can be given to aid in grouping eigenvalues that are similar numerically. However, if the parameter is too small it might split too finely. Too large, and it can go wrong very badly. Use with care.

```
sage: G = graphs.PetersenGraph()
sage: G.spectrum()
[3, 1, 1, 1, 1, 1, -2, -2, -2, -2]

sage: A = G.adjacency_matrix().change_ring(RDF)
sage: A.eigenvalues(algorithm='symmetric', tol=1.0e-5)  # tol 1e-15
[(-1.9999999999999998, 4), (1.0, 5), (2.9999999999999996, 1)]

sage: A.eigenvalues(algorithm='symmetric', tol=2.5)  # tol 1e-15
[(-1.9999999999999998, 4), (1.3333333333333333, 6)]
```

An (extreme) example of properly grouping similar eigenvalues.

```
sage: G = graphs.HigmanSimsGraph()
sage: A = G.adjacency_matrix().change_ring(RDF)
sage: A.eigenvalues(algorithm='symmetric', tol=1.0e-5)  # tol 2e-15
[(-8.0, 22), (1.9999999999999984, 77), (21.999999999999996, 1)]
```

eigenvectors_left ( )

Compute the left eigenvectors of a matrix of double precision real or complex numbers (i.e. RDF or CDF).

OUTPUT: Returns a list of triples, each of the form `(e,[v],1)`, where `e` is the eigenvalue, and `v` is an associated left eigenvector. If the matrix is of size n, then there are n triples. Values are computed with the SciPy library.

The format of this output is designed to match the format for exact results. However, since matrices here have numerical entries, the resulting eigenvalues will also be numerical. No attempt is made to determine if two eigenvalues are equal, or if eigenvalues might actually be zero. So the algebraic multiplicity of each eigenvalue is reported as 1. Decisions about equal eigenvalues or zero eigenvalues should be addressed in the calling routine.

The SciPy routines used for these computations produce eigenvectors normalized to have length 1, but on different hardware they may vary by a sign. So for doctests we have normalized output by forcing their eigenvectors to have their first non-zero entry equal to one.

EXAMPLES:

```
sage: m = matrix(RDF, [[-5, 3, 2, 8],[10, 2, 4, -2],[-1, -10, -10, -17],[-2,␣
↪7, 6, 13]])
sage: m
[ -5.0   3.0   2.0   8.0]
[ 10.0   2.0   4.0  -2.0]
[ -1.0 -10.0 -10.0 -17.0]
[ -2.0   7.0   6.0  13.0]
sage: spectrum = m.left_eigenvectors()
sage: for i in range(len(spectrum)):
....:   spectrum[i][1][0]=matrix(RDF, spectrum[i][1]).echelon_form()[0]
sage: spectrum[0]  # tol 1e-13
(2.000000000000675, [(1.0, 1.0000000000000138, 1.0000000000000147, 1.
↪0000000000000309)], 1)
```

```
sage: spectrum[1]   # tol 1e-13
(0.9999999999999164, [(0.9999999999999999, 0.7999999999999833, 0.
↪7999999999999836, 0.5999999999999696)], 1)
sage: spectrum[2]   # tol 1e-13
(-1.9999999999999782, [(1.0, 0.40000000000000335, 0.6000000000000039, 0.
↪2000000000000051)], 1)
sage: spectrum[3]   # tol 1e-13
(-1.0000000000000018, [(1.0, 0.9999999999999568, 1.9999999999998794, 1.
↪9999999999998472)], 1)
```

eigenvectors_right ( )

Compute the right eigenvectors of a matrix of double precision real or complex numbers (i.e. RDF or CDF).

OUTPUT:

Returns a list of triples, each of the form `(e,[v],1)`, where `e` is the eigenvalue, and `v` is an associated right eigenvector. If the matrix is of size n, then there are n triples. Values are computed with the SciPy library.

The format of this output is designed to match the format for exact results. However, since matrices here have numerical entries, the resulting eigenvalues will also be numerical. No attempt is made to determine if two eigenvalues are equal, or if eigenvalues might actually be zero. So the algebraic multiplicity of each eigenvalue is reported as 1. Decisions about equal eigenvalues or zero eigenvalues should be addressed in the calling routine.

The SciPy routines used for these computations produce eigenvectors normalized to have length 1, but on different hardware they may vary by a sign. So for doctests we have normalized output by forcing their eigenvectors to have their first non-zero entry equal to one.

EXAMPLES:

```
sage: m = matrix(RDF, [[-9, -14, 19, -74],[-1, 2, 4, -11],[-4, -12, 6, -
↪32],[0, -2, -1, 1]])
sage: m
[ -9.0 -14.0  19.0 -74.0]
[ -1.0   2.0   4.0 -11.0]
[ -4.0 -12.0   6.0 -32.0]
[  0.0  -2.0  -1.0   1.0]
sage: spectrum = m.right_eigenvectors()
sage: for i in range(len(spectrum)):
....:     spectrum[i][1][0]=matrix(RDF, spectrum[i][1]).echelon_form()[0]
sage: spectrum[0]   # tol 1e-13
(2.000000000000048, [(1.0, -2.0000000000001523, 3.000000000000181, 1.
↪0000000000000746)], 1)
sage: spectrum[1]   # tol 1e-13
(0.999999999999941, [(1.0, -0.666666666666633, 1.333333333333286, 0.
↪33333333333331555)], 1)
sage: spectrum[2]   # tol 1e-13
(-1.9999999999999483, [(1.0, -0.2000000000000063, 1.0000000000000173, 0.
↪20000000000000498)], 1)
sage: spectrum[3]   # tol 1e-13
(-1.0000000000000406, [(1.0, -0.49999999999996264, 1.9999999999998617, 0.
↪499999999999958)], 1)
```

exp (*algorithm=None*, *order=None*)

Calculate the exponential of this matrix X, which is the matrix

$$e^X = \sum_{k=0}^{\infty} \frac{X^k}{k!}.$$

INPUT:

- algorithm – deprecated
- order – deprecated

EXAMPLES:

```
sage: A=matrix(RDF, 2, [1,2,3,4]); A
[1.0 2.0]
[3.0 4.0]
sage: A.exp()  # tol 1e-15
[51.968956198705044  74.73656456700327]
[112.10484685050491 164.07380304920997]
sage: A=matrix(CDF, 2, [1,2+I,3*I,4]); A
[        1.0 2.0 + 1.0*I]
[      3.0*I         4.0]
sage: A.exp()  # tol 1.1e-14
[-19.614602953804912 + 12.517743846762578*I   3.7949636449582176 + 28.
↪88379930658099*I]
[ -32.383580980922254 + 21.88423595789845*I   2.269633004093535 + 44.
↪901324827684824*I]
```

is_hermitian (*tol=1e-12*, *algorithm='orthonormal'*)

Returns `True` if the matrix is equal to its conjugate-transpose.

INPUT:

- `tol` - default: `1e-12` - the largest value of the absolute value of the difference between two matrix entries for which they will still be considered equal.
- `algorithm` - default: 'orthonormal' - set to 'orthonormal' for a stable procedure and set to 'naive' for a fast procedure.

OUTPUT:

`True` if the matrix is square and equal to the transpose with every entry conjugated, and `False` otherwise.

Note that if conjugation has no effect on elements of the base ring (such as for integers), then the *`is_symmetric()`* method is equivalent and faster.

The tolerance parameter is used to allow for numerical values to be equal if there is a slight difference due to round-off and other imprecisions.

The result is cached, on a per-tolerance and per-algorithm basis.

ALGORITHMS:

The naive algorithm simply compares corresponing entries on either side of the diagonal (and on the diagonal itself) to see if they are conjugates, with equality controlled by the tolerance parameter.

The orthonormal algorithm first computes a Schur decomposition (via the *`schur()`* method) and checks that the result is a diagonal matrix with real entries.

So the naive algorithm can finish quickly for a matrix that is not Hermitian, while the orthonormal algorithm will always compute a Schur decomposition before going through a similar check of the matrix entry-by-entry.

EXAMPLES:

```
sage: A = matrix(CDF, [[ 1 + I,  1 - 6*I, -1 - I],
....:                  [-3 - I,     -4*I,     -2],
....:                  [-1 + I, -2 - 8*I,  2 + I]])
sage: A.is_hermitian(algorithm='orthonormal')
False
sage: A.is_hermitian(algorithm='naive')
False
sage: B = A*A.conjugate_transpose()
sage: B.is_hermitian(algorithm='orthonormal')
True
sage: B.is_hermitian(algorithm='naive')
True
```

A matrix that is nearly Hermitian, but for one non-real diagonal entry.

```
sage: A = matrix(CDF, [[    2,   2-I, 1+4*I],
....:                  [  2+I,   3+I, 2-6*I],
....:                  [1-4*I, 2+6*I,     5]])
sage: A.is_hermitian(algorithm='orthonormal')
False
sage: A[1,1] = 132
sage: A.is_hermitian(algorithm='orthonormal')
True
```

We get a unitary matrix from the SVD routine and use this numerical matrix to create a matrix that should be Hermitian (indeed it should be the identity matrix), but with some imprecision. We use this to illustrate that if the tolerance is set too small, then we can be too strict about the equality of entries and may achieve the wrong result (depending on the system):

```
sage: A = matrix(CDF, [[ 1 + I,  1 - 6*I, -1 - I],
....:                  [-3 - I,     -4*I,     -2],
....:                  [-1 + I, -2 - 8*I,  2 + I]])
sage: U, _, _ = A.SVD()
sage: B=U*U.conjugate_transpose()
sage: B.is_hermitian(algorithm='naive')
True
sage: B.is_hermitian(algorithm='naive', tol=1.0e-17)  # random
False
sage: B.is_hermitian(algorithm='naive', tol=1.0e-15)
True
```

A square, empty matrix is trivially Hermitian.

```
sage: A = matrix(RDF, 0, 0)
sage: A.is_hermitian()
True
```

Rectangular matrices are never Hermitian, no matter which algorithm is requested.

```
sage: A = matrix(CDF, 3, 4)
sage: A.is_hermitian()
False
```

AUTHOR:

- Rob Beezer (2011-03-30)

is_normal (*tol=1e-12*, *algorithm='orthonormal'*)

Returns `True` if the matrix commutes with its conjugate-transpose.

INPUT:

- `tol` - default: `1e-12` - the largest value of the absolute value of the difference between two matrix entries for which they will still be considered equal.
- `algorithm` - default: 'orthonormal' - set to 'orthonormal' for a stable procedure and set to 'naive' for a fast procedure.

OUTPUT:

`True` if the matrix is square and commutes with its conjugate-transpose, and `False` otherwise.

Normal matrices are precisely those that can be diagonalized by a unitary matrix.

The tolerance parameter is used to allow for numerical values to be equal if there is a slight difference due to round-off and other imprecisions.

The result is cached, on a per-tolerance and per-algorithm basis.

ALGORITHMS:

The naive algorithm simply compares entries of the two possible products of the matrix with its conjugate-transpose, with equality controlled by the tolerance parameter.

The orthonormal algorithm first computes a Schur decomposition (via the *schur()* method) and checks that the result is a diagonal matrix. An orthonormal diagonalization is equivalent to being normal.

So the naive algorithm can finish fairly quickly for a matrix that is not normal, once the products have been computed. However, the orthonormal algorithm will compute a Schur decomposition before going through a similar check of a matrix entry-by-entry.

EXAMPLES:

First over the complexes. `B` is Hermitian, hence normal.

```
sage: A = matrix(CDF, [[ 1 + I,  1 - 6*I, -1 - I],
....:                  [-3 - I,     -4*I,     -2],
....:                  [-1 + I, -2 - 8*I,  2 + I]])
sage: B = A*A.conjugate_transpose()
sage: B.is_hermitian()
True
sage: B.is_normal(algorithm='orthonormal')
True
sage: B.is_normal(algorithm='naive')
True
sage: B[0,0] = I
sage: B.is_normal(algorithm='orthonormal')
False
sage: B.is_normal(algorithm='naive')
False
```

Now over the reals. Circulant matrices are normal.

```
sage: G = graphs.CirculantGraph(20, [3, 7])
sage: D = digraphs.Circuit(20)
sage: A = 3*D.adjacency_matrix() - 5*G.adjacency_matrix()
sage: A = A.change_ring(RDF)
sage: A.is_normal()
True
sage: A.is_normal(algorithm = 'naive')
```

```
True
sage: A[19,0] = 4.0
sage: A.is_normal()
False
sage: A.is_normal(algorithm = 'naive')
False
```

Skew-Hermitian matrices are normal.

```
sage: A = matrix(CDF, [[ 1 + I,  1 - 6*I, -1 - I],
....:                  [-3 - I,     -4*I,     -2],
....:                  [-1 + I, -2 - 8*I,  2 + I]])
sage: B = A - A.conjugate_transpose()
sage: B.is_hermitian()
False
sage: B.is_normal()
True
sage: B.is_normal(algorithm='naive')
True
```

A small matrix that does not fit into any of the usual categories of normal matrices.

```
sage: A = matrix(RDF, [[1, -1],
....:                  [1,  1]])
sage: A.is_normal()
True
sage: not A.is_hermitian() and not A.is_skew_symmetric()
True
```

Sage has several fields besides the entire complex numbers where conjugation is non-trivial.

```
sage: F.<b> = QuadraticField(-7)
sage: C = matrix(F, [[-2*b - 3,  7*b - 6, -b + 3],
....:                [-2*b - 3, -3*b + 2,   -2*b],
....:                [   b + 1,        0,     -2]])
sage: C = C*C.conjugate_transpose()
sage: C.is_normal()
True
```

A square, empty matrix is trivially normal.

```
sage: A = matrix(CDF, 0, 0)
sage: A.is_normal()
True
```

Rectangular matrices are never normal, no matter which algorithm is requested.

```
sage: A = matrix(CDF, 3, 4)
sage: A.is_normal()
False
```

AUTHOR:

- Rob Beezer (2011-03-31)

is_positive_definite ()

Determines if a matrix is positive definite.

A matrix A is positive definite if it is square, is Hermitian (which reduces to symmetric in the real case), and for every nonzero vector $\vec{x}$,

$$\vec{x}^* A\vec{x} > 0$$

where $\vec{x}^*$ is the conjugate-transpose in the complex case and just the transpose in the real case. Equivalently, a positive definite matrix has only positive eigenvalues and only positive determinants of leading principal submatrices.

INPUT:

Any matrix over `RDF` or `CDF` .

OUTPUT:

`True` if and only if the matrix is square, Hermitian, and meets the condition above on the quadratic form. The result is cached.

IMPLEMENTATION:

The existence of a Cholesky decomposition and the positive definite property are equivalent. So this method and the *cholesky()* method compute and cache both the Cholesky decomposition and the positive-definiteness. So the *is_positive_definite()* method or catching a `ValueError` from the *cholesky()* method are equally expensive computationally and if the decomposition exists, it is cached as a side-effect of either routine.

EXAMPLES:

A matrix over `RDF` that is positive definite.

```
sage: M = matrix(RDF,[[ 1,   1,    1,     1,     1],
....:                 [ 1,   5,   31,   121,   341],
....:                 [ 1,  31,  341,  1555,  4681],
....:                 [ 1,121, 1555,  7381, 22621],
....:                 [ 1,341, 4681, 22621, 69905]])
sage: M.is_symmetric()
True
sage: M.eigenvalues()
[77547.66..., 82.44..., 2.41..., 0.46..., 0.011...]
sage: [round(M[:i,:i].determinant()) for i in range(1, M.nrows()+1)]
[1, 4, 460, 27936, 82944]
sage: M.is_positive_definite()
True
```

A matrix over `CDF` that is positive definite.

```
sage: C = matrix(CDF, [[        23,  17*I + 3,  24*I + 25,      21*I],
....:                  [ -17*I + 3,        38, -69*I + 89, 7*I + 15],
....:                  [-24*I + 25, 69*I + 89,        976, 24*I + 6],
....:                  [     -21*I, -7*I + 15,  -24*I + 6,       28]])
sage: C.is_hermitian()
True
sage: [x.real() for x in C.eigenvalues()]
[991.46..., 55.96..., 3.69..., 13.87...]
sage: [round(C[:i,:i].determinant().real()) for i in range(1, C.nrows()+1)]
[23, 576, 359540, 2842600]
sage: C.is_positive_definite()
True
```

A matrix over `RDF` that is not positive definite.

```
sage: A = matrix(RDF, [[ 3,  -6,   9,   6,  -9],
....:                  [-6,  11, -16, -11,  17],
....:                  [ 9, -16,  28,  16, -40],
....:                  [ 6, -11,  16,   9, -19],
....:                  [-9,  17, -40, -19,  68]])
sage: A.is_symmetric()
True
sage: A.eigenvalues()
[108.07..., 13.02..., -0.02..., -0.70..., -1.37...]
sage: [round(A[:i,:i].determinant()) for i in range(1, A.nrows()+1)]
[3, -3, -15, 30, -30]
sage: A.is_positive_definite()
False
```

A matrix over CDF that is not positive definite.

```
sage: B = matrix(CDF, [[      2, 4 - 2*I, 2 + 2*I],
....:                  [4 + 2*I,       8,    10*I],
....:                  [2 - 2*I,   -10*I,      -3]])
sage: B.is_hermitian()
True
sage: [ev.real() for ev in B.eigenvalues()]
[15.88..., 0.08..., -8.97...]
sage: [round(B[:i,:i].determinant().real()) for i in range(1, B.nrows()+1)]
[2, -4, -12]
sage: B.is_positive_definite()
False
```

A large random matrix that is guaranteed by theory to be positive definite.

```
sage: R = random_matrix(CDF, 200)
sage: H = R.conjugate_transpose()*R
sage: H.is_positive_definite()
True
```

AUTHOR:

- Rob Beezer (2012-05-28)

is_symmetric (*tol=1e-12*)

Return whether this matrix is symmetric, to the given tolerance.

EXAMPLES:

```
sage: m = matrix(RDF,2,2,range(4)); m
[0.0 1.0]
[2.0 3.0]
sage: m.is_symmetric()
False
sage: m[1,0] = 1.1; m
[0.0 1.0]
[1.1 3.0]
sage: m.is_symmetric()
False
```

The tolerance inequality is strict: sage: m.is_symmetric(tol=0.1) False sage: m.is_symmetric(tol=0.11)
True

is_unitary (*tol=1e-12*, *algorithm='orthonormal'*)

Returns `True` if the columns of the matrix are an orthonormal basis.

For a matrix with real entries this determines if a matrix is "orthogonal" and for a matrix with complex entries this determines if the matrix is "unitary."

INPUT:

- `tol` - default: `1e-12` - the largest value of the absolute value of the difference between two matrix entries for which they will still be considered equal.
- `algorithm` - default: 'orthonormal' - set to 'orthonormal' for a stable procedure and set to 'naive' for a fast procedure.

OUTPUT:

`True` if the matrix is square and its conjugate-transpose is its inverse, and `False` otherwise. In other words, a matrix is orthogonal or unitary if the product of its conjugate-transpose times the matrix is the identity matrix.

The tolerance parameter is used to allow for numerical values to be equal if there is a slight difference due to round-off and other imprecisions.

The result is cached, on a per-tolerance and per-algorithm basis.

ALGORITHMS:

The naive algorithm simply computes the product of the conjugate-transpose with the matrix and compares the entries to the identity matrix, with equality controlled by the tolerance parameter.

The orthonormal algorithm first computes a Schur decomposition (via the *schur()* method) and checks that the result is a diagonal matrix with entries of modulus 1, which is equivalent to being unitary.

So the naive algorithm might finish fairly quickly for a matrix that is not unitary, once the product has been computed. However, the orthonormal algorithm will compute a Schur decomposition before going through a similar check of a matrix entry-by-entry.

EXAMPLES:

A matrix that is far from unitary.

```
sage: A = matrix(RDF, 4, range(16))
sage: A.conjugate().transpose()*A
[224.0 248.0 272.0 296.0]
[248.0 276.0 304.0 332.0]
[272.0 304.0 336.0 368.0]
[296.0 332.0 368.0 404.0]
sage: A.is_unitary()
False
sage: A.is_unitary(algorithm='naive')
False
sage: A.is_unitary(algorithm='orthonormal')
False
```

The QR decomposition will produce a unitary matrix as Q and the SVD decomposition will create two unitary matrices, U and V.

```
sage: A = matrix(CDF, [[   1 - I,   -3*I,  -2 + I,        1, -2 + 3*I],
....:                  [   1 - I, -2 + I, 1 + 4*I,        0,    2 + I],
....:                  [      -1, -5 + I,  -2 + I,    1 + I, -5 - 4*I],
....:                  [-2 + 4*I,  2 - I, 8 - 4*I,  1 - 8*I,  3 - 2*I]])
sage: Q, R = A.QR()
sage: Q.is_unitary()
```

```
True
sage: U, S, V = A.SVD()
sage: U.is_unitary(algorithm='naive')
True
sage: U.is_unitary(algorithm='orthonormal')
True
sage: V.is_unitary(algorithm='naive')
True
```

If we make the tolerance too strict we can get misleading results.

```
sage: A = matrix(RDF, 10, 10, [1/(i+j+1) for i in range(10) for j in␣
↪range(10)])
sage: Q, R = A.QR()
sage: Q.is_unitary(algorithm='naive', tol=1e-16)
False
sage: Q.is_unitary(algorithm='orthonormal', tol=1e-17)
False
```

Rectangular matrices are not unitary/orthogonal, even if their columns form an orthonormal set.

```
sage: A = matrix(CDF, [[1,0], [0,0], [0,1]])
sage: A.is_unitary()
False
```

The smallest cases. The Schur decomposition used by the orthonormal algorithm will fail on a matrix of size zero.

```
sage: P = matrix(CDF, 0, 0)
sage: P.is_unitary(algorithm='naive')
True

sage: P = matrix(CDF, 1, 1, [1])
sage: P.is_unitary(algorithm='orthonormal')
True

sage: P = matrix(CDF, 0, 0,)
sage: P.is_unitary(algorithm='orthonormal')
Traceback (most recent call last):
...
ValueError: failed to create intent(cache|hide)|optional array-- must have␣
↪defined dimensions but got (0,)
```

AUTHOR:

- Rob Beezer (2011-05-04)

left_eigenvectors ()

Compute the left eigenvectors of a matrix of double precision real or complex numbers (i.e. RDF or CDF).

OUTPUT: Returns a list of triples, each of the form `(e,[v],1)`, where `e` is the eigenvalue, and `v` is an associated left eigenvector. If the matrix is of size n, then there are n triples. Values are computed with the SciPy library.

The format of this output is designed to match the format for exact results. However, since matrices here have numerical entries, the resulting eigenvalues will also be numerical. No attempt is made to determine if two eigenvalues are equal, or if eigenvalues might actually be zero. So the algebraic multiplicity of each eigenvalue is reported as 1. Decisions about equal eigenvalues or zero eigenvalues should be addressed in the calling routine.

The SciPy routines used for these computations produce eigenvectors normalized to have length 1, but on different hardware they may vary by a sign. So for doctests we have normalized output by forcing their eigenvectors to have their first non-zero entry equal to one.

EXAMPLES:

```
sage: m = matrix(RDF, [[-5, 3, 2, 8],[10, 2, 4, -2],[-1, -10, -10, -17],[-2,␣
↪7, 6, 13]])
sage: m
[ -5.0   3.0   2.0   8.0]
[ 10.0   2.0   4.0  -2.0]
[ -1.0 -10.0 -10.0 -17.0]
[ -2.0   7.0   6.0  13.0]
sage: spectrum = m.left_eigenvectors()
sage: for i in range(len(spectrum)):
....:   spectrum[i][1][0]=matrix(RDF, spectrum[i][1]).echelon_form()[0]
sage: spectrum[0]   # tol 1e-13
(2.0000000000000675, [(1.0, 1.0000000000000138, 1.0000000000000147, 1.
↪0000000000000309)], 1)
sage: spectrum[1]   # tol 1e-13
(0.9999999999999164, [(0.9999999999999999, 0.7999999999999833, 0.
↪7999999999999836, 0.5999999999999696)], 1)
sage: spectrum[2]   # tol 1e-13
(-1.9999999999999782, [(1.0, 0.40000000000000335, 0.6000000000000039, 0.
↪2000000000000051)], 1)
sage: spectrum[3]   # tol 1e-13
(-1.0000000000000018, [(1.0, 0.9999999999999568, 1.9999999999998794, 1.
↪9999999999998472)], 1)
```

log_determinant ( )

Compute the log of the absolute value of the determinant using LU decomposition.

Note: This is useful if the usual determinant overflows.

EXAMPLES:

```
sage: m = matrix(RDF,2,2,range(4)); m
[0.0 1.0]
[2.0 3.0]
sage: RDF(log(abs(m.determinant())))
0.6931471805599453
sage: m.log_determinant()
0.6931471805599453
sage: m = matrix(RDF,0,0,[]); m
[]
sage: m.log_determinant()
0.0
sage: m = matrix(CDF,2,2,range(4)); m
[0.0 1.0]
[2.0 3.0]
sage: RDF(log(abs(m.determinant())))
0.6931471805599453
sage: m.log_determinant()
0.6931471805599453
sage: m = matrix(CDF,0,0,[]); m
[]
sage: m.log_determinant()
```

```
0.0
```

norm (*p=2*)

Returns the norm of the matrix.

INPUT:

- p - default: 2 - controls which norm is computed, allowable values are 'frob' (for the Frobenius norm), integers -2, -1, 1, 2, positive and negative infinity. See output discussion for specifics.

OUTPUT:

Returned value is a double precision floating point value in `RDF` . Row and column sums described below are sums of the absolute values of the entries, where the absolute value of the complex number $a + bi$ is $\sqrt{a^2 + b^2}$. Singular values are the "diagonal" entries of the "S" matrix in the singular value decomposition.

- `p = 'frob'` : the Frobenius norm, which for a matrix $A = (a_{ij})$ computes

$$\left(\sum_{i,j} |a_{i,j}|^2\right)^{1/2}$$

- `p = Infinity` or `p = oo` : the maximum row sum.
- `p = -Infinity` or `p = -oo` : the minimum column sum.
- `p = 1` : the maximum column sum.
- `p = -1` : the minimum column sum.
- `p = 2` : the induced 2-norm, equal to the maximum singular value.
- `p = -2` : the minimum singular value.

ALGORITHM:

Computation is performed by the `norm()` function of the SciPy/NumPy library.

EXAMPLES:

First over the reals.

```
sage: A = matrix(RDF, 3, range(-3, 6)); A
[-3.0 -2.0 -1.0]
[ 0.0  1.0  2.0]
[ 3.0  4.0  5.0]
sage: A.norm()
7.99575670...
sage: A.norm(p='frob')
8.30662386...
sage: A.norm(p=Infinity)
12.0
sage: A.norm(p=-Infinity)
3.0
sage: A.norm(p=1)
8.0
sage: A.norm(p=-1)
6.0
sage: A.norm(p=2)
7.99575670...
sage: A.norm(p=-2) < 10^-15
True
```

And over the complex numbers.

```
sage: B = matrix(CDF, 2, [[1+I, 2+3*I],[3+4*I,3*I]]); B
[1.0 + 1.0*I 2.0 + 3.0*I]
[3.0 + 4.0*I       3.0*I]
sage: B.norm()
6.66189877...
sage: B.norm(p='frob')
7.0
sage: B.norm(p=Infinity)
8.0
sage: B.norm(p=-Infinity)
5.01976483...
sage: B.norm(p=1)
6.60555127...
sage: B.norm(p=-1)
6.41421356...
sage: B.norm(p=2)
6.66189877...
sage: B.norm(p=-2)
2.14921023...
```

Since it is invariant under unitary multiplication, the Frobenius norm is equal to the square root of the sum of squares of the singular values.

```
sage: A = matrix(RDF, 5, range(1,26))
sage: f = A.norm(p='frob')
sage: U, S, V = A.SVD()
sage: s = sqrt(sum([S[i,i]^2 for i in range(5)]))
sage: abs(f-s) < 1.0e-12
True
```

Return values are in RDF.

```
sage: A = matrix(CDF, 2, range(4))
sage: A.norm() in RDF
True
```

Improper values of `p` are caught.

```
sage: A.norm(p='bogus')
Traceback (most recent call last):
...
ValueError: matrix norm 'p' must be +/- infinity, 'frob' or an integer, not␣
↪bogus
sage: A.norm(p=632)
Traceback (most recent call last):
...
ValueError: matrix norm integer values of 'p' must be -2, -1, 1 or 2, not 632
```

numpy (*dtype=None*)

This method returns a copy of the matrix as a numpy array. It uses the numpy C/api so is very fast.

INPUT:

- `dtype` - The desired data-type for the array. If not given, then the type will be determined as the minimum type required to hold the objects in the sequence.

EXAMPLES:

```
sage: m = matrix(RDF,[[1,2],[3,4]])
sage: n = m.numpy()
sage: import numpy
sage: numpy.linalg.eig(n)
(array([-0.37228132,  5.37228132]), array([[-0.82456484, -0.41597356],
       [ 0.56576746, -0.90937671]]))
sage: m = matrix(RDF, 2, range(6)); m
[0.0 1.0 2.0]
[3.0 4.0 5.0]
sage: m.numpy()
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.]])
```

Alternatively, numpy automatically calls this function (via the magic `__array__()` method) to convert Sage matrices to numpy arrays:

```
sage: import numpy
sage: m = matrix(RDF, 2, range(6)); m
[0.0 1.0 2.0]
[3.0 4.0 5.0]
sage: numpy.array(m)
array([[ 0.,  1.,  2.],
       [ 3.,  4.,  5.]])
sage: numpy.array(m).dtype
dtype('float64')
sage: m = matrix(CDF, 2, range(6)); m
[0.0 1.0 2.0]
[3.0 4.0 5.0]
sage: numpy.array(m)
array([[ 0.+0.j,  1.+0.j,  2.+0.j],
       [ 3.+0.j,  4.+0.j,  5.+0.j]])
sage: numpy.array(m).dtype
dtype('complex128')
```

right_eigenvectors ()

Compute the right eigenvectors of a matrix of double precision real or complex numbers (i.e. RDF or CDF).

OUTPUT:

Returns a list of triples, each of the form `(e,[v],1)`, where `e` is the eigenvalue, and `v` is an associated right eigenvector. If the matrix is of size n, then there are n triples. Values are computed with the SciPy library.

The format of this output is designed to match the format for exact results. However, since matrices here have numerical entries, the resulting eigenvalues will also be numerical. No attempt is made to determine if two eigenvalues are equal, or if eigenvalues might actually be zero. So the algebraic multiplicity of each eigenvalue is reported as 1. Decisions about equal eigenvalues or zero eigenvalues should be addressed in the calling routine.

The SciPy routines used for these computations produce eigenvectors normalized to have length 1, but on different hardware they may vary by a sign. So for doctests we have normalized output by forcing their eigenvectors to have their first non-zero entry equal to one.

EXAMPLES:

```
sage: m = matrix(RDF, [[-9, -14, 19, -74],[-1, 2, 4, -11],[-4, -12, 6, -
↪32],[0, -2, -1, 1]])
sage: m
```

```
[ -9.0 -14.0  19.0 -74.0]
[ -1.0   2.0   4.0 -11.0]
[ -4.0 -12.0   6.0 -32.0]
[  0.0  -2.0  -1.0   1.0]
sage: spectrum = m.right_eigenvectors()
sage: for i in range(len(spectrum)):
....:   spectrum[i][1][0]=matrix(RDF, spectrum[i][1]).echelon_form()[0]
sage: spectrum[0]  # tol 1e-13
(2.000000000000048, [(1.0, -2.0000000000001523, 3.000000000000181, 1.
↪0000000000000746)], 1)
sage: spectrum[1]  # tol 1e-13
(0.999999999999941, [(1.0, -0.666666666666633, 1.333333333333286, 0.
↪33333333333331555)], 1)
sage: spectrum[2]  # tol 1e-13
(-1.9999999999999483, [(1.0, -0.2000000000000063, 1.0000000000000173, 0.
↪20000000000000498)], 1)
sage: spectrum[3]  # tol 1e-13
(-1.0000000000000406, [(1.0, -0.49999999999996264, 1.9999999999998617, 0.
↪499999999999958)], 1)
```

round (*ndigits=0*)

Returns a copy of the matrix where all entries have been rounded to a given precision in decimal digits (default 0 digits).

INPUT:

- `ndigits` - The precision in number of decimal digits

OUTPUT:

A modified copy of the matrix

EXAMPLES:

```
sage: M = matrix(CDF, [[10.234r + 34.2343jr, 34e10r]])
sage: M
[10.234 + 34.2343*I     340000000000.0]
sage: M.round(2)
[10.23 + 34.23*I  340000000000.0]
sage: M.round()
[ 10.0 + 34.0*I 340000000000.0]
```

schur (*base_ring=None*)

Returns the Schur decomposition of the matrix.

INPUT:

- `base_ring` - optional, defaults to the base ring of `self`. Use this to request the base ring of the returned matrices, which will affect the format of the results.

OUTPUT:

A pair of immutable matrices. The first is a unitary matrix Q. The second, T, is upper-triangular when returned over the complex numbers, while it is almost upper-triangular over the reals. In the latter case, there can be some 2×2 blocks on the diagonal which represent a pair of conjugate complex eigenvalues of `self`.

If `self` is the matrix A, then

$$A = QT(\overline{Q})^t$$

where the latter matrix is the conjugate-transpose of `Q`, which is also the inverse of `Q`, since `Q` is unitary.

Note that in the case of a normal matrix (Hermitian, symmetric, and others), the upper-triangular matrix is a diagonal matrix with eigenvalues of `self` on the diagonal, and the unitary matrix has columns that form an orthonormal basis composed of eigenvectors of `self`. This is known as "orthonormal diagonalization".

Warning: The Schur decomposition is not unique, as there may be numerous choices for the vectors of the orthonormal basis, and consequently different possibilities for the upper-triangular matrix. However, the diagonal of the upper-triangular matrix will always contain the eigenvalues of the matrix (in the complex version), or 2×2 block matrices in the real version representing pairs of conjugate complex eigenvalues.

In particular, results may vary across systems and processors.

EXAMPLES:

First over the complexes. The similar matrix is always upper-triangular in this case.

```
sage: A = matrix(CDF, 4, 4, range(16)) + matrix(CDF, 4, 4, [x^3*I for x in␣
↪range(0, 16)])
sage: Q, T = A.schur()
sage: (Q*Q.conjugate().transpose()).zero_at(1e-12)  # tol 1e-12
[ 0.999999999999999                0.0                0.0                0.0]
[               0.0 0.9999999999999996                0.0                0.0]
[               0.0                0.0 0.9999999999999992                0.0]
[               0.0                0.0                0.0 0.9999999999999999]
sage: all([T.zero_at(1.0e-12)[i,j] == 0 for i in range(4) for j in range(i)])
True
sage: (Q*T*Q.conjugate().transpose()-A).zero_at(1.0e-11)
[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]
sage: eigenvalues = [T[i,i] for i in range(4)]; eigenvalues
[30.733... + 4648.541...*I, -0.184... - 159.057...*I, -0.523... + 11.158...
↪*I, -0.025... - 0.642...*I]
sage: A.eigenvalues()
[30.733... + 4648.541...*I, -0.184... - 159.057...*I, -0.523... + 11.158...
↪*I, -0.025... - 0.642...*I]
sage: abs(A.norm()-T.norm()) < 1e-10
True
```

We begin with a real matrix but ask for a decomposition over the complexes. The result will yield an upper-triangular matrix over the complex numbers for `T`.

```
sage: A = matrix(RDF, 4, 4, [x^3 for x in range(16)])
sage: Q, T = A.schur(base_ring=CDF)
sage: (Q*Q.conjugate().transpose()).zero_at(1e-12)  # tol 1e-12
[0.9999999999999987                0.0                0.0                0.0]
[               0.0 0.9999999999999999                0.0                0.0]
[               0.0                0.0 1.0000000000000013                0.0]
[               0.0                0.0                0.0 1.0000000000000007]
sage: T.parent()
Full MatrixSpace of 4 by 4 dense matrices over Complex Double Field
sage: all([T.zero_at(1.0e-12)[i,j] == 0 for i in range(4) for j in range(i)])
True
sage: (Q*T*Q.conjugate().transpose()-A).zero_at(1.0e-11)
```

```
[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]
```

Now totally over the reals. But with complex eigenvalues, the similar matrix may not be upper-triangular. But "at worst" there may be some 2×2 blocks on the diagonal which represent a pair of conjugate complex eigenvalues. These blocks will then just interrupt the zeros below the main diagonal. This example has a pair of these of the blocks.

```
sage: A = matrix(RDF, 4, 4, [[1, 0, -3, -1],
....:                         [4, -16, -7, 0],
....:                         [1, 21, 1, -2],
....:                         [26, -1, -2, 1]])
sage: Q, T = A.schur()
sage: (Q*Q.conjugate().transpose())  # tol 1e-12
[0.9999999999999994                0.0                0.0                0.0]
[               0.0 1.0000000000000013                0.0                0.0]
[               0.0                0.0 1.0000000000000004                0.0]
[               0.0                0.0                0.0 1.0000000000000016]
sage: all([T.zero_at(1.0e-12)[i,j] == 0 for i in range(4) for j in range(i)])
False
sage: all([T.zero_at(1.0e-12)[i,j] == 0 for i in range(4) for j in range(i-
↪1)])
True
sage: (Q*T*Q.conjugate().transpose()-A).zero_at(1.0e-11)
[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]
sage: sorted(T[0:2,0:2].eigenvalues() + T[2:4,2:4].eigenvalues())
[-5.710... - 8.382...*I, -5.710... + 8.382...*I, -0.789... - 2.336...*I, -0.
↪789... + 2.336...*I]
sage: sorted(A.eigenvalues())
[-5.710... - 8.382...*I, -5.710... + 8.382...*I, -0.789... - 2.336...*I, -0.
↪789... + 2.336...*I]
sage: abs(A.norm()-T.norm()) < 1e-12
True
```

Starting with complex numbers and requesting a result over the reals will never happen.

```
sage: A = matrix(CDF, 2, 2, [[2+I, -1+3*I], [5-4*I, 2-7*I]])
sage: A.schur(base_ring=RDF)
Traceback (most recent call last):
...
TypeError: unable to convert input matrix over CDF to a matrix over RDF
```

If theory predicts your matrix is real, but it contains some very small imaginary parts, you can specify the cutoff for "small" imaginary parts, then request the output as real matrices, and let the routine do the rest.

```
sage: A = matrix(RDF, 2, 2, [1, 1, -1, 0]) + matrix(CDF, 2, 2, [1.0e-14*I]*4)
sage: B = A.zero_at(1.0e-12)
sage: B.parent()
Full MatrixSpace of 2 by 2 dense matrices over Complex Double Field
sage: Q, T = B.schur(RDF)
sage: Q.parent()
Full MatrixSpace of 2 by 2 dense matrices over Real Double Field
sage: T.parent()
```

```
Full MatrixSpace of 2 by 2 dense matrices over Real Double Field
sage: Q.round(6)
[ 0.707107  0.707107]
[-0.707107  0.707107]
sage: T.round(6)
[ 0.5  1.5]
[-0.5  0.5]
sage: (Q*T*Q.conjugate().transpose()-B).zero_at(1.0e-11)
[0.0 0.0]
[0.0 0.0]
```

A Hermitian matrix has real eigenvalues, so the similar matrix will be upper-triangular. Furthermore, a Hermitian matrix is diagonalizable with respect to an orthonormal basis, composed of eigenvectors of the matrix. Here that basis is the set of columns of the unitary matrix.

```
sage: A = matrix(CDF, [[        52,   -9*I - 8,     6*I - 187,  -188*I + 2],
....:                  [   9*I - 8,         12,    -58*I + 59,    30*I + 42],
....:                  [-6*I - 187,  58*I + 59,          2677, 2264*I + 65],
....:                  [ 188*I + 2, -30*I + 42, -2264*I + 65,        2080]])
sage: Q, T = A.schur()
sage: T = T.zero_at(1.0e-12).change_ring(RDF)
sage: T.round(6)
[4680.13301        0.0        0.0        0.0]
[       0.0 102.715967        0.0        0.0]
[       0.0        0.0  35.039344        0.0]
[       0.0        0.0        0.0    3.11168]
sage: (Q*Q.conjugate().transpose()).zero_at(1e-12)  # tol 1e-12
[1.0000000000000004                0.0                0.0                0.0]
[               0.0 0.9999999999999989                0.0                0.0]
[               0.0                0.0 1.0000000000000002                0.0]
[               0.0                0.0                0.0 0.9999999999999992]
sage: (Q*T*Q.conjugate().transpose()-A).zero_at(1.0e-11)
[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]
```

Similarly, a real symmetric matrix has only real eigenvalues, and there is an orthonormal basis composed of eigenvectors of the matrix.

```
sage: A = matrix(RDF, [[ 1, -2, 5, -3],
....:                  [-2,  9, 1,  5],
....:                  [ 5,  1, 3 , 7],
....:                  [-3,  5, 7, -8]])
sage: Q, T = A.schur()
sage: Q.round(4)
[-0.3027  -0.751   0.576 -0.1121]
[  0.139 -0.3892 -0.2648  0.8713]
[ 0.4361   0.359  0.7599  0.3217]
[ -0.836  0.3945  0.1438  0.3533]
sage: T = T.zero_at(10^-12)
sage: all(abs(e) < 10^-4 for e in (T - diagonal_matrix(RDF, [-13.5698, -0.
↪8508, 7.7664, 11.6542])).list())
True
sage: (Q*Q.transpose())  # tol 1e-12
[0.9999999999999998                0.0                0.0                0.0]
[               0.0                1.0                0.0                0.0]
[               0.0                0.0 0.9999999999999998                0.0]
```

```
[                0.0                0.0                0.0 0.9999999999999996]
sage: (Q*T*Q.transpose()-A).zero_at(1.0e-11)
[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]
```

The results are cached, both as a real factorization and also as a complex factorization. This means the returned matrices are immutable.

```
sage: A = matrix(RDF, 2, 2, [[0, -1], [1, 0]])
sage: Qr, Tr = A.schur(base_ring=RDF)
sage: Qc, Tc = A.schur(base_ring=CDF)
sage: all([M.is_immutable() for M in [Qr, Tr, Qc, Tc]])
True
sage: Tr.round(6) != Tc.round(6)
True
```

AUTHOR:

•Rob Beezer (2011-03-31)

singular_values (*eps=None*)

Returns a sorted list of the singular values of the matrix.

INPUT:

•`eps` - default: `None` - the largest number which will be considered to be zero. May also be set to the string 'auto'. See the discussion below.

OUTPUT:

A sorted list of the singular values of the matrix, which are the diagonal entries of the "S" matrix in the SVD decomposition. As such, the values are real and are returned as elements of `RDF` . The list is sorted with larger values first, and since theory predicts these values are always positive, for a rank-deficient matrix the list should end in zeros (but in practice may not). The length of the list is the minimum of the row count and column count for the matrix.

The number of non-zero singular values will be the rank of the matrix. However, as a numerical matrix, it is impossible to control the difference between zero entries and very small non-zero entries. As an informed consumer it is up to you to use the output responsibly. We will do our best, and give you the tools to work with the output, but we cannot give you a guarantee.

With `eps` set to `None` you will get the raw singular values and can manage them as you see fit. You may also set `eps` to any positive floating point value you wish. If you set `eps` to 'auto' this routine will compute a reasonable cutoff value, based on the size of the matrix, the largest singular value and the smallest nonzero value representable by the 53-bit precision values used. See the discussion at page 268 of [Wat2010].

See the examples for a way to use the "verbose" facility to easily watch the zero cutoffs in action.

ALGORITHM:

The singular values come from the SVD decomposition computed by SciPy/NumPy.

EXAMPLES:

Singular values close to zero have trailing digits that may vary on different hardware. For exact matrices, the number of non-zero singular values will equal the rank of the matrix. So for some of the doctests we round the small singular values that ideally would be zero, to control the variability across hardware.

This matrix has a determinant of one. A chain of two or three theorems implies the product of the singular values must also be one.

```
sage: A = matrix(QQ, [[ 1,  0,  0,  0,  0,  1,  3],
....:                 [-2,  1,  1, -2,  0, -4,  0],
....:                 [ 1,  0,  1, -4, -6, -3,  7],
....:                 [-2,  2,  1,  1,  7,  1, -1],
....:                 [-1,  0, -1,  5,  8,  4, -6],
....:                 [ 4, -2, -2,  1, -3,  0,  8],
....:                 [-2,  1,  0,  2,  7,  3, -4]])
sage: A.determinant()
1
sage: B = A.change_ring(RDF)
sage: sv = B.singular_values(); sv  # tol 1e-12
[20.523980658874265, 8.486837028536643, 5.86168134845073, 2.4429165899286978,␣
↪0.5831970144724045, 0.26933287286576313, 0.0025524488076110402]
sage: prod(sv)  # tol 1e-12
0.9999999999999525
```

An exact matrix that is obviously not of full rank, and then a computation of the singular values after conversion to an approximate matrix.

```
sage: A = matrix(QQ, [[1/3, 2/3, 11/3],
....:                 [2/3, 1/3,  7/3],
....:                 [2/3, 5/3, 27/3]])
sage: A.rank()
2
sage: B = A.change_ring(CDF)
sage: sv = B.singular_values()
sage: sv[0:2]
[10.1973039..., 0.487045871...]
sage: sv[2] < 1e-14
True
```

A matrix of rank 3 over the complex numbers.

```
sage: A = matrix(CDF, [[46*I - 28, -47*I - 50, 21*I + 51, -62*I - 782, 13*I +␣
↪22],
....:                  [35*I - 20, -32*I - 46, 18*I + 43, -57*I - 670, 7*I +␣
↪3],
....:                  [22*I - 13, -23*I - 23, 9*I + 24, -26*I - 347, 7*I +␣
↪13],
....:                  [-44*I + 23, 41*I + 57, -19*I - 54, 60*I + 757, -11*I -
↪ 9],
....:                  [30*I - 18, -30*I - 34, 14*I + 34, -42*I - 522, 8*I +␣
↪12]])
sage: sv = A.singular_values()
sage: sv[0:3]  # tol 1e-14
[1440.7336659952966, 18.404403413369227, 6.839707797136151]
sage: (sv[3] < 10^-13) or sv[3]
True
sage: (sv[4] < 10^-14) or sv[4]
True
```

A full-rank matrix that is ill-conditioned. We use this to illustrate ways of using the various possibilities for `eps`, including one that is ill-advised. Notice that the automatically computed cutoff gets this (difficult) example slightly wrong. This illustrates the impossibility of any automated process always getting this right. Use with caution and judgement.

```
sage: entries = [1/(i+j+1) for i in range(12) for j in range(12)]
sage: B = matrix(QQ, 12, 12, entries)
sage: B.rank()
12
sage: A = B.change_ring(RDF)
sage: A.condition() > 1.59e16 or A.condition()
True

sage: A.singular_values(eps=None)  # abs tol 7e-16
[1.7953720595619975, 0.38027524595503703, 0.04473854875218107, 0.
↪0037223122378911614, 0.0002330890890217751, 1.116335748323284e-05, 4.
↪082376110397296e-07, 1.1228610675717613e-08, 2.2519645713496478e-10, 3.
↪1113486853814003e-12, 2.6500422260778388e-14, 9.87312834948426e-17]
sage: A.singular_values(eps='auto')  # abs tol 7e-16
[1.7953720595619975, 0.38027524595503703, 0.04473854875218107, 0.
↪0037223122378911614, 0.0002330890890217751, 1.116335748323284e-05, 4.
↪082376110397296e-07, 1.1228610675717613e-08, 2.2519645713496478e-10, 3.
↪1113486853814003e-12, 2.6500422260778388e-14, 0.0]
sage: A.singular_values(eps=1e-4)  # abs tol 7e-16
[1.7953720595619975, 0.38027524595503703, 0.04473854875218107, 0.
↪0037223122378911614, 0.0002330890890217751, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.
↪0]
```

With Sage's "verbose" facility, you can compactly see the cutoff at work. In any application of this routine, or those that build upon it, it would be a good idea to conduct this exercise on samples. We also test here that all the values are returned in RDF since singular values are always real.

```
sage: A = matrix(CDF, 4, range(16))
sage: set_verbose(1)
sage: sv = A.singular_values(eps='auto'); sv
verbose 1 (<module>) singular values,
smallest-non-zero:cutoff:largest-zero,
2.2766...:6.2421...e-14:...
[35.13996365902..., 2.27661020871472..., 0.0, 0.0]
sage: set_verbose(0)

sage: all([s in RDF for s in sv])
True
```

AUTHOR:

•Rob Beezer - (2011-02-18)

solve_left (*b*)

Solve the vector equation `x*A = b` for a nonsingular `A`.

INPUT:

•`self` - a square matrix that is nonsingular (of full rank).

•`b` - a vector of the correct size. Elements of the vector must coerce into the base ring of the coefficient matrix. In particular, if `b` has entries from `CDF` then `self` must have `CDF` as its base ring.

OUTPUT:

The unique solution `x` to the matrix equation `x*A = b`, as a vector over the same base ring as `self`.

ALGORITHM:

Uses the `solve()` routine from the SciPy `scipy.linalg` module, after taking the tranpose of the coefficient matrix.

EXAMPLES:

Over the reals.

```
sage: A = matrix(RDF, 3,3, [1,2,5,7.6,2.3,1,1,2,-1]); A
[ 1.0  2.0  5.0]
[ 7.6  2.3  1.0]
[ 1.0  2.0 -1.0]
sage: b = vector(RDF,[1,2,3])
sage: x = A.solve_left(b); x.zero_at(1e-17) # fix noisy zeroes
(0.666666666..., 0.0, 0.333333333...)
sage: x.parent()
Vector space of dimension 3 over Real Double Field
sage: x*A  # tol 1e-14
(0.9999999999999999, 1.9999999999999998, 3.0)
```

Over the complex numbers.

```
sage: A = matrix(CDF, [[      0, -1 + 2*I,  1 - 3*I,        I],
....:                  [2 + 4*I, -2 + 3*I, -1 + 2*I,   -1 - I],
....:                  [  2 + I,    1 - I,       -1,        5],
....:                  [    3*I,   -1 - I,   -1 + I,   -3 + I]])
sage: b = vector(CDF, [2 -3*I, 3, -2 + 3*I, 8])
sage: x = A.solve_left(b); x
(-1.55765124... - 0.644483985...*I, 0.183274021... + 0.286476868...*I, 0.
↪270818505... + 0.246619217...*I, -1.69003558... - 0.828113879...*I)
sage: x.parent()
Vector space of dimension 4 over Complex Double Field
sage: abs(x*A - b) < 1e-14
True
```

The vector of constants, `b` , can be given in a variety of forms, so long as it coerces to a vector over the same base ring as the coefficient matrix.

```
sage: A=matrix(CDF, 5, [1/(i+j+1) for i in range(5) for j in range(5)])
sage: A.solve_left([1]*5)  # tol 1e-11
(5.0, -120.0, 630.0, -1120.0, 630.0)
```

solve_right (*b*)

Solve the vector equation `A*x = b` for a nonsingular `A` .

INPUT:

- `self` - a square matrix that is nonsigular (of full rank).
- `b` - a vector of the correct size. Elements of the vector must coerce into the base ring of the coefficient matrix. In particular, if `b` has entries from `CDF` then `self` must have `CDF` as its base ring.

OUTPUT:

The unique solution `x` to the matrix equation `A*x = b` , as a vector over the same base ring as `self` .

ALGORITHM:

Uses the `solve()` routine from the SciPy `scipy.linalg` module.

EXAMPLES:

Over the reals.

```
sage: A = matrix(RDF, 3,3, [1,2,5,7.6,2.3,1,1,2,-1]); A
[ 1.0  2.0  5.0]
```

```
[ 7.6  2.3  1.0]
[ 1.0  2.0 -1.0]
sage: b = vector(RDF,[1,2,3])
sage: x = A.solve_right(b); x  # tol 1e-14
(-0.1136950904392765, 1.3901808785529717, -0.33333333333333337)
sage: x.parent()
Vector space of dimension 3 over Real Double Field
sage: A*x  # tol 1e-14
(1.0, 1.9999999999999996, 3.0000000000000004)
```

Over the complex numbers.

```
sage: A = matrix(CDF, [[      0, -1 + 2*I,  1 - 3*I,        I],
....:                  [2 + 4*I, -2 + 3*I, -1 + 2*I,   -1 - I],
....:                  [  2 + I,    1 - I,       -1,        5],
....:                  [    3*I,   -1 - I,   -1 + I,   -3 + I]])
sage: b = vector(CDF, [2 -3*I, 3, -2 + 3*I, 8])
sage: x = A.solve_right(b); x
(1.96841637... - 1.07606761...*I, -0.614323843... + 1.68416370...*I, 0.
↪0733985765... + 1.73487544...*I, -1.6018683... + 0.524021352...*I)
sage: x.parent()
Vector space of dimension 4 over Complex Double Field
sage: abs(A*x - b) < 1e-14
True
```

The vector of constants, `b` , can be given in a variety of forms, so long as it coerces to a vector over the same base ring as the coefficient matrix.

```
sage: A=matrix(CDF, 5, [1/(i+j+1) for i in range(5) for j in range(5)])
sage: A.solve_right([1]*5)  # tol 1e-11
(5.0, -120.0, 630.0, -1120.0, 630.0)
```

transpose ( )

Return the transpose of this matrix, without changing self.

EXAMPLES:

```
sage: m = matrix(RDF,2,3,range(6)); m
[0.0 1.0 2.0]
[3.0 4.0 5.0]
sage: m2 = m.transpose()
sage: m[0,0] = 2
sage: m2           #note that m2 hasn't changed
[0.0 3.0]
[1.0 4.0]
[2.0 5.0]
```

`.T` is a convenient shortcut for the transpose:

```
sage: m.T
[2.0 3.0]
[1.0 4.0]
[2.0 5.0]

sage: m = matrix(RDF,0,3); m
[]
sage: m.transpose()
[]
```

```
sage: m.transpose().parent()
Full MatrixSpace of 3 by 0 dense matrices over Real Double Field
```

zero_at (*eps*)

Returns a copy of the matrix where elements smaller than or equal to `eps` are replaced with zeroes. For complex matrices, the real and imaginary parts are considered individually.

This is useful for modifying output from algorithms which have large relative errors when producing zero elements, e.g. to create reliable doctests.

INPUT:

- `eps` - Cutoff value

OUTPUT:

A modified copy of the matrix.

EXAMPLES:

```
sage: a = matrix(CDF, [[1, 1e-4r, 1+1e-100jr], [1e-8+3j, 0, 1e-58r]])
sage: a
[           1.0          0.0001 1.0 + 1e-100*I]
[ 1e-08 + 3.0*I             0.0           1e-58]
sage: a.zero_at(1e-50)
[          1.0        0.0001           1.0]
[1e-08 + 3.0*I           0.0           0.0]
sage: a.zero_at(1e-4)
[  1.0    0.0    1.0]
[3.0*I    0.0    0.0]
```

CHAPTER

TWENTY

DENSE MATRICES OVER THE REAL DOUBLE FIELD USING NUMPY

EXAMPLES:

```
sage: b=Mat(RDF,2,3).basis()
sage: b[0]
[1.0 0.0 0.0]
[0.0 0.0 0.0]
```

We deal with the case of zero rows or zero columns:

```
sage: m = MatrixSpace(RDF,0,3)
sage: m.zero_matrix()
[]
```

AUTHORS:

- Jason Grout (2008-09): switch to NumPy backend, factored out the Matrix_double_dense class
- Josh Kantor
- William Stein: many bug fixes and touch ups.

class sage.matrix.matrix_real_double_dense.**Matrix_real_double_dense**

Bases: *sage.matrix.matrix_double_dense.Matrix_double_dense*

Class that implements matrices over the real double field. These are supposed to be fast matrix operations using C doubles. Most operations are implemented using numpy which will call the underlying BLAS on the system.

EXAMPLES:

```
sage: m = Matrix(RDF, [[1,2],[3,4]])
sage: m**2
[ 7.0 10.0]
[15.0 22.0]
sage: n = m^(-1); n     # rel tol 1e-15
[-1.9999999999999996  0.9999999999999998]
[ 1.4999999999999998 -0.4999999999999999]
```

To compute eigenvalues the use the functions left_eigenvectors or right_eigenvectors

```
sage: p,e = m.right_eigenvectors()
```

the result of eigen is a pair (p,e), where p is a list of eigenvalues and the e is a matrix whose columns are the eigenvectors.

To solve a linear system Ax = b where A = [[1,2],[3,4]] and b = [5,6].

```
sage: b = vector(RDF,[5,6])
sage: m.solve_right(b)  # rel tol 1e-15
(-3.9999999999999987, 4.499999999999999)
```

See the commands qr, lu, and svd for QR, LU, and singular value decomposition.

CHAPTER

TWENTYONE

DENSE MATRICES OVER THE COMPLEX DOUBLE FIELD USING NUMPY

EXAMPLES:

```
sage: b=Mat(CDF,2,3).basis()
sage: b[0]
[1.0 0.0 0.0]
[0.0 0.0 0.0]
```

We deal with the case of zero rows or zero columns:

```
sage: m = MatrixSpace(CDF,0,3)
sage: m.zero_matrix()
[]
```

AUTHORS:

- Jason Grout (2008-09): switch to NumPy backend
- Josh Kantor
- William Stein: many bug fixes and touch ups.

class sage.matrix.matrix_complex_double_dense.**Matrix_complex_double_dense**

Bases: *sage.matrix.matrix_double_dense.Matrix_double_dense*

Class that implements matrices over the real double field. These are supposed to be fast matrix operations using C doubles. Most operations are implemented using numpy which will call the underlying BLAS on the system.

EXAMPLES:

```
sage: m = Matrix(CDF, [[1,2*I],[3+I,4]])
sage: m**2
[-1.0 + 6.0*I       10.0*I]
[15.0 + 5.0*I 14.0 + 6.0*I]
sage: n= m^(-1); n  # abs tol 1e-15
[  0.3333333333333333 + 0.3333333333333333*I 0.16666666666666669 - 0.
↪16666666666666666*I]
[-0.16666666666666666 - 0.3333333333333333*I 0.08333333333333331 + 0.
↪08333333333333333*I]
```

To compute eigenvalues the use the functions `left_eigenvectors` or `right_eigenvectors`:

```
sage: p,e = m.right_eigenvectors()
```

the result of eigen is a pair (p,e), where p is a list of eigenvalues and the e is a matrix whose columns are the eigenvectors.

To solve a linear system Ax = b where A = [[1,2] and b = [5,6] [3,4]]

```
sage: b = vector(CDF,[5,6])
sage: m.solve_right(b)  # abs tol 1e-14
(2.6666666666666665 + 0.6666666666666669*I, -0.3333333333333333 - 1.
↪1666666666666667*I)
```

See the commands qr, lu, and svd for QR, LU, and singular value decomposition.

CHAPTER TWENTYTWO

ARBITRARY PRECISION COMPLEX BALL MATRICES USING ARB

AUTHORS:

- Clemens Heuberger (2014-10-25): Initial version.

This is a rudimentary binding to the Arb library; it may be useful to refer to its documentation for more details.

class sage.matrix.matrix_complex_ball_dense.**Matrix_complex_ball_dense**

Bases: *sage.matrix.matrix_dense.Matrix_dense*

Matrix over a complex ball field. Implemented using the `acb_mat` type of the Arb library.

EXAMPLES:

```
sage: MatrixSpace(CBF, 3)(2)
[2.000000000000000                 0                 0]
[                0 2.000000000000000                 0]
[                0                 0 2.000000000000000]
sage: matrix(CBF, 1, 3, [1, 2, -3])
[ 1.000000000000000  2.000000000000000 -3.000000000000000]
```

CHAPTER TWENTYTHREE

DENSE MATRICES OVER UNIVARIATE POLYNOMIALS OVER FIELDS

This implementation inherits from Matrix_generic_dense, i.e., it is not optimized for speed but only some methods were added.

AUTHORS:

- Kwankyu Lee (2016-12-15): Initial version with code moved from other files.

class sage.matrix.matrix_polynomial_dense.**Matrix_polynomial_dense**

Bases: *sage.matrix.matrix_generic_dense.Matrix_generic_dense*

Dense matrix over a univariate polynomial ring over a field.

is_weak_popov()

Return True if the matrix is in weak Popov form.

OUTPUT:

A matrix over a polynomial ring is in weak Popov form if all leading positions are different [MS2003]. A leading position is the position i in a row with the highest degree; in case of tie, the maximal i is used (i.e. furthest to the right).

EXAMPLES:

A matrix with the same leading position in two rows is not in weak Popov form:

```
sage: PF = PolynomialRing(GF(2^12,'a'),'x')
sage: A = matrix(PF,3,[x,   x^2, x^3,\
                       x^2, x^2, x^2,\
                       x^3, x^2, x    ])
sage: A.is_weak_popov()
False
```

If a matrix has different leading positions, it is in weak Popov form:

```
sage: B = matrix(PF,3,[1,    1,   x^3,\
                       x^2,  1,   1,\
                       1,x^  2,   1  ])
sage: B.is_weak_popov()
True
```

Weak Popov form is not restricted to square matrices:

```
sage: PF = PolynomialRing(GF(7),'x')
sage: D = matrix(PF,2,4,[x^2+1, 1, 2, x,\
                         3*x+2, 0, 0, 0 ])
sage: D.is_weak_popov()
False
```

Even a matrix with more rows than columns can still be in weak Popov form:

```
sage: E = matrix(PF,4,2,[4*x^3+x, x^2+5*x+2,\
                         0,       0,\
                         4,       x,\
                         0,       0          ])
sage: E.is_weak_popov()
True
```

A matrix with fewer columns than non-zero rows is never in weak Popov form:

```
sage: F = matrix(PF,3,2,[x^2,    x,\
                         x^3+2, x,\
                         4,      5])
sage: F.is_weak_popov()
False
```

See also:

•weak_popov_form

AUTHOR:

•David Moedinger (2014-07-30)

row_reduced_form (*transformation=None*, *shifts=None*)

Return a row reduced form of this matrix.

A matrix M is row reduced if the (row-wise) leading term matrix has the same rank as M. The (row-wise) leading term matrix of a polynomial matrix M is the matrix over k whose (i, j)'th entry is the x^{d_i} coefficient of $M[i, j]$, where d_i is the greatest degree among polynomials in the i'th row of M_0.

A row reduced form is non-canonical so a given matrix has many row reduced forms.

INPUT:

•transformation – (default: False). If this True , the transformation matrix U will be returned as well: this is an invertible matrix over $k[x]$ such that self equals UW, where W is the output matrix.

•shifts – (default: None) A tuple or list of integers $s_1, \ldots, s_n$, where n is the number of columns of the matrix. If given, a "shifted row reduced form" is computed, i.e. such that the matrix $A\,\mathrm{diag}(x^{s_1}, \ldots, x^{s_n})$ is row reduced, where diag denotes a diagonal matrix.

OUTPUT:

•W – a row reduced form of this matrix.

EXAMPLES:

```
sage: R.<t> = GF(3)['t']
sage: M = matrix([[(t-1)^2],[(t-1)]])
sage: M.row_reduced_form()
[    0]
[t + 2]
```

If the matrix is an $n \times 1$ matrix with at least one non-zero entry, W has a single non-zero entry and that entry is a scalar multiple of the greatest-common-divisor of the entries of the matrix:

```
sage: M1 = matrix([[t*(t-1)*(t+1)],[t*(t-2)*(t+2)],[t]])
sage: output1 = M1.row_reduced_form()
sage: output1
[0]
[0]
[t]
```

The following is the first half of example 5 in [Hes2002] *except* that we have transposed the matrix; [Hes2002] uses column operations and we use row:

```
sage: R.<t> = QQ['t']
sage: M = matrix([[t^3 - t,t^2 - 2],[0,t]]).transpose()
sage: M.row_reduced_form()
[      t    -t^2]
[t^2 - 2       t]
```

The next example demonstrates what happens when the matrix is a zero matrix:

```
sage: R.<t> = GF(5)['t']
sage: M = matrix(R, 2, [0,0,0,0])
sage: M.row_reduced_form()
[0 0]
[0 0]
```

In the following example, the original matrix is already row reduced, but the output is a different matrix. This is because currently this method simply computes a weak Popov form, which is always also a row reduced matrix (see `weak_popov_form()`). This behavior is likely to change when a faster algorithm designed specifically for row reduced form is implemented in Sage:

```
sage: R.<t> = QQ['t']
sage: M = matrix([[t,t,t],[0,0,t]]); M
[t t t]
[0 0 t]
sage: M.row_reduced_form()
[ t  t  t]
[-t -t  0]
```

The last example shows the usage of the transformation parameter:

```
sage: Fq.<a> = GF(2^3)
sage: Fx.<x> = Fq[]
sage: A = matrix(Fx,[[x^2+a,x^4+a],[x^3,a*x^4]])
sage: W,U = A.row_reduced_form(transformation=True);
sage: W,U
(
[          x^2 + a           x^4 + a]  [1 0]
[x^3 + a*x^2 + a^2               a^2], [a 1]
)
sage: U*W == A
True
sage: U.is_invertible()
True
```

weak_popov_form (*transformation=False*, *shifts=None*)

Return a weak Popov form of this matrix.

A matrix is in weak Popov form if the leading positions of the nonzero rows are all different. The leading position of a row is the right-most position whose entry has the maximal degree in the row.

The weak Popov form is non-canonical, so an input matrix have many weak Popov forms.

INPUT:

- `transformation` – (default: `False`) If `True`, the transformation matrix is returned together with the weak Popov form.
- `shifts` – (default: `None`) A tuple or list of integers $s_1, \ldots, s_n$, where n is the number of columns of the matrix. If given, a "shifted weak Popov form" is computed, i.e. such that the matrix $A\,\mathrm{diag}(x^{s_1}, \ldots, x^{s_n})$ is in weak Popov form, where diag denotes a diagonal matrix.

ALGORITHM:

This method implements the Mulders-Storjohann algorithm of [MS2003].

EXAMPLES:

```
sage: F.<a> = GF(2^4,'a')
sage: PF.<x> = F[]
sage: A = matrix(PF,[[1,   a*x^17 + 1 ],\
                     [0,   a*x^11 + a^2*x^7 + 1 ]])
sage: M, U = A.weak_popov_form(transformation=True)
sage: U * A == M
True
sage: M.is_weak_popov()
True
sage: U.is_invertible()
True
```

A zero matrix will return itself:

```
sage: Z = matrix(PF,5,3)
sage: Z.weak_popov_form()
[0 0 0]
[0 0 0]
[0 0 0]
[0 0 0]
[0 0 0]
```

Shifted weak popov form is computed if `shifts` is given:

```
sage: PF.<x> = QQ[]
sage: A = matrix(PF,3,[x,    x^2, x^3,\
                       x^2, x^1, 0,\
                       x^3, x^3, x^3])
sage: A.weak_popov_form()
[        x       x^2       x^3]
[      x^2         x         0]
[  x^3 - x x^3 - x^2         0]
sage: H,U = A.weak_popov_form(transformation=True, shifts=[16,8,0])
sage: H
[               x               x^2                x^3]
[               0          -x^2 + x        -x^4 + x^3]
[               0                 0 -x^5 + x^4 + x^3]
sage: U * A == H
True
```

AUTHOR:

- Johan Rosenkilde (2017-02-07)

See also:

`is_weak_popov`

CHAPTER

TWENTYFOUR

DENSE MATRICES OVER MULTIVARIATE POLYNOMIALS OVER FIELDS

This implementation inherits from Matrix_generic_dense, i.e. it is not optimized for speed only some methods were added.

AUTHOR:

- Martin Albrecht <malb@informatik.uni-bremen.de>

class sage.matrix.matrix_mpolynomial_dense.**Matrix_mpolynomial_dense**
Bases: *sage.matrix.matrix_generic_dense.Matrix_generic_dense*

Dense matrix over a multivariate polynomial ring over a field.

determinant (*algorithm=None*)
Return the determinant of this matrix

INPUT:

- algorithm – ignored

EXAMPLES:

We compute the determinant of the arbitrary $3x3$ matrix:

```
sage: R = PolynomialRing(QQ, 9, 'x')
sage: A = matrix(R, 3, R.gens())
sage: A
[x0 x1 x2]
[x3 x4 x5]
[x6 x7 x8]
sage: A.determinant()
-x2*x4*x6 + x1*x5*x6 + x2*x3*x7 - x0*x5*x7 - x1*x3*x8 + x0*x4*x8
```

We check if two implementations agree on the result:

```
sage: R.<x,y> = QQ[]
sage: C = random_matrix(R, 2, 2, terms=2)
sage: C
[-6/5*x*y - y^2 -6*y^2 - 1/4*y]
[  -1/3*x*y - 3         x*y - x]
sage: C.determinant()
-6/5*x^2*y^2 - 3*x*y^3 + 6/5*x^2*y + 11/12*x*y^2 - 18*y^2 - 3/4*y

sage: C.change_ring(R.change_ring(QQbar)).det()
(-6/5)*x^2*y^2 + (-3)*x*y^3 + 6/5*x^2*y + 11/12*x*y^2 + (-18)*y^2 + (-3/4)*y
```

Finally, we check whether the Singular interface is working:

```
sage: R.<x,y> = RR[]
sage: C = random_matrix(R, 2, 2, terms=2)
sage: C
[0.368965517352886*y^2 + 0.425700773972636*x  -0.800362171389760*y^2 - 0.
↪807635502485287]
[  0.706173539423122*y^2 - 0.915986060298440     0.897165181570476*y + 0.
↪107903328188376]
sage: C.determinant()
0.565194587390682*y^4 + 0.331023015369146*y^3 + 0.381923912175852*x*y - 0.
↪122977163520282*y^2 + 0.0459345303240150*x - 0.739782862078649
```

ALGORITHM: Calls Singular, libSingular or native implementation.

echelon_form (*algorithm='row_reduction'*, ***kwds*)

Return an echelon form of `self` using chosen algorithm.

By default only a usual row reduction with no divisions or column swaps is returned.

If Gauss-Bareiss algorithm is chosen, column swaps are recorded and can be retrieved via *swapped_columns()* .

INPUT:

- `algorithm` – string, which algorithm to use (default: 'row_reduction'). Valid options are:
  - `'row_reduction'` (default) – reduce as far as possible, only divide by constant entries
  - `'frac'` – reduced echelon form over fraction field
  - `'bareiss'` – fraction free Gauss-Bareiss algorithm with column swaps

OUTPUT:

The row echelon form of A depending on the chosen algorithm, as an immutable matrix. Note that `self` is *not* changed by this command. Use `A.echelonize()`` to change A in place.

EXAMPLES:

```
sage: P.<x,y> = PolynomialRing(GF(127), 2)
sage: A = matrix(P, 2, 2, [1, x, 1, y])
sage: A
[1 x]
[1 y]
sage: A.echelon_form()
[      1       x]
[      0 -x + y]
```

The reduced row echelon form over the fraction field is as follows:

```
sage: A.echelon_form('frac') # over fraction field
[1 0]
[0 1]
```

Alternatively, the Gauss-Bareiss algorithm may be chosen:

```
sage: E = A.echelon_form('bareiss'); E
[    1     y]
[    0 x - y]
```

After the application of the Gauss-Bareiss algorithm the swapped columns may inspected:

```
sage: E.swapped_columns(), E.pivots()
((0, 1), (0, 1))
sage: A.swapped_columns(), A.pivots()
(None, (0, 1))
```

Another approach is to row reduce as far as possible:

```
sage: A.echelon_form('row_reduction')
[      1       x]
[      0 -x + y]
```

echelonize (*algorithm='row_reduction'*, ***kwds*)

Transform self into a matrix in echelon form over the same base ring as `self` .

If Gauss-Bareiss algorithm is chosen, column swaps are recorded and can be retrieved via *swapped_columns()* .

INPUT:

- `algorithm` – string, which algorithm to use. Valid options are:
  - `'row_reduction'` – reduce as far as possible, only divide by constant entries
  - `'bareiss'` – fraction free Gauss-Bareiss algorithm with column swaps

EXAMPLES:

```
sage: P.<x,y> = PolynomialRing(QQ, 2)
sage: A = matrix(P, 2, 2, [1/2, x, 1, 3/4*y+1])
sage: A
[      1/2           x]
[        1 3/4*y + 1]

sage: B = copy(A)
sage: B.echelonize('bareiss'); B
[              1       3/4*y + 1]
[              0 x - 3/8*y - 1/2]

sage: B = copy(A)
sage: B.echelonize('row_reduction'); B
[                1               2*x]
[                0 -2*x + 3/4*y + 1]

sage: P.<x,y> = PolynomialRing(QQ, 2)
sage: A = matrix(P,2,3,[2,x,0,3,y,1]); A
[2 x 0]
[3 y 1]

sage: E = A.echelon_form('bareiss'); E
[1 3 y]
[0 2 x]
sage: E.swapped_columns()
(2, 0, 1)
sage: A.pivots()
(0, 1, 2)
```

pivots ( )

Return the pivot column positions of this matrix as a list of integers.

This returns a list, of the position of the first nonzero entry in each row of the echelon form.

OUTPUT:

A list of Python ints.

EXAMPLES:

```
sage: matrix([PolynomialRing(GF(2), 2, 'x').gen()]).pivots()
(0,)
sage: K = QQ['x,y']
sage: x, y = K.gens()
sage: m = matrix(K, [(-x, 1, y, x - y), (-x*y, y, y^2 - 1, x*y - y^2 + x), (-
↪x*y + x, y - 1, y^2 - y - 2, x*y - y^2 + x + y)])
sage: m.pivots()
(0, 2)
sage: m.rank()
2
```

swapped_columns ( )

Return which columns were swapped during the Gauss-Bareiss reduction

OUTPUT:

Return a tuple representing the column swaps during the last application of the Gauss-Bareiss algorithm (see *echelon_form()* for details).

The tuple as length equal to the rank of self and the value at the i-th position indicates the source column which was put as the i-th column.

If no Gauss-Bareiss reduction was performed yet, None is returned.

EXAMPLES:

```
sage: R.<x,y> = QQ[]
sage: C = random_matrix(R, 2, 2, terms=2)
sage: C.swapped_columns()
sage: E = C.echelon_form('bareiss')
sage: E.swapped_columns()
(0, 1)
```

CHAPTER

TWENTYFIVE

OPERATION TABLES

This module implements general operation tables, which are very matrix-like.

class `sage.matrix.operation_table.`**`OperationTable`** (*S*, *operation*, *names='letters'*, *elements=None*)

Bases: `sage.structure.sage_object.SageObject`

An object that represents a binary operation as a table.

Primarily this object is used to provide a `multiplication_table()` for objects in the category of magmas (monoids, groups, ...) and `addition_table()` for objects in the category of commutative additive magmas (additive monoids, groups, ...).

INPUT:

- `S` - a finite algebraic structure (or finite iterable)
- **`operation` - a function of two variables that accepts pairs** of elements from `S` . A natural source of such functions is the Python `operator` module, and in particular `operator.add()` and `operator.mul()` . This may also be a function defined with `lambda` or `def`.
- `names` - (default: `'letters'`) The type of names used, values are:
  - `'letters'` - lowercase ASCII letters are used for a base 26 representation of the elements' positions in the list given by *`column_keys()`* , padded to a common width with leading 'a's.
  - `'digits'` - base 10 representation of the elements' positions in the list given by *`column_keys()`* , padded to a common width with leading zeros.
  - `'elements'` - the string representations of the elements themselves.
  - a list - a list of strings, where the length of the list equals the number of elements.
- `elements` - (default: `None`) A list of elements of `S` , in forms that can be coerced into the structure, eg. their string representations. This may be used to impose an alternate ordering on the elements of S`, perhaps when this is used in the context of a particular structure. The default is to use whatever ordering the `S.list()` method returns. *elements*` can also be a subset which is closed under the operation, useful perhaps when the set is infinite.

OUTPUT: An object with methods that abstracts multiplication tables, addition tables, Cayley tables, etc. It should be general enough to be useful for any finite algebraic structure whose elements can be combined with a binary operation. This is not necessarily meant be constructed directly, but instead should be useful for constructing operation tables of various algebraic structures that have binary operations.

EXAMPLES:

In it's most basic use, the table needs a structure and an operation:

```
sage: from sage.matrix.operation_table import OperationTable
sage: G=SymmetricGroup(3)
sage: OperationTable(G, operation = operator.mul)
*  a b c d e f
 +------------
a| a b c d e f
b| b a f e d c
c| c e d a f b
d| d f a c b e
e| e c b f a d
f| f d e b c a
```

With two operations present, we can specify which operation we want:

```
sage: from sage.matrix.operation_table import OperationTable
sage: R=Integers(6)
sage: OperationTable(R, operation=operator.add)
+  a b c d e f
 +------------
a| a b c d e f
b| b c d e f a
c| c d e f a b
d| d e f a b c
e| e f a b c d
f| f a b c d e
```

The default symbol set for elements is lowercase ASCII letters, which take on a base 26 flavor for structures with more than 26 elements.

```
sage: from sage.matrix.operation_table import OperationTable
sage: G=DihedralGroup(14)
sage: OperationTable(G, operator.mul, names='letters')
 *  aa ab ac ad ae af ag ah ai aj ak al am an ao ap aq ar as at au av aw ax ay az␣
↪ba bb
  +------------------------------------------------------------------------------
↪-----
aa| aa ab ac ad ae af ag ah ai aj ak al am an ao ap aq ar as at au av aw ax ay az␣
↪ba bb
ab| ab aa ae ah ac aj ak ad am af ag ar ai ap at an av al aw ao az aq as ba bb au␣
↪ax ay
ac| ac ad af ai ab ag al aa an ae aj aq ah ao au am as ak ax ap ay ar av bb ba at␣
↪aw az
ad| ad ac ab aa af ae aj ai ah ag al ak an am ap ao ar aq av au at as ax aw az ay␣
↪bb ba
ae| ae ah aj am aa ak ar ab ap ac af av ad at az ai aw ag ba an bb al aq ay ax ao␣
↪as au
af| af ai ag an ad al aq ac ao ab ae as aa au ay ah ax aj bb am ba ak ar az aw ap␣
↪av at
ag| ag an al ao ai aq as af au ad ab ax ac ay ba aa bb ae az ah aw aj ak at av am␣
↪ar ap
ah| ah ae aa ab aj ac af am ad ak ar ag ap ai an at al av aq az ao aw ba as au bb␣
↪ay ax
ai| ai af ad ac ag ab ae an aa al aq aj ao ah am au ak as ar ay ap ax bb av at ba␣
↪az aw
aj| aj am ak ap ah ar av ae at aa ac aw ab az bb ad ba af ay ai ax ag al au as an␣
↪aq ao
ak| ak ap ar at am av aw aj az ah aa ba ae bb ax ab ay ac au ad as af ag ao aq ai␣
↪al an
```

```
al| al ao aq au an as ax ag ay ai ad bb af ba aw ac az ab at aa av ae aj ap ar ah␣
↪ak am
am| am aj ah ae ak aa ac ap ab ar av af at ad ai az ag aw al bb an ba ay aq ao ax␣
↪au as
an| an ag ai af al ad ab ao ac aq as ae au aa ah ay aj ax ak ba am bb az ar ap aw␣
↪at av
ao| ao al an ag aq ai ad au af as ax ab ay ac aa ba ae bb aj aw ah az at ak am av␣
↪ap ar
ap| ap ak am aj ar ah aa at ae av aw ac az ab ad bb af ba ag ax ai ay au al an as␣
↪ao aq
aq| aq au as ay ao ax bb al ba an ai az ag aw av af at ad ap ac ar ab ae am ak aa␣
↪aj ah
ar| ar at av az ap aw ba ak bb am ah ay aj ax as ae au aa ao ab aq ac af an al ad␣
↪ag ai
as| as ay ax ba au bb az aq aw ao an at al av ar ag ap ai am af ak ad ab ah aj ac␣
↪ae aa
at| at ar ap ak av am ah az aj aw ba aa bb ae ab ax ac ay af as ad au ao ag ai aq␣
↪an al
au| au aq ao al as an ai ay ag ax bb ad ba af ac aw ab az ae av aa at ap aj ah ar␣
↪am ak
av| av az aw bb at ba ay ar ax ap am au ak as aq aj ao ah an ae al aa ac ai ag ab␣
↪af ad
aw| aw bb ba ax az ay au av as at ap ao ar aq al ak an am ai aj ag ah aa ad af ae␣
↪ac ab
ax| ax ba bb aw ay az at as av au ao ap aq ar ak al am an ah ag aj ai ad aa ae af␣
↪ab ac
ay| ay as au aq ax ao an ba al bb az ai aw ag af av ad at ab ar ac ap am ae aa ak␣
↪ah aj
az| az av at ar aw ap am bb ak ba ay ah ax aj ae as aa au ac aq ab ao an af ad al␣
↪ai ag
ba| ba ax ay as bb au ao aw aq az at an av al ag ar ai ap ad ak af am ah ab ac aj␣
↪aa ae
bb| bb aw az av ba at ap ax ar ay au am as ak aj aq ah ao aa al ae an ai ac ab ag␣
↪ad af
```

Another symbol set is base 10 digits, padded with leading zeros to make a common width.

```
sage: from sage.matrix.operation_table import OperationTable
sage: G=AlternatingGroup(4)
sage: OperationTable(G, operator.mul, names='digits')
 *  00 01 02 03 04 05 06 07 08 09 10 11
  +------------------------------------
00| 00 01 02 03 04 05 06 07 08 09 10 11
01| 01 05 03 07 06 00 08 02 04 10 11 09
02| 02 04 06 05 10 09 00 11 07 03 01 08
03| 03 06 08 00 11 10 01 09 02 07 05 04
04| 04 09 05 11 00 02 07 06 10 01 08 03
05| 05 00 07 02 08 01 04 03 06 11 09 10
06| 06 10 00 09 01 03 02 08 11 05 04 07
07| 07 08 04 01 09 11 05 10 03 02 00 06
08| 08 11 01 10 05 07 03 04 09 00 06 02
09| 09 02 11 06 07 04 10 05 00 08 03 01
10| 10 03 09 08 02 06 11 00 01 04 07 05
11| 11 07 10 04 03 08 09 01 05 06 02 00
```

If the group's elements are not too cumbersome, or the group is small, then the string representation of the elements can be used.

```
sage: from sage.matrix.operation_table import OperationTable
sage: G=AlternatingGroup(3)
sage: OperationTable(G, operator.mul, names='elements')
      *       ()  (1,2,3)  (1,3,2)
       +------------------------
     ()|      ()  (1,2,3)  (1,3,2)
(1,2,3)| (1,2,3)  (1,3,2)       ()
(1,3,2)| (1,3,2)       ()  (1,2,3)
```

You can give the elements any names you like, but they need to be ordered in the same order as returned by the `column_keys()` method.

```
sage: from sage.matrix.operation_table import OperationTable
sage: G=QuaternionGroup()
sage: T=OperationTable(G, operator.mul)
sage: T.column_keys()
((), (1,2,3,4)(5,6,7,8), ..., (1,7,3,5)(2,6,4,8))
sage: names=['1', 'I', 'J', '-1', '-K', 'K', '-I', '-J']
sage: T.change_names(names=names)
sage: sorted(T.translation().items())
[('-1', (1,3)(2,4)(5,7)(6,8)),..., ('K', (1,8,3,6)(2,7,4,5))]
sage: T
 *   1  I  J -1 -K  K -I -J
  +------------------------
 1|  1  I  J -1 -K  K -I -J
 I|  I -1  K -I  J -J  1 -K
 J|  J -K -1 -J -I  I  K  1
-1| -1 -I -J  1  K -K  I  J
-K| -K -J  I  K -1  1  J -I
 K|  K  J -I -K  1 -1 -J  I
-I| -I  1 -K  I -J  J -1  K
-J| -J  K  1  J  I -I -K -1
```

With the right functions and a list comprehension, custom names can be easier. A multiplication table for hex digits (without carries):

```
sage: from sage.matrix.operation_table import OperationTable
sage: R=Integers(16)
sage: names=[hex(Integer(a)) for a in R]
sage: OperationTable(R, operation=operator.mul, names=names)
*  0 1 2 3 4 5 6 7 8 9 a b c d e f
 +--------------------------------
0| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
1| 0 1 2 3 4 5 6 7 8 9 a b c d e f
2| 0 2 4 6 8 a c e 0 2 4 6 8 a c e
3| 0 3 6 9 c f 2 5 8 b e 1 4 7 a d
4| 0 4 8 c 0 4 8 c 0 4 8 c 0 4 8 c
5| 0 5 a f 4 9 e 3 8 d 2 7 c 1 6 b
6| 0 6 c 2 8 e 4 a 0 6 c 2 8 e 4 a
7| 0 7 e 5 c 3 a 1 8 f 6 d 4 b 2 9
8| 0 8 0 8 0 8 0 8 0 8 0 8 0 8 0 8
9| 0 9 2 b 4 d 6 f 8 1 a 3 c 5 e 7
a| 0 a 4 e 8 2 c 6 0 a 4 e 8 2 c 6
b| 0 b 6 1 c 7 2 d 8 3 e 9 4 f a 5
c| 0 c 8 4 0 c 8 4 0 c 8 4 0 c 8 4
d| 0 d a 7 4 1 e b 8 5 2 f c 9 6 3
e| 0 e c a 8 6 4 2 0 e c a 8 6 4 2
f| 0 f e d c b a 9 8 7 6 5 4 3 2 1
```

This should be flexible enough to create a variety of such tables.

```
sage: from sage.matrix.operation_table import OperationTable
sage: from operator import xor
sage: T=OperationTable(ZZ, xor, elements=range(8))
sage: T
.  a b c d e f g h
 +----------------
a| a b c d e f g h
b| b a d c f e h g
c| c d a b g h e f
d| d c b a h g f e
e| e f g h a b c d
f| f e h g b a d c
g| g h e f c d a b
h| h g f e d c b a
sage: names=['000', '001','010','011','100','101','110','111']
sage: T.change_names(names)
sage: T.set_print_symbols('^', '\\land')
sage: T
  ^  000 001 010 011 100 101 110 111
   +--------------------------------
000| 000 001 010 011 100 101 110 111
001| 001 000 011 010 101 100 111 110
010| 010 011 000 001 110 111 100 101
011| 011 010 001 000 111 110 101 100
100| 100 101 110 111 000 001 010 011
101| 101 100 111 110 001 000 011 010
110| 110 111 100 101 010 011 000 001
111| 111 110 101 100 011 010 001 000

sage: T = OperationTable([False, True], operator.or_, names = 'elements')
sage: T
     .  False  True
      +------------
False|  False  True
 True|   True  True
```

TODO:

Provide color and grayscale graphical representations of tables. See commented-out stubs in source code.

AUTHOR:

- Rob Beezer (2010-03-15)

change_names (*names*)

For an existing operation table, change the names used for the elements.

INPUT:

- `names` - the type of names used, values are:
  - `'letters'` - lowercase ASCII letters are used for a base 26 representation of the elements' positions in the list given by `list()`, padded to a common width with leading 'a's.
  - `'digits'` - base 10 representation of the elements' positions in the list given by `list()`, padded to a common width with leading zeros.
  - `'elements'` - the string representations of the elements themselves.

–a list - a list of strings, where the length of the list equals the number of elements.

OUTPUT: `None` . This method changes the table "in-place", so any printed version will change and the output of the `dict()` will also change. So any items of interest about a particular table need to be copied/saved prior to calling this method.

EXAMPLES:

More examples can be found in the documentation for *OperationTable* since creating a new operation table uses the same routine.

```
sage: from sage.matrix.operation_table import OperationTable
sage: D=DihedralGroup(2)
sage: T=OperationTable(D, operator.mul)
sage: T
*  a b c d
 +--------
a| a b c d
b| b a d c
c| c d a b
d| d c b a
sage: T.translation()['c']
(1,2)
sage: T.change_names('digits')
sage: T
*  0 1 2 3
 +--------
0| 0 1 2 3
1| 1 0 3 2
2| 2 3 0 1
3| 3 2 1 0
sage: T.translation()['2']
(1,2)
sage: T.change_names('elements')
sage: T
         *           ()       (3,4)       (1,2)  (1,2)(3,4)
          +--------------------------------------------
        ()|          ()       (3,4)       (1,2)  (1,2)(3,4)
     (3,4)|       (3,4)          ()  (1,2)(3,4)       (1,2)
     (1,2)|       (1,2)  (1,2)(3,4)          ()       (3,4)
(1,2)(3,4)|  (1,2)(3,4)       (1,2)       (3,4)          ()
sage: T.translation()['(1,2)']
(1,2)
sage: T.change_names(['w', 'x', 'y', 'z'])
sage: T
*  w x y z
 +--------
w| w x y z
x| x w z y
y| y z w x
z| z y x w
sage: T.translation()['y']
(1,2)
```

column_keys ( )

Returns a tuple of the elements used to build the table.

Note: `column_keys` and `row_keys` are identical. Both list the elements in the order used to label the table.

OUTPUT:

The elements of the algebraic structure used to build the table, as a list. But most importantly, elements are present in the list in the order which they appear in the table's column headings.

EXAMPLES:

```
sage: from sage.matrix.operation_table import OperationTable
sage: G=AlternatingGroup(3)
sage: T=OperationTable(G, operator.mul)
sage: T.column_keys()
((), (1,2,3), (1,3,2))
```

matrix_of_variables ( )

This method provides some backward compatibility for Cayley tables of groups, whose output was restricted to this single format.

EXAMPLES:

The output here is from the doctests for the old `cayley_table()` method for permutation groups.

```
sage: from sage.matrix.operation_table import OperationTable
sage: G=PermutationGroup(['(1,2,3)', '(2,3)'])
sage: T=OperationTable(G, operator.mul)
sage: T.matrix_of_variables()
[x0 x1 x2 x3 x4 x5]
[x1 x0 x3 x2 x5 x4]
[x2 x5 x4 x1 x0 x3]
[x3 x4 x5 x0 x1 x2]
[x4 x3 x0 x5 x2 x1]
[x5 x2 x1 x4 x3 x0]
sage: T.column_keys()[3]*T.column_keys()[3] == T.column_keys()[0]
True
```

row_keys ( )

Returns a tuple of the elements used to build the table.

Note: `column_keys` and `row_keys` are identical. Both list the elements in the order used to label the table.

OUTPUT:

The elements of the algebraic structure used to build the table, as a list. But most importantly, elements are present in the list in the order which they appear in the table's column headings.

EXAMPLES:

```
sage: from sage.matrix.operation_table import OperationTable
sage: G=AlternatingGroup(3)
sage: T=OperationTable(G, operator.mul)
sage: T.column_keys()
((), (1,2,3), (1,3,2))
```

set_print_symbols (*ascii*, *latex*)

Set the symbols used for text and LaTeX printing of operation tables.

INPUT:

- `ascii` - a single character for text table
- `latex` - a string to represent an operation in LaTeX math mode. Note the need for double-backslashes to escape properly.

EXAMPLES:

```
sage: from sage.matrix.operation_table import OperationTable
sage: G=AlternatingGroup(3)
sage: T=OperationTable(G, operator.mul)
sage: T.set_print_symbols('@', '\\times')
sage: T
@  a b c
 +------
a| a b c
b| b c a
c| c a b
sage: T._latex_()
'{\\setlength{\\arraycolsep}{2ex}\n\\begin{array}{r|*{3}{r}}\n\\multicolumn{1}
↪{c|}{\\times}&a&b&c\\\\\hline\n{}a&a&b&c\\\\\n{}b&b&c&a\\\\\n{}c&c&a&
↪b\\\\\n\\end{array}}'
```

table ()

Returns the table as a list of lists, using integers to reference the elements.

OUTPUT: The rows of the table, as a list of rows, each row being a list of integer entries. The integers correspond to the order of the elements in the headings of the table and the order of the output of the `list()` method.

EXAMPLES:

```
sage: from sage.matrix.operation_table import OperationTable
sage: C=CyclicPermutationGroup(3)
sage: T=OperationTable(C, operator.mul)
sage: T.table()
[[0, 1, 2], [1, 2, 0], [2, 0, 1]]
```

translation ()

Returns a dictionary associating names with elements.

OUTPUT: A dictionary whose keys are strings used as names for entries of the table and values that are the actual elements of the algebraic structure.

EXAMPLES:

```
sage: from sage.matrix.operation_table import OperationTable
sage: G=AlternatingGroup(3)
sage: T=OperationTable(G, operator.mul, names=['p','q','r'])
sage: sorted(T.translation().items())
[('p', ()), ('q', (1,2,3)), ('r', (1,3,2))]
```

CHAPTER

TWENTYSIX

ACTIONS USED BY THE COERCION MODEL FOR MATRIX AND VECTOR MULTIPLICATIONS

Warning: The class `MatrixMulAction` and its descendants extends the class `Action`. As a consequence objects from these classes only keep weak references to the underlying sets which are acted upon. This decision was made in trac ticket #715 in order to allow garbage collection within the coercion framework, where actions are mainly used, and avoid memory leaks.

To ensure that the underlying set of such an object does not get garbage collected, it is sufficient to explicitly create a strong reference to it before creating the action.

```
sage: MSQ = MatrixSpace(QQ, 2)
sage: MSZ = MatrixSpace(ZZ['x'], 2)
sage: A = MSQ.get_action(MSZ)
sage: A
Left action by Full MatrixSpace of 2 by 2 dense matrices over Rational Field on␣
↪Full MatrixSpace of 2 by 2 dense matrices over Univariate Polynomial Ring in x␣
↪over Integer Ring
sage: import gc
sage: _ = gc.collect()
sage: A
Left action by Full MatrixSpace of 2 by 2 dense matrices over Rational Field on␣
↪Full MatrixSpace of 2 by 2 dense matrices over Univariate Polynomial Ring in x␣
↪over Integer Ring
```

Note: The `MatrixSpace()` function caches the objects it creates. Therefore, the underlying set `MSZ` in the above example will not be garbage collected, even if it is not strongly ref'ed. Nonetheless, there is no guarantee that the set that is acted upon will always be cached in such a way, so that following the above example is good practice.

EXAMPLES:

An action requires a common parent for the base rings, so the following doesn't work (see trac ticket #17859):

```
sage: vector(QQ, [1]) * matrix(Zmod(2), [[1]])
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for *: 'Vector space of
dimension 1 over Rational Field' and 'Full MatrixSpace of 1 by 1
dense matrices over Ring of integers modulo 2'
```

AUTHOR:

- Robert Bradshaw (2007-09): Initial version.

class sage.matrix.action.**MatrixMatrixAction**

Bases: *sage.matrix.action.MatrixMulAction*

EXAMPLES:

By trac ticket #715, there only is a weak reference on the underlying set, so that it can be garbage collected if only the action itself is explicitly referred to. Hence, we first assign the involved matrix spaces to a variable:

```
sage: R.<x> = ZZ[]
sage: MSR = MatrixSpace(R, 3, 3)
sage: MSQ = MatrixSpace(QQ, 3, 2)
sage: from sage.matrix.action import MatrixMatrixAction
sage: A = MatrixMatrixAction(MSR, MSQ); A
Left action by Full MatrixSpace of 3 by 3 dense matrices over Univariate␣
↪Polynomial Ring in x over Integer Ring on Full MatrixSpace of 3 by 2 dense␣
↪matrices over Rational Field
sage: A.codomain()
Full MatrixSpace of 3 by 2 dense matrices over Univariate Polynomial Ring in x␣
↪over Rational Field
sage: A(matrix(R, 3, 3, x), matrix(QQ, 3, 2, range(6)))
[  0   x]
[2*x 3*x]
[4*x 5*x]
```

Note: The *MatrixSpace()* function caches the object it creates. Therefore, the underlying set MSZ in the above example will not be garbage collected, even if it is not strongly ref'ed. Nonetheless, there is no guarantee that the set that is acted upon will always be cached in such a way, so that following the above example is good practice.

class sage.matrix.action.**MatrixMulAction**

Bases: sage.categories.action.Action

codomain ( )

domain ( )

EXAMPLES:

By trac ticket #715, there only is a weak reference on the underlying set, so that it can be garbage collected if only the action itself is explicitly referred to. Hence, we first assign the involved matrix spaces to a variable:

```
sage: MSQ = MatrixSpace(QQ, 2)
sage: MSZ = MatrixSpace(ZZ['x'], 2)
sage: A = MSQ.get_action(MSZ); A
Left action by Full MatrixSpace of 2 by 2 dense matrices over Rational Field␣
↪on Full MatrixSpace of 2 by 2 dense matrices over Univariate Polynomial␣
↪Ring in x over Integer Ring
sage: A.actor()
Full MatrixSpace of 2 by 2 dense matrices over Rational Field
sage: A.domain()
Full MatrixSpace of 2 by 2 dense matrices over Univariate Polynomial Ring in␣
↪x over Integer Ring
sage: A.codomain()
Full MatrixSpace of 2 by 2 dense matrices over Univariate Polynomial Ring in␣
↪x over Rational Field
```

Note: The *MatrixSpace()* function caches the object it creates. Therefore, the underlying set MSZ in the above example will not be garbage collected, even if it is not strongly ref'ed. Nonetheless, there is no guarantee that the set that is acted upon will always be cached in such a way, so that following the above example is good practice.

---

class sage.matrix.action.**MatrixVectorAction**

Bases: *sage.matrix.action.MatrixMulAction*

EXAMPLES:

```
sage: from sage.matrix.action import MatrixVectorAction
sage: A = MatrixVectorAction(MatrixSpace(QQ, 3, 3), VectorSpace(CDF, 4)); A
Traceback (most recent call last):
...
TypeError: incompatible dimensions 3, 4
```

class sage.matrix.action.**VectorMatrixAction**

Bases: *sage.matrix.action.MatrixMulAction*

EXAMPLES:

```
sage: from sage.matrix.action import VectorMatrixAction
sage: A = VectorMatrixAction(MatrixSpace(QQ, 5, 3), VectorSpace(CDF, 3)); A
Traceback (most recent call last):
...
TypeError: incompatible dimensions 5, 3
```

CHAPTER

TWENTYSEVEN

FUNCTIONS FOR CHANGING THE BASE RING OF MATRICES QUICKLY.

sage.matrix.change_ring.**integer_to_real_double_dense**(*A*)

Fast conversion of a matrix over the integers to a matrix with real double entries.

INPUT: A – a dense matrix over the integers

OUTPUT: – a dense real double matrix

EXAMPLES: sage: a = matrix(ZZ,2,3,[-2,-5,3,4,8,1030339830489349908]) sage: a.change_ring(RDF) [-2.0 -5.0 3.0] [4.0 8.0 1.0303398304893499e+18] sage: import sage.matrix.change_ring sage: sage.matrix.change_ring.integer_to_real_double_dense(a) [-2.0 -5.0 3.0] [4.0 8.0 1.0303398304893499e+18]

CHAPTER

TWENTYEIGHT

ECHELON MATRICES OVER FINITE FIELDS.

sage.matrix.echelon_matrix.**reduced_echelon_matrix_iterator**(*K*, *k*, *n*, *sparse=False*, *copy=True*, *set_immutable=False*)

An iterator over (k, n) reduced echelon matrices over the finite field K.

INPUT:

- `K` – a finite field
- `k` – number of rows (or the size of the subspace)
- `n` – number of columns (or the dimension of the ambient space)
- `sparse` – boolean (default is `False`)
- `copy` – boolean. If set to `False` then iterator yields the same matrix over and over (but with different entries). Default is `True` which is safer but might be slower.
- `set_immutable` – boolean. If set to `True` then the output matrices are immutable. This option automatically turns `copy` into `True`.

Note: We ensure that the iteration order is so that all matrices with given pivot columns are generated consecutively. Furthermore, the order in which the pivot columns appear is lexicographic.

It would be faster to generate the pivots columns following a Gray code. There would be only one pivot changing at a time, avoiding the possibly expensive `m0.__copy__()`. However that would modify the generation order some functions depend upon.

EXAMPLES:

```
sage: from sage.matrix.echelon_matrix import reduced_echelon_matrix_iterator
sage: it = reduced_echelon_matrix_iterator(GF(2),2,3)
sage: for m in it:
....:     print(m)
....:     print(m.pivots())
....:     print("*******")
[1 0 0]
[0 1 0]
(0, 1)
*******
[1 0 0]
[0 1 1]
(0, 1)
*******
```

```
[1 0 1]
[0 1 0]
(0, 1)
*******
[1 0 1]
[0 1 1]
(0, 1)
*******
[1 0 0]
[0 0 1]
(0, 2)
*******
[1 1 0]
[0 0 1]
(0, 2)
*******
[0 1 0]
[0 0 1]
(1, 2)
*******
```

CHAPTER TWENTYNINE

MATRICES OVER CYCLOTOMIC FIELDS

The underlying matrix for a Matrix_cyclo_dense object is stored as follows: given an n x m matrix over a cyclotomic field of degree d, we store a d x (nm) matrix over QQ, each column of which corresponds to an element of the original matrix. This can be retrieved via the _rational_matrix method. Here is an example illustrating this:

EXAMPLES:

```
sage: F.<zeta> = CyclotomicField(5)
sage: M = Matrix(F, 2, 3, [zeta, 3, zeta**4+5, (zeta+1)**4, 0, 1])
sage: M
[                        zeta                            3  -zeta^3 - zeta^2 - zeta + 4]
[3*zeta^3 + 5*zeta^2 + 3*zeta                            0                            1]

sage: M._rational_matrix()
[ 0  3  4  0  0  1]
[ 1  0 -1  3  0  0]
[ 0  0 -1  5  0  0]
[ 0  0 -1  3  0  0]
```

AUTHORS:

- William Stein
- Craig Citro

class sage.matrix.matrix_cyclo_dense.**Matrix_cyclo_dense**
Bases: *sage.matrix.matrix_dense.Matrix_dense*

Initialize a newly created cyclotomic matrix.

INPUT:

- `parent` – a matrix space over a cyclotomic field
- `entries` – a list of entries or scalar
- `coerce` – boolean; if true entries are coerced to base ring
- `copy` – boolean; ignored due to underlying data structure

EXAMPLES:

This function is called implicitly when you create new cyclotomic dense matrices:

```
sage: W.<a> = CyclotomicField(100)
sage: A = matrix(2, 3, [1, 1/a, 1-a,a, -2/3*a, a^19])
sage: A
[                        1 -a^39 + a^29 - a^19 + a^9                    -a + 1]
```

```
[                          a                      -2/3*a                        a^19]
sage: TestSuite(A).run()
```

charpoly (*var='x'*, *algorithm='multimodular'*, *proof=None*)

Return the characteristic polynomial of self, as a polynomial over the base ring.

INPUT:

- algorithm
  - 'multimodular' (default): reduce modulo primes, compute charpoly mod p, and lift (very fast)
  - 'pari': use pari (quite slow; comparable to Magma v2.14 though)
  - 'hessenberg': put matrix in Hessenberg form (double dog slow)
- proof – bool (default: None) proof flag determined by global linalg proof.

OUTPUT:

polynomial

EXAMPLES:

```
sage: K.<z> = CyclotomicField(5)
sage: a = matrix(K, 3, [1,z,1+z^2, z/3,1,2,3,z^2,1-z])
sage: f = a.charpoly(); f
x^3 + (z - 3)*x^2 + (-16/3*z^2 - 2*z)*x - 2/3*z^3 + 16/3*z^2 - 5*z + 5/3
sage: f(a)
[0 0 0]
[0 0 0]
[0 0 0]
sage: a.charpoly(algorithm='pari')
x^3 + (z - 3)*x^2 + (-16/3*z^2 - 2*z)*x - 2/3*z^3 + 16/3*z^2 - 5*z + 5/3
sage: a.charpoly(algorithm='hessenberg')
x^3 + (z - 3)*x^2 + (-16/3*z^2 - 2*z)*x - 2/3*z^3 + 16/3*z^2 - 5*z + 5/3

sage: Matrix(K, 1, [0]).charpoly()
x
sage: Matrix(K, 1, [5]).charpoly(var='y')
y - 5

sage: Matrix(CyclotomicField(13),3).charpoly()
x^3
sage: Matrix(CyclotomicField(13),3).charpoly()[2].parent()
Cyclotomic Field of order 13 and degree 12
```

coefficient_bound ( )

Return an upper bound for the (complex) absolute values of all entries of self with respect to all embeddings.

Use `self.height()` for a sharper bound.

This is computed using just the Cauchy-Schwarz inequality, i.e., we use the fact that

```
\left| \sum_i a_i\zeta^i \right| \leq \sum_i |a_i|,
```

as $|\zeta| = 1$.

EXAMPLES:

```
sage: W.<z> = CyclotomicField(5)
sage: A = matrix(W, 2, 2, [1+z, 0, 9*z+7, -3 + 4*z]); A
[  z + 1        0]
[9*z + 7 4*z - 3]
sage: A.coefficient_bound()
16
```

The above bound is just $9 + 7$, coming from the lower left entry. A better bound would be the following:

```
sage: (A[1,0]).abs()
12.997543663...
```

denominator ( )

Return the denominator of the entries of this matrix.

OUTPUT:

integer – the smallest integer d so that d * self has entries in the ring of integers

EXAMPLES:

```
sage: W.<z> = CyclotomicField(5)
sage: A = matrix(W, 2, 2, [-2/7,2/3*z+z^2,-z,1+z/19]); A
[       -2/7 z^2 + 2/3*z]
[         -z  1/19*z + 1]
sage: d = A.denominator(); d
399
```

echelon_form (*algorithm='multimodular'*, *height_guess=None*)

Find the echelon form of self, using the specified algorithm.

The result is cached for each algorithm separately.

EXAMPLES:

```
sage: W.<z> = CyclotomicField(3)
sage: A = matrix(W, 2, 3, [1+z, 2/3, 9*z+7, -3 + 4*z, z, -7*z]); A
[  z + 1     2/3 9*z + 7]
[4*z - 3       z    -7*z]
sage: A.echelon_form()
[                  1                   0  -192/97*z - 361/97]
[                  0                   1 1851/97*z + 1272/97]
sage: A.echelon_form(algorithm='classical')
[                  1                   0  -192/97*z - 361/97]
[                  0                   1 1851/97*z + 1272/97]
```

We verify that the result is cached and that the caches are separate:

```
sage: A.echelon_form() is A.echelon_form()
True
sage: A.echelon_form() is A.echelon_form(algorithm='classical')
False
```

height ( )

Return the height of self.

If we let a_{ij} be the i, j entry of self, then we define the height of self to be

$\max_v \max_{i,j} |a_{ij}|_v$,

where v runs over all complex embeddings of `self.base_ring()`.

EXAMPLES:

```
sage: W.<z> = CyclotomicField(5)
sage: A = matrix(W, 2, 2, [1+z, 0, 9*z+7, -3 + 4*z]); A
[  z + 1        0]
[9*z + 7 4*z - 3]
sage: A.height()
12.997543663...
sage: (A[1,0]).abs()
12.997543663...
```

randomize (*density=1*, *num_bound=2*, *den_bound=2*, *distribution=None*, *nonzero=False*, **args*, ***kwds*)

Randomize the entries of `self`.

Choose rational numbers according to `distribution`, whose numerators are bounded by `num_bound` and whose denominators are bounded by `den_bound`.

EXAMPLES:

```
sage: A = Matrix(CyclotomicField(5),2,2,range(4)) ; A
[0 1]
[2 3]
sage: A.randomize()
sage: A   # random output
[       1/2*zeta5^2 + zeta5                        1/2]
[        -zeta5^2 + 2*zeta5 -2*zeta5^3 + 2*zeta5^2 + 2]
```

set_immutable ()

Change this matrix so that it is immutable.

EXAMPLES:

```
sage: W.<z> = CyclotomicField(5)
sage: A = matrix(W, 2, 2, [1,2/3*z+z^2,-z,1+z/2])
sage: A[0,0] = 10
sage: A.set_immutable()
sage: A[0,0] = 20
Traceback (most recent call last):
...
ValueError: matrix is immutable; please change a copy instead (i.e., use
↪copy(M) to change a copy of M).
```

Note that there is no function to set a matrix to be mutable again, since such a function would violate the whole point. Instead make a copy, which is always mutable by default.:

```
sage: A.set_mutable()
Traceback (most recent call last):
...
AttributeError: 'sage.matrix.matrix_cyclo_dense.Matrix_cyclo_dense' object
↪has no attribute 'set_mutable'
sage: B = A.__copy__()
sage: B[0,0] = 20
sage: B[0,0]
20
```

tensor_product (*A*, *subdivide=True*)

Return the tensor product of two matrices.

INPUT:

- `A` – a matrix
- `subdivide` – (default: `True`) whether or not to return natural subdivisions with the matrix

OUTPUT:

Replace each element of `self` by a copy of `A`, but first create a scalar multiple of `A` by the element it replaces. So if `self` is an $m \times n$ matrix and `A` is a $p \times q$ matrix, then the tensor product is an $mp \times nq$ matrix. By default, the matrix will be subdivided into submatrices of size $p \times q$.

EXAMPLES:

```
sage: C = CyclotomicField(12)
sage: M = matrix.random(C, 3, 3)
sage: N = matrix.random(C, 50, 50)
sage: M.tensor_product(M) == super(type(M), M).tensor_product(M)
True
sage: N = matrix.random(C, 15, 20)
sage: M.tensor_product(N) == super(type(M), M).tensor_product(N)
True
```

CHAPTER

THIRTY

MODULAR ALGORITHM TO COMPUTE HERMITE NORMAL FORMS OF INTEGER MATRICES.

AUTHORS:

- Clement Pernet and William Stein (2008-02-07): initial version

sage.matrix.matrix_integer_dense_hnf.**add_column**(*B*, *H_B*, *a*, *proof*)

The add column procedure.

INPUT:

- B – a square matrix (may be singular)
- H_B – the Hermite normal form of B
- a – an n x 1 matrix, where B has n rows
- proof – bool; whether to prove result correct, in case we use fallback method.

OUTPUT:

- x – a vector such that H' = H_B.augment(x) is the HNF of A = B.augment(a).

EXAMPLES:

```
sage: B = matrix(ZZ, 3, 3, [1,2,5, 0,-5,3, 1,1,2])
sage: H_B = B.echelon_form()
sage: a = matrix(ZZ, 3, 1, [1,8,-2])
sage: import sage.matrix.matrix_integer_dense_hnf as hnf
sage: x = hnf.add_column(B, H_B, a, True); x
[18]
[ 3]
[23]
sage: H_B.augment(x)
[ 1  0 17 18]
[ 0  1  3  3]
[ 0  0 18 23]
sage: B.augment(a).echelon_form()
[ 1  0 17 18]
[ 0  1  3  3]
[ 0  0 18 23]
```

sage.matrix.matrix_integer_dense_hnf.**add_column_fallback**(*B*, *a*, *proof*)

Simplistic version of add_column, in case the powerful clever one fails (e.g., B is singular).

INPUT:

B – a square matrix (may be singular) a – an n x 1 matrix, where B has n rows proof – bool; whether to prove result correct

OUTPUT:

x – a vector such that H' = H_B.augment(x) is the HNF of A = B.augment(a).

EXAMPLES:

```
sage: B = matrix(ZZ,3, [-1, -1, 1, -3, 8, -2, -1, -1, -1])
sage: a = matrix(ZZ,3,1, [1,2,3])
sage: import sage.matrix.matrix_integer_dense_hnf as matrix_integer_dense_hnf
sage: matrix_integer_dense_hnf.add_column_fallback(B, a, True)
[-3]
[-7]
[-2]
sage: matrix_integer_dense_hnf.add_column_fallback(B, a, False)
[-3]
[-7]
[-2]
sage: B.augment(a).hermite_form()
[ 1  1  1 -3]
[ 0 11  1 -7]
[ 0  0  2 -2]
```

sage.matrix.matrix_integer_dense_hnf.**add_row**(*A*, *b*, *pivots*, *include_zero_rows*)

The add row procedure.

INPUT:

•A – a matrix in Hermite normal form with n column

•b – an n x 1 row matrix

•pivots – sorted list of integers; the pivot positions of A.

OUTPUT:

•H – the Hermite normal form of A.stack(b).

•new_pivots – the pivot columns of H.

EXAMPLES:

```
sage: import sage.matrix.matrix_integer_dense_hnf as hnf
sage: A = matrix(ZZ, 2, 3, [-21, -7, 5, 1,20,-7])
sage: b = matrix(ZZ, 1,3, [-1,1,-1])
sage: hnf.add_row(A, b, A.pivots(), True)
(
[ 1  6 29]
[ 0  7 28]
[ 0  0 46], [0, 1, 2]
)
sage: A.stack(b).echelon_form()
[ 1  6 29]
[ 0  7 28]
[ 0  0 46]
```

sage.matrix.matrix_integer_dense_hnf.**benchmark_hnf**(*nrange*, *bits=4*)

Run benchmark program.

EXAMPLES:

```
sage: import sage.matrix.matrix_integer_dense_hnf as hnf
sage: hnf.benchmark_hnf([50,100],32)
```

```
('sage', 50, 32, ...),
('sage', 100, 32, ...),
```

sage.matrix.matrix_integer_dense_hnf.**benchmark_magma_hnf**(*nrange*, *bits=4*)

EXAMPLES:

```
sage: import sage.matrix.matrix_integer_dense_hnf as hnf
sage: hnf.benchmark_magma_hnf([50,100],32)      # optional - magma
('magma', 50, 32, ...),
('magma', 100, 32, ...),
```

sage.matrix.matrix_integer_dense_hnf.**det_from_modp_and_divisor**(*A*, *d*, *p*, *z_mod*, *moduli*, *z_so_far=1*, *N_so_far=1*)

This is used for internal purposes for computing determinants quickly (with the hybrid p-adic / multimodular algorithm).

INPUT:

- A – a square matrix
- d – a divisor of the determinant of A
- p – a prime
- z_mod – values of det/d (mod ...)
- moduli – the moduli so far
- z_so_far – for a modulus p in the list moduli, (z_so_far mod p) is the determinant of A modulo p.
- N_so_far – N_so_far is the product over the primes in the list moduli.

OUTPUT:

- A triple (det bound, new z_so_far, new N_so_far).

EXAMPLES:

```
sage: a = matrix(ZZ, 3, [6, 1, 2, -56, -2, -1, -11, 2, -3])
sage: factor(a.det())
-1 * 13 * 29
sage: d = 13
sage: import sage.matrix.matrix_integer_dense_hnf as matrix_integer_dense_hnf
sage: matrix_integer_dense_hnf.det_from_modp_and_divisor(a, d, 97, [], [])
(-377, -29, 97)
sage: a.det()
-377
```

sage.matrix.matrix_integer_dense_hnf.**det_given_divisor**(*A*, *d*, *proof=True*, *stabilize=2*)

Given a divisor d of the determinant of A, compute the determinant of A.

INPUT:

- `A` – a square integer matrix
- `d` – a nonzero integer that is assumed to divide the determinant of A
- `proof` – bool (default: True) compute det modulo enough primes so that the determinant is computed provably correctly (via the Hadamard bound). It would be VERY hard for `det()` to fail even with proof=False.

- `stabilize` – int (default: 2) if proof = False, then compute the determinant modulo p until `stabilize` successive modulo determinant computations stabilize.

OUTPUT:

integer – determinant

EXAMPLES:

```
sage: import sage.matrix.matrix_integer_dense_hnf as matrix_integer_dense_hnf
sage: a = matrix(ZZ,3,[-1, -1, -1, -20, 4, 1, -1, 1, 2])
sage: matrix_integer_dense_hnf.det_given_divisor(a, 3)
-30
sage: matrix_integer_dense_hnf.det_given_divisor(a, 3, proof=False)
-30
sage: matrix_integer_dense_hnf.det_given_divisor(a, 3, proof=False, stabilize=1)
-30
sage: a.det()
-30
```

Here we illustrate proof=False giving a wrong answer:

```
sage: p = matrix_integer_dense_hnf.max_det_prime(2)
sage: q = previous_prime(p)
sage: a = matrix(ZZ, 2, [p, 0, 0, q])
sage: p * q
70368442188091
sage: matrix_integer_dense_hnf.det_given_divisor(a, 1, proof=False, stabilize=2)
0
```

This still works, because we don't work modulo primes that divide the determinant bound, which is found using a p-adic algorithm:

```
sage: a.det(proof=False, stabilize=2)
70368442188091
```

3 primes is enough:

```
sage: matrix_integer_dense_hnf.det_given_divisor(a, 1, proof=False, stabilize=3)
70368442188091
sage: matrix_integer_dense_hnf.det_given_divisor(a, 1, proof=False, stabilize=5)
70368442188091
sage: matrix_integer_dense_hnf.det_given_divisor(a, 1, proof=True)
70368442188091
```

`sage.matrix.matrix_integer_dense_hnf.`**`det_padic`**`(`*A*, *proof=True*, *stabilize=2*`)`

Return the determinant of A, computed using a p-adic/multimodular algorithm.

INPUT:

- `A` – a square matrix
- `proof` – boolean
- `stabilize` (default: 2) – if proof False, number of successive primes so that CRT det must stabilize.

EXAMPLES:

```
sage: import sage.matrix.matrix_integer_dense_hnf as h
sage: a = matrix(ZZ, 3, [1..9])
sage: h.det_padic(a)
```

```
0
sage: a = matrix(ZZ, 3, [1,2,5,-7,8,10,192,5,18])
sage: h.det_padic(a)
-3669
sage: a.determinant(algorithm='ntl')
-3669
```

sage.matrix.matrix_integer_dense_hnf.**double_det**(*A*, *b*, *c*, *proof*)

Compute the determinants of the stacked integer matrices A.stack(b) and A.stack(c).

INPUT:

- A – an (n-1) x n matrix
- b – an 1 x n matrix
- c – an 1 x n matrix
- proof – whether or not to compute the det modulo enough times to provably compute the determinant.

OUTPUT:

- a pair of two integers.

EXAMPLES:

```
sage: from sage.matrix.matrix_integer_dense_hnf import double_det
sage: A = matrix(ZZ, 2, 3, [1,2,3, 4,-2,5])
sage: b = matrix(ZZ, 1, 3, [1,-2,5])
sage: c = matrix(ZZ, 1, 3, [8,2,10])
sage: A.stack(b).det()
-48
sage: A.stack(c).det()
42
sage: double_det(A, b, c, False)
(-48, 42)
```

sage.matrix.matrix_integer_dense_hnf.**extract_ones_data**(*H*, *pivots*)

Compute ones data and corresponding submatrices of H. This is used to optimized the add_row function.

INPUT:

- H – a matrix in HNF
- pivots – list of all pivot column positions of H

OUTPUT:

C, D, E, onecol, onerow, non_onecol, non_onerow where onecol, onerow, non_onecol, non_onerow are as for the ones function, and C, D, E are matrices:

- C – submatrix of all non-onecol columns and onecol rows
- D – all non-onecol columns and other rows
- E – inverse of D

If D isn't invertible or there are 0 or more than 2 non onecols, then C, D, and E are set to None.

EXAMPLES:

```
sage: H = matrix(ZZ, 3, 4, [1, 0, 0, 7, 0, 1, 5, 2, 0, 0, 6, 6])
sage: import sage.matrix.matrix_integer_dense_hnf as matrix_integer_dense_hnf
sage: matrix_integer_dense_hnf.extract_ones_data(H, [0,1,2])
```

```
(
[0]
[5], [6], [1/6], [0, 1], [0, 1], [2], [2]
)
```

Here we get None's since the (2,2) position submatrix is not invertible. sage: H = matrix(ZZ, 3, 5, [1, 0, 0, 45, -36, 0, 1, 0, 131, -107, 0, 0, 0, 178, -145]); H [1 0 0 45 -36] [0 1 0 131 -107] [0 0 0 178 -145] sage: import sage.matrix.matrix_integer_dense_hnf as matrix_integer_dense_hnf sage: matrix_integer_dense_hnf.extract_ones_data(H, [0,1,3]) (None, None, None, [0, 1], [0, 1], [2], [2])

sage.matrix.matrix_integer_dense_hnf.**hnf**(*A*, *include_zero_rows=True*, *proof=True*)

Return the Hermite Normal Form of a general integer matrix A, along with the pivot columns.

INPUT:

- A – an n x m matrix A over the integers.
- include_zero_rows – bool (default: True) whether or not to include zero rows in the output matrix
- proof – whether or not to prove the result correct.

OUTPUT:

- matrix – the Hermite normal form of A
- pivots – the pivot column positions of A

EXAMPLES:

```
sage: import sage.matrix.matrix_integer_dense_hnf as matrix_integer_dense_hnf
sage: a = matrix(ZZ,3,5,[-2, -6, -3, -17, -1, 2, -1, -1, -2, -1, -2, -2, -6, 9,␣
↪2])
sage: matrix_integer_dense_hnf.hnf(a)
(
[   2    0   26  -75  -10]
[   0    1   27  -73   -9]
[   0    0   37 -106  -13], [0, 1, 2]
)
sage: matrix_integer_dense_hnf.hnf(a.transpose())
(
[1 0 0]
[0 1 0]
[0 0 1]
[0 0 0]
[0 0 0], [0, 1, 2]
)
sage: matrix_integer_dense_hnf.hnf(a.transpose(), include_zero_rows=False)
(
[1 0 0]
[0 1 0]
[0 0 1], [0, 1, 2]
)
```

sage.matrix.matrix_integer_dense_hnf.**hnf_square**(*A*, *proof*)

INPUT:

- a nonsingular n x n matrix A over the integers.

OUTPUT:

- the Hermite normal form of A.

EXAMPLES:

```
sage: import sage.matrix.matrix_integer_dense_hnf as hnf
sage: A = matrix(ZZ, 3, [-21, -7, 5, 1,20,-7, -1,1,-1])
sage: hnf.hnf_square(A, False)
[ 1  6 29]
[ 0  7 28]
[ 0  0 46]
sage: A.echelon_form()
[ 1  6 29]
[ 0  7 28]
[ 0  0 46]
```

sage.matrix.matrix_integer_dense_hnf.**hnf_with_transformation**(*A*, *proof=True*)

Compute the HNF H of A along with a transformation matrix U such that U*A = H.

INPUT:

- A – an n x m matrix A over the integers.
- proof – whether or not to prove the result correct.

OUTPUT:

- matrix – the Hermite normal form H of A
- U – a unimodular matrix such that U * A = H

EXAMPLES:

```
sage: import sage.matrix.matrix_integer_dense_hnf as matrix_integer_dense_hnf
sage: A = matrix(ZZ, 2, [1, -5, -10, 1, 3, 197]); A
[  1  -5 -10]
[  1   3 197]
sage: H, U = matrix_integer_dense_hnf.hnf_with_transformation(A)
sage: H
[  1   3 197]
[  0   8 207]
sage: U
[ 0  1]
[-1  1]
sage: U*A
[  1   3 197]
[  0   8 207]
```

sage.matrix.matrix_integer_dense_hnf.**hnf_with_transformation_tests**(*n=10*, *m=5*, *trials=10*)

Use this to randomly test that hnf with transformation matrix is working.

EXAMPLES:

```
sage: from sage.matrix.matrix_integer_dense_hnf import hnf_with_transformation_
↪tests
sage: hnf_with_transformation_tests(n=15,m=10, trials=10)
0 1 2 3 4 5 6 7 8 9
```

sage.matrix.matrix_integer_dense_hnf.**interleave_matrices**(*A*, *B*, *cols1*, *cols2*)

INPUT:

- A, B – matrices with the same number of rows

- cols1, cols2 – disjoint lists of integers

OUTPUT:

construct a new matrix C by sticking the columns of A at the positions specified by cols1 and the columns of B at the positions specified by cols2.

EXAMPLES:

```
sage: A = matrix(ZZ, 2, [1,2,3,4]); B = matrix(ZZ, 2, [-1,5,2,3])
sage: A
[1 2]
[3 4]
sage: B
[-1  5]
[ 2  3]
sage: import sage.matrix.matrix_integer_dense_hnf as hnf
sage: hnf.interleave_matrices(A, B, [1,3], [0,2])
[-1  1  5  2]
[ 2  3  3  4]
```

sage.matrix.matrix_integer_dense_hnf.**is_in_hnf_form**(*H*, *pivots*)

Return True precisely if the matrix H is in Hermite normal form with given pivot columns.

INPUT:

H – matrix pivots – sorted list of integers

OUTPUT:

bool – True or False

EXAMPLES:

```
sage: a = matrix(ZZ,3,5,[-2, -6, -3, -17, -1, 2, -1, -1, -2, -1, -2, -2, -6, 9,␣
↪2])
sage: import sage.matrix.matrix_integer_dense_hnf as matrix_integer_dense_hnf
sage: matrix_integer_dense_hnf.is_in_hnf_form(a,range(3))
False
sage: e = a.hermite_form(); p = a.pivots()
sage: matrix_integer_dense_hnf.is_in_hnf_form(e, p)
True
```

sage.matrix.matrix_integer_dense_hnf.**max_det_prime**(*n*)

Return the largest prime so that it is reasonably efficienct to compute modulo that prime with n x n matrices in LinBox.

INPUT:

- n – a positive integer

OUTPUT:

a prime number

EXAMPLES:

```
sage: from sage.matrix.matrix_integer_dense_hnf import max_det_prime
sage: max_det_prime(10000)
8388593
sage: max_det_prime(1000)
8388593
```

```
sage: max_det_prime(10)
8388593
```

sage.matrix.matrix_integer_dense_hnf.**ones**(*H*, *pivots*)

Find all 1 pivot columns of the matrix H in Hermite form, along with the corresponding rows, and also the non 1 pivot columns and non-pivot rows. Here a 1 pivot column is a pivot column so that the leading bottom entry is 1.

INPUT:

- H – matrix in Hermite form
- pivots – list of integers (all pivot positions of H).

OUTPUT:

4-tuple of integer lists: onecol, onerow, non_oneol, non_onerow

EXAMPLES:

```
sage: H = matrix(ZZ, 3, 5, [1, 0, 0, 45, -36, 0, 1, 0, 131, -107, 0, 0, 0, 178, -
↪145]); H
[   1    0    0   45  -36]
[   0    1    0  131 -107]
[   0    0    0  178 -145]
sage: import sage.matrix.matrix_integer_dense_hnf as matrix_integer_dense_hnf
sage: matrix_integer_dense_hnf.ones(H, [0,1,3])
([0, 1], [0, 1], [2], [2])
```

sage.matrix.matrix_integer_dense_hnf.**pad_zeros**(*A*, *nrows*)

Add zeros to the bottom of A so that the resulting matrix has nrows.

INPUT:

- A – a matrix
- nrows – an integer that is at least as big as the number of rows of A.

OUTPUT:

a matrix with nrows rows.

EXAMPLES:

```
sage: import sage.matrix.matrix_integer_dense_hnf as matrix_integer_dense_hnf
sage: a = matrix(ZZ, 2, 4, [1, 0, 0, 7, 0, 1, 5, 2])
sage: matrix_integer_dense_hnf.pad_zeros(a, 4)
[1 0 0 7]
[0 1 5 2]
[0 0 0 0]
[0 0 0 0]
sage: matrix_integer_dense_hnf.pad_zeros(a, 2)
[1 0 0 7]
[0 1 5 2]
```

sage.matrix.matrix_integer_dense_hnf.**pivots_of_hnf_matrix**(*H*)

Return the pivot columns of a matrix H assumed to be in HNF.

INPUT:

- H – a matrix that must be HNF

OUTPUT:

- list – list of pivots

EXAMPLES:

```
sage: H = matrix(ZZ, 3, 5, [1, 0, 0, 45, -36, 0, 1, 0, 131, -107, 0, 0, 0, 178, -
↪145]); H
[   1    0    0   45  -36]
[   0    1    0  131 -107]
[   0    0    0  178 -145]
sage: import sage.matrix.matrix_integer_dense_hnf as matrix_integer_dense_hnf
sage: matrix_integer_dense_hnf.pivots_of_hnf_matrix(H)
[0, 1, 3]
```

sage.matrix.matrix_integer_dense_hnf.**probable_hnf**(*A*, *include_zero_rows*, *proof*)

Return the HNF of A or raise an exception if something involving the randomized nature of the algorithm goes wrong along the way. Calling this function again a few times should result it in it working, at least if proof=True.

INPUT:

- A – a matrix
- include_zero_rows – bool
- proof – bool

OUTPUT:

the Hermite normal form of A. cols – pivot columns

EXAMPLES:

```
sage: a = matrix(ZZ,4,3,[-1, -1, -1, -20, 4, 1, -1, 1, 2,1,2,3])
sage: import sage.matrix.matrix_integer_dense_hnf as matrix_integer_dense_hnf
sage: matrix_integer_dense_hnf.probable_hnf(a, True, True)
(
[1 0 0]
[0 1 0]
[0 0 1]
[0 0 0], [0, 1, 2]
)
sage: matrix_integer_dense_hnf.probable_hnf(a, False, True)
(
[1 0 0]
[0 1 0]
[0 0 1], [0, 1, 2]
)
sage: matrix_integer_dense_hnf.probable_hnf(a, False, False)
(
[1 0 0]
[0 1 0]
[0 0 1], [0, 1, 2]
)
```

sage.matrix.matrix_integer_dense_hnf.**probable_pivot_columns**(*A*)

INPUT:

- A – a matrix

OUTPUT:

a tuple of integers

EXAMPLES:

```
sage: import sage.matrix.matrix_integer_dense_hnf as matrix_integer_dense_hnf
sage: a = matrix(ZZ,3,[0, -1, -1, 0, -20, 1, 0, 1, 2])
sage: a
[  0  -1  -1]
[  0 -20   1]
[  0   1   2]
sage: matrix_integer_dense_hnf.probable_pivot_columns(a)
(1, 2)
```

sage.matrix.matrix_integer_dense_hnf.**probable_pivot_rows**(*A*)

Return rows of A that are very likely to be pivots.

This really finds the pivots of A modulo a random prime.

INPUT:

- A – a matrix

OUTPUT:

a tuple of integers

EXAMPLES:

```
sage: import sage.matrix.matrix_integer_dense_hnf as matrix_integer_dense_hnf
sage: a = matrix(ZZ,3,[0, -1, -1, 0, -20, 1, 0, 1, 2])
sage: a
[  0  -1  -1]
[  0 -20   1]
[  0   1   2]
sage: matrix_integer_dense_hnf.probable_pivot_rows(a)
(0, 1)
```

sage.matrix.matrix_integer_dense_hnf.**sanity_checks**(*times=50*, *n=8*, *m=5*, *proof=True*, *stabilize=2*, *check_using_magma=True*)

Run random sanity checks on the modular p-adic HNF with tall and wide matrices both dense and sparse.

INPUT:

- times – number of times to randomly try matrices with each shape
- n – number of rows
- m – number of columns
- proof – test with proof true
- stabilize – parameter to pass to hnf algorithm when proof is False
- check_using_magma – if True use Magma instead of PARI to check correctness of computed HNF's. Since PARI's HNF is buggy and slow (as of 2008-02-16 non-pivot entries sometimes aren't normalized to be nonnegative) the default is Magma.

EXAMPLES:

```
sage: import sage.matrix.matrix_integer_dense_hnf as matrix_integer_dense_hnf
sage: matrix_integer_dense_hnf.sanity_checks(times=5, check_using_magma=False)
small 8 x 5
0 1 2 3 4  (done)
big 8 x 5
0 1 2 3 4  (done)
small 5 x 8
```

```
0 1 2 3 4  (done)
big 5 x 8
0 1 2 3 4  (done)
sparse 8 x 5
0 1 2 3 4  (done)
sparse 5 x 8
0 1 2 3 4  (done)
ill conditioned -- 1000*A -- 8 x 5
0 1 2 3 4  (done)
ill conditioned -- 1000*A but one row -- 8 x 5
0 1 2 3 4  (done)
```

sage.matrix.matrix_integer_dense_hnf.**solve_system_with_difficult_last_row**(B, a)

Solve B*x = a when the last row of B contains huge entries using a clever trick that reduces the problem to solve C*x = a where C is B but with the last row replaced by something small, along with one easy null space computation. The latter are both solved p-adically.

INPUT:

- B – a square n x n nonsingular matrix with painful big bottom row.
- a – an n x 1 column matrix

OUTPUT:

- the unique solution to B*x = a.

EXAMPLES:

```
sage: from sage.matrix.matrix_integer_dense_hnf import solve_system_with_
↪difficult_last_row
sage: B = matrix(ZZ, 3, [1,2,4, 3,-4,7, 939082,2930982,132902384098234])
sage: a = matrix(ZZ,3,1, [1,2,5])
sage: z = solve_system_with_difficult_last_row(B, a)
sage: z
[ 106321906985474/132902379815497]
[132902385037291/1329023798154970]
[        -5221794/664511899077485]
sage: B*z
[1]
[2]
[5]
```

CHAPTER

THIRTYONE

SATURATION OVER ZZ

sage.matrix.matrix_integer_dense_saturation.**index_in_saturation**(*A*, *proof=True*)

The index of A in its saturation.

INPUT:

- `A` – matrix over $\mathbf{Z}$
- `proof` – boolean (`True` or `False`)

OUTPUT:

An integer

EXAMPLES:

```
sage: from sage.matrix.matrix_integer_dense_saturation import index_in_saturation
sage: A = matrix(ZZ, 2, 2, [3,2,3,4]); B = matrix(ZZ, 2,3,[1,2,3,4,5,6]); C = A*B;
↪ C
[11 16 21]
[19 26 33]
sage: index_in_saturation(C)
18
sage: W = C.row_space()
sage: S = W.saturation()
sage: W.index_in(S)
18
```

For any zero matrix the index in its saturation is 1 (see trac ticket #13034):

```
sage: m = matrix(ZZ, 3)
sage: m
[0 0 0]
[0 0 0]
[0 0 0]
sage: m.index_in_saturation()
1
sage: m = matrix(ZZ, 2, 3)
sage: m
[0 0 0]
[0 0 0]
sage: m.index_in_saturation()
1
```

sage.matrix.matrix_integer_dense_saturation.**p_saturation**(*A*, *p*, *proof=True*)

INPUT:

- A – a matrix over ZZ
- p – a prime
- proof – bool (default: True)

OUTPUT:

The p-saturation of the matrix A, i.e., a new matrix in Hermite form whose row span a ZZ-module that is p-saturated.

EXAMPLES:

```
sage: from sage.matrix.matrix_integer_dense_saturation import p_saturation
sage: A = matrix(ZZ, 2, 2, [3,2,3,4]); B = matrix(ZZ, 2,3,[1,2,3,4,5,6])
sage: A.det()
6
sage: C = A*B; C
[11 16 21]
[19 26 33]
sage: C2 = p_saturation(C, 2); C2
[ 1  8 15]
[ 0  9 18]
sage: C2.index_in_saturation()
9
sage: C3 = p_saturation(C, 3); C3
[ 1  0 -1]
[ 0  2  4]
sage: C3.index_in_saturation()
2
```

`sage.matrix.matrix_integer_dense_saturation.`**`random_sublist_of_size`**(*k*, *n*)

INPUT:

- k – an integer
- n – an integer

OUTPUT:

a randomly chosen sublist of range(k) of size n.

EXAMPLES:

```
sage: import sage.matrix.matrix_integer_dense_saturation as s
sage: s.random_sublist_of_size(10,3)
[0, 1, 5]
sage: s.random_sublist_of_size(10,7)
[0, 1, 3, 4, 5, 7, 8]
```

`sage.matrix.matrix_integer_dense_saturation.`**`saturation`**(*A*, *proof=True*, *p=0*, *max_dets=5*)

Compute a saturation matrix of A.

INPUT:

- A – a matrix over ZZ
- proof – bool (default: True)
- p – int (default: 0); if not 0 only guarantees that output is p-saturated
- max_dets – int (default: 4) max number of dets of submatrices to compute.

OUTPUT:

matrix – saturation of the matrix A.

EXAMPLES:

```
sage: from sage.matrix.matrix_integer_dense_saturation import saturation
sage: A = matrix(ZZ, 2, 2, [3,2,3,4]); B = matrix(ZZ, 2,3,[1,2,3,4,5,6]); C = A*B
sage: C
[11 16 21]
[19 26 33]
sage: C.index_in_saturation()
18
sage: S = saturation(C); S
[11 16 21]
[-2 -3 -4]
sage: S.index_in_saturation()
1
sage: saturation(C, proof=False)
[11 16 21]
[-2 -3 -4]
sage: saturation(C, p=2)
[11 16 21]
[-2 -3 -4]
sage: saturation(C, p=2, max_dets=1)
[11 16 21]
[-2 -3 -4]
```

sage.matrix.matrix_integer_dense_saturation.**solve_system_with_difficult_last_row**(*B*, *A*)

Solve the matrix equation B*Z = A when the last row of B contains huge entries.

INPUT:

- B – a square n x n nonsingular matrix with painful big bottom row.
- A – an n x k matrix.

OUTPUT:

the unique solution to B*Z = A.

EXAMPLES:

```
sage: from sage.matrix.matrix_integer_dense_saturation import solve_system_with_
↪difficult_last_row
sage: B = matrix(ZZ, 3, [1,2,3, 3,-1,2,939239082,39202803080,2939028038402834]);␣
↪A = matrix(ZZ,3,2,[1,2,4,3,-1,0])
sage: X = solve_system_with_difficult_last_row(B, A); X
[  290668794698843/226075992027744         468068726971/409557956572]
[-226078357385539/1582531944194208       1228691305937/2866905696004]
[      2365357795/1582531944194208           -17436221/2866905696004]
sage: B*X == A
True
```

CHAPTER

THIRTYTWO

SPARSE INTEGER MATRICES.

AUTHORS:

- William Stein (2007-02-21)
- Soroosh Yazdani (2007-02-21)

class `sage.matrix.matrix_integer_sparse.`**`Matrix_integer_sparse`**

Bases: `sage.matrix.matrix_sparse.Matrix_sparse`

Create a sparse matrix over the integers.

INPUT:

- `parent` – a matrix space
- `entries` – can be one of the following:
  - a Python dictionary whose items have the form `(i,j): x`, where `0 <= i < nrows`, `0 <= j < ncols`, and `x` is coercible to an integer. The `i,j` entry of `self` is set to `x`. The `x` 's can be `0`.
  - Alternatively, entries can be a list of *all* the entries of the sparse matrix, read row-by-row from top to bottom (so they would be mostly 0).
- `copy` – ignored
- `coerce` – ignored

`elementary_divisors` (*algorithm='pari'*)

Return the elementary divisors of self, in order.

The elementary divisors are the invariants of the finite abelian group that is the cokernel of *left* multiplication by this matrix. They are ordered in reverse by divisibility.

INPUT:

- self – matrix
- algorithm – (default: 'pari')
  - 'pari': works robustly, but is slower.
  - 'linbox' – use linbox (currently off, broken)

OUTPUT:

list of integers

EXAMPLES:

```
sage: matrix(3, range(9),sparse=True).elementary_divisors()
[1, 3, 0]
sage: M = matrix(ZZ, 3, [1,5,7, 3,6,9, 0,1,2], sparse=True)
sage: M.elementary_divisors()
[1, 1, 6]
```

This returns a copy, which is safe to change:

```
sage: edivs = M.elementary_divisors()
sage: edivs.pop()
6
sage: M.elementary_divisors()
[1, 1, 6]
```

..SEEALSO:: `smith_form()`

hermite_form (*algorithm='default'*, *cutoff=0*, ***kwds*)

Return the echelon form of self.

Note: This row reduction does not use division if the matrix is not over a field (e.g., if the matrix is over the integers). If you want to calculate the echelon form using division, then use `rref()` , which assumes that the matrix entries are in a field (specifically, the field of fractions of the base ring of the matrix).

INPUT:

- `algorithm` – string. Which algorithm to use. Choices are
  - `'default'` : Let Sage choose an algorithm (default).
  - `'classical'` : Gauss elimination.
  - `'strassen'` : use a Strassen divide and conquer algorithm (if available)
- `cutoff` – integer. Only used if the Strassen algorithm is selected.
- `transformation` – boolean. Whether to also return the transformation matrix. Some matrix backends do not provide this information, in which case this option is ignored.

OUTPUT:

The reduced row echelon form of `self` , as an immutable matrix. Note that self is *not* changed by this command. Use `echelonize()` to change `self` in place.

If the optional parameter `transformation=True` is specified, the output consists of a pair (E, T) of matrices where E is the echelon form of `self` and T is the transformation matrix.

EXAMPLES:

```
sage: MS = MatrixSpace(GF(19),2,3)
sage: C = MS.matrix([1,2,3,4,5,6])
sage: C.rank()
2
sage: C.nullity()
0
sage: C.echelon_form()
[ 1  0 18]
[ 0  1  2]
```

The matrix library used for $\mathbf{Z}/p$-matrices does not return the transformation matrix, so the `transformation` option is ignored:

```
sage: C.echelon_form(transformation=True)
[ 1  0 18]
[ 0  1  2]

sage: D = matrix(ZZ, 2, 3, [1,2,3,4,5,6])
sage: D.echelon_form(transformation=True)
(
[1 2 3]  [ 1  0]
[0 3 6], [ 4 -1]
)
sage: E, T = D.echelon_form(transformation=True)
sage: T*D == E
True
```

rational_reconstruction (*N*)

Use rational reconstruction to lift self to a matrix over the rational numbers (if possible), where we view self as a matrix modulo N.

EXAMPLES:

```
sage: A = matrix(ZZ, 3, 4, [(1/3)%500, 2, 3, (-4)%500, 7, 2, 2, 3, 4, 3, 4,
↪(5/7)%500], sparse=True)
sage: A.rational_reconstruction(500)
[1/3   2   3  -4]
[  7   2   2   3]
[  4   3   4 5/7]
```

smith_form ( )

Returns matrices S, U, and V such that S = U*self*V, and S is in Smith normal form. Thus S is diagonal with diagonal entries the ordered elementary divisors of S.

This version is for sparse matrices and simply makes the matrix dense and calls the version for dense integer matrices.

Warning: The elementary_divisors function, which returns the diagonal entries of S, is VASTLY faster than this function.

The elementary divisors are the invariants of the finite abelian group that is the cokernel of this matrix. They are ordered in reverse by divisibility.

EXAMPLES:

```
sage: A = MatrixSpace(IntegerRing(), 3, sparse=True)(range(9))
sage: D, U, V = A.smith_form()
sage: D
[1 0 0]
[0 3 0]
[0 0 0]
sage: U
[ 0  1  0]
[ 0 -1  1]
[-1  2 -1]
sage: V
[-1  4  1]
[ 1 -3 -2]
[ 0  0  1]
sage: U*A*V
```

```
[1 0 0]
[0 3 0]
[0 0 0]
```

It also makes sense for nonsquare matrices:

```
sage: A = Matrix(ZZ,3,2,range(6), sparse=True)
sage: D, U, V = A.smith_form()
sage: D
[1 0]
[0 2]
[0 0]
sage: U
[ 0  1  0]
[ 0 -1  1]
[-1  2 -1]
sage: V
[-1  3]
[ 1 -2]
sage: U * A * V
[1 0]
[0 2]
[0 0]
```

The examples above show that trac ticket #10626 has been implemented.

See also:

elementary_divisors()

CHAPTER

THIRTYTHREE

DENSE MATRICES OVER GF(2) USING THE M4RI LIBRARY.

AUTHOR: Martin Albrecht <malb@informatik.uni-bremen.de>

EXAMPLES:

```
sage: a = matrix(GF(2),3,range(9),sparse=False); a
[0 1 0]
[1 0 1]
[0 1 0]
sage: a.rank()
2
sage: type(a)
<type 'sage.matrix.matrix_mod2_dense.Matrix_mod2_dense'>
sage: a[0,0] = 1
sage: a.rank()
3
sage: parent(a)
Full MatrixSpace of 3 by 3 dense matrices over Finite Field of size 2

sage: a^2
[0 1 1]
[1 0 0]
[1 0 1]
sage: a+a
[0 0 0]
[0 0 0]
[0 0 0]

sage: b = a.new_matrix(2,3,range(6)); b
[0 1 0]
[1 0 1]

sage: a*b
Traceback (most recent call last):
...
TypeError: unsupported operand parent(s) for *: 'Full MatrixSpace of 3 by 3 dense␣
↪matrices over Finite Field of size 2' and 'Full MatrixSpace of 2 by 3 dense␣
↪matrices over Finite Field of size 2'
sage: b*a
[1 0 1]
[1 0 0]

sage: TestSuite(a).run()
sage: TestSuite(b).run()

sage: a.echelonize(); a
```

```
[1 0 0]
[0 1 0]
[0 0 1]
sage: b.echelonize(); b
[1 0 1]
[0 1 0]
```

TODO:

- make LinBox frontend and use it
  - charpoly ?
  - minpoly ?
- make Matrix_modn_frontend and use it (?)

class sage.matrix.matrix_mod2_dense.**Matrix_mod2_dense**
Bases: *sage.matrix.matrix_dense.Matrix_dense*

Dense matrix over GF(2).

augment (*right*, *subdivide=False*)
Augments `self` with `right`.

EXAMPLES:

```
sage: MS = MatrixSpace(GF(2),3,3)
sage: A = MS([0, 1, 0, 1, 1, 0, 1, 1, 1]); A
[0 1 0]
[1 1 0]
[1 1 1]
sage: B = A.augment(MS(1)); B
[0 1 0 1 0 0]
[1 1 0 0 1 0]
[1 1 1 0 0 1]
sage: B.echelonize(); B
[1 0 0 1 1 0]
[0 1 0 1 0 0]
[0 0 1 0 1 1]
sage: C = B.matrix_from_columns([3,4,5]); C
[1 1 0]
[1 0 0]
[0 1 1]
sage: C == ~A
True
sage: C*A == MS(1)
True
```

A vector may be augmented to a matrix.

```
sage: A = matrix(GF(2), 3, 4, range(12))
sage: v = vector(GF(2), 3, range(3))
sage: A.augment(v)
[0 1 0 1 0]
[0 1 0 1 1]
[0 1 0 1 0]
```

The `subdivide` option will add a natural subdivision between `self` and `right`. For more details about how subdivisions are managed when augmenting, see *sage.matrix.matrix1.Matrix.augment()*.

```
sage: A = matrix(GF(2), 3, 5, range(15))
sage: B = matrix(GF(2), 3, 3, range(9))
sage: A.augment(B, subdivide=True)
[0 1 0 1 0|0 1 0]
[1 0 1 0 1|1 0 1]
[0 1 0 1 0|0 1 0]
```

density (*approx=False*)

Return the density of this matrix.

By density we understand the ration of the number of nonzero positions and the self.nrows() * self.ncols(), i.e. the number of possible nonzero positions.

INPUT:

- approx – return floating point approximation (default: False)

EXAMPLES:

```
sage: A = random_matrix(GF(2),1000,1000)
sage: d = A.density(); d
62483/125000

sage: float(d)
0.499864

sage: A.density(approx=True)
0.499864000...

sage: float(len(A.nonzero_positions())/1000^2)
0.499864
```

determinant ( )

Return the determinant of this matrix over GF(2).

EXAMPLES:

```
sage: matrix(GF(2),2,[1,1,0,1]).determinant()
1
sage: matrix(GF(2),2,[1,1,1,1]).determinant()
0
```

echelonize (*algorithm='heuristic'*, *cutoff=0*, *reduced=True*, ***kwds*)

Puts self in (reduced) row echelon form.

INPUT:

- self – a mutable matrix
- algorithm
  - 'heuristic' – uses M4RI and PLUQ (default)
  - 'm4ri' – uses M4RI
  - 'pluq' – uses PLUQ factorization
  - 'classical' – uses classical Gaussian elimination
- k – the parameter 'k' of the M4RI algorithm. It MUST be between 1 and 16 (inclusive). If it is not specified it will be calculated as 3/4 * log_2(min(nrows, ncols)) as suggested in the M4RI paper.
- reduced – return reduced row echelon form (default:True)

EXAMPLES:

```
sage: A = random_matrix(GF(2), 10, 10)
sage: B = A.__copy__(); B.echelonize() # fastest
sage: C = A.__copy__(); C.echelonize(k=2) # force k
sage: E = A.__copy__(); E.echelonize(algorithm='classical') # force Gaussian␣
↪elimination
sage: B == C == E
True
```

ALGORITHM:

Uses M4RI library

REFERENCES:

- [Bar2006]

randomize (*density=1*, *nonzero=False*)

Randomize `density` proportion of the entries of this matrix, leaving the rest unchanged.

INPUT:

- `density` - float; proportion (roughly) to be considered for changes
- `nonzero` - Bool (default: `False`); whether the new entries are forced to be non-zero

OUTPUT:

- None, the matrix is modified in-space

EXAMPLES:

```
sage: A = matrix(GF(2), 5, 5, 0)
sage: A.randomize(0.5); A
[0 0 0 1 1]
[0 1 0 0 1]
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 0 1 0]
sage: A.randomize(); A
[0 0 1 1 0]
[1 1 0 0 1]
[1 1 1 1 0]
[1 1 1 1 1]
[0 0 1 1 0]
```

rank (*algorithm='ple'*)

Return the rank of this matrix.

On average 'ple' should be faster than 'm4ri' and hence it is the default choice. However, for small - i.e. quite few thousand rows & columns - and sparse matrices 'm4ri' might be a better choice.

INPUT:

- `algorithm` - either "ple" or "m4ri"

EXAMPLES:

```
sage: A = random_matrix(GF(2), 1000, 1000)
sage: A.rank()
999
```

```
sage: A = matrix(GF(2),10, 0)
sage: A.rank()
0
```

row (*i*, *from_list=False*)
Return the i 'th row of this matrix as a vector.

This row is a dense vector if and only if the matrix is a dense matrix.

INPUT:

- i - integer
- from_list - bool (default: False); if True , returns the i 'th element of self.rows() (see rows()), which may be faster, but requires building a list of all rows the first time it is called after an entry of the matrix is changed.

EXAMPLES:

```
sage: A = random_matrix(GF(2),10,10); A
[0 1 0 1 1 0 0 0 1 1]
[0 1 1 1 0 1 1 0 0 1]
[0 0 0 1 0 1 0 0 1 0]
[0 1 1 0 0 1 0 1 1 0]
[0 0 0 1 1 1 1 0 1 1]
[0 0 1 1 1 1 0 0 0 0]
[1 1 1 1 0 1 0 1 1 0]
[0 0 0 1 1 0 0 0 1 1]
[1 0 0 0 1 1 1 0 1 1]
[1 0 0 1 1 0 1 0 0 0]

sage: A.row(0)
(0, 1, 0, 1, 1, 0, 0, 0, 1, 1)

sage: A.row(-1)
(1, 0, 0, 1, 1, 0, 1, 0, 0, 0)

sage: A.row(2,from_list=True)
(0, 0, 0, 1, 0, 1, 0, 0, 1, 0)

sage: A = Matrix(GF(2),1,0)
sage: A.row(0)
()
```

str (*rep_mapping=None*, *zero=None*, *plus_one=None*, *minus_one=None*)
Return a nice string representation of the matrix.

INPUT:

- rep_mapping - a dictionary or callable used to override the usual representation of elements. For a dictionary, keys should be elements of the base ring and values the desired string representation.
- zero - string (default: None); if not None use the value of zero as the representation of the zero element.
- plus_one - string (default: None); if not None use the value of plus_one as the representation of the one element.
- minus_one - Ignored. Only for compatibility with generic matrices.

EXAMPLES:

```
sage: B = random_matrix(GF(2),3,3)
sage: B # indirect doctest
[0 1 0]
[0 1 1]
[0 0 0]
sage: block_matrix([[B, 1], [0, B]])
[0 1 0|1 0 0]
[0 1 1|0 1 0]
[0 0 0|0 0 1]
[-----+-----]
[0 0 0|0 1 0]
[0 0 0|0 1 1]
[0 0 0|0 0 0]
sage: B.str(zero='.')
'[. 1 .]\n[. 1 1]\n[. . .]'
```

submatrix (*row=0*, *col=0*, *nrows=-1*, *ncols=-1*)

Return submatrix from the index row, col (inclusive) with dimension nrows x ncols.

INPUT:

- row – index of start row
- col – index of start column
- nrows – number of rows of submatrix
- ncols – number of columns of submatrix

EXAMPLES:

```
sage: A = random_matrix(GF(2),200,200)
sage: A[0:2,0:2] == A.submatrix(0,0,2,2)
True
sage: A[0:100,0:100] == A.submatrix(0,0,100,100)
True
sage: A == A.submatrix(0,0,200,200)
True

sage: A[1:3,1:3] == A.submatrix(1,1,2,2)
True
sage: A[1:100,1:100] == A.submatrix(1,1,99,99)
True
sage: A[1:200,1:200] == A.submatrix(1,1,199,199)
True
```

TESTS for handling of default arguments (trac ticket #18761):

```
sage: A.submatrix(17,15) == A.submatrix(17,15,183,185)
True
sage: A.submatrix(row=100,col=37,nrows=1,ncols=3) == A.submatrix(100,37,1,3)
True
```

transpose ( )

Returns transpose of self and leaves self untouched.

EXAMPLES:

```
sage: A = Matrix(GF(2),3,5,[1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0])
sage: A
```

```
[1 0 1 0 0]
[0 1 1 0 0]
[1 1 0 1 0]
sage: B = A.transpose(); B
[1 0 1]
[0 1 1]
[1 1 0]
[0 0 1]
[0 0 0]
sage: B.transpose() == A
True
```

.T is a convenient shortcut for the transpose:

```
sage: A.T
[1 0 1]
[0 1 1]
[1 1 0]
[0 0 1]
[0 0 0]
```

sage.matrix.matrix_mod2_dense.**free_m4ri**()

Free global Gray code tables.

sage.matrix.matrix_mod2_dense.**from_png**(*filename*)

Returns a dense matrix over GF(2) from a 1-bit PNG image read from `filename`. No attempt is made to verify that the filename string actually points to a PNG image.

INPUT:

- filename – a string

EXAMPLES:

```
sage: from sage.matrix.matrix_mod2_dense import from_png, to_png
sage: A = random_matrix(GF(2),10,10)
sage: fn = tmp_filename()
sage: to_png(A, fn)
sage: B = from_png(fn)
sage: A == B
True
```

sage.matrix.matrix_mod2_dense.**parity**(*a*)

Returns the parity of the number of bits in a.

EXAMPLES:

```
sage: from sage.matrix.matrix_mod2_dense import parity
sage: parity(1)
1L
sage: parity(3)
0L
sage: parity(0x10000101011)
1L
```

sage.matrix.matrix_mod2_dense.**ple**(*A, algorithm='standard', param=0*)

Return PLE factorization of A.

INPUT:

- A – matrix
- algorithm
  - 'standard' asymptotically fast (default)
  - 'russian' M4RI inspired
  - 'naive' naive cubic
- param – either k for 'mmpf' is chosen or matrix multiplication cutoff for 'standard' (default: 0)

EXAMPLES:

```
sage: from sage.matrix.matrix_mod2_dense import ple
sage: A = random_matrix(GF(2),4,4); A
[0 1 0 1]
[0 1 1 1]
[0 0 0 1]
[0 1 1 0]

sage: LU, P, Q = ple(A)
sage: LU
[1 0 0 1]
[1 1 0 0]
[0 0 1 0]
[1 1 1 0]

sage: P
[0, 1, 2, 3]

sage: Q
[1, 2, 3, 3]

sage: A = random_matrix(GF(2),1000,1000)
sage: ple(A) == ple(A,'russian') == ple(A,'naive')
True
```

`sage.matrix.matrix_mod2_dense.`**`pluq`**`(`*A*, *algorithm='standard'*, *param=0*`)`

Return PLUQ factorization of A.

INPUT:

- A – matrix
- algorithm
  - 'standard' asymptotically fast (default)
  - 'mmpf' M4RI inspired
  - 'naive' naive cubic
- param – either k for 'mmpf' is chosen or matrix multiplication cutoff for 'standard' (default: 0)

EXAMPLES:

```
sage: from sage.matrix.matrix_mod2_dense import pluq
sage: A = random_matrix(GF(2),4,4); A
[0 1 0 1]
[0 1 1 1]
[0 0 0 1]
[0 1 1 0]
```

```
sage: LU, P, Q = pluq(A)
sage: LU
[1 0 1 0]
[1 1 0 0]
[0 0 1 0]
[1 1 1 0]

sage: P
[0, 1, 2, 3]

sage: Q
[1, 2, 3, 3]
```

sage.matrix.matrix_mod2_dense.**to_png**(*A*, *filename*)

Saves the matrix A to filename as a 1-bit PNG image.

INPUT:

- A - a matrix over GF(2)
- filename - a string for a file in a writable position

EXAMPLES:

```
sage: from sage.matrix.matrix_mod2_dense import from_png, to_png
sage: A = random_matrix(GF(2),10,10)
sage: fn = tmp_filename()
sage: to_png(A, fn)
sage: B = from_png(fn)
sage: A == B
True
```

sage.matrix.matrix_mod2_dense.**unpickle_matrix_mod2_dense_v1**(*r*, *c*, *data*, *size*)

Deserialize a matrix encoded in the string s.

INPUT:

- r – number of rows of matrix
- c – number of columns of matrix
- s – a string
- size – length of the string s

EXAMPLES:

```
sage: A = random_matrix(GF(2),100,101)
sage: _,(r,c,s,s2) = A.__reduce__()
sage: from sage.matrix.matrix_mod2_dense import unpickle_matrix_mod2_dense_v1
sage: unpickle_matrix_mod2_dense_v1(r,c,s,s2) == A
True
sage: loads(dumps(A)) == A
True
```

CHAPTER

THIRTYFOUR

DENSE MATRICES OVER $\mathbf{F}_{2^E}$ FOR $2 <= E <= 10$ USING THE M4RIE LIBRARY.

The M4RIE library offers two matrix representations:

1. `mzed_t`

   m x n matrices over $\mathbf{F}_{2^e}$ are internally represented roughly as m x (en) matrices over $\mathbf{F}_2$. Several elements are packed into words such that each element is filled with zeroes until the next power of two. Thus, for example, elements of $\mathbf{F}_{2^3}$ are represented as `[0xxx|0xxx|0xxx|0xxx|...]`. This representation is wrapped as *`Matrix_gf2e_dense`* in Sage.

   Multiplication and elimination both use "Newton-John" tables. These tables are simply all possible multiples of a given row in a matrix such that a scale+add operation is reduced to a table lookup + add. On top of Newton-John multiplication M4RIE implements asymptotically fast Strassen-Winograd multiplication. Elimination uses simple Gaussian elimination which requires $O(n^3)$ additions but only $O(n^2 * 2^e)$ multiplications.

2. `mzd_slice_t`

   m x n matrices over $\mathbf{F}_{2^e}$ are internally represented as slices of m x n matrices over $\mathbf{F}_2$. This representation allows for very fast matrix times matrix products using Karatsuba's polynomial multiplication for polynomials over matrices. However, it is not feature complete yet and hence not wrapped in Sage for now.

See http://m4ri.sagemath.org for more details on the M4RIE library.

EXAMPLES:

```
sage: K.<a> = GF(2^8)
sage: A = random_matrix(K, 3,4)
sage: A
[          a^6 + a^5 + a^4 + a^2               a^6 + a^3 + a + 1           a^5 + a^3 +␣
↪a^2 + a + 1               a^7 + a^6 + a + 1]
[                a^7 + a^6 + a^3             a^7 + a^6 + a^5 + 1           a^5 + a^4 +␣
↪a^3 + a + 1 a^6 + a^5 + a^4 + a^3 + a^2 + 1]
[              a^6 + a^5 + a + 1                   a^7 + a^3 + 1                 a^7 +␣
↪a^3 + a + 1   a^7 + a^6 + a^3 + a^2 + a + 1]

sage: A.echelon_form()
[                        1                          0                          0      a^
↪6 + a^5 + a^4 + a^2]
[                        0                          1                          0    a^7␣
↪+ a^5 + a^3 + a + 1]
[                        0                          0                          1 a^6 +␣
↪a^4 + a^3 + a^2 + 1]
```

AUTHOR:

- Martin Albrecht <martinralbrecht@googlemail.com>

TODO:

- wrap `mzd_slice_t`

REFERENCES:

- [BB2009]

class `sage.matrix.matrix_gf2e_dense.`**`M4RIE_finite_field`**

Bases: `object`

A thin wrapper around the M4RIE finite field class such that we can put it in a hash table. This class is not meant for public consumption.

class `sage.matrix.matrix_gf2e_dense.`**`Matrix_gf2e_dense`**

Bases: `sage.matrix.matrix_dense.Matrix_dense`

Create new matrix over $GF(2^e)$ for 2<=e<=10.

INPUT:

- `parent` - a `MatrixSpace`.
- `entries` - may be list or a finite field element.
- `copy` - ignored, elements are always copied
- `coerce` - ignored, elements are always coerced

EXAMPLES:

```
sage: K.<a> = GF(2^4)
sage: l = [K.random_element() for _ in range(3*4)]; l
[a^2 + 1, a^3 + 1, 0, 0, a, a^3 + a + 1, a + 1, a + 1, a^2, a^3 + a + 1, a^3 + a,
↪a^3 + a]

sage: A = Matrix(K, 3, 4, l); A
[    a^2 + 1     a^3 + 1           0           0]
[          a a^3 + a + 1       a + 1       a + 1]
[        a^2 a^3 + a + 1     a^3 + a     a^3 + a]

sage: A.list()
[a^2 + 1, a^3 + 1, 0, 0, a, a^3 + a + 1, a + 1, a + 1, a^2, a^3 + a + 1, a^3 + a,
↪a^3 + a]

sage: l[0], A[0,0]
(a^2 + 1, a^2 + 1)

sage: A = Matrix(K, 3, 3, a); A
[a 0 0]
[0 a 0]
[0 0 a]
```

`augment` (*right*)

Augments `self` with `right`.

INPUT:

- `right` - a matrix

EXAMPLES:

```
sage: K.<a> = GF(2^4)
sage: MS = MatrixSpace(K,3,3)
sage: A = random_matrix(K,3,3)
sage: B = A.augment(MS(1)); B
[               a^2         a^3 + a + 1 a^3 + a^2 + a + 1                 1       ␣
↪             0                   0]
[             a + 1                 a^3                 1                 0       ␣
↪             1                   0]
[       a^3 + a + 1       a^3 + a^2 + 1             a + 1                 0       ␣
↪             0                   1]

sage: B.echelonize(); B
[                 1                   0                 0           a^2 + a       ␣
↪       a^3 + 1             a^3 + a]
[                 0                   1                 0     a^3 + a^2 + a a^3␣
↪+ a^2 + a + 1             a^2 + a]
[                 0                   0                 1             a + 1       ␣
↪           a^3                 a^3]

sage: C = B.matrix_from_columns([3,4,5]); C
[          a^2 + a             a^3 + 1             a^3 + a]
[    a^3 + a^2 + a a^3 + a^2 + a + 1             a^2 + a]
[            a + 1                 a^3                 a^3]

sage: C == ~A
True

sage: C*A == MS(1)
True
```

cling (**C*)

Pack the matrices over $\mathbf{F}_2$ into this matrix over $\mathbf{F}_{2^e}$.

Elements in $\mathbf{F}_{2^e}$ can be represented as $\sum c_i a^i$ where a is a root the minimal polynomial. If this matrix is A then this function writes $c_i a^i$ to the entry $A[x, y]$ where c_i is the entry $C_i[x, y]$.

INPUT:

- `C` - a list of matrices over GF(2)

EXAMPLES:

```
sage: K.<a> = GF(2^2)
sage: A = matrix(K, 5, 5)
sage: A0 = random_matrix(GF(2), 5, 5); A0
[0 1 0 1 1]
[0 1 1 1 0]
[0 0 0 1 0]
[0 1 1 0 0]
[0 0 0 1 1]

sage: A1 = random_matrix(GF(2), 5, 5); A1
[0 0 1 1 1]
[1 1 1 1 0]
[0 0 0 1 1]
[1 0 0 0 1]
[1 0 0 1 1]

sage: A.cling(A1, A0); A
```

```
[      0      a      1 a + 1 a + 1]
[      1 a + 1 a + 1 a + 1      0]
[      0      0      0 a + 1      1]
[      1      a      a      0      1]
[      1      0      0 a + 1 a + 1]

sage: A0[0,3]*a + A1[0,3], A[0,3]
(a + 1, a + 1)
```

Slicing and clinging are inverse operations:

```
sage: B1, B0 = A.slice()
sage: B0 == A0 and B1 == A1
True
```

echelonize (*algorithm='heuristic'*, *reduced=True*, ***kwds*)

Compute the row echelon form of `self` in place.

INPUT:

- `algorithm` - one of the following - `heuristic` - let M4RIE decide (default) - `newton_john` - use newton_john table based algorithm - `ple` - use PLE decomposition - `naive` - use naive cubic Gaussian elimination (M4RIE implementation) - `builtin` - use naive cubic Gaussian elimination (Sage implementation)
- `reduced` - if `True` return reduced echelon form. No guarantee is given that the matrix is *not* reduced if `False` (default: `True`)

EXAMPLES:

```
sage: K.<a> = GF(2^4)
sage: m,n  = 3, 5
sage: A = random_matrix(K, 3, 5); A
[              a^2         a^3 + a + 1 a^3 + a^2 + a + 1               a + 1     ␣
↪         a^3]
[                1         a^3 + a + 1     a^3 + a^2 + 1               a + 1     ␣
↪     a^3 + 1]
[      a^3 + a + 1 a^3 + a^2 + a + 1           a^2 + a             a^2 + 1     ␣
↪     a^2 + a]

sage: A.echelonize(); A
[             1              0              0          a + 1        a^2 + 1]
[             0              1              0            a^2          a + 1]
[             0              0              1 a^3 + a^2 + a            a^3]

sage: K.<a> = GF(2^3)
sage: m,n  = 3, 5
sage: MS = MatrixSpace(K,m,n)
sage: A = random_matrix(K, 3, 5)

sage: copy(A).echelon_form('newton_john')
[          1           0       a + 1           0     a^2 + 1]
[          0           1 a^2 + a + 1           0           a]
[          0           0           0           1 a^2 + a + 1]

sage: copy(A).echelon_form('naive');
[          1           0       a + 1           0     a^2 + 1]
[          0           1 a^2 + a + 1           0           a]
[          0           0           0           1 a^2 + a + 1]
```

```
sage: copy(A).echelon_form('builtin');
[          1           0       a + 1           0     a^2 + 1]
[          0           1 a^2 + a + 1           0           a]
[          0           0           0           1 a^2 + a + 1]
```

randomize (*density=1*, *nonzero=False*, **args*, ***kwds*)

Randomize `density` proportion of the entries of this matrix, leaving the rest unchanged.

INPUT:

- `density` - float; proportion (roughly) to be considered for changes
- `nonzero` - Bool (default: `False`); whether the new entries are forced to be non-zero

OUTPUT:

- None, the matrix is modified in-place

EXAMPLES:

```
sage: K.<a> = GF(2^4)
sage: A = Matrix(K,3,3)

sage: A.randomize(); A
[              a^2       a^3 + a + 1 a^3 + a^2 + a + 1]
[            a + 1               a^3                 1]
[      a^3 + a + 1     a^3 + a^2 + 1             a + 1]

sage: K.<a> = GF(2^4)
sage: A = random_matrix(K,1000,1000,density=0.1)
sage: float(A.density())
0.0999...

sage: A = random_matrix(K,1000,1000,density=1.0)
sage: float(A.density())
1.0

sage: A = random_matrix(K,1000,1000,density=0.5)
sage: float(A.density())
0.4996...
```

Note, that the matrix is updated and not zero-ed out before being randomized:

```
sage: A = matrix(K,1000,1000)
sage: A.randomize(nonzero=False, density=0.1)
sage: float(A.density())
0.0936...

sage: A.randomize(nonzero=False, density=0.05)
sage: float(A.density())
0.135854
```

rank ( )

Return the rank of this matrix (cached).

EXAMPLES:

```
sage: K.<a> = GF(2^4)
sage: A = random_matrix(K, 1000, 1000)
```

```
sage: A.rank()
1000

sage: A = matrix(K, 10, 0)
sage: A.rank()
0
```

slice ( )

Unpack this matrix into matrices over $\mathbf{F}_2$.

Elements in $\mathbf{F}_{2^e}$ can be represented as $\sum c_i a^i$ where a is a root the minimal polynomial. This function returns a tuple of matrices C whose entry $C_i[x, y]$ is the coefficient of c_i in $A[x, y]$ if this matrix is A.

EXAMPLES:

```
sage: K.<a> = GF(2^2)
sage: A = random_matrix(K, 5, 5); A
[    0 a + 1 a + 1 a + 1     0]
[    1 a + 1     1 a + 1     1]
[a + 1 a + 1     a     1     a]
[    a     1 a + 1     1     0]
[    a     1 a + 1 a + 1     0]

sage: A1,A0 = A.slice()
sage: A0
[0 1 1 1 0]
[0 1 0 1 0]
[1 1 1 0 1]
[1 0 1 0 0]
[1 0 1 1 0]

sage: A1
[0 1 1 1 0]
[1 1 1 1 1]
[1 1 0 1 0]
[0 1 1 1 0]
[0 1 1 1 0]

sage: A0[2,4]*a + A1[2,4], A[2,4]
(a, a)

sage: K.<a> = GF(2^3)
sage: A = random_matrix(K, 5, 5); A
[      a + 1     a^2 + a           1           a     a^2 + a]
[      a + 1     a^2 + a         a^2         a^2     a^2 + 1]
[a^2 + a + 1 a^2 + a + 1           0 a^2 + a + 1     a^2 + 1]
[    a^2 + a           0 a^2 + a + 1           a           a]
[        a^2       a + 1           a     a^2 + 1 a^2 + a + 1]

sage: A0,A1,A2 = A.slice()
sage: A0
[1 0 1 0 0]
[1 0 0 0 1]
[1 1 0 1 1]
[0 0 1 0 0]
[0 1 0 1 1]
```

Slicing and clinging are inverse operations:

```
sage: B = matrix(K, 5, 5)
sage: B.cling(A0,A1,A2)
sage: B == A
True
```

stack (*other*)

Stack `self` on top of `other`.

INPUT:

- `other` - a matrix

EXAMPLES:

```
sage: K.<a> = GF(2^4)
sage: A = random_matrix(K,2,2); A
[              a^2       a^3 + a + 1]
[a^3 + a^2 + a + 1             a + 1]

sage: B = random_matrix(K,2,2); B
[          a^3                1]
[  a^3 + a + 1 a^3 + a^2 + 1]

sage: A.stack(B)
[              a^2       a^3 + a + 1]
[a^3 + a^2 + a + 1             a + 1]
[              a^3                 1]
[      a^3 + a + 1     a^3 + a^2 + 1]

sage: B.stack(A)
[              a^3                 1]
[      a^3 + a + 1     a^3 + a^2 + 1]
[              a^2       a^3 + a + 1]
[a^3 + a^2 + a + 1             a + 1]
```

submatrix (*row=0*, *col=0*, *nrows=-1*, *ncols=-1*)

Return submatrix from the index `row,col` (inclusive) with dimension `nrows x ncols`.

INPUT:

- `row` – index of start row
- `col` – index of start column
- `nrows` – number of rows of submatrix
- `ncols` – number of columns of submatrix

EXAMPLES:

```
sage: K.<a> = GF(2^10)
sage: A = random_matrix(K,200,200)
sage: A[0:2,0:2] == A.submatrix(0,0,2,2)
True
sage: A[0:100,0:100] == A.submatrix(0,0,100,100)
True
sage: A == A.submatrix(0,0,200,200)
True

sage: A[1:3,1:3] == A.submatrix(1,1,2,2)
True
```

```
sage: A[1:100,1:100] == A.submatrix(1,1,99,99)
True
sage: A[1:200,1:200] == A.submatrix(1,1,199,199)
True
```

TESTS for handling of default arguments (trac ticket #18761):

```
sage: A.submatrix(17,15) == A.submatrix(17,15,183,185)
True
sage: A.submatrix(row=100,col=37,nrows=1,ncols=3) == A.submatrix(100,37,1,3)
True
```

sage.matrix.matrix_gf2e_dense.**unpickle_matrix_gf2e_dense_v0**(*a*, *base_ring*, *nrows*, *ncols*)

EXAMPLES:

```
sage: K.<a> = GF(2^2)
sage: A = random_matrix(K,10,10)
sage: f, s= A.__reduce__()
sage: from sage.matrix.matrix_gf2e_dense import unpickle_matrix_gf2e_dense_v0
sage: f == unpickle_matrix_gf2e_dense_v0
True
sage: f(*s) == A
True
```

We can still unpickle pickles from before trac ticket #19240:

```
sage: old_pickle = 'x\x9c\x85RKo\xd3@\x10\xae\xdd$$\xdb&\xe5U\x1e-
↪\x8f\xc2\xc9\x12RD#$\xce\xa0\xb4\x80\x07\xa2\xca\xc2\x07\x0e\xd5\xe2:
↪\x1b\xdb\x8acg\x1c\xa7J\x85*!\xa4\x90\xe6\x07p\xe0\xc4\x01q\xe5\xc4\x19\xf5\xd0?
↪\xc1\x81\xdf\x80\xb8q\x0b\xb3\x8eMS\xa1\x82V;;
↪\xb3\xdf\xce\xf7\xcd\x8e\xe6\xb5j\xf7,GT;
↪V\x1cy\x83\xf4\xe0\x9d\xb0Y\x13\xbc)\x82\x9e`\xfd\xa0\xeb\xd9m_\xf0\xbf1\xbe
↪{\x97\xa1\xa2\x9d\xc6\xf0\x0f\x82,\x7f\x9d\xa1\xaa\x81\n\xb9m\x9c\xd7\xf4\xf1d2\x81-
↪h\xc0
↪#(\x03\x83\x15\xdas\xc9*\xc3\x13x\x0cu0\xd28\x97\x9e*(0\x9f\xfa\x1b\xd0\xd2\x7fH\x82\xb5\xf4\x
↪\xcd\x07i\xbe\xcb\xd4ib\t\xba\xa4\xf6\x02zIT\xd1\x8f2(u\x15\xfd\x9d
↪<\xee@\x05V\xd3\x94E*\xb0\x0e\x0fH\xad\xa8\xbf\x97\xa0\r\x03\xfd\xf0\xb8\x1aU\xff\x92\x90\xe8?
↪\xa5\xd6\x814_\xa5\xf9(\xcd\xafc\xe99\xe2\xd9\xa0\x06\xd4\xf5\xcf\xf2\xf2!
↪\xbc\xd4\xdf\x90#\xc0\x8f\r\xccM\x1b\xdd\x8b\xa3\xbe\x1d\xf7#QmYv\x1cF
↪{\xcc\x11\x81\x88<\x9b\xa71\xcf:
↪\xce0\xaf\x9d\x96\xe3\x87a\xbb\xdf\xe5\x8e\x1f\xeeX>\xc3\x82\xb9\xb0\xe9\x05^
↪,6=\xe17\xf1\xcc\xd0\xc0"u\xb0d\xe6wDl\xdd\x1fa)e\x8a\xbc\xc0\xe9U\xbd␣
↪\x16\x8e\x88X\xc7j\x0b\x9e\x05\xc8L\xe5\x1e%.
↪\x98\x8a5\xc4\xc5\xd9\xf7\xdd\xd0\xdf\x0b\xc2\x8eg\xf93.wZ\xb5\xc1\x94B\xf8\xa2
↪#\x82\x98a\xf9\xffY\x12\xe3v\x18L\xff\x14Fl\xeb\x0ff\x10\xc4\xb0\xa2\xb9y\xcd-
↪\xba%\xcd\xa5\x8ajT\xd1\x92\xa9\x0c\x86x\xb6a\xe6h\xf8\x02<g\xaa\xaf\xf6\xdd
↪%\x89\xae\x13z\xfe \xc6\x0b\xfb1^
↪4p\x99\x1e6\xc6\xd4\xebK\xdbx\xf9\xc4\x8f[Iw\xf8\x89\xef\xcbQf\xcfh\xe3\x95\x8c\xebj&
↪\xb9\xe2.\x8f\x0c\\ui\x89\xf1x\xf4\xd6\xc0kf\xc1\xf1v\xad(\xc4\xeb\x89~
↪\xfa\xf0\x06\xa8\xa4\x7f\x93\xf4\xd7\x0c\xbcE#\xad\x92\xfc\xed\xeao\xefX\\\x03'
sage: loads(old_pickle)
[    0     a]
[a + 1     1]
```

CHAPTER THIRTYFIVE

DENSE MATRICES OVER $\mathbf{Z}/N\mathbf{Z}$ FOR $N < 2^{23}$ USING LINBOX'S MODULAR<DOUBLE>

AUTHORS:

- Burcin Erocal
- Martin Albrecht

class sage.matrix.matrix_modn_dense_double. **Matrix_modn_dense_double**

Bases: *sage.matrix.matrix_modn_dense_double.Matrix_modn_dense_template*

Dense matrices over $\mathbf{Z}/n\mathbf{Z}$ for $n < 2^{23}$ using LinBox's `Modular<double>`

These are matrices with integer entries mod `n` represented as floating-point numbers in a 64-bit word for use with LinBox routines. This allows for `n` up to 2^{23}. The analogous `Matrix_modn_dense_float` class is used for smaller moduli.

Routines here are for the most basic access, see the $matrix_modn_dense_template.pxi$ file for higher-level routines.

class sage.matrix.matrix_modn_dense_double. **Matrix_modn_dense_template**

Bases: *sage.matrix.matrix_dense.Matrix_dense*

Create a new matrix.

INPUT:

- `parent` - a matrix space
- `entries` - a list of entries or a scalar
- `copy` - ignroed
- `coerce` - perform modular reduction first?

EXAMPLES:

```
sage: A = random_matrix(GF(3),1000,1000)
sage: type(A)
<type 'sage.matrix.matrix_modn_dense_float.Matrix_modn_dense_float'>
sage: A = random_matrix(Integers(10),1000,1000)
sage: type(A)
<type 'sage.matrix.matrix_modn_dense_float.Matrix_modn_dense_float'>
sage: A = random_matrix(Integers(2^16),1000,1000)
sage: type(A)
<type 'sage.matrix.matrix_modn_dense_double.Matrix_modn_dense_double'>
```

charpoly (*var='x'*, *algorithm='linbox'*)

Return the characteristic polynomial of `self`.

INPUT:

- `var` - a variable name
- `algorithm` - 'generic', 'linbox' or 'all' (default: linbox)

EXAMPLES:

```
sage: A = random_matrix(GF(19), 10, 10); A
[ 3  1  8 10  5 16 18  9  6  1]
[ 5 14  4  4 14 15  5 11  3  0]
[ 4  1  0  7 11  6 17  8  5  6]
[ 4  6  9  4  8  1 18 17  8 18]
[11  2  0  6 13  7  4 11 16 10]
[12  6 12  3 15 10  5 11  3  8]
[15  1 16  2 18 15 14  7  2 11]
[16 16 17  7 14 12  7  7  0  5]
[13 15  9  2 12 16  1 15 18  7]
[10  8 16 18  9 18  2 13  5 10]

sage: B = copy(A)
sage: char_p = A.characteristic_polynomial(); char_p
x^10 + 2*x^9 + 18*x^8 + 4*x^7 + 13*x^6 + 11*x^5 + 2*x^4 + 5*x^3 + 7*x^2 +␣
↪16*x + 6
sage: char_p(A) == 0
True
sage: B == A              # A is not modified
True

sage: min_p = A.minimal_polynomial(proof=True); min_p
x^10 + 2*x^9 + 18*x^8 + 4*x^7 + 13*x^6 + 11*x^5 + 2*x^4 + 5*x^3 + 7*x^2 +␣
↪16*x + 6
sage: min_p.divides(char_p)
True
```

```
sage: A = random_matrix(GF(2916337), 7, 7); A
[ 446196 2267054   36722 2092388 1694559  514193 1196222]
[1242955 1040744   99523 2447069   40527  930282 2685786]
[2892660 1347146 1126775 2131459  869381 1853546 2266414]
[2897342 1342067 1054026  373002   84731 1270068 2421818]
[ 569466  537440  572533  297105 1415002 2079710  355705]
[2546914 2299052 2883413 1558788 1494309 1027319 1572148]
[ 250822  522367 2516720  585897 2296292 1797050 2128203]

sage: B = copy(A)
sage: char_p = A.characteristic_polynomial(); char_p
x^7 + 1191770*x^6 + 547840*x^5 + 215639*x^4 + 2434512*x^3 + 1039968*x^2 +␣
↪483592*x + 733817
sage: char_p(A) == 0
True
sage: B == A              # A is not modified
True

sage: min_p = A.minimal_polynomial(proof=True); min_p
x^7 + 1191770*x^6 + 547840*x^5 + 215639*x^4 + 2434512*x^3 + 1039968*x^2 +␣
↪483592*x + 733817
sage: min_p.divides(char_p)
True
```

```
sage: A = Mat(Integers(6),3,3)(range(9))
sage: A.charpoly()
x^3
```

ALGORITHM: Uses LinBox if `self.base_ring()` is a field, otherwise use Hessenberg form algorithm.

determinant ( )

Return the determinant of this matrix.

EXAMPLES:

```
sage: A = random_matrix(GF(7), 10, 10); A
[3 1 6 6 4 4 2 2 3 5]
[4 5 6 2 2 1 2 5 0 5]
[3 2 0 5 0 1 5 4 2 3]
[6 4 5 0 2 4 2 0 6 3]
[2 2 4 2 4 5 3 4 4 4]
[2 5 2 5 4 5 1 1 1 1]
[0 6 3 4 2 2 3 5 1 1]
[4 2 6 5 6 3 4 5 5 3]
[5 2 4 3 6 2 3 6 2 1]
[3 3 5 3 4 2 2 1 6 2]

sage: A.determinant()
6
```

```
    sage: A = random_matrix(GF(7), 100, 100)
    sage: A.determinant()
    2

    sage: A.transpose().determinant()
    2

    sage: B = random_matrix(GF(7), 100, 100)
    sage: B.determinant()
    4

    sage: (A*B).determinant() == A.determinant() * B.determinant()
    True

::

    sage: A = random_matrix(GF(16007), 10, 10); A
    [ 5037  2388  4150  1400   345  5945  4240 14022 10514   700]
    [15552  8539  1927  3870  9867  3263 11637   609 15424  2443]
    [ 3761 15836 12246 15577 10178 13602 13183 15918 13942  2958]
    [ 4526 10817  6887  6678  1764  9964  6107  1705  5619  5811]
    [13537 15004  8307 11846 14779   550 14113  5477  7271  7091]
    [13338  4927 11406 13065  5437 12431  6318  5119 14198   496]
    [ 1044   179 12881   353 12975 12567  1092 10433 12304   954]
    [10072  8821 14118 13895  6543 13484 10685 14363  2612 11070]
    [15113   237  2612 14127 11589  5808   117  9656 15957 14118]
    [15233 11080  5716  9029 11402  9380 13045 13986 14544  5771]

    sage: A.determinant()
    10207
```

```
::

    sage: A = random_matrix(GF(16007), 100, 100)
    sage: A.determinant()
    3576


    sage: A.transpose().determinant()
    3576

    sage: B = random_matrix(GF(16007), 100, 100)
    sage: B.determinant()
    4075

    sage: (A*B).determinant() == A.determinant() * B.determinant()
    True
```

echelonize (*algorithm='linbox'*, ***kwds*)

Put `self` in reduced row echelon form.

INPUT:

- `self` - a mutable matrix
- `algorithm`
  - `linbox` - uses the LinBox library (`EchelonFormDomain` implementation, default)
  - `linbox_noefd` - uses the LinBox library (FFPACK directly, less memory but slower)
  - `gauss` - uses a custom slower $O(n^3)$ Gauss elimination implemented in Sage.
  - `all` - compute using both algorithms and verify that the results are the same.
- `**kwds` - these are all ignored

OUTPUT:

- `self` is put in reduced row echelon form.
- the rank of self is computed and cached
- the pivot columns of self are computed and cached.
- the fact that self is now in echelon form is recorded and cached so future calls to echelonize return immediately.

EXAMPLES:

```
sage: A = random_matrix(GF(7), 10, 20); A
[3 1 6 6 4 4 2 2 3 5 4 5 6 2 2 1 2 5 0 5]
[3 2 0 5 0 1 5 4 2 3 6 4 5 0 2 4 2 0 6 3]
[2 2 4 2 4 5 3 4 4 4 2 5 2 5 4 5 1 1 1 1]
[0 6 3 4 2 2 3 5 1 1 4 2 6 5 6 3 4 5 5 3]
[5 2 4 3 6 2 3 6 2 1 3 3 5 3 4 2 2 1 6 2]
[0 5 6 3 2 5 6 6 3 2 1 4 5 0 2 6 5 2 5 1]
[4 0 4 2 6 3 3 5 3 0 0 1 2 5 5 1 6 0 0 3]
[2 0 1 0 0 3 0 2 4 2 2 4 4 4 5 4 1 2 3 4]
[2 4 1 4 3 0 6 2 2 5 2 5 3 6 4 2 2 6 4 4]
[0 0 2 2 1 6 2 0 5 0 4 3 1 6 0 6 0 4 6 5]

sage: A.echelon_form()
[1 0 0 0 0 0 0 0 0 0 6 2 6 0 1 1 2 5 6 2]
```

```
[0 1 0 0 0 0 0 0 0 0 0 4 5 4 3 4 2 5 1 2]
[0 0 1 0 0 0 0 0 0 0 6 3 4 6 1 0 3 6 5 6]
[0 0 0 1 0 0 0 0 0 0 0 3 5 2 3 4 0 6 5 3]
[0 0 0 0 1 0 0 0 0 0 0 6 3 4 5 3 0 4 3 2]
[0 0 0 0 0 1 0 0 0 0 1 1 0 2 4 2 5 5 5 0]
[0 0 0 0 0 0 1 0 0 0 1 0 1 3 2 0 0 0 5 3]
[0 0 0 0 0 0 0 1 0 0 4 4 2 6 5 4 3 4 1 0]
[0 0 0 0 0 0 0 0 1 0 1 0 4 2 3 5 4 6 4 0]
[0 0 0 0 0 0 0 0 0 1 2 0 5 0 5 5 3 1 1 4]
```

```
sage: A = random_matrix(GF(13), 10, 10); A
[ 8  3 11 11  9  4  8  7  9  9]
[ 2  9  6  5  7 12  3  4 11  5]
[12  6 11 12  4  3  3  8  9  5]
[ 4  2 10  5 10  1  1  1  6  9]
[12  8  5  5 11  4  1  2  8 11]
[ 2  6  9 11  4  7  1  0 12  2]
[ 8  9  0  7  7  7 10  4  1  4]
[ 0  8  2  6  7  5  7 12  2  3]
[ 2 11 12  3  4  7  2  9  6  1]
[ 0 11  5  9  4  5  5  8  7 10]

sage: MS = parent(A)
sage: B = A.augment(MS(1))
sage: B.echelonize()
sage: A.rank()
10
sage: C = B.submatrix(0,10,10,10); C
[ 4  9  4  4  0  4  7 11  9 11]
[11  7  6  8  2  8  6 11  9  5]
[ 3  9  9  2  4  8  9  2  9  4]
[ 7  0 11  4  0  9  6 11  8  1]
[12 12  4 12  3 12  6  1  7 12]
[12  2 11  6  6  6  7  0 10  6]
[ 0  7  3  4  7 11 10 12  4  6]
[ 5 11  0  5  3 11  4 12  5 12]
[ 6  7  3  5  1  4 11  7  4  1]
[ 4  9  6  7 11  1  2 12  6  7]

sage: ~A == C
True
```

```
sage: A = random_matrix(Integers(10), 10, 20)
sage: A.echelon_form()
Traceback (most recent call last):
...
NotImplementedError: Echelon form not implemented over 'Ring of integers␣
↪modulo 10'.
```

:: sage: A = random_matrix(GF(16007), 10, 20); A [15455 1177 10072 4693 3887 4102 10746 15265 6684 14559 4535 13921 9757 9525 9301 8566 2460 9609 3887 6205] [8602 10035 1242 9776 162 7893 12619 6660 13250 1988 14263 11377 2216 1247 7261 8446 15081 14412 7371 7948] [12634 7602 905 9617 13557 2694 13039 4936 12208 15480 3787 11229 593 12462 5123 14167 6460 3649 5821 6736] [10554 2511 11685 12325 12287 6534 11636 5004 6468 3180 3607 11627 13436 5106 3138 13376 8641 9093 2297 5893] [1025 11376 10288 609 12330 3021 908 13012 2112 11505 56 5971 338 2317 2396 8561 5593 3782 7986 13173] [7607 588 6099 12749 10378 111 2852 10375

8996 7969 774 13498 12720 4378 6817 6707 5299 9406 13318 2863] [15545 538 4840 1885 8471 1303 11086 14168 1853 14263 3995 12104 1294 7184 1188 11901 15971 2899 4632 711] [584 11745 7540 15826 15027 5953 7097 14329 10889 12532 13309 15041 6211 1749 10481 9999 2751 11068 21 2795] [761 11453 3435 10596 2173 7752 15941 14610 1072 8012 9458 5440 612 10581 10400 101 11472 13068 7758 7898] [10658 4035 6662 655 7546 4107 6987 1877 4072 4221 7679 14579 2474 8693 8127 12999 11141 605 9404 10003] sage: A.echelon_form() [1 0 0 0 0 0 0 0 0 0 8416 8364 10318 1782 13872 4566 14855 7678 11899 2652] [0 1 0 0 0 0 0 0 0 0 4782 15571 3133 10964 5581 10435 9989 14303 5951 8048] [0 0 1 0 0 0 0 0 0 0 15688 6716 13819 4144 257 5743 14865 15680 4179 10478] [0 0 0 1 0 0 0 0 0 0 4307 9488 2992 9925 13984 15754 8185 11598 14701 10784] [0 0 0 0 1 0 0 0 0 0 927 3404 15076 1040 2827 9317 14041 10566 5117 7452] [0 0 0 0 0 1 0 0 0 0 1144 10861 5241 6288 9282 5748 3715 13482 7258 9401] [0 0 0 0 0 0 1 0 0 0 769 1804 1879 4624 6170 7500 11883 9047 874 597] [0 0 0 0 0 0 0 1 0 0 15591 13686 5729 11259 10219 13222 15177 15727 5082 11211] [0 0 0 0 0 0 0 0 1 0 8375 14939 13471 12221 8103 4212 11744 10182 2492 11068] [0 0 0 0 0 0 0 0 0 1 6534 396 6780 14734 1206 3848 7712 9770 10755 410]

```
sage: A = random_matrix(Integers(10000), 10, 20)
sage: A.echelon_form()
Traceback (most recent call last):
...
NotImplementedError: Echelon form not implemented over 'Ring of integers␣
↪modulo 10000'.
```

hessenbergize ( )

Transforms self in place to its Hessenberg form.

EXAMPLES:

```
sage: A = random_matrix(GF(17), 10, 10, density=0.1); A
[ 0  0  0  0 12  0  0  0  0  0]
[ 0  0  0  4  0  0  0  0  0  0]
[ 0  0  0  0  2  0  0  0  0  0]
[ 0 14  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0 10  0  0  0  0]
[ 0  0  0  0  0 16  0  0  0  0]
[ 0  0  0  0  0  0  6  0  0  0]
[15  0  0  0  0  0  0  0  0  0]
[ 0  0  0 16  0  0  0  0  0  0]
[ 0  5  0  0  0  0  0  0  0  0]
sage: A.hessenbergize(); A
[ 0  0  0  0  0  0  0 12  0  0]
[15  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  2  0  0]
[ 0  0  0  0 14  0  0  0  0  0]
[ 0  0  0  4  0  0  0  0  0  0]
[ 0  0  0  0  5  0  0  0  0  0]
[ 0  0  0  0  0  0  6  0  0  0]
[ 0  0  0  0  0  0  0  0  0 10]
[ 0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0 16]
```

lift ( )

Return the lift of this matrix to the integers.

EXAMPLES:

```
sage: A = matrix(GF(7),2,3,[1..6])
sage: A.lift()
```

```
[1 2 3]
[4 5 6]
sage: A.lift().parent()
Full MatrixSpace of 2 by 3 dense matrices over Integer Ring

sage: A = matrix(GF(16007),2,3,[1..6])
sage: A.lift()
[1 2 3]
[4 5 6]
sage: A.lift().parent()
Full MatrixSpace of 2 by 3 dense matrices over Integer Ring
```

Subdivisions are preserved when lifting:

```
sage: A.subdivide([], [1,1]); A
[1||2 3]
[4||5 6]
sage: A.lift()
[1||2 3]
[4||5 6]
```

minpoly (*var='x'*, *algorithm='linbox'*, *proof=None*)

Returns the minimal polynomial of`` self``.

INPUT:

- `var` - a variable name
- `algorithm` - `generic` or `linbox` (default: `linbox`)
- `proof` – (default: `True`); whether to provably return the true minimal polynomial; if `False` , we only guarantee to return a divisor of the minimal polynomial. There are also certainly cases where the computed results is frequently not exactly equal to the minimal polynomial (but is instead merely a divisor of it).

> **Warning:** If `proof=True` , minpoly is insanely slow compared to `proof=False` . This matters since proof=True is the default, unless you first type `proof.linear_algebra(False)` .

EXAMPLES:

```
sage: A = random_matrix(GF(17), 10, 10); A
[ 2 14  0 15 11 10 16  2  9  4]
[10 14  1 14  3 14 12 14  3 13]
[10  1 14  6  2 14 13  7  6 14]
[10  3  9 15  8  1  5  8 10 11]
[ 5 12  4  9 15  2  6 11  2 12]
[ 6 10 12  0  6  9  7  7  3  8]
[ 2  9  1  5 12 13  7 16  7 11]
[11  1  0  2  0  4  7  9  8 15]
[ 5  3 16  2 11 10 12 14  0  7]
[16  4  6  5  2  3 14 15 16  4]

sage: B = copy(A)
sage: min_p = A.minimal_polynomial(proof=True); min_p
x^10 + 13*x^9 + 10*x^8 + 9*x^7 + 10*x^6 + 4*x^5 + 10*x^4 + 10*x^3 + 12*x^2 +
↪14*x + 7
```

```
sage: min_p(A) == 0
True
sage: B == A
True

sage: char_p = A.characteristic_polynomial(); char_p
x^10 + 13*x^9 + 10*x^8 + 9*x^7 + 10*x^6 + 4*x^5 + 10*x^4 + 10*x^3 + 12*x^2 +␣
↪14*x + 7
sage: min_p.divides(char_p)
True
```

```
sage: A = random_matrix(GF(1214471), 10, 10); A
[ 266673  745841  418200  521668  905837  160562  831940   65852  173001 ␣
↪515930]
[ 714380  778254  844537  584888  392730  502193  959391  614352  775603 ␣
↪240043]
[1156372  104118 1175992  612032 1049083  660489 1066446  809624   15010␣
↪1002045]
[ 470722  314480 1155149 1173111   14213 1190467 1079166  786442  429883 ␣
↪563611]
[ 625490 1015074  888047 1090092  892387    4724  244901  696350  384684 ␣
↪254561]
[ 898612   44844   83752 1091581  349242  130212  580087  253296  472569 ␣
↪913613]
[ 919150   38603  710029  438461  736442  943501  792110  110470  850040 ␣
↪713428]
[ 668799 1122064  325250 1084368  520553 1179743  791517   34060 1183757␣
↪1118938]
[ 642169   47513   73428 1076788  216479  626571  105273  400489 1041378␣
↪1186801]
[ 158611  888598 1138220 1089631   56266 1092400  890773 1060810  211135 ␣
↪719636]

sage: B = copy(A)
sage: min_p = A.minimal_polynomial(proof=True); min_p
x^10 + 283013*x^9 + 252503*x^8 + 512435*x^7 + 742964*x^6 + 130817*x^5 +␣
↪581471*x^4 + 899760*x^3 + 207023*x^2 + 470831*x + 381978

sage: min_p(A) == 0
True
sage: B == A
True

sage: char_p = A.characteristic_polynomial(); char_p
x^10 + 283013*x^9 + 252503*x^8 + 512435*x^7 + 742964*x^6 + 130817*x^5 +␣
↪581471*x^4 + 899760*x^3 + 207023*x^2 + 470831*x + 381978

sage: min_p.divides(char_p)
True
```

```
sage: A = random_matrix(GF(2535919), 0, 0)
sage: A.minimal_polynomial()
1

sage: A = random_matrix(GF(2535919), 0, 1)
sage: A.minimal_polynomial()
Traceback (most recent call last):
```

```
...
ValueError: matrix must be square

sage: A = random_matrix(GF(2535919), 1, 0)
sage: A.minimal_polynomial()
Traceback (most recent call last):
...
ValueError: matrix must be square

sage: A = matrix(GF(2535919), 10, 10)
sage: A.minimal_polynomial()
x
```

EXAMPLES:

```
sage: R.<x>=GF(3)[]
sage: A = matrix(GF(3),2,[0,0,1,2])
sage: A.minpoly()
x^2 + x

sage: A.minpoly(proof=False) in [x, x+1, x^2+x]
True
```

randomize (*density=1*, *nonzero=False*)

Randomize `density` proportion of the entries of this matrix, leaving the rest unchanged.

INPUT:

- **`density` - Integer; proportion (roughly) to be considered** for changes
- **`nonzero` - Bool (default: `False`); whether the new** entries are forced to be non-zero

OUTPUT:

- None, the matrix is modified in-space

EXAMPLES:

```
sage: A = matrix(GF(5), 5, 5, 0)
sage: A.randomize(0.5); A
[0 0 0 2 0]
[0 3 0 0 2]
[4 0 0 0 0]
[4 0 0 0 0]
[0 1 0 0 0]

sage: A.randomize(); A
[3 3 2 1 2]
[4 3 3 2 2]
[0 3 3 3 3]
[3 3 2 2 4]
[2 2 2 1 4]
```

The matrix is updated instead of overwritten:

```
sage: A = random_matrix(GF(5), 100, 100, density=0.1)
sage: A.density()
961/10000

sage: A.randomize(density=0.1)
```

```
sage: A.density()
801/5000
```

rank ( )

Return the rank of this matrix.

EXAMPLES:

```
sage: A = random_matrix(GF(3), 100, 100)
sage: B = copy(A)
sage: A.rank()
99
sage: B == A
True

sage: A = random_matrix(GF(3), 100, 100, density=0.01)
sage: A.rank()
63

sage: A = matrix(GF(3), 100, 100)
sage: A.rank()
0
```

Rank is not implemented over the integers modulo a composite yet.:

```
sage: M = matrix(Integers(4), 2, [2,2,2,2])
sage: M.rank()
Traceback (most recent call last):
...
NotImplementedError: Echelon form not implemented over 'Ring of integers␣
↪modulo 4'.
```

```
sage: A = random_matrix(GF(16007), 100, 100)
sage: B = copy(A)
sage: A.rank()
100
sage: B == A
True
sage: MS = A.parent()
sage: MS(1) == ~A*A
True
```

CHAPTER

THIRTYSIX

DENSE MATRICES OVER $\mathbf{Z}/N\mathbf{Z}$ FOR $N < 2^{11}$ USING LINBOX'S MODULAR<FLOAT>

AUTHORS: - Burcin Erocal - Martin Albrecht

class sage.matrix.matrix_modn_dense_float.**Matrix_modn_dense_float**

Bases: *sage.matrix.matrix_modn_dense_float.Matrix_modn_dense_template*

Dense matrices over $\mathbf{Z}/n\mathbf{Z}$ for $n < 2^{11}$ using LinBox's `Modular<float>`

These are matrices with integer entries mod `n` represented as floating-point numbers in a 32-bit word for use with LinBox routines. This allows for `n` up to 2^{11}. The `Matrix_modn_dense_double` class is used for larger moduli.

Routines here are for the most basic access, see the $matrix_modn_dense_template.pxi$ file for higher-level routines.

class sage.matrix.matrix_modn_dense_float.**Matrix_modn_dense_template**

Bases: *sage.matrix.matrix_dense.Matrix_dense*

Create a new matrix.

INPUT:

- `parent` - a matrix space
- `entries` - a list of entries or a scalar
- `copy` - ignroed
- `coerce` - perform modular reduction first?

EXAMPLES:

```
sage: A = random_matrix(GF(3),1000,1000)
sage: type(A)
<type 'sage.matrix.matrix_modn_dense_float.Matrix_modn_dense_float'>
sage: A = random_matrix(Integers(10),1000,1000)
sage: type(A)
<type 'sage.matrix.matrix_modn_dense_float.Matrix_modn_dense_float'>
sage: A = random_matrix(Integers(2^16),1000,1000)
sage: type(A)
<type 'sage.matrix.matrix_modn_dense_double.Matrix_modn_dense_double'>
```

charpoly (*var='x'*, *algorithm='linbox'*)

Return the characteristic polynomial of `self`.

INPUT:

- `var` - a variable name

•`algorithm` - 'generic', 'linbox' or 'all' (default: linbox)

EXAMPLES:

```
sage: A = random_matrix(GF(19), 10, 10); A
[ 3  1  8 10  5 16 18  9  6  1]
[ 5 14  4  4 14 15  5 11  3  0]
[ 4  1  0  7 11  6 17  8  5  6]
[ 4  6  9  4  8  1 18 17  8 18]
[11  2  0  6 13  7  4 11 16 10]
[12  6 12  3 15 10  5 11  3  8]
[15  1 16  2 18 15 14  7  2 11]
[16 16 17  7 14 12  7  7  0  5]
[13 15  9  2 12 16  1 15 18  7]
[10  8 16 18  9 18  2 13  5 10]

sage: B = copy(A)
sage: char_p = A.characteristic_polynomial(); char_p
x^10 + 2*x^9 + 18*x^8 + 4*x^7 + 13*x^6 + 11*x^5 + 2*x^4 + 5*x^3 + 7*x^2 +
→16*x + 6
sage: char_p(A) == 0
True
sage: B == A              # A is not modified
True

sage: min_p = A.minimal_polynomial(proof=True); min_p
x^10 + 2*x^9 + 18*x^8 + 4*x^7 + 13*x^6 + 11*x^5 + 2*x^4 + 5*x^3 + 7*x^2 +
→16*x + 6
sage: min_p.divides(char_p)
True
```

```
sage: A = random_matrix(GF(2916337), 7, 7); A
[ 446196 2267054   36722 2092388 1694559  514193 1196222]
[1242955 1040744   99523 2447069   40527  930282 2685786]
[2892660 1347146 1126775 2131459  869381 1853546 2266414]
[2897342 1342067 1054026  373002   84731 1270068 2421818]
[ 569466  537440  572533  297105 1415002 2079710  355705]
[2546914 2299052 2883413 1558788 1494309 1027319 1572148]
[ 250822  522367 2516720  585897 2296292 1797050 2128203]

sage: B = copy(A)
sage: char_p = A.characteristic_polynomial(); char_p
x^7 + 1191770*x^6 + 547840*x^5 + 215639*x^4 + 2434512*x^3 + 1039968*x^2 +
→483592*x + 733817
sage: char_p(A) == 0
True
sage: B == A              # A is not modified
True

sage: min_p = A.minimal_polynomial(proof=True); min_p
x^7 + 1191770*x^6 + 547840*x^5 + 215639*x^4 + 2434512*x^3 + 1039968*x^2 +
→483592*x + 733817
sage: min_p.divides(char_p)
True

sage: A = Mat(Integers(6),3,3)(range(9))
sage: A.charpoly()
x^3
```

ALGORITHM: Uses LinBox if `self.base_ring()` is a field, otherwise use Hessenberg form algorithm.

determinant ( )

Return the determinant of this matrix.

EXAMPLES:

```
sage: A = random_matrix(GF(7), 10, 10); A
[3 1 6 6 4 4 2 2 3 5]
[4 5 6 2 2 1 2 5 0 5]
[3 2 0 5 0 1 5 4 2 3]
[6 4 5 0 2 4 2 0 6 3]
[2 2 4 2 4 5 3 4 4 4]
[2 5 2 5 4 5 1 1 1 1]
[0 6 3 4 2 2 3 5 1 1]
[4 2 6 5 6 3 4 5 5 3]
[5 2 4 3 6 2 3 6 2 1]
[3 3 5 3 4 2 2 1 6 2]

sage: A.determinant()
6
```

```
    sage: A = random_matrix(GF(7), 100, 100)
    sage: A.determinant()
    2

    sage: A.transpose().determinant()
    2

    sage: B = random_matrix(GF(7), 100, 100)
    sage: B.determinant()
    4

    sage: (A*B).determinant() == A.determinant() * B.determinant()
    True

::

    sage: A = random_matrix(GF(16007), 10, 10); A
    [ 5037  2388  4150  1400   345  5945  4240 14022 10514   700]
    [15552  8539  1927  3870  9867  3263 11637   609 15424  2443]
    [ 3761 15836 12246 15577 10178 13602 13183 15918 13942  2958]
    [ 4526 10817  6887  6678  1764  9964  6107  1705  5619  5811]
    [13537 15004  8307 11846 14779   550 14113  5477  7271  7091]
    [13338  4927 11406 13065  5437 12431  6318  5119 14198   496]
    [ 1044   179 12881   353 12975 12567  1092 10433 12304   954]
    [10072  8821 14118 13895  6543 13484 10685 14363  2612 11070]
    [15113   237  2612 14127 11589  5808   117  9656 15957 14118]
    [15233 11080  5716  9029 11402  9380 13045 13986 14544  5771]

    sage: A.determinant()
    10207

::

    sage: A = random_matrix(GF(16007), 100, 100)
    sage: A.determinant()
```

```
3576


sage: A.transpose().determinant()
3576

sage: B = random_matrix(GF(16007), 100, 100)
sage: B.determinant()
4075

sage: (A*B).determinant() == A.determinant() * B.determinant()
True
```

echelonize (*algorithm='linbox'*, ***kwds*)

Put `self` in reduced row echelon form.

INPUT:

- `self` - a mutable matrix
- `algorithm`
  - `linbox` - uses the LinBox library (`EchelonFormDomain` implementation, default)
  - `linbox_noefd` - uses the LinBox library (FFPACK directly, less memory but slower)
  - `gauss` - uses a custom slower $O(n^3)$ Gauss elimination implemented in Sage.
  - `all` - compute using both algorithms and verify that the results are the same.
- `**kwds` - these are all ignored

OUTPUT:

- `self` is put in reduced row echelon form.
- the rank of self is computed and cached
- the pivot columns of self are computed and cached.
- the fact that self is now in echelon form is recorded and cached so future calls to echelonize return immediately.

EXAMPLES:

```
sage: A = random_matrix(GF(7), 10, 20); A
[3 1 6 6 4 4 2 2 3 5 4 5 6 2 2 1 2 5 0 5]
[3 2 0 5 0 1 5 4 2 3 6 4 5 0 2 4 2 0 6 3]
[2 2 4 2 4 5 3 4 4 4 2 5 2 5 4 5 1 1 1 1]
[0 6 3 4 2 2 3 5 1 1 4 2 6 5 6 3 4 5 5 3]
[5 2 4 3 6 2 3 6 2 1 3 3 5 3 4 2 2 1 6 2]
[0 5 6 3 2 5 6 6 3 2 1 4 5 0 2 6 5 2 5 1]
[4 0 4 2 6 3 3 5 3 0 0 1 2 5 5 1 6 0 0 3]
[2 0 1 0 0 3 0 2 4 2 2 4 4 4 5 4 1 2 3 4]
[2 4 1 4 3 0 6 2 2 5 2 5 3 6 4 2 2 6 4 4]
[0 0 2 2 1 6 2 0 5 0 4 3 1 6 0 6 0 4 6 5]

sage: A.echelon_form()
[1 0 0 0 0 0 0 0 0 0 6 2 6 0 1 1 2 5 6 2]
[0 1 0 0 0 0 0 0 0 0 0 4 5 4 3 4 2 5 1 2]
[0 0 1 0 0 0 0 0 0 0 6 3 4 6 1 0 3 6 5 6]
[0 0 0 1 0 0 0 0 0 0 0 3 5 2 3 4 0 6 5 3]
[0 0 0 0 1 0 0 0 0 0 0 6 3 4 5 3 0 4 3 2]
```

```
[0 0 0 0 0 1 0 0 0 0 1 1 0 2 4 2 5 5 5 0]
[0 0 0 0 0 0 1 0 0 0 1 0 1 3 2 0 0 0 5 3]
[0 0 0 0 0 0 0 1 0 0 4 4 2 6 5 4 3 4 1 0]
[0 0 0 0 0 0 0 0 1 0 1 0 4 2 3 5 4 6 4 0]
[0 0 0 0 0 0 0 0 0 1 2 0 5 0 5 5 3 1 1 4]
```

```
sage: A = random_matrix(GF(13), 10, 10); A
[ 8  3 11 11  9  4  8  7  9  9]
[ 2  9  6  5  7 12  3  4 11  5]
[12  6 11 12  4  3  3  8  9  5]
[ 4  2 10  5 10  1  1  1  6  9]
[12  8  5  5 11  4  1  2  8 11]
[ 2  6  9 11  4  7  1  0 12  2]
[ 8  9  0  7  7  7 10  4  1  4]
[ 0  8  2  6  7  5  7 12  2  3]
[ 2 11 12  3  4  7  2  9  6  1]
[ 0 11  5  9  4  5  5  8  7 10]

sage: MS = parent(A)
sage: B = A.augment(MS(1))
sage: B.echelonize()
sage: A.rank()
10
sage: C = B.submatrix(0,10,10,10); C
[ 4  9  4  4  0  4  7 11  9 11]
[11  7  6  8  2  8  6 11  9  5]
[ 3  9  9  2  4  8  9  2  9  4]
[ 7  0 11  4  0  9  6 11  8  1]
[12 12  4 12  3 12  6  1  7 12]
[12  2 11  6  6  6  7  0 10  6]
[ 0  7  3  4  7 11 10 12  4  6]
[ 5 11  0  5  3 11  4 12  5 12]
[ 6  7  3  5  1  4 11  7  4  1]
[ 4  9  6  7 11  1  2 12  6  7]

sage: ~A == C
True
```

```
sage: A = random_matrix(Integers(10), 10, 20)
sage: A.echelon_form()
Traceback (most recent call last):
...
NotImplementedError: Echelon form not implemented over 'Ring of integers␣
↪modulo 10'.
```

:: sage: A = random_matrix(GF(16007), 10, 20); A [15455 1177 10072 4693 3887 4102 10746 15265 6684 14559 4535 13921 9757 9525 9301 8566 2460 9609 3887 6205] [8602 10035 1242 9776 162 7893 12619 6660 13250 1988 14263 11377 2216 1247 7261 8446 15081 14412 7371 7948] [12634 7602 905 9617 13557 2694 13039 4936 12208 15480 3787 11229 593 12462 5123 14167 6460 3649 5821 6736] [10554 2511 11685 12325 12287 6534 11636 5004 6468 3180 3607 11627 13436 5106 3138 13376 8641 9093 2297 5893] [1025 11376 10288 609 12330 3021 908 13012 2112 11505 56 5971 338 2317 2396 8561 5593 3782 7986 13173] [7607 588 6099 12749 10378 111 2852 10375 8996 7969 774 13498 12720 4378 6817 6707 5299 9406 13318 2863] [15545 538 4840 1885 8471 1303 11086 14168 1853 14263 3995 12104 1294 7184 1188 11901 15971 2899 4632 711] [584 11745 7540 15826 15027 5953 7097 14329 10889 12532 13309 15041 6211 1749 10481 9999 2751 11068 21 2795] [761 11453 3435 10596 2173 7752 15941 14610 1072 8012 9458 5440 612 10581

10400 101 11472 13068 7758 7898] [10658 4035 6662 655 7546 4107 6987 1877 4072 4221 7679 14579 2474 8693 8127 12999 11141 605 9404 10003] sage: A.echelon_form() [1 0 0 0 0 0 0 0 0 0 8416 8364 10318 1782 13872 4566 14855 7678 11899 2652] [0 1 0 0 0 0 0 0 0 0 4782 15571 3133 10964 5581 10435 9989 14303 5951 8048] [0 0 1 0 0 0 0 0 0 0 15688 6716 13819 4144 257 5743 14865 15680 4179 10478] [0 0 0 1 0 0 0 0 0 0 4307 9488 2992 9925 13984 15754 8185 11598 14701 10784] [0 0 0 0 1 0 0 0 0 0 927 3404 15076 1040 2827 9317 14041 10566 5117 7452] [0 0 0 0 0 1 0 0 0 0 1144 10861 5241 6288 9282 5748 3715 13482 7258 9401] [0 0 0 0 0 0 1 0 0 0 769 1804 1879 4624 6170 7500 11883 9047 874 597] [0 0 0 0 0 0 0 1 0 0 15591 13686 5729 11259 10219 13222 15177 15727 5082 11211] [0 0 0 0 0 0 0 0 1 0 8375 14939 13471 12221 8103 4212 11744 10182 2492 11068] [0 0 0 0 0 0 0 0 0 1 6534 396 6780 14734 1206 3848 7712 9770 10755 410]

```
sage: A = random_matrix(Integers(10000), 10, 20)
sage: A.echelon_form()
Traceback (most recent call last):
...
NotImplementedError: Echelon form not implemented over 'Ring of integers␣
↪modulo 10000'.
```

hessenbergize ( )

Transforms self in place to its Hessenberg form.

EXAMPLES:

```
sage: A = random_matrix(GF(17), 10, 10, density=0.1); A
[ 0  0  0  0 12  0  0  0  0  0]
[ 0  0  0  4  0  0  0  0  0  0]
[ 0  0  0  0  2  0  0  0  0  0]
[ 0 14  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0 10  0  0  0  0]
[ 0  0  0  0  0 16  0  0  0  0]
[ 0  0  0  0  0  0  6  0  0  0]
[15  0  0  0  0  0  0  0  0  0]
[ 0  0  0 16  0  0  0  0  0  0]
[ 0  5  0  0  0  0  0  0  0  0]
sage: A.hessenbergize(); A
[ 0  0  0  0  0  0  0 12  0  0]
[15  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  2  0  0]
[ 0  0  0  0 14  0  0  0  0  0]
[ 0  0  0  4  0  0  0  0  0  0]
[ 0  0  0  0  5  0  0  0  0  0]
[ 0  0  0  0  0  0  6  0  0  0]
[ 0  0  0  0  0  0  0  0  0 10]
[ 0  0  0  0  0  0  0  0  0  0]
[ 0  0  0  0  0  0  0  0  0 16]
```

lift ( )

Return the lift of this matrix to the integers.

EXAMPLES:

```
sage: A = matrix(GF(7),2,3,[1..6])
sage: A.lift()
[1 2 3]
[4 5 6]
sage: A.lift().parent()
Full MatrixSpace of 2 by 3 dense matrices over Integer Ring
```

```
sage: A = matrix(GF(16007),2,3,[1..6])
sage: A.lift()
[1 2 3]
[4 5 6]
sage: A.lift().parent()
Full MatrixSpace of 2 by 3 dense matrices over Integer Ring
```

Subdivisions are preserved when lifting:

```
sage: A.subdivide([], [1,1]); A
[1||2 3]
[4||5 6]
sage: A.lift()
[1||2 3]
[4||5 6]
```

minpoly (*var='x'*, *algorithm='linbox'*, *proof=None*)
Returns the minimal polynomial of`` self``.

INPUT:

- `var` - a variable name
- `algorithm` - `generic` or `linbox` (default: `linbox`)
- `proof` – (default: `True`); whether to provably return the true minimal polynomial; if `False` , we only guarantee to return a divisor of the minimal polynomial. There are also certainly cases where the computed results is frequently not exactly equal to the minimal polynomial (but is instead merely a divisor of it).

> **Warning:** If `proof=True` , minpoly is insanely slow compared to `proof=False` . This matters since proof=True is the default, unless you first type `proof.linear_algebra(False)` .

EXAMPLES:

```
sage: A = random_matrix(GF(17), 10, 10); A
[ 2 14  0 15 11 10 16  2  9  4]
[10 14  1 14  3 14 12 14  3 13]
[10  1 14  6  2 14 13  7  6 14]
[10  3  9 15  8  1  5  8 10 11]
[ 5 12  4  9 15  2  6 11  2 12]
[ 6 10 12  0  6  9  7  7  3  8]
[ 2  9  1  5 12 13  7 16  7 11]
[11  1  0  2  0  4  7  9  8 15]
[ 5  3 16  2 11 10 12 14  0  7]
[16  4  6  5  2  3 14 15 16  4]

sage: B = copy(A)
sage: min_p = A.minimal_polynomial(proof=True); min_p
x^10 + 13*x^9 + 10*x^8 + 9*x^7 + 10*x^6 + 4*x^5 + 10*x^4 + 10*x^3 + 12*x^2 +
↪14*x + 7
sage: min_p(A) == 0
True
sage: B == A
True
```

```
sage: char_p = A.characteristic_polynomial(); char_p
x^10 + 13*x^9 + 10*x^8 + 9*x^7 + 10*x^6 + 4*x^5 + 10*x^4 + 10*x^3 + 12*x^2 +␣
↪14*x + 7
sage: min_p.divides(char_p)
True
```

```
sage: A = random_matrix(GF(1214471), 10, 10); A
[ 266673  745841  418200  521668  905837  160562  831940   65852  173001 ␣
↪515930]
[ 714380  778254  844537  584888  392730  502193  959391  614352  775603 ␣
↪240043]
[1156372  104118 1175992  612032 1049083  660489 1066446  809624   15010␣
↪1002045]
[ 470722  314480 1155149 1173111   14213 1190467 1079166  786442  429883 ␣
↪563611]
[ 625490 1015074  888047 1090092  892387    4724  244901  696350  384684 ␣
↪254561]
[ 898612   44844   83752 1091581  349242  130212  580087  253296  472569 ␣
↪913613]
[ 919150   38603  710029  438461  736442  943501  792110  110470  850040 ␣
↪713428]
[ 668799 1122064  325250 1084368  520553 1179743  791517   34060 1183757␣
↪1118938]
[ 642169   47513   73428 1076788  216479  626571  105273  400489 1041378␣
↪1186801]
[ 158611  888598 1138220 1089631   56266 1092400  890773 1060810  211135 ␣
↪719636]

sage: B = copy(A)
sage: min_p = A.minimal_polynomial(proof=True); min_p
x^10 + 283013*x^9 + 252503*x^8 + 512435*x^7 + 742964*x^6 + 130817*x^5 +␣
↪581471*x^4 + 899760*x^3 + 207023*x^2 + 470831*x + 381978

sage: min_p(A) == 0
True
sage: B == A
True

sage: char_p = A.characteristic_polynomial(); char_p
x^10 + 283013*x^9 + 252503*x^8 + 512435*x^7 + 742964*x^6 + 130817*x^5 +␣
↪581471*x^4 + 899760*x^3 + 207023*x^2 + 470831*x + 381978

sage: min_p.divides(char_p)
True
```

```
sage: A = random_matrix(GF(2535919), 0, 0)
sage: A.minimal_polynomial()
1

sage: A = random_matrix(GF(2535919), 0, 1)
sage: A.minimal_polynomial()
Traceback (most recent call last):
...
ValueError: matrix must be square

sage: A = random_matrix(GF(2535919), 1, 0)
```

```
sage: A.minimal_polynomial()
Traceback (most recent call last):
...
ValueError: matrix must be square

sage: A = matrix(GF(2535919), 10, 10)
sage: A.minimal_polynomial()
x
```

EXAMPLES:

```
sage: R.<x>=GF(3)[]
sage: A = matrix(GF(3),2,[0,0,1,2])
sage: A.minpoly()
x^2 + x

sage: A.minpoly(proof=False) in [x, x+1, x^2+x]
True
```

randomize (*density=1*, *nonzero=False*)

Randomize `density` proportion of the entries of this matrix, leaving the rest unchanged.

INPUT:

- **`density` - Integer; proportion (roughly) to be considered** for changes
- **`nonzero` - Bool (default: `False`); whether the new** entries are forced to be non-zero

OUTPUT:

- None, the matrix is modified in-space

EXAMPLES:

```
sage: A = matrix(GF(5), 5, 5, 0)
sage: A.randomize(0.5); A
[0 0 0 2 0]
[0 3 0 0 2]
[4 0 0 0 0]
[4 0 0 0 0]
[0 1 0 0 0]

sage: A.randomize(); A
[3 3 2 1 2]
[4 3 3 2 2]
[0 3 3 3 3]
[3 3 2 2 4]
[2 2 2 1 4]
```

The matrix is updated instead of overwritten:

```
sage: A = random_matrix(GF(5), 100, 100, density=0.1)
sage: A.density()
961/10000

sage: A.randomize(density=0.1)
sage: A.density()
801/5000
```

rank ( )

Return the rank of this matrix.

EXAMPLES:

```
sage: A = random_matrix(GF(3), 100, 100)
sage: B = copy(A)
sage: A.rank()
99
sage: B == A
True

sage: A = random_matrix(GF(3), 100, 100, density=0.01)
sage: A.rank()
63

sage: A = matrix(GF(3), 100, 100)
sage: A.rank()
0
```

Rank is not implemented over the integers modulo a composite yet.:

```
sage: M = matrix(Integers(4), 2, [2,2,2,2])
sage: M.rank()
Traceback (most recent call last):
...
NotImplementedError: Echelon form not implemented over 'Ring of integers␣
↪modulo 4'.
```

```
sage: A = random_matrix(GF(16007), 100, 100)
sage: B = copy(A)
sage: A.rank()
100
sage: B == A
True
sage: MS = A.parent()
sage: MS(1) == ~A*A
True
```

CHAPTER

THIRTYSEVEN

SPARSE RATIONAL MATRICES.

AUTHORS:

- William Stein (2007-02-21)
- Soroosh Yazdani (2007-02-21)

class `sage.matrix.matrix_rational_sparse.` **`Matrix_rational_sparse`**

Bases: *`sage.matrix.matrix_sparse.Matrix_sparse`*

Create a sparse matrix over the rational numbers.

INPUT:

- `parent` – a matrix space
- `entries` – can be one of the following:
  - a Python dictionary whose items have the form `(i, j): x`, where `0 <= i < nrows`, `0 <= j < ncols`, and `x` is coercible to a rational. The `i, j` entry of `self` is set to `x`. The `x` 's can be `0`.
  - Alternatively, entries can be a list of *all* the entries of the sparse matrix, read row-by-row from top to bottom (so they would be mostly 0).
- `copy` – ignored
- `coerce` – ignored

`denominator` ( )

Return the denominator of this matrix.

OUTPUT:

– Sage Integer

EXAMPLES:

```
sage: b = matrix(QQ,2,range(6)); b[0,0]=-5007/293; b
[-5007/293         1         2]
[        3         4         5]
sage: b.denominator()
293
```

`dense_matrix` ( )

Return dense version of this matrix.

EXAMPLES:

```
sage: a = matrix(QQ,2,[1..4],sparse=True); type(a)
<type 'sage.matrix.matrix_rational_sparse.Matrix_rational_sparse'>
sage: type(a.dense_matrix())
```

```
<type 'sage.matrix.matrix_rational_dense.Matrix_rational_dense'>
sage: a.dense_matrix()
[1 2]
[3 4]
```

Check that subdivisions are preserved when converting between dense and sparse matrices:

```
sage: a.subdivide([1,1], [2])
sage: b = a.dense_matrix().sparse_matrix().dense_matrix()
sage: b.subdivisions() == a.subdivisions()
True
```

echelon_form (*algorithm='default'*, *height_guess=None*, *proof=True*, ***kwds*)

INPUT:

`height_guess`, `proof`, `**kwds` – all passed to the multimodular algorithm; ignored by the p-adic algorithm.

OUTPUT:

self is no in reduced row echelon form.

EXAMPLES:

```
sage: a = matrix(QQ, 4, range(16), sparse=True); a[0,0] = 1/19; a[0,1] = 1/5;␣
↪a
[1/19  1/5    2    3]
[   4    5    6    7]
[   8    9   10   11]
[  12   13   14   15]
sage: a.echelon_form()
[      1       0       0 -76/157]
[      0       1       0  -5/157]
[      0       0       1 238/157]
[      0       0       0       0]
```

echelonize (*height_guess=None*, *proof=True*, ***kwds*)

Transform the matrix `self` into reduced row echelon form in place.

INPUT:

`height_guess`, `proof`, `**kwds` – all passed to the multimodular algorithm; ignored by the p-adic algorithm.

OUTPUT:

Nothing. The matrix `self` is transformed into reduced row echelon form in place.

ALGORITHM: a multimodular algorithm.

EXAMPLES:

```
sage: a = matrix(QQ, 4, range(16), sparse=True); a[0,0] = 1/19; a[0,1] = 1/5;␣
↪a
[1/19  1/5    2    3]
[   4    5    6    7]
[   8    9   10   11]
[  12   13   14   15]
sage: a.echelonize(); a
[      1       0       0 -76/157]
[      0       1       0  -5/157]
```

```
[       0        0        1 238/157]
[       0        0        0        0]
```

trac ticket #10319 has been fixed:

```
sage: m = Matrix(QQ, [1], sparse=True); m.echelonize()
sage: m = Matrix(QQ, [1], sparse=True); m.echelonize(); m
[1]
```

height ( )

Return the height of this matrix, which is the least common multiple of all numerators and denominators of elements of this matrix.

OUTPUT:

– Integer

EXAMPLES:

```
sage: b = matrix(QQ,2,range(6), sparse=True); b[0,0]=-5007/293; b
[-5007/293         1         2]
[        3         4         5]
sage: b.height()
5007
```

set_row_to_multiple_of_row (*i*, *j*, *s*)

Set row i equal to s times row j.

EXAMPLES:

```
sage: a = matrix(QQ,2,3,range(6), sparse=True); a
[0 1 2]
[3 4 5]
sage: a.set_row_to_multiple_of_row(1,0,-3)
sage: a
[ 0  1  2]
[ 0 -3 -6]
```

CHAPTER

THIRTYEIGHT

MATRIX WINDOWS

class `sage.matrix.matrix_window.` **`MatrixWindow`**

Bases: `object`

`add` (*A*)

`add_prod` (*A*, *B*)

`echelon_in_place` ( )

Calculate the echelon form of this matrix, returning the list of pivot columns

`element_is_zero` (*i*, *j*)

`get_unsafe` (*i*, *j*)

`matrix` ( )

Returns the underlying matrix that this window is a view of.

`matrix_window` (*row*, *col*, *n_rows*, *n_cols*)

Returns a matrix window relative to this window of the underlying matrix.

`ncols` ( )

`new_empty_window` (*nrows*, *ncols*)

`new_matrix_window` (*matrix*, *row*, *col*, *n_rows*, *n_cols*)

This method is here only to provide a fast cdef way of constructing new matrix windows. The only implicit assumption is that self._matrix and matrix are over the same base ring (so share the zero).

`nrows` ( )

`set` (*src*)

`set_to` (*A*)

Change self, making it equal A.

`set_to_diff` (*A*, *B*)

`set_to_prod` (*A*, *B*)

`set_to_sum` (*A*, *B*)

`set_to_zero` ( )

`set_unsafe` (*i*, *j*, *x*)

`subtract` (*A*)

`subtract_prod` (*A*, *B*)

`swap_rows` (*a*, *b*)

to_matrix ( )

Returns an actual matrix object representing this view.

CHAPTER

THIRTYNINE

MISC MATRIX ALGORITHMS

Code goes here mainly when it needs access to the internal structure of several classes, and we want to avoid circular cimports.

NOTE: The whole problem of avoiding circular imports – the reason for existence of this file – is now a non-issue, since some bugs in Cython were fixed. Probably all this code should be moved into the relevant classes and this file deleted.

sage.matrix.misc.**cmp_pivots**(*x*, *y*)

Compare two sequences of pivot columns.

If x is shorter than y, return -1, i.e., x < y, "not as good". If x is longer than y, then x > y, so "better" and return +1. If the length is the same, then x is better, i.e., x > y if the entries of x are correspondingly <= those of y with one being strictly less.

INPUT:

- x, y – list of integers

EXAMPLES:

We illustrate each of the above comparisons.

```
sage: sage.matrix.misc.cmp_pivots([1,2,3], [4,5,6,7])
-1
sage: sage.matrix.misc.cmp_pivots([1,2,3,5], [4,5,6])
1
sage: sage.matrix.misc.cmp_pivots([1,2,4], [1,2,3])
-1
sage: sage.matrix.misc.cmp_pivots([1,2,3], [1,2,3])
0
sage: sage.matrix.misc.cmp_pivots([1,2,3], [1,2,4])
1
```

sage.matrix.misc.**hadamard_row_bound_mpfr**(*A*)

Given a matrix A with entries that coerce to RR, compute the row Hadamard bound on the determinant.

INPUT:

A – a matrix over RR

OUTPUT:

integer – an integer n such that the absolute value of the determinant of this matrix is at most 10^n.

EXAMPLES:

We create a very large matrix, compute the row Hadamard bound, and also compute the row Hadamard bound of the transpose, which happens to be sharp.

```
sage: a = matrix(ZZ, 2, [2^10000,3^10000,2^50,3^19292])
sage: import sage.matrix.misc
sage: sage.matrix.misc.hadamard_row_bound_mpfr(a.change_ring(RR))
13976
sage: len(str(a.det()))
12215
sage: sage.matrix.misc.hadamard_row_bound_mpfr(a.transpose().change_ring(RR))
12215
```

Note that in the above example using RDF would overflow:

```
sage: b = a.change_ring(RDF)
sage: b._hadamard_row_bound()
Traceback (most recent call last):
...
OverflowError: cannot convert float infinity to integer
```

`sage.matrix.misc.`**`matrix_integer_dense_rational_reconstruction`** (*A*, *N*)

Given a matrix over the integers and an integer modulus, do rational reconstruction on all entries of the matrix, viewed as numbers mod N. This is done efficiently by assuming there is a large common factor dividing the denominators.

INPUT:

A – matrix N – an integer

EXAMPLES:

sage: B = ((matrix(ZZ, 3,4, [1,2,3,-4,7,2,18,3,4,3,4,5])/3)%500).change_ring(ZZ) sage: sage.matrix.misc.matrix_integer_dense_rational_reconstruction(B, 500) [1/3 2/3 1 -4/3] [7/3 2/3 6 1] [4/3 1 4/3 5/3]

`sage.matrix.misc.`**`matrix_integer_sparse_rational_reconstruction`** (*A*, *N*)

Given a sparse matrix over the integers and an integer modulus, do rational reconstruction on all entries of the matrix, viewed as numbers mod N.

EXAMPLES:

sage: A = matrix(ZZ, 3, 4, [(1/3)%500, 2, 3, (-4)%500, 7, 2, 2, 3, 4, 3, 4, (5/7)%500], sparse=True) sage: sage.matrix.misc.matrix_integer_sparse_rational_reconstruction(A, 500) [1/3 2 3 -4] [7 2 2 3] [4 3 4 5/7]

`sage.matrix.misc.`**`matrix_rational_echelon_form_multimodular`** (*self*, *height_guess=None*, *proof=None*)

Returns reduced row-echelon form using a multi-modular algorithm. Does not change self.

REFERENCE: Chapter 7 of Stein's "Explicitly Computing Modular Forms".

INPUT:

- height_guess – integer or None
- proof – boolean or None (default: None, see proof.linear_algebra or sage.structure.proof). Note that the global Sage default is proof=True

ALGORITHM:

The following is a modular algorithm for computing the echelon form. Define the height of a matrix to be the max of the absolute values of the entries.

Given Matrix A with n columns (self).

0. Rescale input matrix A to have integer entries. This does not change echelon form and makes reduction modulo lots of primes significantly easier if there were denominators. Henceforth we assume A has integer entries.

1. Let c be a guess for the height of the echelon form. E.g., c=1000, e.g., if matrix is very sparse and application is to computing modular symbols.

2. Let M = n * c * H(A) + 1, where n is the number of columns of A.

3. List primes p_1, p_2, ..., such that the product of the p_i is at least M.

4. Try to compute the rational reconstruction CRT echelon form of A mod the product of the p_i. If rational reconstruction fails, compute 1 more echelon forms mod the next prime, and attempt again. Make sure to keep the result of CRT on the primes from before, so we don't have to do that computation again. Let E be this matrix.

5. Compute the denominator d of E. Attempt to prove that result is correct by checking that

   H(d*E)*ncols(A)*H(A) < (prod of reduction primes)

   where H denotes the height. If this fails, do step 4 with a few more primes.

EXAMPLES:

sage: A = matrix(QQ, 3, 7, [1..21]) sage: sage.matrix.misc.matrix_rational_echelon_form_multimodular(A) [1 0 -1 -2 -3 -4 -5] [0 1 2 3 4 5 6] [0 0 0 0 0 0 0]

sage: A = matrix(QQ, 3, 4, [0,0] + [1..9] + [-1/2^20]) sage: sage.matrix.misc.matrix_rational_echelon_form_multimodular(A) [1 0 0 -10485761/1048576] [0 1 0 27262979/4194304] [0 0 1 2] sage: A.echelon_form() [1 0 0 -10485761/1048576] [0 1 0 27262979/4194304] [0 0 1 2]

CHAPTER FORTY

CALCULATE SYMPLECTIC BASES FOR MATRICES OVER FIELDS AND THE INTEGERS.

This module finds a symplectic basis for an anti-symmetric, alternating matrix M defined over a field or the integers.

Anti-symmetric means that $M = -M^t$, where M^t denotes the transpose of M. Alternating means that the diagonal of M is identically zero.

A symplectic basis is a basis of the form $e_1, \ldots, e_j, f_1, \ldots, f_j, z_1, \ldots, z_k$ such that

- $z_i M v^t$ = 0 for all vectors v;
- $e_i M {e_j}^t = 0$ for all i, j;
- $f_i M {f_j}^t = 0$ for all i, j;
- $e_i M {f_j}^t = 0$ for all i not equal j;

and such that the non-zero terms

- $e_i M {f_i}^t$ are "as nice as possible": 1 over fields, or integers satisfying divisibility properties otherwise.

REFERENCES:

Bourbaki gives a nice proof that can be made constructive but is not efficient (see Section 5, Number 1, Theorem 1, page 79):

Bourbaki, N. Elements of Mathematics, Algebra III, Springer Verlag 2007.

Kuperburg gives a more efficient and constructive exposition (see Theorem 18).

Kuperberg, Greg. Kasteleyn Cokernels. Electr. J. Comb. 9(1), 2002.

TODO:

The routine over the integers applies over general principal ideal domains.

WARNING:

This code is not a good candidate for conversion to Cython. The majority of the execution time is spent adding multiples of columns and rows, which is already fast. It would be better to devise a better algorithm, perhaps modular or based on a fast `smith_form` implementation.

AUTHOR:

- Nick Alexander: initial implementation
- David Loeffler (2008-12-08): changed conventions for consistency with smith_form

`sage.matrix.symplectic_basis.`**`symplectic_basis_over_ZZ`**`(`*M*`)`

Find a symplectic basis for an anti-symmetric, alternating matrix M defined over the integers.

Returns a pair (F, C) such that the rows of C form a symplectic basis for M and F = C * M * C.transpose().

Anti-symmetric means that $M = -M^t$. Alternating means that the diagonal of M is identically zero.

A symplectic basis is a basis of the form $e_1, \ldots, e_j, f_1, \ldots, f_j, z_1, \ldots, z_k$ such that

- $z_i M v^t$ = 0 for all vectors v;
- $e_i M {e_j}^t = 0$ for all i, j;
- $f_i M {f_j}^t = 0$ for all i, j;
- $e_i M {f_i}^t = d_i$ for all i, where d_i are positive integers such that $d_i | d_{i+1}$ for all i;
- $e_i M {f_j}^t = 0$ for all i not equal j.

The ordering for the factors $d_i | d_{i+1}$ and for the placement of zeroes was chosen to agree with the output of `smith_form`.

See the examples for a pictorial description of such a basis.

EXAMPLES:

```
sage: from sage.matrix.symplectic_basis import symplectic_basis_over_ZZ
```

An example which does not have full rank:

```
sage: E = matrix(ZZ, 4, 4, [0, 16, 0, 2, -16, 0, 0, -4, 0, 0, 0, 0, -2, 4, 0, 0]);
↪ E
[  0  16   0   2]
[-16   0   0  -4]
[  0   0   0   0]
[ -2   4   0   0]
sage: F, C = symplectic_basis_over_ZZ(E)
sage: F
[ 0  2  0  0]
[-2  0  0  0]
[ 0  0  0  0]
[ 0  0  0  0]
sage: C * E * C.transpose() == F
True
```

A larger example:

```
sage: E = matrix(ZZ, 8, 8, [0, 25, 0, 0, -37, -3, 2, -5, -25, 0, 1, -5, -54, -3,␣
↪3, 3, 0, -1, 0, 7, 0, -4, -20, 0, 0, 5, -7, 0, 0, 14, 0, -3, 37, 54, 0, 0, 0,␣
↪2, 3, -12, 3, 3, 4, -14, -2, 0, -3, 2, -2, -3, 20, 0, -3, 3, 0, -2, 5, -3, 0,␣
↪3, 12, -2, 2, 0]); E
[  0  25   0   0 -37  -3   2  -5]
[-25   0   1  -5 -54  -3   3   3]
[  0  -1   0   7   0  -4 -20   0]
[  0   5  -7   0   0  14   0  -3]
[ 37  54   0   0   0   2   3 -12]
[  3   3   4 -14  -2   0  -3   2]
[ -2  -3  20   0  -3   3   0  -2]
[  5  -3   0   3  12  -2   2   0]
sage: F, C = symplectic_basis_over_ZZ(E)
sage: F
[     0      0      0      0      1      0      0      0]
[     0      0      0      0      0      1      0      0]
[     0      0      0      0      0      0      1      0]
[     0      0      0      0      0      0      0  20191]
[    -1      0      0      0      0      0      0      0]
[     0     -1      0      0      0      0      0      0]
```

```
[     0      0     -1      0      0      0      0      0]
[     0      0      0 -20191      0      0      0      0]
sage: F == C * E * C.transpose()
True
sage: E.smith_form()[0]
[    1     0     0     0     0     0     0     0]
[    0     1     0     0     0     0     0     0]
[    0     0     1     0     0     0     0     0]
[    0     0     0     1     0     0     0     0]
[    0     0     0     0     1     0     0     0]
[    0     0     0     0     0     1     0     0]
[    0     0     0     0     0     0 20191     0]
[    0     0     0     0     0     0     0 20191]
```

An odd dimensional example:

```
sage: E = matrix(ZZ, 5, 5, [0, 14, 0, -8, -2, -14, 0, -3, -11, 4, 0, 3, 0, 0, 0,␣
↪8, 11, 0, 0, 8, 2, -4, 0, -8, 0]); E
[  0  14   0  -8  -2]
[-14   0  -3 -11   4]
[  0   3   0   0   0]
[  8  11   0   0   8]
[  2  -4   0  -8   0]
sage: F, C = symplectic_basis_over_ZZ(E)
sage: F
[ 0  0  1  0  0]
[ 0  0  0  2  0]
[-1  0  0  0  0]
[ 0 -2  0  0  0]
[ 0  0  0  0  0]
sage: F == C * E * C.transpose()
True
sage: E.smith_form()[0]
[1 0 0 0 0]
[0 1 0 0 0]
[0 0 2 0 0]
[0 0 0 2 0]
[0 0 0 0 0]

sage: F.parent()
Full MatrixSpace of 5 by 5 dense matrices over Integer Ring
sage: C.parent()
Full MatrixSpace of 5 by 5 dense matrices over Integer Ring
```

`sage.matrix.symplectic_basis.`**`symplectic_basis_over_field`**`(`*M*`)`

Find a symplectic basis for an anti-symmetric, alternating matrix M defined over a field.

Returns a pair (F, C) such that the rows of C form a symplectic basis for M and `F = C * M * C.transpose()`.

Anti-symmetric means that $M = -M^t$. Alternating means that the diagonal of M is identically zero.

A symplectic basis is a basis of the form $e_1, \ldots, e_j, f_1, \ldots f_j, z_1, \ldots, z_k$ such that

- $z_i M v^t = 0$ for all vectors v;
- $e_i M {e_j}^t = 0$ for all i, j;
- $f_i M {f_j}^t = 0$ for all i, j;

- $e_i M {f_i}^t = 1$ for all i;
- $e_i M {f_j}^t = 0$ for all i not equal j.

See the examples for a pictorial description of such a basis.

EXAMPLES:

```
sage: from sage.matrix.symplectic_basis import symplectic_basis_over_field
```

A full rank exact example:

```
sage: E = matrix(QQ, 8, 8, [0, -1/2, -2, 1/2, 2, 0, -2, 1, 1/2, 0, -1, -3, 0, 2,␣
↪5/2, -3, 2, 1, 0, 3/2, -1, 0, -1, -2, -1/2, 3, -3/2, 0, 1, 3/2, -1/2, -1/2, -2,␣
↪0, 1, -1, 0, 0, 1, -1, 0, -2, 0, -3/2, 0, 0, 1/2, -2, 2, -5/2, 1, 1/2, -1, -1/
↪2, 0, -1, -1, 3, 2, 1/2, 1, 2, 1, 0]); E
[   0 -1/2   -2  1/2    2    0   -2    1]
[ 1/2    0   -1   -3    0    2  5/2   -3]
[   2    1    0  3/2   -1    0   -1   -2]
[-1/2    3 -3/2    0    1  3/2 -1/2 -1/2]
[  -2    0    1   -1    0    0    1   -1]
[   0   -2    0 -3/2    0    0  1/2   -2]
[   2 -5/2    1  1/2   -1 -1/2    0   -1]
[  -1    3    2  1/2    1    2    1    0]
sage: F, C = symplectic_basis_over_field(E); F
[ 0  0  0  0  1  0  0  0]
[ 0  0  0  0  0  1  0  0]
[ 0  0  0  0  0  0  1  0]
[ 0  0  0  0  0  0  0  1]
[-1  0  0  0  0  0  0  0]
[ 0 -1  0  0  0  0  0  0]
[ 0  0 -1  0  0  0  0  0]
[ 0  0  0 -1  0  0  0  0]
sage: F == C * E * C.transpose()
True
```

An example over a finite field:

```
sage: E = matrix(GF(7), 8, 8, [0, -1/2, -2, 1/2, 2, 0, -2, 1, 1/2, 0, -1, -3, 0,␣
↪2, 5/2, -3, 2, 1, 0, 3/2, -1, 0, -1, -2, -1/2, 3, -3/2, 0, 1, 3/2, -1/2, -1/2, -
↪2, 0, 1, -1, 0, 0, 1, -1, 0, -2, 0, -3/2, 0, 0, 1/2, -2, 2, -5/2, 1, 1/2, -1, -
↪1/2, 0, -1, -1, 3, 2, 1/2, 1, 2, 1, 0]); E
[0 3 5 4 2 0 5 1]
[4 0 6 4 0 2 6 4]
[2 1 0 5 6 0 6 5]
[3 3 2 0 1 5 3 3]
[5 0 1 6 0 0 1 6]
[0 5 0 2 0 0 4 5]
[2 1 1 4 6 3 0 6]
[6 3 2 4 1 2 1 0]
sage: F, C = symplectic_basis_over_field(E); F
[0 0 0 0 1 0 0 0]
[0 0 0 0 0 1 0 0]
[0 0 0 0 0 0 1 0]
[0 0 0 0 0 0 0 1]
[6 0 0 0 0 0 0 0]
[0 6 0 0 0 0 0 0]
[0 0 6 0 0 0 0 0]
[0 0 0 6 0 0 0 0]
```

```
sage: F == C * E * C.transpose()
True
```

The tricky case of characteristic 2:

```
sage: E = matrix(GF(2), 8, 8, [0, 0, 1, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1,␣
↪0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, 1, 0, 1, 0, 0,␣
↪0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0, 0]); E
[0 0 1 1 0 1 0 1]
[0 0 0 0 0 0 0 0]
[1 0 0 0 0 0 1 1]
[1 0 0 0 0 0 0 1]
[0 0 0 0 0 1 1 0]
[1 0 0 0 1 0 1 1]
[0 0 1 0 1 1 0 0]
[1 0 1 1 0 1 0 0]
sage: F, C = symplectic_basis_over_field(E); F
[0 0 0 1 0 0 0 0]
[0 0 0 0 1 0 0 0]
[0 0 0 0 0 1 0 0]
[1 0 0 0 0 0 0 0]
[0 1 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 0 0 0]
sage: F == C * E * C.transpose()
True
```

An inexact example:

```
sage: E = matrix(RR, 8, 8, [0.000000000000000, 0.420674846479344, -0.
↪839702420666807, 0.658715385244413, 1.69467394825853, -1.14718543053828, 1.
↪03076138152950, -0.227739521708484, -0.420674846479344, 0.000000000000000, 0.
↪514381455379082, 0.282194064028260, -1.38977093018412, 0.278305070958958, -0.
↪0781320488361574, -0.496003664217833, 0.839702420666807, -0.514381455379082, 0.
↪000000000000000, -0.00618222322875384, -0.318386939149028, -0.0840205427053993,␣
↪1.28202592892333, -0.512563654267693, -0.658715385244413, -0.282194064028260, 0.
↪00618222322875384, 0.000000000000000, 0.852525732369211, -0.356957405431611, -0.
↪699960114607661, 0.0260496330859998, -1.69467394825853, 1.38977093018412, 0.
↪318386939149028, -0.852525732369211, 0.000000000000000, -0.836072521423577, 0.
↪450137632758469, -0.696145287292091, 1.14718543053828, -0.278305070958958, 0.
↪0840205427053993, 0.356957405431611, 0.836072521423577, 0.000000000000000, 0.
↪214878541347751, -1.20221688928379, -1.03076138152950, 0.0781320488361574, -1.
↪28202592892333, 0.699960114607661, -0.450137632758469, -0.214878541347751, 0.
↪000000000000000, 0.785074452163036, 0.227739521708484, 0.496003664217833, 0.
↪512563654267693, -0.0260496330859998, 0.696145287292091, 1.20221688928379, -0.
↪785074452163036, 0.000000000000000]); E
[   0.000000000000000    0.420674846479344   -0.839702420666807    0.
↪658715385244413     1.69467394825853    -1.14718543053828     1.03076138152950 ␣
↪ -0.227739521708484]
[  -0.420674846479344    0.000000000000000    0.514381455379082    0.
↪282194064028260    -1.38977093018412    0.278305070958958   -0.0781320488361574 ␣
↪ -0.496003664217833]
[   0.839702420666807   -0.514381455379082    0.000000000000000 -0.
↪00618222322875384   -0.318386939149028   -0.0840205427053993     1.
↪28202592892333   -0.512563654267693]
[  -0.658715385244413   -0.282194064028260  0.00618222322875384    0.
↪000000000000000    0.852525732369211   -0.356957405431611   -0.699960114607661 ␣
↪ 0.0260496330859998]
```

```
[   -1.69467394825853     1.38977093018412    0.318386939149028   -0.
↪852525732369211    0.000000000000000   -0.836072521423577    0.450137632758469 ␣
↪ -0.696145287292091]
[    1.14718543053828   -0.278305070958958   0.0840205427053993    0.
↪356957405431611    0.836072521423577    0.000000000000000    0.214878541347751 ␣
↪  -1.20221688928379]
[   -1.03076138152950   0.0781320488361574    -1.28202592892333    0.
↪699960114607661   -0.450137632758469   -0.214878541347751    0.000000000000000 ␣
↪  0.785074452163036]
[   0.227739521708484    0.496003664217833    0.512563654267693  -0.
↪0260496330859998    0.696145287292091     1.20221688928379   -0.785074452163036␣
↪   0.000000000000000]
sage: F, C = symplectic_basis_over_field(E); F # random
[    0.000000000000000     0.000000000000000  2.22044604925031e-16 -2.
↪22044604925031e-16      1.00000000000000     0.000000000000000     0.
↪000000000000000 -3.33066907387547e-16]
[    0.000000000000000  8.14814392305203e-17 -1.66533453693773e-16 -1.
↪11022302462516e-16     0.000000000000000      1.00000000000000 -1.
↪11022302462516e-16     0.000000000000000]
[-5.27829526256056e-16 -2.40004077757759e-16  1.28373418199470e-16 -1.
↪11022302462516e-16     0.000000000000000 -3.15483812822081e-16      1.
↪00000000000000 -4.44089209850063e-16]
[ 1.31957381564014e-16  1.41622049084608e-16 -6.68515202578511e-17 -3.
↪95597468756028e-17 -4.85722573273506e-17 -5.32388011580111e-17 -1.
↪31328455615552e-16      1.00000000000000]
[    -1.00000000000000     0.000000000000000     0.000000000000000  4.
↪85722573273506e-17     0.000000000000000 -5.55111512312578e-17 -1.
↪11022302462516e-16  2.22044604925031e-16]
[    0.000000000000000     -1.00000000000000     0.000000000000000 -2.
↪77555756156289e-17  5.55111512312578e-17 -8.69223574327834e-17     0.
↪000000000000000 -4.44089209850063e-16]
[    0.000000000000000 -1.05042437087238e-17     -1.00000000000000  3.
↪33066907387547e-16  1.11022302462516e-16 -1.18333563634309e-16  4.
↪40064433050777e-17  2.22044604925031e-16]
[ 5.27829526256056e-16  1.99901485752317e-16  1.65710718121313e-17     -1.
↪00000000000000 -2.22044604925031e-16  5.52150940090699e-16 -3.93560383111738e-
↪16  1.01155762925061e-16]
sage: F == C * E * C.transpose()
True
sage: abs(F[0, 4] - 1) < 1e-10
True
sage: abs(F[4, 0] + 1) < 1e-10
True

sage: F.parent()
Full MatrixSpace of 8 by 8 dense matrices over Real Field with 53 bits of␣
↪precision
sage: C.parent()
Full MatrixSpace of 8 by 8 dense matrices over Real Field with 53 bits of␣
↪precision
```

CHAPTER

FORTYONE

J-IDEALS OF MATRICES

Let B be an $n \times n$-matrix over a principal ideal domain D.

For an ideal J, the J-ideal of B is defined to be $N_J(B) = \{f \in D[X] \mid f(B) \in M_n(J)\}$.

For a prime element p of D and $t \geq 0$, a (p^t)-minimal polynomial of B is a monic polynomial $f \in N_{(p^t)}(B)$ of minimal degree.

This module computes these minimal polynomials.

Let p be a prime element of D. Then there is a finite set $\mathcal{S}_p$ of positive integers and monic polynomials ν_{ps} for $s \in \mathcal{S}_p$ such that for $t \geq 1$,

$$N_{(p^t)}(B) = \mu_B D[X] + p^t D[X] + \sum_{\substack{s \in \mathcal{S}_p \\ s \leq b(t)}} p^{\max\{0,t-s\}} \nu_{ps} D[X]$$

holds where $b(t) = \min\{r \in \mathcal{S}_p \mid r \geq s\}$. The degree of ν_{ps} is strictly increasing in $s \in \mathcal{S}_p$ and ν_{ps} is a (p^s)-minimal polynomial. If $t \leq \max \mathcal{S}_p$, then the summand $\mu_B D[X]$ can be omitted.

All computations are done by the class *ComputeMinimalPolynomials* where various intermediate results are cached. It provides the following methods:

- *p_minimal_polynomials()* computes $\mathcal{S}_p$ and the monic polynomials ν_{ps}.
- *null_ideal()* determines $N_{(p^t)}(B)$.
- *prime_candidates()* determines all primes p where $\mathcal{S}_p$ might be non-empty.
- *integer_valued_polynomials_generators()* determines the generators of the ring $\{f \in K[X] \mid f(B) \in M_n(D)\}$ of integer valued polynomials on B.

EXAMPLES:

```
sage: from sage.matrix.compute_J_ideal import ComputeMinimalPolynomials
sage: B = matrix(ZZ, [[1, 0, 1], [1, -2, -1], [10, 0, 0]])
sage: C = ComputeMinimalPolynomials(B)
sage: C.prime_candidates()
[2, 3, 5]
sage: for t in range(4):
....:     print(C.null_ideal(2^t))
Principal ideal (1) of
    Univariate Polynomial Ring in x over Integer Ring
Ideal (2, x^2 + x) of
    Univariate Polynomial Ring in x over Integer Ring
Ideal (4, x^2 + 3*x + 2) of
    Univariate Polynomial Ring in x over Integer Ring
Ideal (8, x^3 + x^2 - 12*x - 20, 2*x^2 + 6*x + 4) of
    Univariate Polynomial Ring in x over Integer Ring
```

```
sage: C.p_minimal_polynomials(2)
{2: x^2 + 3*x + 2}
sage: C.integer_valued_polynomials_generators()
(x^3 + x^2 - 12*x - 20, [1, 1/4*x^2 + 3/4*x + 1/2])
```

The last output means that

$$\{f \in \mathbf{Q}[X] \mid f(B) \in M_3(\mathbf{Z})\} = (x^3 + x^2 - 12x - 20)\mathbf{Q}[X] + \mathbf{Z}[X] + \frac{1}{4}(x^2 + 3x + 2)\mathbf{Z}[X].$$

Todo

Test code over PIDs other than ZZ.

This requires implementation of *frobenius()* over more general domains than ZZ.

Additionally, *lifting()* requires modification or a bug needs fixing, see AskSage Question 35555.

REFERENCES:

[Ris2016], [HR2016]

AUTHORS:

- Clemens Heuberger (2016)
- Roswitha Rissner (2016)

ACKNOWLEDGEMENT:

- Clemens Heuberger is supported by the Austrian Science Fund (FWF): P 24644-N26.
- Roswitha Rissner is supported by the Austrian Science Fund (FWF): P 27816-N26.

41.1 Classes and Methods

class `sage.matrix.compute_J_ideal.`**`ComputeMinimalPolynomials`**`(`*B*`)`

Bases: `sage.structure.sage_object.SageObject`

Create an object for computing (p^t)-minimal polynomials and J-ideals.

For an ideal J and a square matrix B over a principal ideal domain D, the J-ideal of B is defined to be $N_J(B) = \{f \in D[X] \mid f(B) \in M_n(J)\}$.

For a prime element p of D and $t \geq 0$, a (p^t)-minimal polynomial of B is a monic polynomial $f \in N_{(p^t)}(B)$ of minimal degree.

The characteristic polynomial of B is denoted by χ_B; n is the size of B.

INPUT:

- `B` – a square matrix over a principal ideal domain D

OUTPUT:

An object which allows to call *p_minimal_polynomials()*, *null_ideal()* and *integer_valued_polynomials_generators()*.

EXAMPLES:

```
sage: from sage.matrix.compute_J_ideal import ComputeMinimalPolynomials
sage: B = matrix(ZZ, [[1, 0, 1], [1, -2, -1], [10, 0, 0]])
sage: C = ComputeMinimalPolynomials(B)
sage: C.prime_candidates()
[2, 3, 5]
sage: for t in range(4):
....:     print(C.null_ideal(2^t))
Principal ideal (1) of
    Univariate Polynomial Ring in x over Integer Ring
Ideal (2, x^2 + x) of
    Univariate Polynomial Ring in x over Integer Ring
Ideal (4, x^2 + 3*x + 2) of
    Univariate Polynomial Ring in x over Integer Ring
Ideal (8, x^3 + x^2 - 12*x - 20, 2*x^2 + 6*x + 4) of
    Univariate Polynomial Ring in x over Integer Ring
sage: C.p_minimal_polynomials(2)
{2: x^2 + 3*x + 2}
sage: C.integer_valued_polynomials_generators()
(x^3 + x^2 - 12*x - 20, [1, 1/4*x^2 + 3/4*x + 1/2])
```

current_nu (*p*, *t*, *pt_generators*, *prev_nu*)

Compute (p^t)-minimal polynomial of B.

INPUT:

- `p` – a prime element of D
- `t` – a positive integer
- `pt_generators` – a list $(g_1,\ldots,g_s)$ of polynomials in $D[X]$ such that $N_{(p^t)}(B) = (g_1,\ldots,g_s) + pN_{(p^{t-1})}(B)$
- `prev_nu` – a (p^{t-1})-minimal polynomial of B

OUTPUT:

A (p^t)-minimal polynomial of B.

EXAMPLES:

```
sage: from sage.matrix.compute_J_ideal import ComputeMinimalPolynomials
sage: B = matrix(ZZ, [[1, 0, 1], [1, -2, -1], [10, 0, 0]])
sage: C = ComputeMinimalPolynomials(B)
sage: x = polygen(ZZ, 'x')
sage: nu_1 = x^2 + x
sage: generators_4 = [2*x^2 + 2*x, x^2 + 3*x + 2]
sage: C.current_nu(2, 2, generators_4, nu_1)
x^2 + 3*x + 2
```

ALGORITHM:

[HR2016], Algorithm 4.

find_monic_replacements (*p*, *t*, *pt_generators*, *prev_nu*)

Replace possibly non-monic generators of $N_{(p^t)}(B)$ by monic generators.

INPUT:

- `p` – a prime element of D
- `t` – a non-negative integer

- `pt_generators` – a list $(g_1, \ldots, g_s)$ of polynomials in $D[X]$ such that $N_{(p^t)}(B) = (g_1, \ldots, g_s) + pN_{(p^{t-1})}(B)$
- `prev_nu` – a (p^{t-1})-minimal polynomial of B

OUTPUT:

A list $(h_1, \ldots, h_r)$ of monic polynomials such that $N_{(p^t)}(B) = (h_1, \ldots, h_r) + pN_{(p^{t-1})}(B)$.

EXAMPLES:

```
sage: from sage.matrix.compute_J_ideal import ComputeMinimalPolynomials
sage: B = matrix(ZZ, [[1, 0, 1], [1, -2, -1], [10, 0, 0]])
sage: C = ComputeMinimalPolynomials(B)
sage: x = polygen(ZZ, 'x')
sage: nu_1 = x^2 + x
sage: generators_4 = [2*x^2 + 2*x, x^2 + 3*x + 2]
sage: C.find_monic_replacements(2, 2, generators_4, nu_1)
[x^2 + 3*x + 2]
```

ALGORITHM:

[HR2016], Algorithms 2 and 3.

integer_valued_polynomials_generators ( )

Determine the generators of the ring of integer valued polynomials on B.

OUTPUT:

A pair `(mu_B,P)` where `P` is a list of polynomials in $K[X]$ such that

$$\{f \in K[X] \mid f(B) \in M_n(D)\} = \mu_B K[X] + \sum_{g \in P} gD[X]$$

where K denotes the fraction field of D.

EXAMPLES:

```
sage: from sage.matrix.compute_J_ideal import ComputeMinimalPolynomials
sage: B = matrix(ZZ, [[1, 0, 1], [1, -2, -1], [10, 0, 0]])
sage: C = ComputeMinimalPolynomials(B)
sage: C.integer_valued_polynomials_generators()
(x^3 + x^2 - 12*x - 20, [1, 1/4*x^2 + 3/4*x + 1/2])
```

mccoy_column (*p*, *t*, *nu*)

Compute matrix for McCoy's criterion.

INPUT:

- `p` – a prime element in D
- `t` – a positive integer
- `nu` – a (p^t)-minimal polynomial of B

OUTPUT:

An $(n^2 + 1) \times 1$ matrix g with first entry `nu` such that $\left(b \quad -\chi_B I\right) g \equiv 0 \pmod{p^t}$ where b consists of the entries of $\mathrm{adj}(X - B)$.

EXAMPLES:

```
sage: from sage.matrix.compute_J_ideal import ComputeMinimalPolynomials
sage: B = matrix(ZZ, [[1, 0, 1], [1, -2, -1], [10, 0, 0]])
sage: C = ComputeMinimalPolynomials(B)
sage: x = polygen(ZZ, 'x')
sage: nu_4 = x^2 + 3*x + 2
sage: g = C.mccoy_column(2, 2, nu_4)
sage: b = matrix(9, 1, (x-B).adjoint().list())
sage: M = matrix.block([[b , -B.charpoly(x)*matrix.identity(9)]])
sage: (M*g % 4).is_zero()
True
```

ALGORITHM:

[HR2016], Algorithm 5.

null_ideal (*b=0*)

Return the (b)-ideal $N_{(b)}(B) = \{f \in D[X] \mid f(B) \in M_n(bD)\}$.

INPUT:

- `b` – an element of D (default: 0)

OUTPUT:

An ideal in $D[X]$.

EXAMPLES:

```
sage: from sage.matrix.compute_J_ideal import ComputeMinimalPolynomials
sage: B = matrix(ZZ, [[1, 0, 1], [1, -2, -1], [10, 0, 0]])
sage: C = ComputeMinimalPolynomials(B)
sage: C.null_ideal()
Principal ideal (x^3 + x^2 - 12*x - 20) of
    Univariate Polynomial Ring in x over Integer Ring
sage: C.null_ideal(2)
Ideal (2, x^2 + x) of
    Univariate Polynomial Ring in x over Integer Ring
sage: C.null_ideal(4)
Ideal (4, x^2 + 3*x + 2) of
    Univariate Polynomial Ring in x over Integer Ring
sage: C.null_ideal(8)
Ideal (8, x^3 + x^2 - 12*x - 20, 2*x^2 + 6*x + 4) of
    Univariate Polynomial Ring in x over Integer Ring
sage: C.null_ideal(3)
Ideal (3, x^3 + x^2 - 12*x - 20) of
    Univariate Polynomial Ring in x over Integer Ring
sage: C.null_ideal(6)
Ideal (6, 2*x^3 + 2*x^2 - 24*x - 40, 3*x^2 + 3*x) of
    Univariate Polynomial Ring in x over Integer Ring
```

p_minimal_polynomials (*p*, *s_max=None*)

Compute (p^s)-minimal polynomials ν_s of B.

Compute a finite subset $\mathcal{S}$ of the positive integers and (p^s)-minimal polynomials ν_s for $s \in \mathcal{S}$.

For $0 < t \leq \max \mathcal{S}$, a (p^t)-minimal polynomial is given by ν_s where $s = \min\{r \in \mathcal{S} \mid r \geq t\}$. For $t > \max \mathcal{S}$, the minimal polynomial of B is also a (p^t)-minimal polynomial.

INPUT:

- `p` – a prime in D

- `s_max` – a positive integer (default: `None`); if set, only (p^s)-minimal polynomials for `s <= s_max` are computed (see below for details)

OUTPUT:

A dictionary. Keys are the finite set $\mathcal{S}$, the values are the associated (p^s)-minimal polynomials ν_s, $s \in \mathcal{S}$.

Setting `s_max` only affects the output if `s_max` is at most $\max \mathcal{S}$ where $\mathcal{S}$ denotes the full set. In that case, only those ν_s with `s <= s_max` are returned where `s_max` is always included even if it is not included in the full set $\mathcal{S}$.

EXAMPLES:

```
sage: from sage.matrix.compute_J_ideal import ComputeMinimalPolynomials
sage: B = matrix(ZZ, [[1, 0, 1], [1, -2, -1], [10, 0, 0]])
sage: C = ComputeMinimalPolynomials(B)
sage: C.p_minimal_polynomials(2)
{2: x^2 + 3*x + 2}
sage: set_verbose(1)
sage: C = ComputeMinimalPolynomials(B)
sage: C.p_minimal_polynomials(2)
verbose 1 (...: compute_J_ideal.py, p_minimal_polynomials)
------------------------------------------
verbose 1 (...: compute_J_ideal.py, p_minimal_polynomials)
p = 2, t = 1:
verbose 1 (...: compute_J_ideal.py, p_minimal_polynomials)
Result of lifting:
verbose 1 (...: compute_J_ideal.py, p_minimal_polynomials)
F =
verbose 1 (...: compute_J_ideal.py, p_minimal_polynomials)
[x^2 + x]
[      x]
[      0]
[      1]
[      1]
[  x + 1]
[      1]
[      0]
[      0]
[  x + 1]
verbose 1 (...: compute_J_ideal.py, current_nu)
------------------------------------------
verbose 1 (...: compute_J_ideal.py, current_nu)
(x^2 + x)
verbose 1 (...: compute_J_ideal.py, current_nu)
Generators with (p^t)-generating property:
verbose 1 (...: compute_J_ideal.py, current_nu)
[x^2 + x]
verbose 1 (...: compute_J_ideal.py, p_minimal_polynomials)
nu = x^2 + x
verbose 1 (...: compute_J_ideal.py, p_minimal_polynomials)
corresponding columns for G
verbose 1 (...: compute_J_ideal.py, p_minimal_polynomials)
[x^2 + x]
[  x + 2]
[      0]
[      1]
[      1]
[  x - 1]
[     -1]
```

```
[     10]
[      0]
[  x + 1]
verbose 1 (...: compute_J_ideal.py, p_minimal_polynomials)
------------------------------------------
verbose 1 (...: compute_J_ideal.py, p_minimal_polynomials)
p = 2, t = 2:
verbose 1 (...: compute_J_ideal.py, p_minimal_polynomials)
Result of lifting:
verbose 1 (...: compute_J_ideal.py, p_minimal_polynomials)
F =
verbose 1 (...: compute_J_ideal.py, p_minimal_polynomials)
[  2*x^2 + 2*x x^2 + 3*x + 2]
[         2*x         x + 4]
[           0             0]
[           2             1]
[           2             1]
[     2*x + 2         x + 1]
[           2            -1]
[           0            10]
[           0             0]
[     2*x + 2         x + 3]
verbose 1 (...: compute_J_ideal.py, current_nu)
------------------------------------------
verbose 1 (...: compute_J_ideal.py, current_nu)
(2*x^2 + 2*x, x^2 + 3*x + 2)
verbose 1 (...: compute_J_ideal.py, current_nu)
Generators with (p^t)-generating property:
verbose 1 (...: compute_J_ideal.py, current_nu)
[x^2 + 3*x + 2]
verbose 1 (...: compute_J_ideal.py, p_minimal_polynomials)
nu = x^2 + 3*x + 2
verbose 1 (...: compute_J_ideal.py, p_minimal_polynomials)
corresponding columns for G
verbose 1 (...: compute_J_ideal.py, p_minimal_polynomials)
[x^2 + 3*x + 2]
[        x + 4]
[            0]
[            1]
[            1]
[        x + 1]
[           -1]
[           10]
[            0]
[        x + 3]
verbose 1 (...: compute_J_ideal.py, p_minimal_polynomials)
------------------------------------------
verbose 1 (...: compute_J_ideal.py, p_minimal_polynomials)
p = 2, t = 3:
verbose 1 (...: compute_J_ideal.py, p_minimal_polynomials)
Result of lifting:
verbose 1 (...: compute_J_ideal.py, p_minimal_polynomials)
F =
verbose 1 (...: compute_J_ideal.py, p_minimal_polynomials)
[x^3 + 7*x^2 + 6*x x^3 + 3*x^2 + 2*x]
[        x^2 + 8*x         x^2 + 4*x]
[                0                 0]
[                x             x + 4]
```

```
[            x + 4                 x]
[    x^2 + 5*x + 4           x^2 + x]
[           -x + 4                -x]
[             10*x              10*x]
[                0                 0]
[        x^2 + 7*x     x^2 + 3*x + 4]
verbose 1 (...: compute_J_ideal.py, current_nu)
------------------------------------------
verbose 1 (...: compute_J_ideal.py, current_nu)
(x^3 + 7*x^2 + 6*x, x^3 + 3*x^2 + 2*x)
verbose 1 (...: compute_J_ideal.py, current_nu)
Generators with (p^t)-generating property:
verbose 1 (...: compute_J_ideal.py, current_nu)
[x^3 + 7*x^2 + 6*x, x^3 + 3*x^2 + 2*x]
verbose 1 (...: compute_J_ideal.py, current_nu)
[x^3 + 3*x^2 + 2*x]
verbose 1 (...: compute_J_ideal.py, p_minimal_polynomials)
nu = x^3 + 3*x^2 + 2*x
{2: x^2 + 3*x + 2}
sage: set_verbose(0)
sage: C.p_minimal_polynomials(2, s_max=1)
{1: x^2 + x}
sage: C.p_minimal_polynomials(2, s_max=2)
{2: x^2 + 3*x + 2}
sage: C.p_minimal_polynomials(2, s_max=3)
{2: x^2 + 3*x + 2}
```

ALGORITHM:

[HR2016], Algorithm 5.

prime_candidates ( )

Determine those primes p where μ_B might not be a (p)-minimal polynomial.

OUTPUT:

A list of primes.

EXAMPLES:

```
sage: from sage.matrix.compute_J_ideal import ComputeMinimalPolynomials
sage: B = matrix(ZZ, [[1, 0, 1], [1, -2, -1], [10, 0, 0]])
sage: C = ComputeMinimalPolynomials(B)
sage: C.prime_candidates()
[2, 3, 5]
sage: C.p_minimal_polynomials(2)
{2: x^2 + 3*x + 2}
sage: C.p_minimal_polynomials(3)
{}
sage: C.p_minimal_polynomials(5)
{}
```

This means that 3 and 5 were candidates, but actually, μ_B turns out to be a (3)-minimal polynomial and a (5)-minimal polynomial.

sage.matrix.compute_J_ideal.**lifting** (p, t, A, G)

Compute generators of $\{f \in D[X]^d \mid Af \equiv 0 \pmod{p^t}\}$ given generators of $\{f \in D[X]^d \mid Af \equiv 0 \pmod{p^{t-1}}\}$.

INPUT:

- `p` – a prime element of some principal ideal domain D
- `t` – a non-negative integer
- `A` – a $c \times d$ matrix over $D[X]$
- `G` – a matrix over $D[X]$. The columns of $\begin{pmatrix} p^{t-1}I & G \end{pmatrix}$ are generators of $\{f \in D[X]^d \mid Af \equiv 0 \pmod{p^{t-1}}\}$; can be set to `None` if `t` is zero

OUTPUT:

A matrix F over $D[X]$ such that the columns of $\begin{pmatrix} p^t I & F & pG \end{pmatrix}$ are generators of $\{f \in D[X]^d \mid Af \equiv 0 \pmod{p^t}\}$.

EXAMPLES:

```
sage: from sage.matrix.compute_J_ideal import lifting
sage: X = polygen(ZZ, 'X')
sage: A = matrix([[1, X], [2*X, X^2]])
sage: G0 = lifting(5, 0, A, None)
sage: G1 = lifting(5, 1, A, G0); G1
[]
sage: (A*G1 % 5).is_zero()
True
sage: A = matrix([[1, X, X^2], [2*X, X^2, 3*X^3]])
sage: G0 = lifting(5, 0, A, None)
sage: G1 = lifting(5, 1, A, G0); G1
[3*X^2]
[    X]
[    1]
sage: (A*G1 % 5).is_zero()
True
sage: G2 = lifting(5, 2, A, G1); G2
[15*X^2 23*X^2]
[   5*X      X]
[     5      1]
sage: (A*G2 % 25).is_zero()
True
sage: lifting(5, 10, A, G1)
Traceback (most recent call last):
...
ValueError: A*G not zero mod 5^9
```

ALGORITHM:

[HR2016], Algorithm 1.

`sage.matrix.compute_J_ideal.`**`p_part`**`(f, p)`

Compute the p-part of a polynomial.

INPUT:

- `f` – a polynomial over D
- `p` – a prime in D

OUTPUT:

A polynomial g such that $\deg g \leq \deg f$ and all non-zero coefficients of $f - pg$ are not divisible by p.

EXAMPLES:

```
sage: from sage.matrix.compute_J_ideal import p_part
sage: X = polygen(ZZ, 'X')
sage: f = X^3 + 5*X + 25
sage: g = p_part(f, 5); g
X + 5
sage: f - 5*g
X^3
```

CHAPTER

FORTYTWO

BENCHMARKS FOR MATRICES

This file has many functions for computing timing benchmarks of various methods for random matrices with given bounds for the entries. The systems supported are Sage and Magma.

The basic command syntax is as follows:

```
sage: import sage.matrix.benchmark as b
sage: print("starting"); import sys; sys.stdout.flush(); b.report([b.det_ZZ], 'Test',␣
↪systems=['sage'])
starting...
======================================================================
          Test
======================================================================
...
======================================================================
```

sage.matrix.benchmark.**MatrixVector_QQ**(*n=1000*, *h=100*, *system='sage'*, *times=1*)

Compute product of square `n` matrix by random vector with num and denom bounded by `h` the given number of `times`.

INPUT:

- `n` - matrix dimension (default: `300`)
- `h` - numerator and denominator bound (default: `bnd`)
- `system` - either 'sage' or 'magma' (default: 'sage')
- `times` - number of experiments (default: `1`)

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: ts = b.MatrixVector_QQ(500)
sage: tm = b.MatrixVector_QQ(500, system='magma')  # optional - magma
```

sage.matrix.benchmark.**charpoly_GF**(*n=100*, *p=16411*, *system='sage'*)

Given a n x n matrix over GF with random entries, compute the charpoly.

INPUT:

- `n` - matrix dimension (default: 100)
- `p` - prime number (default: `16411`)
- `system` - either 'magma' or 'sage' (default: 'sage')

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: ts = b.charpoly_GF(100)
sage: tm = b.charpoly_GF(100, system='magma')  # optional - magma
```

sage.matrix.benchmark.**charpoly_ZZ**(*n=100*, *min=0*, *max=9*, *system='sage'*)

Characteristic polynomial over ZZ: Given a n x n matrix over ZZ with random entries between min and max, compute the charpoly.

INPUT:

- `n` - matrix dimension (default: `100`)
- `min` - minimal value for entries of matrix (default: `0`)
- `max` - maximal value for entries of matrix (default: `9`)
- `system` - either 'sage' or 'magma' (default: 'sage')

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: ts = b.charpoly_ZZ(100)
sage: tm = b.charpoly_ZZ(100, system='magma')  # optional - magma
```

sage.matrix.benchmark.**det_GF**(*n=400*, *p=16411*, *system='sage'*)

Dense determinant over GF(p). Given an n x n matrix A over GF with random entries compute det(A).

INPUT:

- `n` - matrix dimension (default: 300)
- `p` - prime number (default: `16411`)
- `system` - either 'magma' or 'sage' (default: 'sage')

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: ts = b.det_GF(1000)
sage: tm = b.det_GF(1000, system='magma')  # optional - magma
```

sage.matrix.benchmark.**det_QQ**(*n=300*, *num_bound=10*, *den_bound=10*, *system='sage'*)

Dense rational determinant over QQ. Given an n x n matrix A over QQ with random entries with numerator bound and denominator bound, compute det(A).

INPUT:

- `n` - matrix dimension (default: `200`)
- `num_bound` - numerator bound, inclusive (default: `10`)
- `den_bound` - denominator bound, inclusive (default: `10`)
- `system` - either 'sage' or 'magma' (default: 'sage')

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: ts = b.det_QQ(200)
sage: ts = b.det_QQ(10, num_bound=100000, den_bound=10000)
sage: tm = b.det_QQ(200, system='magma')  # optional - magma
```

sage.matrix.benchmark. **det_ZZ** (*n=200*, *min=1*, *max=100*, *system='sage'*)

Dense integer determinant over ZZ. Given an n x n matrix A over ZZ with random entries between min and max, inclusive, compute det(A).

INPUT:

- `n` - matrix dimension (default: `200`)
- `min` - minimal value for entries of matrix (default: `1`)
- `max` - maximal value for entries of matrix (default: `100`)
- `system` - either 'sage' or 'magma' (default: 'sage')

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: ts = b.det_ZZ(200)
sage: tm = b.det_ZZ(200, system='magma')  # optional - magma
```

sage.matrix.benchmark. **det_hilbert_QQ** (*n=80*, *system='sage'*)

Runs the benchmark for calculating the determinant of the hilbert matrix over rationals of dimension n.

INPUT:

- `n` - matrix dimension (default: `300`)
- `system` - either 'sage' or 'magma' (default: 'sage')

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: ts = b.det_hilbert_QQ(50)
sage: tm = b.det_hilbert_QQ(50, system='magma')  # optional - magma
```

sage.matrix.benchmark. **echelon_QQ** (*n=100*, *min=0*, *max=9*, *system='sage'*)

Given a n x (2*n) matrix over QQ with random integer entries between min and max, compute the reduced row echelon form.

INPUT:

- `n` - matrix dimension (default: `300`)
- `min` - minimal value for entries of matrix (default: `-9`)
- `max` - maximal value for entries of matrix (default: `9`)
- `system` - either 'sage' or 'magma' (default: 'sage')

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: ts = b.echelon_QQ(100)
sage: tm = b.echelon_QQ(100, system='magma')  # optional - magma
```

sage.matrix.benchmark. **hilbert_matrix** (*n*)

Returns the Hilbert matrix of size n over rationals.

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: b.hilbert_matrix(3)
[  1 1/2 1/3]
```

```
[1/2 1/3 1/4]
[1/3 1/4 1/5]
```

sage.matrix.benchmark. **inverse_QQ** (*n=100*, *min=0*, *max=9*, *system='sage'*)

Given a n x n matrix over QQ with random integer entries between min and max, compute the reduced row echelon form.

INPUT:

- `n` - matrix dimension (default: `300`)
- `min` - minimal value for entries of matrix (default: `-9`)
- `max` - maximal value for entries of matrix (default: `9`)
- `system` - either 'sage' or 'magma' (default: 'sage')

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: ts = b.inverse_QQ(100)
sage: tm = b.inverse_QQ(100, system='magma')   # optional - magma
```

sage.matrix.benchmark. **invert_hilbert_QQ** (*n=40*, *system='sage'*)

Runs the benchmark for calculating the inverse of the hilbert matrix over rationals of dimension n.

INPUT:

- `n` - matrix dimension (default: `300`)
- `system` - either 'sage' or 'magma' (default: 'sage')

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: ts = b.invert_hilbert_QQ(30)
sage: tm = b.invert_hilbert_QQ(30, system='magma')   # optional - magma
```

sage.matrix.benchmark. **matrix_add_GF** (*n=1000*, *p=16411*, *system='sage'*, *times=100*)

Given two n x n matrix over GF(p) with random entries, add them.

INPUT:

- `n` - matrix dimension (default: 300)
- `p` - prime number (default: `16411`)
- `system` - either 'magma' or 'sage' (default: 'sage')
- `times` - number of experiments (default: `100`)

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: ts = b.matrix_add_GF(500, p=19)
sage: tm = b.matrix_add_GF(500, p=19, system='magma')   # optional - magma
```

sage.matrix.benchmark. **matrix_add_ZZ** (*n=200*, *min=-9*, *max=9*, *system='sage'*, *times=50*)

Matrix addition over ZZ Given an n x n matrix A and B over ZZ with random entries between `min` and `max` , inclusive, compute A + B `times` times.

INPUT:

- `n` - matrix dimension (default: `200`)

- `min` - minimal value for entries of matrix (default: `-9`)
- `max` - maximal value for entries of matrix (default: `9`)
- `system` - either 'sage' or 'magma' (default: 'sage')
- `times` - number of experiments (default: `50`)

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: ts = b.matrix_add_ZZ(200)
sage: tm = b.matrix_add_ZZ(200, system='magma')  # optional - magma
```

`sage.matrix.benchmark.`**`matrix_add_ZZ_2`** (*n=200*, *bits=16*, *system='sage'*, *times=50*)

Matrix addition over ZZ. Given an n x n matrix A and B over ZZ with random `bits` -bit entries, compute A + B.

INPUT:

- `n` - matrix dimension (default: `200`)
- `bits` - bitsize of entries
- `system` - either 'sage' or 'magma' (default: 'sage')
- `times` - number of experiments (default: `50`)

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: ts = b.matrix_add_ZZ_2(200)
sage: tm = b.matrix_add_ZZ_2(200, system='magma')  # optional - magma
```

`sage.matrix.benchmark.`**`matrix_multiply_GF`** (*n=100*, *p=16411*, *system='sage'*, *times=3*)

Given an n x n matrix A over GF(p) with random entries, compute A * (A+1).

INPUT:

- `n` - matrix dimension (default: 100)
- `p` - prime number (default: `16411`)
- `system` - either 'magma' or 'sage' (default: 'sage')
- `times` - number of experiments (default: `3`)

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: ts = b.matrix_multiply_GF(100, p=19)
sage: tm = b.matrix_multiply_GF(100, p=19, system='magma')  # optional - magma
```

`sage.matrix.benchmark.`**`matrix_multiply_QQ`** (*n=100*, *bnd=2*, *system='sage'*, *times=1*)

Given an n x n matrix A over QQ with random entries whose numerators and denominators are bounded by bnd, compute A * (A+1).

INPUT:

- `n` - matrix dimension (default: `300`)
- `bnd` - numerator and denominator bound (default: `bnd`)
- `system` - either 'sage' or 'magma' (default: 'sage')
- `times` - number of experiments (default: `1`)

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: ts = b.matrix_multiply_QQ(100)
sage: tm = b.matrix_multiply_QQ(100, system='magma')  # optional - magma
```

sage.matrix.benchmark.**matrix_multiply_ZZ**(*n=300*, *min=-9*, *max=9*, *system='sage'*, *times=1*)

Matrix multiplication over ZZ Given an n x n matrix A over ZZ with random entries between min and max, inclusive, compute A * (A+1).

INPUT:

- `n` - matrix dimension (default: `300`)
- `min` - minimal value for entries of matrix (default: `-9`)
- `max` - maximal value for entries of matrix (default: `9`)
- `system` - either 'sage' or 'magma' (default: 'sage')
- `times` - number of experiments (default: `1`)

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: ts = b.matrix_multiply_ZZ(200)
sage: tm = b.matrix_multiply_ZZ(200, system='magma')  # optional - magma
```

sage.matrix.benchmark.**nullspace_GF**(*n=300*, *p=16411*, *system='sage'*)

Given a n+1 x n matrix over GF(p) with random entries, compute the nullspace.

INPUT:

- `n` - matrix dimension (default: 300)
- `p` - prime number (default: `16411`)
- `system` - either 'magma' or 'sage' (default: 'sage')

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: ts = b.nullspace_GF(300)
sage: tm = b.nullspace_GF(300, system='magma')  # optional - magma
```

sage.matrix.benchmark.**nullspace_RDF**(*n=300*, *min=0*, *max=10*, *system='sage'*)

Nullspace over RDF: Given a n+1 x n matrix over RDF with random entries between min and max, compute the nullspace.

INPUT:

- `n` - matrix dimension (default: `300`)
- `min` - minimal value for entries of matrix (default: `0`)
- `max` - maximal value for entries of matrix (default: 10')
- `system` - either 'sage' or 'magma' (default: 'sage')

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: ts = b.nullspace_RDF(100)  # long time
sage: tm = b.nullspace_RDF(100, system='magma')  # optional - magma
```

sage.matrix.benchmark.**nullspace_RR**(*n=300*, *min=0*, *max=10*, *system='sage'*)

Nullspace over RR: Given a n+1 x n matrix over RR with random entries between min and max, compute the nullspace.

INPUT:

- `n` - matrix dimension (default: `300`)
- `min` - minimal value for entries of matrix (default: `0`)
- `max` - maximal value for entries of matrix (default: `10`)
- `system` - either 'sage' or 'magma' (default: 'sage')

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: ts = b.nullspace_RR(100)
sage: tm = b.nullspace_RR(100, system='magma')  # optional - magma
```

sage.matrix.benchmark.**nullspace_ZZ**(*n=200*, *min=0*, *max=4294967296*, *system='sage'*)

Nullspace over ZZ: Given a n+1 x n matrix over ZZ with random entries between min and max, compute the nullspace.

INPUT:

- `n` - matrix dimension (default: `200`)
- `min` - minimal value for entries of matrix (default: `0`)
- `max` - maximal value for entries of matrix (default: `2**32`)
- `system` - either 'sage' or 'magma' (default: 'sage')

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: ts = b.nullspace_ZZ(200)
sage: tm = b.nullspace_ZZ(200, system='magma')  # optional - magma
```

sage.matrix.benchmark.**rank2_GF**(*n=500*, *p=16411*, *system='sage'*)

Rank over GF(p): Given a (n + 10) x n matrix over GF(p) with random entries, compute the rank.

INPUT:

- `n` - matrix dimension (default: 300)
- `p` - prime number (default: `16411`)
- `system` - either 'magma' or 'sage' (default: 'sage')

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: ts = b.rank2_GF(500)
sage: tm = b.rank2_GF(500, system='magma')  # optional - magma
```

sage.matrix.benchmark.**rank2_ZZ**(*n=400*, *min=0*, *max=18446744073709551616L*, *system='sage'*)

Rank 2 over ZZ: Given a (n + 10) x n matrix over ZZ with random entries between min and max, compute the rank.

INPUT:

- `n` - matrix dimension (default: `400`)

•`min` - minimal value for entries of matrix (default: `0`)

•`max` - maximal value for entries of matrix (default: `2**64`)

•`system` - either 'sage' or 'magma' (default: 'sage')

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: ts = b.rank2_ZZ(300)
sage: tm = b.rank2_ZZ(300, system='magma')   # optional - magma
```

`sage.matrix.benchmark.` **`rank_GF`** (*n=500*, *p=16411*, *system='sage'*)

Rank over GF(p): Given a n x (n+10) matrix over GF(p) with random entries, compute the rank.

INPUT:

•`n` - matrix dimension (default: 300)

•`p` - prime number (default: `16411`)

•`system` - either 'magma' or 'sage' (default: 'sage')

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: ts = b.rank_GF(1000)
sage: tm = b.rank_GF(1000, system='magma')   # optional - magma
```

`sage.matrix.benchmark.` **`rank_ZZ`** (*n=700*, *min=0*, *max=9*, *system='sage'*)

Rank over ZZ: Given a n x (n+10) matrix over ZZ with random entries between min and max, compute the rank.

INPUT:

•`n` - matrix dimension (default: `700`)

•`min` - minimal value for entries of matrix (default: `0`)

•`max` - maximal value for entries of matrix (default: `9`)

•`system` - either 'sage' or 'magma' (default: 'sage')

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: ts = b.rank_ZZ(300)
sage: tm = b.rank_ZZ(300, system='magma')  # optional - magma
```

`sage.matrix.benchmark.` **`report`** (*F*, *title*, *systems=['sage', 'magma']*, ***kwds*)

Run benchmarks with default arguments for each function in the list F.

INPUT:

•`F` - a list of callables used for benchmarking

•`title` - a string describing this report

•`systems` - a list of systems (supported entries are 'sage' and 'magma')

•`**kwds` - keyword arguments passed to all functions in `F`

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: print("starting"); import sys; sys.stdout.flush(); b.report([b.det_ZZ],
↪'Test', systems=['sage'])
starting...
======================================================================
          Test
======================================================================
...
======================================================================
```

sage.matrix.benchmark. **report_GF** (*p=16411*, ***kwds*)

Runs all the reports for finite field matrix operations, for prime p=16411.

INPUT:

- p - ignored
- **kwds - passed through to *report()*

Note: right now, even though p is an input, it is being ignored! If you need to check the performance for other primes, you can call individual benchmark functions.

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: print("starting"); import sys; sys.stdout.flush(); b.report_GF(systems=[
↪'sage'])
starting...
======================================================================
Dense benchmarks over GF with prime 16411
======================================================================
...
======================================================================
```

sage.matrix.benchmark. **report_ZZ** (***kwds*)

Reports all the benchmarks for integer matrices and few rational matrices.

INPUT:

- **kwds - passed through to *report()*

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: print("starting"); import sys; sys.stdout.flush(); b.report_ZZ(systems=[
↪'sage'])  # long time (15s on sage.math, 2012)
starting...
======================================================================
Dense benchmarks over ZZ
======================================================================
...
======================================================================
```

sage.matrix.benchmark. **smithform_ZZ** (*n=128*, *min=0*, *max=9*, *system='sage'*)

Smith Form over ZZ: Given a n x n matrix over ZZ with random entries between min and max, compute the Smith normal form.

INPUT:

- `n` - matrix dimension (default: `128`)
- `min` - minimal value for entries of matrix (default: `0`)
- `max` - maximal value for entries of matrix (default: `9`)
- `system` - either 'sage' or 'magma' (default: 'sage')

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: ts = b.smithform_ZZ(100)
sage: tm = b.smithform_ZZ(100, system='magma')  # optional - magma
```

`sage.matrix.benchmark.`**`vecmat_ZZ`** (*n=300*, *min=-9*, *max=9*, *system='sage'*, *times=200*)

Vector matrix multiplication over ZZ.

Given an n x n matrix A over ZZ with random entries between min and max, inclusive, and v the first row of A, compute the product v * A.

INPUT:

- `n` - matrix dimension (default: `300`)
- `min` - minimal value for entries of matrix (default: `-9`)
- `max` - maximal value for entries of matrix (default: `9`)
- `system` - either 'sage' or 'magma' (default: 'sage')
- `times` - number of runs (default: `200`)

EXAMPLES:

```
sage: import sage.matrix.benchmark as b
sage: ts = b.vecmat_ZZ(300)  # long time
sage: tm = b.vecmat_ZZ(300, system='magma')  # optional - magma
```

CHAPTER

FORTYTHREE

INDICES AND TABLES

- Index
- Module Index
- Search Page

PYTHON MODULE INDEX

m

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z