Matrices (linear algebra)

Creating Matrices

The linear algebra module is designed to be as simple as possible. First, we import and declare our first Matrix object:

```
>>> from sympy import pprint
>>> import sys
>>> sys.displayhook = pprint
>>> from sympy.matrices import *
>>> Matrix([[1,0], [0,1]])
[1  0]
[    ]
[0  1]
>>> Matrix((
...   Matrix((
...     (1, 0, 0),
...     (0, 0, 0)
...   )),
...   (0, 0, -1)
... ))
[1  0  0 ]
[        ]
[0  0  0 ]
[        ]
[0  0  -1]
```

This is the standard manner one creates a matrix, i.e. with a list of appropriately-sizes lists and/or matrices. SymPy also supports more advanced methods of matrix creation including a single list of values and dimension inputs:

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

More interestingly (and usefully), we can use a 2-variable function (or lambda) to make one. Here we create an indicator function which is 1 on the diagonal and then use it to make the identity matrix:

```
>>> def f(i,j):
...     if i == j:
...             return 1
...     else:
...             return 0
...
>>> Matrix(4, 4, f)
[1  0  0  0]
[          ]
[0  1  0  0]
[          ]
[0  0  1  0]
[          ]
[0  0  0  1]
```

Finally let’s use lambda to create a 1-line matrix with 1’s in the even permutation entries:

```
>>> Matrix(3, 4, lambda i,j: 1 - (i+j) % 2)
[1  0  1  0]
[          ]
[0  1  0  1]
[          ]
[1  0  1  0]
```

There are also a couple of special constructors for quick matrix construction - `eye` is the identity matrix, `zeros` and `ones` for matrices of all zeros and ones, respectively:

```
>>> eye(4)
[1  0  0  0]
[          ]
[0  1  0  0]
[          ]
[0  0  1  0]
[          ]
[0  0  0  1]
>>> zeros(2)
[0  0]
[    ]
[0  0]
>>> zeros((2, 5))
[0  0  0  0  0]
```

```
[              ]
[0  0  0  0  0]
>>> ones(3)
[1  1  1]
[       ]
[1  1  1]
[       ]
[1  1  1]
>>> ones((1, 3))
[1  1  1]
```

Basic Manipulation

While learning to work with matrices, let’s choose one where the entries are readily identifiable. One useful thing to know is that while matrices are 2-dimensional, the storage is not and so it is allowable - though one should be careful - to access the entries as if they were a 1-d list.

```
>>> M = Matrix(2, 3, [1, 2, 3, 4, 5, 6])
>>> M[4]
5
```

Now, the more standard entry access is a pair of indices:

```
>>> M[1,2]
6
>>> M[0,0]
1
>>> M[1,1]
5
```

Since this is Python we’re also able to slice submatrices:

```
>>> M[0:2,0:2]
[1  2]
[    ]
[4  5]
>>> M[1:2,2]
[6]
```

```
>>> M[:,2]
[3]
[ ]
[6]
```

Remember in the 2nd example above that slicing 2:2 gives an empty range and that, as in python, a 4 column list is indexed from 0 to 3. In particular, this mean a quick way to create a copy of the matrix is:

```
>>> M2 = M[:,:]
>>> M2[0,0] = 100
>>> M
[1  2  3]
[       ]
[4  5  6]
```

See? Changing M2 didn't change M. Since we can slice, we can also assign entries:

```
>>> M = Matrix(([1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]))
>>> M
[1   2   3   4 ]
[              ]
[5   6   7   8 ]
[              ]
[9   10  11  12]
[              ]
[13  14  15  16]
>>> M[2,2] = M[0,3] = 0
>>> M
[1   2   3   0 ]
[              ]
[5   6   7   8 ]
[              ]
[9   10  0   12]
[              ]
[13  14  15  16]
```

as well as assign slices:

```
>>> M = Matrix(([1,2,3,4],[5,6,7,8],[9,10,11,12],[13,14,15,16]))
>>> M[2:,2:] = Matrix(2,2,lambda i,j: 0)
>>> M
[1   2   3  4]
[            ]
[5   6   7  8]
[            ]
[9   10  0  0]
[            ]
[13  14  0  0]
```

All the standard arithmetic operations are supported:

```
>>> M = Matrix(([1,2,3],[4,5,6],[7,8,9]))
>>> M - M
[0  0  0]
[       ]
[0  0  0]
[       ]
[0  0  0]
>>> M + M
[2   4   6 ]
[          ]
[8   10  12]
[          ]
[14  16  18]
>>> M * M
[30   36   42 ]
[             ]
[66   81   96 ]
[             ]
[102  126  150]
>>> M2 = Matrix(3,1,[1,5,0])
>>> M*M2
[11]
[  ]
[29]
[  ]
[47]
>>> M**2
[30   36   42 ]
[             ]
[66   81   96 ]
[             ]
[102  126  150]
```

As well as some useful vector operations:

```
>>> M.row_del(0)
None
>>> M
[4  5  6]
[       ]
[7  8  9]
>>> M.col_del(1)
None
>>> M
[4  6]
[    ]
[7  9]
>>> v1 = Matrix([1,2,3])
>>> v2 = Matrix([4,5,6])
>>> v3 = v1.cross(v2)
>>> v1.dot(v2)
32
>>> v2.dot(v3)
0
>>> v1.dot(v3)
0
```

Recall that the row_del() and col_del() operations don’t return a value - they simply change the matrix object. We can also ‘’glue’’ together matrices of the appropriate size:

```
>>> M1 = eye(3)
>>> M2 = zeros((3, 4))
>>> M1.row_join(M2)
[1  0  0  0  0  0  0]
[                   ]
[0  1  0  0  0  0  0]
[                   ]
[0  0  1  0  0  0  0]
>>> M3 = zeros((4, 3))
>>> M1.col_join(M3)
[1  0  0]
[       ]
[0  1  0]
[       ]
[0  0  1]
```

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

Operations on entries

We are not restricted to having multiplication between two matrices:

```
>>> M = eye(3)
>>> 2*M
[2  0  0]
[       ]
[0  2  0]
[       ]
[0  0  2]
>>> 3*M
[3  0  0]
[       ]
[0  3  0]
[       ]
[0  0  3]
```

but we can also apply functions to our matrix entries using applyfunc(). Here we'll declare a function that double any input number. Then we apply it to the 3x3 identity matrix:

```
>>> f = lambda x: 2*x
>>> eye(3).applyfunc(f)
[2  0  0]
[       ]
[0  2  0]
[       ]
[0  0  2]
```

One more useful matrix-wide entry application function is the substitution function. Let's declare a matrix with symbolic entries then substitute a value. Remember we can substitute anything - even another symbol!:

```
>>> from sympy import Symbol
>>> x = Symbol('x')
>>> M = eye(3) * x
>>> M
[x  0  0]
[       ]
[0  x  0]
[       ]
[0  0  x]
>>> M.subs(x, 4)
[4  0  0]
[       ]
[0  4  0]
[       ]
[0  0  4]
>>> y = Symbol('y')
>>> M.subs(x, y)
[y  0  0]
[       ]
[0  y  0]
[       ]
[0  0  y]
```

Linear algebra

Now that we have the basics out of the way, let's see what we can do with the actual matrices. Of course the first things that come to mind are the basics like the determinant:

```
>>> M = Matrix(( [1, 2, 3], [3, 6, 2], [2, 0, 1] ))
>>> M.det()
-28
>>> M2 = eye(3)
>>> M2.det()
1
>>> M3 = Matrix(( [1, 0, 0], [1, 0, 0], [1, 0, 0] ))
>>> M3.det()
0
```

and the inverse. In SymPy the inverse is computed by Gaussian elimination by default but we can specify it be done by LU decomposition as well:

```
>>> M2.inv()
[1  0  0]
[       ]
[0  1  0]
[       ]
[0  0  1]
>>> M2.inv("LU")
[1  0  0]
[       ]
[0  1  0]
[       ]
[0  0  1]
>>> M.inv("LU")
[-3/14  1/14  1/2 ]
[                 ]
[-1/28  5/28  -1/4]
[                 ]
[ 3/7   -1/7   0  ]
>>> M * M.inv("LU")
[1  0  0]
[       ]
[0  1  0]
[       ]
[0  0  1]
```

We can perform a QR factorization which is handy for solving systems:

```
>>> A = Matrix([[1,1,1],[1,1,3],[2,3,4]])
>>> Q, R = A.QRdecomposition()
>>> Q
[  ___     ___     ___]
[\/ 6   -\/ 3   -\/ 2 ]
[-----  ------  ------]
[  6      3       2   ]
[                     ]
[  ___     ___    ___ ]
[\/ 6   -\/ 3   \/ 2  ]
[-----  ------  ----- ]
[  6      3       2   ]
[                      ]
[  ___    ___          ]
[\/ 6   \/ 3           ]
[-----  -----     0    ]
[  3      3            ]
>>> R
[         ___            ]
[  ___  4*\/ 6        ___]
[\/ 6   -------  2*\/ 6 ]
[          3             ]
[                        ]
[          ___           ]
[        \/ 3            ]
[  0     -----      0    ]
[          3             ]
[                        ]
[                    ___ ]
[  0       0       \/ 2  ]
>>> Q*R
[1  1  1]
[       ]
[1  1  3]
[       ]
[2  3  4]
```

In addition to the solvers in the solver.py file, we can solve the system Ax=b by passing the b vector to the matrix A's LUsolve function. Here we'll cheat a little choose A and x then multiply to get b. Then we can solve for x and check that it's correct:

```
>>> A = Matrix([ [2, 3, 5], [3, 6, 2], [8, 3, 6] ])
>>> x = Matrix(3,1,[3,7,5])
>>> b = A*x
>>> soln = A.LUsolve(b)
>>> soln
[3]
[ ]
[7]
[ ]
[5]
```

There's also a nice Gram-Schmidt orthogonalizer which will take a set of vectors and orthogonalize then with respect to another another. There is an optional argument which specifies whether or not the output should also be normalized, it defaults to False. Let's take some vectors and orthogonalize them - one normalized and one not:

```
>>> L = [Matrix((2,3,5)), Matrix((3,6,2)), Matrix((8,3,6))]
>>> out1 = GramSchmidt(L)
>>> out2 = GramSchmidt(L, True)
```

Let’s take a look at the vectors:

```
>>> for i in out1:
...     print i
...
[2]
[3]
[5]
[ 23/19]
[ 63/19]
[-47/19]
[ 1692/353]
[-1551/706]
[ -423/706]
>>> for i in out2:
...      print i
...
[  38**(1/2)/19]
[3*38**(1/2)/38]
[5*38**(1/2)/38]
[ 23*6707**(1/2)/6707]
[ 63*6707**(1/2)/6707]
[-47*6707**(1/2)/6707]
[ 12*706**(1/2)/353]
[-11*706**(1/2)/706]
[ -3*706**(1/2)/706]
```

We can spot-check their orthogonality with dot() and their normality with norm():

```
>>> out1[0].dot(out1[1])
0
>>> out1[0].dot(out1[2])
0
>>> out1[1].dot(out1[2])
0
```

```
>>> out2[0].norm()
1
>>> out2[1].norm()
1
>>> out2[2].norm()
1
```

So there is quite a bit that can be done with the module including eigenvalues, eigenvectors, nullspace calculation, cofactor expansion tools, and so on. From here one might want to look over the matrices.py file for all functionality.

Matrix Class Reference

class `sympy.matrices.matrices.`**`Matrix`**(**args*)

`C`

By-element conjugation.

`D`

Dirac conjugation.

`H`

Hermite conjugation.

```
>>> from sympy import Matrix, I
>>> m=Matrix(((1,2+I),(3,4)))
>>> m
[1, 2 + I]
[3,     4]
>>> m.H
[    1, 3]
[2 - I, 4]
```

`LDLdecomposition`()

Returns the LDL Decomposition (L,D) of matrix A, such that L * D * L.T == A This method eliminates the use of square root. Further this ensures that all the diagonal entries of L are 1. A must be a square, symmetric, positive-definite and non-singular matrix.

```
>>> from sympy.matrices import Matrix, eye
>>> A = Matrix(((25,15,-5),(15,18,0),(-5,0,11)))
>>> L, D = A.LDLdecomposition()
>>> L
[   1,    0, 0]
[ 3/5,    1, 0]
[-1/5, 1/3, 1]
>>> D
[25, 0, 0]
[ 0, 9, 0]
[ 0, 0, 9]
>>> L * D * L.T * A.inv() == eye(A.rows)
True
```

`LDLsolve`(*rhs*)

Solves Ax = B using LDL decomposition, for a general square and non-singular matrix.

For a non-square matrix with rows > cols, the least squares solution is returned.

`LUdecomposition`(*iszerofunc=<function _iszero at 0x102c12398>*)

Returns the decomposition LU and the row swaps p.

Example: >>> from sympy import Matrix >>> a = Matrix([[4, 3], [6, 3]]) >>> L, U, _ = a.LUdecomposition() >>> L [1, 0] [3/2, 1] >>> U [4, 3] [0, -3/2]

`LUdecompositionFF`()

Compute a fraction-free LU decomposition.

Returns 4 matrices P, L, D, U such that PA = L D**-1 U. If the elements of the matrix belong to some integral domain I, then all elements of L, D and U are guaranteed to belong to I.

Reference

- W. Zhou & D.J. Jeffrey, “Fraction-free matrix factors: new forms for LU and QR factors”. Frontiers in Computer Science in China, Vol 2, no. 1, pp. 67-80, 2008.

`LUdecomposition_Simple`(*iszerofunc=<function _iszero at 0x102c12398>*)

Returns A comprised of L,U (L's diag entries are 1) and p which is the list of the row swaps (in order).

`LUsolve`(*rhs*, *iszerofunc=<function _iszero at 0x102c12398>*)

Solve the linear system Ax = b for x. self is the coefficient matrix A and rhs is the right side b.

This is for symbolic matrices, for real or complex ones use sympy.mpmath.lu_solve or sympy.mpmath.qr_solve.

`QRdecomposition`()

Return Q,R where A = Q*R, Q is orthogonal and R is upper triangular.

Examples:: This is the example from wikipedia >>> from sympy import Matrix, eye >>> A = Matrix([[12,-51,4],[6,167,-68], [-4,24,-41]]) >>> Q, R = A.QRdecomposition() >>> Q [6/7, -69/175, -58/175] [3/7, 158/175, 6/175] [-2/7, 6/35, -33/35] >>> R [14, 21, -14] [0, 175, -70] [0, 0, 35] >>> A == Q*R True

QR factorization of an identity matrix >>> A = Matrix([[1,0,0],[0,1,0],[0,0,1]]) >>> Q, R = A.QRdecomposition() >>> Q [1, 0, 0] [0, 1, 0] [0, 0, 1] >>> R [1, 0, 0] [0, 1, 0] [0, 0, 1]

`QRsolve`(*b*)

Solve the linear system 'Ax = b'.

'self' is the matrix 'A', the method argument is the vector 'b'. The method returns the solution vector 'x'. If 'b' is a matrix, the system is solved for each column of 'b' and the return value is a matrix of the same shape as 'b'.

This method is slower (approximately by a factor of 2) but more stable for floating-point arithmetic than the LUsolve method. However, LUsolve usually uses an exact arithmetic, so you don't need to use QRsolve.

This is mainly for educational purposes and symbolic matrices, for real (or complex) matrices use sympy.mpmath.qr_solve.

`T`

Matrix transposition.

`add`(*b*)

Return self+b

adjugate(*method='berkowitz'*)

Returns the adjugate matrix.

Adjugate matrix is the transpose of the cofactor matrix.

http://en.wikipedia.org/wiki/Adjugate

See also: .cofactorMatrix(), .T

applyfunc(*f*)

```
>>> from sympy import Matrix
>>> m = Matrix(2,2,lambda i,j: i*2+j)
>>> m
[0, 1]
[2, 3]
>>> m.applyfunc(lambda i: 2*i)
[0, 2]
[4, 6]
```

berkowitz()

The Berkowitz algorithm.

Given N x N matrix with symbolic content, compute efficiently coefficients of characteristic polynomials of 'self' and all its square sub-matrices composed by removing both i-th row and column, without division in the ground domain.

This method is particularly useful for computing determinant, principal minors and characteristic polynomial, when 'self' has complicated coefficients e.g. polynomials. Semi-direct usage of this algorithm is also important in computing efficiently sub-resultant PRS.

Assuming that M is a square matrix of dimension N x N and I is N x N identity matrix, then the following following definition of characteristic polynomial is begin used:

charpoly(M) = det(t*I - M)

As a consequence, all polynomials generated by Berkowitz algorithm are monic.

```
>>> from sympy import Matrix
>>> from sympy.abc import x, y, z
```

```
>>> M = Matrix([ [x,y,z], [1,0,0], [y,z,x] ])
```

```
>>> p, q, r = M.berkowitz()
```

```
>>> print p # 1 x 1 M's sub-matrix
(1, -x)
```

```
>>> print q # 2 x 2 M's sub-matrix
(1, -x, -y)
```

```
>>> print r # 3 x 3 M's sub-matrix
(1, -2*x, x**2 - y*z - y, x*y - z**2)
```

For more information on the implemented algorithm refer to:

[1] S.J. Berkowitz, On computing the determinant in small
  parallel time using a small number of processors, ACM, Information Processing Letters 18, 1984, pp. 147-150

[2] M. Keber, Division-Free computation of sub-resultants
  using Bezout matrices, Tech. Report MPI-I-2006-1-006, Saarbrucken, 2006

berkowitz_charpoly(*x*, *simplify=<function simplify at 0x102b76848>*)

Computes characteristic polynomial minors using Berkowitz method.

berkowitz_det()

Computes determinant using Berkowitz method.

berkowitz_eigenvals(***flags*)

Computes eigenvalues of a Matrix using Berkowitz method.

berkowitz_minors()

Computes principal minors using Berkowitz method.

charpoly(*x*, *simplify=<function simplify at 0x102b76848>*)

Computes characteristic polynomial minors using Berkowitz method.

cholesky()

Returns the Cholesky Decomposition L of a Matrix A such that L * L.T = A

A must be a square, symmetric, positive-definite and non-singular matrix

```
>>> from sympy.matrices import Matrix
>>> A = Matrix(((25,15,-5),(15,18,0),(-5,0,11)))
>>> A.cholesky()
[ 5, 0, 0]
[ 3, 3, 0]
[-1, 1, 3]
>>> A.cholesky() * A.cholesky().T
[25, 15, -5]
[15, 18,  0]
[-5,  0, 11]
```

cholesky_solve(*rhs*)

Solves Ax = B using Cholesky decomposition, for a general square non-singular matrix. For a non-square matrix with rows > cols, the least squares solution is returned.

col(*j*, *f*)

Elementary column operation using functor

```
>>> from sympy import ones
>>> I = ones(3)
>>> I.col(0,lambda i,j: i*3)
>>> I
[3, 1, 1]
[3, 1, 1]
[3, 1, 1]
```

col_del(*i*)

```
>>> import sympy
>>> M = sympy.matrices.eye(3)
>>> M.col_del(1)
>>> M
[1, 0]
[0, 0]
[0, 1]
```

col_insert(*pos*, *mti*)

```
>>> from sympy import Matrix, zeros
>>> M = Matrix(3,3,lambda i,j: i+j)
>>> M
[0, 1, 2]
[1, 2, 3]
[2, 3, 4]
>>> V = zeros((3, 1))
>>> V
[0]
[0]
[0]
>>> M.col_insert(1,V)
[0, 0, 1, 2]
[1, 0, 2, 3]
[2, 0, 3, 4]
```

col_join(*bott*)

Concatenates two matrices along self's last and bott's first row

```
>>> from sympy import Matrix
>>> M = Matrix(3,3,lambda i,j: i+j)
>>> V = Matrix(1,3,lambda i,j: 3+i+j)
>>> M.col_join(V)
[0, 1, 2]
[1, 2, 3]
[2, 3, 4]
[3, 4, 5]
```

conjugate()

By-element conjugation.

det(*method='bareis'*)

Computes the matrix determinant using the method “method”.

Possible values for “method”:

bareis ... det_bareis berkowitz ... berkowitz_det

det_bareis()

Compute matrix determinant using Bareis’ fraction-free algorithm which is an extension of the well known Gaussian elimination method. This approach is best suited for dense symbolic matrices and will result in a determinant with minimal number of fractions. It means that less term rewriting is needed on resulting formulae.

TODO: Implement algorithm for sparse matrices (SFF).

diagonal_solve(*rhs*)

Solves Ax = B efficiently, where A is a diagonal Matrix, with non-zero diagonal entries.

diagonalize(*reals_only=False*)

Return diagonalized matrix D and transformation P such as

D = P^-1 * M * P

where M is current matrix.

Example:

```
>>> from sympy import Matrix
>>> m = Matrix(3,3,[1, 2, 0, 0, 3, 0, 2, -4, 2])
>>> m
[1,  2, 0]
[0,  3, 0]
[2, -4, 2]
>>> (P, D) = m.diagonalize()
>>> D
[1, 0, 0]
```

```
[0, 2, 0]
[0, 0, 3]
>>> P
[-1/2, 0, -1/2]
[   0, 0, -1/2]
[   1, 1,    1]
>>> P.inv() * m * P
[1, 0, 0]
[0, 2, 0]
[0, 0, 3]
```

See also: .is_diagonalizable(), .is_diagonal()

`eigenvals`(***flags*)

Computes eigenvalues of a Matrix using Berkowitz method.

`eigenvects`(***flags*)

Return list of triples (eigenval, multiplicity, basis).

`exp`()

Returns the exponent of a matrix

`extract`(*rowsList*, *colsList*)

Extract a submatrix by specifying a list of rows and columns

Examples:

```
>>> from sympy import Matrix
>>> m = Matrix(4, 3, lambda i, j: i*3 + j)
>>> m
[0,  1,  2]
[3,  4,  5]
[6,  7,  8]
[9, 10, 11]
>>> m.extract([0,1,3],[0,1])
[0,  1]
[3,  4]
[9, 10]
```

See also: .submatrix()

eye(*n*)

Returns the identity matrix of size n.

fill(*value*)

Fill the matrix with the scalar value.

get_diag_blocks()

Obtains the square sub-matrices on the main diagonal of a square matrix.

Useful for inverting symbolic matrices or solving systems of linear equations which may be decoupled by having a block diagonal structure.

Example:

```
>>> from sympy import Matrix, symbols
>>> from sympy.abc import x, y, z
>>> A = Matrix([[1, 3, 0, 0], [y, z*z, 0, 0], [0, 0, x, 0], [0, 0, 0, 0]])
>>> a1, a2, a3 = A.get_diag_blocks()
>>> a1
[1,    3]
[y, z**2]
>>> a2
[x]
>>> a3
[0]
>>>
```

has(**patterns*)

Test whether any subexpression matches any of the patterns.

Examples: >>> from sympy import Matrix, Float >>> from sympy.abc import x, y >>> A = Matrix(((1, x), (0.2, 3))) >>> A.has(x) True >>> A.has(y) False >>> A.has(Float) True

hash()

Compute a hash every time, because the matrix elements could change.

inv(*method='GE'*, *iszerofunc=<function _iszero at 0x102c12398>*, *try_block_diag=False*)

Calculates the matrix inverse.

According to the “method” parameter, it calls the appropriate method:

GE .... inverse_GE() LU .... inverse_LU() ADJ ... inverse_ADJ()

According to the “try_block_diag” parameter, it will try to form block diagonal matrices using the method get_diag_blocks(), invert these individually, and then reconstruct the full inverse matrix.

Note, the GE and LU methods may require the matrix to be simplified before it is inverted in order to properly detect zeros during pivoting. In difficult cases a custom zero detection function can be provided by setting the iszerosfunc argument to a function that should return True if its argument is zero.

inverse_ADJ()

Calculates the inverse using the adjugate matrix and a determinant.

inverse_GE(*iszerofunc=<function _iszero at 0x102c12398>*)

Calculates the inverse using Gaussian elimination.

inverse_LU(*iszerofunc=<function _iszero at 0x102c12398>*)

Calculates the inverse using LU decomposition.

is_diagonal()

Check if matrix is diagonal, that is matrix in which the entries outside the main diagonal are all zero.

Example:

```
>>> from sympy import Matrix, diag
>>> m = Matrix(2,2,[1, 0, 0, 2])
>>> m
[1, 0]
[0, 2]
```

```
>>> m.is_diagonal()
True
```

```
>>> m = Matrix(2,2,[1, 1, 0, 2])
>>> m
[1, 1]
[0, 2]
>>> m.is_diagonal()
False
```

```
>>> m = diag(1, 2, 3)
>>> m
[1, 0, 0]
[0, 2, 0]
[0, 0, 3]
>>> m.is_diagonal()
True
```

See also: .is_lower(), is_upper() .is_diagonalizable()

is_diagonalizable(*reals_only=False*, *clear_subproducts=True*)

Check if matrix is diagonalizable.

If reals_only==True then check that diagonalized matrix consists of the only not complex values.

Some subproducts could be used further in other methods to avoid double calculations, By default (if clear_subproducts==True) they will be deleted.

Example:

```
>>> from sympy import Matrix
>>> m = Matrix(3,3,[1, 2, 0, 0, 3, 0, 2, -4, 2])
>>> m
[1,  2, 0]
[0,  3, 0]
[2, -4, 2]
>>> m.is_diagonalizable()
True
>>> m = Matrix(2,2,[0, 1, 0, 0])
>>> m
[0, 1]
[0, 0]
>>> m.is_diagonalizable()
False
>>> m = Matrix(2,2,[0, 1, -1, 0])
>>> m
[ 0, 1]
[-1, 0]
>>> m.is_diagonalizable()
True
>>> m.is_diagonalizable(True)
False
```

is_lower()

Check if matrix is a lower triangular matrix.

Example: >>> from sympy import Matrix >>> m = Matrix(2,2,[1, 0, 0, 1]) >>> m [1, 0] [0, 1] >>> m.is_lower() True

```
>>> m = Matrix(3,3,[2, 0, 0, 1, 4 , 0, 6, 6, 5])
>>> m
[2, 0, 0]
[1, 4, 0]
[6, 6, 5]
>>> m.is_lower()
True
```

```
>>> from sympy.abc import x, y
>>> m = Matrix(2,2,[x**2 + y, y**2 + x, 0, x + y])
>>> m
[x**2 + y, x + y**2]
[       0,    x + y]
>>> m.is_lower()
False
```

is_lower_hessenberg()

Checks if the matrix is in the lower hessenberg form.

The lower hessenberg matrix has zero entries above the first superdiagonal.

Example: >>> from sympy.matrices import Matrix >>> a = Matrix([[1,2,0,0],[5,2,3,0],[3,4,3,7],[5,6,1,1]]) >>> a [1, 2, 0, 0] [5, 2, 3, 0] [3, 4, 3, 7] [5, 6, 1, 1] >>> a.is_lower_hessenberg() True

is_nilpotent()

Checks if a matrix is nilpotent.

A matrix B is nilpotent if for some integer k, B**k is a zero matrix.

Example:

```
>>> from sympy import Matrix
>>> a = Matrix([[0,0,0],[1,0,0],[1,1,0]])
>>> a.is_nilpotent()
True
```

```
>>> a = Matrix([[1,0,1],[1,0,0],[1,1,0]])
>>> a.is_nilpotent()
False
```

is_symmetric(*simplify=True*)

Check if matrix is symmetric matrix, that is square matrix and is equal to its transpose.

By default, simplifications occur before testing symmetry. They can be skipped using ‘simplify=False’; while speeding things a bit, this may however induce false negatives.

Example:

```
>>> from sympy import Matrix
>>> m = Matrix(2,2,[0, 1, 1, 2])
>>> m
[0, 1]
[1, 2]
>>> m.is_symmetric()
True
```

```
>>> m = Matrix(2,2,[0, 1, 2, 0])
>>> m
```

```
[0, 1]
[2, 0]
>>> m.is_symmetric()
False
```

```
>>> m = Matrix(2,3,[0, 0, 0, 0, 0, 0])
>>> m
[0, 0, 0]
[0, 0, 0]
>>> m.is_symmetric()
False
```

```
>>> from sympy.abc import x, y
>>> m = Matrix(3,3,[1, x**2 + 2*x + 1, y, (x + 1)**2 , 2, 0, y, 0, 3])
>>> m
[         1, x**2 + 2*x + 1, y]
[(x + 1)**2,              2, 0]
[         y,              0, 3]
>>> m.is_symmetric()
True
```

If the matrix is already simplified, you may speed-up is_symmetric() test by using 'simplify=False'.

```
>>> m.is_symmetric(simplify=False)
False
>>> m1 = m.expand()
>>> m1.is_symmetric(simplify=False)
True
```

is_upper()

Check if matrix is an upper triangular matrix.

Example: >>> from sympy import Matrix >>> m = Matrix(2,2,[1, 0, 0, 1]) >>> m [1, 0] [0, 1] >>> m.is_upper() True

```
>>> m = Matrix(3,3,[5, 1, 9, 0, 4 , 6, 0, 0, 5])
>>> m
[5, 1, 9]
[0, 4, 6]
[0, 0, 5]
>>> m.is_upper()
```

```
True
```

```
>>> m = Matrix(2,3,[4, 2, 5, 6, 1, 1])
>>> m
[4, 2, 5]
[6, 1, 1]
>>> m.is_upper()
False
```

is_upper_hessenberg()

Checks if the matrix is the upper hessenberg form.

The upper hessenberg matrix has zero entries below the first subdiagonal.

Example: >>> from sympy.matrices import Matrix >>> a = Matrix([[1,4,2,3],[3,4,1,7],[0,2,3,4],[0,0,1,3]]) >>> a [1, 4, 2, 3] [3, 4, 1, 7] [0, 2, 3, 4] [0, 0, 1, 3] >>> a.is_upper_hessenberg() True

jacobian(*X*)

Calculates the Jacobian matrix (derivative of a vectorial function).

self

A vector of expressions representing functions f_i(x_1, ..., x_n).

X

The set of x_i's in order, it can be a list or a Matrix

Both self and X can be a row or a column matrix in any order (jacobian() should always work).

Examples:

```
>>> from sympy import sin, cos, Matrix
>>> from sympy.abc import rho, phi
>>> X = Matrix([rho*cos(phi), rho*sin(phi), rho**2])
>>> Y = Matrix([rho, phi])
>>> X.jacobian(Y)
[cos(phi), -rho*sin(phi)]
[sin(phi),  rho*cos(phi)]
```

```
[   2*rho,             0]
>>> X = Matrix([rho*cos(phi), rho*sin(phi)])
>>> X.jacobian(Y)
[cos(phi), -rho*sin(phi)]
[sin(phi),  rho*cos(phi)]
```

`jordan_cells`(*calc_transformation=True*)

Return a list of Jordan cells of current matrix. This list shape Jordan matrix J.

If calc_transformation is specified as False, then transformation P such that

J = P^-1 * M * P

will not be calculated.

Note:

Calculation of transformation P is not implemented yet

Example:

```
>>> from sympy import Matrix
>>> m = Matrix(4, 4, [6, 5, -2, -3, -3, -1, 3, 3, 2, 1, -2, -3, -1, 1, 5, 5])
>>> m
[ 6,  5, -2, -3]
[-3, -1,  3,  3]
[ 2,  1, -2, -3]
[-1,  1,  5,  5]
```

```
>>> (P, Jcells) = m.jordan_cells()
>>> Jcells[0]
[2, 1]
[0, 2]
>>> Jcells[1]
[2, 1]
[0, 2]
```

See also: jordan_form()

jordan_form(*calc_transformation=True*)

Return Jordan form J of current matrix.

If calc_transformation is specified as False, then transformation P such that

J = P^-1 * M * P

will not be calculated.

Note:

Calculation of transformation P is not implemented yet

Example:

```
>>> from sympy import Matrix
>>> m = Matrix(4, 4, [6, 5, -2, -3, -3, -1, 3, 3, 2, 1, -2, -3, -1, 1, 5, 5])
>>> m
[ 6,  5, -2, -3]
[-3, -1,  3,  3]
[ 2,  1, -2, -3]
[-1,  1,  5,  5]
```

```
>>> (P, J) = m.jordan_form()
>>> J
[2, 1, 0, 0]
[0, 2, 0, 0]
[0, 0, 2, 1]
[0, 0, 0, 2]
```

See also: jordan_cells()

key2ij(*key*)

Converts key=(4,6) to 4,6 and ensures the key is correct.

lower_triangular_solve(*rhs*)

Solves Ax = B, where A is a lower triangular matrix.

multiply(*b*)

Returns self*b

multiply_elementwise(*b*)

Return the Hadamard product (elementwise product) of A and B

```
>>> import sympy
>>> A = sympy.Matrix([[0, 1, 2], [3, 4, 5]])
>>> B = sympy.Matrix([[1, 10, 100], [100, 10, 1]])
>>> print A.multiply_elementwise(B)
[  0, 10, 200]
[300, 40,   5]
```

nullspace(*simplified=False*)

Returns list of vectors (Matrix objects) that span nullspace of self

print_nonzero(*symb='X'*)

Shows location of non-zero entries for fast shape lookup

```
>>> from sympy import Matrix, matrices
>>> m = Matrix(2,3,lambda i,j: i*3+j)
>>> m
[0, 1, 2]
[3, 4, 5]
>>> m.print_nonzero()
[ XX]
[XXX]
>>> m = matrices.eye(4)
>>> m.print_nonzero("x")
[x   ]
[ x  ]
[  x ]
[   x]
```

project(*v*)

Project onto v.

reshape(*_rows*, *_cols*)

```
>>> from sympy import Matrix
>>> m = Matrix(2,3,lambda i,j: 1)
>>> m
[1, 1, 1]
[1, 1, 1]
>>> m.reshape(1,6)
[1, 1, 1, 1, 1, 1]
>>> m.reshape(3,2)
[1, 1]
[1, 1]
[1, 1]
```

row(*i*, *f*)

Elementary row operation using functor

```
>>> from sympy import ones
>>> I = ones(3)
>>> I.row(1,lambda i,j: i*3)
>>> I
[1, 1, 1]
[3, 3, 3]
[1, 1, 1]
```

row_insert(*pos*, *mti*)

```
>>> from sympy import Matrix, zeros
>>> M = Matrix(3,3,lambda i,j: i+j)
>>> M
[0, 1, 2]
[1, 2, 3]
[2, 3, 4]
>>> V = zeros((1, 3))
>>> V
[0, 0, 0]
>>> M.row_insert(1,V)
[0, 1, 2]
[0, 0, 0]
[1, 2, 3]
[2, 3, 4]
```

row_join(*rhs*)

Concatenates two matrices along self's last and rhs's first column

```
>>> from sympy import Matrix
>>> M = Matrix(3,3,lambda i,j: i+j)
>>> V = Matrix(3,1,lambda i,j: 3+i+j)
>>> M.row_join(V)
[0, 1, 2, 3]
[1, 2, 3, 4]
[2, 3, 4, 5]
```

rref(*simplified=False*, *iszerofunc=<function _iszero at 0x102c12398>*, *simplify=<function simplify at 0x102b76848>*)

Take any matrix and return reduced row-echelon form and indices of pivot vars

To simplify elements before finding nonzero pivots set simplified=True. To set a custom simplify function, use the simplify keyword argument.

simplify(*simplify=<function simplify at 0x102b76848>*, *ratio=1.7*)

Simplify the elements of a matrix in place.

If (result length)/(input length) > ratio, then input is returned unmodified. If 'ratio=oo', then simplify() is applied anyway.

See also simplify().

slice2bounds(*key*, *defmax*)

Takes slice or number and returns (min,max) for iteration Takes a default maxval to deal with the slice ':' which is (none, none)

submatrix(*keys*)

```
>>> from sympy import Matrix
>>> m = Matrix(4,4,lambda i,j: i+j)
>>> m
[0, 1, 2, 3]
[1, 2, 3, 4]
```

```
[2, 3, 4, 5]
[3, 4, 5, 6]
>>> m[0:1, 1]
[1]
>>> m[0:2, 0:1]
[0]
[1]
>>> m[2:4, 2:4]
[4, 5]
[5, 6]
```

tolist()

Return the Matrix converted in a python list.

```
>>> from sympy import Matrix
>>> m=Matrix(3, 3, range(9))
>>> m
[0, 1, 2]
[3, 4, 5]
[6, 7, 8]
>>> m.tolist()
[[0, 1, 2], [3, 4, 5], [6, 7, 8]]
```

transpose()

Matrix transposition.

```
>>> from sympy import Matrix, I
>>> m=Matrix(((1,2+I),(3,4)))
>>> m
[1, 2 + I]
[3,     4]
>>> m.transpose()
[    1, 3]
[2 + I, 4]
>>> m.T == m.transpose()
True
```

upper_triangular_solve(*rhs*)

Solves Ax = B, where A is an upper triangular matrix.

vec()

Return the Matrix converted into a one column matrix by stacking columns

```
>>> from sympy import Matrix
>>> m=Matrix([ [1,3], [2,4] ])
>>> m
[1, 3]
[2, 4]
>>> m.vec()
[1]
[2]
[3]
[4]
```

vech(*diagonal=True*, *check_symmetry=True*)

Return the unique elements of a symmetric Matrix as a one column matrix by stacking the elements in the lower triangle.

Arguments: diagonal – include the diagonal cells of self or not check_symmetry – checks symmetry of self but not completely reliably

```
>>> from sympy import Matrix
>>> m=Matrix([ [1,2], [2,3] ])
>>> m
[1, 2]
[2, 3]
>>> m.vech()
[1]
[2]
[3]
>>> m.vech(diagonal=False)
[2]
```

zeros(*dims*)

Returns a dims = (d1,d2) matrix of zeros.