Notes

Although some people trivialize the definition of positive definite matrices only for symmetric or hermitian matrices, this restriction is not correct because it does not classify all instances of positive definite matrices from the definition $x^T Ax > 0$ or $\text{re}(x^H Ax) > 0$.

For instance, `Matrix([[1, 2], [-2, 1]])` presented in the example above is an example of real positive definite matrix that is not symmetric.

However, since the following formula holds true;

$$\text{re}(x^H Ax) > 0 \iff \text{re}(x^H \frac{A + A^H}{2} x) > 0$$

We can classify all positive definite matrices that may or may not be symmetric or hermitian by transforming the matrix to $\frac{A+A^T}{2}$ or $\frac{A+A^H}{2}$ (which is guaranteed to be always real symmetric or complex hermitian) and we can defer most of the studies to symmetric or hermitian positive definite matrices.

But it is a different problem for the existance of Cholesky decomposition. Because even though a non symmetric or a non hermitian matrix can be positive definite, Cholesky or LDL decomposition does not exist because the decompositions require the matrix to be symmetric or hermitian.

References

[R587], [R588], [R589]

jordan_form(*calc_transform=True*, ***kwargs*)

Return (P, J) where J is a Jordan block matrix and P is a matrix such that $M = PJP^{-1}$

Parameters

calc_transform : bool

If `False`, then only J is returned.

chop : bool

All matrices are converted to exact types when computing eigenvalues and eigenvectors. As a result, there may be approximation errors. If `chop==True`, these errors will be truncated.

Examples

```
>>> from sympy import Matrix
>>> M = Matrix([[ 6,  5, -2, -3], [-3, -1,  3,  3], [ 2,  1, -2, -3],␣
↪[-1,  1,  5,  5]])
>>> P, J = M.jordan_form()
>>> J
Matrix([
[2, 1, 0, 0],
[0, 2, 0, 0],
[0, 0, 2, 1],
[0, 0, 0, 2]])
```

See also:

jordan_block (page 1345)

left_eigenvects(***flags*)

Returns left eigenvectors and eigenvalues.

This function returns the list of triples (eigenval, multiplicity, basis) for the left eigenvectors. Options are the same as for eigenvects(), i.e. the `**flags` arguments gets passed directly to eigenvects().

Examples

```
>>> from sympy import Matrix
>>> M = Matrix([[0, 1, 1], [1, 0, 0], [1, 1, 1]])
>>> M.eigenvects()
[(-1, 1, [Matrix([
[-1],
[ 1],
[ 0]])]), (0, 1, [Matrix([
[ 0],
[-1],
[ 1]])]), (2, 1, [Matrix([
[2/3],
[1/3],
[  1]])])]
>>> M.left_eigenvects()
[(-1, 1, [Matrix([[-2, 1, 1]])]), (0, 1, [Matrix([[-1, -1, 1]])]), (2,
1, [Matrix([[1, 1, 1]])])]
```

singular_values()

Compute the singular values of a Matrix

Examples

```
>>> from sympy import Matrix, Symbol
>>> x = Symbol('x', real=True)
>>> M = Matrix([[0, 1, 0], [0, x, 0], [-1, 0, 0]])
>>> M.singular_values()
[sqrt(x**2 + 1), 1, 0]
```

See also:

condition_number (page 1293)

MatrixCalculus Class Reference

class sympy.matrices.matrices.**MatrixCalculus**

Provides calculus-related matrix operations.

diff(**args*, ***kwargs*)

Calculate the derivative of each element in the matrix. `args` will be passed to the `integrate` function.

Examples

```
>>> from sympy import Matrix
>>> from sympy.abc import x, y
>>> M = Matrix([[x, y], [1, 0]])
>>> M.diff(x)
Matrix([
[1, 0],
[0, 0]])
```

See also:

integrate (page 1279), *limit* (page 1280)

integrate(**args*, ***kwargs*)

Integrate each element of the matrix. `args` will be passed to the `integrate` function.

Examples

```
>>> from sympy import Matrix
>>> from sympy.abc import x, y
>>> M = Matrix([[x, y], [1, 0]])
>>> M.integrate((x, ))
Matrix([
[x**2/2, x*y],
[     x,   0]])
>>> M.integrate((x, 0, 2))
Matrix([
[2, 2*y],
[2,   0]])
```

See also:

limit (page 1280), *diff* (page 1279)

jacobian(*X*)

Calculates the Jacobian matrix (derivative of a vector-valued function).

Parameters

``self`` : vector of expressions representing functions f_i(x_1, ..., x_n).

X : 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

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

See also:

hessian (page 1320), *wronskian* (page 1321)

limit(**args*)

Calculate the limit of each element in the matrix. `args` will be passed to the `limit` function.

Examples

```
>>> from sympy import Matrix
>>> from sympy.abc import x, y
>>> M = Matrix([[x, y], [1, 0]])
>>> M.limit(x, 2)
Matrix([
[2, y],
[1, 0]])
```

See also:

integrate (page 1279), *diff* (page 1279)

MatrixBase Class Reference

class sympy.matrices.matrices.**MatrixBase**

Base class for matrix objects.

property D

Return Dirac conjugate (if `self.rows == 4`).

Examples

```
>>> from sympy import Matrix, I, eye
>>> m = Matrix((0, 1 + I, 2, 3))
>>> m.D
Matrix([[0, 1 - I, -2, -3]])
>>> m = (eye(4) + I*eye(4))
>>> m[0, 3] = 2
>>> m.D
Matrix([
[1 - I,     0,      0,      0],
[    0, 1 - I,      0,      0],
[    0,     0, -1 + I,      0],
[    2,     0,      0, -1 + I]])
```

If the matrix does not have 4 rows an AttributeError will be raised because this property is only defined for matrices with 4 rows.

```
>>> Matrix(eye(2)).D
Traceback (most recent call last):
...
AttributeError: Matrix has no attribute D.
```

See also:

sympy.matrices.common.MatrixCommon.conjugate **(page 1330)**
By-element conjugation

sympy.matrices.common.MatrixCommon.H **(page 1327)**
Hermite conjugation

LDLdecomposition(*hermitian=True*)

Returns the LDL Decomposition (L, D) of matrix A, such that L * D * L.H == A if hermitian flag is True, or L * D * L.T == A if hermitian is False. This method eliminates the use of square root. Further this ensures that all the diagonal entries of L are 1. A must be a Hermitian positive-definite matrix if hermitian is True, or a symmetric matrix otherwise.

Examples

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

(continues on next page)

(continued from previous page)

```
[ 0, 0, 9]])
>>> L * D * L.T * A.inv() == eye(A.rows)
True
```

The matrix can have complex entries:

```
>>> from sympy import I
>>> A = Matrix(((9, 3*I), (-3*I, 5)))
>>> L, D = A.LDLdecomposition()
>>> L
Matrix([
[   1, 0],
[-I/3, 1]])
>>> D
Matrix([
[9, 0],
[0, 4]])
>>> L*D*L.H == A
True
```

See also:

sympy.matrices.dense.DenseMatrix.cholesky (page 1362), *sympy.matrices.matrices.MatrixBase.LUdecomposition* (page 1282), *QRdecomposition* (page 1287)

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.

Examples

```
>>> from sympy import Matrix, eye
>>> A = eye(2)*2
>>> B = Matrix([[1, 2], [3, 4]])
>>> A.LDLsolve(B) == B/2
True
```

See also:

sympy.matrices.dense.DenseMatrix.LDLdecomposition (page 1361), *sympy.matrices.dense.DenseMatrix.lower_triangular_solve* (page 1363), *sympy.matrices.dense.DenseMatrix.upper_triangular_solve* (page 1363), *gauss_jordan_solve* (page 1297), *cholesky_solve* (page 1292), *diagonal_solve* (page 1295), *LUsolve* (page 1287), *QRsolve* (page 1291), *pinv_solve* (page 1306)

LUdecomposition(*iszerofunc=<function _iszero>*, *simpfunc=None*, *rankcheck=False*)

Returns (L, U, perm) where L is a lower triangular matrix with unit diagonal, U is an upper triangular matrix, and perm is a list of row swap index pairs. If A is the original matrix, then `A = (L*U).permuteBkwd(perm)`, and the row permutation matrix P such that $PA = LU$ can be computed by `P = eye(A.rows).permuteFwd(perm)`.

See documentation for LUCombined for details about the keyword argument rankcheck, iszerofunc, and simpfunc.

Parameters

rankcheck : bool, optional

Determines if this function should detect the rank deficiency of the matrixis and should raise a `ValueError`.

iszerofunc : function, optional

A function which determines if a given expression is zero.

The function should be a callable that takes a single SymPy expression and returns a 3-valued boolean value `True`, `False`, or `None`.

It is internally used by the pivot searching algorithm. See the notes section for a more information about the pivot searching algorithm.

simpfunc : function or None, optional

A function that simplifies the input.

If this is specified as a function, this function should be a callable that takes a single SymPy expression and returns an another SymPy expression that is algebraically equivalent.

If `None`, it indicates that the pivot search algorithm should not attempt to simplify any candidate pivots.

It is internally used by the pivot searching algorithm. See the notes section for a more information about the pivot searching algorithm.

Examples

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

See also:

sympy.matrices.dense.DenseMatrix.cholesky (page 1362), *sympy.matrices.dense.DenseMatrix.LDLdecomposition* (page 1361), *QRdecomposition* (page 1287), *LUdecomposition_Simple* (page 1284), *LUdecompositionFF* (page 1283), *LUsolve* (page 1287)

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.

See also:

`sympy.matrices.matrices.MatrixBase.LUdecomposition` (page 1282), `LUdecomposition_Simple` (page 1284), `LUsolve` (page 1287)

References

[R590]

`LUdecomposition_Simple`(*iszerofunc=<function _iszero>*, *simpfunc=None*, *rankcheck=False*)

Compute the PLU decomposition of the matrix.

Parameters

rankcheck : bool, optional

Determines if this function should detect the rank deficiency of the matrixis and should raise a `ValueError`.

iszerofunc : function, optional

A function which determines if a given expression is zero.

The function should be a callable that takes a single SymPy expression and returns a 3-valued boolean value `True`, `False`, or `None`.

It is internally used by the pivot searching algorithm. See the notes section for a more information about the pivot searching algorithm.

simpfunc : function or None, optional

A function that simplifies the input.

If this is specified as a function, this function should be a callable that takes a single SymPy expression and returns an another SymPy expression that is algebraically equivalent.

If `None`, it indicates that the pivot search algorithm should not attempt to simplify any candidate pivots.

It is internally used by the pivot searching algorithm. See the notes section for a more information about the pivot searching algorithm.

Returns

(lu, row_swaps) : (Matrix, list)

If the original matrix is a m, n matrix:

lu is a m, n matrix, which contains result of the decomposition in a compresed form. See the notes section to see how the matrix is compressed.

row_swaps is a m-element list where each element is a pair of row exchange indices.

`A = (L*U).permute_backward(perm)`, and the row permutation matrix P from the formula $PA = LU$ can be computed by `P=eye(A.row).permute_forward(perm)`.

Raises

ValueError

Raised if `rankcheck=True` and the matrix is found to be rank deficient during the computation.

Notes

About the PLU decomposition:

PLU decomposition is a generalization of a LU decomposition which can be extended for rank-deficient matrices.

It can further be generalized for non-square matrices, and this is the notation that SymPy is using.

PLU decomposition is a decomposition of a m, n matrix A in the form of $PA = LU$ where

- L **is a** m, m **lower triangular matrix with unit diagonal**
  entries.
- U is a m, n upper triangular matrix.
- P is a m, m permutation matrix.

So, for a square matrix, the decomposition would look like:

$$L = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ L_{1,0} & 1 & 0 & \cdots & 0 \\ L_{2,0} & L_{2,1} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L_{n-1,0} & L_{n-1,1} & L_{n-1,2} & \cdots & 1 \end{bmatrix}$$

$$U = \begin{bmatrix} U_{0,0} & U_{0,1} & U_{0,2} & \cdots & U_{0,n-1} \\ 0 & U_{1,1} & U_{1,2} & \cdots & U_{1,n-1} \\ 0 & 0 & U_{2,2} & \cdots & U_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & U_{n-1,n-1} \end{bmatrix}$$

And for a matrix with more rows than the columns, the decomposition would look like:

$$L = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ L_{1,0} & 1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ L_{2,0} & L_{2,1} & 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ L_{n-1,0} & L_{n-1,1} & L_{n-1,2} & \cdots & 1 & 0 & \cdots & 0 \\ L_{n,0} & L_{n,1} & L_{n,2} & \cdots & L_{n,n-1} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ L_{m-1,0} & L_{m-1,1} & L_{m-1,2} & \cdots & L_{m-1,n-1} & 0 & \cdots & 1 \end{bmatrix}$$

$$U = \begin{bmatrix} U_{0,0} & U_{0,1} & U_{0,2} & \cdots & U_{0,n-1} \\ 0 & U_{1,1} & U_{1,2} & \cdots & U_{1,n-1} \\ 0 & 0 & U_{2,2} & \cdots & U_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & U_{n-1,n-1} \\ 0 & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 0 \end{bmatrix}$$

Finally, for a matrix with more columns than the rows, the decomposition would look like:

$$L = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \\ L_{1,0} & 1 & 0 & \cdots & 0 \\ L_{2,0} & L_{2,1} & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L_{m-1,0} & L_{m-1,1} & L_{m-1,2} & \cdots & 1 \end{bmatrix}$$

$$U = \begin{bmatrix} U_{0,0} & U_{0,1} & U_{0,2} & \cdots & U_{0,m-1} & \cdots & U_{0,n-1} \\ 0 & U_{1,1} & U_{1,2} & \cdots & U_{1,m-1} & \cdots & U_{1,n-1} \\ 0 & 0 & U_{2,2} & \cdots & U_{2,m-1} & \cdots & U_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \cdots & \vdots \\ 0 & 0 & 0 & \cdots & U_{m-1,m-1} & \cdots & U_{m-1,n-1} \end{bmatrix}$$

About the compressed LU storage:

The results of the decomposition are often stored in compressed forms rather than returning L and U matrices individually.

It may be less intiuitive, but it is commonly used for a lot of numeric libraries because of the efficiency.

The storage matrix is defined as following for this specific method:

- **The subdiagonal elements of L are stored in the subdiagonal** portion of LU, that is $LU_{i,j} = L_{i,j}$ whenever $i > j$.
- **The elements on the diagonal of L are all 1, and are not** explicitly stored.
- **U is stored in the upper triangular portion of LU, that is** $LU_{i,j} = U_{i,j}$ whenever $i <= j$.
- **For a case of $m > n$, the right side of the L matrix is** trivial to store.
- **For a case of $m < n$, the below side of the U matrix is** trivial to store.

So, for a square matrix, the compressed output matrix would be:

$$LU = \begin{bmatrix} U_{0,0} & U_{0,1} & U_{0,2} & \cdots & U_{0,n-1} \\ L_{1,0} & U_{1,1} & U_{1,2} & \cdots & U_{1,n-1} \\ L_{2,0} & L_{2,1} & U_{2,2} & \cdots & U_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L_{n-1,0} & L_{n-1,1} & L_{n-1,2} & \cdots & U_{n-1,n-1} \end{bmatrix}$$

For a matrix with more rows than the columns, the compressed output matrix would be:

$$LU = \begin{bmatrix} U_{0,0} & U_{0,1} & U_{0,2} & \cdots & U_{0,n-1} \\ L_{1,0} & U_{1,1} & U_{1,2} & \cdots & U_{1,n-1} \\ L_{2,0} & L_{2,1} & U_{2,2} & \cdots & U_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L_{n-1,0} & L_{n-1,1} & L_{n-1,2} & \cdots & U_{n-1,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ L_{m-1,0} & L_{m-1,1} & L_{m-1,2} & \cdots & L_{m-1,n-1} \end{bmatrix}$$

For a matrix with more columns than the rows, the compressed output matrix would be:

$$LU = \begin{bmatrix} U_{0,0} & U_{0,1} & U_{0,2} & \cdots & U_{0,m-1} & \cdots & U_{0,n-1} \\ L_{1,0} & U_{1,1} & U_{1,2} & \cdots & U_{1,m-1} & \cdots & U_{1,n-1} \\ L_{2,0} & L_{2,1} & U_{2,2} & \cdots & U_{2,m-1} & \cdots & U_{2,n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots & \cdots & \vdots \\ L_{m-1,0} & L_{m-1,1} & L_{m-1,2} & \cdots & U_{m-1,m-1} & \cdots & U_{m-1,n-1} \end{bmatrix}$$

About the pivot searching algorithm:

When a matrix contains symbolic entries, the pivot search algorithm differs from the case where every entry can be categorized as zero or nonzero. The algorithm searches column by column through the submatrix whose top left entry coincides with the pivot position. If it exists, the pivot is the first entry in the current search column that iszerofunc guarantees is nonzero. If no such candidate exists, then each candidate pivot is simplified if simpfunc is not None. The search is repeated, with the difference that a candidate may be the pivot if `iszerofunc()` cannot guarantee that it is nonzero. In the second search the pivot is the first candidate that iszerofunc can guarantee is nonzero. If no such candidate exists, then the pivot is the first candidate for which iszerofunc returns None. If no such candidate exists, then the search is repeated in the next column to the right. The pivot search algorithm differs from the one in `rref()`, which relies on `_find_reasonable_pivot()`. Future versions of `LUdecomposition_simple()` may use `_find_reasonable_pivot()`.

See also:

sympy.matrices.matrices.MatrixBase.LUdecomposition (page 1282), *LUdecompositionFF* (page 1283), *LUsolve* (page 1287)

LUsolve(*rhs*, *iszerofunc=<function _iszero>*)

Solve the linear system `Ax = rhs` for `x` where `A = M`.

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

See also:

sympy.matrices.dense.DenseMatrix.lower_triangular_solve (page 1363), *sympy.matrices.dense.DenseMatrix.upper_triangular_solve* (page 1363), *gauss_jordan_solve* (page 1297), *cholesky_solve* (page 1292), *diagonal_solve* (page 1295), *LDLsolve* (page 1282), *QRsolve* (page 1291), *pinv_solve* (page 1306), *LUdecomposition* (page 1282)

QRdecomposition()

Returns a QR decomposition.

Explanation

A QR decomposition is a decomposition in the form $A = QR$ where

- Q is a column orthogonal matrix.
- R is a upper triangular (trapezoidal) matrix.

A column orthogonal matrix satisfies $\mathbb{I} = Q^H Q$ while a full orthogonal matrix satisfies relation $\mathbb{I} = QQ^H = Q^H Q$ where I is an identity matrix with matching dimensions.

For matrices which are not square or are rank-deficient, it is sufficient to return a column orthogonal matrix because augmenting them may introduce redundant computations. And an another advantage of this is that you can easily inspect the matrix rank by counting the number of columns of Q.

If you want to augment the results to return a full orthogonal decomposition, you should use the following procedures.

- Augment the Q matrix with columns that are orthogonal to every other columns and make it square.
- Augument the R matrix with zero rows to make it have the same shape as the original matrix.

The procedure will be illustrated in the examples section.

Examples

A full rank matrix example:

```
>>> from sympy import Matrix
>>> A = Matrix([[12, -51, 4], [6, 167, -68], [-4, 24, -41]])
>>> Q, R = A.QRdecomposition()
>>> Q
Matrix([
[ 6/7, -69/175, -58/175],
[ 3/7, 158/175,   6/175],
[-2/7,    6/35,  -33/35]])
>>> R
Matrix([
[14,  21, -14],
[ 0, 175, -70],
[ 0,   0,  35]])
```

If the matrix is square and full rank, the Q matrix becomes orthogonal in both directions, and needs no augmentation.

```
>>> Q * Q.H
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> Q.H * Q
Matrix([
[1, 0, 0],
```

(continues on next page)

(continued from previous page)

```
[0, 1, 0],
[0, 0, 1]])
```

```
>>> A == Q*R
True
```

A rank deficient matrix example:

```
>>> A = Matrix([[12, -51, 0], [6, 167, 0], [-4, 24, 0]])
>>> Q, R = A.QRdecomposition()
>>> Q
Matrix([
[ 6/7, -69/175],
[ 3/7, 158/175],
[-2/7,    6/35]])
>>> R
Matrix([
[14,  21, 0],
[ 0, 175, 0]])
```

QRdecomposition might return a matrix Q that is rectangular. In this case the orthogonality condition might be satisfied as $\mathbb{I} = Q.H*Q$ but not in the reversed product $\mathbb{I} = Q*Q.H$.

```
>>> Q.H * Q
Matrix([
[1, 0],
[0, 1]])
>>> Q * Q.H
Matrix([
[27261/30625,   348/30625, -1914/6125],
[  348/30625, 30589/30625,   198/6125],
[ -1914/6125,    198/6125,   136/1225]])
```

If you want to augment the results to be a full orthogonal decomposition, you should augment Q with an another orthogonal column.

You are able to append an arbitrary standard basis that are linearly independent to every other columns and you can run the Gram-Schmidt process to make them augmented as orthogonal basis.

```
>>> Q_aug = Q.row_join(Matrix([0, 0, 1]))
>>> Q_aug = Q_aug.QRdecomposition()[0]
>>> Q_aug
Matrix([
[ 6/7, -69/175, 58/175],
[ 3/7, 158/175, -6/175],
[-2/7,    6/35,  33/35]])
>>> Q_aug.H * Q_aug
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
```

(continues on next page)

(continued from previous page)

```
>>> Q_aug * Q_aug.H
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
```

Augmenting the R matrix with zero row is straightforward.

```
>>> R_aug = R.col_join(Matrix([[0, 0, 0]]))
>>> R_aug
Matrix([
[14,  21, 0],
[ 0, 175, 0],
[ 0,   0, 0]])
>>> Q_aug * R_aug == A
True
```

A zero matrix example:

```
>>> from sympy import Matrix
>>> A = Matrix.zeros(3, 4)
>>> Q, R = A.QRdecomposition()
```

They may return matrices with zero rows and columns.

```
>>> Q
Matrix(3, 0, [])
>>> R
Matrix(0, 4, [])
>>> Q*R
Matrix([
[0, 0, 0, 0],
[0, 0, 0, 0],
[0, 0, 0, 0]])
```

As the same augmentation rule described above, Q can be augmented with columns of an identity matrix and R can be augmented with rows of a zero matrix.

```
>>> Q_aug = Q.row_join(Matrix.eye(3))
>>> R_aug = R.col_join(Matrix.zeros(3, 4))
>>> Q_aug * Q_aug.T
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> R_aug
Matrix([
[0, 0, 0, 0],
[0, 0, 0, 0],
[0, 0, 0, 0]])
>>> Q_aug * R_aug == A
True
```

See also:

sympy.matrices.dense.DenseMatrix.cholesky (page 1362), *sympy.matrices.dense.DenseMatrix.LDLdecomposition* (page 1361), *sympy.matrices.matrices.MatrixBase.LUdecomposition* (page 1282), *QRsolve* (page 1291)

QRsolve(*b*)

Solve the linear system `Ax = b`.

`M` 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 do not need to use QRsolve.

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

See also:

sympy.matrices.dense.DenseMatrix.lower_triangular_solve (page 1363), *sympy.matrices.dense.DenseMatrix.upper_triangular_solve* (page 1363), *gauss_jordan_solve* (page 1297), *cholesky_solve* (page 1292), *diagonal_solve* (page 1295), *LDLsolve* (page 1282), *LUsolve* (page 1287), *pinv_solve* (page 1306), *QRdecomposition* (page 1287)

add(*b*)

Return self + b

analytic_func(*f*, *x*)

Computes f(A) where A is a Square Matrix and f is an analytic function.

Parameters

f : Expr

Analytic Function

x : Symbol

parameter of f

Examples

```
>>> from sympy import Symbol, Matrix, S, log
```

```
>>> x = Symbol('x')
>>> m = Matrix([[S(5)/4, S(3)/4], [S(3)/4, S(5)/4]])
>>> f = log(x)
>>> m.analytic_func(f, x)
Matrix([
[     0, log(2)],
[log(2),      0]])
```

cholesky(*hermitian=True*)

Returns the Cholesky-type decomposition L of a matrix A such that L * L.H == A if hermitian flag is True, or L * L.T == A if hermitian is False.

A must be a Hermitian positive-definite matrix if hermitian is True, or a symmetric matrix if it is False.

Examples

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

The matrix can have complex entries:

```
>>> from sympy import I
>>> A = Matrix(((9, 3*I), (-3*I, 5)))
>>> A.cholesky()
Matrix([
[ 3, 0],
[-I, 2]])
>>> A.cholesky() * A.cholesky().H
Matrix([
[   9, 3*I],
[-3*I,   5]])
```

Non-hermitian Cholesky-type decomposition may be useful when the matrix is not positive-definite.

```
>>> A = Matrix([[1, 2], [2, 1]])
>>> L = A.cholesky(hermitian=False)
>>> L
Matrix([
[1,         0],
[2, sqrt(3)*I]])
>>> L*L.T == A
True
```

See also:

sympy.matrices.dense.DenseMatrix.LDLdecomposition (page 1361), *sympy.matrices.matrices.MatrixBase.LUdecomposition* (page 1282), *QRdecomposition* (page 1287)

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.

See also:

sympy.matrices.dense.DenseMatrix.lower_triangular_solve (page 1363), *sympy.matrices.dense.DenseMatrix.upper_triangular_solve* (page 1363), *gauss_jordan_solve* (page 1297), *diagonal_solve* (page 1295), *LDLsolve* (page 1282), *LUsolve* (page 1287), *QRsolve* (page 1291), *pinv_solve* (page 1306)

condition_number()

Returns the condition number of a matrix.

This is the maximum singular value divided by the minimum singular value

Examples

```
>>> from sympy import Matrix, S
>>> A = Matrix([[1, 0, 0], [0, 10, 0], [0, 0, S.One/10]])
>>> A.condition_number()
100
```

See also:

singular_values (page 1278)

connected_components()

Returns the list of connected vertices of the graph when a square matrix is viewed as a weighted graph.

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([
...     [66, 0, 0, 68, 0, 0, 0, 0, 67],
...     [0, 55, 0, 0, 0, 0, 54, 53, 0],
...     [0, 0, 0, 0, 1, 2, 0, 0, 0],
...     [86, 0, 0, 88, 0, 0, 0, 0, 87],
...     [0, 0, 10, 0, 11, 12, 0, 0, 0],
...     [0, 0, 20, 0, 21, 22, 0, 0, 0],
...     [0, 45, 0, 0, 0, 0, 44, 43, 0],
...     [0, 35, 0, 0, 0, 0, 34, 33, 0],
...     [76, 0, 0, 78, 0, 0, 0, 0, 77]])
>>> A.connected_components()
[[0, 3, 8], [1, 6, 7], [2, 4, 5]]
```

Notes

Even if any symbolic elements of the matrix can be indeterminate to be zero mathematically, this only takes the account of the structural aspect of the matrix, so they will considered to be nonzero.

connected_components_decomposition()

Decomposes a square matrix into block diagonal form only using the permutations.

Returns

P, B : PermutationMatrix, BlockDiagMatrix

P is a permutation matrix for the similarity transform as in the explanation. And B is the block diagonal matrix of the result of the permutation.

If you would like to get the diagonal blocks from the BlockDiagMatrix, see *get_diag_blocks()* (page 1385).

Explanation

The decomposition is in a form of $A = P^{-1}BP$ where P is a permutation matrix and B is a block diagonal matrix.

Examples

```
>>> from sympy import Matrix, pprint
>>> A = Matrix([
...     [66, 0, 0, 68, 0, 0, 0, 0, 67],
...     [0, 55, 0, 0, 0, 0, 54, 53, 0],
...     [0, 0, 0, 0, 1, 2, 0, 0, 0],
...     [86, 0, 0, 88, 0, 0, 0, 0, 87],
...     [0, 0, 10, 0, 11, 12, 0, 0, 0],
...     [0, 0, 20, 0, 21, 22, 0, 0, 0],
...     [0, 45, 0, 0, 0, 0, 44, 43, 0],
...     [0, 35, 0, 0, 0, 0, 34, 33, 0],
...     [76, 0, 0, 78, 0, 0, 0, 0, 77]])
```

```
>>> P, B = A.connected_components_decomposition()
>>> pprint(P)
PermutationMatrix((1 3)(2 8 5 7 4 6))
>>> pprint(B)
[[66  68  67]                            ]
[[          ]                            ]
[[86  88  87]       0             0      ]
[[          ]                            ]
[[76  78  77]                            ]
[                                        ]
[              [55  54  53]              ]
[              [          ]              ]
[     0        [45  44  43]       0      ]
[              [          ]              ]
```

(continues on next page)

(continued from previous page)

```
[               [35  34  33]              ]
[                                         ]
[                             [0   1   2 ]]
[                             [          ]]
[     0             0         [10  11  12]]
[                             [          ]]
[                             [20  21  22]]
```

```
>>> P = P.as_explicit()
>>> B = B.as_explicit()
>>> P.T*B*P == A
True
```

Notes

This problem corresponds to the finding of the connected components of a graph, when a matrix is viewed as a weighted graph.

copy()

Returns the copy of a matrix.

Examples

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

cross(*b*)

Return the cross product of `self` and `b` relaxing the condition of compatible dimensions: if each has 3 elements, a matrix of the same type and shape as `self` will be returned. If `b` has the same shape as `self` then common identities for the cross product (like $a \times b = -b \times a$) will hold.

Parameters

b : 3x1 or 1x3 Matrix

See also:

dot (page 1296), *multiply* (page 1347), *multiply_elementwise* (page 1347)

diagonal_solve(*rhs*)

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

Examples

```
>>> from sympy import Matrix, eye
>>> A = eye(2)*2
>>> B = Matrix([[1, 2], [3, 4]])
>>> A.diagonal_solve(B) == B/2
True
```

See also:

sympy.matrices.dense.DenseMatrix.lower_triangular_solve (page 1363), *sympy.matrices.dense.DenseMatrix.upper_triangular_solve* (page 1363), *gauss_jordan_solve* (page 1297), *cholesky_solve* (page 1292), *LDLsolve* (page 1282), *LUsolve* (page 1287), *QRsolve* (page 1291), *pinv_solve* (page 1306)

dot(*b*, *hermitian=None*, *conjugate_convention=None*)

Return the dot or inner product of two vectors of equal length. Here `self` must be a `Matrix` of size 1 x n or n x 1, and `b` must be either a matrix of size 1 x n, n x 1, or a list/tuple of length n. A scalar is returned.

By default, `dot` does not conjugate `self` or `b`, even if there are complex entries. Set `hermitian=True` (and optionally a `conjugate_convention`) to compute the hermitian inner product.

Possible kwargs are `hermitian` and `conjugate_convention`.

If `conjugate_convention` is `”left”`, `”math”` or `”maths”`, the conjugate of the first vector (`self`) is used. If `”right”` or `”physics”` is specified, the conjugate of the second vector `b` is used.

Examples

```
>>> from sympy import Matrix
>>> M = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> v = Matrix([1, 1, 1])
>>> M.row(0).dot(v)
6
>>> M.col(0).dot(v)
12
>>> v = [3, 2, 1]
>>> M.row(0).dot(v)
10
```

```
>>> from sympy import I
>>> q = Matrix([1*I, 1*I, 1*I])
>>> q.dot(q, hermitian=False)
-3
```

```
>>> q.dot(q, hermitian=True)
3
```

```
>>> q1 = Matrix([1, 1, 1*I])
>>> q.dot(q1, hermitian=True, conjugate_convention="maths")
```

(continues on next page)

(continued from previous page)

```
1 - 2*I
>>> q.dot(q1, hermitian=True, conjugate_convention="physics")
1 + 2*I
```

See also:

cross (page 1295), *multiply* (page 1347), *multiply_elementwise* (page 1347)

dual()

Returns the dual of a matrix, which is:

`(1/2)*levicivita(i, j, k, l)*M(k, l)` summed over indices k and l

Since the levicivita method is anti_symmetric for any pairwise exchange of indices, the dual of a symmetric matrix is the zero matrix. Strictly speaking the dual defined here assumes that the 'matrix' M is a contravariant anti_symmetric second rank tensor, so that the dual is a covariant second rank tensor.

exp()

Return the exponential of a square matrix

Examples

```
>>> from sympy import Symbol, Matrix
```

```
>>> t = Symbol('t')
>>> m = Matrix([[0, 1], [-1, 0]]) * t
>>> m.exp()
Matrix([
[    exp(I*t)/2 + exp(-I*t)/2, -I*exp(I*t)/2 + I*exp(-I*t)/2],
[I*exp(I*t)/2 - I*exp(-I*t)/2,      exp(I*t)/2 + exp(-I*t)/2]])
```

gauss_jordan_solve(*B*, *freevar=False*)

Solves `Ax = B` using Gauss Jordan elimination.

There may be zero, one, or infinite solutions. If one solution exists, it will be returned. If infinite solutions exist, it will be returned parametrically. If no solutions exist, It will throw ValueError.

Parameters

B : Matrix

The right hand side of the equation to be solved for. Must have the same number of rows as matrix A.

freevar : boolean, optional

Flag, when set to $True$ will return the indices of the free variables in the solutions (column Matrix), for a system that is undetermined (e.g. A has more columns than rows), for which infinite solutions are possible, in terms of arbitrary values of free variables. Default $False$.

Returns

x : Matrix

The matrix that will satisfy Ax = B. Will have as many rows as matrix A has columns, and as many columns as matrix B.

params : Matrix

If the system is underdetermined (e.g. A has more columns than rows), infinite solutions are possible, in terms of arbitrary parameters. These arbitrary parameters are returned as params Matrix.

free_var_index : List, optional

If the system is underdetermined (e.g. A has more columns than rows), infinite solutions are possible, in terms of arbitrary values of free variables. Then the indices of the free variables in the solutions (column Matrix) are returned by free_var_index, if the flag *freevar* is set to *True*.

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[1, 2, 1, 1], [1, 2, 2, -1], [2, 4, 0, 6]])
>>> B = Matrix([7, 12, 4])
>>> sol, params = A.gauss_jordan_solve(B)
>>> sol
Matrix([
[-2*tau0 - 3*tau1 + 2],
[                tau0],
[          2*tau1 + 5],
[                tau1]])
>>> params
Matrix([
[tau0],
[tau1]])
>>> taus_zeroes = { tau:0 for tau in params }
>>> sol_unique = sol.xreplace(taus_zeroes)
>>> sol_unique
    Matrix([
[2],
[0],
[5],
[0]])
```

```
>>> A = Matrix([[1, 2, 3], [4, 5, 6], [7, 8, 10]])
>>> B = Matrix([3, 6, 9])
>>> sol, params = A.gauss_jordan_solve(B)
>>> sol
Matrix([
[-1],
[ 2],
[ 0]])
>>> params
Matrix(0, 1, [])
```

```
>>> A = Matrix([[2, -7], [-1, 4]])
>>> B = Matrix([[-21, 3], [12, -2]])
>>> sol, params = A.gauss_jordan_solve(B)
>>> sol
Matrix([
[0, -2],
[3, -1]])
>>> params
Matrix(0, 2, [])
```

```
>>> from sympy import Matrix
>>> A = Matrix([[1, 2, 1, 1], [1, 2, 2, -1], [2, 4, 0, 6]])
>>> B = Matrix([7, 12, 4])
>>> sol, params, freevars = A.gauss_jordan_solve(B, freevar=True)
>>> sol
Matrix([
[-2*tau0 - 3*tau1 + 2],
[                tau0],
[          2*tau1 + 5],
[                tau1]])
>>> params
Matrix([
[tau0],
[tau1]])
>>> freevars
[1, 3]
```

See also:

sympy.matrices.dense.DenseMatrix.lower_triangular_solve (page 1363), *sympy.matrices.dense.DenseMatrix.upper_triangular_solve* (page 1363), *cholesky_solve* (page 1292), *diagonal_solve* (page 1295), *LDLsolve* (page 1282), *LUsolve* (page 1287), *QRsolve* (page 1291), *pinv* (page 1305)

References

[R591]

inv(*method=None, iszerofunc=<function _iszero>, try_block_diag=False*)

Return the inverse of a matrix using the method indicated. Default for dense matrices is is Gauss elimination, default for sparse matrices is LDL.

Parameters

method : ('GE', 'LU', 'ADJ', 'CH', 'LDL')

iszerofunc : function, optional

Zero-testing function to use.

try_block_diag : bool, optional

If True then will try to form block diagonal matrices using the method get_diag_blocks(), invert these individually, and then reconstruct the full inverse matrix.

Raises

ValueError

If the determinant of the matrix is zero.

Examples

```
>>> from sympy import SparseMatrix, Matrix
>>> A = SparseMatrix([
... [ 2, -1,  0],
... [-1,  2, -1],
... [ 0,  0,  2]])
>>> A.inv('CH')
Matrix([
[2/3, 1/3, 1/6],
[1/3, 2/3, 1/3],
[  0,   0, 1/2]])
>>> A.inv(method='LDL') # use of 'method=' is optional
Matrix([
[2/3, 1/3, 1/6],
[1/3, 2/3, 1/3],
[  0,   0, 1/2]])
>>> A * _
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> A = Matrix(A)
>>> A.inv('CH')
Matrix([
[2/3, 1/3, 1/6],
[1/3, 2/3, 1/3],
[  0,   0, 1/2]])
>>> A.inv('ADJ') == A.inv('GE') == A.inv('LU') == A.inv('CH') == A.
↪inv('LDL') == A.inv('QR')
True
```

Notes

According to the `method` keyword, it calls the appropriate method:

GE …. inverse_GE(); default for dense matrices LU …. inverse_LU() ADJ … inverse_ADJ() CH … inverse_CH() LDL … inverse_LDL(); default for sparse matrices QR … inverse_QR()

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 `iszerofunc` argument to a function that should return True if its argument is zero. The ADJ routine computes the determinant and uses that to detect singular matrices in addition to testing for zeros on the diagonal.

See also:

inverse_ADJ (page 1301), *inverse_GE* (page 1301), *inverse_LU* (page 1302), *inverse_CH* (page 1301), *inverse_LDL* (page 1301)

inv_mod(*m*)

Returns the inverse of the matrix $K \pmod{m}$, if it exists.

Method to find the matrix inverse of $K \pmod{m}$ implemented in this function:

- Compute $\mathrm{adj}(K) = \mathrm{cof}(K)^t$, the adjoint matrix of K.
- Compute $r = 1/\det(K) \pmod{m}$.
- $K^{-1} = r \cdot \mathrm{adj}(K) \pmod{m}$.

Examples

```
>>> from sympy import Matrix
>>> A = Matrix(2, 2, [1, 2, 3, 4])
>>> A.inv_mod(5)
Matrix([
[3, 1],
[4, 2]])
>>> A.inv_mod(3)
Matrix([
[1, 1],
[0, 1]])
```

inverse_ADJ(*iszerofunc=<function _iszero>*)

Calculates the inverse using the adjugate matrix and a determinant.

See also:

inv (page 1299), *inverse_GE* (page 1301), *inverse_LU* (page 1302), *inverse_CH* (page 1301), *inverse_LDL* (page 1301)

inverse_BLOCK(*iszerofunc=<function _iszero>*)

Calculates the inverse using BLOCKWISE inversion.

See also:

inv (page 1299), *inverse_ADJ* (page 1301), *inverse_GE* (page 1301), *inverse_CH* (page 1301), *inverse_LDL* (page 1301)

inverse_CH(*iszerofunc=<function _iszero>*)

Calculates the inverse using cholesky decomposition.

See also:

inv (page 1299), *inverse_ADJ* (page 1301), *inverse_GE* (page 1301), *inverse_LU* (page 1302), *inverse_LDL* (page 1301)

inverse_GE(*iszerofunc=<function _iszero>*)

Calculates the inverse using Gaussian elimination.

See also:

inv (page 1299), *inverse_ADJ* (page 1301), *inverse_LU* (page 1302), *inverse_CH* (page 1301), *inverse_LDL* (page 1301)

inverse_LDL(*iszerofunc=<function _iszero>*)

Calculates the inverse using LDL decomposition.

See also:

inv (page 1299), *inverse_ADJ* (page 1301), *inverse_GE* (page 1301), *inverse_LU* (page 1302), *inverse_CH* (page 1301)

inverse_LU(*iszerofunc=<function _iszero>*)

Calculates the inverse using LU decomposition.

See also:

inv (page 1299), *inverse_ADJ* (page 1301), *inverse_GE* (page 1301), *inverse_CH* (page 1301), *inverse_LDL* (page 1301)

inverse_QR(*iszerofunc=<function _iszero>*)

Calculates the inverse using QR decomposition.

See also:

inv (page 1299), *inverse_ADJ* (page 1301), *inverse_GE* (page 1301), *inverse_CH* (page 1301), *inverse_LDL* (page 1301)

classmethod irregular(*ntop, *matrices, **kwargs*)

Return a matrix filled by the given matrices which are listed in order of appearance from left to right, top to bottom as they first appear in the matrix. They must fill the matrix completely.

Examples

```
>>> from sympy import ones, Matrix
>>> Matrix.irregular(3, ones(2,1), ones(3,3)*2, ones(2,2)*3,
...   ones(1,1)*4, ones(2,2)*5, ones(1,2)*6, ones(1,2)*7)
Matrix([
  [1, 2, 2, 2, 3, 3],
  [1, 2, 2, 2, 3, 3],
  [4, 2, 2, 2, 5, 5],
  [6, 6, 7, 7, 5, 5]])
```

is_nilpotent()

Checks if a matrix is nilpotent.

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

Examples

```
>>> 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
```

key2bounds(*keys*)

Converts a key with potentially mixed types of keys (integer and slice) into a tuple of ranges and raises an error if any index is out of `self`'s range.

See also:

key2ij (page 1303)

key2ij(*key*)

Converts key into canonical form, converting integers or indexable items into valid integers for `self`'s range or returning slices unchanged.

See also:

key2bounds (page 1302)

log(*simplify=<function cancel>*)

Return the logarithm of a square matrix

Parameters

simplify : function, bool

The function to simplify the result with.

Default is `cancel`, which is effective to reduce the expression growing for taking reciprocals and inverses for symbolic matrices.

Examples

```
>>> from sympy import S, Matrix
```

Examples for positive-definite matrices:

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

```
>>> m = Matrix([[S(5)/4, S(3)/4], [S(3)/4, S(5)/4]])
>>> m.log()
Matrix([
[     0, log(2)],
[log(2),      0]])
```

Examples for non positive-definite matrices:

```
>>> m = Matrix([[S(3)/4, S(5)/4], [S(5)/4, S(3)/4]])
>>> m.log()
Matrix([
[         I*pi/2, log(2) - I*pi/2],
[log(2) - I*pi/2,          I*pi/2]])
```

```
>>> m = Matrix(
...     [[0, 0, 0, 1],
```

(continues on next page)

(continued from previous page)

```
...       [0, 0, 1, 0],
...       [0, 1, 0, 0],
...       [1, 0, 0, 0]])
>>> m.log()
Matrix([
[ I*pi/2,       0,       0, -I*pi/2],
[      0,  I*pi/2, -I*pi/2,       0],
[      0, -I*pi/2,  I*pi/2,       0],
[-I*pi/2,       0,       0,  I*pi/2]])
```

lower_triangular_solve(*rhs*)

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

See also:

upper_triangular_solve (page 1318), *gauss_jordan_solve* (page 1297), *cholesky_solve* (page 1292), *diagonal_solve* (page 1295), *LDLsolve* (page 1282), *LUsolve* (page 1287), *QRsolve* (page 1291), *pinv_solve* (page 1306)

norm(*ord=None*)

Return the Norm of a Matrix or Vector. In the simplest case this is the geometric size of the vector Other norms can be specified by the ord parameter

ord	norm for matrices	norm for vectors
None	Frobenius norm	2-norm
'fro'	Frobenius norm	• does not exist
inf	maximum row sum	max(abs(x))
-inf	–	min(abs(x))
1	maximum column sum	as below
-1	–	as below
2	2-norm (largest sing. value)	as below
-2	smallest singular value	as below
other	• does not exist	sum(abs(x)**ord)**(1./ord)

Examples

```
>>> from sympy import Matrix, Symbol, trigsimp, cos, sin, oo
>>> x = Symbol('x', real=True)
>>> v = Matrix([cos(x), sin(x)])
>>> trigsimp( v.norm() )
1
>>> v.norm(10)
(sin(x)**10 + cos(x)**10)**(1/10)
>>> A = Matrix([[1, 1], [1, 1]])
>>> A.norm(1) # maximum sum of absolute values of A is 2
```

(continues on next page)

(continued from previous page)

```
2
>>> A.norm(2) # Spectral norm (max of |Ax|/|x| under 2-vector-norm)
2
>>> A.norm(-2) # Inverse spectral norm (smallest singular value)
0
>>> A.norm() # Frobenius Norm
2
>>> A.norm(oo) # Infinity Norm
2
>>> Matrix([1, -2]).norm(oo)
2
>>> Matrix([-1, 2]).norm(-oo)
1
```

See also:

normalized (page 1305)

normalized(*iszerofunc=<function _iszero>*)

Return the normalized version of `self`.

Parameters

iszerofunc : Function, optional

A function to determine whether `self` is a zero vector. The default `_iszero` tests to see if each element is exactly zero.

Returns

Matrix

Normalized vector form of `self`. It has the same length as a unit vector. However, a zero vector will be returned for a vector with norm 0.

Raises

ShapeError

If the matrix is not in a vector form.

See also:

norm (page 1304)

pinv(*method='RD'*)

Calculate the Moore-Penrose pseudoinverse of the matrix.

The Moore-Penrose pseudoinverse exists and is unique for any matrix. If the matrix is invertible, the pseudoinverse is the same as the inverse.

Parameters

method : String, optional

Specifies the method for computing the pseudoinverse.

If `'RD'`, Rank-Decomposition will be used.

If `'ED'`, Diagonalization will be used.

Examples

Computing pseudoinverse by rank decomposition :

```
>>> from sympy import Matrix
>>> A = Matrix([[1, 2, 3], [4, 5, 6]])
>>> A.pinv()
Matrix([
[-17/18,  4/9],
[  -1/9,  1/9],
[ 13/18, -2/9]])
```

Computing pseudoinverse by diagonalization :

```
>>> B = A.pinv(method='ED')
>>> B.simplify()
>>> B
Matrix([
[-17/18,  4/9],
[  -1/9,  1/9],
[ 13/18, -2/9]])
```

See also:

inv (page 1299), *pinv_solve* (page 1306)

References

[R592]

pinv_solve(*B*, *arbitrary_matrix=None*)

Solve `Ax = B` using the Moore-Penrose pseudoinverse.

There may be zero, one, or infinite solutions. If one solution exists, it will be returned. If infinite solutions exist, one will be returned based on the value of arbitrary_matrix. If no solutions exist, the least-squares solution is returned.

Parameters

B : Matrix

The right hand side of the equation to be solved for. Must have the same number of rows as matrix A.

arbitrary_matrix : Matrix

If the system is underdetermined (e.g. A has more columns than rows), infinite solutions are possible, in terms of an arbitrary matrix. This parameter may be set to a specific matrix to use for that purpose; if so, it must be the same shape as x, with as many rows as matrix A has columns, and as many columns as matrix B. If left as None, an appropriate matrix containing dummy symbols in the form of `wn_m` will be used, with n and m being row and column position of each symbol.

Returns

x : Matrix

The matrix that will satisfy `Ax = B`. Will have as many rows as matrix A has columns, and as many columns as matrix B.

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([[1, 2, 3], [4, 5, 6]])
>>> B = Matrix([7, 8])
>>> A.pinv_solve(B)
Matrix([
[ _w0_0/6 - _w1_0/3 + _w2_0/6 - 55/18],
[-_w0_0/3 + 2*_w1_0/3 - _w2_0/3 + 1/9],
[ _w0_0/6 - _w1_0/3 + _w2_0/6 + 59/18]])
>>> A.pinv_solve(B, arbitrary_matrix=Matrix([0, 0, 0]))
Matrix([
[-55/18],
[   1/9],
[ 59/18]])
```

Notes

This may return either exact solutions or least squares solutions. To determine which, check `A * A.pinv() * B == B`. It will be True if exact solutions exist, and False if only a least-squares solution exists. Be aware that the left hand side of that equation may need to be simplified to correctly compare to the right hand side.

See also:

sympy.matrices.dense.DenseMatrix.lower_triangular_solve (page 1363), *sympy.matrices.dense.DenseMatrix.upper_triangular_solve* (page 1363), *gauss_jordan_solve* (page 1297), *cholesky_solve* (page 1292), *diagonal_solve* (page 1295), *LDLsolve* (page 1282), *LUsolve* (page 1287), *QRsolve* (page 1291), *pinv* (page 1305)

References

[R593]

print_nonzero(*symb='X'*)

Shows location of non-zero entries for fast shape lookup.

Examples

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

(continues on next page)

(continued from previous page)

```
>>> m.print_nonzero("x")
[x   ]
[ x  ]
[  x ]
[   x]
```

project(*v*)

Return the projection of `self` onto the line containing `v`.

Examples

```
>>> from sympy import Matrix, S, sqrt
>>> V = Matrix([sqrt(3)/2, S.Half])
>>> x = Matrix([[1, 0]])
>>> V.project(x)
Matrix([[sqrt(3)/2, 0]])
>>> V.project(-x)
Matrix([[sqrt(3)/2, 0]])
```

rank_decomposition(*iszerofunc=<function _iszero>, simplify=False*)

Returns a pair of matrices (C, F) with matching rank such that $A = CF$.

Parameters

iszerofunc : Function, optional

A function used for detecting whether an element can act as a pivot. `lambda x: x.is_zero` is used by default.

simplify : Bool or Function, optional

A function used to simplify elements when looking for a pivot. By default SymPy's `simplify` is used.

Returns

(C, F) : Matrices

C and F are full-rank matrices with rank as same as A, whose product gives A.

See Notes for additional mathematical details.

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([
...     [1, 3, 1, 4],
...     [2, 7, 3, 9],
...     [1, 5, 3, 1],
...     [1, 2, 0, 8]
... ])
>>> C, F = A.rank_decomposition()
>>> C
```

(continues on next page)

(continued from previous page)

```
Matrix([
[1, 3, 4],
[2, 7, 9],
[1, 5, 1],
[1, 2, 8]])
>>> F
Matrix([
[1, 0, -2, 0],
[0, 1,  1, 0],
[0, 0,  0, 1]])
>>> C * F == A
True
```

Notes

Obtaining F, an RREF of A, is equivalent to creating a product

$$E_n E_{n-1} ... E_1 A = F$$

where $E_n, E_{n-1}, \dots, E_1$ are the elimination matrices or permutation matrices equivalent to each row-reduction step.

The inverse of the same product of elimination matrices gives C:

$$C = (E_n E_{n-1} \dots E_1)^{-1}$$

It is not necessary, however, to actually compute the inverse: the columns of C are those from the original matrix with the same column indices as the indices of the pivot columns of F.

See also:

sympy.matrices.matrices.MatrixReductions.rref (page 1234)

References

[R594], [R595]

singular_value_decomposition()

Returns a Condensed Singular Value decomposition.

Explanation

A Singular Value decomposition is a decomposition in the form $A = U\Sigma V$ where

- U, V are column orthogonal matrix.
- Σ is a diagonal matrix, where the main diagonal contains singular values of matrix A.

A column orthogonal matrix satisfies $\mathbb{I} = U^H U$ while a full orthogonal matrix satisfies relation $\mathbb{I} = U U^H = U^H U$ where $\mathbb{I}$ is an identity matrix with matching dimensions.

For matrices which are not square or are rank-deficient, it is sufficient to return a column orthogonal matrix because augmenting them may introduce redundant computations. In condensed Singular Value Decomposition we only return column orthognal matrices because of this reason

If you want to augment the results to return a full orthogonal decomposition, you should use the following procedures.

- Augment the U, V matrices with columns that are orthogonal to every other columns and make it square.
- Augument the Σ matrix with zero rows to make it have the same shape as the original matrix.

The procedure will be illustrated in the examples section.

Examples

we take a full rank matrix first:

```
>>> from sympy import Matrix
>>> A = Matrix([[1, 2],[2,1]])
>>> U, S, V = A.singular_value_decomposition()
>>> U
Matrix([
[ sqrt(2)/2, sqrt(2)/2],
[-sqrt(2)/2, sqrt(2)/2]])
>>> S
Matrix([
[1, 0],
[0, 3]])
>>> V
Matrix([
[-sqrt(2)/2, sqrt(2)/2],
[ sqrt(2)/2, sqrt(2)/2]])
```

If a matrix if square and full rank both U, V are orthogonal in both directions

```
>>> U * U.H
Matrix([
[1, 0],
[0, 1]])
>>> U.H * U
Matrix([
[1, 0],
[0, 1]])
```

```
>>> V * V.H
Matrix([
[1, 0],
[0, 1]])
>>> V.H * V
```

(continues on next page)

(continued from previous page)

```
Matrix([
[1, 0],
[0, 1]])
>>> A == U * S * V.H
True
```

```
>>> C = Matrix([
...         [1, 0, 0, 0, 2],
...         [0, 0, 3, 0, 0],
...         [0, 0, 0, 0, 0],
...         [0, 2, 0, 0, 0],
...     ])
>>> U, S, V = C.singular_value_decomposition()
```

```
>>> V.H * V
Matrix([
[1, 0, 0],
[0, 1, 0],
[0, 0, 1]])
>>> V * V.H
Matrix([
[1/5, 0, 0, 0, 2/5],
[  0, 1, 0, 0,   0],
[  0, 0, 1, 0,   0],
[  0, 0, 0, 0,   0],
[2/5, 0, 0, 0, 4/5]])
```

If you want to augment the results to be a full orthogonal decomposition, you should augment V with an another orthogonal column.

You are able to append an arbitrary standard basis that are linearly independent to every other columns and you can run the Gram-Schmidt process to make them augmented as orthogonal basis.

```
>>> V_aug = V.row_join(Matrix([[0,0,0,0,1],
... [0,0,0,1,0]]).H)
>>> V_aug = V_aug.QRdecomposition()[0]
>>> V_aug
Matrix([
[0,   sqrt(5)/5, 0, -2*sqrt(5)/5, 0],
[1,           0, 0,            0, 0],
[0,           0, 1,            0, 0],
[0,           0, 0,            0, 1],
[0, 2*sqrt(5)/5, 0,    sqrt(5)/5, 0]])
>>> V_aug.H * V_aug
Matrix([
[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]])
>>> V_aug * V_aug.H
```

(continues on next page)

(continued from previous page)

```
Matrix([
[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]])
```

Similarly we augment U

```
>>> U_aug = U.row_join(Matrix([0,0,1,0]))
>>> U_aug = U_aug.QRdecomposition()[0]
>>> U_aug
Matrix([
[0, 1, 0, 0],
[0, 0, 1, 0],
[0, 0, 0, 1],
[1, 0, 0, 0]])
```

```
>>> U_aug.H * U_aug
Matrix([
[1, 0, 0, 0],
[0, 1, 0, 0],
[0, 0, 1, 0],
[0, 0, 0, 1]])
>>> U_aug * U_aug.H
Matrix([
[1, 0, 0, 0],
[0, 1, 0, 0],
[0, 0, 1, 0],
[0, 0, 0, 1]])
```

We add 2 zero columns and one row to S

```
>>> S_aug = S.col_join(Matrix([[0,0,0]]))
>>> S_aug = S_aug.row_join(Matrix([[0,0,0,0],
... [0,0,0,0]]).H)
>>> S_aug
Matrix([
[2,       0, 0, 0, 0],
[0, sqrt(5), 0, 0, 0],
[0,       0, 3, 0, 0],
[0,       0, 0, 0, 0]])
```

```
>>> U_aug * S_aug * V_aug.H == C
True
```

solve(*rhs*, *method='GJ'*)

Solves linear equation where the unique solution exists.

Parameters

rhs : Matrix

Vector representing the right hand side of the linear equation.

method : string, optional

If set to `'GJ'` or `'GE'`, the Gauss-Jordan elimination will be used, which is implemented in the routine `gauss_jordan_solve`.

If set to `'LU'`, `LUsolve` routine will be used.

If set to `'QR'`, `QRsolve` routine will be used.

If set to `'PINV'`, `pinv_solve` routine will be used.

It also supports the methods available for special linear systems

For positive definite systems:

If set to `'CH'`, `cholesky_solve` routine will be used.

If set to `'LDL'`, `LDLsolve` routine will be used.

To use a different method and to compute the solution via the inverse, use a method defined in the .inv() docstring.

Returns

solutions : Matrix

Vector representing the solution.

Raises

ValueError

If there is not a unique solution then a `ValueError` will be raised.

If `M` is not square, a `ValueError` and a different routine for solving the system will be suggested.

`solve_least_squares`(*rhs*, *method='CH'*)

Return the least-square fit to the data.

Parameters

rhs : Matrix

Vector representing the right hand side of the linear equation.

method : string or boolean, optional

If set to `'CH'`, `cholesky_solve` routine will be used.

If set to `'LDL'`, `LDLsolve` routine will be used.

If set to `'QR'`, `QRsolve` routine will be used.

If set to `'PINV'`, `pinv_solve` routine will be used.

Otherwise, the conjugate of `M` will be used to create a system of equations that is passed to `solve` along with the hint defined by `method`.

Returns

solutions : Matrix

Vector representing the solution.

Examples

```
>>> from sympy import Matrix, ones
>>> A = Matrix([1, 2, 3])
>>> B = Matrix([2, 3, 4])
>>> S = Matrix(A.row_join(B))
>>> S
Matrix([
[1, 2],
[2, 3],
[3, 4]])
```

If each line of S represent coefficients of Ax + By and x and y are [2, 3] then S*xy is:

```
>>> r = S*Matrix([2, 3]); r
Matrix([
[ 8],
[13],
[18]])
```

But let's add 1 to the middle value and then solve for the least-squares value of xy:

```
>>> xy = S.solve_least_squares(Matrix([8, 14, 18])); xy
Matrix([
[ 5/3],
[10/3]])
```

The error is given by S*xy - r:

```
>>> S*xy - r
Matrix([
[1/3],
[1/3],
[1/3]])
>>> _.norm().n(2)
0.58
```

If a different xy is used, the norm will be higher:

```
>>> xy += ones(2, 1)/10
>>> (S*xy - r).norm().n(2)
1.5
```

strongly_connected_components()

Returns the list of strongly connected vertices of the graph when a square matrix is viewed as a weighted graph.

Examples

```
>>> from sympy import Matrix
>>> A = Matrix([
...     [44, 0, 0, 0, 43, 0, 45, 0, 0],
...     [0, 66, 62, 61, 0, 68, 0, 60, 67],
...     [0, 0, 22, 21, 0, 0, 0, 20, 0],
...     [0, 0, 12, 11, 0, 0, 0, 10, 0],
...     [34, 0, 0, 0, 33, 0, 35, 0, 0],
...     [0, 86, 82, 81, 0, 88, 0, 80, 87],
...     [54, 0, 0, 0, 53, 0, 55, 0, 0],
...     [0, 0, 2, 1, 0, 0, 0, 0, 0],
...     [0, 76, 72, 71, 0, 78, 0, 70, 77]])
>>> A.strongly_connected_components()
[[0, 4, 6], [2, 3, 7], [1, 5, 8]]
```

strongly_connected_components_decomposition(*lower=True*)

Decomposes a square matrix into block triangular form only using the permutations.

Parameters

lower : bool

Makes B lower block triangular when `True`. Otherwise, makes B upper block triangular.

Returns

P, B : PermutationMatrix, BlockMatrix

P is a permutation matrix for the similarity transform as in the explanation. And B is the block triangular matrix of the result of the permutation.

Explanation

The decomposition is in a form of $A = P^{-1}BP$ where P is a permutation matrix and B is a block diagonal matrix.

Examples

```
>>> from sympy import Matrix, pprint
>>> A = Matrix([
...     [44, 0, 0, 0, 43, 0, 45, 0, 0],
...     [0, 66, 62, 61, 0, 68, 0, 60, 67],
...     [0, 0, 22, 21, 0, 0, 0, 20, 0],
...     [0, 0, 12, 11, 0, 0, 0, 10, 0],
...     [34, 0, 0, 0, 33, 0, 35, 0, 0],
...     [0, 86, 82, 81, 0, 88, 0, 80, 87],
...     [54, 0, 0, 0, 53, 0, 55, 0, 0],
...     [0, 0, 2, 1, 0, 0, 0, 0, 0],
...     [0, 76, 72, 71, 0, 78, 0, 70, 77]])
```

A lower block triangular decomposition:

```
>>> P, B = A.strongly_connected_components_decomposition()
>>> pprint(P)
PermutationMatrix((8)(1 4 3 2 6)(5 7))
>>> pprint(B)
[[44  43  45]   [0  0  0]     [0  0  0]  ]
[[          ]   [       ]     [       ]  ]
[[34  33  35]   [0  0  0]     [0  0  0]  ]
[[          ]   [       ]     [       ]  ]
[[54  53  55]   [0  0  0]     [0  0  0]  ]
[                                        ]
[ [0  0  0]    [22  21  20]   [0  0  0]  ]
[ [       ]    [          ]   [       ]  ]
[ [0  0  0]    [12  11  10]   [0  0  0]  ]
[ [       ]    [          ]   [       ]  ]
[ [0  0  0]    [2   1   0 ]   [0  0  0]  ]
[                                        ]
[ [0  0  0]    [62  61  60]  [66  68  67]]
[ [       ]    [          ]  [          ]]
[ [0  0  0]    [82  81  80]  [86  88  87]]
[ [       ]    [          ]  [          ]]
[ [0  0  0]    [72  71  70]  [76  78  77]]
```

```
>>> P = P.as_explicit()
>>> B = B.as_explicit()
>>> P.T * B * P == A
True
```

An upper block triangular decomposition:

```
>>> P, B = A.strongly_connected_components_decomposition(lower=False)
>>> pprint(P)
PermutationMatrix((0 1 5 7 4 3 2 8 6))
>>> pprint(B)
[[66  68  67]  [62  61  60]   [0  0  0]  ]
[[          ]  [          ]   [       ]  ]
[[86  88  87]  [82  81  80]   [0  0  0]  ]
[[          ]  [          ]   [       ]  ]
[[76  78  77]  [72  71  70]   [0  0  0]  ]
[                                        ]
[ [0  0  0]    [22  21  20]   [0  0  0]  ]
[ [       ]    [          ]   [       ]  ]
[ [0  0  0]    [12  11  10]   [0  0  0]  ]
[ [       ]    [          ]   [       ]  ]
[ [0  0  0]    [2   1   0 ]   [0  0  0]  ]
[                                        ]
[ [0  0  0]     [0  0  0]    [44  43  45]]
[ [       ]     [       ]    [          ]]
[ [0  0  0]     [0  0  0]    [34  33  35]]
[ [       ]     [       ]    [          ]]
[ [0  0  0]     [0  0  0]    [54  53  55]]
```

```
>>> P = P.as_explicit()
```

(continues on next page)